www.konkur.in

Christopher M. Bishop
with Hugh Bishop

Deep Learning

Foundations and Concepts

Springer

Telegram: @uni_k

www.konkur.in

Deep Learning

Telegram: @uni_k

www.konkur.in

Christopher M. Bishop • Hugh Bishop

Deep Learning

Foundations and Concepts

Springer

Telegram: @uni_k

www.konkur.in

Christopher M. Bishop
Microsoft Research
Cambridge, UK

Hugh Bishop
Wayve Technologies Ltd
London, UK

ISBN 978-3-031-45467-7 ISBN 978-3-031-45468-4 (eBook)
https://doi.org/10.1007/978-3-031-45468-4

© The Editor(s) (if applicable) and The Author(s), under exclusive license to Springer Nature Switzerland AG 2024
This work is subject to copyright. All rights are solely and exclusively licensed by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.
The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.
The publisher, the authors, and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, expressed or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Cover illustration: maksimee / Alamy Stock Photo
This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Paper in this product is recyclable.

Telegram: @uni_k

www.konkur.in

Preface

Deep learning uses multilayered neural networks trained with large data sets to solve complex information processing tasks and has emerged as the most successful paradigm in the field of machine learning. Over the last decade, deep learning has revolutionized many domains including computer vision, speech recognition, and natural language processing, and it is being used in a growing multitude of applications across healthcare, manufacturing, commerce, finance, scientific discovery, and many other sectors. Recently, massive neural networks, known as large language models and comprising of the order of a trillion learnable parameters, have been found to exhibit the first indications of general artificial intelligence and are now driving one of the biggest disruptions in the history of technology.

Goals of the book

This expanding impact has been accompanied by an explosion in the number and breadth of research publications in machine learning, and the pace of innovation continues to accelerate. For newcomers to the field, the challenge of getting to grips with the key ideas, let alone catching up to the research frontier, can seem daunting. Against this backdrop, *Deep Learning: Foundations and Concepts* aims to provide newcomers to machine learning, as well as those already experienced in the field, with a thorough understanding of both the foundational ideas that underpin deep learning as well as the key concepts of modern deep learning architectures and techniques. This material will equip the reader with a strong basis for future specialization. Due to the breadth and pace of change in the field, we have deliberately avoided trying to create a comprehensive survey of the latest research. Instead, much of the value of the book derives from a distillation of key ideas, and although the field itself can be expected to continue its rapid advance, these foundations and concepts are likely to stand the test of time. For example, large language models have been evolving very rapidly at the time of writing, yet the underlying transformer architecture and attention mechanism have remained largely unchanged for the last five years, while many core principles of machine learning have been known for decades.

Telegram: @uni_k

www.konkur.in

Responsible use of technology

Deep learning is a powerful technology with broad applicability that has the potential to create huge value for the world and address some of society's most pressing challenges. However, these same attributes mean that deep learning also has potential both for deliberate misuse and to cause unintended harms. We have chosen not to discuss ethical or societal aspects of the use of deep learning, as these topics are of such importance and complexity that they warrant a more thorough treatment than is possible in a technical textbook such as this. Such considerations should, however, be informed by a solid grounding in the underlying technology and how it works, and so we hope that this book will make a valuable contribution towards these important discussions. The reader is, nevertheless, strongly encouraged to be mindful about the broader implications of their work and to learn about the responsible use of deep learning and artificial intelligence alongside their studies of the technology itself.

Structure of the book

The book is structured into a relatively large number of smaller bite-sized chapters, each of which explores a specific topic. The book has a linear structure in the sense that each chapter depends only on material covered in earlier chapters. It is well suited to teaching a two-semester undergraduate or postgraduate course on machine learning but is equally relevant to those engaged in active research or in self-study.

A clear understanding of machine learning can be achieved only through the use of some level of mathematics. Specifically, three areas of mathematics lie at the heart of machine learning: probability theory, linear algebra, and multivariate calculus. The book provides a self-contained introduction to the required concepts in probability theory and includes an appendix that summarizes some useful results in linear algebra. It is assumed that the reader already has some familiarity with the basic concepts of multivariate calculus although there are appendices that provide introductions to the calculus of variations and to Lagrange multipliers. The focus of the book, however, is on conveying a clear understanding of ideas, and the emphasis is on techniques that have real-world practical value rather than on abstract theory. Where possible we try to present more complex concepts from multiple complementary perspectives including textual description, diagrams, and mathematical formulae. In addition, many of the key algorithms discussed in the text are summarized in separate boxes. These do not address issues of computational efficiency, but are provided as a complement to the mathematical explanations given in the text. We therefore hope that the material in this book will be accessible to readers from a variety of backgrounds.

Conceptually, this book is perhaps most naturally viewed as a successor to *Neural Networks for Pattern Recognition* (Bishop, 1995b), which provided the first comprehensive treatment of neural networks from a statistical perspective. It can also be considered as a companion volume to *Pattern Recognition and Machine Learning* (Bishop, 2006), which covered a broader range of topics in machine learning although it predated the deep learning revolution. However, to ensure that this

Telegram: @uni_k

www.konkur.in

new book is self-contained, appropriate material has been carried over from Bishop (2006) and refactored to focus on those foundational ideas that are needed for deep learning. This means that there are many interesting topics in machine learning discussed in Bishop (2006) that remain of interest today but which have been omitted from this new book. For example, Bishop (2006) discusses Bayesian methods in some depth, whereas this book is almost entirely non-Bayesian.

The book is accompanied by a web site that provides supporting material, including a free-to-use digital version of the book as well as solutions to the exercises and downloadable versions of the figures in PDF and JPEG formats:

```
https://www.bishopbook.com
```

The book can be cited using the following BibTex entry:

```
@book{Bishop:DeepLearning24,
  author = {Christopher M. Bishop and Hugh Bishop},
  title = {Deep Learning: Foundations and Concepts},
  year = {2024},
  publisher = {Springer}
}
```

If you have any feedback on the book or would like to report any errors, please send these to `feedback@bishopbook.com`

References

In the spirit of focusing on core ideas, we make no attempt to provide a comprehensive literature review, which in any case would be impossible given the scale and pace of change of the field. We do, however, provide references to some of the key research papers as well as review articles and other sources of further reading. In many cases, these also provide important implementation details that we gloss over in the text in order not to distract the reader from the central concepts being discussed.

Many books have been written on the subject of machine learning in general and on deep learning in particular. Those which are closest in level and style to this book include Bishop (2006), Goodfellow, Bengio, and Courville (2016), Murphy (2022), Murphy (2023), and Prince (2023).

Over the last decade, the nature of machine learning scholarship has changed significantly, with many papers being posted online on archival sites ahead of, or even instead of, submission to peer-reviewed conferences and journals. The most popular of these sites is *arXiv*, pronounced 'archive', and is available at

```
https://arXiv.org
```

The site allows papers to be updated, often leading to multiple versions associated with different calendar years, which can result in some ambiguity as to which version should be cited and for which year. It also provides free access to a PDF of each paper. We have therefore adopted a simple approach of referencing the paper according to the year of first upload, although we recommend reading the most recent version.

Telegram: @uni_k

www.konkur.in

Papers on arXiv are indexed using a notation `arXiv:YYMM.XXXXX` where `YY` and `MM` denote the year and month of first upload, respectively. Subsequent versions are denoted by appending a version number `N` in the form `arXiv:YYMM.XXXXXvN`.

Exercises

Each chapter concludes with a set of exercises designed to reinforce the key ideas explained in the text or to develop and generalize them in significant ways. These exercises form an important part of the text and each is graded according to difficulty ranging from (⋆), which denotes a simple exercise taking a few moments to complete, through to (⋆⋆⋆), which denotes a significantly more complex exercise. The reader is strongly encouraged to attempt the exercises since active participation with the material greatly increases the effectiveness of learning. Worked solutions to all of the exercises are available as a downloadable PDF file from the book web site.

Mathematical notation

We follow the same notation as Bishop (2006). For an overview of mathematics in the context of machine learning, see Deisenroth, Faisal, and Ong (2020).

Vectors are denoted by lower case bold roman letters such as $\mathbf{x}$, whereas matrices are denoted by uppercase bold roman letters, such as $\mathbf{M}$. All vectors are assumed to be column vectors unless otherwise stated. A superscript T denotes the transpose of a matrix or vector, so that $\mathbf{x}^{\mathrm{T}}$ will be a row vector. The notation $(w_1, \ldots, w_M)$ denotes a row vector with M elements, and the corresponding column vector is written as $\mathbf{w} = (w_1, \ldots, w_M)^{\mathrm{T}}$. The $M \times M$ identity matrix (also known as the unit matrix) is denoted $\mathbf{I}_M$, which will be abbreviated to $\mathbf{I}$ if there is no ambiguity about its dimensionality. It has elements I_{ij} that equal 1 if $i = j$ and 0 if $i \neq j$. The elements of a unit matrix are sometimes denoted by δ_{ij}. The notation $\mathbf{1}$ denotes a column vector in which all elements have the value 1. $\mathbf{a} \oplus \mathbf{b}$ denotes the concatenation of vectors $\mathbf{a}$ and $\mathbf{b}$, so that if $\mathbf{a} = (a_1, \ldots, a_N)$ and $\mathbf{b} = (b_1, \ldots, b_M)$ then $\mathbf{a} \oplus \mathbf{b} = (a_1, \ldots, a_N, b_1, \ldots, b_M)$. $|x|$ denotes the modulus (the positive part) of a scalar x, also known as the absolute value. We use $\det \mathbf{A}$ to denote the determinant of a matrix $\mathbf{A}$.

The notation $x \sim p(x)$ signifies that x is sampled from the distribution $p(x)$. Where there is ambiguity, we will use subscripts as in $p_x(\cdot)$ to denote which density is referred to. The expectation of a function $f(x, y)$ with respect to a random variable x is denoted by $\mathbb{E}_x[f(x, y)]$. In situations where there is no ambiguity as to which variable is being averaged over, this will be simplified by omitting the suffix, for instance $\mathbb{E}[x]$. If the distribution of x is conditioned on another variable z, then the corresponding conditional expectation will be written $\mathbb{E}_x[f(x)|z]$. Similarly, the variance of $f(x)$ is denoted $\mathrm{var}[f(x)]$, and for vector variables, the covariance is written $\mathrm{cov}[\mathbf{x}, \mathbf{y}]$. We will also use $\mathrm{cov}[\mathbf{x}]$ as a shorthand notation for $\mathrm{cov}[\mathbf{x}, \mathbf{x}]$.

The symbol $\forall$ means 'for all', so that $\forall m \in \mathcal{M}$ denotes all values of m within the set $\mathcal{M}$. We use $\mathbb{R}$ to denote the real numbers. On a graph, the set of neighbours of node i is denoted $\mathcal{N}(i)$, which should not be confused with the Gaussian or normal distribution $\mathcal{N}(x|\mu, \sigma^2)$. A functional is denoted $f[y]$ where $y(x)$ is some function. The concept of a functional is discussed in Appendix B. Curly braces $\{\ \}$ denote a

Telegram: @uni_k

www.konkur.in

set. The notation $g(x) = \mathcal{O}(f(x))$ denotes that $|f(x)/g(x)|$ is bounded as $x \to \infty$. For instance, if $g(x) = 3x^2 + 2$, then $g(x) = \mathcal{O}(x^2)$. The notation $\lfloor x \rfloor$ denotes the 'floor' of x, i.e., the largest integer that is less than or equal to x.

If we have N independent and identically distributed (i.i.d.) values $\mathbf{x}_1, \ldots, \mathbf{x}_N$ of a D-dimensional vector $\mathbf{x} = (x_1, \ldots, x_D)^{\mathrm{T}}$, we can combine the observations into a data matrix $\mathbf{X}$ of dimension $N \times D$ in which the nth row of $\mathbf{X}$ corresponds to the row vector $\mathbf{x}_n^{\mathrm{T}}$. Thus, the n, i element of $\mathbf{X}$ corresponds to the ith element of the nth observation $\mathbf{x}_n$ and is written x_{ni}. For one-dimensional variables, we denote such a matrix by $\mathbf{x}$, which is a column vector whose nth element is x_n. Note that $\mathbf{x}$ (which has dimensionality N) uses a different typeface to distinguish it from $\mathbf{x}$ (which has dimensionality D).

Acknowledgements

We would like to express our sincere gratitude to the many people who reviewed draft chapters and provided valuable feedback. In particular, we wish to thank Samuel Albanie, Cristian Bodnar, John Bronskill, Wessel Bruinsma, Ignas Budvytis, Chi Chen, Yaoyi Chen, Long Chen, Fergal Cotter, Sam Devlin, Aleksander Durumeric, Sebastian Ehlert, Katarina Elez, Andrew Foong, Hong Ge, Paul Gladkov, Paula Gori Giorgi, John Gossman, Tengda Han, Juyeon Heo, Katja Hofmann, Chin-Wei Huang, Yongchaio Huang, Giulio Isacchini, Matthew Johnson, Pragya Kale, Atharva Kelkar, Leon Klein, Pushmeet Kohli, Bonnie Kruft, Adrian Li, Haiguang Liu, Ziheng Lu, Giulia Luise, Stratis Markou, Sergio Valcarcel Macua, Krzysztof Maziarz, Matěj Mezera, Laurence Midgley, Usman Munir, Félix Musil, Elise van der Pol, Tao Qin, Isaac Reid, David Rosenberger, Lloyd Russell, Maximilian Schebek, Megan Stanley, Karin Strauss, Clark Templeton, Marlon Tobaben, Aldo Sayeg Pasos-Trejo, Richard Turner, Max Welling, Furu Wei, Robert Weston, Chris Williams, Yingce Xia, Shufang Xie, Iryna Zaporozhets, Claudio Zeni, Xieyuan Zhang, and many other colleagues who contributed through valuable discussions. We would also like to thank our editor Paul Drougas and many others at Springer, as well as the copy editor Jonathan Webley, for their support during the production of the book.

We would like to say a special thank-you to Markus Svensén, who provided immense help with the figures and typesetting for Bishop (2006) including the LaTeX style files, which have also been used for this new book. We are also grateful to the many scientists who allowed us to reproduce diagrams from their published work. Acknowledgements for specific figures appear in the associated figure captions.

Chris would like to express sincere gratitude to Microsoft for creating a highly stimulating research environment and for providing the opportunity to write this book. The views and opinions expressed in this book, however, are those of the authors and are therefore not necessarily the same as those of Microsoft or its affiliates. It has been a huge privilege and pleasure to collaborate with my son Hugh in preparing this book, which started as a joint project during the first Covid lockdown.

Telegram: @uni_k

www.konkur.in

Hugh would like to thank Wayve Technologies Ltd for generously allowing him to work part time so that he could collaborate in writing this book as well as for providing an inspiring and supportive environment for him to work and learn in. The views expressed in this book are not necessarily the same as those of Wayve or its affiliates. He would like to express his gratitude to his fiancée Jemima for her constant support as well as her grammatical and stylistic consultations. He would also like to thank Chris, who has been an excellent colleague and an inspiration to Hugh throughout his life.

Finally, we would both like to say a huge thank-you to our family members Jenna and Mark for so many things far too numerous to list here. It seems a very long time ago that we all gathered on the beach in Antalya to watch a total eclipse of the sun and to take a family photo for the dedication page of *Pattern Recognition and Machine Learning*!

Chris Bishop and Hugh Bishop
Cambridge, UK
October, 2023

Telegram: @uni_k

www.konkur.in

Contents

Preface v

Contents xi

1 The Deep Learning Revolution 1
- 1.1 The Impact of Deep Learning . . . 2
  - 1.1.1 Medical diagnosis . . . 2
  - 1.1.2 Protein structure . . . 3
  - 1.1.3 Image synthesis . . . 4
  - 1.1.4 Large language models . . . 5
- 1.2 A Tutorial Example . . . 6
  - 1.2.1 Synthetic data . . . 6
  - 1.2.2 Linear models . . . 8
  - 1.2.3 Error function . . . 8
  - 1.2.4 Model complexity . . . 9
  - 1.2.5 Regularization . . . 12
  - 1.2.6 Model selection . . . 14
- 1.3 A Brief History of Machine Learning . . . 16
  - 1.3.1 Single-layer networks . . . 17
  - 1.3.2 Backpropagation . . . 18
  - 1.3.3 Deep networks . . . 20

2 Probabilities 23
- 2.1 The Rules of Probability . . . 25
  - 2.1.1 A medical screening example . . . 25
  - 2.1.2 The sum and product rules . . . 26
  - 2.1.3 Bayes' theorem . . . 28
  - 2.1.4 Medical screening revisited . . . 30
  - 2.1.5 Prior and posterior probabilities . . . 31

Telegram: @uni_k

www.konkur.in

2.1.6 Independent variables . . . 31
2.2 Probability Densities . . . 32
2.2.1 Example distributions . . . 33
2.2.2 Expectations and covariances . . . 34
2.3 The Gaussian Distribution . . . 36
2.3.1 Mean and variance . . . 37
2.3.2 Likelihood function . . . 37
2.3.3 Bias of maximum likelihood . . . 39
2.3.4 Linear regression . . . 40
2.4 Transformation of Densities . . . 42
2.4.1 Multivariate distributions . . . 44
2.5 Information Theory . . . 46
2.5.1 Entropy . . . 46
2.5.2 Physics perspective . . . 47
2.5.3 Differential entropy . . . 49
2.5.4 Maximum entropy . . . 50
2.5.5 Kullback–Leibler divergence . . . 51
2.5.6 Conditional entropy . . . 53
2.5.7 Mutual information . . . 54
2.6 Bayesian Probabilities . . . 54
2.6.1 Model parameters . . . 55
2.6.2 Regularization . . . 56
2.6.3 Bayesian machine learning . . . 57
Exercises . . . 58

3 Standard Distributions 65
3.1 Discrete Variables . . . 66
3.1.1 Bernoulli distribution . . . 66
3.1.2 Binomial distribution . . . 67
3.1.3 Multinomial distribution . . . 68
3.2 The Multivariate Gaussian . . . 70
3.2.1 Geometry of the Gaussian . . . 71
3.2.2 Moments . . . 74
3.2.3 Limitations . . . 75
3.2.4 Conditional distribution . . . 76
3.2.5 Marginal distribution . . . 79
3.2.6 Bayes' theorem . . . 81
3.2.7 Maximum likelihood . . . 84
3.2.8 Sequential estimation . . . 85
3.2.9 Mixtures of Gaussians . . . 86
3.3 Periodic Variables . . . 89
3.3.1 Von Mises distribution . . . 89
3.4 The Exponential Family . . . 94
3.4.1 Sufficient statistics . . . 97
3.5 Nonparametric Methods . . . 98

Telegram: @uni_k

www.konkur.in

3.5.1 Histograms . . . 98
3.5.2 Kernel densities . . . 100
3.5.3 Nearest-neighbours . . . 103
Exercises . . . 105

4 Single-layer Networks: Regression 111
4.1 Linear Regression . . . 112
4.1.1 Basis functions . . . 112
4.1.2 Likelihood function . . . 114
4.1.3 Maximum likelihood . . . 115
4.1.4 Geometry of least squares . . . 117
4.1.5 Sequential learning . . . 117
4.1.6 Regularized least squares . . . 118
4.1.7 Multiple outputs . . . 119
4.2 Decision theory . . . 120
4.3 The Bias–Variance Trade-off . . . 123
Exercises . . . 128

5 Single-layer Networks: Classification 131
5.1 Discriminant Functions . . . 132
5.1.1 Two classes . . . 132
5.1.2 Multiple classes . . . 134
5.1.3 1-of-K coding . . . 135
5.1.4 Least squares for classification . . . 136
5.2 Decision Theory . . . 138
5.2.1 Misclassification rate . . . 139
5.2.2 Expected loss . . . 140
5.2.3 The reject option . . . 142
5.2.4 Inference and decision . . . 143
5.2.5 Classifier accuracy . . . 147
5.2.6 ROC curve . . . 148
5.3 Generative Classifiers . . . 150
5.3.1 Continuous inputs . . . 152
5.3.2 Maximum likelihood solution . . . 153
5.3.3 Discrete features . . . 156
5.3.4 Exponential family . . . 156
5.4 Discriminative Classifiers . . . 157
5.4.1 Activation functions . . . 158
5.4.2 Fixed basis functions . . . 158
5.4.3 Logistic regression . . . 159
5.4.4 Multi-class logistic regression . . . 161
5.4.5 Probit regression . . . 163
5.4.6 Canonical link functions . . . 164
Exercises . . . 166

Telegram: @uni_k

www.konkur.in

6 Deep Neural Networks **171**
6.1 Limitations of Fixed Basis Functions 172
6.1.1 The curse of dimensionality 172
6.1.2 High-dimensional spaces 175
6.1.3 Data manifolds 176
6.1.4 Data-dependent basis functions 178
6.2 Multilayer Networks 180
6.2.1 Parameter matrices 181
6.2.2 Universal approximation 181
6.2.3 Hidden unit activation functions 182
6.2.4 Weight-space symmetries 185
6.3 Deep Networks 186
6.3.1 Hierarchical representations 187
6.3.2 Distributed representations 187
6.3.3 Representation learning 188
6.3.4 Transfer learning 189
6.3.5 Contrastive learning 191
6.3.6 General network architectures 193
6.3.7 Tensors 194
6.4 Error Functions 194
6.4.1 Regression 194
6.4.2 Binary classification 196
6.4.3 multiclass classification 197
6.5 Mixture Density Networks 198
6.5.1 Robot kinematics example 198
6.5.2 Conditional mixture distributions 199
6.5.3 Gradient optimization 201
6.5.4 Predictive distribution 202
Exercises 204

7 Gradient Descent **209**
7.1 Error Surfaces 210
7.1.1 Local quadratic approximation 211
7.2 Gradient Descent Optimization 213
7.2.1 Use of gradient information 214
7.2.2 Batch gradient descent 214
7.2.3 Stochastic gradient descent 214
7.2.4 Mini-batches 216
7.2.5 Parameter initialization 216
7.3 Convergence 218
7.3.1 Momentum 220
7.3.2 Learning rate schedule 222
7.3.3 RMSProp and Adam 223
7.4 Normalization 224
7.4.1 Data normalization 226

Telegram: @uni_k

www.konkur.in

7.4.2 Batch normalization . . . 227
7.4.3 Layer normalization . . . 229
Exercises . . . 230

8 Backpropagation 233
8.1 Evaluation of Gradients . . . 234
8.1.1 Single-layer networks . . . 234
8.1.2 General feed-forward networks . . . 235
8.1.3 A simple example . . . 238
8.1.4 Numerical differentiation . . . 239
8.1.5 The Jacobian matrix . . . 240
8.1.6 The Hessian matrix . . . 242
8.2 Automatic Differentiation . . . 244
8.2.1 Forward-mode automatic differentiation . . . 246
8.2.2 Reverse-mode automatic differentiation . . . 249
Exercises . . . 250

9 Regularization 253
9.1 Inductive Bias . . . 254
9.1.1 Inverse problems . . . 254
9.1.2 No free lunch theorem . . . 255
9.1.3 Symmetry and invariance . . . 256
9.1.4 Equivariance . . . 259
9.2 Weight Decay . . . 260
9.2.1 Consistent regularizers . . . 262
9.2.2 Generalized weight decay . . . 264
9.3 Learning Curves . . . 266
9.3.1 Early stopping . . . 266
9.3.2 Double descent . . . 268
9.4 Parameter Sharing . . . 270
9.4.1 Soft weight sharing . . . 271
9.5 Residual Connections . . . 274
9.6 Model Averaging . . . 277
9.6.1 Dropout . . . 279
Exercises . . . 281

10 Convolutional Networks 287
10.1 Computer Vision . . . 288
10.1.1 Image data . . . 289
10.2 Convolutional Filters . . . 290
10.2.1 Feature detectors . . . 290
10.2.2 Translation equivariance . . . 291
10.2.3 Padding . . . 294
10.2.4 Strided convolutions . . . 294
10.2.5 Multi-dimensional convolutions . . . 295
10.2.6 Pooling . . . 296

Telegram: @uni_k

www.konkur.in

10.2.7 Multilayer convolutions . . . 298
10.2.8 Example network architectures . . . 299
10.3 Visualizing Trained CNNs . . . 302
10.3.1 Visual cortex . . . 302
10.3.2 Visualizing trained filters . . . 303
10.3.3 Saliency maps . . . 305
10.3.4 Adversarial attacks . . . 306
10.3.5 Synthetic images . . . 308
10.4 Object Detection . . . 308
10.4.1 Bounding boxes . . . 309
10.4.2 Intersection-over-union . . . 310
10.4.3 Sliding windows . . . 311
10.4.4 Detection across scales . . . 313
10.4.5 Non-max suppression . . . 314
10.4.6 Fast region CNNs . . . 314
10.5 Image Segmentation . . . 315
10.5.1 Convolutional segmentation . . . 315
10.5.2 Up-sampling . . . 316
10.5.3 Fully convolutional networks . . . 318
10.5.4 The U-net architecture . . . 319
10.6 Style Transfer . . . 320
Exercises . . . 322

11 Structured Distributions **325**
11.1 Graphical Models . . . 326
11.1.1 Directed graphs . . . 326
11.1.2 Factorization . . . 327
11.1.3 Discrete variables . . . 329
11.1.4 Gaussian variables . . . 332
11.1.5 Binary classifier . . . 334
11.1.6 Parameters and observations . . . 334
11.1.7 Bayes' theorem . . . 336
11.2 Conditional Independence . . . 337
11.2.1 Three example graphs . . . 338
11.2.2 Explaining away . . . 341
11.2.3 D-separation . . . 343
11.2.4 Naive Bayes . . . 344
11.2.5 Generative models . . . 346
11.2.6 Markov blanket . . . 347
11.2.7 Graphs as filters . . . 348
11.3 Sequence Models . . . 349
11.3.1 Hidden variables . . . 352
Exercises . . . 353

Telegram: @uni_k

www.konkur.in

12 Transformers **357**
12.1 Attention . . . 358
12.1.1 Transformer processing . . . 360
12.1.2 Attention coefficients . . . 361
12.1.3 Self-attention . . . 362
12.1.4 Network parameters . . . 363
12.1.5 Scaled self-attention . . . 366
12.1.6 Multi-head attention . . . 366
12.1.7 Transformer layers . . . 368
12.1.8 Computational complexity . . . 370
12.1.9 Positional encoding . . . 371
12.2 Natural Language . . . 374
12.2.1 Word embedding . . . 375
12.2.2 Tokenization . . . 377
12.2.3 Bag of words . . . 378
12.2.4 Autoregressive models . . . 379
12.2.5 Recurrent neural networks . . . 380
12.2.6 Backpropagation through time . . . 381
12.3 Transformer Language Models . . . 382
12.3.1 Decoder transformers . . . 383
12.3.2 Sampling strategies . . . 386
12.3.3 Encoder transformers . . . 388
12.3.4 Sequence-to-sequence transformers . . . 390
12.3.5 Large language models . . . 390
12.4 Multimodal Transformers . . . 394
12.4.1 Vision transformers . . . 395
12.4.2 Generative image transformers . . . 396
12.4.3 Audio data . . . 399
12.4.4 Text-to-speech . . . 400
12.4.5 Vision and language transformers . . . 402
Exercises . . . 403

13 Graph Neural Networks **407**
13.1 Machine Learning on Graphs . . . 409
13.1.1 Graph properties . . . 410
13.1.2 Adjacency matrix . . . 410
13.1.3 Permutation equivariance . . . 411
13.2 Neural Message-Passing . . . 412
13.2.1 Convolutional filters . . . 413
13.2.2 Graph convolutional networks . . . 414
13.2.3 Aggregation operators . . . 416
13.2.4 Update operators . . . 418
13.2.5 Node classification . . . 419
13.2.6 Edge classification . . . 420
13.2.7 Graph classification . . . 420

Telegram: @uni_k

www.konkur.in

13.3 General Graph Networks 420
13.3.1 Graph attention networks 421
13.3.2 Edge embeddings 421
13.3.3 Graph embeddings 422
13.3.4 Over-smoothing 422
13.3.5 Regularization 423
13.3.6 Geometric deep learning 424
Exercises 425

14 Sampling **429**
14.1 Basic Sampling Algorithms 430
14.1.1 Expectations 430
14.1.2 Standard distributions 431
14.1.3 Rejection sampling 433
14.1.4 Adaptive rejection sampling 435
14.1.5 Importance sampling 437
14.1.6 Sampling-importance-resampling 439
14.2 Markov Chain Monte Carlo 440
14.2.1 The Metropolis algorithm 441
14.2.2 Markov chains 442
14.2.3 The Metropolis–Hastings algorithm 445
14.2.4 Gibbs sampling 446
14.2.5 Ancestral sampling 450
14.3 Langevin Sampling 451
14.3.1 Energy-based models 452
14.3.2 Maximizing the likelihood 453
14.3.3 Langevin dynamics 454
Exercises 456

15 Discrete Latent Variables **459**
15.1 K-means Clustering 460
15.1.1 Image segmentation 464
15.2 Mixtures of Gaussians 466
15.2.1 Likelihood function 468
15.2.2 Maximum likelihood 470
15.3 Expectation–Maximization Algorithm 474
15.3.1 Gaussian mixtures 478
15.3.2 Relation to K-means 480
15.3.3 Mixtures of Bernoulli distributions 481
15.4 Evidence Lower Bound 485
15.4.1 EM revisited 486
15.4.2 Independent and identically distributed data 488
15.4.3 Parameter priors 489
15.4.4 Generalized EM 489
15.4.5 Sequential EM 490
Exercises 490

Telegram: @uni_k

www.konkur.in

16 Continuous Latent Variables **495**
16.1 Principal Component Analysis 497
16.1.1 Maximum variance formulation 497
16.1.2 Minimum-error formulation 499
16.1.3 Data compression 501
16.1.4 Data whitening 502
16.1.5 High-dimensional data 504
16.2 Probabilistic Latent Variables 506
16.2.1 Generative model 506
16.2.2 Likelihood function 507
16.2.3 Maximum likelihood 509
16.2.4 Factor analysis 513
16.2.5 Independent component analysis 514
16.2.6 Kalman filters 515
16.3 Evidence Lower Bound 516
16.3.1 Expectation maximization 518
16.3.2 EM for PCA 519
16.3.3 EM for factor analysis 520
16.4 Nonlinear Latent Variable Models 522
16.4.1 Nonlinear manifolds 522
16.4.2 Likelihood function 524
16.4.3 Discrete data 526
16.4.4 Four approaches to generative modelling 527
Exercises 527

17 Generative Adversarial Networks **533**
17.1 Adversarial Training 534
17.1.1 Loss function 535
17.1.2 GAN training in practice 536
17.2 Image GANs 539
17.2.1 CycleGAN 539
Exercises 544

18 Normalizing Flows **547**
18.1 Coupling Flows 549
18.2 Autoregressive Flows 552
18.3 Continuous Flows 554
18.3.1 Neural differential equations 554
18.3.2 Neural ODE backpropagation 555
18.3.3 Neural ODE flows 557
Exercises 559

Telegram: @uni_k

www.konkur.in

19 Autoencoders **563**
19.1 Deterministic Autoencoders 564
19.1.1 Linear autoencoders 564
19.1.2 Deep autoencoders 565
19.1.3 Sparse autoencoders 566
19.1.4 Denoising autoencoders 567
19.1.5 Masked autoencoders 567
19.2 Variational Autoencoders 569
19.2.1 Amortized inference 572
19.2.2 The reparameterization trick 574
Exercises 578

20 Diffusion Models **581**
20.1 Forward Encoder 582
20.1.1 Diffusion kernel 583
20.1.2 Conditional distribution 584
20.2 Reverse Decoder 585
20.2.1 Training the decoder 587
20.2.2 Evidence lower bound 588
20.2.3 Rewriting the ELBO 589
20.2.4 Predicting the noise 591
20.2.5 Generating new samples 592
20.3 Score Matching 594
20.3.1 Score loss function 595
20.3.2 Modified score loss 596
20.3.3 Noise variance 597
20.3.4 Stochastic differential equations 598
20.4 Guided Diffusion 599
20.4.1 Classifier guidance 600
20.4.2 Classifier-free guidance 600
Exercises 603

Appendix A Linear Algebra **609**
A.1 Matrix Identities 609
A.2 Traces and Determinants 610
A.3 Matrix Derivatives 611
A.4 Eigenvectors 612

Appendix B Calculus of Variations **617**

Appendix C Lagrange Multipliers **621**

Bibliography **625**

Index **641**

Telegram: @uni_k

www.konkur.in



1 The Deep Learning Revolution

Machine learning today is one of the most important, and fastest growing, fields of technology. Applications of machine learning are becoming ubiquitous, and solutions learned from data are increasingly displacing traditional hand-crafted algorithms. This has not only led to improved performance for existing technologies but has opened the door to a vast range of new capabilities that would be inconceivable if new algorithms had to be designed explicitly by hand.

One particular branch of machine learning, known as *deep learning*, has emerged as an exceptionally powerful and general-purpose framework for learning from data. Deep learning is based on computational models called *neural networks* which were originally inspired by mechanisms of learning and information processing in the human brain. The field of *artificial intelligence*, or AI, seeks to recreate the powerful capabilities of the brain in machines, and today the terms machine learning and AI are often used interchangeably. Many of the AI systems in current use represent ap-

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024
C. M. Bishop, H. Bishop, *Deep Learning*, https://doi.org/10.1007/978-3-031-45468-4_1

Telegram: @uni_k

www.konkur.in

plications of machine learning which are designed to solve very specific and focused problems, and while these are extremely useful they fall far short of the tremendous breadth of capabilities of the human brain. This has led to the introduction of the term *artificial general intelligence*, or AGI, to describe the aspiration of building machines with this much greater flexibility. After many decades of steady progress, machine learning has now entered a phase of very rapid development. Recently, massive deep learning systems called large language models have started to exhibit remarkable capabilities that have been described as the first indications of artificial general intelligence (Bubeck *et al.*, 2023).

Chapter 12

1.1. The Impact of Deep Learning

We begin our discussion of machine learning by considering four examples drawn from diverse fields to illustrate the huge breadth of applicability of this technology and to introduce some basic concepts and terminology. What is particularly remarkable about these and many other examples is that they have all been addressed using variants of the same fundamental framework of deep learning. This is in sharp contrast to conventional approaches in which different applications are tackled using widely differing and specialist techniques. It should be emphasized that the examples we have chosen represent only a tiny fraction of the breadth of applicability for deep neural networks and that almost every domain where computation has a role is amenable to the transformational impact of deep learning.

1.1.1 Medical diagnosis

Consider first the application of machine learning to the problem of diagnosing skin cancer. Melanoma is the most dangerous kind of skin cancer but is curable if detected early. Figure 1.1 shows example images of skin lesions, with malignant melanomas on the top row and benign nevi on the bottom row. Distinguishing between these two classes of image is clearly very challenging, and it would be virtually impossible to write an algorithm by hand that could successfully classify such images with any reasonable level of accuracy.

This problem has been successfully addressed using deep learning (Esteva *et al.*, 2017). The solution was created using a large set of lesion images, known as

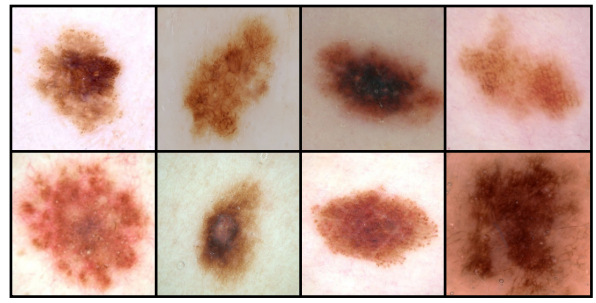


Figure 1.1 Examples of skin lesions corresponding to dangerous malignant melanomas on the top row and benign nevi on the bottom row. It is difficult for the untrained eye to distinguish between these two classes.

Telegram: @uni_k

www.konkur.in

a *training set*, each of which is labelled as either malignant or benign, where the labels are obtained from a biopsy test that is considered to provide the true class of the lesion. The training set is used to determine the values of some 25 million adjustable parameters, known as weights, in a deep neural network. This process of setting the parameter values from data is known as *learning* or *training*. The goal is for the trained network to predict the correct label for a new lesion just from the image alone without needing the time-consuming step of taking a biopsy. This is an example of a *supervised learning* problem because, for each training example, the network is told the correct label. It is also an example of a *classification* problem because each input must be assigned to a discrete set of classes (benign or malignant in this case). Applications in which the output consists of one or more continuous variables are called *regression* problems. An example of a regression problem would be the prediction of the yield in a chemical manufacturing process in which the inputs consist of the temperature, the pressure, and the concentrations of reactants.

An interesting aspect of this application is that the number of labelled training images available, roughly 129,000, is considered relatively small, and so the deep neural network was first trained on a much larger data set of 1.28 million images of everyday objects (such as dogs, buildings, and mushrooms) and then *fine-tuned* on the data set of lesion images. This is an example of *transfer learning* in which the network learns the general properties of natural images from the large data set of everyday objects and is then specialized to the specific problem of lesion classification. Through the use of deep learning, the classification of skin lesion images has reached a level of accuracy that exceeds that of professional dermatologists (Brinker *et al.*, 2019).

1.1.2 Protein structure

Proteins are sometimes called the building blocks of living organisms. They are biological molecules that consist of one or more long chains of units called amino acids, of which there are 22 different types, and the protein is specified by the sequence of amino acids. Once a protein has been synthesized inside a living cell, it folds into a complex three-dimensional structure whose behaviour and interactions are strongly determined by its shape. Calculating this 3D structure, given the amino acid sequence, has been a fundamental open problem in biology for half a century that had seen relatively little progress until the advent of deep learning.

The 3D structure can be measured experimentally using techniques such as X-ray crystallography, cryogenic electron microscopy, or nuclear magnetic resonance spectroscopy. However, this can be extremely time-consuming and for some proteins can prove to be challenging, for example due to the difficulty of obtaining a pure sample or because the structure is dependent on the context. In contrast, the amino acid sequence of a protein can be determined experimentally at lower cost and higher throughput. Consequently, there is considerable interest in being able to predict the 3D structures of proteins directly from their amino acid sequences in order to better understand biological processes or for practical applications such as drug discovery. A deep learning model can be trained to take an amino acid sequence as input and generate the 3D structure as output, in which the training data

Telegram: @uni_k

www.konkur.in



Figure 1.2 Illustration of the 3D shape of a protein called T1044/6VR4. The green structure shows the ground truth as determined by X-ray crystallography, whereas the superimposed blue structure shows the prediction obtained by a deep learning model called AlphaFold. [From Jumper *et al.* (2021) with permission.]

consist of a set of proteins for which the amino acid sequence and the 3D structure are both known. Protein structure prediction is therefore another example of supervised learning. Once the system is trained it can take a new amino acid sequence as input and can predict the associated 3D structure (Jumper *et al.*, 2021). Figure 1.2 compares the predicted 3D structure of a protein and the ground truth obtained by X-ray crystallography.

1.1.3 Image synthesis

In the two applications discussed so far, a neural network learned to transform an input (a skin image or an amino acid sequence) into an output (a lesion classification or a 3D protein structure, respectively). We turn now to an example where the training data consist simply of a set of sample images and the goal of the trained network is to create new images of the same kind. This is an example of *unsupervised learning* because the images are unlabelled, in contrast to the lesion classification and protein structure examples. Figure 1.3 shows examples of synthetic images generated by a deep neural network trained on a set of images of human faces taken in a studio against a plain background. Such synthetic images are of exceptionally high quality and it can be difficult tell them apart from photographs of real people.

This is an example of a *generative model* because it can generate new output examples that differ from those used to train the model but which share the same statistical properties. A variant of this approach allows images to be generated that depend on an input text string known, as a *prompt*, so that the image content reflects the semantics of the text input. The term *generative AI* is used to describe deep learning models that generate outputs in the form of images, video, audio, text, candidate drug molecules, or other modalities.

Chapter 10

Telegram: @uni_k

www.konkur.in



Figure 1.3 Synthetic face images generated by a deep neural network trained using unsupervised learning. [From https://generated.photos.]

1.1.4 Large language models

One of most important advances in machine learning in recent years has been the development of powerful models for processing natural language and other forms of sequential data such as source code. A *large language model*, or LLM, uses deep learning to build rich internal representations that capture the semantic properties of language. An important class of large language models, called *autoregressive* language models, can generate language as output, and therefore, they are a form of generative AI. Such models take a sequence of words as the input and for the output, generate a single word that represents the next word in the sequence. The augmented sequence, with the new word appended at the end, can then be fed through the model again to generate the subsequent word, and this process can be repeated to generate a long sequence of words. Such models can also output a special 'stop' word that signals the end of text generation, thereby allowing them to output text of finite length and then halt. At that point, a user could append their own series of words to the sequence before feeding the complete sequence back through the model to trigger further word generation. In this way, it is possible for a human to have a conversation with the neural network.

Such models can be trained on large data sets of text by extracting training pairs each consisting of a randomly selected sequence of words as input with the known next word as the target output. This is an example of *self-supervised learning* in which a function from inputs to outputs is learned but where the labelled outputs are obtained automatically from the input training data without needing separate human-

Telegram: @uni_k

www.konkur.in

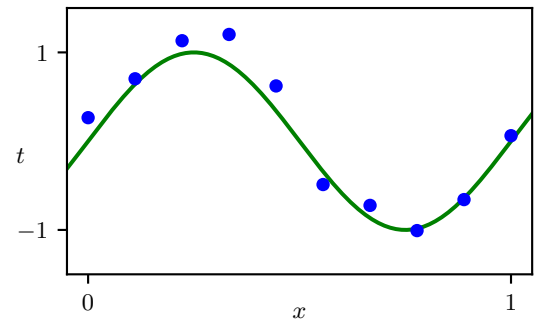


Figure 1.4 Plot of a training data set of $N = 10$ points, shown as blue circles, each comprising an observation of the input variable x along with the corresponding target variable t. The green curve shows the function $\sin(2\pi x)$ used to generate the data. Our goal is to predict the value of t for some new value of x, without knowledge of the green curve.

derived labels. Since large volumes of text are available from multiple sources, this approach allows for scaling to very large training sets and associated very large neural networks.

Large language models can exhibit extraordinary capabilities that have been described as the first indications of emerging artificial general intelligence (Bubeck *et al.*, 2023), and we discuss such models at length later in the book. On the next page, we give an illustration of language generation, based on a model called GPT-4 (OpenAI, 2023), in response to an input prompt '*Write a proof of the fact that there are infinitely many primes; do it in the style of a Shakespeare play through a dialogue between two parties arguing over the proof.*'.

Chapter 12

1.2. A Tutorial Example

For the newcomer to the field of machine learning, many of the basic concepts and much of the terminology can be introduced in the context of a simple example involving the fitting of a polynomial to a small synthetic data set (Bishop, 2006). This is a form of supervised learning problem in which we would like to make a prediction for a target variable, given the value of an input variable.

1.2.1 Synthetic data

We denote the input variable by x and the target variable by t, and we assume that both variables take continuous values on the real axis. Suppose that we are given a training set comprising N observations of x, written $x_1, \ldots, x_N$, together with corresponding observations of the values of t, denoted $t_1, \ldots, t_N$. Our goal is to predict the value of t for some new value of x. The ability to make accurate predictions on previously unseen inputs is a key goal in machine learning and is known as *generalization*.

We can illustrate this using a synthetic data set generated by sampling from a sinusoidal function. Figure 1.4 shows a plot of a training set comprising $N = 10$ data points in which the input values were generated by choosing values of x_n, for $n = 1, \ldots, N$, spaced uniformly in the range $[0, 1]$. The associated target data values were obtained by first computing the values of the function $\sin(2\pi x)$ for each value of x

Telegram: @uni_k

www.konkur.in

GPT-4: Proof of infinitely many primes

```
FIRST PARTY:
Thou dost not understand, my friend,
There simply is no upper end
To primes, those numbers indivisible,
No matter if it seems impossible.

SECOND PARTY:
But how dost thou so surely know?
To prove this truth, what canst thou show?

FIRST PARTY:
Consider this, my doubtful peer,
A clever proof that will make clear:
Assume that there's a final prime,
The largest one we'll see in time.
Let's call this number "P" for short,
And then, my friend, we will purport
To build a number "Q" that's new,
One larger than "P" and prime too.

SECOND PARTY:
But how, my friend, will this be done?
The proof is not yet fairly won.

FIRST PARTY:
It's simple, friend, just wait and see,
We'll multiply the primes, that's key.
All primes from two to "P" combined,
Will make this new "Q" we will find.
But one thing more we must include,
To make the proof more watertight,
We'll add a one onto the end,
And now the proof is in our sight.

SECOND PARTY:
I see it now, it's crystal clear,
For "Q" must be a prime, my dear.
It can't divide by any prime,
All smaller than itself in time.
And since we chose "P" as the last,
This proves that our assumption's past.
There is no final prime, we see,
The numbers go on endlessly.
```

Telegram: @uni_k

www.konkur.in

Section 2.3

and then adding a small level of random noise (governed by a Gaussian distribution) to each such point to obtain the corresponding target value t_n. By generating data in this way, we are capturing an important property of many real-world data sets, namely that they possess an underlying regularity, which we wish to learn, but that individual observations are corrupted by random noise. This noise might arise from intrinsically *stochastic* (i.e., random) processes such as radioactive decay but more typically is due to there being sources of variability that are themselves unobserved.

In this tutorial example we know the true process that generated the data, namely the sinusoidal function. In a practical application of machine learning, our goal is to discover the underlying trends in the data given the finite training set. Knowing the process that generated the data, however, allows us to illustrate important concepts in machine learning.

1.2.2 Linear models

Our goal is to exploit this training set to predict the value $\widehat{t}$ of the target variable for some new value $\widehat{x}$ of the input variable. As we will see later, this involves implicitly trying to discover the underlying function $\sin(2\pi x)$. This is intrinsically a difficult problem as we have to generalize from a finite data set to an entire function. Furthermore, the observed data is corrupted with noise, and so for a given $\widehat{x}$ there is uncertainty as to the appropriate value for $\widehat{t}$. *Probability theory* provides a framework for expressing such uncertainty in a precise and quantitative manner, whereas *decision theory* allows us to exploit this probabilistic representation to make predictions that are optimal according to appropriate criteria. Learning probabilities from data lies at the heart of machine learning and will be explored in great detail in this book.

Chapter 2

Chapter 5

To start with, however, we will proceed rather informally and consider a simple approach based on curve fitting. In particular, we will fit the data using a polynomial function of the form

$$y(x, \mathbf{w}) = w_0 + w_1 x + w_2 x^2 + \ldots + w_M x^M = \sum_{j=0}^{M} w_j x^j \tag{1.1}$$

where M is the *order* of the polynomial, and x^j denotes x raised to the power of j. The polynomial coefficients $w_0, \ldots, w_M$ are collectively denoted by the vector $\mathbf{w}$. Note that, although the polynomial function $y(x, \mathbf{w})$ is a nonlinear function of x, it is a linear function of the coefficients $\mathbf{w}$. Functions, such as this polynomial, that are linear in the unknown parameters have important properties, as well as significant limitations, and are called *linear models*.

Chapter 4

1.2.3 Error function

The values of the coefficients will be determined by fitting the polynomial to the training data. This can be done by minimizing an *error function* that measures the misfit between the function $y(x, \mathbf{w})$, for any given value of $\mathbf{w}$, and the training set data points. One simple choice of error function, which is widely used, is the sum of

Telegram: @uni_k

www.konkur.in

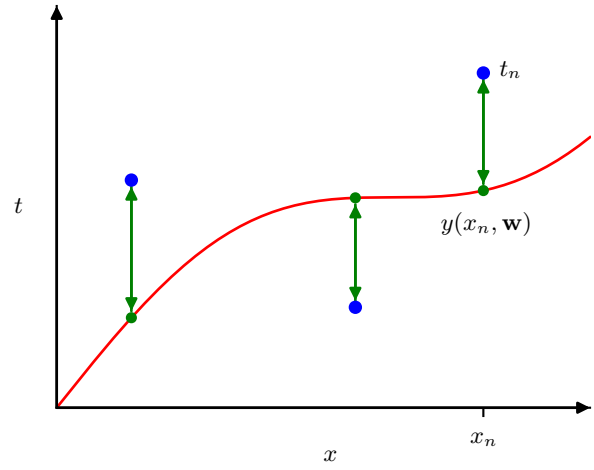


Figure 1.5 The error function (1.2) corresponds to (one half of) the sum of the squares of the displacements (shown by the vertical green arrows) of each data point from the function $y(x, \mathbf{w})$.

the squares of the differences between the predictions $y(x_n, \mathbf{w})$ for each data point x_n and the corresponding target value t_n, given by

$$E(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N}\{y(x_n, \mathbf{w}) - t_n\}^2 \tag{1.2}$$

where the factor of $1/2$ is included for later convenience. We will later derive this error function starting from probability theory. Here we simply note that it is a non-negative quantity that would be zero if, and only if, the function $y(x, \mathbf{w})$ were to pass exactly through each training data point. The geometrical interpretation of the sum-of-squares error function is illustrated in Figure 1.5.

Section 2.3.4

We can solve the curve fitting problem by choosing the value of $\mathbf{w}$ for which $E(\mathbf{w})$ is as small as possible. Because the error function is a quadratic function of the coefficients $\mathbf{w}$, its derivatives with respect to the coefficients will be linear in the elements of $\mathbf{w}$, and so the minimization of the error function has a unique solution, denoted by $\mathbf{w}^\star$, which can be found in closed form. The resulting polynomial is given by the function $y(x, \mathbf{w}^\star)$.

Exercise 4.1

1.2.4 Model complexity

There remains the problem of choosing the order M of the polynomial, and as we will see this will turn out to be an example of an important concept called *model comparison* or *model selection*. In Figure 1.6, we show four examples of the results of fitting polynomials having orders $M = 0, 1, 3$, and 9 to the data set shown in Figure 1.4.

Notice that the constant ($M = 0$) and first-order ($M = 1$) polynomials give poor fits to the data and consequently poor representations of the function $\sin(2\pi x)$. The third-order ($M = 3$) polynomial seems to give the best fit to the function $\sin(2\pi x)$ of the examples shown in Figure 1.6. When we go to a much higher order polynomial ($M = 9$), we obtain an excellent fit to the training data. In fact, the polynomial

Telegram: @uni_k

www.konkur.in

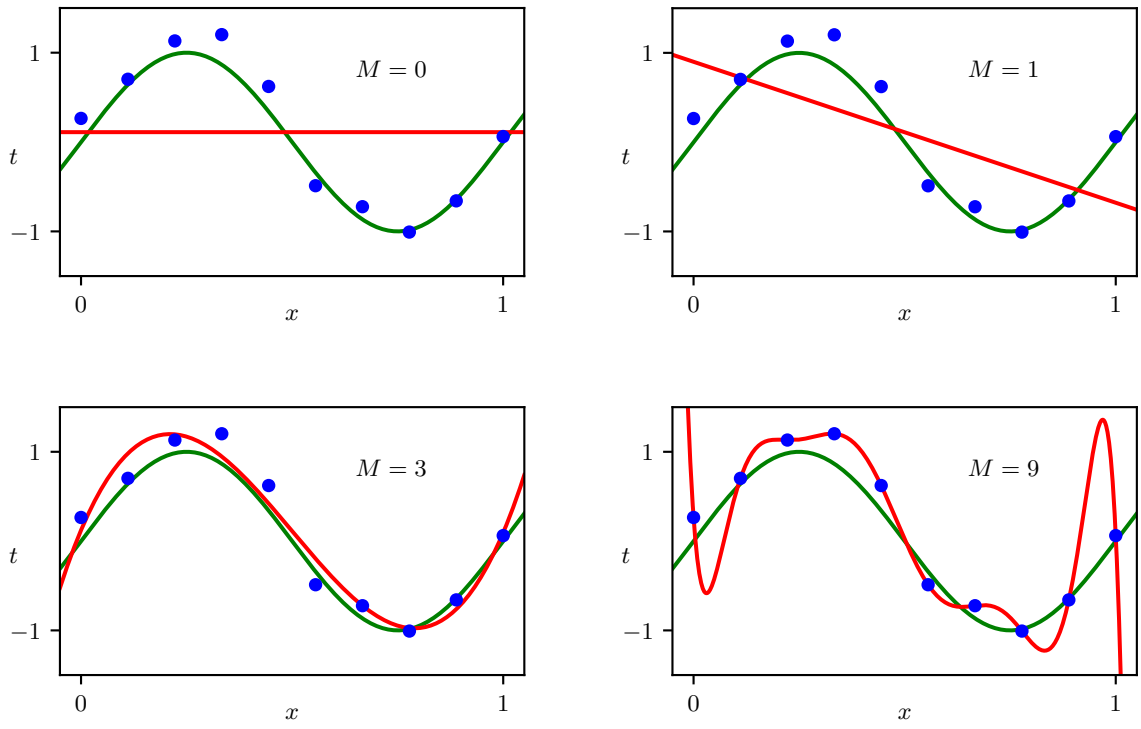


Figure 1.6 Plots of polynomials having various orders M, shown as red curves, fitted to the data set shown in Figure 1.4 by minimizing the error function (1.2).

passes exactly through each data point and $E(\mathbf{w}^\star) = 0$. However, the fitted curve oscillates wildly and gives a very poor representation of the function $\sin(2\pi x)$. This latter behaviour is known as *over-fitting*.

Our goal is to achieve good generalization by making accurate predictions for new data. We can obtain some quantitative insight into the dependence of the generalization performance on M by considering a separate set of data known as a *test set*, comprising 100 data points generated using the same procedure as used to generate the training set points. For each value of M, we can evaluate the residual value of $E(\mathbf{w}^\star)$ given by (1.2) for the training data, and we can also evaluate $E(\mathbf{w}^\star)$ for the test data set. Instead of evaluating the error function $E(\mathbf{w})$, it is sometimes more convenient to use the root-mean-square (RMS) error defined by

$$E_{\mathrm{RMS}} = \sqrt{\frac{1}{N}\sum_{n=1}^{N}\{y(x_n, \mathbf{w}) - t_n\}^2} \tag{1.3}$$

in which the division by N allows us to compare different sizes of data sets on an equal footing, and the square root ensures that E_{RMS} is measured on the same scale (and in the same units) as the target variable t. Graphs of the training-set and test-set

Telegram: @uni_k

www.konkur.in

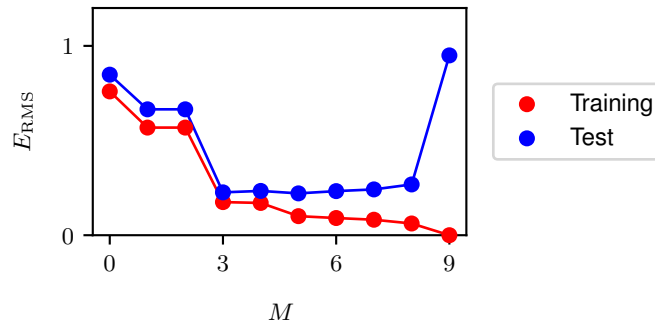


Figure 1.7 Graphs of the root-mean-square error, defined by (1.3), evaluated on the training set, and on an independent test set, for various values of M.

RMS errors are shown, for various values of M, in Figure 1.7. The test set error is a measure of how well we are doing in predicting the values of t for new data observations of x. Note from Figure 1.7 that small values of M give relatively large values of the test set error, and this can be attributed to the fact that the corresponding polynomials are rather inflexible and are incapable of capturing the oscillations in the function $\sin(2\pi x)$. Values of M in the range $3 \leqslant M \leqslant 8$ give small values for the test set error, and these also give reasonable representations of the generating function $\sin(2\pi x)$, as can be seen for $M = 3$ in Figure 1.6.

For $M = 9$, the training set error goes to zero, as we might expect because this polynomial contains 10 degrees of freedom corresponding to the 10 coefficients $w_0, \ldots, w_9$, and so can be tuned exactly to the 10 data points in the training set. However, the test set error has become very large and, as we saw in Figure 1.6, the corresponding function $y(x, \mathbf{w}^\star)$ exhibits wild oscillations.

This may seem paradoxical because a polynomial of a given order contains all lower-order polynomials as special cases. The $M = 9$ polynomial is therefore capable of generating results at least as good as the $M = 3$ polynomial. Furthermore, we might suppose that the best predictor of new data would be the function $\sin(2\pi x)$ from which the data was generated (and we will see later that this is indeed the case). We know that a power series expansion of the function $\sin(2\pi x)$ contains terms of all orders, so we might expect that results should improve monotonically as we increase M.

We can gain some insight into the problem by examining the values of the coefficients $\mathbf{w}^\star$ obtained from polynomials of various orders, as shown in Table 1.1. We see that, as M increases, the magnitude of the coefficients typically gets larger. In particular for the $M = 9$ polynomial, the coefficients have become finely tuned to the data. They have large positive and negative values so that the corresponding polynomial function matches each of the data points exactly, but between data points (particularly near the ends of the range) the function exhibits the large oscillations observed in Figure 1.6. Intuitively, what is happening is that the more flexible polynomials with larger values of M are increasingly tuned to the random noise on the target values.

Further insight into this phenomenon can be gained by examining the behaviour of the learned model as the size of the data set is varied, as shown in Figure 1.8. We see that, for a given model complexity, the over-fitting problem become less severe

Telegram: @uni_k

www.konkur.in

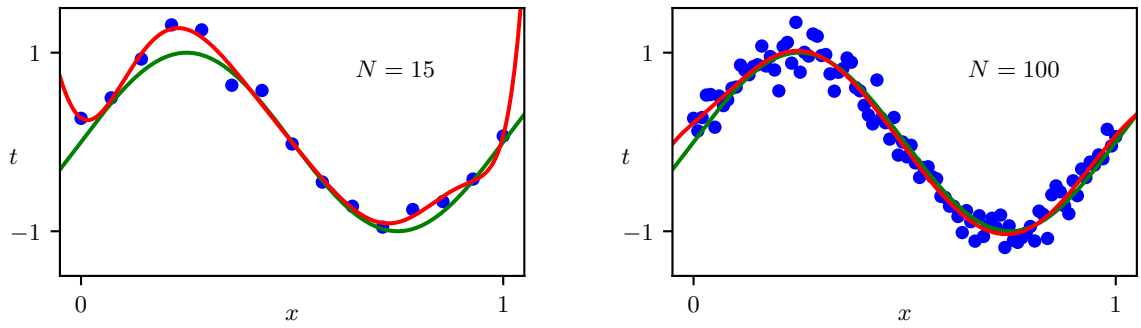


Figure 1.8 Plots of the solutions obtained by minimizing the sum-of-squares error function (1.2) using the $M = 9$ polynomial for $N = 15$ data points (left plot) and $N = 100$ data points (right plot). We see that increasing the size of the data set reduces the over-fitting problem.

as the size of the data set increases. Another way to say this is that with a larger data set, we can afford to fit a more complex (in other words more flexible) model to the data. One rough heuristic that is sometimes advocated in classical statistics is that the number of data points should be no less than some multiple (say 5 or 10) of the number of learnable parameters in the model. However, when we discuss deep learning later in this book, we will see that excellent results can be obtained using models that have significantly more parameters than the number of training data points.

Section 9.3.2

1.2.5 Regularization

There is something rather unsatisfying about having to limit the number of parameters in a model according to the size of the available training set. It would seem more reasonable to choose the complexity of the model according to the complexity of the problem being solved. One technique that is often used to control the over-fitting phenomenon, as an alternative to limiting the number of parameters, is that of *regularization*, which involves adding a penalty term to the error function (1.2) to discourage the coefficients from having large magnitudes. The simplest such penalty

Table 1.1 Table of the coefficients $\mathbf{w}^\star$ for polynomials of various order. Observe how the typical magnitude of the coefficients increases dramatically as the order of the polynomial increases.

	$M = 0$	$M = 1$	$M = 3$	$M = 9$
$w_0^\star$	0.11	0.90	0.12	0.26
$w_1^\star$		−1.58	11.20	−66.13
$w_2^\star$			−33.67	1,665.69
$w_3^\star$			22.43	−15,566.61
$w_4^\star$				76,321.23
$w_5^\star$				−217,389.15
$w_6^\star$				370,626.48
$w_7^\star$				−372,051.47
$w_8^\star$				202,540.70
$w_9^\star$				−46,080.94

Telegram: @uni_k

www.konkur.in

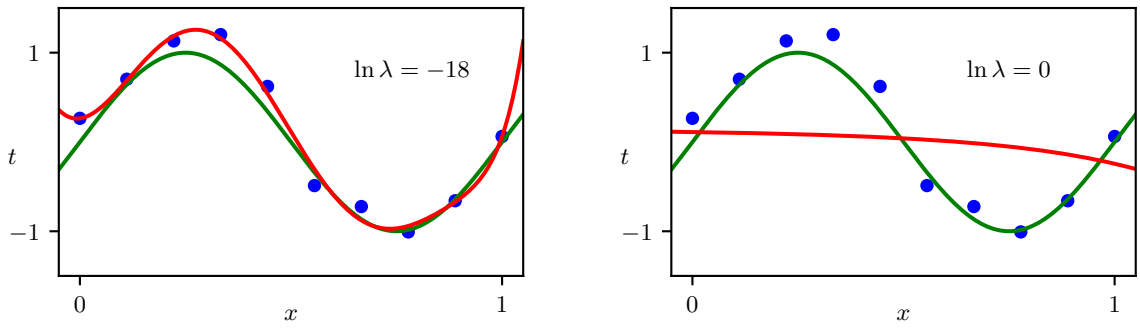


Figure 1.9 Plots of $M = 9$ polynomials fitted to the data set shown in Figure 1.4 using the regularized error function (1.4) for two values of the regularization parameter λ corresponding to $\ln \lambda = -18$ and $\ln \lambda = 0$. The case of no regularizer, i.e., $\lambda = 0$, corresponding to $\ln \lambda = -\infty$, is shown at the bottom right of Figure 1.6.

term takes the form of the sum of the squares of all of the coefficients, leading to a modified error function of the form

$$\widetilde{E}(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N}\{y(x_n, \mathbf{w}) - t_n\}^2 + \frac{\lambda}{2}\|\mathbf{w}\|^2 \tag{1.4}$$

where $\|\mathbf{w}\|^2 \equiv \mathbf{w}^{\mathrm{T}}\mathbf{w} = w_0^2 + w_1^2 + \ldots + w_M^2$, and the coefficient λ governs the relative importance of the regularization term compared with the sum-of-squares error term. Note that often the coefficient w_0 is omitted from the regularizer because its inclusion causes the results to depend on the choice of origin for the target variable (Hastie, Tibshirani, and Friedman, 2009), or it may be included but with its own regularization coefficient. Again, the error function in (1.4) can be minimized exactly in closed form. Techniques such as this are known in the statistics literature as *shrinkage* methods because they reduce the value of the coefficients. In the context of neural networks, this approach is known as *weight decay* because the parameters in a neural network are called weights and this regularizer encourages them to decay towards zero.

Section 9.2.1
Exercise 4.2

Figure 1.9 shows the results of fitting the polynomial of order $M = 9$ to the same data set as before but now using the regularized error function given by (1.4). We see that, for a value of $\ln \lambda = -18$, the over-fitting has been suppressed and we now obtain a much closer representation of the underlying function $\sin(2\pi x)$. If, however, we use too large a value for λ then we again obtain a poor fit, as shown in Figure 1.9 for $\ln \lambda = 0$. The corresponding coefficients from the fitted polynomials are given in Table 1.2, showing that regularization has the desired effect of reducing the magnitude of the coefficients.

The impact of the regularization term on the generalization error can be seen by plotting the value of the RMS error (1.3) for both training and test sets against $\ln \lambda$, as shown in Figure 1.10. We see that λ now controls the effective complexity of the model and hence determines the degree of over-fitting.

Telegram: @uni_k

www.konkur.in

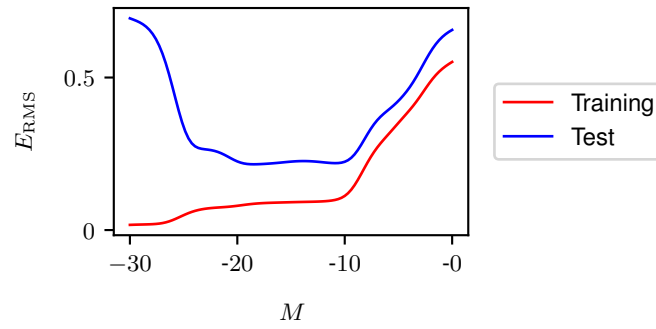


Figure 1.10 Graph of the root-mean-square error (1.3) versus $\ln \lambda$ for the $M = 9$ polynomial.

1.2.6 Model selection

The quantity λ is an example of a *hyperparameter* whose values are fixed during the minimization of the error function to determine the model parameters $\mathbf{w}$. We cannot simply determine the value of λ by minimizing the error function jointly with respect to $\mathbf{w}$ and λ since this will lead to $\lambda \to 0$ and an over-fitted model with small or zero training error. Similarly, the order M of the polynomial is a hyperparameter of the model, and simply optimizing the training set error with respect to M will lead to large values of M and associated over-fitting. We therefore need to find a way to determine suitable values for hyperparameters. The results above suggest a simple way of achieving this, namely by taking the available data and partitioning it into a training set, used to determine the coefficients $\mathbf{w}$, and a separate *validation* set, also called a *hold-out* set or a *development set*. We then select the model having the lowest error on the validation set. If the model design is iterated many times using a data set of limited size, then some over-fitting to the validation data can occur, and so it may be necessary to keep aside a third *test set* on which the performance of the selected model can finally be evaluated.

For some applications, the supply of data for training and testing will be limited. To build a good model, we should use as much of the available data as possible for training. However, if the validation set is too small, it will give a relatively noisy estimate of predictive performance. One solution to this dilemma is to use *cross-*

Table 1.2 Table of the coefficients $\mathbf{w}^\star$ for $M = 9$ polynomials with various values for the regularization parameter λ. Note that $\ln \lambda = -\infty$ corresponds to a model with no regularization, i.e., to the graph at the bottom right in Figure 1.6. We see that, as the value of λ increases, the magnitude of a typical coefficient gets smaller.

	$\ln \lambda = -\infty$	$\ln \lambda = -18$	$\ln \lambda = 0$
$w_0^\star$	0.26	0.26	0.11
$w_1^\star$	−66.13	0.64	−0.07
$w_2^\star$	1,665.69	43.68	−0.09
$w_3^\star$	−15,566.61	−144.00	−0.07
$w_4^\star$	76,321.23	57.90	−0.05
$w_5^\star$	−217,389.15	117.36	−0.04
$w_6^\star$	370,626.48	9.87	−0.02
$w_7^\star$	−372,051.47	−90.02	−0.01
$w_8^\star$	202,540.70	−70.90	−0.01
$w_9^\star$	−46,080.94	75.26	0.00

Telegram: @uni_k

www.konkur.in

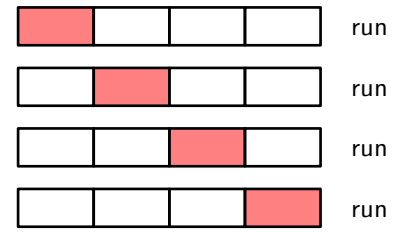


Figure 1.11 The technique of S-fold cross-validation, illustrated here for the case of $S = 4$, involves taking the available data and partitioning it into S groups of equal size. Then $S - 1$ of the groups are used to train a set of models that are then evaluated on the remaining group. This procedure is then repeated for all S possible choices for the held-out group, indicated here by the red blocks, and the performance scores from the S runs are then averaged.

validation, which is illustrated in Figure 1.11. This allows a proportion $(S-1)/S$ of the available data to be used for training while making use of all of the data to assess performance. When data is particularly scarce, it may be appropriate to consider the case $S = N$, where N is the total number of data points, which gives the *leave-one-out* technique.

The main drawback of cross-validation is that the number of training runs that must be performed is increased by a factor of S, and this can prove problematic for models in which the training is itself computationally expensive. A further problem with techniques such as cross-validation that use separate data to assess performance is that we might have multiple complexity hyperparameters for a single model (for instance, there might be several regularization hyperparameters). Exploring combinations of settings for such hyperparameters could, in the worst case, require a number of training runs that is exponential in the number of hyperparameters. The state of the art in modern machine learning involves extremely large models, trained on commensurately large data sets. Consequently, there is limited scope for exploration of hyperparameter settings, and heavy reliance is placed on experience obtained with smaller models and on heuristics.

This simple example of fitting a polynomial to a synthetic data set generated from a sinusoidal function has illustrated many key ideas from machine learning, and we will make further use of this example in future chapters. However, real-world applications of machine learning differ in several important respects. The size of the data sets used for training can be many orders of magnitude larger, and there will generally be many more input variables, perhaps numbering in the millions for image analysis, for example, as well as multiple output variables. The learnable function that relates outputs to inputs is governed by a class of models known as neural networks, and these may have a large number of parameters perhaps numbering in the hundreds of billions, and the error function will be a highly nonlinear function of those parameters. The error function can no longer be minimized through a closed-form solution and instead must be minimized through iterative optimization techniques based on evaluation of the derivatives of the error function with respect to the parameters, all of which may require specialist computational hardware and incur substantial computational cost.

Telegram: @uni_k

www.konkur.in

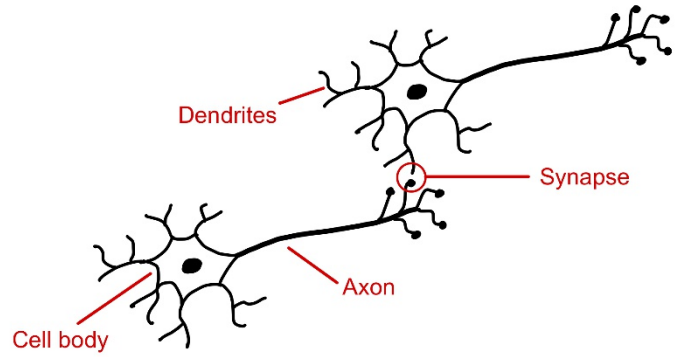


Figure 1.12 Schematic illustration showing two neurons from the human brain. These electrically active cells communicate through junctions called synapses whose strengths change as the network learns.

1.3. A Brief History of Machine Learning

Machine learning has a long and rich history, including the pursuit of multiple alternative approaches. Here we focus on the evolution of machine learning methods based on neural networks as these represent the foundation of deep learning and have proven to be the most effective approach to machine learning for real-world applications.

Neural network models were originally inspired by studies of information processing in the brains of humans and other mammals. The basic processing units in the brain are electrically active cells called neurons, as illustrated in Figure 1.12. When a neuron 'fires', it sends an electrical impulse down the axon where it reaches junctions, called synapses, which form connections with other neurons. Chemical signals called neurotransmitters are released at the synapses, and these can stimulate, or inhibit, the firing of subsequent neurons.

A human brain contains around 90 billion neurons in total, each of which has on average several thousand synapses with other neurons, creating a complex network having a total of around 100 trillion (10^{14}) synapses. If a particular neuron receives sufficient stimulation from the firing of other neurons then it too can be induced to fire. However, some synapses have a negative, or inhibitory, effect whereby the firing of the input neuron makes it less likely that the output neuron will fire. The extent to which one neuron can cause another to fire depends on the strength of the synapse, and it is changes in these strengths that represents a key mechanism whereby the brain can store information and learn from experience.

These properties of neurons have been captured in very simple mathematical models, known as *artificial neural networks*, which then form the basis for computational approaches to learning (McCulloch and Pitts, 1943). Many of these models describe the properties of a single neuron by forming a linear combination of the outputs of other neurons, which is then transformed using a nonlinear function. This

Telegram: @uni_k

www.konkur.in

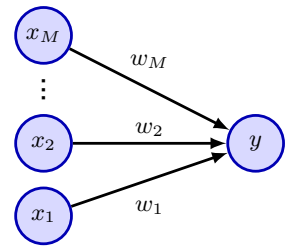


Figure 1.13 A simple neural network diagram representing the transformations (1.5) and (1.6) describing a single neuron. The polynomial function (1.1) can be seen as a special case of this model.

can be expressed mathematically in the form

$$a = \sum_{i=1}^{M} w_i x_i \tag{1.5}$$

$$y = f(a) \tag{1.6}$$

where $x_1, \ldots, x_M$ represent M inputs corresponding to the activities of other neurons that send connections to this neuron, and $w_1, \ldots, w_M$ are continuous variables, called *weights*, which represent the strengths of the associated synapses. The quantity a is called the *pre-activation*, the nonlinear function $f(\cdot)$ is called the *activation function*, and the output y is called the *activation*. We can see that the polynomial (1.1) can be viewed as a specific instance of this representation in which the inputs x_i are given by powers of a single variable x, and the function $f(\cdot)$ is just the identity $f(a) = a$. The simple mathematical formulation given by (1.5) and (1.6) has formed the basis of neural network models from the 1960s up to the present day, and can be represented in diagram form as shown in Figure 1.13.

1.3.1 Single-layer networks

The history of artificial neural networks can broadly be divided into three distinct phases according to the level of sophistication of the networks as measured by the number of 'layers' of processing. A simple neural model described by (1.5) and (1.6) can be viewed as having a single layer of processing corresponding to the single layer of connections in Figure 1.13. One of the most important such models in the history of neural computing is the *perceptron* (Rosenblatt, 1962) in which the activation function $f(\cdot)$ is a step function of the form

$$f(a) = \begin{cases} 0, & \text{if } a \leqslant 0, \\ 1, & \text{if } a > 0\,. \end{cases} \tag{1.7}$$

This can be viewed as a simplified model of neural firing in which a neuron fires if, and only if, the total weighted input exceeds a threshold of 0. The perceptron was pioneered by Rosenblatt (1962), who developed a specific training algorithm that has the interesting property that if there exists a set of weight values for which the perceptron can achieve perfect classification of its training data then the algorithm is guaranteed to find the solution in a finite number of steps (Bishop, 2006). As well as a learning algorithm, the perceptron also had a dedicated analogue hardware

Telegram: @uni_k

www.konkur.in

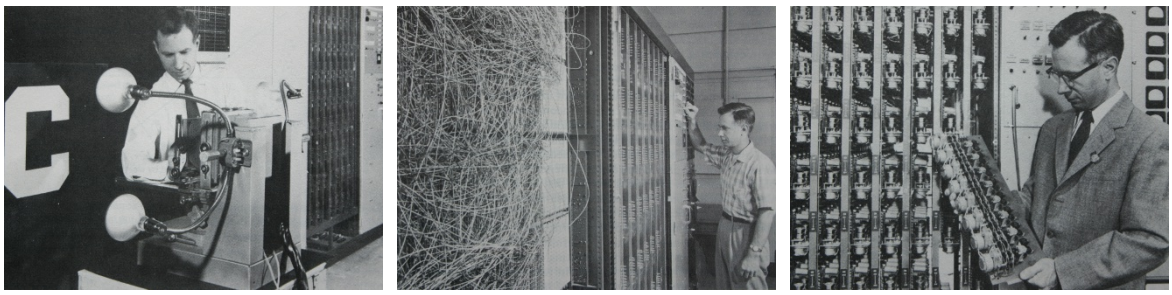


Figure 1.14 Illustration of the Mark 1 perceptron hardware. The photograph on the left shows how the inputs were obtained using a simple camera system in which an input scene, in this case a printed character, was illuminated by powerful lights, and an image focused onto a 20×20 array of cadmium sulphide photocells, giving a primitive 400-pixel image. The perceptron also had a patch board, shown in the middle photograph, which allowed different configurations of input features to be tried. Often these were wired up at random to demonstrate the ability of the perceptron to learn without the need for precise wiring, in contrast to a modern digital computer. The photograph on the right shows one of the racks of learnable weights. Each weight was implemented using a rotary variable resistor, also called a potentiometer, driven by an electric motor thereby allowing the value of the weight to be adjusted automatically by the learning algorithm.

implementation, as shown in Figure 1.14. A typical perceptron configuration had multiple layers of processing, but only one of those layers was learnable from data, and so the perceptron is considered to be a 'single-layer' neural network.

At first, the ability of perceptrons to learn from data in a brain-like way was considered remarkable. However, it became apparent that the model also has major limitations. The properties of perceptrons were analysed by Minsky and Papert (1969), who gave formal proofs of the limited capabilities of single-layer networks. Unfortunately, they also speculated that similar limitations would extend to networks having multiple layers of learnable parameters. Although this latter conjecture proved to be wildly incorrect, the effect was to dampen enthusiasm for neural network models, and this contributed to the lack of interest, and funding, for neural networks during the 1970s and early 1980s. Furthermore, researchers were unable to explore the properties of multilayered networks due to the lack of an effective algorithm for training them, since techniques such as the perceptron algorithm were specific to single-layer models. Note that although perceptrons have long disappeared from practical machine learning, the name lives on because a modern neural network is also sometimes called a *multilayer perceptron* or *MLP*.

1.3.2 Backpropagation

The solution to the problem of training neural networks having more than one layer of learnable parameters came from the use of differential calculus and the application of gradient-based optimization methods. An important change was to replace the step function (1.7) with continuous differentiable activation functions having a non-zero gradient. Another key modification was to introduce differentiable error functions that define how well a given choice of parameter values predicts the target variables in the training set. We saw an example of such an error function when we

Telegram: @uni_k

www.konkur.in

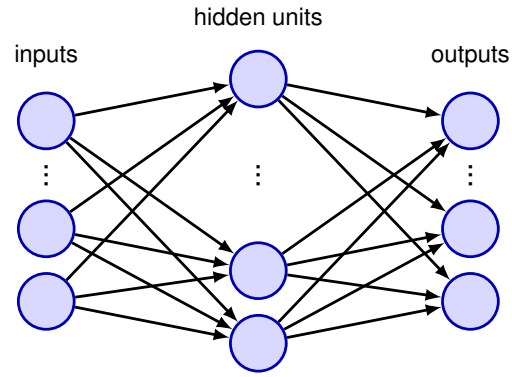


Figure 1.15 A neural network having two layers of parameters in which arrows denote the direction of information flow through the network. Each of the hidden units and each of the output units computes a function of the form given by (1.5) and (1.6) in which the activation function $f(\cdot)$ is differentiable.

Section 1.2.3

used the sum-of-squares error function (1.2) to fit polynomials.

With these changes, we now have an error function whose derivatives with respect to each of the parameters in the network can be evaluated. We can now consider networks having more than one layer of parameters. Figure 1.15 shows a simple network with two processing layers. Nodes in the middle layer called *hidden units* because their values do not appear in the training set, which only provides values for inputs and outputs. Each of the hidden units and each of the output units in Figure 1.15 computes a function of the form given by (1.5) and (1.6). For a given set of input values, the states of all of the hidden and output units can be evaluated by repeated application of (1.5) and (1.6) in which information is flowing forward through the network in the direction of the arrows. For this reason, such models are sometimes also called *feed-forward neural networks*.

To train such a network the parameters are first initialized using a random number generator and are then iteratively updated using gradient-based optimization techniques. This involves evaluating the derivatives of the error function, which

Chapter 8

can be done efficiently in a process known as *error backpropagation*. In backpropagation, information flows backwards through the network from the outputs towards the inputs (Rumelhart, Hinton, and Williams, 1986). There exist many different optimization algorithms that make use of gradients of the function to be optimized, but the one that is most prevalent in machine learning is also the simplest and is known

Chapter 7

as *stochastic gradient descent*.

The ability to train neural networks having multiple layers of weights was a breakthrough that led to a resurgence of interest in the field starting around the mid-1980s. This was also a period in which the field moved beyond a focus on neurobiological inspiration and developed a more rigorous and principled foundation (Bishop, 1995b). In particular, it was recognized that probability theory, and ideas from the field of statistics, play a central role in neural networks and machine learning. One key insight is that learning from data involves background assumptions, sometimes called *prior knowledge* or *inductive biases*. These might be incorporated explicitly, for example by designing the structure of a neural network such that the classification of a skin lesion does not depend on the location of the lesion within the image, or they might take the form of implicit assumptions that arise from the mathematical

Telegram: @uni_k

www.konkur.in

form of the model or the way it is trained.

The development of backpropagation and gradient-based optimization dramatically increased the capability of neural networks to solve practical problems. However, it was also observed that in networks with many layers, it was only weights in the final two layers that would learn useful values. With a few exceptions, notably models used for image analysis known as convolutional neural networks (LeCun *et al.*, 1998), there were very few successful applications of networks having more than two layers. Again, this constrained the complexity of the problems that could be addressed effectively with these kinds of network. To achieve reasonable performance on many applications, it was necessary to use hand-crafted *pre-processing* to transform the input variables into some new space where, it was hoped, the machine learning problem would be easier to solve. This pre-processing stage is sometimes also called *feature extraction*. Although this approach was sometimes effective, it would clearly be much better if features could be learned from the data rather than being hand-crafted.

Chapter 10

By the start of the new millennium, the available neural network methods were once again reaching the limits of their capability. Researchers began to explore a raft of alternatives to neural networks, such as kernel methods, support vector machines, Gaussian processes, and many others. Neural networks fell into disfavour once again, although a core of enthusiastic researchers continued to pursue the goal of a truly effective approach to training networks with many layers.

1.3.3 Deep networks

The third, and current, phase in the development of neural networks began during the second decade of the 21st century. A series of developments allowed neural networks with many layers of weights to be trained effectively, thereby removing previous limitations on the capabilities of these techniques. Networks with many layers of weights are called *deep neural networks* and the sub-field of machine learning that focuses on such networks is called *deep learning* (LeCun, Bengio, and Hinton, 2015).

One important theme in the origins of deep learning was a significant increase in the scale of neural networks, measured in terms of the number of parameters. Although networks with a few hundred or a few thousand parameters were common in the 1980s, this steadily rose to the millions, and then billions, whereas current state-of-the-art models can have in the region of one trillion (10^{12}) parameters. Networks with many parameters require commensurately large data sets so that the training signals can produced good values for those parameters. The combination of massive models and massive data sets in turn requires computation on a massive scale when training the model. Specialist processors called *graphics processing units*, or GPUs, which had been developed for very fast rendering of graphical data for applications such as video games, proved to be well suited to the training of neural networks because the functions computed by the units in one layer of a network can be evaluated in parallel, and this maps well onto the massive parallelism of GPUs (Krizhevsky, Sutskever, and Hinton, 2012). Today, training for the largest models is performed on large arrays of thousands of GPUs linked by specialist high-speed interconnections.

Telegram: @uni_k

www.konkur.in

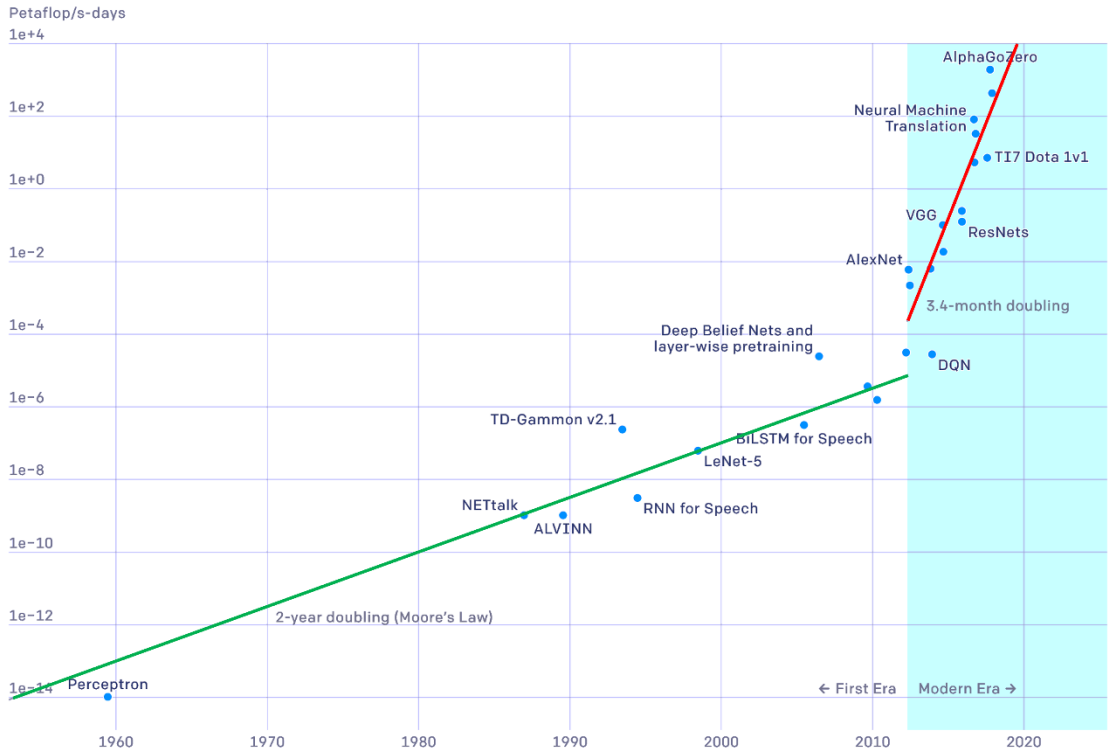


Figure 1.16 Plot of the number of compute cycles, measured in petaflop/s-days, needed to train a state-of-the-art neural network as a function of date, showing two distinct phases of exponential growth. [From OpenAI with permission.]

Figure 1.16 illustrates how the number of compute cycles needed to train a state-of-the-art neural network has grown over the years, showing two distinct phases of growth. The vertical axis has an exponential scale and has units of petaflop/s-days, where a petaflop represents 10^{15} (a thousand trillion) floating point operations, and a petaflop/s is one petaflop per second. One petaflop/s-day represents computation at the rate of a petaflop/s for a period of 24 hours, which is roughly 10^{20} floating point operations, and therefore, the top line of the graph represents an impressive 10^{24} floating point operations. A straight line on the graph represents exponential growth, and we see that from the era of the perceptron up to around 2012, the doubling time was around 2 years, which is consistent with the general growth of computing power as a consequence of Moore's law. From 2012 onward, which marks the era of deep learning, we again see exponential growth but the doubling time is now 3.4 months corresponding to a factor of 10 increase in compute power every year!

It is often found that improvements in performance due to innovations in the architecture or incorporation of more sophisticated forms of inductive bias are soon

Telegram: @uni_k

www.konkur.in

superseded simply by scaling up the quantity of training data, along with commensurate scaling of the model size and associated compute power used for training (Sutton, 2019). Not only can large models have superior performance on a specific task but they may be capable of solving a broader range of different problems with the same trained neural network. Large language models are a notable example as a single network not only has an extraordinary breadth of capability but is even able to outperform specialist networks designed to solve specific problems.

Section 12.3.5

We have seen that depth plays an important role in allowing neural networks to achieve high performance. One way to view the role of the hidden layers in a deep neural network is that of *representation learning* (Bengio, Courville, and Vincent, 2012) in which the network learns to transform input data into a new representation that is semantically meaningful thereby creating a much easier problem for the final layer or layers to solve. Such internal representations can be repurposed to allow for the solution of related problems through transfer learning, as we saw for skin lesion classification. It is interesting to note that neural networks used to process images may learn internal representations that are remarkably like those observed in the mammalian visual cortex. Large neural networks that can be adapted or *fine-tuned* to a range of downstream tasks are called *foundation models*, and can take advantage of large, heterogeneous data sets to create models having broad applicability (Bommasani *et al.*, 2021).

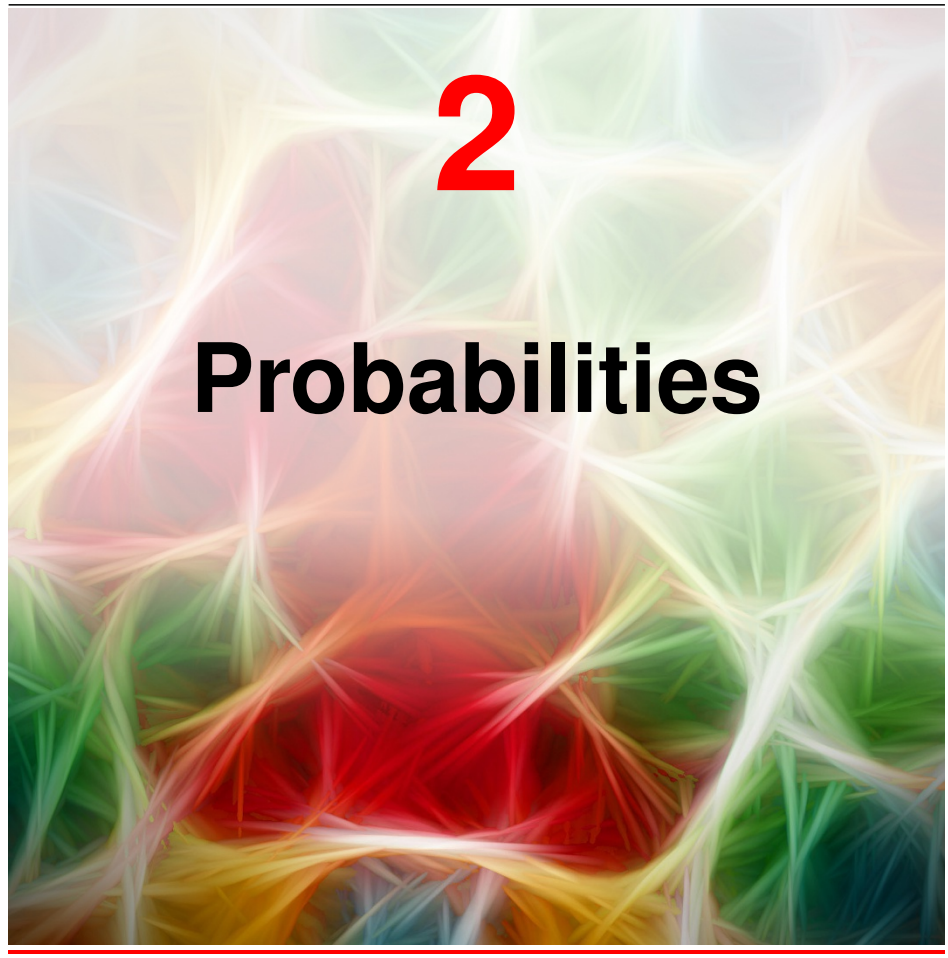
Section 10.3

In addition to scaling, there were other developments that helped in the success of deep learning. For example, in simple neural networks, the training signals become weaker as they are backpropagated through successive layers of a deep network. One technique for addressing this is the introduction of *residual connections* (He *et al.*, 2015a) that facilitate the training of networks having hundreds of layers. Another key development was the introduction of *automatic differentiation* methods in which the code that performs backpropagation to evaluate error function gradients is generated automatically from the code used to specify the forward propagation. This allows researchers to experiment rapidly with different architectures for a neural network and to combine different architectural elements in multiple ways very easily since only the relatively simple forward propagation functions need to be coded explicitly. Also, much of the research in machine learning has been conducted through open source, allowing researchers to build on the work of others, thereby further accelerating the rate of progress in the field.

Section 9.5

Telegram: @uni_k

www.konkur.in



2 Probabilities

In almost every application of machine learning we have to deal with uncertainty. For example, a system that classifies images of skin lesions as benign or malignant can never in practice achieve perfect accuracy. We can distinguish between two kinds of uncertainty. The first is *epistemic uncertainty* (derived from the Greek word episteme meaning knowledge), sometimes called *systematic uncertainty*. It arises because we only get to see data sets of finite size. As we observe more data, for instance more examples of benign and malignant skin lesion images, we are better able to predict the class of a new example. However, even with an infinitely large data set, we would still not be able to achieve perfect accuracy due to the second kind of uncertainty known as *aleatoric uncertainty*, also called *intrinsic* or *stochastic* uncertainty, or sometimes simply called *noise*. Generally speaking, the noise arises because we are able to observe only partial information about the world, and therefore, one way to reduce this source of uncertainty is to gather different kinds of data. This is illustrated

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024
C. M. Bishop, H. Bishop, *Deep Learning*, https://doi.org/10.1007/978-3-031-45468-4_2

Telegram: @uni_k

www.konkur.in

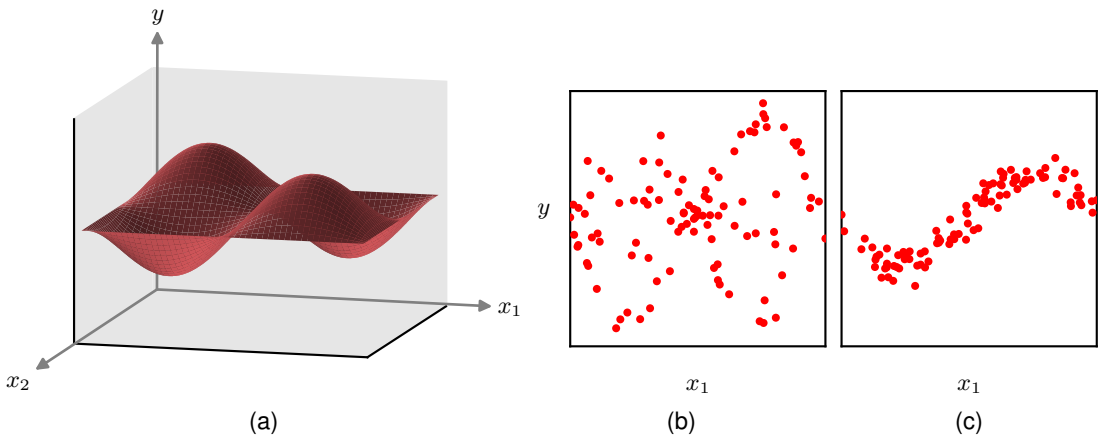


Figure 2.1 An extension of the simple sine curve regression problem to two dimensions. (a) A plot of the function $y(x_1, x_2) = \sin(2\pi x_1)\sin(2\pi x_2)$. Data is generated by selecting values for x_1 and x_2, computing the corresponding value of $y(x_1, x_2)$, and then adding Gaussian noise. (b) Plot of 100 data points in which x_2 is unobserved showing high levels of noise. (c) Plot of 100 data points in which x_2 is fixed to the value $x_2 = \frac{\pi}{2}$, simulating the effect of being able to measure x_2 as well as x_1, showing much lower levels of noise.

Section 1.2

using an extension of the sine curve example to two dimensions in Figure 2.1.

As a practical example of this, a biopsy sample of the skin lesion is much more informative than the image alone and might greatly improve the accuracy with which we can determine if a new lesion is malignant. Given both the image and the biopsy data, the intrinsic uncertainty might be very small, and by collecting a large training data set, we may be able to reduce the systematic uncertainty to a low level and thereby make predictions of the class of the lesion with high accuracy.

Both kinds of uncertainty can be handled using the framework of *probability theory*, which provides a consistent paradigm for the quantification and manipulation of uncertainty and therefore forms one of the central foundations for machine learning. We will see that probabilities are governed by two simple formulae known as the *sum rule* and the *product rule*. When coupled with *decision theory*, these rules allow us, at least in principle, to make optimal predictions given all the information available to us, even though that information may be incomplete or ambiguous.

Section 2.1
Section 5.2

The concept of probability is often introduced in terms of frequencies of repeatable events. Consider, for example, the bent coin shown in Figure 2.2, and suppose that the shape of the coin is such that if it is flipped a large number of times, it lands concave side up 60% of the time, and therefore lands convex side up 40% of the time. We say that the *probability* of landing concave side up is 60% or 0.6. Strictly, the probability is defined in the limit of an infinite number of 'trials' or coin flips in this case. Because the coin must land either concave side up or convex side up, these probabilities add to 100% or 1.0. This definition of probability in terms of the frequency of repeatable events is the basis for the *frequentist* view of statistics.

Now suppose that, although we know that the probability that the coin will land concave side up is 0.6, we are not allowed to look at the coin itself and we do not

Telegram: @uni_k

www.konkur.in



Figure 2.2 Probability can be viewed either as a frequency associated with a repeatable event or as a quantification of uncertainty. A bent coin can be used to illustrate the difference, as discussed in the text.

know which side is heads and which is tails. If asked to take a bet on whether the coin will land heads or tails when flipped, then symmetry suggests that our bet should be based on the assumption that the probability of seeing heads is 0.5, and indeed a more careful analysis shows that, in the absence of any additional information, this is indeed the rational choice. Here we are using probabilities in a more general sense than simply the frequency of events. Whether the convex side of the coin is heads or tails is not itself a repeatable event, it is simply unknown. The use of probability as a quantification of uncertainty is the *Bayesian* perspective and is more general in that it includes frequentist probability as a special case. We can learn about which side of the coin is heads if we are given results from a sequence of coin flips by making use of Bayesian reasoning. The more results we observe, the lower our uncertainty as to which side of the coin is which.

Section 2.6

Exercise 2.40

Having introduced the concept of probability informally, we turn now to a more detailed exploration of probabilities and discuss how to use them quantitatively. Concepts developed in the remainder of this chapter will form a core foundation for many of the topics discussed throughout the book.

2.1. The Rules of Probability

In this section we will derive two simple rules that govern the behaviour of probabilities. However, in spite of their apparent simplicity, these rules will prove to be very powerful and widely applicable. We will motivate the rules of probability by first introducing a simple example.

2.1.1 A medical screening example

Consider the problem of screening a population in order to provide early detection of cancer, and let us suppose that 1% of the population actually have cancer. Ideally our test for cancer would give a positive result for anyone who has cancer and a negative result for anyone who does not. However, tests are not perfect, so we will suppose that when the test is given to people who are free of cancer, 3% of them will test positive. These are known as *false positives*. Similarly, when the test is given to people who do have cancer, 10% of them will test negative. These are called *false negatives*. The various error rates are illustrated in Figure 2.3.

Given this information, we might ask the following questions: (1) 'If we screen the population, what is the probability that someone will test positive?', (2) 'If some-

Telegram: @uni_k

www.konkur.in

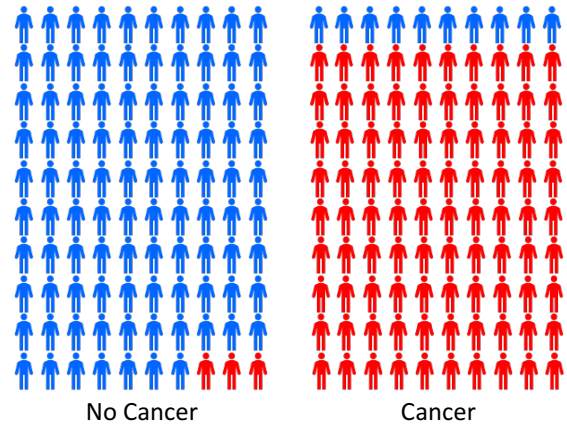


Figure 2.3 Illustration of the accuracy of a cancer test. Out of every hundred people taking the test who do not have cancer, shown on the left, on average three will test positive. For those who have cancer, shown on the right, out of every hundred people taking the test, on average 90 will test positive.

one receives a positive test result, what is the probability that they actually have cancer?'. We could answer such questions by working through the cancer screening case in detail. Instead, however, we will pause our discussion of this specific example and first derive the general rules of probability, known as the *sum rule of probability* and the *product rule*. We will then illustrate the use of these rules by answering our two questions.

2.1.2 The sum and product rules

To derive the rules of probability, consider the slightly more general example shown in Figure 2.4 involving two variables X and Y. In our cancer example, X could represent the presence or absence of cancer, and Y could be a variable denoting the outcome of the test. Because the values of these variables can vary from one person to another in a way that is generally unknown, they are called *random variables* or *stochastic variables*. We will suppose that X can take any of the values x_i where $i = 1, \ldots, L$ and that Y can take the values y_j where $j = 1, \ldots, M$. Consider a total of N trials in which we sample both of the variables X and Y, and let the number of such trials in which $X = x_i$ and $Y = y_j$ be n_{ij}. Also, let the number of trials in which X takes the value x_i (irrespective of the value that Y takes) be denoted by c_i, and similarly let the number of trials in which Y takes the value y_j be denoted by r_j.

The probability that X will take the value x_i and Y will take the value y_j is written $p(X = x_i, Y = y_j)$ and is called the *joint* probability of $X = x_i$ and $Y = y_j$. It is given by the number of points falling in the cell i,j as a fraction of the total number of points, and hence

$$p(X = x_i, Y = y_j) = \frac{n_{ij}}{N}. \tag{2.1}$$

Here we are implicitly considering the limit $N \to \infty$. Similarly, the probability that X takes the value x_i irrespective of the value of Y is written as $p(X = x_i)$ and is

Telegram: @uni_k

www.konkur.in

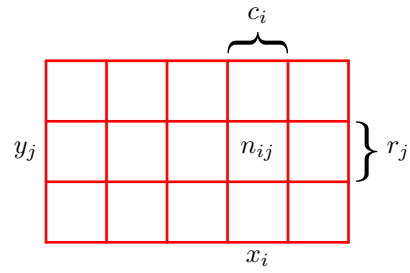


Figure 2.4 We can derive the sum and product rules of probability by considering a random variable X, which takes the values $\{x_i\}$ where $i = 1, \ldots, L$, and a second random variable Y, which takes the values $\{y_j\}$ where $j = 1, \ldots, M$. In this illustration, we have $L = 5$ and $M = 3$. If we consider the total number N of instances of these variables, then we denote the number of instances where $X = x_i$ and $Y = y_j$ by n_{ij}, which is the number of instances in the corresponding cell of the array. The number of instances in column i, corresponding to $X = x_i$, is denoted by c_i, and the number of instances in row j, corresponding to $Y = y_j$, is denoted by r_j.

given by the fraction of the total number of points that fall in column i, so that

$$p(X = x_i) = \frac{c_i}{N}. \tag{2.2}$$

Since $\sum_i c_i = N$, we see that

$$\sum_{i=1}^{L} p(X = x_i) = 1 \tag{2.3}$$

and, hence, the probabilities sum to one as required. Because the number of instances in column i in Figure 2.4 is just the sum of the number of instances in each cell of that column, we have $c_i = \sum_j n_{ij}$ and therefore, from (2.1) and (2.2), we have

$$p(X = x_i) = \sum_{j=1}^{M} p(X = x_i, Y = y_j), \tag{2.4}$$

which is the *sum rule* of probability. Note that $p(X = x_i)$ is sometimes called the *marginal* probability and is obtained by marginalizing, or summing out, the other variables (in this case Y).

If we consider only those instances for which $X = x_i$, then the fraction of such instances for which $Y = y_j$ is written $p(Y = y_j|X = x_i)$ and is called the *conditional* probability of $Y = y_j$ given $X = x_i$. It is obtained by finding the fraction of those points in column i that fall in cell i,j and, hence, is given by

$$p(Y = y_j|X = x_i) = \frac{n_{ij}}{c_i}. \tag{2.5}$$

Summing both sides over j and using $\sum_j n_{ij} = c_i$, we obtain

$$\sum_{j=1}^{M} p(Y = y_j|X = x_i) = 1 \tag{2.6}$$

Telegram: @uni_k

www.konkur.in

showing that the conditional probabilities are correctly normalized. From (2.1), (2.2), and (2.5), we can then derive the following relationship:

$$p(X = x_i, Y = y_j) = \frac{n_{ij}}{N} = \frac{n_{ij}}{c_i} \cdot \frac{c_i}{N}$$
$$= p(Y = y_j|X = x_i)p(X = x_i), \tag{2.7}$$

which is the *product rule* of probability.

So far, we have been quite careful to make a distinction between a random variable, such as X, and the values that the random variable can take, for example x_i. Thus, the probability that X takes the value x_i is denoted $p(X = x_i)$. Although this helps to avoid ambiguity, it leads to a rather cumbersome notation, and in many cases there will be no need for such pedantry. Instead, we may simply write $p(X)$ to denote a distribution over the random variable X, or $p(x_i)$ to denote the distribution evaluated for the particular value x_i, provided that the interpretation is clear from the context.

With this more compact notation, we can write the two fundamental rules of probability theory in the following form:

sum rule $$p(X) = \sum_Y p(X, Y) \tag{2.8}$$

product rule $$p(X, Y) = p(Y|X)p(X). \tag{2.9}$$

Here $p(X, Y)$ is a joint probability and is verbalized as 'the probability of X *and* Y'. Similarly, the quantity $p(Y|X)$ is a conditional probability and is verbalized as 'the probability of Y *given* X'. Finally, the quantity $p(X)$ is a marginal probability and is simply 'the probability of X'. These two simple rules form the basis for all of the probabilistic machinery that we will use throughout this book.

2.1.3 Bayes' theorem

From the product rule, together with the symmetry property $p(X, Y) = p(Y, X)$, we immediately obtain the following relationship between conditional probabilities:

$$p(Y|X) = \frac{p(X|Y)p(Y)}{p(X)}, \tag{2.10}$$

which is called *Bayes' theorem* and which plays an important role in machine learning. Note how Bayes' theorem relates the conditional distribution $p(Y|X)$ on the left-hand side of the equation, to the 'reversed' conditional distribution $p(X|Y)$ on the right-hand side. Using the sum rule, the denominator in Bayes' theorem can be expressed in terms of the quantities appearing in the numerator:

$$p(X) = \sum_Y p(X|Y)p(Y). \tag{2.11}$$

Thus, we can view the denominator in Bayes' theorem as being the normalization constant required to ensure that the sum over the conditional probability distribution on the left-hand side of (2.10) over all values of Y equals one.

Telegram: @uni_k

www.konkur.in

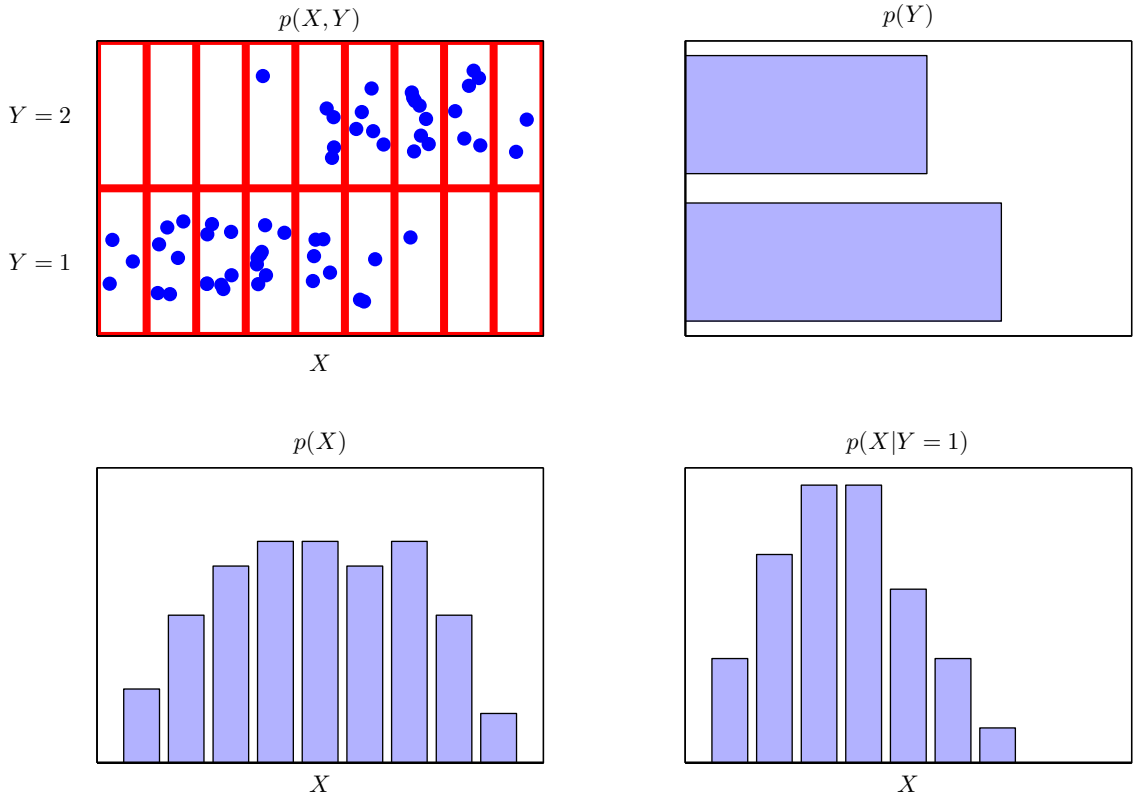


Figure 2.5 An illustration of a distribution over two variables, X, which takes nine possible values, and Y, which takes two possible values. The top left figure shows a sample of 60 points drawn from a joint probability distribution over these variables. The remaining figures show histogram estimates of the marginal distributions $p(X)$ and $p(Y)$, as well as the conditional distribution $p(X|Y = 1)$ corresponding to the bottom row in the top left figure.

Section 3.5.1

In Figure 2.5, we show a simple example involving a joint distribution over two variables to illustrate the concept of marginal and conditional distributions. Here a finite sample of $N = 60$ data points has been drawn from the joint distribution and is shown in the top left. In the top right is a histogram of the fractions of data points having each of the two values of Y. From the definition of probability, these fractions would equal the corresponding probabilities $p(Y)$ in the limit when the sample size $N \to \infty$. We can view the histogram as a simple way to model a probability distribution given only a finite number of points drawn from that distribution. The remaining two plots in Figure 2.5 show the corresponding histogram estimates of $p(X)$ and $p(X|Y = 1)$.

Telegram: @uni_k

www.konkur.in

2.1.4 Medical screening revisited

Let us now return to our cancer screening example and apply the sum and product rules of probability to answer our two questions. For clarity, when working through this example, we will once again be explicit about distinguishing between the random variables and their instantiations. We will denote the presence or absence of cancer by the variable C, which can take two values: $C = 0$ corresponds to 'no cancer' and $C = 1$ corresponds to 'cancer'. We have assumed that one person in a hundred in the population has cancer, and so we have

$$p(C = 1) = 1/100 \tag{2.12}$$
$$p(C = 0) = 99/100, \tag{2.13}$$

respectively. Note that these satisfy $p(C = 0) + p(C = 1) = 1$.

Now let us introduce a second random variable T representing the outcome of a screening test, where $T = 1$ denotes a positive result, indicative of cancer, and $T = 0$ a negative result, indicative of the absence of cancer. As illustrated in Figure 2.3, we know that for those who have cancer the probability of a positive test result is 90%, while for those who do not have cancer the probability of a positive test result is 3%. We can therefore write out all four conditional probabilities:

$$p(T = 1|C = 1) = 90/100 \tag{2.14}$$
$$p(T = 0|C = 1) = 10/100 \tag{2.15}$$
$$p(T = 1|C = 0) = 3/100 \tag{2.16}$$
$$p(T = 0|C = 0) = 97/100. \tag{2.17}$$

Again, note that these probabilities are normalized so that

$$p(T = 1|C = 1) + p(T = 0|C = 1) = 1 \tag{2.18}$$

and similarly

$$p(T = 1|C = 0) + p(T = 0|C = 0) = 1. \tag{2.19}$$

We can now use the sum and product rules of probability to answer our first question and evaluate the overall probability that someone who is tested at random will have a positive test result:

$$\begin{aligned} p(T = 1) &= p(T = 1|C = 0)p(C = 0) + p(T = 1|C = 1)p(C = 1) \\ &= \frac{3}{100} \times \frac{99}{100} + \frac{90}{100} \times \frac{1}{100} = \frac{387}{10,000} = 0.0387. \end{aligned} \tag{2.20}$$

We see that if a person is tested at random there is a roughly 4% chance that the test will be positive even though there is a 1% chance that they actually have cancer. From this it follows, using the sum rule, that $p(T = 0) = 1 - 387/10,000 = 9613/10,000 = 0.9613$ and, hence, there is a roughly 96% chance that the do not have cancer.

Now consider our second question, which is the one that is of particular interest to a person being screened: if a test is positive, what is the probability that the person

Telegram: @uni_k

www.konkur.in

has cancer? This requires that we evaluate the probability of cancer conditional on the outcome of the test, whereas the probabilities in (2.14) to (2.17) give the probability distribution over the test outcome conditioned on whether the person has cancer. We can solve the problem of reversing the conditional probability by using Bayes' theorem (2.10) to give

$$p(C=1|T=1) = \frac{p(T=1|C=1)p(C=1)}{p(T=1)} \tag{2.21}$$

$$= \frac{90}{100} \times \frac{1}{100} \times \frac{10,000}{387} = \frac{90}{387} \simeq 0.23 \tag{2.22}$$

so that if a person is tested at random and the test is positive, there is a 23% probability that they actually have cancer. From the sum rule, it then follows that $p(C = 0|T = 1) = 1 - 90/387 = 297/387 \simeq 0.77$, which is a 77% chance that they do not have cancer.

2.1.5 Prior and posterior probabilities

We can use the cancer screening example to provide an important interpretation of Bayes' theorem as follows. If we had been asked whether someone is likely to have cancer, before they have received a test, then the most complete information we have available is provided by the probability $p(C)$. We call this the *prior probability* because it is the probability available *before* we observe the result of the test. Once we are told that this person has received a positive test, we can then use Bayes' theorem to compute the probability $p(C|T)$, which we will call the *posterior probability* because it is the probability obtained *after* we have observed the test result T.

In this example, the prior probability of having cancer is 1%. However, once we have observed that the test result is positive, we find that the posterior probability of cancer is now 23%, which is a substantially higher probability of cancer, as we would intuitively expect. We note, however, that a person with a positive test still has only a 23% change of actually having cancer, even though the test appears, from Figure 2.3 to be reasonably 'accurate'. This conclusion seems counter-intuitive to many people. The reason has to do with the low prior probability of having cancer. Although the test provides strong evidence of cancer, this has to be combined with the prior probability using Bayes' theorem to arrive at the correct posterior probability.

Exercise 2.1

2.1.6 Independent variables

Finally, if the joint distribution of two variables factorizes into the product of the marginals, so that $p(X, Y) = p(X)p(Y)$, then X and Y are said to be *independent*. An example of independent events would be the successive flips of a coin. From the product rule, we see that $p(Y|X) = p(Y)$, and so the conditional distribution of Y given X is indeed independent of the value of X. In our cancer screening example, if the probability of a positive test is independent of whether the person has cancer, then $p(T|C) = p(T)$, which means that from Bayes' theorem (2.10) we have $p(C|T) = p(C)$, and therefore probability of cancer is not changed by observing the test outcome. Of course, such a test would be useless because the outcome of the test tells us nothing about whether the person has cancer.

Telegram: @uni_k

www.konkur.in

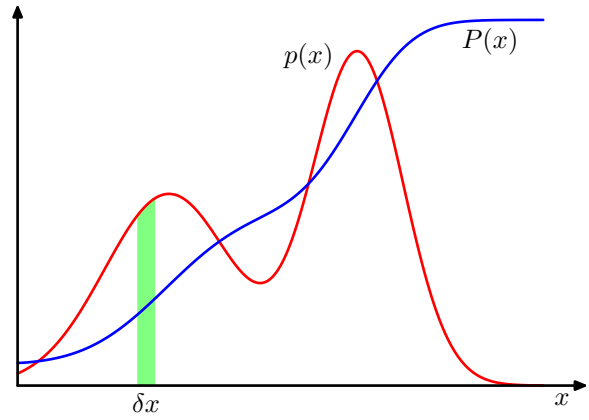


Figure 2.6 The concept of probability for discrete variables can be extended to that of a probability density $p(x)$ over a continuous variable x and is such that the probability of x lying in the interval $(x, x + \delta x)$ is given by $p(x)\delta x$ for $\delta x \to 0$. The probability density can be expressed as the derivative of a cumulative distribution function $P(x)$.

2.2. Probability Densities

As well as considering probabilities defined over discrete sets of values, we also wish to consider probabilities with respect to continuous variables. For instance, we might wish to predict what dose of drug to give to a patient. Since there will be uncertainty in this prediction, we want to quantify this uncertainty and again we can make use of probabilities. However, we cannot simply apply the concepts of probability discussed so far directly, since the probability of observing a specific value for a continuous variable, to infinite precision, will effectively be zero. Instead, we need to introduce the concept of a *probability density*. Here we will limit ourselves to a relatively informal discussion.

We define the probability density $p(x)$ over a continuous variable x to be such that the probability of x falling in the interval $(x, x + \delta x)$ is given by $p(x)\delta x$ for $\delta x \to 0$. This is illustrated in Figure 2.6. The probability that x will lie in an interval (a, b) is then given by

$$p(x \in (a, b)) = \int_a^b p(x)\,\mathrm{d}x. \tag{2.23}$$

Because probabilities are non-negative, and because the value of x must lie somewhere on the real axis, the probability density $p(x)$ must satisfy the two conditions

$$p(x) \geqslant 0 \tag{2.24}$$

$$\int_{-\infty}^{\infty} p(x)\,\mathrm{d}x = 1. \tag{2.25}$$

The probability that x lies in the interval $(-\infty, z)$ is given by the *cumulative distribution function* defined by

$$P(z) = \int_{-\infty}^{z} p(x)\,\mathrm{d}x, \tag{2.26}$$

Telegram: @uni_k

www.konkur.in

which satisfies $P'(x) = p(x)$, as shown in Figure 2.6.

If we have several continuous variables $x_1, \ldots, x_D$, denoted collectively by the vector $\mathbf{x}$, then we can define a joint probability density $p(\mathbf{x}) = p(x_1, \ldots, x_D)$ such that the probability of $\mathbf{x}$ falling in an infinitesimal volume $\delta\mathbf{x}$ containing the point $\mathbf{x}$ is given by $p(\mathbf{x})\delta\mathbf{x}$. This multivariate probability density must satisfy

$$p(\mathbf{x}) \geqslant 0 \tag{2.27}$$

$$\int p(\mathbf{x})\,\mathrm{d}\mathbf{x} = 1 \tag{2.28}$$

in which the integral is taken over the whole of $\mathbf{x}$ space. More generally, we can also consider joint probability distributions over a combination of discrete and continuous variables.

The sum and product rules of probability, as well as Bayes' theorem, also apply to probability densities as well as to combinations of discrete and continuous variables. If $\mathbf{x}$ and $\mathbf{y}$ are two real variables, then the sum and product rules take the form

$$\textbf{sum rule} \qquad p(\mathbf{x}) = \int p(\mathbf{x}, \mathbf{y})\,\mathrm{d}\mathbf{y} \tag{2.29}$$

$$\textbf{product rule} \qquad p(\mathbf{x}, \mathbf{y}) = p(\mathbf{y}|\mathbf{x})p(\mathbf{x}). \tag{2.30}$$

Similarly, Bayes' theorem can be written in the form

$$p(\mathbf{y}|\mathbf{x}) = \frac{p(\mathbf{x}|\mathbf{y})p(\mathbf{y})}{p(\mathbf{x})} \tag{2.31}$$

where the denominator is given by

$$p(\mathbf{x}) = \int p(\mathbf{x}|\mathbf{y})p(\mathbf{y})\,\mathrm{d}\mathbf{y}. \tag{2.32}$$

A formal justification of the sum and product rules for continuous variables requires a branch of mathematics called *measure theory* (Feller, 1966) and lies outside the scope of this book. Its validity can be seen informally, however, by dividing each real variable into intervals of width Δ and considering the discrete probability distribution over these intervals. Taking the limit $\Delta \to 0$ then turns sums into integrals and gives the desired result.

2.2.1 Example distributions

There are many forms of probability density that are in widespread use and that are important both in their own right and as building blocks for more complex probabilistic models. The simplest form would be one in which $p(x)$ is a constant, independent of x, but this cannot be normalized because the integral in (2.28) will be divergent. Distributions that cannot be normalized are called *improper*. We can, however, have the uniform distribution that is constant over a finite region, say (c, d), and zero elsewhere, in which case (2.28) implies

$$p(x) = 1/(d - c), \qquad x \in (c, d). \tag{2.33}$$

Telegram: @uni_k

www.konkur.in

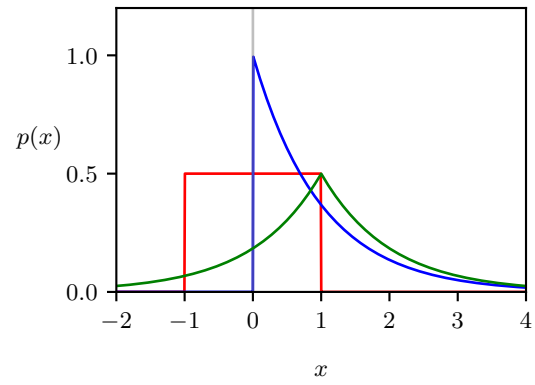


Figure 2.7 Plots of a uniform distribution over the range $(-1, 1)$, shown in red, the exponential distribution with $\lambda = 1$, shown in blue, and a Laplace distribution with $\mu = 1$ and $\gamma = 1$, shown in green.

Another simple form of density is the *exponential distribution* given by

$$p(x|\lambda) = \lambda \exp(-\lambda x), \qquad x \geqslant 0. \tag{2.34}$$

A variant of the exponential distribution, known as the *Laplace distribution*, allows the peak to be moved to a location μ and is given by

$$p(x|\mu, \gamma) = \frac{1}{2\gamma} \exp\left(-\frac{|x - \mu|}{\gamma}\right). \tag{2.35}$$

The constant, exponential, and Laplace distributions are illustrated in Figure 2.7.

Another important distribution is the *Dirac delta function*, which is written

$$p(x|\mu) = \delta(x - \mu). \tag{2.36}$$

This is defined to be zero everywhere except at $x = \mu$ and to have the property of integrating to unity according to (2.28). Informally, we can think of this as an infinitely narrow and infinitely tall spike located at $x = \mu$ with the property of having unit area. Finally, if we have a finite set of observations of x given by $\mathcal{D} = \{x_1, \ldots, x_N\}$ then we can use the delta function to construct the *empirical distribution* given by

$$p(x|\mathcal{D}) = \frac{1}{N} \sum_{n=1}^{N} \delta(x - x_n), \tag{2.37}$$

Exercise 2.6 which consists of a Dirac delta function centred on each of the data points. The probability density defined by (2.37) integrates to one as required.

2.2.2 Expectations and covariances

One of the most important operations involving probabilities is that of finding weighted averages of functions. The weighted average of some function $f(x)$ under a probability distribution $p(x)$ is called the *expectation* of $f(x)$ and will be denoted by $\mathbb{E}[f]$. For a discrete distribution, it is given by summing over all possible values of x in the form

$$\mathbb{E}[f] = \sum_{x} p(x) f(x) \tag{2.38}$$

Telegram: @uni_k

www.konkur.in

where the average is weighted by the relative probabilities of the different values of x. For continuous variables, expectations are expressed in terms of an integration with respect to the corresponding probability density:

$$\mathbb{E}[f] = \int p(x)f(x)\,\mathrm{d}x. \tag{2.39}$$

In either case, if we are given a finite number N of points drawn from the probability distribution or probability density, then the expectation can be approximated as a finite sum over these points:

Exercise 2.7

$$\mathbb{E}[f] \simeq \frac{1}{N}\sum_{n=1}^{N} f(x_n). \tag{2.40}$$

The approximation in (2.40) becomes exact in the limit $N \to \infty$.

Sometimes we will be considering expectations of functions of several variables, in which case we can use a subscript to indicate which variable is being averaged over, so that for instance

$$\mathbb{E}_x[f(x, y)] \tag{2.41}$$

denotes the average of the function $f(x, y)$ with respect to the distribution of x. Note that $\mathbb{E}_x[f(x, y)]$ will be a function of y.

We can also consider a *conditional expectation* with respect to a conditional distribution, so that

$$\mathbb{E}_x[f|y] = \sum_x p(x|y)f(x), \tag{2.42}$$

which is also a function of y. For continuous variables, the conditional expectation takes the form

$$\mathbb{E}_x[f|y] = \int p(x|y)f(x)\,\mathrm{d}x. \tag{2.43}$$

The *variance* of $f(x)$ is defined by

$$\mathrm{var}[f] = \mathbb{E}\left[(f(x) - \mathbb{E}[f(x)])^2\right] \tag{2.44}$$

and provides a measure of how much $f(x)$ varies around its mean value $\mathbb{E}[f(x)]$. Expanding out the square, we see that the variance can also be written in terms of the expectations of $f(x)$ and $f(x)^2$:

Exercise 2.8

$$\mathrm{var}[f] = \mathbb{E}[f(x)^2] - \mathbb{E}[f(x)]^2. \tag{2.45}$$

In particular, we can consider the variance of the variable x itself, which is given by

$$\mathrm{var}[x] = \mathbb{E}[x^2] - \mathbb{E}[x]^2. \tag{2.46}$$

For two random variables x and y, the *covariance* measures the extent to which the two variables vary together and is defined by

$$\begin{aligned} \mathrm{cov}[x, y] &= \mathbb{E}_{x,y}\left[\{x - \mathbb{E}[x]\}\{y - \mathbb{E}[y]\}\right] \\ &= \mathbb{E}_{x,y}[xy] - \mathbb{E}[x]\mathbb{E}[y]. \end{aligned} \tag{2.47}$$

Telegram: @uni_k

www.konkur.in

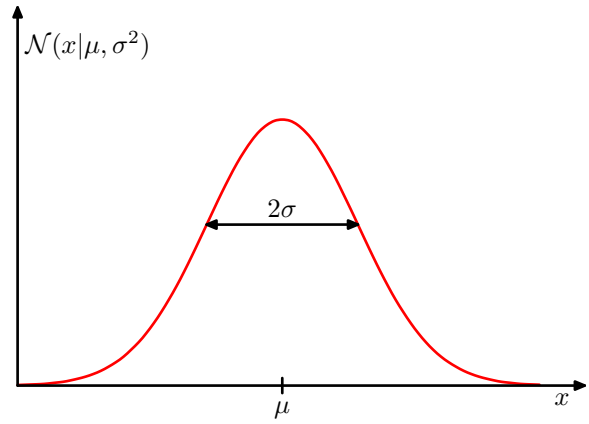


Figure 2.8 Plot of a Gaussian distribution for a single continuous variable x showing the mean μ and the standard deviation σ.

Exercise 2.9

If x and y are independent, then their covariance equals zero.

For two vectors $\mathbf{x}$ and $\mathbf{y}$, their covariance is a matrix given by

$$\begin{aligned} \mathrm{cov}[\mathbf{x}, \mathbf{y}] &= \mathbb{E}_{\mathbf{x},\mathbf{y}}\left[\{\mathbf{x} - \mathbb{E}[\mathbf{x}]\}\{\mathbf{y}^{\mathrm{T}} - \mathbb{E}[\mathbf{y}^{\mathrm{T}}]\}\right] \\ &= \mathbb{E}_{\mathbf{x},\mathbf{y}}[\mathbf{x}\mathbf{y}^{\mathrm{T}}] - \mathbb{E}[\mathbf{x}]\mathbb{E}[\mathbf{y}^{\mathrm{T}}]. \end{aligned} \tag{2.48}$$

If we consider the covariance of the components of a vector $\mathbf{x}$ with each other, then we use a slightly simpler notation $\mathrm{cov}[\mathbf{x}] \equiv \mathrm{cov}[\mathbf{x}, \mathbf{x}]$.

2.3. The Gaussian Distribution

One of the most important probability distributions for continuous variables is called the *normal* or *Gaussian* distribution, and we will make extensive use of this distribution throughout the rest of the book. For a single real-valued variable x, the Gaussian distribution is defined by

$$\mathcal{N}\left(x|\mu, \sigma^2\right) = \frac{1}{(2\pi\sigma^2)^{1/2}} \exp\left\{-\frac{1}{2\sigma^2}(x-\mu)^2\right\}, \tag{2.49}$$

which represents a probability density over x governed by two parameters: μ, called the *mean*, and σ^2, called the *variance*. The square root of the variance, given by σ, is called the *standard deviation*, and the reciprocal of the variance, written as $\beta = 1/\sigma^2$, is called the *precision*. We will see the motivation for this terminology shortly. Figure 2.8 shows a plot of the Gaussian distribution. Although the form of the Gaussian distribution might seem arbitrary, we will see later that it arises naturally from the concept of maximum entropy and from the perspective of the central limit theorem.

Section 2.5.4
Section 3.2

From (2.49) we see that the Gaussian distribution satisfies

$$\mathcal{N}(x|\mu, \sigma^2) > 0. \tag{2.50}$$

Telegram: @uni_k

www.konkur.in

Exercise 2.12 Also, it is straightforward to show that the Gaussian is normalized, so that

$$\int_{-\infty}^{\infty} \mathcal{N}\left(x|\mu, \sigma^2\right)\, \mathrm{d}x = 1. \tag{2.51}$$

Thus, (2.49) satisfies the two requirements for a valid probability density.

2.3.1 Mean and variance

Exercise 2.13 We can readily find expectations of functions of x under the Gaussian distribution. In particular, the average value of x is given by

$$\mathbb{E}[x] = \int_{-\infty}^{\infty} \mathcal{N}\left(x|\mu, \sigma^2\right) x\, \mathrm{d}x = \mu. \tag{2.52}$$

Because the parameter μ represents the average value of x under the distribution, it is referred to as the mean. The integral in (2.52) is known as the *first-order moment* of the distribution because it is the expectation of x raised to the power one. We can similarly evaluate the second-order moment given by

$$\mathbb{E}[x^2] = \int_{-\infty}^{\infty} \mathcal{N}\left(x|\mu, \sigma^2\right) x^2\, \mathrm{d}x = \mu^2 + \sigma^2. \tag{2.53}$$

From (2.52) and (2.53), it follows that the variance of x is given by

$$\mathrm{var}[x] = \mathbb{E}[x^2] - \mathbb{E}[x]^2 = \sigma^2 \tag{2.54}$$

and hence σ^2 is referred to as the variance parameter. The maximum of a distribution is known as its mode. *Exercise 2.14* For a Gaussian, the mode coincides with the mean.

2.3.2 Likelihood function

Suppose that we have a data set of observations represented as a row vector $\mathbf{x} = (x_1, \ldots, x_N)$, representing N observations of the scalar variable x. Note that we are using the typeface $\mathbf{x}$ to distinguish this from a single observation of a D-dimensional vector-valued variable, which we represent by a column vector $\mathbf{x} = (x_1, \ldots, x_D)^{\mathrm{T}}$. We will suppose that the observations are drawn independently from a Gaussian distribution whose mean μ and variance σ^2 are unknown, and we would like to determine these parameters from the data set. The problem of estimating a distribution, given a finite set of observations, is known as *density estimation*. It should be emphasized that the problem of density estimation is fundamentally ill-posed, because there are infinitely many probability distributions that could have given rise to the observed finite data set. Indeed, any distribution $p(\mathbf{x})$ that is non-zero at each of the data points $\mathbf{x}_1, \ldots, \mathbf{x}_N$ is a potential candidate. Here we constrain the space of distributions to be Gaussians, which leads to a well-defined solution.

Data points that are drawn independently from the same distribution are said to be *independent and identically distributed*, which is often abbreviated to i.i.d. or IID. We have seen that the joint probability of two independent events is given by the product of the marginal probabilities for each event separately. Because our data

Telegram: @uni_k

www.konkur.in

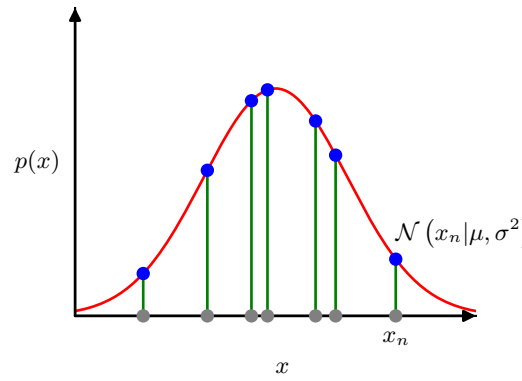


Figure 2.9 Illustration of the likelihood function for the Gaussian distribution shown by the red curve. Here the grey points denote a data set of values $\{x_n\}$, and the likelihood function (2.55) is given by the product of the corresponding values of $p(x)$ denoted by the blue points. Maximizing the likelihood involves adjusting the mean and variance of the Gaussian so as to maximize this product.

set $\mathbf{x}$ is i.i.d., we can therefore write the probability of the data set, given μ and σ^2, in the form

$$p(\mathbf{x}|\mu,\sigma^2) = \prod_{n=1}^{N} \mathcal{N}\left(x_n|\mu,\sigma^2\right). \tag{2.55}$$

When viewed as a function of μ and σ^2, this is called the *likelihood function* for the Gaussian and is interpreted diagrammatically in Figure 2.9.

One common approach for determining the parameters in a probability distribution using an observed data set, known as *maximum likelihood*, is to find the parameter values that maximize the likelihood function. This might appear to be a strange criterion because, from our foregoing discussion of probability theory, it would seem more natural to maximize the probability of the parameters given the data, not the probability of the data given the parameters. In fact, these two criteria are related.

Section 2.6.2

To start with, however, we will determine values for the unknown parameters μ and σ^2 in the Gaussian by maximizing the likelihood function (2.55). In practice, it is more convenient to maximize the log of the likelihood function. Because the logarithm is a monotonically increasing function of its argument, maximizing the log of a function is equivalent to maximizing the function itself. Taking the log not only simplifies the subsequent mathematical analysis, but it also helps numerically because the product of a large number of small probabilities can easily underflow the numerical precision of the computer, and this is resolved by computing the sum of the log probabilities instead. From (2.49) and (2.55), the log likelihood function can be written in the form

$$\ln p\left(\mathbf{x}|\mu,\sigma^2\right) = -\frac{1}{2\sigma^2}\sum_{n=1}^{N}(x_n-\mu)^2 - \frac{N}{2}\ln\sigma^2 - \frac{N}{2}\ln(2\pi). \tag{2.56}$$

Maximizing (2.56) with respect to μ, we obtain the maximum likelihood solution given by

Exercise 2.15

$$\mu_{\mathrm{ML}} = \frac{1}{N}\sum_{n=1}^{N} x_n, \tag{2.57}$$

Telegram: @uni_k

www.konkur.in

which is the *sample mean*, i.e., the mean of the observed values $\{x_n\}$. Similarly, maximizing (2.56) with respect to σ^2, we obtain the maximum likelihood solution for the variance in the form

$$\sigma^2_{\mathrm{ML}} = \frac{1}{N}\sum_{n=1}^{N}(x_n - \mu_{\mathrm{ML}})^2, \tag{2.58}$$

which is the *sample variance* measured with respect to the sample mean μ_{ML}. Note that we are performing a joint maximization of (2.56) with respect to μ and σ^2, but for a Gaussian distribution, the solution for μ decouples from that for σ^2 so that we can first evaluate (2.57) and then subsequently use this result to evaluate (2.58).

2.3.3 Bias of maximum likelihood

The technique of maximum likelihood is widely used in deep learning and forms the foundation for most machine learning algorithms. However, it has some limitations, which we can illustrate using a univariate Gaussian.

We first note that the maximum likelihood solutions μ_{ML} and σ^2_{ML} are functions of the data set values $x_1, \ldots, x_N$. Suppose that each of these values has been generated independently from a Gaussian distribution whose true parameters are μ and σ^2. Now consider the expectations of μ_{ML} and σ^2_{ML} with respect to these data set values. It is straightforward to show that

Exercise 2.16

$$\mathbb{E}[\mu_{\mathrm{ML}}] = \mu \tag{2.59}$$

$$\mathbb{E}[\sigma^2_{\mathrm{ML}}] = \left(\frac{N-1}{N}\right)\sigma^2. \tag{2.60}$$

We see that, when averaged over data sets of a given size, the maximum likelihood solution for the mean will equal the true mean. However, the maximum likelihood estimate of the variance will underestimate the true variance by a factor $(N-1)/N$. This is an example of a phenomenon called *bias* in which the estimator of a random quantity is systematically different from the true value. The intuition behind this result is given by Figure 2.10.

Note that bias arises because the variance is measured relative to the maximum likelihood estimate of the mean, which itself is tuned to the data. Suppose instead we had access to the true mean μ and we used this to determine the variance using the estimator

$$\widehat{\sigma}^2 = \frac{1}{N}\sum_{n=1}^{N}(x_n - \mu)^2. \tag{2.61}$$

Exercise 2.17

Then we find that

$$\mathbb{E}\left[\widehat{\sigma}^2\right] = \sigma^2, \tag{2.62}$$

which is unbiased. Of course, we do not have access to the true mean but only to the observed data values. From the result (2.60) it follows that for a Gaussian distribution, the following estimate for the variance parameter is unbiased:

$$\widetilde{\sigma}^2 = \frac{N}{N-1}\sigma^2_{\mathrm{ML}} = \frac{1}{N-1}\sum_{n=1}^{N}(x_n - \mu_{\mathrm{ML}})^2. \tag{2.63}$$

Telegram: @uni_k

www.konkur.in

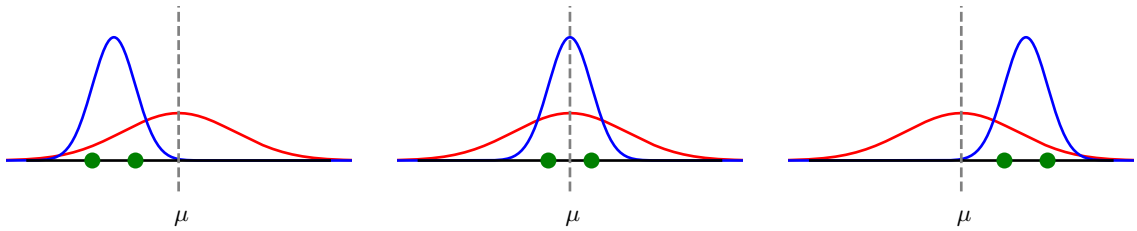


Figure 2.10 Illustration of how bias arises when using maximum likelihood to determine the mean and variance of a Gaussian. The red curves show the true Gaussian distribution from which data is generated, and the three blue curves show the Gaussian distributions obtained by fitting to three data sets, each consisting of two data points shown in green, using the maximum likelihood results (2.57) and (2.58). Averaged across the three data sets, the mean is correct, but the variance is systematically underestimated because it is measured relative to the sample mean and not relative to the true mean.

Correcting for the bias of maximum likelihood in complex models such as neural networks is not so easy, however.

Note that the bias of the maximum likelihood solution becomes less significant as the number N of data points increases. In the limit $N \to \infty$ the maximum likelihood solution for the variance equals the true variance of the distribution that generated the data. In the case of the Gaussian, for anything other than small N, this bias will not prove to be a serious problem. However, throughout this book we will be interested in complex models with many parameters, for which the bias problems associated with maximum likelihood will be much more severe. In fact, the issue of bias in maximum likelihood is closely related to the problem of *over-fitting*.

Section 2.6.3

2.3.4 Linear regression

Section 1.2

We have seen how the problem of linear regression can be expressed in terms of error minimization. Here we return to this example and view it from a probabilistic perspective, thereby gaining some insights into error functions and regularization.

The goal in the regression problem is to be able to make predictions for the target variable t given some new value of the input variable x by using a set of training data comprising N input values $\mathbf{x} = (x_1, \ldots, x_N)$ and their corresponding target values $\mathbf{t} = (t_1, \ldots, t_N)$. We can express our uncertainty over the value of the target variable using a probability distribution. For this purpose, we will assume that, given the value of x, the corresponding value of t has a Gaussian distribution with a mean equal to the value $y(x, \mathbf{w})$ of the polynomial curve given by (1.1), where $\mathbf{w}$ are the polynomial coefficients, and a variance σ^2. Thus, we have

$$p(t|x, \mathbf{w}, \sigma^2) = \mathcal{N}\left(t|y(x, \mathbf{w}), \sigma^2\right). \tag{2.64}$$

This is illustrated schematically in Figure 2.11.

We now use the training data $\{\mathbf{x}, \mathbf{t}\}$ to determine the values of the unknown parameters $\mathbf{w}$ and σ^2 by maximum likelihood. If the data is assumed to be drawn

Telegram: @uni_k

www.konkur.in

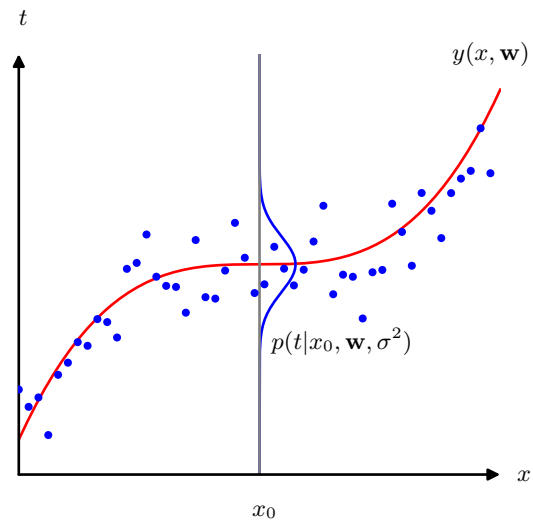


Figure 2.11 Schematic illustration of a Gaussian conditional distribution for t given x, defined by (2.64), in which the mean is given by the polynomial function $y(x, \mathbf{w})$, and the variance is given by the parameter σ^2.

independently from the distribution (2.64), then the likelihood function is given by

$$p(\mathbf{t}|\mathbf{x}, \mathbf{w}, \sigma^2) = \prod_{n=1}^{N} \mathcal{N}\left(t_n|y(x_n, \mathbf{w}), \sigma^2\right). \tag{2.65}$$

As we did for the simple Gaussian distribution earlier, it is convenient to maximize the logarithm of the likelihood function. Substituting for the Gaussian distribution, given by (2.49), we obtain the log likelihood function in the form

$$\ln p(\mathbf{t}|\mathbf{x}, \mathbf{w}, \sigma^2) = -\frac{1}{2\sigma^2}\sum_{n=1}^{N}\{y(x_n, \mathbf{w}) - t_n\}^2 - \frac{N}{2}\ln\sigma^2 - \frac{N}{2}\ln(2\pi). \tag{2.66}$$

Consider first the evaluation of the maximum likelihood solution for the polynomial coefficients, which will be denoted by $\mathbf{w}_{\mathrm{ML}}$. These are determined by maximizing (2.66) with respect to $\mathbf{w}$. For this purpose, we can omit the last two terms on the right-hand side of (2.66) because they do not depend on $\mathbf{w}$. Also, note that scaling the log likelihood by a positive constant coefficient does not alter the location of the maximum with respect to $\mathbf{w}$, and so we can replace the coefficient $1/2\sigma^2$ with $1/2$. Finally, instead of maximizing the log likelihood, we can equivalently minimize the negative log likelihood. We therefore see that maximizing the likelihood is equivalent, so far as determining $\mathbf{w}$ is concerned, to minimizing the *sum-of-squares error function* defined by

$$E(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N}\{y(x_n, \mathbf{w}) - t_n\}^2. \tag{2.67}$$

Thus, the sum-of-squares error function has arisen as a consequence of maximizing the likelihood under the assumption of a Gaussian noise distribution.

Telegram: @uni_k

www.konkur.in

Exercise 2.18

We can also use maximum likelihood to determine the variance parameter σ^2. Maximizing (2.66) with respect to σ^2 gives

$$\sigma^2_{\mathrm{ML}} = \frac{1}{N}\sum_{n=1}^{N}\{y(x_n, \mathbf{w}_{\mathrm{ML}}) - t_n\}^2 . \tag{2.68}$$

Note that we can first determine the parameter vector $\mathbf{w}_{\mathrm{ML}}$ governing the mean, and subsequently use this to find the variance σ^2_{ML} as was the case for the simple Gaussian distribution.

Having determined the parameters $\mathbf{w}$ and σ^2, we can now make predictions for new values of x. Because we now have a probabilistic model, these are expressed in terms of the *predictive distribution* that gives the probability distribution over t, rather than simply a point estimate, and is obtained by substituting the maximum likelihood parameters into (2.64) to give

$$p(t|x, \mathbf{w}_{\mathrm{ML}}, \sigma^2_{\mathrm{ML}}) = \mathcal{N}\left(t|y(x, \mathbf{w}_{\mathrm{ML}}), \sigma^2_{\mathrm{ML}}\right) . \tag{2.69}$$

2.4. Transformation of Densities

Chapter 18

We turn now to a discussion of how a probability density transforms under a nonlinear change of variable. This property will play a crucial role when we discuss a class of generative models called *normalizing flows*. It also highlights that a probability density has a different behaviour than a simple function under such transformations.

Consider a single variable x and suppose we make a change of variables $x = g(y)$, then a function $f(x)$ becomes a new function $\widetilde{f}(y)$ defined by

$$\widetilde{f}(y) = f(g(y)). \tag{2.70}$$

Now consider a probability density $p_x(x)$, and again change variables using $x = g(y)$, giving rise to a density $p_y(y)$ with respect to the new variable y, where the suffixes denote that $p_x(x)$ and $p_y(y)$ are different densities. Observations falling in the range $(x, x + \delta x)$ will, for small values of δx, be transformed into the range $(y, y + \delta y)$, where $x = g(y)$, and $p_x(x)\delta x \simeq p_y(y)\delta y$. Hence, if we take the limit $\delta x \to 0$, we obtain

$$\begin{aligned} p_y(y) &= p_x(x)\left|\frac{\mathrm{d}x}{\mathrm{d}y}\right| \\ &= p_x(g(y))\left|\frac{\mathrm{d}g}{\mathrm{d}y}\right| . \end{aligned} \tag{2.71}$$

Here the modulus $|\cdot|$ arises because the derivative $\mathrm{d}y/\,\mathrm{d}x$ could be negative, whereas the density is scaled by the ratio of lengths, which is always positive.

Telegram: @uni_k

www.konkur.in

This procedure for transforming densities can be very powerful. Any density $p(y)$ can be obtained from a fixed density $q(x)$ that is everywhere non-zero by making a nonlinear change of variable $y = f(x)$ in which $f(x)$ is a monotonic function so that $0 \leqslant f'(x) < \infty$.

Exercise 2.19

One consequence of the transformation property (2.71) is that the concept of the maximum of a probability density is dependent on the choice of variable. Suppose $f(x)$ has a mode (i.e., a maximum) at $\widehat{x}$ so that $f'(\widehat{x}) = 0$. The corresponding mode of $\widetilde{f}(y)$ will occur for a value $\widehat{y}$ obtained by differentiating both sides of (2.70) with respect to y:

$$\widetilde{f}'(\widehat{y}) = f'(g(\widehat{y}))g'(\widehat{y}) = 0. \tag{2.72}$$

Assuming $g'(\widehat{y}) \neq 0$ at the mode, then $f'(g(\widehat{y})) = 0$. However, we know that $f'(\widehat{x}) = 0$, and so we see that the locations of the mode expressed in terms of each of the variables x and y are related by $\widehat{x} = g(\widehat{y})$, as one would expect. Thus, finding a mode with respect to the variable x is equivalent to first transforming to the variable y, then finding a mode with respect to y, and then transforming back to x.

Now consider the behaviour of a probability density $p_x(x)$ under the change of variables $x = g(y)$, where the density with respect to the new variable is $p_y(y)$ and is given by (2.71). To deal with the modulus in (2.71) we can write $g'(y) = s|g'(y)|$ where $s \in \{-1, +1\}$. Then (2.71) can be written as

$$p_y(y) = p_x(g(y))sg'(y)$$

where we have used $1/s = s$. Differentiating both sides with respect to y then gives

$$p'_y(y) = sp'_x(g(y))\{g'(y)\}^2 + sp_x(g(y))g''(y). \tag{2.73}$$

Due to the presence of the second term on the right-hand side of (2.73), the relationship $\widehat{x} = g(\widehat{y})$ no longer holds. Thus, the value of x obtained by maximizing $p_x(x)$ will not be the value obtained by transforming to $p_y(y)$ then maximizing with respect to y and then transforming back to x. This causes modes of densities to be dependent on the choice of variables. However, for a linear transformation, the second term on the right-hand side of (2.73) vanishes, and so in this case the location of the maximum transforms according to $\widehat{x} = g(\widehat{y})$.

This effect can be illustrated with a simple example, as shown in Figure 2.12. We begin by considering a Gaussian distribution $p_x(x)$ over x shown by the red curve in Figure 2.12. Next we draw a sample of $N = 50{,}000$ points from this distribution and plot a histogram of their values, which as expected agrees with the distribution $p_x(x)$. Now consider a nonlinear change of variables from x to y given by

$$x = g(y) = \ln(y) - \ln(1 - y) + 5. \tag{2.74}$$

The inverse of this function is given by

$$y = g^{-1}(x) = \frac{1}{1 + \exp(-x + 5)}, \tag{2.75}$$

which is a *logistic sigmoid* function and is shown in Figure 2.12 by the blue curve.

Telegram: @uni_k

www.konkur.in

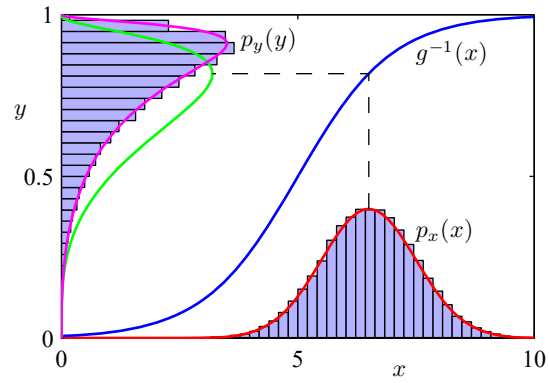


Figure 2.12 Example of the transformation of the mode of a density under a nonlinear change of variables, illustrating the different behaviour compared to a simple function.

If we simply transform $p_x(x)$ as a function of x we obtain the green curve $p_x(g(y))$ shown in Figure 2.12, and we see that the mode of the density $p_x(x)$ is transformed via the sigmoid function to the mode of this curve. However, the density over y transforms instead according to (2.71) and is shown by the magenta curve on the left side of the diagram. Note that this has its mode shifted relative to the mode of the green curve.

To confirm this result, we take our sample of $50{,}000$ values of x, evaluate the corresponding values of y using (2.75), and then plot a histogram of their values. We see that this histogram matches the magenta curve in Figure 2.12 and not the green curve.

2.4.1 Multivariate distributions

We can extend the result (2.71) to densities defined over multiple variables. Consider a density $p(\mathbf{x})$ over a D-dimensional variable $\mathbf{x} = (x_1, \ldots, x_D)^{\mathrm{T}}$, and suppose we transform to a new variable $\mathbf{y} = (y_1, \ldots, y_D)^{\mathrm{T}}$ where $\mathbf{x} = \mathbf{g}(\mathbf{y})$. Here we will limit ourselves to the case where $\mathbf{x}$ and $\mathbf{y}$ have the same dimensionality. The transformed density is then given by the generalization of (2.71) in the form

$$p_{\mathbf{y}}(\mathbf{y}) = p_{\mathbf{x}}(\mathbf{x})\,|\det \mathbf{J}| \tag{2.76}$$

where $\mathbf{J}$ is the *Jacobian matrix* whose elements are given by the partial derivatives $J_{ij} = \partial g_i / \partial y_j$, so that

$$\mathbf{J} = \begin{bmatrix} \dfrac{\partial g_1}{\partial y_1} & \cdots & \dfrac{\partial g_1}{\partial y_D} \\ \vdots & \ddots & \vdots \\ \dfrac{\partial g_D}{\partial y_1} & \cdots & \dfrac{\partial g_D}{\partial y_D} \end{bmatrix}. \tag{2.77}$$

Intuitively, we can view the change of variables as expanding some regions of space and contracting others, with an infinitesimal region $\Delta\mathbf{x}$ around a point $\mathbf{x}$ being transformed to a region $\Delta\mathbf{y}$ around the point $\mathbf{y} = \mathbf{g}(\mathbf{x})$. The absolute value of the determinant of the Jacobian represents the ratio of these volumes and is the same factor

Telegram: @uni_k

www.konkur.in

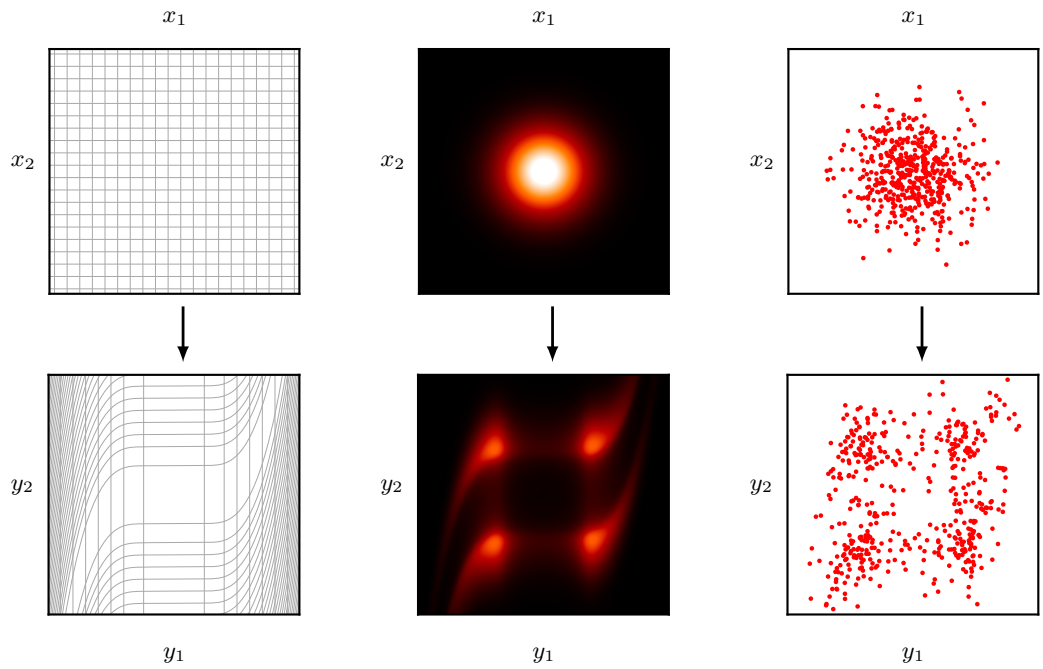


Figure 2.13 Illustration of the effect of a change of variables on a probability distribution in two dimensions. The left column shows the transforming of the variables whereas the middle and right columns show the corresponding effects on a Gaussian distribution and on samples from that distribution, respectively.

that arises when changing variables within an integral. The formula (2.77) follows from the fact that the probability mass in region $\Delta\mathbf{x}$ is the same as the probability mass in $\Delta\mathbf{y}$. Once again, we take the modulus to ensure that the density is non-negative.

We can illustrate this by applying a change of variables to a Gaussian distribution in two dimensions, as shown in the top row in Figure 2.13. Here the transformation from $\mathbf{x}$ to $\mathbf{y}$ is given by

Exercise 2.20

$$y_1 = x_1 + \tanh(5x_1) \tag{2.78}$$

$$y_2 = x_2 + \tanh(5x_2) + \frac{x_1^3}{3}. \tag{2.79}$$

Also shown on the bottom row are samples from a Gaussian distribution in $\mathbf{x}$-space along with the corresponding transformed samples in $\mathbf{y}$-space.

Telegram: @uni_k

www.konkur.in

2.5. Information Theory

Probability theory forms the basis for another important framework called *information theory*, which quantifies the information present in a data set and which plays an important role in machine learning. Here we give a brief introduction to some of the key elements of information theory that we will need later in the book, including the important concept of entropy in its various forms. For a more comprehensive introduction to information theory, with connections to machine learning, see MacKay (2003).

2.5.1 Entropy

We begin by considering a discrete random variable x and we ask how much information is received when we observe a specific value for this variable. The amount of information can be viewed as the 'degree of surprise' on learning the value of x. If we are told that a highly improbable event has just occurred, we will have received more information than if we were told that some very likely event has just occurred, and if we knew that the event was certain to happen, we would receive no information. Our measure of information content will therefore depend on the probability distribution $p(x)$, and so we look for a quantity $h(x)$ that is a monotonic function of the probability $p(x)$ and that expresses the information content. The form of $h(\cdot)$ can be found by noting that if we have two events x and y that are unrelated, then the information gained from observing both of them should be the sum of the information gained from each of them separately, so that $h(x, y) = h(x) + h(y)$. Two unrelated events are statistically independent and so $p(x, y) = p(x)p(y)$. From these two relationships, it is easily shown that $h(x)$ must be given by the logarithm of $p(x)$ and so we have

Exercise 2.21

$$h(x) = -\log_2 p(x) \tag{2.80}$$

where the negative sign ensures that information is positive or zero. Note that low probability events x correspond to high information content. The choice of base for the logarithm is arbitrary, and for the moment we will adopt the convention prevalent in information theory of using logarithms to the base of 2. In this case, as we will see shortly, the units of $h(x)$ are bits ('binary digits').

Now suppose that a sender wishes to transmit the value of a random variable to a receiver. The average amount of information that they transmit in the process is obtained by taking the expectation of (2.80) with respect to the distribution $p(x)$ and is given by

$$\mathrm{H}[x] = -\sum_x p(x) \log_2 p(x). \tag{2.81}$$

This important quantity is called the *entropy* of the random variable x. Note that $\lim_{\epsilon \to 0} (\epsilon \ln \epsilon) = 0$ and so we will take $p(x) \ln p(x) = 0$ whenever we encounter a value for x such that $p(x) = 0$.

So far, we have given a rather heuristic motivation for the definition of information (2.80) and the corresponding entropy (2.81). We now show that these definitions

Telegram: @uni_k

www.konkur.in

indeed possess useful properties. Consider a random variable x having eight possible states, each of which is equally likely. To communicate the value of x to a receiver, we would need to transmit a message of length 3 bits. Notice that the entropy of this variable is given by

$$\mathrm{H}[x] = -8 \times \frac{1}{8} \log_2 \frac{1}{8} = 3 \text{ bits}.$$

Now consider an example (Cover and Thomas, 1991) of a variable having eight possible states $\{a, b, c, d, e, f, g, h\}$ for which the respective probabilities are given by $(\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{16}, \frac{1}{64}, \frac{1}{64}, \frac{1}{64}, \frac{1}{64})$. The entropy in this case is given by

$$\mathrm{H}[x] = -\frac{1}{2}\log_2\frac{1}{2} - \frac{1}{4}\log_2\frac{1}{4} - \frac{1}{8}\log_2\frac{1}{8} - \frac{1}{16}\log_2\frac{1}{16} - \frac{4}{64}\log_2\frac{1}{64} = 2 \text{ bits}.$$

We see that the nonuniform distribution has a smaller entropy than the uniform one, and we will gain some insight into this shortly when we discuss the interpretation of entropy in terms of disorder. For the moment, let us consider how we would transmit the identity of the variable's state to a receiver. We could do this, as before, using a 3-bit number. However, we can take advantage of the nonuniform distribution by using shorter codes for the more probable events, at the expense of longer codes for the less probable events, in the hope of getting a shorter average code length. This can be done by representing the states $\{a, b, c, d, e, f, g, h\}$ using, for instance, the following set of code strings: 0, 10, 110, 1110, 111100, 111101, 111110, and 111111. The average length of the code that has to be transmitted is then

$$\text{average code length} = \frac{1}{2} \times 1 + \frac{1}{4} \times 2 + \frac{1}{8} \times 3 + \frac{1}{16} \times 4 + 4 \times \frac{1}{64} \times 6 = 2 \text{ bits},$$

which again is the same as the entropy of the random variable. Note that shorter code strings cannot be used because it must be possible to disambiguate a concatenation of such strings into its component parts. For instance, 11001110 decodes uniquely into the state sequence c, a, d. This relation between entropy and shortest coding length is a general one. The *noiseless coding theorem* (Shannon, 1948) states that the entropy is a lower bound on the number of bits needed to transmit the state of a random variable.

From now on, we will switch to the use of natural logarithms in defining entropy, as this will provide a more convenient link with ideas elsewhere in this book. In this case, the entropy is measured in units of *nats* (from 'natural logarithm') instead of bits, which differ simply by a factor of $\ln 2$.

2.5.2 Physics perspective

We have introduced the concept of entropy in terms of the average amount of information needed to specify the state of a random variable. In fact, the concept of entropy has much earlier origins in physics where it was introduced in the context of equilibrium thermodynamics and later given a deeper interpretation as a measure of disorder through developments in statistical mechanics. We can understand this alternative view of entropy by considering a set of N identical objects that are to be divided amongst a set of bins, such that there are n_i objects in the ith bin. Consider

Telegram: @uni_k

www.konkur.in

the number of different ways of allocating the objects to the bins. There are N ways to choose the first object, $(N-1)$ ways to choose the second object, and so on, leading to a total of $N!$ ways to allocate all N objects to the bins, where $N!$ (pronounced 'N factorial') denotes the product $N \times (N-1) \times \cdots \times 2 \times 1$. However, we do not wish to distinguish between rearrangements of objects within each bin. In the ith bin there are $n_i!$ ways of reordering the objects, and so the total number of ways of allocating the N objects to the bins is given by

$$W = \frac{N!}{\prod_i n_i!}, \tag{2.82}$$

which is called the *multiplicity*. The entropy is then defined as the logarithm of the multiplicity scaled by a constant factor $1/N$ so that

$$\mathrm{H} = \frac{1}{N} \ln W = \frac{1}{N} \ln N! - \frac{1}{N} \sum_i \ln n_i!. \tag{2.83}$$

We now consider the limit $N \to \infty$, in which the fractions n_i/N are held fixed, and apply Stirling's approximation:

$$\ln N! \simeq N \ln N - N, \tag{2.84}$$

which gives

$$\mathrm{H} = -\lim_{N\to\infty} \sum_i \left(\frac{n_i}{N}\right) \ln \left(\frac{n_i}{N}\right) = -\sum_i p_i \ln p_i \tag{2.85}$$

where we have used $\sum_i n_i = N$. Here $p_i = \lim_{N\to\infty}(n_i/N)$ is the probability of an object being assigned to the ith bin. In physics terminology, the specific allocation of objects into bins is called a *microstate*, and the overall distribution of occupation numbers, expressed through the ratios n_i/N, is called a *macrostate*. The multiplicity W, which expresses the number of microstates in a given macrostate, is also known as the *weight* of the macrostate.

We can interpret the bins as the states x_i of a discrete random variable X, where $p(X = x_i) = p_i$. The entropy of the random variable X is then

$$\mathrm{H}[p] = -\sum_i p(x_i) \ln p(x_i). \tag{2.86}$$

Distributions $p(x_i)$ that are sharply peaked around a few values will have a relatively low entropy, whereas those that are spread more evenly across many values will have higher entropy, as illustrated in Figure 2.14.

Because $0 \leqslant p_i \leqslant 1$, the entropy is non-negative, and it will equal its minimum value of 0 when one of the $p_i = 1$ and all other $p_{j\neq i} = 0$. The maximum entropy configuration can be found by maximizing H using a Lagrange multiplier to enforce the normalization constraint on the probabilities. Thus, we maximize

Appendix C

$$\widetilde{\mathrm{H}} = -\sum_i p(x_i) \ln p(x_i) + \lambda \left(\sum_i p(x_i) - 1 \right) \tag{2.87}$$

Telegram: @uni_k

www.konkur.in

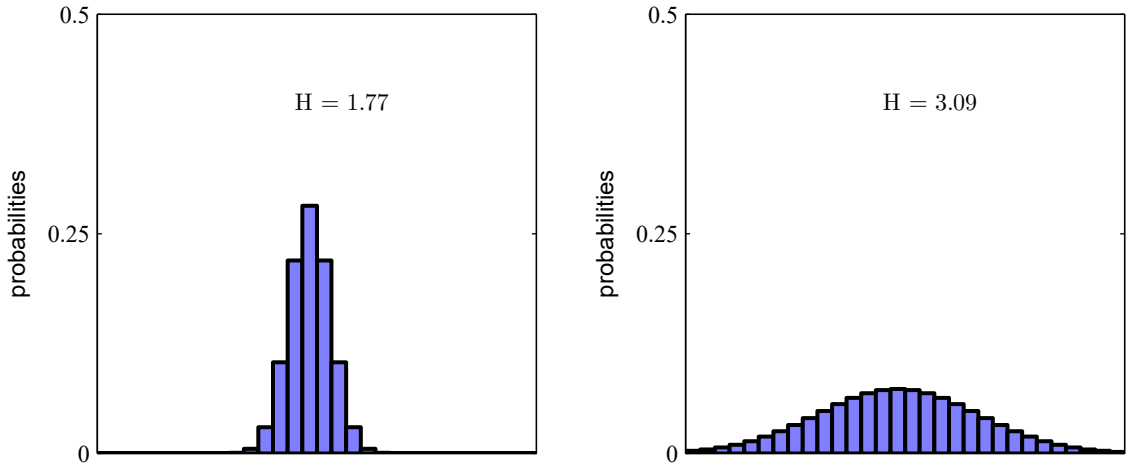


Figure 2.14 Histograms of two probability distributions over 30 bins illustrating the higher value of the entropy H for the broader distribution. The largest entropy would arise from a uniform distribution which would give $\mathrm{H} = -\ln(1/30) = 3.40$.

Exercise 2.22
Exercise 2.23

from which we find that all of the $p(x_i)$ are equal and are given by $p(x_i) = 1/M$ where M is the total number of states x_i. The corresponding value of the entropy is then $\mathrm{H} = \ln M$. This result can also be derived from Jensen's inequality (to be discussed shortly). To verify that the stationary point is indeed a maximum, we can evaluate the second derivative of the entropy, which gives

$$\frac{\partial \widetilde{\mathrm{H}}}{\partial p(x_i)\partial p(x_j)} = -I_{ij}\frac{1}{p_i} \tag{2.88}$$

where I_{ij} are the elements of the identity matrix. We see that these values are all negative and, hence, the stationary point is indeed a maximum.

2.5.3 Differential entropy

We can extend the definition of entropy to include distributions $p(x)$ over continuous variables x as follows. First divide x into bins of width Δ. Then, assuming that $p(x)$ is continuous, the *mean value theorem* (Weisstein, 1999) tells us that, for each such bin, there must exist a value x_i in the range $i\Delta \leqslant x_i \leqslant (i+1)\Delta$ such that

$$\int_{i\Delta}^{(i+1)\Delta} p(x)\,\mathrm{d}x = p(x_i)\Delta. \tag{2.89}$$

We can now quantize the continuous variable x by assigning any value x to the value x_i whenever x falls in the ith bin. The probability of observing the value x_i is then

Telegram: @uni_k

www.konkur.in

$p(x_i)\Delta$. This gives a discrete distribution for which the entropy takes the form

$$\mathrm{H}_\Delta = -\sum_i p(x_i)\Delta \ln\left(p(x_i)\Delta\right) = -\sum_i p(x_i)\Delta \ln p(x_i) - \ln \Delta \tag{2.90}$$

where we have used $\sum_i p(x_i)\Delta = 1$, which follows from (2.89) and (2.25). We now omit the second term $-\ln\Delta$ on the right-hand side of (2.90), since it is independent of $p(x)$, and then consider the limit $\Delta \to 0$. The first term on the right-hand side of (2.90) will approach the integral of $p(x)\ln p(x)$ in this limit so that

$$\lim_{\Delta\to 0}\left\{-\sum_i p(x_i)\Delta \ln p(x_i)\right\} = -\int p(x)\ln p(x)\,\mathrm{d}x \tag{2.91}$$

where the quantity on the right-hand side is called the *differential entropy*. We see that the discrete and continuous forms of the entropy differ by a quantity $\ln\Delta$, which diverges in the limit $\Delta \to 0$. This reflects that specifying a continuous variable very precisely requires a large number of bits. For a density defined over multiple continuous variables, denoted collectively by the vector $\mathbf{x}$, the differential entropy is given by

$$\mathrm{H}[\mathbf{x}] = -\int p(\mathbf{x})\ln p(\mathbf{x})\,\mathrm{d}\mathbf{x}. \tag{2.92}$$

2.5.4 Maximum entropy

We saw for discrete distributions that the maximum entropy configuration corresponds to a uniform distribution of probabilities across the possible states of the variable. Let us now consider the corresponding result for a continuous variable. If this maximum is to be well defined, it will be necessary to constrain the first and second moments of $p(x)$ and to preserve the normalization constraint. We therefore maximize the differential entropy with the three constraints:

$$\int_{-\infty}^{\infty} p(x)\,\mathrm{d}x = 1 \tag{2.93}$$

$$\int_{-\infty}^{\infty} xp(x)\,\mathrm{d}x = \mu \tag{2.94}$$

$$\int_{-\infty}^{\infty} (x-\mu)^2 p(x)\,\mathrm{d}x = \sigma^2. \tag{2.95}$$

Appendix C The constrained maximization can be performed using Lagrange multipliers so that we maximize the following functional with respect to $p(x)$:

$$-\int_{-\infty}^{\infty} p(x)\ln p(x)\,\mathrm{d}x + \lambda_1\left(\int_{-\infty}^{\infty} p(x)\,\mathrm{d}x - 1\right) + \lambda_2\left(\int_{-\infty}^{\infty} xp(x)\,\mathrm{d}x - \mu\right) + \lambda_3\left(\int_{-\infty}^{\infty} (x-\mu)^2 p(x)\,\mathrm{d}x - \sigma^2\right). \tag{2.96}$$

Telegram: @uni_k

www.konkur.in

Appendix B Using the calculus of variations, we set the derivative of this functional to zero giving

$$p(x) = \exp\left\{-1 + \lambda_1 + \lambda_2 x + \lambda_3 (x-\mu)^2\right\}. \tag{2.97}$$

Exercise 2.24 The Lagrange multipliers can be found by back-substitution of this result into the three constraint equations, leading finally to the result:

$$p(x) = \frac{1}{(2\pi\sigma^2)^{1/2}} \exp\left\{-\frac{(x-\mu)^2}{2\sigma^2}\right\}, \tag{2.98}$$

and so the distribution that maximizes the differential entropy is the Gaussian. Note that we did not constrain the distribution to be non-negative when we maximized the entropy. However, because the resulting distribution is indeed non-negative, we see with hindsight that such a constraint is not necessary.

Exercise 2.25 If we evaluate the differential entropy of the Gaussian, we obtain

$$\mathrm{H}[x] = \frac{1}{2}\left\{1 + \ln(2\pi\sigma^2)\right\}. \tag{2.99}$$

Thus, we see again that the entropy increases as the distribution becomes broader, i.e., as σ^2 increases. This result also shows that the differential entropy, unlike the discrete entropy, can be negative, because $\mathrm{H}(x) < 0$ in (2.99) for $\sigma^2 < 1/(2\pi e)$.

2.5.5 Kullback–Leibler divergence

So far in this section, we have introduced a number of concepts from information theory, including the key notion of entropy. We now start to relate these ideas to machine learning. Consider some unknown distribution $p(\mathbf{x})$, and suppose that we have modelled this using an approximating distribution $q(\mathbf{x})$. If we use $q(\mathbf{x})$ to construct a coding scheme for transmitting values of $\mathbf{x}$ to a receiver, then the average *additional* amount of information (in nats) required to specify the value of $\mathbf{x}$ (assuming we choose an efficient coding scheme) as a result of using $q(\mathbf{x})$ instead of the true distribution $p(\mathbf{x})$ is given by

$$\begin{aligned}\mathrm{KL}(p\|q) &= -\int p(\mathbf{x}) \ln q(\mathbf{x})\,\mathrm{d}\mathbf{x} - \left(-\int p(\mathbf{x}) \ln p(\mathbf{x})\,\mathrm{d}\mathbf{x}\right) \\ &= -\int p(\mathbf{x}) \ln\left\{\frac{q(\mathbf{x})}{p(\mathbf{x})}\right\}\mathrm{d}\mathbf{x}. \end{aligned} \tag{2.100}$$

This is known as the *relative entropy* or *Kullback–Leibler divergence*, or *KL divergence* (Kullback and Leibler, 1951), between the distributions $p(\mathbf{x})$ and $q(\mathbf{x})$. Note that it is not a symmetrical quantity, that is to say $\mathrm{KL}(p\|q) \not\equiv \mathrm{KL}(q\|p)$.

We now show that the Kullback–Leibler divergence satisfies $\mathrm{KL}(p\|q) \geqslant 0$ with equality if, and only if, $p(\mathbf{x}) = q(\mathbf{x})$. To do this we first introduce the concept of *convex* functions. A function $f(x)$ is said to be convex if it has the property that every chord lies on or above the function, as shown in Figure 2.15.

Any value of x in the interval from $x = a$ to $x = b$ can be written in the form $\lambda a + (1-\lambda)b$ where $0 \leqslant \lambda \leqslant 1$. The corresponding point on the chord

Telegram: @uni_k

www.konkur.in

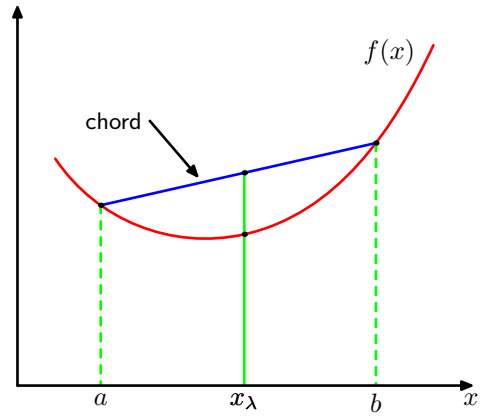


Figure 2.15 A convex function $f(x)$ is one for which every chord (shown in blue) lies on or above the function (shown in red).

is given by $\lambda f(a) + (1-\lambda)f(b)$, and the corresponding value of the function is $f(\lambda a + (1-\lambda)b)$. Convexity then implies

$$f(\lambda a + (1-\lambda)b) \leqslant \lambda f(a) + (1-\lambda)f(b). \tag{2.101}$$

Exercise 2.32 This is equivalent to the requirement that the second derivative of the function be everywhere positive. Examples of convex functions are $x \ln x$ (for $x > 0$) and x^2. A function is called *strictly convex* if the equality is satisfied only for $\lambda = 0$ and $\lambda = 1$. If a function has the opposite property, namely that every chord lies on or below the function, it is called *concave*, with a corresponding definition for *strictly concave*. If a function $f(x)$ is convex, then $-f(x)$ will be concave.

Exercise 2.33 Using the technique of proof by induction, we can show from (2.101) that a convex function $f(x)$ satisfies

$$f\left(\sum_{i=1}^{M} \lambda_i x_i\right) \leqslant \sum_{i=1}^{M} \lambda_i f(x_i) \tag{2.102}$$

where $\lambda_i \geqslant 0$ and $\sum_i \lambda_i = 1$, for any set of points $\{x_i\}$. The result (2.102) is known as *Jensen's inequality*. If we interpret the λ_i as the probability distribution over a discrete variable x taking the values $\{x_i\}$, then (2.102) can be written

$$f\left(\mathbb{E}[x]\right) \leqslant \mathbb{E}[f(x)] \tag{2.103}$$

where $\mathbb{E}[\cdot]$ denotes the expectation. For continuous variables, Jensen's inequality takes the form

$$f\left(\int \mathbf{x} p(\mathbf{x})\, \mathrm{d}\mathbf{x}\right) \leqslant \int f(\mathbf{x}) p(\mathbf{x})\, \mathrm{d}\mathbf{x}. \tag{2.104}$$

We can apply Jensen's inequality in the form (2.104) to the Kullback–Leibler divergence (2.100) to give

$$\mathrm{KL}(p\|q) = -\int p(\mathbf{x}) \ln \left\{\frac{q(\mathbf{x})}{p(\mathbf{x})}\right\} \mathrm{d}\mathbf{x} \geqslant -\ln \int q(\mathbf{x})\, \mathrm{d}\mathbf{x} = 0 \tag{2.105}$$

Telegram: @uni_k

www.konkur.in

where we have used $-\ln x$ is a convex function, together with the normalization condition $\int q(\mathbf{x})\,\mathrm{d}\mathbf{x} = 1$. In fact, $-\ln x$ is a strictly convex function, so the equality will hold if, and only if, $q(\mathbf{x}) = p(\mathbf{x})$ for all $\mathbf{x}$. Thus, we can interpret the Kullback–Leibler divergence as a measure of the dissimilarity of the two distributions $p(\mathbf{x})$ and $q(\mathbf{x})$.

We see that there is an intimate relationship between data compression and density estimation (i.e., the problem of modelling an unknown probability distribution) because the most efficient compression is achieved when we know the true distribution. If we use a distribution that is different from the true one, then we must necessarily have a less efficient coding, and on average the additional information that must be transmitted is (at least) equal to the Kullback–Leibler divergence between the two distributions.

Suppose that data is being generated from an unknown distribution $p(\mathbf{x})$ that we wish to model. We can try to approximate this distribution using some parametric distribution $q(\mathbf{x}|\boldsymbol{\theta})$, governed by a set of adjustable parameters $\boldsymbol{\theta}$. One way to determine $\boldsymbol{\theta}$ is to minimize the Kullback–Leibler divergence between $p(\mathbf{x})$ and $q(\mathbf{x}|\boldsymbol{\theta})$ with respect to $\boldsymbol{\theta}$. We cannot do this directly because we do not know $p(\mathbf{x})$. Suppose, however, that we have observed a finite set of training points $\mathbf{x}_n$, for $n = 1, \ldots, N$, drawn from $p(\mathbf{x})$. Then the expectation with respect to $p(\mathbf{x})$ can be approximated by a finite sum over these points, using (2.40), so that

$$\mathrm{KL}(p\|q) \simeq \frac{1}{N}\sum_{n=1}^{N}\left\{-\ln q(\mathbf{x}_n|\boldsymbol{\theta}) + \ln p(\mathbf{x}_n)\right\}. \tag{2.106}$$

The second term on the right-hand side of (2.106) is independent of $\boldsymbol{\theta}$, and the first term is the negative log likelihood function for $\boldsymbol{\theta}$ under the distribution $q(\mathbf{x}|\boldsymbol{\theta})$ evaluated using the training set. Thus, we see that minimizing this Kullback–Leibler divergence is equivalent to maximizing the log likelihood function.

Exercise 2.34

2.5.6 Conditional entropy

Now consider the joint distribution between two sets of variables $\mathbf{x}$ and $\mathbf{y}$ given by $p(\mathbf{x}, \mathbf{y})$ from which we draw pairs of values of $\mathbf{x}$ and $\mathbf{y}$. If a value of $\mathbf{x}$ is already known, then the additional information needed to specify the corresponding value of $\mathbf{y}$ is given by $-\ln p(\mathbf{y}|\mathbf{x})$. Thus the average additional information needed to specify $\mathbf{y}$ can be written as

$$\mathrm{H}[\mathbf{y}|\mathbf{x}] = -\iint p(\mathbf{y}, \mathbf{x}) \ln p(\mathbf{y}|\mathbf{x})\,\mathrm{d}\mathbf{y}\,\mathrm{d}\mathbf{x}, \tag{2.107}$$

which is called the *conditional entropy* of $\mathbf{y}$ given $\mathbf{x}$. It is easily seen, using the product rule, that the conditional entropy satisfies the relation:

Exercise 2.35

$$\mathrm{H}[\mathbf{x}, \mathbf{y}] = \mathrm{H}[\mathbf{y}|\mathbf{x}] + \mathrm{H}[\mathbf{x}] \tag{2.108}$$

where $\mathrm{H}[\mathbf{x}, \mathbf{y}]$ is the differential entropy of $p(\mathbf{x}, \mathbf{y})$ and $\mathrm{H}[\mathbf{x}]$ is the differential entropy of the marginal distribution $p(\mathbf{x})$. Thus, the information needed to describe $\mathbf{x}$ and $\mathbf{y}$ is given by the sum of the information needed to describe $\mathbf{x}$ alone plus the additional information required to specify $\mathbf{y}$ given $\mathbf{x}$.

Telegram: @uni_k

www.konkur.in

2.5.7 Mutual information

When two variables $\mathbf{x}$ and $\mathbf{y}$ are independent, their joint distribution will factorize into the product of their marginals $p(\mathbf{x}, \mathbf{y}) = p(\mathbf{x})p(\mathbf{y})$. If the variables are not independent, we can gain some idea of whether they are 'close' to being independent by considering the Kullback–Leibler divergence between the joint distribution and the product of the marginals, given by

$$\begin{aligned} \mathrm{I}[\mathbf{x}, \mathbf{y}] &\equiv \mathrm{KL}(p(\mathbf{x}, \mathbf{y}) \| p(\mathbf{x})p(\mathbf{y})) \\ &= -\iint p(\mathbf{x}, \mathbf{y}) \ln \left(\frac{p(\mathbf{x})p(\mathbf{y})}{p(\mathbf{x}, \mathbf{y})} \right) \mathrm{d}\mathbf{x}\, \mathrm{d}\mathbf{y}, \end{aligned} \tag{2.109}$$

which is called the *mutual information* between the variables $\mathbf{x}$ and $\mathbf{y}$. From the properties of the Kullback–Leibler divergence, we see that $I[\mathbf{x}, \mathbf{y}] \geqslant 0$ with equality if, and only if, $\mathbf{x}$ and $\mathbf{y}$ are independent. Using the sum and product rules of probability, we see that the mutual information is related to the conditional entropy through

Exercise 2.38

$$\mathrm{I}[\mathbf{x}, \mathbf{y}] = \mathrm{H}[\mathbf{x}] - \mathrm{H}[\mathbf{x}|\mathbf{y}] = \mathrm{H}[\mathbf{y}] - \mathrm{H}[\mathbf{y}|\mathbf{x}]. \tag{2.110}$$

Thus, the mutual information represents the reduction in the uncertainty about $\mathbf{x}$ by virtue of being told the value of $\mathbf{y}$ (or vice versa). From a Bayesian perspective, we can view $p(\mathbf{x})$ as the prior distribution for $\mathbf{x}$ and $p(\mathbf{x}|\mathbf{y})$ as the posterior distribution after we have observed new data $\mathbf{y}$. The mutual information therefore represents the reduction in uncertainty about $\mathbf{x}$ as a consequence of the new observation $\mathbf{y}$.

2.6. Bayesian Probabilities

When we considered the bent coin in Figure 2.2, we introduced the concept of probability in terms of the frequencies of random, repeatable events, such as the probability of the coin landing concave side up. We will refer to this as the *classical* or *frequentist* interpretation of probability. We also introduced the more general *Bayesian* view, in which probabilities provide a quantification of uncertainty. In this case, our uncertainty is whether the concave side of the coin is heads or tails.

The use of probability to represent uncertainty is not an ad hoc choice but is inevitable if we are to respect common sense while making rational and coherent inferences. For example, Cox (1946) showed that if numerical values are used to represent degrees of belief, then a simple set of axioms encoding common sense properties of such beliefs leads uniquely to a set of rules for manipulating degrees of belief that are equivalent to the sum and product rules of probability. It is therefore natural to refer to these quantities as (Bayesian) probabilities.

For the bent coin we assumed, in the absence of further information, that the probability of the concave side of the coin being heads is 0.5. Now suppose we are told the results of flipping the coin a few times. Intuitively, it seems that this should provide us with some information as to whether the concave side is heads. For instance, suppose we see many more flips that land tails than land heads. Given

Telegram: @uni_k

www.konkur.in

that the coin is more likely to land concave side up, this provides evidence to suggest that the concave side is more likely to be tails. In fact, this intuition is correct, and furthermore, we can quantify this using the rules of probability. Bayes' theorem now acquires a new significance, because it allows us to convert the prior probability for the concave side being heads into a posterior probability by incorporating the data provided by the coin flips. Moreover, this process is iterative, meaning the posterior probability becomes the prior for incorporating data from further coin flips.

Exercise 2.40

One aspect of the Bayesian viewpoint is that the inclusion of prior knowledge arises naturally. Suppose, for instance, that a fair-looking coin is tossed three times and lands heads each time. The maximum likelihood estimate of the probability of landing heads would give 1, implying that all future tosses will land heads! By contrast, a Bayesian approach with any reasonable prior will lead to a less extreme conclusion.

Section 3.1.2

2.6.1 Model parameters

The Bayesian perspective provides valuable insights into several aspects of machine learning, and we can illustrate these using the sine curve regression example. Here we denote the training data set by $\mathcal{D}$. We have already seen in the context of linear regression that the parameters can be chosen using *maximum likelihood*, in which $\mathbf{w}$ is set to the value that maximizes the likelihood function $p(\mathcal{D}|\mathbf{w})$. This corresponds to choosing the value of $\mathbf{w}$ for which the probability of the observed data set is maximized. In the machine learning literature, the negative log of the likelihood function is called an *error function*. Because the negative logarithm is a monotonically decreasing function, maximizing the likelihood is equivalent to minimizing the error. This leads to a specific choice of parameter values, denoted $\mathbf{w}_{\mathrm{ML}}$, which are then used to make predictions for new data.

Section 1.2

We have seen that different choices of training data set, for example containing different numbers of data points, give rise to different solutions for $\mathbf{w}_{\mathrm{ML}}$. From a Bayesian perspective, we can also use the machinery of probability theory to describe this uncertainty in the model parameters. We can capture our assumptions about $\mathbf{w}$, *before* observing the data, in the form of a prior probability distribution $p(\mathbf{w})$. The effect of the observed data $\mathcal{D}$ is expressed through the likelihood function $p(\mathcal{D}|\mathbf{w})$, and Bayes' theorem now takes the form

$$p(\mathbf{w}|\mathcal{D}) = \frac{p(\mathcal{D}|\mathbf{w})p(\mathbf{w})}{p(\mathcal{D})}, \tag{2.111}$$

which allows us to evaluate the uncertainty in $\mathbf{w}$ *after* we have observed $\mathcal{D}$ in the form of the posterior probability $p(\mathbf{w}|\mathcal{D})$.

It is important to emphasize that the quantity $p(\mathcal{D}|\mathbf{w})$ is called the likelihood function when it is viewed as a function of the parameter vector $\mathbf{w}$, and it expresses how probable the observed data set is for different values of $\mathbf{w}$. Note that the likelihood $p(\mathcal{D}|\mathbf{w})$ is not a probability distribution over $\mathbf{w}$, and its integral with respect to $\mathbf{w}$ does not (necessarily) equal one.

Given this definition of likelihood, we can state Bayes' theorem in words:

$$\text{posterior} \propto \text{likelihood} \times \text{prior} \tag{2.112}$$

Telegram: @uni_k

www.konkur.in

where all of these quantities are viewed as functions of $\mathbf{w}$. The denominator in (2.111) is the normalization constant, which ensures that the posterior distribution on the left-hand side is a valid probability density and integrates to one. Indeed, by integrating both sides of (2.111) with respect to $\mathbf{w}$, we can express the denominator in Bayes' theorem in terms of the prior distribution and the likelihood function:

$$p(\mathcal{D}) = \int p(\mathcal{D}|\mathbf{w})p(\mathbf{w})\,\mathrm{d}\mathbf{w}. \tag{2.113}$$

In both the Bayesian and frequentist paradigms, the likelihood function $p(\mathcal{D}|\mathbf{w})$ plays a central role. However, the manner in which it is used is fundamentally different in the two approaches. In a frequentist setting, $\mathbf{w}$ is considered to be a fixed parameter, whose value is determined by some form of 'estimator', and error bars on this estimate are determined (conceptually, at least) by considering the distribution of possible data sets $\mathcal{D}$. By contrast, from the Bayesian viewpoint there is only a single data set $\mathcal{D}$ (namely the one that is actually observed), and the uncertainty in the parameters is expressed through a probability distribution over $\mathbf{w}$.

2.6.2 Regularization

Section 1.2.5

We can use this Bayesian perspective to gain insight into the technique of regularization that was used in the sine curve regression example to reduce over-fitting. Instead of choosing the model parameters by maximizing the likelihood function with respect to $\mathbf{w}$, we can maximize the posterior probability (2.111). This technique is called the *maximum a posteriori* estimate, or simply *MAP* estimate. Equivalently, we can minimize the negative log of the posterior probability. Taking negative logs of both sides of (2.111), we have

$$-\ln p(\mathbf{w}|\mathcal{D}) = -\ln p(\mathcal{D}|\mathbf{w}) - \ln p(\mathbf{w}) + \ln p(\mathcal{D}). \tag{2.114}$$

The first term on the right-hand side of (2.114) is the usual log likelihood. The third term can be omitted since it does not depend on $\mathbf{w}$. The second term takes the form of a function of $\mathbf{w}$, which is added to the log likelihood, and we can recognize this as a form of regularization. To make this more explicit, suppose we choose the prior distribution $p(\mathbf{w})$ to be the product of independent zero-mean Gaussian distributions for each of the elements of $\mathbf{w}$ such that each has the same variance s^2 so that

$$p(\mathbf{w}|s) = \prod_{i=0}^{M} \mathcal{N}(w_i|0, s^2) = \prod_{i=0}^{M} \left(\frac{1}{2\pi s^2}\right)^{1/2} \exp\left\{-\frac{w_i^2}{2s^2}\right\}. \tag{2.115}$$

Substituting into (2.114), we obtain

$$-\ln p(\mathbf{w}|\mathcal{D}) = -\ln p(\mathcal{D}|\mathbf{w}) + \frac{1}{2s^2}\sum_{i=0}^{M} w_i^2 + \text{const}. \tag{2.116}$$

If we consider the particular case of the linear regression model whose log likelihood is given by (2.66), then we find that maximizing the posterior distribution is equivalent to minimizing the function

Exercise 2.41

Telegram: @uni_k

www.konkur.in

$$E(\mathbf{w}) = \frac{1}{2\sigma^2}\sum_{n=1}^{N}\{y(x_n, \mathbf{w}) - t_n\}^2 + \frac{1}{2s^2}\mathbf{w}^{\mathrm{T}}\mathbf{w}. \tag{2.117}$$

We see that this takes the form of the regularized sum-of-squares error function encountered earlier in the form (1.4).

2.6.3 Bayesian machine learning

The Bayesian perspective has allowed us to motivate the use of regularization and to derive a specific form for the regularization term. However, the use of Bayes' theorem alone does not constitute a truly Bayesian treatment of machine learning since it is still finding a single solution for $\mathbf{w}$ and does not therefore take account of uncertainty in the value of $\mathbf{w}$. Suppose we have a training data set $\mathcal{D}$ and our goal is to predict some target variable t given a new input value x. We are therefore interested in the distribution of t given both x and $\mathcal{D}$. From the sum and product rules of probability, we have

$$p(t|x, \mathcal{D}) = \int p(t|x, \mathbf{w})p(\mathbf{w}|\mathcal{D})\,\mathrm{d}\mathbf{w}. \tag{2.118}$$

We see that the prediction is obtained by taking a weighted average $p(t|x, \mathbf{w})$ over all possible values of $\mathbf{w}$ in which the weighting function is given by the posterior probability distribution $p(\mathbf{w}|\mathcal{D})$. The key difference that distinguishes Bayesian methods is this integration over the space of parameters. By contrast, conventional frequentist methods use point estimates for parameters obtained by optimizing a loss function such as a regularized sum-of-squares.

This fully Bayesian treatment of machine learning offers some powerful insights. For example, the problem of over-fitting, encountered earlier in the context of polynomial regression, is an example of a pathology arising from the use of maximum likelihood, and does not arise when we marginalize over parameters using the Bayesian approach. Similarly, we may have multiple potential models that we could use to solve a given problem, such as polynomials of different orders in the regression example. A maximum likelihood approach simply picks the model that gives the highest probability of the data, but this favours ever more complex models, leading to over-fitting. A fully Bayesian treatment involves averaging over all possible models, with the contribution of each model weighted by its posterior probability. Moreover, this probability is typically highest for models of intermediate complexity. Very simple models (such as polynomials of low order) have low probability as they are unable to fit the data well, whereas very complex models (such as polynomials of very high order) also have low probability because the Bayesian integration over parameters automatically and elegantly penalizes complexity. For a comprehensive overview of Bayesian methods applied to machine learning, including neural networks, see Bishop (2006).

Section 1.2

Section 9.6

Unfortunately, there is a major drawback with the Bayesian framework, and this is apparent in (2.118), which involves integrating over the space of parameters. Modern deep learning models can have millions or billions of parameters and even simple approximations to such integrals are typically infeasible. In fact, given a

Telegram: @uni_k

www.konkur.in

limited compute budget and an ample source of training data, it will often be better to apply maximum likelihood techniques, generally augmented with one or more forms of regularization, to a large neural network rather than apply a Bayesian treatment to a much smaller model.

Exercises

2.1 (⋆) In the cancer screening example, we used a prior probability of cancer of $p(C = 1) = 0.01$. In reality, the prevalence of cancer is generally very much lower. Consider a situation in which $p(C = 1) = 0.001$, and recompute the probability of having cancer given a positive test $p(C = 1|T = 1)$. Intuitively, the result can appear surprising to many people since the test seems to have high accuracy and yet a positive test still leads to a low probability of having cancer.

2.2 (⋆⋆) Deterministic numbers satisfy the property of *transitivity*, so that if $x > y$ and $y > z$ then it follows that $x > z$. When we go to random numbers, however, this property need no longer apply. Figure 2.16 shows a set of four cubical dice that have been arranged in a cyclic order. Show that each of the four dice has a 2/3 probability of rolling a higher number than the previous die in the cycle. Such dice are known as *non-transitive dice*, and the specific examples shown here are called *Efron dice*.

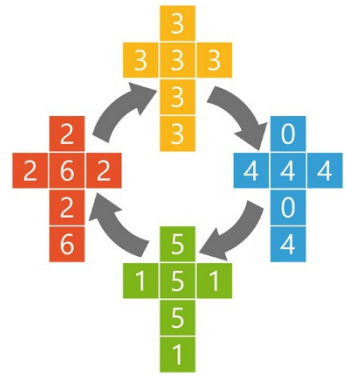


Figure 2.16 An example of non-transitive cubical dice, in which each die has been 'flattened' to reveal the numbers on each of the faces. The dice have been arranged in a cycle, such that each die has a 2/3 probability of rolling a higher number than the previous die in the cycle.

2.3 (⋆) Consider a variable $\mathbf{y}$ given by the sum of two independent random variables $\mathbf{y} = \mathbf{u} + \mathbf{v}$ where $\mathbf{u} \sim p_{\mathbf{u}}(\mathbf{u})$ and $\mathbf{v} \sim p_{\mathbf{v}}(\mathbf{v})$. Show that the distribution $p_{\mathbf{y}}(\mathbf{y})$ is given by

$$p(\mathbf{y}) = \int p_{\mathbf{u}}(\mathbf{u}) p_{\mathbf{v}}(\mathbf{y} - \mathbf{u}) \, \mathrm{d}\mathbf{u}. \tag{2.119}$$

This is known as the *convolution* of $p_{\mathbf{u}}(\mathbf{u})$ and $p_{\mathbf{v}}(\mathbf{v})$.

2.4 (⋆⋆) Verify that the uniform distribution (2.33) is correctly normalized, and find expressions for its mean and variance.

2.5 (⋆⋆) Verify that the exponential distribution (2.34) and the Laplace distribution (2.35) are correctly normalized.

Telegram: @uni_k

www.konkur.in

2.6 (⋆) Using the properties of the Dirac delta function, show that the empirical density (2.37) is correctly normalized.

2.7 (⋆) By making use of the empirical density (2.37), show that the expectation given by (2.39) can be approximated by a sum over a finite set of samples drawn from the density in the form (2.40).

2.8 (⋆) Using the definition (2.44), show that $\mathrm{var}[f(x)]$ satisfies (2.45).

2.9 (⋆) Show that if two variables x and y are independent, then their covariance is zero.

2.10 (⋆) Suppose that the two variables x and z are statistically independent. Show that the mean and variance of their sum satisfies

$$\mathbb{E}[x+z] = \mathbb{E}[x] + \mathbb{E}[z] \tag{2.120}$$

$$\mathrm{var}[x+z] = \mathrm{var}[x] + \mathrm{var}[z]. \tag{2.121}$$

2.11 (⋆) Consider two variables x and y with joint distribution $p(x, y)$. Prove the following two results:

$$\mathbb{E}[x] = \mathbb{E}_y\left[\mathbb{E}_x[x|y]\right] \tag{2.122}$$

$$\mathrm{var}[x] = \mathbb{E}_y\left[\mathrm{var}_x[x|y]\right] + \mathrm{var}_y\left[\mathbb{E}_x[x|y]\right]. \tag{2.123}$$

Here $\mathbb{E}_x[x|y]$ denotes the expectation of x under the conditional distribution $p(x|y)$, with a similar notation for the conditional variance.

2.12 (⋆⋆⋆) In this exercise, we prove the normalization condition (2.51) for the univariate Gaussian. To do this consider, the integral

$$I = \int_{-\infty}^{\infty} \exp\left(-\frac{1}{2\sigma^2}x^2\right) \mathrm{d}x \tag{2.124}$$

which we can evaluate by first writing its square in the form

$$I^2 = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} \exp\left(-\frac{1}{2\sigma^2}x^2 - \frac{1}{2\sigma^2}y^2\right) \mathrm{d}x\,\mathrm{d}y. \tag{2.125}$$

Now make the transformation from Cartesian coordinates (x, y) to polar coordinates (r, θ) and then substitute $u = r^2$. Show that, by performing the integrals over θ and u and then taking the square root of both sides, we obtain

$$I = \left(2\pi\sigma^2\right)^{1/2}. \tag{2.126}$$

Finally, use this result to show that the Gaussian distribution $\mathcal{N}(x|\mu, \sigma^2)$ is normalized.

Telegram: @uni_k

www.konkur.in

2.13 (⋆⋆) By using a change of variables, verify that the univariate Gaussian distribution given by (2.49) satisfies (2.52). Next, by differentiating both sides of the normalization condition

$$\int_{-\infty}^{\infty} \mathcal{N}\left(x|\mu,\sigma^2\right)\,\mathrm{d}x = 1 \tag{2.127}$$

with respect to σ^2, verify that the Gaussian satisfies (2.53). Finally, show that (2.54) holds.

2.14 (⋆) Show that the mode (i.e., the maximum) of the Gaussian distribution (2.49) is given by μ.

2.15 (⋆) By setting the derivatives of the log likelihood function (2.56) with respect to μ and σ^2 equal to zero, verify the results (2.57) and (2.58).

2.16 (⋆⋆) Using the results (2.52) and (2.53), show that

$$\mathbb{E}[x_n x_m] = \mu^2 + I_{nm}\sigma^2 \tag{2.128}$$

where x_n and x_m denote data points sampled from a Gaussian distribution with mean μ and variance σ^2 and I_{nm} satisfies $I_{nm} = 1$ if $n = m$ and $I_{nm} = 0$ otherwise. Hence prove the results (2.59) and (2.60).

2.17 (⋆⋆) Using the definition (2.61), prove the result (2.62) which shows that the expectation of the variance estimator for a Gaussian distribution based on the true mean is given by the true variance σ^2.

2.18 (⋆) Show that maximizing (2.66) with respect to σ^2 gives the result (2.68).

2.19 (⋆⋆) Use the transformation property (2.71) of a probability density under a change of variable to show that any density $p(y)$ can be obtained from a fixed density $q(x)$ that is everywhere non-zero by making a nonlinear change of variable $y = f(x)$ in which $f(x)$ is a monotonic function so that $0 \leqslant f'(x) < \infty$. Write down the differential equation satisfied by $f(x)$ and draw a diagram illustrating the transformation of the density.

2.20 (⋆) Evaluate the elements of the Jacobian matrix for the transformation defined by (2.78) and (2.79).

2.21 (⋆) In Section 2.5, we introduced the idea of entropy $h(x)$ as the information gained on observing the value of a random variable x having distribution $p(x)$. We saw that, for independent variables x and y for which $p(x, y) = p(x)p(y)$, the entropy functions are additive, so that $h(x, y) = h(x) + h(y)$. In this exercise, we derive the relation between h and p in the form of a function $h(p)$. First show that $h(p^2) = 2h(p)$ and, hence, by induction that $h(p^n) = nh(p)$ where n is a positive integer. Hence, show that $h(p^{n/m}) = (n/m)h(p)$ where m is also a positive integer. This implies that $h(p^x) = xh(p)$ where x is a positive rational number and, hence, by continuity when it is a positive real number. Finally, show that this implies $h(p)$ must take the form $h(p) \propto \ln p$.

Telegram: @uni_k

www.konkur.in

2.22 (⋆) Use a Lagrange multiplier to show that maximization of the entropy (2.86) for a discrete variable gives a distribution in which all of the probabilities $p(x_i)$ are equal and that the corresponding value of the entropy is then $\ln M$.

2.23 (⋆) Consider an M-state discrete random variable x, and use Jensen's inequality in the form (2.102) to show that the entropy of its distribution $p(x)$ satisfies $\mathrm{H}[x] \leqslant \ln M$.

2.24 (⋆⋆) Use the calculus of variations to show that the stationary point of the functional (2.96) is given by (2.97). Then use the constraints (2.93), (2.94), and (2.95) to eliminate the Lagrange multipliers and, hence, show that the maximum entropy solution is given by the Gaussian (2.98).

2.25 (⋆) Use the results (2.94) and (2.95) to show that the entropy of the univariate Gaussian (2.98) is given by (2.99).

2.26 (⋆⋆) Suppose that $p(\mathbf{x})$ is some fixed distribution and that we wish to approximate it using a Gaussian distribution $q(\mathbf{x}) = \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma})$. By writing down the form of the Kullback–Leibler divergence $\mathrm{KL}(p\|q)$ for a Gaussian $q(\mathbf{x})$ and then differentiating, show that minimization of $\mathrm{KL}(p\|q)$ with respect to $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$ leads to the result that $\boldsymbol{\mu}$ is given by the expectation of $\mathbf{x}$ under $p(\mathbf{x})$ and that $\boldsymbol{\Sigma}$ is given by the covariance.

2.27 (⋆⋆) Evaluate the Kullback–Leibler divergence (2.100) between the two Gaussians $p(x) = \mathcal{N}(x|\mu, \sigma^2)$ and $q(x) = \mathcal{N}(x|m, s^2)$.

2.28 (⋆⋆) The *alpha family* of divergences is defined by

$$\mathrm{D}_\alpha(p\|q) = \frac{4}{1-\alpha^2}\left(1 - \int p(x)^{(1+\alpha)/2} q(x)^{(1-\alpha)/2}\,\mathrm{d}x\right) \tag{2.129}$$

where $-\infty < \alpha < \infty$ is a continuous parameter. Show that the Kullback–Leibler divergence $\mathrm{KL}(p\|q)$ corresponds to $\alpha \to 1$. This can be done by writing $p^\epsilon = \exp(\epsilon \ln p) = 1 + \epsilon \ln p + O(\epsilon^2)$ and then taking $\epsilon \to 0$. Similarly, show that $\mathrm{KL}(q\|p)$ corresponds to $\alpha \to -1$.

2.29 (⋆⋆) Consider two variables $\mathbf{x}$ and $\mathbf{y}$ having joint distribution $p(\mathbf{x}, \mathbf{y})$. Show that the differential entropy of this pair of variables satisfies

$$\mathrm{H}[\mathbf{x}, \mathbf{y}] \leqslant \mathrm{H}[\mathbf{x}] + \mathrm{H}[\mathbf{y}] \tag{2.130}$$

with equality if, and only if, $\mathbf{x}$ and $\mathbf{y}$ are statistically independent.

2.30 (⋆) Consider a vector $\mathbf{x}$ of continuous variables with distribution $p(\mathbf{x})$ and corresponding entropy $\mathrm{H}[\mathbf{x}]$. Suppose that we make a non-singular linear transformation of $\mathbf{x}$ to obtain a new variable $\mathbf{y} = \mathbf{A}\mathbf{x}$. Show that the corresponding entropy is given by $\mathrm{H}[\mathbf{y}] = \mathrm{H}[\mathbf{x}] + \ln \det \mathbf{A}$ where $\det \mathbf{A}$ denotes the determinant of $\mathbf{A}$.

2.31 (⋆⋆) Suppose that the conditional entropy $\mathrm{H}[y|x]$ between two discrete random variables x and y is zero. Show that, for all values of x such that $p(x) > 0$, the variable y must be a function of x. In other words, for each x there is only one value of y such that $p(y|x) \neq 0$.

Telegram: @uni_k

www.konkur.in

2.32 (⋆) A strictly convex function is defined as one for which every chord lies above the function. Show that this is equivalent to the condition that the second derivative of the function is positive.

2.33 (⋆⋆) Using proof by induction, show that the inequality (2.101) for convex functions implies the result (2.102).

2.34 (⋆) Show that, up to an additive constant, the Kullback–Leibler divergence (2.100) between the empirical distribution (2.37) and a model distribution $q(\mathbf{x}|\boldsymbol{\theta})$ is equal to the negative log likelihood function.

2.35 (⋆) Using the definition (2.107) together with the product rule of probability, prove the result (2.108).

2.36 (⋆⋆⋆) Consider two binary variables x and y having the joint distribution given by

		y = 0	y = 1
x	0	1/3	1/3
	1	0	1/3

Evaluate the following quantities:

(a) $\mathrm{H}[x]$ **(b)** $\mathrm{H}[y]$ **(c)** $\mathrm{H}[y|x]$ **(d)** $\mathrm{H}[x|y]$ **(e)** $\mathrm{H}[x, y]$ **(f)** $\mathrm{I}[x, y]$.

Draw a Venn diagram to show the relationship between these various quantities.

2.37 (⋆) By applying Jensen's inequality (2.102) with $f(x) = \ln x$, show that the arithmetic mean of a set of real numbers is never less than their geometrical mean.

2.38 (⋆) Using the sum and product rules of probability, show that the mutual information $I(\mathbf{x}, \mathbf{y})$ satisfies the relation (2.110).

2.39 (⋆⋆) Suppose that two variables z_1 and z_2 are independent so that $p(z_1, z_2) = p(z_1)p(z_2)$. Show that the covariance matrix between these variables is diagonal. This shows that independence is a sufficient condition for two variables to be uncorrelated. Now consider two variables y_1 and y_2 where y_1 is symmetrically distributed around 0 and $y_2 = y_1^2$. Write down the conditional distribution $p(y_2|y_1)$ and observe that this is dependent on y_1, thus showing that the two variables are not independent. Now show that the covariance matrix between these two variables is again diagonal. To do this, use the relation $p(y_1, y_2) = p(y_1)p(y_2|y_1)$ to show that the off-diagonal terms are zero. This counterexample shows that zero correlation is not a sufficient condition for independence.

2.40 (⋆) Consider the bent coin in Figure 2.2. Assume that the prior probability that the convex side is heads is 0.1. Now suppose the coin is flipped 10 times and we are told that eight of the flips landed heads up and two of the flips landed tails up. Use Bayes' theorem to evaluate the posterior probability that the concave side is heads. Calculate the probability that the next flip will land heads up.

Telegram: @uni_k

www.konkur.in

2.41 (⋆) By substituting (2.115) into (2.114) and making use of the result (2.66) for the log likelihood of the linear regression model, derive the result (2.117) for the regularized error function.

Telegram: @uni_k

www.konkur.in



3 Standard Distributions

In this chapter we discuss some specific examples of probability distributions and their properties. As well as being of interest in their own right, these distributions can form building blocks for more complex models and will be used extensively throughout the book.

One role for the distributions discussed in this chapter is to model the probability distribution $p(\mathbf{x})$ of a random variable $\mathbf{x}$, given a finite set $\mathbf{x}_1, \ldots, \mathbf{x}_N$ of observations. This problem is known as *density estimation*. It should be emphasized that the problem of density estimation is fundamentally ill-posed, because there are infinitely many probability distributions that could have given rise to the observed finite data set. Indeed, any distribution $p(\mathbf{x})$ that is non-zero at each of the data points $\mathbf{x}_1, \ldots, \mathbf{x}_N$ is a potential candidate. The issue of choosing an appropriate distribution relates to the problem of model selection, which has already been encountered in the context of polynomial curve fitting and which is a central issue in machine

Section 1.2

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024
C. M. Bishop, H. Bishop, *Deep Learning*, https://doi.org/10.1007/978-3-031-45468-4_3

Telegram: @uni_k

www.konkur.in

learning.

We begin by considering distributions for discrete variables before exploring the Gaussian distribution for continuous variables. These are specific examples of *parametric* distributions, so called because they are governed by a relatively small number of adjustable parameters, such as the mean and variance of a Gaussian. To apply such models to the problem of density estimation, we need a procedure for determining suitable values for the parameters, given an observed data set, and our main focus will be on maximizing the likelihood function. In this chapter, we will assume that the data observations are independent and identically distributed (i.i.d.), whereas in future chapters we will explore more complex scenarios involving *structured data* where this assumption no longer holds.

One limitation of the parametric approach is that it assumes a specific functional form for the distribution, which may turn out to be inappropriate for a particular application. An alternative approach is given by *nonparametric* density estimation methods in which the form of the distribution typically depends on the size of the data set. Such models still contain parameters, but these control the model complexity rather than the form of the distribution. We end this chapter by briefly considering three nonparametric methods based respectively on histograms, nearest neighbours, and kernels. A major limitation of nonparametric techniques such as these is that they involve storing all the training data. In other words, the number of parameters grows with the size of the data set, so that the method become very inefficient for large data sets. Deep learning combines the efficiency of parametric models with the generality of nonparametric methods by considering flexible distributions based on neural networks having a large, but fixed, number of parameters.

3.1. Discrete Variables

We begin by considering simple distributions for discrete variables, starting with binary variables and then moving on to multi-state variables.

3.1.1 Bernoulli distribution

Consider a single binary random variable $x \in \{0, 1\}$. For example, x might describe the outcome of flipping a coin, with $x = 1$ representing 'heads' and $x = 0$ representing 'tails'. If this were a damaged coin, such as the one shown in Figure 2.2, the probability of landing heads is not necessarily the same as that of landing tails. The probability of $x = 1$ will be denoted by the parameter μ so that

$$p(x = 1|\mu) = \mu \tag{3.1}$$

where $0 \leqslant \mu \leqslant 1$, from which it follows that $p(x = 0|\mu) = 1 - \mu$. The probability distribution over x can therefore be written in the form

$$\text{Bern}(x|\mu) = \mu^x(1 - \mu)^{1-x}, \tag{3.2}$$

Exercise 3.1 which is known as the *Bernoulli* distribution. It is easily verified that this distribution

Telegram: @uni_k

www.konkur.in

is normalized and that it has mean and variance given by

$$\mathbb{E}[x] = \mu \tag{3.3}$$
$$\mathrm{var}[x] = \mu(1-\mu). \tag{3.4}$$

Now suppose we have a data set $\mathcal{D} = \{x_1, \ldots, x_N\}$ of observed values of x. We can construct the likelihood function, which is a function of μ, on the assumption that the observations are drawn independently from $p(x|\mu)$, so that

$$p(\mathcal{D}|\mu) = \prod_{n=1}^{N} p(x_n|\mu) = \prod_{n=1}^{N} \mu^{x_n}(1-\mu)^{1-x_n}. \tag{3.5}$$

We can estimate a value for μ by maximizing the likelihood function or equivalently by maximizing the logarithm of the likelihood, since the log is a monotonic function. The log likelihood function of the Bernoulli distribution is given by

$$\ln p(\mathcal{D}|\mu) = \sum_{n=1}^{N} \ln p(x_n|\mu) = \sum_{n=1}^{N} \{x_n \ln \mu + (1-x_n)\ln(1-\mu)\}. \tag{3.6}$$

At this point, note that the log likelihood function depends on the N observations x_n only through their sum $\sum_n x_n$. This sum provides an example of a *sufficient statistic* for the data under this distribution. If we set the derivative of $\ln p(\mathcal{D}|\mu)$ with respect to μ equal to zero, we obtain the maximum likelihood estimator:

Section 3.4

$$\mu_{\mathrm{ML}} = \frac{1}{N}\sum_{n=1}^{N} x_n, \tag{3.7}$$

which is also known as the *sample mean*. Denoting the number of observations of $x = 1$ (heads) within this data set by m, we can write (3.7) in the form

$$\mu_{\mathrm{ML}} = \frac{m}{N} \tag{3.8}$$

so that the probability of landing heads is given, in this maximum likelihood framework, by the fraction of observations of heads in the data set.

3.1.2 Binomial distribution

We can also work out the distribution for the binary variable x of the number m of observations of $x = 1$, given that the data set has size N. This is called the *binomial* distribution, and from (3.5) we see that it is proportional to $\mu^m(1-\mu)^{N-m}$. To obtain the normalization coefficient, note that out of N coin flips, we have to add up all of the possible ways of obtaining m heads, so that the binomial distribution can be written as

$$\mathrm{Bin}(m|N,\mu) = \binom{N}{m}\mu^m(1-\mu)^{N-m} \tag{3.9}$$

Telegram: @uni_k

www.konkur.in

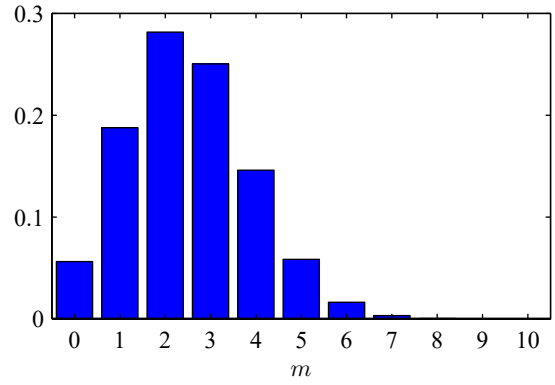


Figure 3.1 Histogram plot of the binomial distribution (3.9) as a function of m for $N = 10$ and $\mu = 0.25$.

where

$$\binom{N}{m} \equiv \frac{N!}{(N-m)!m!} \tag{3.10}$$

is the number of ways of choosing m objects out of a total of N identical objects without replacement. Figure 3.1 shows a plot of the binomial distribution for $N = 10$ and $\mu = 0.25$.

Exercise 3.3

The mean and variance of the binomial distribution can be found by using the results that, for independent events, the mean of the sum is the sum of the means and the variance of the sum is the sum of the variances. Because $m = x_1 + \ldots + x_N$ and because for each observation the mean and variance are given by (3.3) and (3.4), respectively, we have

Exercise 2.10

$$\mathbb{E}[m] \equiv \sum_{m=0}^{N} m \,\mathrm{Bin}(m|N,\mu) = N\mu \tag{3.11}$$

$$\mathrm{var}[m] \equiv \sum_{m=0}^{N} (m - \mathbb{E}[m])^2 \,\mathrm{Bin}(m|N,\mu) = N\mu(1-\mu). \tag{3.12}$$

Exercise 3.4

These results can also be proved directly by using calculus.

3.1.3 Multinomial distribution

Binary variables can be used to describe quantities that can take one of two possible values. Often, however, we encounter discrete variables that can take on one of K possible mutually exclusive states. Although there are various alternative ways to express such variables, we will see shortly that a particularly convenient representation is the 1-of-K scheme, sometimes called 'one-hot encoding', in which the variable is represented by a K-dimensional vector $\mathbf{x}$ in which one of the elements x_k equals 1 and all remaining elements equal 0. So, for instance, if we have a variable that can take $K = 6$ states and a particular observation of the variable happens to

Telegram: @uni_k

www.konkur.in

correspond to the state where $x_3 = 1$, then $\mathbf{x}$ will be represented by

$$\mathbf{x} = (0, 0, 1, 0, 0, 0)^{\mathrm{T}}. \tag{3.13}$$

Note that such vectors satisfy $\sum_{k=1}^{K} x_k = 1$. If we denote the probability of $x_k = 1$ by the parameter μ_k, then the distribution of $\mathbf{x}$ is given by

$$p(\mathbf{x}|\boldsymbol{\mu}) = \prod_{k=1}^{K} \mu_k^{x_k} \tag{3.14}$$

where $\boldsymbol{\mu} = (\mu_1, \ldots, \mu_K)^{\mathrm{T}}$, and the parameters μ_k are constrained to satisfy $\mu_k \geqslant 0$ and $\sum_k \mu_k = 1$, because they represent probabilities. The distribution (3.14) can be regarded as a generalization of the Bernoulli distribution to more than two outcomes. It is easily seen that the distribution is normalized:

$$\sum_{\mathbf{x}} p(\mathbf{x}|\boldsymbol{\mu}) = \sum_{k=1}^{K} \mu_k = 1 \tag{3.15}$$

and that

$$\mathbb{E}[\mathbf{x}|\boldsymbol{\mu}] = \sum_{\mathbf{x}} p(\mathbf{x}|\boldsymbol{\mu})\mathbf{x} = \boldsymbol{\mu}. \tag{3.16}$$

Now consider a data set $\mathcal{D}$ of N independent observations $\mathbf{x}_1, \ldots, \mathbf{x}_N$. The corresponding likelihood function takes the form

$$p(\mathcal{D}|\boldsymbol{\mu}) = \prod_{n=1}^{N} \prod_{k=1}^{K} \mu_k^{x_{nk}} = \prod_{k=1}^{K} \mu_k^{\left(\sum_n x_{nk}\right)} = \prod_{k=1}^{K} \mu_k^{m_k} \tag{3.17}$$

where we see that the likelihood function depends on the N data points only through the K quantities:

$$m_k = \sum_{n=1}^{N} x_{nk}, \tag{3.18}$$

which represent the number of observations of $x_k = 1$. These are called the *sufficient statistics* for this distribution. Note that the variables m_k are subject to the constraint

Section 3.4

$$\sum_{k=1}^{K} m_k = N. \tag{3.19}$$

To find the maximum likelihood solution for $\boldsymbol{\mu}$, we need to maximize $\ln p(\mathcal{D}|\boldsymbol{\mu})$ with respect to μ_k taking account of the constraint (3.15) that the μ_k must sum to one. This can be achieved using a Lagrange multiplier λ and maximizing

Appendix C

$$\sum_{k=1}^{K} m_k \ln \mu_k + \lambda \left(\sum_{k=1}^{K} \mu_k - 1 \right). \tag{3.20}$$

Telegram: @uni_k

www.konkur.in

Setting the derivative of (3.20) with respect to μ_k to zero, we obtain

$$\mu_k = -m_k/\lambda. \tag{3.21}$$

We can solve for the Lagrange multiplier λ by substituting (3.21) into the constraint $\sum_k \mu_k = 1$ to give $\lambda = -N$. Thus, we obtain the maximum likelihood solution for μ_k in the form

$$\mu_k^{\mathrm{ML}} = \frac{m_k}{N}, \tag{3.22}$$

which is the fraction of the N observations for which $x_k = 1$.

We can also consider the joint distribution of the quantities $m_1, \ldots, m_K$, conditioned on the parameter vector $\boldsymbol{\mu}$ and on the total number N of observations. From (3.17), this takes the form

$$\mathrm{Mult}(m_1, m_2, \ldots, m_K|\boldsymbol{\mu}, N) = \binom{N}{m_1 m_2 \ldots m_K} \prod_{k=1}^{K} \mu_k^{m_k}, \tag{3.23}$$

which is known as the *multinomial* distribution. The normalization coefficient is the number of ways of partitioning N objects into K groups of size $m_1, \ldots, m_K$ and is given by

$$\binom{N}{m_1 m_2 \ldots m_K} = \frac{N!}{m_1! m_2! \ldots m_K!}. \tag{3.24}$$

Note that two-state quantities can be represented either as binary variables and modelled using the binomial distribution (3.9) or as 1-of-2 variables and modelled using the distribution (3.14) with $K = 2$.

3.2. The Multivariate Gaussian

The Gaussian, also known as the normal distribution, is a widely used model for the distribution of continuous variables. We have already seen that for of a single variable x, the Gaussian distribution can be written in the form

Section 2.3

$$\mathcal{N}(x|\mu, \sigma^2) = \frac{1}{(2\pi\sigma^2)^{1/2}} \exp\left\{-\frac{1}{2\sigma^2}(x-\mu)^2\right\} \tag{3.25}$$

where μ is the mean and σ^2 is the variance. For a D-dimensional vector $\mathbf{x}$, the multivariate Gaussian distribution takes the form

$$\mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{(2\pi)^{D/2}} \frac{1}{|\boldsymbol{\Sigma}|^{1/2}} \exp\left\{-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^{\mathrm{T}} \boldsymbol{\Sigma}^{-1} (\mathbf{x}-\boldsymbol{\mu})\right\} \tag{3.26}$$

where $\boldsymbol{\mu}$ is the D-dimensional mean vector, $\boldsymbol{\Sigma}$ is the $D \times D$ covariance matrix, and $\det \boldsymbol{\Sigma}$ denotes the determinant of $\boldsymbol{\Sigma}$.

The Gaussian distribution arises in many different contexts and can be motivated from a variety of different perspectives. For example, we have already seen that for

Section 2.5

Telegram: @uni_k

www.konkur.in

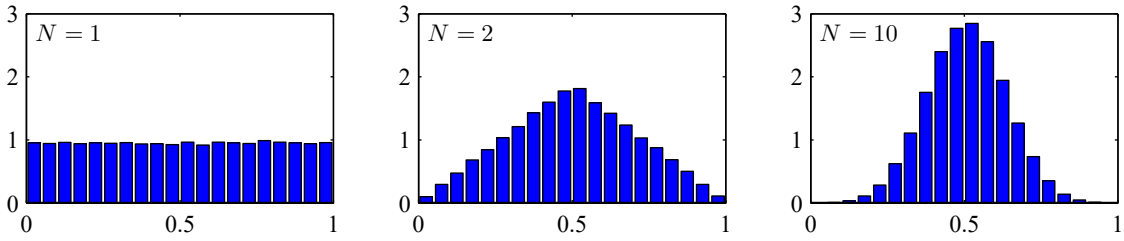


Figure 3.2 Histogram plots of the mean of N uniformly distributed numbers for various values of N. We observe that as N increases, the distribution tends towards a Gaussian.

Exercise 3.8

a single real variable, the distribution that maximizes the entropy is the Gaussian. This property applies also to the multivariate Gaussian.

Another situation in which the Gaussian distribution arises is when we consider the sum of multiple random variables. The *central limit theorem* tells us that, subject to certain mild conditions, the sum of a set of random variables, which is of course itself a random variable, has a distribution that becomes increasingly Gaussian as the number of terms in the sum increases (Walker, 1969). We can illustrate this by considering N variables $x_1, \ldots, x_N$ each of which has a uniform distribution over the interval $[0, 1]$ and then considering the distribution of the mean $(x_1 + \cdots + x_N)/N$. For large N, this distribution tends to a Gaussian, as illustrated in Figure 3.2. In practice, the convergence to a Gaussian as N increases can be very rapid. One consequence of this result is that the binomial distribution (3.9), which is a distribution over m defined by the sum of N observations of the random binary variable x, will tend to a Gaussian as $N \to \infty$ (see Figure 3.1 for $N = 10$).

The Gaussian distribution has many important analytical properties, and we will consider several of these in detail. As a result, this section will be rather more technically involved than some of the earlier sections and will require familiarity with various matrix identities.

Appendix A

3.2.1 Geometry of the Gaussian

We begin by considering the geometrical form of the Gaussian distribution. The functional dependence of the Gaussian on $\mathbf{x}$ is through the quadratic form

$$\Delta^2 = (\mathbf{x} - \boldsymbol{\mu})^{\mathrm{T}} \boldsymbol{\Sigma}^{-1} (\mathbf{x} - \boldsymbol{\mu}), \tag{3.27}$$

which appears in the exponent. The quantity Δ is called the *Mahalanobis distance* from $\boldsymbol{\mu}$ to $\mathbf{x}$. It reduces to the Euclidean distance when $\boldsymbol{\Sigma}$ is the identity matrix. The Gaussian distribution is constant on surfaces in $\mathbf{x}$-space for which this quadratic form is constant.

Exercise 3.11

First, note that the matrix $\boldsymbol{\Sigma}$ can be taken to be symmetric, without loss of generality, because any antisymmetric component would disappear from the exponent. Now consider the eigenvector equation for the covariance matrix

$$\boldsymbol{\Sigma}\mathbf{u}_i = \lambda_i \mathbf{u}_i \tag{3.28}$$

Telegram: @uni_k

www.konkur.in

where $i = 1, \ldots, D$. Because $\boldsymbol{\Sigma}$ is a real, symmetric matrix, its eigenvalues will be
Exercise 3.12 real, and its eigenvectors can be chosen to form an orthonormal set, so that

$$\mathbf{u}_i^{\mathrm{T}} \mathbf{u}_j = I_{ij} \tag{3.29}$$

where I_{ij} is the i, j element of the identity matrix and satisfies

$$I_{ij} = \begin{cases} 1, & \text{if } i = j \\ 0, & \text{otherwise.} \end{cases} \tag{3.30}$$

The covariance matrix $\boldsymbol{\Sigma}$ can be expressed as an expansion in terms of its eigenvec-
Exercise 3.13 tors in the form

$$\boldsymbol{\Sigma} = \sum_{i=1}^{D} \lambda_i \mathbf{u}_i \mathbf{u}_i^{\mathrm{T}} \tag{3.31}$$

and similarly the inverse covariance matrix $\boldsymbol{\Sigma}^{-1}$ can be expressed as

$$\boldsymbol{\Sigma}^{-1} = \sum_{i=1}^{D} \frac{1}{\lambda_i} \mathbf{u}_i \mathbf{u}_i^{\mathrm{T}}. \tag{3.32}$$

Substituting (3.32) into (3.27), the quadratic form becomes

$$\Delta^2 = \sum_{i=1}^{D} \frac{y_i^2}{\lambda_i} \tag{3.33}$$

where we have defined

$$y_i = \mathbf{u}_i^{\mathrm{T}} (\mathbf{x} - \boldsymbol{\mu}). \tag{3.34}$$

We can interpret $\{y_i\}$ as a new coordinate system defined by the orthonormal vectors $\mathbf{u}_i$ that are shifted and rotated with respect to the original x_i coordinates. Forming the vector $\mathbf{y} = (y_1, \ldots, y_D)^{\mathrm{T}}$, we have

$$\mathbf{y} = \mathbf{U}(\mathbf{x} - \boldsymbol{\mu}) \tag{3.35}$$

where $\mathbf{U}$ is a matrix whose rows are given by $\mathbf{u}_i^{\mathrm{T}}$. From (3.29) it follows that $\mathbf{U}$ is
Appendix A an *orthogonal* matrix, i.e., it satisfies $\mathbf{U}\mathbf{U}^{\mathrm{T}} = \mathbf{U}^{\mathrm{T}}\mathbf{U} = \mathbf{I}$, where $\mathbf{I}$ is the identity matrix.

The quadratic form, and hence the Gaussian density, is constant on surfaces for which (3.33) is constant. If all the eigenvalues λ_i are positive, then these surfaces represent ellipsoids, with their centres at $\boldsymbol{\mu}$ and their axes oriented along $\mathbf{u}_i$, and with scaling factors in the directions of the axes given by $\lambda_i^{1/2}$, as illustrated in Figure 3.3.

For the Gaussian distribution to be well defined, it is necessary for all the eigenvalues λ_i of the covariance matrix to be strictly positive, otherwise the distribution cannot be properly normalized. A matrix whose eigenvalues are strictly positive is
Chapter 16 said to be *positive definite*. When we discuss latent variable models, we will encounter Gaussian distributions for which one or more of the eigenvalues are zero, in

Telegram: @uni_k

www.konkur.in

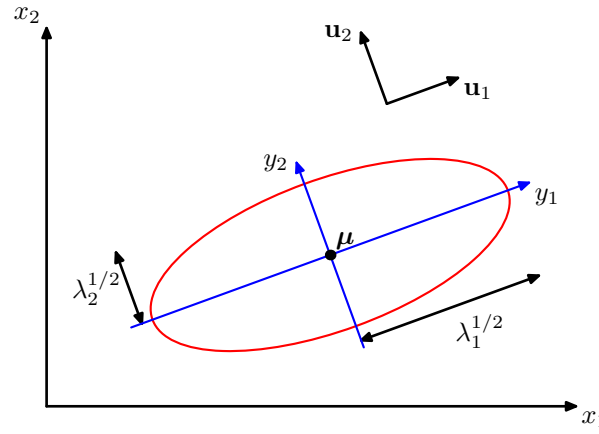


Figure 3.3 The red curve shows the elliptical surface of constant probability density for a Gaussian in a two-dimensional space $\mathbf{x} = (x_1, x_2)$ on which the density is $\exp(-1/2)$ of its value at $\mathbf{x} = \boldsymbol{\mu}$. The axes of the ellipse are defined by the eigenvectors $\mathbf{u}_i$ of the covariance matrix, with corresponding eigenvalues λ_i.

which case the distribution is singular and is confined to a subspace of lower dimensionality. If all the eigenvalues are non-negative, then the covariance matrix is said to be *positive semidefinite*.

Now consider the form of the Gaussian distribution in the new coordinate system defined by the y_i. In going from the $\mathbf{x}$ to the $\mathbf{y}$ coordinate system, we have a Jacobian matrix $\mathbf{J}$ with elements given by

$$J_{ij} = \frac{\partial x_i}{\partial y_j} = U_{ji} \tag{3.36}$$

where U_{ji} are the elements of the matrix $\mathbf{U}^{\mathrm{T}}$. Using the orthonormality property of the matrix $\mathbf{U}$, we see that the square of the determinant of the Jacobian matrix is

$$|\mathbf{J}|^2 = \left|\mathbf{U}^{\mathrm{T}}\right|^2 = \left|\mathbf{U}^{\mathrm{T}}\right||\mathbf{U}| = \left|\mathbf{U}^{\mathrm{T}}\mathbf{U}\right| = |\mathbf{I}| = 1 \tag{3.37}$$

and, hence, $|\mathbf{J}| = 1$. Also, the determinant $|\boldsymbol{\Sigma}|$ of the covariance matrix can be written as the product of its eigenvalues, and hence

$$|\boldsymbol{\Sigma}|^{1/2} = \prod_{j=1}^{D} \lambda_j^{1/2}. \tag{3.38}$$

Thus, in the y_j coordinate system, the Gaussian distribution takes the form

$$p(\mathbf{y}) = p(\mathbf{x})|\mathbf{J}| = \prod_{j=1}^{D} \frac{1}{(2\pi\lambda_j)^{1/2}} \exp\left\{-\frac{y_j^2}{2\lambda_j}\right\}, \tag{3.39}$$

which is the product of D independent univariate Gaussian distributions. The eigenvectors therefore define a new set of shifted and rotated coordinates with respect to which the joint probability distribution factorizes into a product of independent distributions. The integral of the distribution in the $\mathbf{y}$ coordinate system is then

$$\int p(\mathbf{y})\,\mathrm{d}\mathbf{y} = \prod_{j=1}^{D} \int_{-\infty}^{\infty} \frac{1}{(2\pi\lambda_j)^{1/2}} \exp\left\{-\frac{y_j^2}{2\lambda_j}\right\} \mathrm{d}y_j = 1 \tag{3.40}$$

Telegram: @uni_k

www.konkur.in

where we have used the result (2.51) for the normalization of the univariate Gaussian. This confirms that the multivariate Gaussian (3.26) is indeed normalized.

3.2.2 Moments

We now look at the moments of the Gaussian distribution and thereby provide an interpretation of the parameters $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$. The expectation of $\mathbf{x}$ under the Gaussian distribution is given by

$$\begin{aligned}\mathbb{E}[\mathbf{x}] &= \frac{1}{(2\pi)^{D/2}}\frac{1}{|\boldsymbol{\Sigma}|^{1/2}}\int \exp\left\{-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^{\mathrm{T}}\boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})\right\}\mathbf{x}\,\mathrm{d}\mathbf{x} \\ &= \frac{1}{(2\pi)^{D/2}}\frac{1}{|\boldsymbol{\Sigma}|^{1/2}}\int \exp\left\{-\frac{1}{2}\mathbf{z}^{\mathrm{T}}\boldsymbol{\Sigma}^{-1}\mathbf{z}\right\}(\mathbf{z}+\boldsymbol{\mu})\,\mathrm{d}\mathbf{z} \qquad (3.41)\end{aligned}$$

where we have changed variables using $\mathbf{z} = \mathbf{x} - \boldsymbol{\mu}$. Note that the exponent is an even function of the components of $\mathbf{z}$, and because the integrals over these are taken over the range $(-\infty, \infty)$, the term in $\mathbf{z}$ in the factor $(\mathbf{z} + \boldsymbol{\mu})$ will vanish by symmetry. Thus,

$$\mathbb{E}[\mathbf{x}] = \boldsymbol{\mu}, \qquad (3.42)$$

and so we refer to $\boldsymbol{\mu}$ as the mean of the Gaussian distribution.

We now consider second-order moments of the Gaussian. In the univariate case, we considered the second-order moment given by $\mathbb{E}[x^2]$. For the multivariate Gaussian, there are D^2 second-order moments given by $\mathbb{E}[x_i x_j]$, which we can group together to form the matrix $\mathbb{E}[\mathbf{x}\mathbf{x}^{\mathrm{T}}]$. This matrix can be written as

$$\begin{aligned}\mathbb{E}[\mathbf{x}\mathbf{x}^{\mathrm{T}}] &= \frac{1}{(2\pi)^{D/2}}\frac{1}{|\boldsymbol{\Sigma}|^{1/2}}\int \exp\left\{-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^{\mathrm{T}}\boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})\right\}\mathbf{x}\mathbf{x}^{\mathrm{T}}\,\mathrm{d}\mathbf{x} \\ &= \frac{1}{(2\pi)^{D/2}}\frac{1}{|\boldsymbol{\Sigma}|^{1/2}}\int \exp\left\{-\frac{1}{2}\mathbf{z}^{\mathrm{T}}\boldsymbol{\Sigma}^{-1}\mathbf{z}\right\}(\mathbf{z}+\boldsymbol{\mu})(\mathbf{z}+\boldsymbol{\mu})^{\mathrm{T}}\,\mathrm{d}\mathbf{z} \qquad (3.43)\end{aligned}$$

where again we have changed variables using $\mathbf{z} = \mathbf{x} - \boldsymbol{\mu}$. Note that the cross-terms involving $\boldsymbol{\mu}\mathbf{z}^{\mathrm{T}}$ and $\boldsymbol{\mu}^{\mathrm{T}}\mathbf{z}$ will again vanish by symmetry. The term $\boldsymbol{\mu}\boldsymbol{\mu}^{\mathrm{T}}$ is constant and can be taken outside the integral, which itself is unity because the Gaussian distribution is normalized. Consider the term involving $\mathbf{z}\mathbf{z}^{\mathrm{T}}$. Again, we can make use of the eigenvector expansion of the covariance matrix given by (3.28), together with the completeness of the set of eigenvectors, to write

$$\mathbf{z} = \sum_{j=1}^{D} y_j \mathbf{u}_j \qquad (3.44)$$

Telegram: @uni_k

www.konkur.in

where $y_j = \mathbf{u}_j^{\mathrm{T}}\mathbf{z}$, which gives

$$\begin{aligned}
&\frac{1}{(2\pi)^{D/2}}\frac{1}{|\boldsymbol{\Sigma}|^{1/2}}\int \exp\left\{-\frac{1}{2}\mathbf{z}^{\mathrm{T}}\boldsymbol{\Sigma}^{-1}\mathbf{z}\right\}\mathbf{z}\mathbf{z}^{\mathrm{T}}\,\mathrm{d}\mathbf{z} \\
&= \frac{1}{(2\pi)^{D/2}}\frac{1}{|\boldsymbol{\Sigma}|^{1/2}}\sum_{i=1}^{D}\sum_{j=1}^{D}\mathbf{u}_i\mathbf{u}_j^{\mathrm{T}}\int \exp\left\{-\sum_{k=1}^{D}\frac{y_k^2}{2\lambda_k}\right\}y_i y_j\,\mathrm{d}\mathbf{y} \\
&= \sum_{i=1}^{D}\mathbf{u}_i\mathbf{u}_i^{\mathrm{T}}\lambda_i = \boldsymbol{\Sigma}
\end{aligned} \tag{3.45}$$

where we have made use of the eigenvector equation (3.28), together with the fact that the integral on the middle line vanishes by symmetry unless $i = j$. In the final line we have made use of the results (2.53) and (3.38), together with (3.31). Thus, we have

$$\mathbb{E}[\mathbf{x}\mathbf{x}^{\mathrm{T}}] = \boldsymbol{\mu}\boldsymbol{\mu}^{\mathrm{T}} + \boldsymbol{\Sigma}. \tag{3.46}$$

When defining the variance for a single random variable, we subtracted the mean before taking the second moment. Similarly, in the multivariate case it is again convenient to subtract off the mean, giving rise to the *covariance* of a random vector $\mathbf{x}$ defined by

$$\mathrm{cov}[\mathbf{x}] = \mathbb{E}\left[(\mathbf{x} - \mathbb{E}[\mathbf{x}])(\mathbf{x} - \mathbb{E}[\mathbf{x}])^{\mathrm{T}}\right]. \tag{3.47}$$

For the specific case of a Gaussian distribution, we can make use of $\mathbb{E}[\mathbf{x}] = \boldsymbol{\mu}$, together with the result (3.46), to give

$$\mathrm{cov}[\mathbf{x}] = \boldsymbol{\Sigma}. \tag{3.48}$$

Because the parameter matrix $\boldsymbol{\Sigma}$ governs the covariance of $\mathbf{x}$ under the Gaussian distribution, it is called the covariance matrix.

3.2.3 Limitations

Although the Gaussian distribution (3.26) is often used as a simple density model, it suffers from some significant limitations. Consider the number of free parameters in the distribution. A general symmetric covariance matrix $\boldsymbol{\Sigma}$ will have $D(D+1)/2$ independent parameters, and there are another D independent parameters in $\boldsymbol{\mu}$, giving $D(D+3)/2$ parameters in total. For large D, the total number of parameters therefore grows quadratically with D, and the computational task of manipulating and inverting the large matrices can become prohibitive. One way to address this problem is to use restricted forms of the covariance matrix. If we consider covariance matrices that are *diagonal*, so that $\boldsymbol{\Sigma} = \mathrm{diag}(\sigma_i^2)$, we then have a total of $2D$ independent parameters in the density model. The corresponding contours of constant density are given by axis-aligned ellipsoids. We could further restrict the covariance matrix to be proportional to the identity matrix, $\boldsymbol{\Sigma} = \sigma^2\mathbf{I}$, known as an *isotropic* covariance, giving $D + 1$ independent parameters in the model together with spherical surfaces of constant density. The three possibilities of general, diagonal, and isotropic covariance matrices are illustrated in Figure 3.4. Unfortunately,

Exercise 3.15

Telegram: @uni_k

www.konkur.in

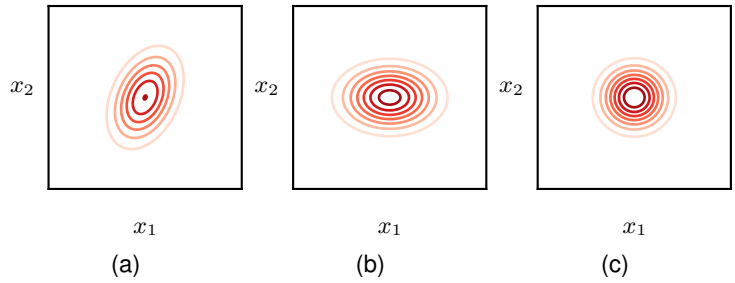


Figure 3.4 Contours of constant probability density for a Gaussian distribution in two dimensions in which the covariance matrix is (a) of general form, (b) diagonal, in which case the elliptical contours are aligned with the coordinate axes, and (c) proportional to the identity matrix, in which case the contours are concentric circles.

whereas such approaches limit the number of degrees of freedom in the distribution and make inversion of the covariance matrix a much faster operation, they also greatly restrict the form of the probability density and limit its ability to capture interesting correlations in the data.

A further limitation of the Gaussian distribution is that it is intrinsically unimodal (i.e., has a single maximum) and so is unable to provide a good approximation to multimodal distributions. Thus, the Gaussian distribution can be both too flexible, in the sense of having too many parameters, and too limited in the range of distributions that it can adequately represent. We will see later that the introduction of *latent* variables, also called *hidden* variables or *unobserved* variables, allows both of these problems to be addressed. In particular, a rich family of multimodal distributions is obtained by introducing discrete latent variables leading to mixtures of Gaussians. Similarly, the introduction of continuous latent variables leads to models in which the number of free parameters can be controlled independently of the dimensionality D of the data space while still allowing the model to capture the dominant correlations in the data set.

Section 3.2.9

Chapter 16

3.2.4 Conditional distribution

An important property of a multivariate Gaussian distribution is that if two sets of variables are jointly Gaussian, then the conditional distribution of one set conditioned on the other is again Gaussian. Similarly, the marginal distribution of either set is also Gaussian.

First, consider the case of conditional distributions. Suppose that $\mathbf{x}$ is a D-dimensional vector with Gaussian distribution $\mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma})$ and that we partition $\mathbf{x}$ into two disjoint subsets $\mathbf{x}_a$ and $\mathbf{x}_b$. Without loss of generality, we can take $\mathbf{x}_a$ to form the first M components of $\mathbf{x}$, with $\mathbf{x}_b$ comprising the remaining $D - M$ components, so that

$$\mathbf{x} = \begin{pmatrix} \mathbf{x}_a \\ \mathbf{x}_b \end{pmatrix}. \tag{3.49}$$

We also define corresponding partitions of the mean vector $\boldsymbol{\mu}$ given by

$$\boldsymbol{\mu} = \begin{pmatrix} \boldsymbol{\mu}_a \\ \boldsymbol{\mu}_b \end{pmatrix} \tag{3.50}$$

Telegram: @uni_k

www.konkur.in

and of the covariance matrix $\boldsymbol{\Sigma}$ given by

$$\boldsymbol{\Sigma} = \begin{pmatrix} \boldsymbol{\Sigma}_{aa} & \boldsymbol{\Sigma}_{ab} \\ \boldsymbol{\Sigma}_{ba} & \boldsymbol{\Sigma}_{bb} \end{pmatrix}. \tag{3.51}$$

Note that the symmetry $\boldsymbol{\Sigma}^{\mathrm{T}} = \boldsymbol{\Sigma}$ of the covariance matrix implies that $\boldsymbol{\Sigma}_{aa}$ and $\boldsymbol{\Sigma}_{bb}$ are symmetric and that $\boldsymbol{\Sigma}_{ba} = \boldsymbol{\Sigma}_{ab}^{\mathrm{T}}$.

In many situations, it will be convenient to work with the inverse of the covariance matrix:

$$\boldsymbol{\Lambda} \equiv \boldsymbol{\Sigma}^{-1}, \tag{3.52}$$

which is known as the *precision matrix*. In fact, we will see that some properties of Gaussian distributions are most naturally expressed in terms of the covariance, whereas others take a simpler form when viewed in terms of the precision. We therefore also introduce the partitioned form of the precision matrix:

$$\boldsymbol{\Lambda} = \begin{pmatrix} \boldsymbol{\Lambda}_{aa} & \boldsymbol{\Lambda}_{ab} \\ \boldsymbol{\Lambda}_{ba} & \boldsymbol{\Lambda}_{bb} \end{pmatrix} \tag{3.53}$$

corresponding to the partitioning (3.49) of the vector $\mathbf{x}$. Because the inverse of a symmetric matrix is also symmetric, we see that $\boldsymbol{\Lambda}_{aa}$ and $\boldsymbol{\Lambda}_{bb}$ are symmetric and that $\boldsymbol{\Lambda}_{ba} = \boldsymbol{\Lambda}_{ab}^{\mathrm{T}}$. It should be stressed at this point that, for instance, $\boldsymbol{\Lambda}_{aa}$ is not simply given by the inverse of $\boldsymbol{\Sigma}_{aa}$. In fact, we will shortly examine the relation between the inverse of a partitioned matrix and the inverses of its partitions.

Exercise 3.16

We begin by finding an expression for the conditional distribution $p(\mathbf{x}_a|\mathbf{x}_b)$. From the product rule of probability, we see that this conditional distribution can be evaluated from the joint distribution $p(\mathbf{x}) = p(\mathbf{x}_a, \mathbf{x}_b)$ simply by fixing $\mathbf{x}_b$ to the observed value and normalizing the resulting expression to obtain a valid probability distribution over $\mathbf{x}_a$. Instead of performing this normalization explicitly, we can obtain the solution more efficiently by considering the quadratic form in the exponent of the Gaussian distribution given by (3.27) and then reinstating the normalization coefficient at the end of the calculation. If we make use of the partitioning (3.49), (3.50), and (3.53), we obtain

$$\begin{aligned} -\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^{\mathrm{T}}\boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu}) = \\ -\frac{1}{2}(\mathbf{x}_a-\boldsymbol{\mu}_a)^{\mathrm{T}}\boldsymbol{\Lambda}_{aa}(\mathbf{x}_a-\boldsymbol{\mu}_a) - \frac{1}{2}(\mathbf{x}_a-\boldsymbol{\mu}_a)^{\mathrm{T}}\boldsymbol{\Lambda}_{ab}(\mathbf{x}_b-\boldsymbol{\mu}_b) \\ -\frac{1}{2}(\mathbf{x}_b-\boldsymbol{\mu}_b)^{\mathrm{T}}\boldsymbol{\Lambda}_{ba}(\mathbf{x}_a-\boldsymbol{\mu}_a) - \frac{1}{2}(\mathbf{x}_b-\boldsymbol{\mu}_b)^{\mathrm{T}}\boldsymbol{\Lambda}_{bb}(\mathbf{x}_b-\boldsymbol{\mu}_b). \end{aligned} \tag{3.54}$$

We see that as a function of $\mathbf{x}_a$, this is again a quadratic form, and hence, the corresponding conditional distribution $p(\mathbf{x}_a|\mathbf{x}_b)$ will be Gaussian. Because this distribution is completely characterized by its mean and its covariance, our goal will be to identify expressions for the mean and covariance of $p(\mathbf{x}_a|\mathbf{x}_b)$ by inspection of (3.54).

This is an example of a rather common operation associated with Gaussian distributions, sometimes called 'completing the square', in which we are given a

Telegram: @uni_k

www.konkur.in

quadratic form defining the exponent terms in a Gaussian distribution and we need to determine the corresponding mean and covariance. Such problems can be solved straightforwardly by noting that the exponent in a general Gaussian distribution $\mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma})$ can be written as

$$-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^{\mathrm{T}}\boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu}) = -\frac{1}{2}\mathbf{x}^{\mathrm{T}}\boldsymbol{\Sigma}^{-1}\mathbf{x} + \mathbf{x}^{\mathrm{T}}\boldsymbol{\Sigma}^{-1}\boldsymbol{\mu} + \text{const} \tag{3.55}$$

where 'const' denotes terms that are independent of $\mathbf{x}$, We have also made use of the symmetry of $\boldsymbol{\Sigma}$. Thus, if we take our general quadratic form and express it in the form given by the right-hand side of (3.55), then we can immediately equate the matrix of coefficients entering the second-order term in $\mathbf{x}$ to the inverse covariance matrix $\boldsymbol{\Sigma}^{-1}$ and the coefficient of the linear term in $\mathbf{x}$ to $\boldsymbol{\Sigma}^{-1}\boldsymbol{\mu}$, from which we can obtain $\boldsymbol{\mu}$.

Now let us apply this procedure to the conditional Gaussian distribution $p(\mathbf{x}_a|\mathbf{x}_b)$ for which the quadratic form in the exponent is given by (3.54). We will denote the mean and covariance of this distribution by $\boldsymbol{\mu}_{a|b}$ and $\boldsymbol{\Sigma}_{a|b}$, respectively. Consider the functional dependence of (3.54) on $\mathbf{x}_a$ in which $\mathbf{x}_b$ is regarded as a constant. If we pick out all terms that are second order in $\mathbf{x}_a$, we have

$$-\frac{1}{2}\mathbf{x}_a^{\mathrm{T}}\boldsymbol{\Lambda}_{aa}\mathbf{x}_a \tag{3.56}$$

from which we can immediately conclude that the covariance (inverse precision) of $p(\mathbf{x}_a|\mathbf{x}_b)$ is given by

$$\boldsymbol{\Sigma}_{a|b} = \boldsymbol{\Lambda}_{aa}^{-1}. \tag{3.57}$$

Now consider all the terms in (3.54) that are linear in $\mathbf{x}_a$:

$$\mathbf{x}_a^{\mathrm{T}}\left\{\boldsymbol{\Lambda}_{aa}\boldsymbol{\mu}_a - \boldsymbol{\Lambda}_{ab}(\mathbf{x}_b - \boldsymbol{\mu}_b)\right\} \tag{3.58}$$

where we have used $\boldsymbol{\Lambda}_{ba}^{\mathrm{T}} = \boldsymbol{\Lambda}_{ab}$. From our discussion of the general form (3.55), the coefficient of $\mathbf{x}_a$ in this expression must equal $\boldsymbol{\Sigma}_{a|b}^{-1}\boldsymbol{\mu}_{a|b}$ and, hence,

$$\begin{aligned}\boldsymbol{\mu}_{a|b} &= \boldsymbol{\Sigma}_{a|b}\left\{\boldsymbol{\Lambda}_{aa}\boldsymbol{\mu}_a - \boldsymbol{\Lambda}_{ab}(\mathbf{x}_b - \boldsymbol{\mu}_b)\right\} \\ &= \boldsymbol{\mu}_a - \boldsymbol{\Lambda}_{aa}^{-1}\boldsymbol{\Lambda}_{ab}(\mathbf{x}_b - \boldsymbol{\mu}_b)\end{aligned} \tag{3.59}$$

where we have made use of (3.57).

The results (3.57) and (3.59) are expressed in terms of the partitioned precision matrix of the original joint distribution $p(\mathbf{x}_a, \mathbf{x}_b)$. We can also express these results in terms of the corresponding partitioned covariance matrix. To do this, we make use of the following identity for the inverse of a partitioned matrix:

Exercise 3.18

$$\begin{pmatrix}\mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D}\end{pmatrix}^{-1} = \begin{pmatrix}\mathbf{M} & -\mathbf{M}\mathbf{B}\mathbf{D}^{-1} \\ -\mathbf{D}^{-1}\mathbf{C}\mathbf{M} & \mathbf{D}^{-1} + \mathbf{D}^{-1}\mathbf{C}\mathbf{M}\mathbf{B}\mathbf{D}^{-1}\end{pmatrix} \tag{3.60}$$

where we have defined

$$\mathbf{M} = (\mathbf{A} - \mathbf{B}\mathbf{D}^{-1}\mathbf{C})^{-1}. \tag{3.61}$$

Telegram: @uni_k

www.konkur.in

The quantity $\mathbf{M}^{-1}$ is known as the *Schur complement* of the matrix on the left-hand side of (3.60) with respect to the submatrix $\mathbf{D}$. Using the definition

$$\begin{pmatrix} \boldsymbol{\Sigma}_{aa} & \boldsymbol{\Sigma}_{ab} \\ \boldsymbol{\Sigma}_{ba} & \boldsymbol{\Sigma}_{bb} \end{pmatrix}^{-1} = \begin{pmatrix} \boldsymbol{\Lambda}_{aa} & \boldsymbol{\Lambda}_{ab} \\ \boldsymbol{\Lambda}_{ba} & \boldsymbol{\Lambda}_{bb} \end{pmatrix} \tag{3.62}$$

and making use of (3.60), we have

$$\boldsymbol{\Lambda}_{aa} = (\boldsymbol{\Sigma}_{aa} - \boldsymbol{\Sigma}_{ab}\boldsymbol{\Sigma}_{bb}^{-1}\boldsymbol{\Sigma}_{ba})^{-1} \tag{3.63}$$

$$\boldsymbol{\Lambda}_{ab} = -(\boldsymbol{\Sigma}_{aa} - \boldsymbol{\Sigma}_{ab}\boldsymbol{\Sigma}_{bb}^{-1}\boldsymbol{\Sigma}_{ba})^{-1}\boldsymbol{\Sigma}_{ab}\boldsymbol{\Sigma}_{bb}^{-1}. \tag{3.64}$$

From these we obtain the following expressions for the mean and covariance of the conditional distribution $p(\mathbf{x}_a|\mathbf{x}_b)$:

$$\boldsymbol{\mu}_{a|b} = \boldsymbol{\mu}_a + \boldsymbol{\Sigma}_{ab}\boldsymbol{\Sigma}_{bb}^{-1}(\mathbf{x}_b - \boldsymbol{\mu}_b) \tag{3.65}$$

$$\boldsymbol{\Sigma}_{a|b} = \boldsymbol{\Sigma}_{aa} - \boldsymbol{\Sigma}_{ab}\boldsymbol{\Sigma}_{bb}^{-1}\boldsymbol{\Sigma}_{ba}. \tag{3.66}$$

Comparing (3.57) and (3.66), we see that the conditional distribution $p(\mathbf{x}_a|\mathbf{x}_b)$ takes a simpler form when expressed in terms of the partitioned precision matrix than when it is expressed in terms of the partitioned covariance matrix. Note that the mean of the conditional distribution $p(\mathbf{x}_a|\mathbf{x}_b)$, given by (3.65), is a linear function of $\mathbf{x}_b$ and that the covariance, given by (3.66), is independent of $\mathbf{x}_b$. This represents an example of a *linear-Gaussian* model.

Section 11.1.4

3.2.5 Marginal distribution

We have seen that if a joint distribution $p(\mathbf{x}_a, \mathbf{x}_b)$ is Gaussian, then the conditional distribution $p(\mathbf{x}_a|\mathbf{x}_b)$ will again be Gaussian. Now we turn to a discussion of the marginal distribution given by

$$p(\mathbf{x}_a) = \int p(\mathbf{x}_a, \mathbf{x}_b)\, \mathrm{d}\mathbf{x}_b, \tag{3.67}$$

which, as we will see, is also Gaussian. Once again, our strategy for calculating this distribution will be to focus on the quadratic form in the exponent of the joint distribution and thereby to identify the mean and covariance of the marginal distribution $p(\mathbf{x}_a)$.

The quadratic form for the joint distribution can be expressed, using the partitioned precision matrix, in the form (3.54). Our goal is to integrate out $\mathbf{x}_b$, which is most easily achieved by first considering the terms involving $\mathbf{x}_b$ and then completing the square to facilitate the integration. Picking out just those terms that involve $\mathbf{x}_b$, we have

$$-\frac{1}{2}\mathbf{x}_b^{\mathrm{T}}\boldsymbol{\Lambda}_{bb}\mathbf{x}_b + \mathbf{x}_b^{T}\mathbf{m} = -\frac{1}{2}(\mathbf{x}_b - \boldsymbol{\Lambda}_{bb}^{-1}\mathbf{m})^{\mathrm{T}}\boldsymbol{\Lambda}_{bb}(\mathbf{x}_b - \boldsymbol{\Lambda}_{bb}^{-1}\mathbf{m}) + \frac{1}{2}\mathbf{m}^{\mathrm{T}}\boldsymbol{\Lambda}_{bb}^{-1}\mathbf{m} \tag{3.68}$$

where we have defined

$$\mathbf{m} = \boldsymbol{\Lambda}_{bb}\boldsymbol{\mu}_b - \boldsymbol{\Lambda}_{ba}(\mathbf{x}_a - \boldsymbol{\mu}_a). \tag{3.69}$$

Telegram: @uni_k

www.konkur.in

We see that the dependence on $\mathbf{x}_b$ has been cast into the standard quadratic form of a Gaussian distribution corresponding to the first term on the right-hand side of (3.68) plus a term that does not depend on $\mathbf{x}_b$ (but that does depend on $\mathbf{x}_a$). Thus, when we take the exponential of this quadratic form, we see that the integration over $\mathbf{x}_b$ required by (3.67) will take the form

$$\int \exp\left\{-\frac{1}{2}(\mathbf{x}_b - \boldsymbol{\Lambda}_{bb}^{-1}\mathbf{m})^{\mathrm{T}}\boldsymbol{\Lambda}_{bb}(\mathbf{x}_b - \boldsymbol{\Lambda}_{bb}^{-1}\mathbf{m})\right\} \mathrm{d}\mathbf{x}_b. \tag{3.70}$$

This integration is easily performed by noting that it is the integral over an unnormalized Gaussian, and so the result will be the reciprocal of the normalization coefficient. We know from the form of the normalized Gaussian given by (3.26) that this coefficient is independent of the mean and depends only on the determinant of the covariance matrix. Thus, by completing the square with respect to $\mathbf{x}_b$, we can integrate out $\mathbf{x}_b$ so that the only term remaining from the contributions on the left-hand side of (3.68) that depends on $\mathbf{x}_a$ is the last term on the right-hand side of (3.68) in which $\mathbf{m}$ is given by (3.69). Combining this term with the remaining terms from (3.54) that depend on $\mathbf{x}_a$, we obtain

$$\begin{aligned}
&\frac{1}{2}\left[\boldsymbol{\Lambda}_{bb}\boldsymbol{\mu}_b - \boldsymbol{\Lambda}_{ba}(\mathbf{x}_a - \boldsymbol{\mu}_a)\right]^{\mathrm{T}}\boldsymbol{\Lambda}_{bb}^{-1}\left[\boldsymbol{\Lambda}_{bb}\boldsymbol{\mu}_b - \boldsymbol{\Lambda}_{ba}(\mathbf{x}_a - \boldsymbol{\mu}_a)\right] \\
&\qquad -\frac{1}{2}\mathbf{x}_a^{\mathrm{T}}\boldsymbol{\Lambda}_{aa}\mathbf{x}_a + \mathbf{x}_a^{\mathrm{T}}(\boldsymbol{\Lambda}_{aa}\boldsymbol{\mu}_a + \boldsymbol{\Lambda}_{ab}\boldsymbol{\mu}_b) + \text{const} \\
&= \quad -\frac{1}{2}\mathbf{x}_a^{\mathrm{T}}(\boldsymbol{\Lambda}_{aa} - \boldsymbol{\Lambda}_{ab}\boldsymbol{\Lambda}_{bb}^{-1}\boldsymbol{\Lambda}_{ba})\mathbf{x}_a \\
&\qquad +\mathbf{x}_a^{\mathrm{T}}(\boldsymbol{\Lambda}_{aa} - \boldsymbol{\Lambda}_{ab}\boldsymbol{\Lambda}_{bb}^{-1}\boldsymbol{\Lambda}_{ba})\boldsymbol{\mu}_a + \text{const}
\end{aligned} \tag{3.71}$$

where 'const' denotes quantities independent of $\mathbf{x}_a$. Again, by comparison with (3.55), we see that the covariance of the marginal distribution $p(\mathbf{x}_a)$ is given by

$$\boldsymbol{\Sigma}_a = (\boldsymbol{\Lambda}_{aa} - \boldsymbol{\Lambda}_{ab}\boldsymbol{\Lambda}_{bb}^{-1}\boldsymbol{\Lambda}_{ba})^{-1}. \tag{3.72}$$

Similarly, the mean is given by

$$\boldsymbol{\Sigma}_a(\boldsymbol{\Lambda}_{aa} - \boldsymbol{\Lambda}_{ab}\boldsymbol{\Lambda}_{bb}^{-1}\boldsymbol{\Lambda}_{ba})\boldsymbol{\mu}_a = \boldsymbol{\mu}_a \tag{3.73}$$

where we have used (3.72). The covariance (3.72) is expressed in terms of the partitioned precision matrix given by (3.53). We can rewrite this in terms of the corresponding partitioning of the covariance matrix given by (3.51), as we did for the conditional distribution. These partitioned matrices are related by

$$\begin{pmatrix}\boldsymbol{\Lambda}_{aa} & \boldsymbol{\Lambda}_{ab} \\ \boldsymbol{\Lambda}_{ba} & \boldsymbol{\Lambda}_{bb}\end{pmatrix}^{-1} = \begin{pmatrix}\boldsymbol{\Sigma}_{aa} & \boldsymbol{\Sigma}_{ab} \\ \boldsymbol{\Sigma}_{ba} & \boldsymbol{\Sigma}_{bb}\end{pmatrix}. \tag{3.74}$$

Making use of (3.60), we then have

$$\left(\boldsymbol{\Lambda}_{aa} - \boldsymbol{\Lambda}_{ab}\boldsymbol{\Lambda}_{bb}^{-1}\boldsymbol{\Lambda}_{ba}\right)^{-1} = \boldsymbol{\Sigma}_{aa}. \tag{3.75}$$

Telegram: @uni_k

www.konkur.in

Thus, we obtain the intuitively satisfying result that the marginal distribution $p(\mathbf{x}_a)$ has mean and covariance given by

$$\mathbb{E}[\mathbf{x}_a] = \boldsymbol{\mu}_a \tag{3.76}$$
$$\text{cov}[\mathbf{x}_a] = \boldsymbol{\Sigma}_{aa}. \tag{3.77}$$

We see that for a marginal distribution, the mean and covariance are most simply expressed in terms of the partitioned covariance matrix, in contrast to the conditional distribution for which the partitioned precision matrix gives rise to simpler expressions.

Our results for the marginal and conditional distributions of a partitioned Gaussian can be summarized as follows. Given a joint Gaussian distribution $\mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma})$ with $\boldsymbol{\Lambda} \equiv \boldsymbol{\Sigma}^{-1}$ and the following partitions

$$\mathbf{x} = \begin{pmatrix} \mathbf{x}_a \\ \mathbf{x}_b \end{pmatrix}, \quad \boldsymbol{\mu} = \begin{pmatrix} \boldsymbol{\mu}_a \\ \boldsymbol{\mu}_b \end{pmatrix} \tag{3.78}$$

$$\boldsymbol{\Sigma} = \begin{pmatrix} \boldsymbol{\Sigma}_{aa} & \boldsymbol{\Sigma}_{ab} \\ \boldsymbol{\Sigma}_{ba} & \boldsymbol{\Sigma}_{bb} \end{pmatrix}, \quad \boldsymbol{\Lambda} = \begin{pmatrix} \boldsymbol{\Lambda}_{aa} & \boldsymbol{\Lambda}_{ab} \\ \boldsymbol{\Lambda}_{ba} & \boldsymbol{\Lambda}_{bb} \end{pmatrix} \tag{3.79}$$

then the conditional distribution is given by

$$p(\mathbf{x}_a|\mathbf{x}_b) = \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_{a|b}, \boldsymbol{\Lambda}_{aa}^{-1}) \tag{3.80}$$
$$\boldsymbol{\mu}_{a|b} = \boldsymbol{\mu}_a - \boldsymbol{\Lambda}_{aa}^{-1}\boldsymbol{\Lambda}_{ab}(\mathbf{x}_b - \boldsymbol{\mu}_b) \tag{3.81}$$

and the marginal distribution is given by

$$p(\mathbf{x}_a) = \mathcal{N}(\mathbf{x}_a|\boldsymbol{\mu}_a, \boldsymbol{\Sigma}_{aa}). \tag{3.82}$$

We illustrate the idea of conditional and marginal distributions associated with a multivariate Gaussian using an example involving two variables in Figure 3.5.

3.2.6 Bayes' theorem

In Sections 3.2.4 and 3.2.5 we considered a Gaussian $p(\mathbf{x})$ in which we partitioned the vector $\mathbf{x}$ into two subvectors $\mathbf{x} = (\mathbf{x}_a, \mathbf{x}_b)$ and then found expressions for the conditional distribution $p(\mathbf{x}_a|\mathbf{x}_b)$ and the marginal distribution $p(\mathbf{x}_a)$. We noted that the mean of the conditional distribution $p(\mathbf{x}_a|\mathbf{x}_b)$ was a linear function of $\mathbf{x}_b$. Here we will suppose that we are given a Gaussian marginal distribution $p(\mathbf{x})$ and a Gaussian conditional distribution $p(\mathbf{y}|\mathbf{x})$ in which $p(\mathbf{y}|\mathbf{x})$ has a mean that is a linear function of $\mathbf{x}$ and a covariance that is independent of $\mathbf{x}$. This is an example of a *linear-Gaussian model* (Roweis and Ghahramani, 1999). We wish to find the marginal distribution $p(\mathbf{y})$ and the conditional distribution $p(\mathbf{x}|\mathbf{y})$. This is a structure that arises in several types of generative model and it will prove convenient to derive the general results here.

Section 11.1.4

Chapter 16

We will take the marginal and conditional distributions to be

$$p(\mathbf{x}) = \mathcal{N}\left(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Lambda}^{-1}\right) \tag{3.83}$$
$$p(\mathbf{y}|\mathbf{x}) = \mathcal{N}\left(\mathbf{y}|\mathbf{A}\mathbf{x} + \mathbf{b}, \mathbf{L}^{-1}\right) \tag{3.84}$$

Telegram: @uni_k

www.konkur.in

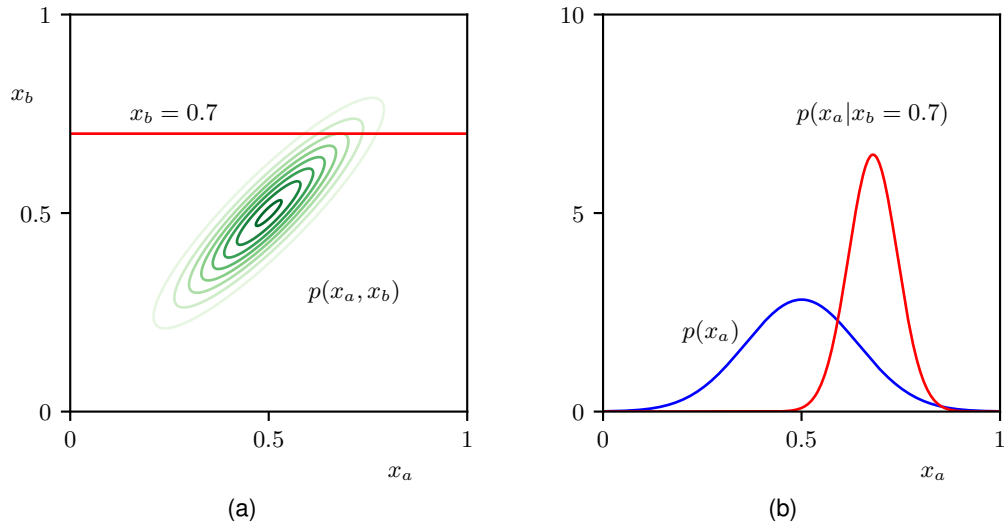


Figure 3.5 (a) Contours of a Gaussian distribution $p(x_a, x_b)$ over two variables. (b) The marginal distribution $p(x_a)$ (blue curve) and the conditional distribution $p(x_a|x_b)$ for $x_b = 0.7$ (red curve).

where $\boldsymbol{\mu}$, $\mathbf{A}$, and $\mathbf{b}$ are parameters governing the means, and $\boldsymbol{\Lambda}$ and $\mathbf{L}$ are precision matrices. If $\mathbf{x}$ has dimensionality M and $\mathbf{y}$ has dimensionality D, then the matrix $\mathbf{A}$ has size $D \times M$.

First we find an expression for the joint distribution over $\mathbf{x}$ and $\mathbf{y}$. To do this, we define

$$\mathbf{z} = \begin{pmatrix} \mathbf{x} \\ \mathbf{y} \end{pmatrix} \tag{3.85}$$

and then consider the log of the joint distribution:

$$\begin{aligned} \ln p(\mathbf{z}) &= \ln p(\mathbf{x}) + \ln p(\mathbf{y}|\mathbf{x}) \\ &= -\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu})^{\mathrm{T}} \boldsymbol{\Lambda} (\mathbf{x} - \boldsymbol{\mu}) \\ &\quad -\frac{1}{2}(\mathbf{y} - \mathbf{A}\mathbf{x} - \mathbf{b})^{\mathrm{T}} \mathbf{L} (\mathbf{y} - \mathbf{A}\mathbf{x} - \mathbf{b}) + \text{const} \end{aligned} \tag{3.86}$$

where 'const' denotes terms independent of $\mathbf{x}$ and $\mathbf{y}$. As before, we see that this is a quadratic function of the components of $\mathbf{z}$, and hence, $p(\mathbf{z})$ is Gaussian distribution. To find the precision of this Gaussian, we consider the second-order terms in (3.86), which can be written as

$$\begin{aligned} &-\frac{1}{2}\mathbf{x}^{\mathrm{T}}(\boldsymbol{\Lambda} + \mathbf{A}^{\mathrm{T}}\mathbf{L}\mathbf{A})\mathbf{x} - \frac{1}{2}\mathbf{y}^{\mathrm{T}}\mathbf{L}\mathbf{y} + \frac{1}{2}\mathbf{y}^{\mathrm{T}}\mathbf{L}\mathbf{A}\mathbf{x} + \frac{1}{2}\mathbf{x}^{\mathrm{T}}\mathbf{A}^{\mathrm{T}}\mathbf{L}\mathbf{y} \\ &= -\frac{1}{2}\begin{pmatrix} \mathbf{x} \\ \mathbf{y} \end{pmatrix}^{\mathrm{T}} \begin{pmatrix} \boldsymbol{\Lambda} + \mathbf{A}^{\mathrm{T}}\mathbf{L}\mathbf{A} & -\mathbf{A}^{\mathrm{T}}\mathbf{L} \\ -\mathbf{L}\mathbf{A} & \mathbf{L} \end{pmatrix} \begin{pmatrix} \mathbf{x} \\ \mathbf{y} \end{pmatrix} = -\frac{1}{2}\mathbf{z}^{\mathrm{T}}\mathbf{R}\mathbf{z} \end{aligned} \tag{3.87}$$

and so the Gaussian distribution over $\mathbf{z}$ has precision (inverse covariance) matrix

Telegram: @uni_k

www.konkur.in

given by

$$\mathbf{R} = \begin{pmatrix} \boldsymbol{\Lambda} + \mathbf{A}^{\mathrm{T}}\mathbf{L}\mathbf{A} & -\mathbf{A}^{\mathrm{T}}\mathbf{L} \\ -\mathbf{L}\mathbf{A} & \mathbf{L} \end{pmatrix}. \tag{3.88}$$

The covariance matrix is found by taking the inverse of the precision, which can be done using the matrix inversion formula (3.60) to give

Exercise 3.23

$$\mathrm{cov}[\mathbf{z}] = \mathbf{R}^{-1} = \begin{pmatrix} \boldsymbol{\Lambda}^{-1} & \boldsymbol{\Lambda}^{-1}\mathbf{A}^{\mathrm{T}} \\ \mathbf{A}\boldsymbol{\Lambda}^{-1} & \mathbf{L}^{-1} + \mathbf{A}\boldsymbol{\Lambda}^{-1}\mathbf{A}^{\mathrm{T}} \end{pmatrix}. \tag{3.89}$$

Similarly, we can find the mean of the Gaussian distribution over $\mathbf{z}$ by identifying the linear terms in (3.86), which are given by

$$\mathbf{x}^{\mathrm{T}}\boldsymbol{\Lambda}\boldsymbol{\mu} - \mathbf{x}^{\mathrm{T}}\mathbf{A}^{\mathrm{T}}\mathbf{L}\mathbf{b} + \mathbf{y}^{\mathrm{T}}\mathbf{L}\mathbf{b} = \begin{pmatrix} \mathbf{x} \\ \mathbf{y} \end{pmatrix}^{\mathrm{T}} \begin{pmatrix} \boldsymbol{\Lambda}\boldsymbol{\mu} - \mathbf{A}^{\mathrm{T}}\mathbf{L}\mathbf{b} \\ \mathbf{L}\mathbf{b} \end{pmatrix}. \tag{3.90}$$

Using our earlier result (3.55) obtained by completing the square over the quadratic form of a multivariate Gaussian, we find that the mean of $\mathbf{z}$ is given by

$$\mathbb{E}[\mathbf{z}] = \mathbf{R}^{-1} \begin{pmatrix} \boldsymbol{\Lambda}\boldsymbol{\mu} - \mathbf{A}^{\mathrm{T}}\mathbf{L}\mathbf{b} \\ \mathbf{L}\mathbf{b} \end{pmatrix}. \tag{3.91}$$

Exercise 3.24

Making use of (3.89), we then obtain

$$\mathbb{E}[\mathbf{z}] = \begin{pmatrix} \boldsymbol{\mu} \\ \mathbf{A}\boldsymbol{\mu} + \mathbf{b} \end{pmatrix}. \tag{3.92}$$

Next we find an expression for the marginal distribution $p(\mathbf{y})$ in which we have marginalized over $\mathbf{x}$. Recall that the marginal distribution over a subset of the components of a Gaussian random vector takes a particularly simple form when expressed in terms of the partitioned covariance matrix. Specifically, its mean and covariance are given by (3.76) and (3.77), respectively. Making use of (3.89) and (3.92), we see that the mean and covariance of the marginal distribution $p(\mathbf{y})$ are given by

Section 3.2

$$\mathbb{E}[\mathbf{y}] = \mathbf{A}\boldsymbol{\mu} + \mathbf{b} \tag{3.93}$$

$$\mathrm{cov}[\mathbf{y}] = \mathbf{L}^{-1} + \mathbf{A}\boldsymbol{\Lambda}^{-1}\mathbf{A}^{\mathrm{T}}. \tag{3.94}$$

A special case of this result is when $\mathbf{A} = \mathbf{I}$, in which case the marginal distribution reduces to the convolution of two Gaussians, for which we see that the mean of the convolution is the sum of the means of the two Gaussians and the covariance of the convolution is the sum of their covariances.

Finally, we seek an expression for the conditional $p(\mathbf{x}|\mathbf{y})$. Recall that the results for the conditional distribution are most easily expressed in terms of the partitioned precision matrix, using (3.57) and (3.59). Applying these results to (3.89) and (3.92), we see that the conditional distribution $p(\mathbf{x}|\mathbf{y})$ has mean and covariance given by

Section 3.2

$$\mathbb{E}[\mathbf{x}|\mathbf{y}] = (\boldsymbol{\Lambda} + \mathbf{A}^{\mathrm{T}}\mathbf{L}\mathbf{A})^{-1}\left\{\mathbf{A}^{\mathrm{T}}\mathbf{L}(\mathbf{y} - \mathbf{b}) + \boldsymbol{\Lambda}\boldsymbol{\mu}\right\} \tag{3.95}$$

$$\mathrm{cov}[\mathbf{x}|\mathbf{y}] = (\boldsymbol{\Lambda} + \mathbf{A}^{\mathrm{T}}\mathbf{L}\mathbf{A})^{-1}. \tag{3.96}$$

Telegram: @uni_k

www.konkur.in

The evaluation of this conditional distribution can be seen as an example of Bayes' theorem, in which we interpret $p(\mathbf{x})$ as a prior distribution over $\mathbf{x}$. If the variable $\mathbf{y}$ is observed, then the conditional distribution $p(\mathbf{x}|\mathbf{y})$ represents the corresponding posterior distribution over $\mathbf{x}$. Having found the marginal and conditional distributions, we have effectively expressed the joint distribution $p(\mathbf{z}) = p(\mathbf{x})p(\mathbf{y}|\mathbf{x})$ in the form $p(\mathbf{x}|\mathbf{y})p(\mathbf{y})$.

These results can be summarized as follows. Given a marginal Gaussian distribution for $\mathbf{x}$ and a conditional Gaussian distribution for $\mathbf{y}$ given $\mathbf{x}$ in the form

$$p(\mathbf{x}) = \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Lambda}^{-1}) \tag{3.97}$$

$$p(\mathbf{y}|\mathbf{x}) = \mathcal{N}(\mathbf{y}|\mathbf{A}\mathbf{x} + \mathbf{b}, \mathbf{L}^{-1}), \tag{3.98}$$

then the marginal distribution of $\mathbf{y}$ and the conditional distribution of $\mathbf{x}$ given $\mathbf{y}$ are given by

$$p(\mathbf{y}) = \mathcal{N}(\mathbf{y}|\mathbf{A}\boldsymbol{\mu} + \mathbf{b}, \mathbf{L}^{-1} + \mathbf{A}\boldsymbol{\Lambda}^{-1}\mathbf{A}^{\mathrm{T}}) \tag{3.99}$$

$$p(\mathbf{x}|\mathbf{y}) = \mathcal{N}(\mathbf{x}|\boldsymbol{\Sigma}\{\mathbf{A}^{\mathrm{T}}\mathbf{L}(\mathbf{y} - \mathbf{b}) + \boldsymbol{\Lambda}\boldsymbol{\mu}\}, \boldsymbol{\Sigma}) \tag{3.100}$$

where

$$\boldsymbol{\Sigma} = (\boldsymbol{\Lambda} + \mathbf{A}^{\mathrm{T}}\mathbf{L}\mathbf{A})^{-1}. \tag{3.101}$$

3.2.7 Maximum likelihood

Given a data set $\mathbf{X} = (\mathbf{x}_1, \ldots, \mathbf{x}_N)^{\mathrm{T}}$ in which the observations $\{\mathbf{x}_n\}$ are assumed to be drawn independently from a multivariate Gaussian distribution, we can estimate the parameters of the distribution by maximum likelihood. The log likelihood function is given by

$$\ln p(\mathbf{X}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) = -\frac{ND}{2}\ln(2\pi) - \frac{N}{2}\ln|\boldsymbol{\Sigma}| - \frac{1}{2}\sum_{n=1}^{N}(\mathbf{x}_n - \boldsymbol{\mu})^{\mathrm{T}}\boldsymbol{\Sigma}^{-1}(\mathbf{x}_n - \boldsymbol{\mu}). \tag{3.102}$$

By simple rearrangement, we see that the likelihood function depends on the data set only through the two quantities

$$\sum_{n=1}^{N}\mathbf{x}_n, \qquad \sum_{n=1}^{N}\mathbf{x}_n\mathbf{x}_n^{\mathrm{T}}. \tag{3.103}$$

These are known as the *sufficient statistics* for the Gaussian distribution. Using

Appendix A

(A.19), the derivative of the log likelihood with respect to $\boldsymbol{\mu}$ is given by

$$\frac{\partial}{\partial\boldsymbol{\mu}}\ln p(\mathbf{X}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) = \sum_{n=1}^{N}\boldsymbol{\Sigma}^{-1}(\mathbf{x}_n - \boldsymbol{\mu}), \tag{3.104}$$

and setting this derivative to zero, we obtain the solution for the maximum likelihood estimate of the mean:

$$\boldsymbol{\mu}_{\mathrm{ML}} = \frac{1}{N}\sum_{n=1}^{N}\mathbf{x}_n, \tag{3.105}$$

Telegram: @uni_k

www.konkur.in

which is the mean of the observed set of data points. The maximization of (3.102) with respect to $\boldsymbol{\Sigma}$ is rather more involved. The simplest approach is to ignore the symmetry constraint and show that the resulting solution is symmetric as required. Alternative derivations of this result, which impose the symmetry and positive definiteness constraints explicitly, can be found in Magnus and Neudecker (1999). The result is as expected and takes the form

Exercise 3.28

$$\boldsymbol{\Sigma}_{\mathrm{ML}} = \frac{1}{N}\sum_{n=1}^{N}(\mathbf{x}_n - \boldsymbol{\mu}_{\mathrm{ML}})(\mathbf{x}_n - \boldsymbol{\mu}_{\mathrm{ML}})^{\mathrm{T}}, \tag{3.106}$$

which involves $\boldsymbol{\mu}_{\mathrm{ML}}$ because this is the result of a joint maximization with respect to $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$. Note that the solution (3.105) for $\boldsymbol{\mu}_{\mathrm{ML}}$ does not depend on $\boldsymbol{\Sigma}_{\mathrm{ML}}$, and so we can first evaluate $\boldsymbol{\mu}_{\mathrm{ML}}$ and then use this to evaluate $\boldsymbol{\Sigma}_{\mathrm{ML}}$.

If we evaluate the expectations of the maximum likelihood solutions under the true distribution, we obtain the following results

Exercise 3.29

$$\mathbb{E}[\boldsymbol{\mu}_{\mathrm{ML}}] = \boldsymbol{\mu} \tag{3.107}$$

$$\mathbb{E}[\boldsymbol{\Sigma}_{\mathrm{ML}}] = \frac{N-1}{N}\boldsymbol{\Sigma}. \tag{3.108}$$

We see that the expectation of the maximum likelihood estimate for the mean is equal to the true mean. However, the maximum likelihood estimate for the covariance has an expectation that is less than the true value, and hence, it is biased. We can correct this bias by defining a different estimator $\widetilde{\boldsymbol{\Sigma}}$ given by

$$\widetilde{\boldsymbol{\Sigma}} = \frac{1}{N-1}\sum_{n=1}^{N}(\mathbf{x}_n - \boldsymbol{\mu}_{\mathrm{ML}})(\mathbf{x}_n - \boldsymbol{\mu}_{\mathrm{ML}})^{\mathrm{T}}. \tag{3.109}$$

Clearly from (3.106) and (3.108), the expectation of $\widetilde{\boldsymbol{\Sigma}}$ is equal to $\boldsymbol{\Sigma}$.

3.2.8 Sequential estimation

Our discussion of the maximum likelihood solution represents a *batch* method in which the entire training data set is considered at once. An alternative is to use *sequential* methods, which allow data points to be processed one at a time and then discarded. These are important for online applications and for large data when the batch processing of all data points at once is infeasible.

Consider the result (3.105) for the maximum likelihood estimator of the mean $\boldsymbol{\mu}_{\mathrm{ML}}$, which we will denote by $\boldsymbol{\mu}_{\mathrm{ML}}^{(N)}$ when it is based on N observations. If we

Telegram: @uni_k

www.konkur.in

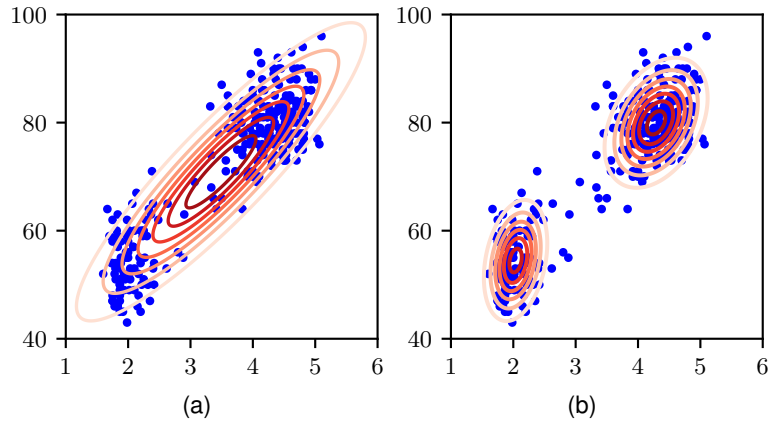


Figure 3.6 Plots of the Old Faithful data in which the red curves are contours of constant probability density. (a) A single Gaussian distribution which has been fitted to the data using maximum likelihood. Note that this distribution fails to capture the two clumps in the data and indeed places much of its probability mass in the central region between the clumps where the data are relatively sparse. (b) The distribution given by a linear combination of two Gaussians, also fitted by maximum likelihood, which gives a better representation of the data.

dissect out the contribution from the final data point $\mathbf{x}_N$, we obtain

$$\begin{aligned}
\boldsymbol{\mu}_{\mathrm{ML}}^{(N)} &= \frac{1}{N}\sum_{n=1}^{N}\mathbf{x}_n \\
&= \frac{1}{N}\mathbf{x}_N + \frac{1}{N}\sum_{n=1}^{N-1}\mathbf{x}_n \\
&= \frac{1}{N}\mathbf{x}_N + \frac{N-1}{N}\boldsymbol{\mu}_{\mathrm{ML}}^{(N-1)} \\
&= \boldsymbol{\mu}_{\mathrm{ML}}^{(N-1)} + \frac{1}{N}(\mathbf{x}_N - \boldsymbol{\mu}_{\mathrm{ML}}^{(N-1)}).
\end{aligned} \tag{3.110}$$

This result has a nice interpretation, as follows. After observing $N-1$ data points, we estimate $\boldsymbol{\mu}$ by $\boldsymbol{\mu}_{\mathrm{ML}}^{(N-1)}$. We now observe data point $\mathbf{x}_N$, and we obtain our revised estimate $\boldsymbol{\mu}_{\mathrm{ML}}^{(N)}$ by moving the old estimate a small amount, proportional to $1/N$, in the direction of the 'error signal' $(\mathbf{x}_N - \boldsymbol{\mu}_{\mathrm{ML}}^{(N-1)})$. Note that, as N increases, so the contributions from successive data points get smaller.

3.2.9 Mixtures of Gaussians

Although the Gaussian distribution has some important analytical properties, it suffers from significant limitations when used to model modelling real data sets. Consider the example shown in Figure 3.6(a). This is known as the 'Old Faithful' data set, and comprises 272 measurements of the eruption of the Old Faithful geyser in Yellowstone National Park in the USA. Each measurement gives the duration of the eruption in minutes (horizontal axis) and the time in minutes to the next eruption (vertical axis). We see that the data set forms two dominant clumps, and that a simple Gaussian distribution is unable to capture this structure.

We might expect that a superposition of two Gaussian distributions would be able to do a much better job of representing the structure in this data set, and indeed

Telegram: @uni_k

www.konkur.in

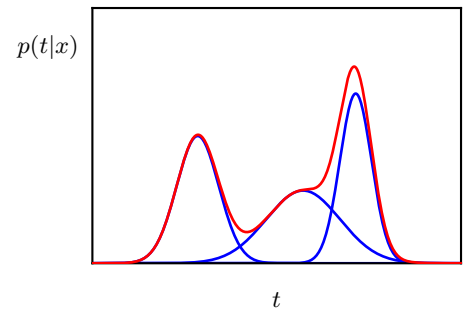


Figure 3.7 Example of a Gaussian mixture distribution in one dimension showing three Gaussians (each scaled by a coefficient) in blue and their sum in red.

this proves to be the case, as can be seen from Figure 3.6(b). Such superpositions, formed by taking linear combinations of more basic distributions such as Gaussians, can be formulated as probabilistic models known as *mixture distributions*. In this section we will consider Gaussians to illustrate the framework of mixture models. More generally, mixture models can comprise linear combinations of other distributions, for example mixtures of Bernoulli distributions for binary variables. In Figure 3.7 we see that a linear combination of Gaussians can give rise to very complex densities. By using a sufficient number of Gaussians and by adjusting their means and covariances as well as the coefficients in the linear combination, almost any continuous distribution can be approximated to arbitrary accuracy.

Chapter 15

We therefore consider a superposition of K Gaussian densities of the form

$$p(\mathbf{x}) = \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k), \tag{3.111}$$

which is called a *mixture of Gaussians*. Each Gaussian density $\mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$ is called a *component* of the mixture and has its own mean $\boldsymbol{\mu}_k$ and covariance $\boldsymbol{\Sigma}_k$. Contour and surface plots for a Gaussian mixture in two dimensions having three components are shown in Figure 3.8.

The parameters π_k in (3.111) are called *mixing coefficients*. If we integrate both sides of (3.111) with respect to $\mathbf{x}$, and note that both $p(\mathbf{x})$ and the individual Gaussian components are normalized, we obtain

$$\sum_{k=1}^{K} \pi_k = 1. \tag{3.112}$$

Also, given that $\mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \geqslant 0$, a sufficient condition for the requirement $p(\mathbf{x}) \geqslant 0$ is that $\pi_k \geqslant 0$ for all k. Combining this with the condition (3.112), we obtain

$$0 \leqslant \pi_k \leqslant 1. \tag{3.113}$$

We can therefore see that the mixing coefficients satisfy the requirements to be probabilities, and we will show that this probabilistic interpretation of mixture distributions is very powerful.

Chapter 15

Telegram: @uni_k

www.konkur.in

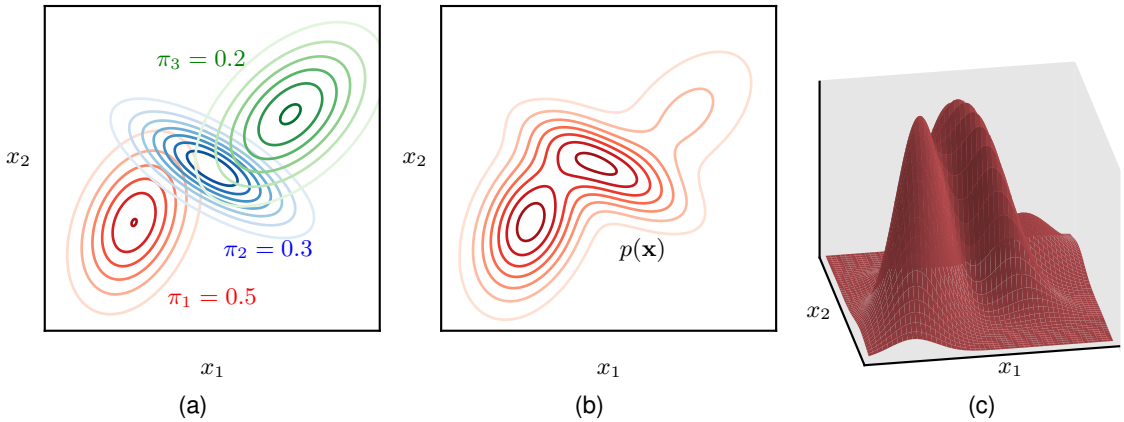


Figure 3.8 Illustration of a mixture of three Gaussians in a two-dimensional space. (a) Contours of constant density for each of the mixture components, in which the three components are denoted red, blue, and green, and the values of the mixing coefficients are shown below each component. (b) Contours of the marginal probability density $p(\mathbf{x})$ of the mixture distribution. (c) A surface plot of the distribution $p(\mathbf{x})$.

From the sum and product rules of probability, the marginal density can be written as

$$p(\mathbf{x}) = \sum_{k=1}^{K} p(k)p(\mathbf{x}|k), \tag{3.114}$$

which is equivalent to (3.111) in which we can view $\pi_k = p(k)$ as the prior probability of picking the kth component, and the density $\mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) = p(\mathbf{x}|k)$ as the probability of $\mathbf{x}$ conditioned on k. As we will see in later chapters, an important role is played by the corresponding posterior probabilities $p(k|\mathbf{x})$, which are also known as *responsibilities*. From Bayes' theorem, these are given by

$$\begin{aligned} \gamma_k(\mathbf{x}) &\equiv p(k|\mathbf{x}) \\ &= \frac{p(k)p(\mathbf{x}|k)}{\sum_l p(l)p(\mathbf{x}|l)} \\ &= \frac{\pi_k \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\sum_l \pi_l \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_l, \boldsymbol{\Sigma}_l)}. \end{aligned} \tag{3.115}$$

The form of the Gaussian mixture distribution is governed by the parameters $\boldsymbol{\pi}$, $\boldsymbol{\mu}$, and $\boldsymbol{\Sigma}$, where we have used the notation $\boldsymbol{\pi} \equiv \{\pi_1, \ldots, \pi_K\}$, $\boldsymbol{\mu} \equiv \{\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_K\}$, and $\boldsymbol{\Sigma} \equiv \{\boldsymbol{\Sigma}_1, \ldots \boldsymbol{\Sigma}_K\}$. One way to set the values of these parameters is to use maximum likelihood. From (3.111), the log of the likelihood function is given by

$$\ln p(\mathbf{X}|\boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \sum_{n=1}^{N} \ln \left\{ \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \right\} \tag{3.116}$$

where $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$. We immediately see that the situation is now much more complex than with a single Gaussian, due to the summation over k inside the log-

Telegram: @uni_k

www.konkur.in

arithm. As a result, the maximum likelihood solution for the parameters no longer has a closed-form analytical solution. One approach for maximizing the likelihood function is to use iterative numerical optimization techniques. Alternatively, we can employ a powerful framework called *expectation maximization*, which has wide applicability to a variety of different deep generative models.

Chapter 15

3.3. Periodic Variables

Although Gaussian distributions are of great practical significance, both in their own right and as building blocks for more complex probabilistic models, there are situations in which they are inappropriate as density models for continuous variables. One important case, which arises in practical applications, is that of periodic variables.

An example of a periodic variable is the wind direction at a particular geographical location. We might, for instance, measure the wind direction at multiple locations and wish to summarize this data using a parametric distribution. Another example is calendar time, where we may be interested in modelling quantities that are believed to be periodic over 24 hours or over an annual cycle. Such quantities can conveniently be represented using an angular (polar) coordinate $0 \leqslant \theta < 2\pi$.

We might be tempted to treat periodic variables by choosing some direction as the origin and then applying a conventional distribution such as the Gaussian. Such an approach, however, would give results that were strongly dependent on the arbitrary choice of origin. Suppose, for instance, that we have two observations at $\theta_1 = 1^\circ$ and $\theta_2 = 359^\circ$, and we model them using a standard univariate Gaussian distribution. If we place the origin at 0°, then the sample mean of this data set will be 180° with standard deviation 179°, whereas if we place the origin at 180°, then the mean will be 0° and the standard deviation will be 1°. We clearly need to develop a special approach for periodic variables.

3.3.1 Von Mises distribution

Let us consider the problem of evaluating the mean of a set of observations $\mathcal{D} = \{\theta_1, \ldots, \theta_N\}$ of a periodic variable θ where θ is measured in radians. We have already seen that the simple average $(\theta_1 + \cdots + \theta_N)/N$ will be strongly coordinate dependent. To find an invariant measure of the mean, note that the observations can be viewed as points on the unit circle and can therefore be described instead by two-dimensional unit vectors $\mathbf{x}_1, \ldots, \mathbf{x}_N$ where $\|\mathbf{x}_n\| = 1$ for $n = 1, \ldots, N$, as illustrated in Figure 3.9. We can average the vectors $\{\mathbf{x}_n\}$ instead to give

$$\overline{\mathbf{x}} = \frac{1}{N} \sum_{n=1}^{N} \mathbf{x}_n \tag{3.117}$$

and then find the corresponding angle $\overline{\theta}$ of this average. Clearly, this definition will ensure that the location of the mean is independent of the origin of the angular coordinate. Note that $\overline{\mathbf{x}}$ will typically lie inside the unit circle. The Cartesian coordinates

Telegram: @uni_k

www.konkur.in

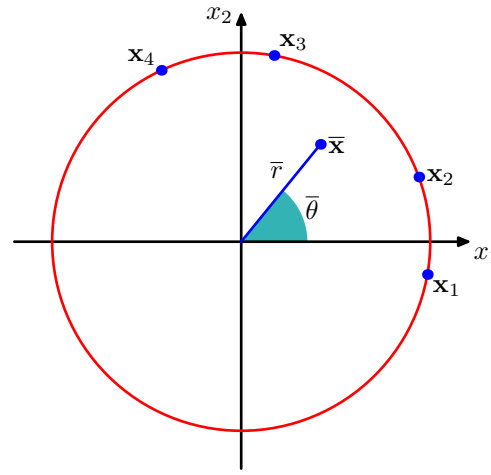


Figure 3.9 Illustration of the representation of values θ_n of a periodic variable as two-dimensional vectors $\mathbf{x}_n$ living on the unit circle. Also shown is the average $\overline{\mathbf{x}}$ of those vectors.

of the observations are given by $\mathbf{x}_n = (\cos\theta_n, \sin\theta_n)$, and we can write the Cartesian coordinates of the sample mean in the form $\overline{\mathbf{x}} = (\overline{r}\cos\overline{\theta}, \overline{r}\sin\overline{\theta})$. Substituting into (3.117) and equating the x_1 and x_2 components then gives

$$\overline{x}_1 = \overline{r}\cos\overline{\theta} = \frac{1}{N}\sum_{n=1}^{N}\cos\theta_n, \qquad \overline{x}_2 = \overline{r}\sin\overline{\theta} = \frac{1}{N}\sum_{n=1}^{N}\sin\theta_n. \tag{3.118}$$

Taking the ratio, and using the identity $\tan\theta = \sin\theta/\cos\theta$, we can solve for $\overline{\theta}$ to give

$$\overline{\theta} = \tan^{-1}\left\{\frac{\sum_n \sin\theta_n}{\sum_n \cos\theta_n}\right\}. \tag{3.119}$$

Shortly, we will see how this result arises naturally as a maximum likelihood estimator.

First, we need to define a periodic generalization of the Gaussian called the *von Mises* distribution. Here we will limit our attention to univariate distributions, although analogous periodic distributions can also be found over hyperspheres of arbitrary dimension (Mardia and Jupp, 2000).

By convention, we will consider distributions $p(\theta)$ that have period 2π. Any probability density $p(\theta)$ defined over θ must not only be non-negative and integrate to one, but it must also be periodic. Thus, $p(\theta)$ must satisfy the three conditions:

$$p(\theta) \geqslant 0 \tag{3.120}$$

$$\int_0^{2\pi} p(\theta)\,\mathrm{d}\theta = 1 \tag{3.121}$$

$$p(\theta + 2\pi) = p(\theta). \tag{3.122}$$

From (3.122), it follows that $p(\theta + M2\pi) = p(\theta)$ for any integer M.

We can easily obtain a Gaussian-like distribution that satisfies these three properties as follows. Consider a Gaussian distribution over two variables $\mathbf{x} = (x_1, x_2)$

Telegram: @uni_k

www.konkur.in

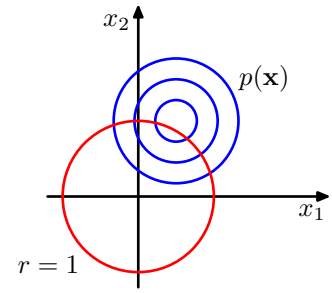


Figure 3.10 The von Mises distribution can be derived by considering a two-dimensional Gaussian of the form (3.123), whose density contours are shown in blue, and conditioning on the unit circle shown in red.

having mean $\boldsymbol{\mu} = (\mu_1, \mu_2)$ and a covariance matrix $\boldsymbol{\Sigma} = \sigma^2\mathbf{I}$ where $\mathbf{I}$ is the 2×2 identity matrix, so that

$$p(x_1, x_2) = \frac{1}{2\pi\sigma^2} \exp\left\{-\frac{(x_1 - \mu_1)^2 + (x_2 - \mu_2)^2}{2\sigma^2}\right\}. \tag{3.123}$$

The contours of constant $p(\mathbf{x})$ are circles, as illustrated in Figure 3.10.

Now suppose we consider the value of this distribution along a circle of fixed radius. Then by construction, this distribution will be periodic, although it will not be normalized. We can determine the form of this distribution by transforming from Cartesian coordinates (x_1, x_2) to polar coordinates (r, θ) so that

$$x_1 = r\cos\theta, \qquad x_2 = r\sin\theta. \tag{3.124}$$

We also map the mean $\boldsymbol{\mu}$ into polar coordinates by writing

$$\mu_1 = r_0\cos\theta_0, \qquad \mu_2 = r_0\sin\theta_0. \tag{3.125}$$

Next we substitute these transformations into the two-dimensional Gaussian distribution (3.123), and then condition on the unit circle $r = 1$, noting that we are interested only in the dependence on θ. Focusing on the exponent in the Gaussian distribution we have

$$\begin{aligned} &-\frac{1}{2\sigma^2}\left\{(r\cos\theta - r_0\cos\theta_0)^2 + (r\sin\theta - r_0\sin\theta_0)^2\right\} \\ &= -\frac{1}{2\sigma^2}\left\{1 + r_0^2 - 2r_0\cos\theta\cos\theta_0 - 2r_0\sin\theta\sin\theta_0\right\} \\ &= \frac{r_0}{\sigma^2}\cos(\theta - \theta_0) + \text{const} \end{aligned} \tag{3.126}$$

where 'const' denotes terms independent of θ. We have made use of the following trigonometrical identities:

$$\cos^2 A + \sin^2 A = 1 \tag{3.127}$$

$$\cos A\cos B + \sin A\sin B = \cos(A - B). \tag{3.128}$$

If we now define $m = r_0/\sigma^2$, we obtain our final expression for the distribution of $p(\theta)$ along the unit circle $r = 1$ in the form

$$p(\theta|\theta_0, m) = \frac{1}{2\pi I_0(m)} \exp\{m\cos(\theta - \theta_0)\}, \tag{3.129}$$

Telegram: @uni_k

www.konkur.in

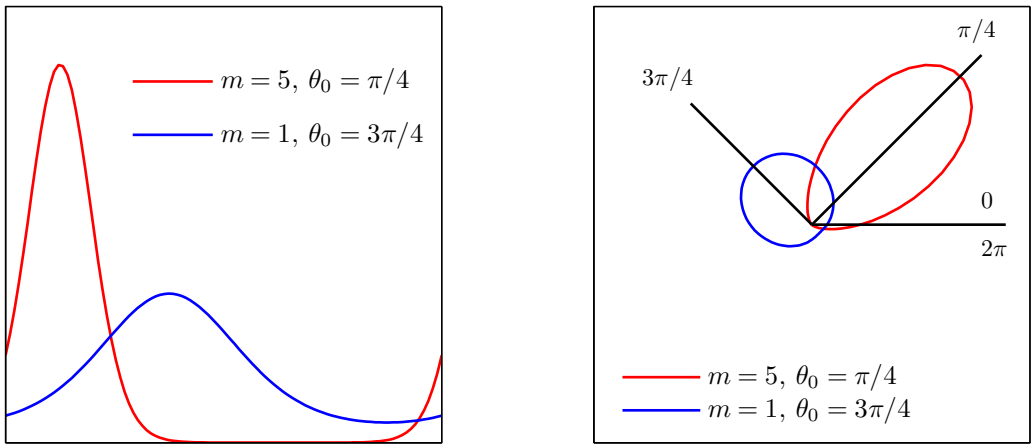


Figure 3.11 The von Mises distribution plotted for two different parameter values, shown as a Cartesian plot on the left and as the corresponding polar plot on the right.

which is called the *von Mises* distribution or the *circular normal*. Here the parameter θ_0 corresponds to the mean of the distribution, whereas m, which is known as the *concentration* parameter, is analogous to the inverse variance (i.e. the precision) for the Gaussian. The normalization coefficient in (3.129) is expressed in terms of $I_0(m)$, which is the zeroth-order modified Bessel function of the first kind (Abramowitz and Stegun, 1965) and is defined by

$$I_0(m) = \frac{1}{2\pi}\int_0^{2\pi} \exp\{m\cos\theta\}\,\mathrm{d}\theta. \tag{3.130}$$

Exercise 3.31

For large m, the distribution becomes approximately Gaussian. The von Mises distribution is plotted in Figure 3.11, and the function $I_0(m)$ is plotted in Figure 3.12.

Now consider the maximum likelihood estimators for the parameters θ_0 and m for the von Mises distribution. The log likelihood function is given by

$$\ln p(\mathcal{D}|\theta_0, m) = -N\ln(2\pi) - N\ln I_0(m) + m\sum_{n=1}^{N}\cos(\theta_n - \theta_0). \tag{3.131}$$

Setting the derivative with respect to θ_0 equal to zero gives

$$\sum_{n=1}^{N}\sin(\theta_n - \theta_0) = 0. \tag{3.132}$$

To solve for θ_0, we make use of the trigonometric identity

$$\sin(A - B) = \cos B\sin A - \cos A\sin B \tag{3.133}$$

Exercise 3.32

from which we obtain

Telegram: @uni_k

www.konkur.in

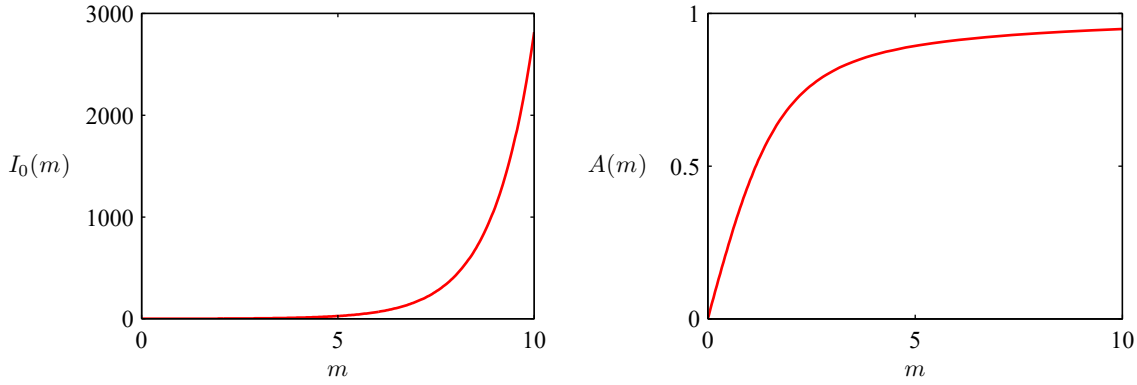


Figure 3.12 Plot of the Bessel function $I_0(m)$ defined by (3.130), together with the function $A(m)$ defined by (3.136).

$$\theta_0^{\mathrm{ML}} = \tan^{-1}\left\{\frac{\sum_n \sin\theta_n}{\sum_n \cos\theta_n}\right\}, \tag{3.134}$$

which we recognize as the result (3.119) obtained earlier for the mean of the observations viewed in a two-dimensional Cartesian space.

Similarly, maximizing (3.131) with respect to m and making use of $I_0'(m) = I_1(m)$ (Abramowitz and Stegun, 1965), we have

$$A(m_{\mathrm{ML}}) = \frac{1}{N}\sum_{n=1}^{N}\cos(\theta_n - \theta_0^{\mathrm{ML}}) \tag{3.135}$$

where we have substituted for the maximum likelihood solution for θ_0^{ML} (recalling that we are performing a joint optimization over θ and m), and we have defined

$$A(m) = \frac{I_1(m)}{I_0(m)}. \tag{3.136}$$

The function $A(m)$ is plotted in Figure 3.12. Making use of the trigonometric identity (3.128), we can write (3.135) in the form

$$A(m_{\mathrm{ML}}) = \left(\frac{1}{N}\sum_{n=1}^{N}\cos\theta_n\right)\cos\theta_0^{\mathrm{ML}} + \left(\frac{1}{N}\sum_{n=1}^{N}\sin\theta_n\right)\sin\theta_0^{\mathrm{ML}}. \tag{3.137}$$

The right-hand side of (3.137) is easily evaluated, and the function $A(m)$ can be inverted numerically. One limitation of the von Mises distribution is that it is unimodal. By forming *mixtures* of von Mises distributions, we obtain a flexible framework for modelling periodic variables that can handle multimodality.

For completeness, we mention briefly some alternative techniques for constructing periodic distributions. The simplest approach is to use a histogram of observations in which the angular coordinate is divided into fixed bins. This has the virtue of

Telegram: @uni_k

www.konkur.in

simplicity and flexibility but also suffers from significant limitations, as we will see when we discuss histogram methods in more detail later. Another approach starts, like the von Mises distribution, from a Gaussian distribution over a Euclidean space but now marginalizes onto the unit circle rather than conditioning (Mardia and Jupp, 2000). However, this leads to more complex forms of distribution and will not be discussed further. Finally, any valid distribution over the real axis (such as a Gaussian) can be turned into a periodic distribution by mapping successive intervals of width 2π onto the periodic variable $(0, 2\pi)$, which corresponds to 'wrapping' the real axis around the unit circle. Again, the resulting distribution is more complex to handle than the von Mises distribution.

Section 3.5

3.4. The Exponential Family

The probability distributions that we have studied so far in this chapter (with the exception of mixture models) are specific examples of a broad class of distributions called the *exponential family* (Duda and Hart, 1973; Bernardo and Smith, 1994). Members of the exponential family have many important properties in common, and it is illuminating to discuss these properties in some generality.

The exponential family of distributions over $\mathbf{x}$, given parameters $\boldsymbol{\eta}$, is defined to be the set of distributions of the form

$$p(\mathbf{x}|\boldsymbol{\eta}) = h(\mathbf{x})g(\boldsymbol{\eta}) \exp\left\{\boldsymbol{\eta}^{\mathrm{T}}\mathbf{u}(\mathbf{x})\right\} \tag{3.138}$$

where $\mathbf{x}$ may be scalar or vector and may be discrete or continuous. Here $\boldsymbol{\eta}$ are called the *natural parameters* of the distribution, and $\mathbf{u}(\mathbf{x})$ is some function of $\mathbf{x}$. The function $g(\boldsymbol{\eta})$ can be interpreted as the coefficient that ensures that the distribution is normalized, and therefore, it satisfies

$$g(\boldsymbol{\eta}) \int h(\mathbf{x}) \exp\left\{\boldsymbol{\eta}^{\mathrm{T}}\mathbf{u}(\mathbf{x})\right\} \, \mathrm{d}\mathbf{x} = 1 \tag{3.139}$$

where the integration is replaced by summation if $\mathbf{x}$ is a discrete variable.

We begin by taking some examples of the distributions introduced earlier in the chapter and showing that they are indeed members of the exponential family. Consider first the Bernoulli distribution:

$$p(x|\mu) = \mathrm{Bern}(x|\mu) = \mu^x(1-\mu)^{1-x}. \tag{3.140}$$

Expressing the right-hand side as the exponential of the logarithm, we have

$$\begin{aligned} p(x|\mu) &= \exp\{x \ln \mu + (1-x)\ln(1-\mu)\} \\ &= (1-\mu)\exp\left\{\ln\left(\frac{\mu}{1-\mu}\right)x\right\}. \end{aligned} \tag{3.141}$$

Comparison with (3.138) allows us to identify

$$\eta = \ln\left(\frac{\mu}{1-\mu}\right) \tag{3.142}$$

Telegram: @uni_k

www.konkur.in

which we can solve for μ to give $\mu = \sigma(\eta)$, where

$$\sigma(\eta) = \frac{1}{1 + \exp(-\eta)} \tag{3.143}$$

is called the *logistic sigmoid* function. Thus, we can write the Bernoulli distribution using the standard representation (3.138) in the form

$$p(x|\eta) = \sigma(-\eta)\exp(\eta x) \tag{3.144}$$

where we have used $1 - \sigma(\eta) = \sigma(-\eta)$, which is easily proved from (3.143). Comparison with (3.138) shows that

$$u(x) = x \tag{3.145}$$
$$h(x) = 1 \tag{3.146}$$
$$g(\eta) = \sigma(-\eta). \tag{3.147}$$

Next consider the multinomial distribution which, for a single observation $\mathbf{x}$, takes the form

$$p(\mathbf{x}|\boldsymbol{\mu}) = \prod_{k=1}^{M} \mu_k^{x_k} = \exp\left\{\sum_{k=1}^{M} x_k \ln \mu_k\right\} \tag{3.148}$$

where $\mathbf{x} = (x_1, \ldots, x_M)^{\mathrm{T}}$. Again, we can write this in the standard representation (3.138) so that

$$p(\mathbf{x}|\boldsymbol{\eta}) = \exp(\boldsymbol{\eta}^{\mathrm{T}}\mathbf{x}) \tag{3.149}$$

where $\eta_k = \ln \mu_k$, and we have defined $\boldsymbol{\eta} = (\eta_1, \ldots, \eta_M)^{\mathrm{T}}$. Again, comparing with (3.138) we have

$$\mathbf{u}(\mathbf{x}) = \mathbf{x} \tag{3.150}$$
$$h(\mathbf{x}) = 1 \tag{3.151}$$
$$g(\boldsymbol{\eta}) = 1. \tag{3.152}$$

Note that the parameters η_k are not independent because the parameters μ_k are subject to the constraint

$$\sum_{k=1}^{M} \mu_k = 1 \tag{3.153}$$

so that, given any $M - 1$ of the parameters μ_k, the value of the remaining parameter is fixed. In some circumstances, it will be convenient to remove this constraint by expressing the distribution in terms of only $M - 1$ parameters. This can be achieved by using the relationship (3.153) to eliminate μ_M by expressing it in terms of the remaining $\{\mu_k\}$ where $k = 1, \ldots, M - 1$, thereby leaving $M - 1$ parameters. Note that these remaining parameters are still subject to the constraints

$$0 \leqslant \mu_k \leqslant 1, \qquad \sum_{k=1}^{M-1} \mu_k \leqslant 1. \tag{3.154}$$

Telegram: @uni_k

www.konkur.in

Making use of the constraint (3.153), the multinomial distribution in this representation then becomes

$$\begin{aligned}
&\exp\left\{\sum_{k=1}^{M} x_k \ln \mu_k\right\} \\
&= \exp\left\{\sum_{k=1}^{M-1} x_k \ln \mu_k + \left(1 - \sum_{k=1}^{M-1} x_k\right) \ln\left(1 - \sum_{k=1}^{M-1} \mu_k\right)\right\} \\
&= \exp\left\{\sum_{k=1}^{M-1} x_k \ln\left(\frac{\mu_k}{1 - \sum_{j=1}^{M-1} \mu_j}\right) + \ln\left(1 - \sum_{k=1}^{M-1} \mu_k\right)\right\}. \qquad (3.155)
\end{aligned}$$

We now identify

$$\ln\left(\frac{\mu_k}{1 - \sum_j \mu_j}\right) = \eta_k, \qquad (3.156)$$

which we can solve for μ_k by first summing both sides over k and then rearranging and back-substituting to give

$$\mu_k = \frac{\exp(\eta_k)}{1 + \sum_j \exp(\eta_j)}. \qquad (3.157)$$

This is called the *softmax* function or the *normalized exponential*. In this representation, the multinomial distribution therefore takes the form

$$p(\mathbf{x}|\boldsymbol{\eta}) = \left(1 + \sum_{k=1}^{M-1} \exp(\eta_k)\right)^{-1} \exp(\boldsymbol{\eta}^{\mathrm{T}}\mathbf{x}). \qquad (3.158)$$

This is the standard form of the exponential family, with parameter vector $\boldsymbol{\eta} = (\eta_1, \ldots, \eta_{M-1})^{\mathrm{T}}$ in which

$$\begin{aligned}
\mathbf{u}(\mathbf{x}) &= \mathbf{x} && (3.159)\\
h(\mathbf{x}) &= 1 && (3.160)\\
g(\boldsymbol{\eta}) &= \left(1 + \sum_{k=1}^{M-1} \exp(\eta_k)\right)^{-1}. && (3.161)
\end{aligned}$$

Finally, let us consider the Gaussian distribution. For the univariate Gaussian, we have

$$\begin{aligned}
p(x|\mu, \sigma^2) &= \frac{1}{(2\pi\sigma^2)^{1/2}} \exp\left\{-\frac{1}{2\sigma^2}(x - \mu)^2\right\} && (3.162)\\
&= \frac{1}{(2\pi\sigma^2)^{1/2}} \exp\left\{-\frac{1}{2\sigma^2}x^2 + \frac{\mu}{\sigma^2}x - \frac{1}{2\sigma^2}\mu^2\right\}, && (3.163)
\end{aligned}$$

which, after some simple rearranging, can be cast in the standard exponential family form (3.138) with

Exercise 3.35

Telegram: @uni_k

www.konkur.in

$$\boldsymbol{\eta} = \begin{pmatrix} \mu/\sigma^2 \\ -1/2\sigma^2 \end{pmatrix} \tag{3.164}$$

$$\mathbf{u}(x) = \begin{pmatrix} x \\ x^2 \end{pmatrix} \tag{3.165}$$

$$h(\mathbf{x}) = (2\pi)^{-1/2} \tag{3.166}$$

$$g(\boldsymbol{\eta}) = (-2\eta_2)^{1/2} \exp\left(\frac{\eta_1^2}{4\eta_2}\right). \tag{3.167}$$

Finally, we shall sometimes make use of a restricted form of (3.138) in which we choose $\mathbf{u}(\mathbf{x}) = \mathbf{x}$. However, this can be somewhat generalized by noting that if $f(\mathbf{x})$ is a normalized density then

$$\frac{1}{s} f\left(\frac{1}{s}\mathbf{x}\right) \tag{3.168}$$

is also a normalized density, where $s > 0$ is a scale parameter. Combining these, we arrive at a restricted set of exponential family class-conditional densities of the form

$$p(\mathbf{x}|\boldsymbol{\lambda}_k, s) = \frac{1}{s} h\left(\frac{1}{s}\mathbf{x}\right) g(\boldsymbol{\lambda}_k) \exp\left\{\frac{1}{s}\boldsymbol{\lambda}_k^{\mathrm{T}}\mathbf{x}\right\}. \tag{3.169}$$

Note that we are allowing each class to have its own parameter vector $\boldsymbol{\lambda}_k$ but we are assuming that the classes share the same scale parameter s.

3.4.1 Sufficient statistics

Let us now consider the problem of estimating the parameter vector $\boldsymbol{\eta}$ in the general exponential family distribution (3.138) using the technique of maximum likelihood. Taking the gradient of both sides of (3.139) with respect to $\boldsymbol{\eta}$, we have

$$\nabla g(\boldsymbol{\eta}) \int h(\mathbf{x}) \exp\left\{\boldsymbol{\eta}^{\mathrm{T}}\mathbf{u}(\mathbf{x})\right\} \mathrm{d}\mathbf{x} + g(\boldsymbol{\eta}) \int h(\mathbf{x}) \exp\left\{\boldsymbol{\eta}^{\mathrm{T}}\mathbf{u}(\mathbf{x})\right\} \mathbf{u}(\mathbf{x}) \mathrm{d}\mathbf{x} = 0. \tag{3.170}$$

Rearranging and making use again of (3.139) then gives

$$-\frac{1}{g(\boldsymbol{\eta})} \nabla g(\boldsymbol{\eta}) = g(\boldsymbol{\eta}) \int h(\mathbf{x}) \exp\left\{\boldsymbol{\eta}^{\mathrm{T}}\mathbf{u}(\mathbf{x})\right\} \mathbf{u}(\mathbf{x}) \mathrm{d}\mathbf{x} = \mathbb{E}[\mathbf{u}(\mathbf{x})]. \tag{3.171}$$

We therefore obtain the result

$$-\nabla \ln g(\boldsymbol{\eta}) = \mathbb{E}[\mathbf{u}(\mathbf{x})]. \tag{3.172}$$

Exercise 3.36

Note that the covariance of $\mathbf{u}(\mathbf{x})$ can be expressed in terms of the second derivatives of $g(\boldsymbol{\eta})$, and similarly for higher-order moments. Thus, provided we can normalize a distribution from the exponential family, we can always find its moments by simple differentiation.

Telegram: @uni_k

www.konkur.in

Now consider a set of independent identically distributed data denoted by $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$, for which the likelihood function is given by

$$p(\mathbf{X}|\boldsymbol{\eta}) = \left(\prod_{n=1}^{N} h(\mathbf{x}_n) \right) g(\boldsymbol{\eta})^N \exp \left\{ \boldsymbol{\eta}^{\mathrm{T}} \sum_{n=1}^{N} \mathbf{u}(\mathbf{x}_n) \right\}. \tag{3.173}$$

Setting the gradient of $\ln p(\mathbf{X}|\boldsymbol{\eta})$ with respect to $\boldsymbol{\eta}$ to zero, we get the following condition to be satisfied by the maximum likelihood estimator $\boldsymbol{\eta}_{\mathrm{ML}}$:

$$-\nabla \ln g(\boldsymbol{\eta}_{\mathrm{ML}}) = \frac{1}{N} \sum_{n=1}^{N} \mathbf{u}(\mathbf{x}_n), \tag{3.174}$$

which can in principle be solved to obtain $\boldsymbol{\eta}_{\mathrm{ML}}$. We see that the solution for the maximum likelihood estimator depends on the data only through $\sum_n \mathbf{u}(\mathbf{x}_n)$, which is therefore called the *sufficient statistic* of the distribution (3.138). We do not need to store the entire data set itself but only the value of the sufficient statistic. For the Bernoulli distribution, for example, the function $\mathbf{u}(x)$ is given just by x and so we need only keep the sum of the data points $\{x_n\}$, whereas for the Gaussian $\mathbf{u}(x) = (x, x^2)^{\mathrm{T}}$, and so we should keep both the sum of $\{x_n\}$ and the sum of $\{x_n^2\}$.

If we consider the limit $N \to \infty$, then the right-hand side of (3.174) becomes $\mathbb{E}[\mathbf{u}(\mathbf{x})]$, and so by comparing with (3.172) we see that in this limit, $\boldsymbol{\eta}_{\mathrm{ML}}$ will equal the true value $\boldsymbol{\eta}$.

3.5. Nonparametric Methods

Throughout this chapter, we have focused on the use of probability distributions having specific functional forms governed by a small number of parameters whose values are to be determined from a data set. This is called the *parametric* approach to density modelling. An important limitation of this approach is that the chosen density might be a poor model of the distribution that generates the data, which can result in poor predictive performance. For instance, if the process that generates the data is multimodal, then this aspect of the distribution can never be captured by a Gaussian, which is necessarily unimodal. In this final section, we consider some *nonparametric* approaches to density estimation that make few assumptions about the form of the distribution.

3.5.1 Histograms

Let us start with a discussion of histogram methods for density estimation, which we have already encountered in the context of marginal and conditional distributions in Figure 2.5 and in the context of the central limit theorem in Figure 3.2. Here we explore the properties of histogram density models in more detail, focusing on cases with a single continuous variable x. Standard histograms simply partition x into distinct bins of width Δ_i and then count the number n_i of observations of x falling

Telegram: @uni_k

www.konkur.in

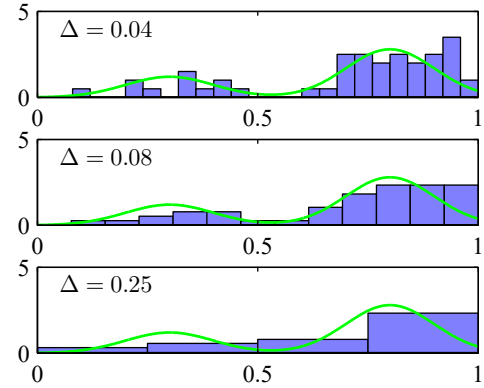


Figure 3.13 An illustration of the histogram approach to density estimation, in which a data set of 50 data points is generated from the distribution shown by the green curve. Histogram density estimates, based on (3.175) with a common bin width Δ, are shown for various values of Δ.

in bin i. To turn this count into a normalized probability density, we simply divide by the total number N of observations and by the width Δ_i of the bins to obtain probability values for each bin:

$$p_i = \frac{n_i}{N\Delta_i} \tag{3.175}$$

for which it is easily seen that $\int p(x)\,\mathrm{d}x = 1$. This gives a model for the density $p(x)$ that is constant over the width of each bin. Often the bins are chosen to have the same width $\Delta_i = \Delta$.

In Figure 3.13, we show an example of histogram density estimation. Here the data is drawn from the distribution corresponding to the green curve, which is formed from a mixture of two Gaussians. Also shown are three examples of histogram density estimates corresponding to three different choices for the bin width Δ. We see that when Δ is very small (top figure), the resulting density model is very spiky, with a lot of structure that is not present in the underlying distribution that generated the data set. Conversely, if Δ is too large (bottom figure) then the result is a model that is too smooth and consequently fails to capture the bimodal property of the green curve. The best results are obtained for some intermediate value of Δ (middle figure). In principle, a histogram density model is also dependent on the choice of edge location for the bins, though this is typically much less significant than the bin width Δ.

Note that the histogram method has the property (unlike the methods to be discussed shortly) that, once the histogram has been computed, the data set itself can be discarded, which can be advantageous if the data set is large. Also, the histogram approach is easily applied if the data points arrive sequentially.

In practice, the histogram technique can be useful for obtaining a quick visualization of data in one or two dimensions but is unsuited to most density estimation applications. One obvious problem is that the estimated density has discontinuities that are due to the bin edges rather than any property of the underlying distribution that generated the data. A major limitation of the histogram approach is its scaling with dimensionality. If we divide each variable in a D-dimensional space into

Telegram: @uni_k

www.konkur.in

M bins, then the total number of bins will be M^D. This exponential scaling with D is an example of the *curse of dimensionality*. In a space of high dimensionality, the quantity of data needed to provide meaningful estimates of the local probability density would be prohibitive.

Section 6.1.1

The histogram approach to density estimation does, however, teach us two important lessons. First, to estimate the probability density at a particular location, we should consider the data points that lie within some local neighbourhood of that point. Note that the concept of locality requires that we assume some form of distance measure, and here we have been assuming Euclidean distance. For histograms, this neighbourhood property was defined by the bins, and there is a natural 'smoothing' parameter describing the spatial extent of the local region, in this case the bin width. Second, to obtain good results, the value of the smoothing parameter should be neither too large nor too small. This is reminiscent of the choice of model complexity in polynomial regression where the degree M of the polynomial, or alternatively the value λ of the regularization parameter, was optimal for some intermediate value, neither too large nor too small. Armed with these insights, we turn now to a discussion of two widely used nonparametric techniques for density estimation, kernel estimators and nearest neighbours, which have better scaling with dimensionality than the simple histogram model.

Chapter 1

3.5.2 Kernel densities

Let us suppose that observations are being drawn from some unknown probability density $p(\mathbf{x})$ in some D-dimensional space, which we will take to be Euclidean, and we wish to estimate the value of $p(\mathbf{x})$. From our earlier discussion of locality, let us consider some small region $\mathcal{R}$ containing $\mathbf{x}$. The probability mass associated with this region is given by

$$P = \int_{\mathcal{R}} p(\mathbf{x})\,\mathrm{d}\mathbf{x}. \tag{3.176}$$

Now suppose that we have collected a data set comprising N observations drawn from $p(\mathbf{x})$. Because each data point has a probability P of falling within $\mathcal{R}$, the total number K of points that lie inside $\mathcal{R}$ will be distributed according to the binomial distribution:

Section 3.1.2

$$\mathrm{Bin}(K|N,P) = \frac{N!}{K!(N-K)!}P^K(1-P)^{N-K}. \tag{3.177}$$

Using (3.11), we see that the mean fraction of points falling inside the region is $\mathbb{E}[K/N] = P$, and similarly using (3.12), we see that the variance around this mean is $\mathrm{var}[K/N] = P(1-P)/N$. For large N, this distribution will be sharply peaked around the mean and so

$$K \simeq NP. \tag{3.178}$$

If, however, we also assume that the region $\mathcal{R}$ is sufficiently small so that the probability density $p(\mathbf{x})$ is roughly constant over the region, then we have

$$P \simeq p(\mathbf{x})V \tag{3.179}$$

Telegram: @uni_k

www.konkur.in

where V is the volume of $\mathcal{R}$. Combining (3.178) and (3.179), we obtain our density estimate in the form

$$p(\mathbf{x}) = \frac{K}{NV}. \tag{3.180}$$

Note that the validity of (3.180) depends on two contradictory assumptions, namely that the region $\mathcal{R}$ is sufficiently small that the density is approximately constant over the region and yet sufficiently large (in relation to the value of that density) that the number K of points falling inside the region is sufficient for the binomial distribution to be sharply peaked.

We can exploit the result (3.180) in two different ways. Either we can fix K and determine the value of V from the data, which gives rise to the K-nearest-neighbour technique discussed shortly, or we can fix V and determine K from the data, giving rise to the kernel approach. It can be shown that both the K-nearest-neighbour density estimator and the kernel density estimator converge to the true probability density in the limit $N \to \infty$ provided that V shrinks with N and that K grows with N, at an appropriate rate (Duda and Hart, 1973).

We begin by discussing the kernel method in detail. To start with we take the region $\mathcal{R}$ to be a small hypercube centred on the point $\mathbf{x}$ at which we wish to determine the probability density. To count the number K of points falling within this region, it is convenient to define the following function:

$$k(\mathbf{u}) = \begin{cases} 1, & |u_i| \leqslant 1/2, \qquad i = 1, \ldots, D, \\ 0, & \text{otherwise}, \end{cases} \tag{3.181}$$

which represents a unit cube centred on the origin. The function $k(\mathbf{u})$ is an example of a *kernel function*, and in this context, it is also called a *Parzen window*. From (3.181), the quantity $k((\mathbf{x} - \mathbf{x}_n)/h)$ will be 1 if the data point $\mathbf{x}_n$ lies inside a cube of side h centred on $\mathbf{x}$, and zero otherwise. The total number of data points lying inside this cube will therefore be

$$K = \sum_{n=1}^{N} k\left(\frac{\mathbf{x} - \mathbf{x}_n}{h}\right). \tag{3.182}$$

Substituting this expression into (3.180) then gives the following result for the estimated density at $\mathbf{x}$:

$$p(\mathbf{x}) = \frac{1}{N} \sum_{n=1}^{N} \frac{1}{h^D} k\left(\frac{\mathbf{x} - \mathbf{x}_n}{h}\right) \tag{3.183}$$

where we have used $V = h^D$ for the volume of a hypercube of side h in D dimensions. Using the symmetry of the function $k(\mathbf{u})$, we can now reinterpret this equation, not as a single cube centred on $\mathbf{x}$ but as the sum over N cubes centred on the N data points $\mathbf{x}_n$.

As it stands, the kernel density estimator (3.183) will suffer from one of the same problems that the histogram method suffered from, namely the presence of artificial discontinuities, in this case at the boundaries of the cubes. We can obtain a smoother

Telegram: @uni_k

www.konkur.in

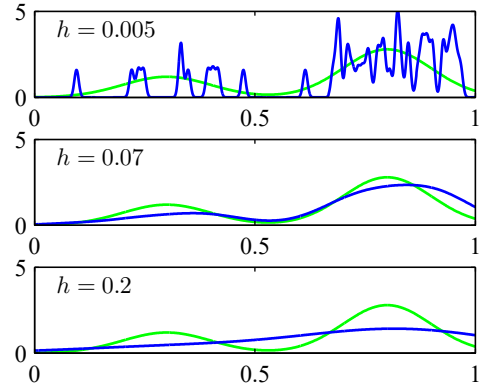


Figure 3.14 Illustration of the kernel density model (3.184) applied to the same data set used to demonstrate the histogram approach in Figure 3.13. We see that h acts as a smoothing parameter and that if it is set too small (top panel), the result is a very noisy density model, whereas if it is set too large (bottom panel), then the bimodal nature of the underlying distribution from which the data is generated (shown by the green curve) is washed out. The best density model is obtained for some intermediate value of h (middle panel).

density model if we choose a smoother kernel function, and a common choice is the Gaussian, which gives rise to the following kernel density model:

$$p(\mathbf{x}) = \frac{1}{N}\sum_{n=1}^{N}\frac{1}{(2\pi h^2)^{D/2}}\exp\left\{-\frac{\|\mathbf{x}-\mathbf{x}_n\|^2}{2h^2}\right\} \tag{3.184}$$

where h represents the standard deviation of the Gaussian components. Thus, our density model is obtained by placing a Gaussian over each data point, adding up the contributions over the whole data set, and then dividing by N so that the density is correctly normalized. In Figure 3.14, we apply the model (3.184) to the data set used earlier to demonstrate the histogram technique. We see that, as expected, the parameter h plays the role of a smoothing parameter, and there is a trade-off between sensitivity to noise at small h and over-smoothing at large h. Again, the optimization of h is a problem in model complexity, analogous to the choice of bin width in histogram density estimation or the degree of the polynomial used in curve fitting.

We can choose any other kernel function $k(\mathbf{u})$ in (3.183) subject to the conditions

$$k(\mathbf{u}) \geqslant 0, \tag{3.185}$$

$$\int k(\mathbf{u})\,\mathrm{d}\mathbf{u} = 1, \tag{3.186}$$

which ensure that the resulting probability distribution is non-negative everywhere and integrates to one. The class of density model given by (3.183) is called a kernel density estimator or *Parzen* estimator. It has a great merit that there is no computation involved in the 'training' phase because this simply requires the training set to be stored. However, this is also one of its great weaknesses because the computational cost of evaluating the density grows linearly with the size of the data set.

Telegram: @uni_k

www.konkur.in

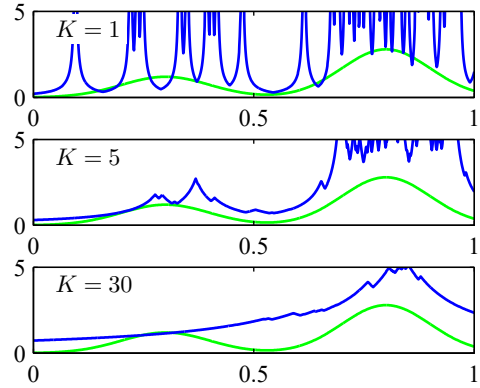


Figure 3.15 Illustration of K-nearest-neighbour density estimation using the same data set as in Figures 3.14 and 3.13. We see that the parameter K governs the degree of smoothing, so that a small value of K leads to a very noisy density model (top panel), whereas a large value (bottom panel) smooths out the bimodal nature of the true distribution (shown by the green curve) from which the data set was generated.

3.5.3 Nearest-neighbours

One of the difficulties with the kernel approach to density estimation is that the parameter h governing the kernel width is fixed for all kernels. In regions of high data density, a large value of h may lead to over-smoothing and a washing out of structure that might otherwise be extracted from the data. However, reducing h may lead to noisy estimates elsewhere in the data space where the density is smaller. Thus, the optimal choice for h may be dependent on the location within the data space. This issue is addressed by nearest-neighbour methods for density estimation.

We therefore return to our general result (3.180) for local density estimation, and instead of fixing V and determining the value of K from the data, we consider a fixed value of K and use the data to find an appropriate value for V. To do this, we consider a small sphere centred on the point $\mathbf{x}$ at which we wish to estimate the density $p(\mathbf{x})$, and we allow the radius of the sphere to grow until it contains precisely K data points. The estimate of the density $p(\mathbf{x})$ is then given by (3.180) with V set to the volume of the resulting sphere. This technique is known as K *nearest neighbours* and is illustrated in Figure 3.15 for various choices of the parameter K using the same data set as used in Figures 3.13 and 3.14. We see that the value of K now governs the degree of smoothing and that again there is an optimum choice for K that is neither too large nor too small. Note that the model produced by K nearest neighbours is not a true density model because the integral over all space diverges.

Exercise 3.38

We close this chapter by showing how the K-nearest-neighbour technique for density estimation can be extended to the problem of classification. To do this, we apply the K-nearest-neighbour density estimation technique to each class separately and then make use of Bayes' theorem. Let us suppose that we have a data set comprising N_k points in class $\mathcal{C}_k$ with N points in total, so that $\sum_k N_k = N$. If we wish to classify a new point $\mathbf{x}$, we draw a sphere centred on $\mathbf{x}$ containing precisely K points irrespective of their class. Suppose this sphere has volume V and contains K_k points from class $\mathcal{C}_k$. Then (3.180) provides an estimate of the density associated

Telegram: @uni_k

www.konkur.in

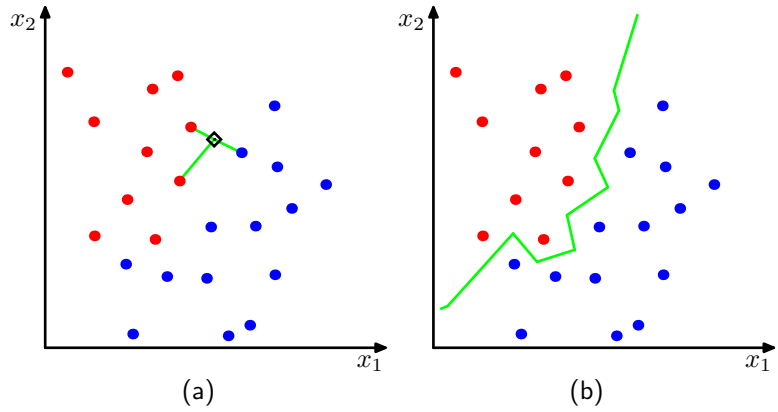


Figure 3.16 (a) In the K-nearest-neighbour classifier, a new point, shown by the black diamond, is classified according to the majority class membership of the K closest training data points, in this case $K = 3$. (b) In the nearest-neighbour ($K = 1$) approach to classification, the resulting decision boundary is composed of hyperplanes that form perpendicular bisectors of pairs of points from different classes.

with each class:

$$p(\mathbf{x}|\mathcal{C}_k) = \frac{K_k}{N_k V}. \tag{3.187}$$

Similarly, the unconditional density is given by

$$p(\mathbf{x}) = \frac{K}{NV} \tag{3.188}$$

and the class priors are given by

$$p(\mathcal{C}_k) = \frac{N_k}{N}. \tag{3.189}$$

We can now combine (3.187), (3.188), and (3.189) using Bayes' theorem to obtain the posterior probability of class membership:

$$p(\mathcal{C}_k|\mathbf{x}) = \frac{p(\mathbf{x}|\mathcal{C}_k)p(\mathcal{C}_k)}{p(\mathbf{x})} = \frac{K_k}{K}. \tag{3.190}$$

We can minimize the probability of misclassification by assigning the test point $\mathbf{x}$ to the class having the largest posterior probability, corresponding to the largest value of K_k/K. Thus, to classify a new point, we identify the K nearest points from the training data set and then assign the new point to the class having the largest number of representatives amongst this set. Ties can be broken at random. The particular case of $K = 1$ is called the *nearest-neighbour* rule, because a test point is simply assigned to the same class as the nearest point from the training set. These concepts are illustrated in Figure 3.16.

An interesting property of the nearest-neighbour ($K = 1$) classifier is that, in the limit $N \to \infty$, the error rate is never more than twice the minimum achievable error rate of an optimal classifier, i.e., one that uses the true class distributions (Cover and Hart, 1967) .

As discussed so far, both the K-nearest-neighbour method and the kernel density estimator require the entire training data set to be stored, leading to expensive

Telegram: @uni_k

www.konkur.in

computation if the data set is large. This effect can be offset, at the expense of some additional one-off computation, by constructing tree-based search structures to allow (approximate) near neighbours to be found efficiently without doing an exhaustive search of the data set. Nevertheless, these nonparametric methods are still severely limited. On the other hand, we have seen that simple parametric models are very restricted in terms of the forms of distribution that they can represent. We therefore need to find density models that are very flexible and yet for which the complexity of the models can be controlled independently of the size of the training set, and this can be achieved using deep neural networks.

Exercises

3.1 ($\star$) Verify that the Bernoulli distribution (3.2) satisfies the following properties:

$$\sum_{x=0}^{1} p(x|\mu) = 1 \tag{3.191}$$

$$\mathbb{E}[x] = \mu \tag{3.192}$$

$$\mathrm{var}[x] = \mu(1-\mu). \tag{3.193}$$

Show that the entropy $\mathrm{H}[x]$ of a Bernoulli-distributed random binary variable x is given by

$$\mathrm{H}[x] = -\mu \ln \mu - (1-\mu)\ln(1-\mu). \tag{3.194}$$

3.2 ($\star\star$) The form of the Bernoulli distribution given by (3.2) is not symmetric between the two values of x. In some situations, it will be more convenient to use an equivalent formulation for which $x \in \{-1, 1\}$, in which case the distribution can be written

$$p(x|\mu) = \left(\frac{1-\mu}{2}\right)^{(1-x)/2} \left(\frac{1+\mu}{2}\right)^{(1+x)/2} \tag{3.195}$$

where $\mu \in [-1, 1]$. Show that the distribution (3.195) is normalized, and evaluate its mean, variance, and entropy.

3.3 ($\star\star$) In this exercise, we prove that the binomial distribution (3.9) is normalized. First, use the definition (3.10) of the number of combinations of m identical objects chosen from a total of N to show that

$$\binom{N}{m} + \binom{N}{m-1} = \binom{N+1}{m}. \tag{3.196}$$

Use this result to prove by induction the following result:

$$(1+x)^N = \sum_{m=0}^{N} \binom{N}{m} x^m, \tag{3.197}$$

Telegram: @uni_k

www.konkur.in

which is known as the *binomial theorem* and which is valid for all real values of x. Finally, show that the binomial distribution is normalized, so that

$$\sum_{m=0}^{N} \binom{N}{m} \mu^m (1-\mu)^{N-m} = 1, \tag{3.198}$$

which can be done by first pulling a factor $(1-\mu)^N$ out of the summation and then making use of the binomial theorem.

3.4 (⋆⋆) Show that the mean of the binomial distribution is given by (3.11). To do this, differentiate both sides of the normalization condition (3.198) with respect to μ and then rearrange to obtain an expression for the mean of n. Similarly, by differentiating (3.198) twice with respect to μ and making use of the result (3.11) for the mean of the binomial distribution, prove the result (3.12) for the variance of the binomial.

3.5 (⋆) Show that the mode of the multivariate Gaussian (3.26) is given by $\boldsymbol{\mu}$.

3.6 (⋆⋆) Suppose that $\mathbf{x}$ has a Gaussian distribution with mean $\boldsymbol{\mu}$ and covariance $\boldsymbol{\Sigma}$. Show that the linearly transformed variable $\mathbf{A}\mathbf{x} + \mathbf{b}$ is also Gaussian, and find its mean and covariance.

3.7 (⋆⋆⋆) Show that the Kullback–Leibler divergence between two Gaussian distributions $q(\mathbf{x}) = \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_q, \boldsymbol{\Sigma}_q)$ and $p(\mathbf{x}) = \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_p, \boldsymbol{\Sigma}_p)$ is given by

$$\begin{aligned} &\mathrm{KL}\left(q(\mathbf{x})\|p(\mathbf{x})\right) \\ &= \frac{1}{2}\left\{\ln\frac{|\boldsymbol{\Sigma}_p|}{|\boldsymbol{\Sigma}_q|} - D + \mathrm{Tr}\left(\boldsymbol{\Sigma}_p^{-1}\boldsymbol{\Sigma}_q\right) + (\boldsymbol{\mu}_p - \boldsymbol{\mu}_q)^{\mathrm{T}}\boldsymbol{\Sigma}_p^{-1}(\boldsymbol{\mu}_p - \boldsymbol{\mu}_q)\right\} \end{aligned} \tag{3.199}$$

where $\mathrm{Tr}(\cdot)$ denotes the trace of a matrix, and D is the dimensionality of $\mathbf{x}$.

3.8 (⋆⋆) This exercise demonstrates that the multivariate distribution with maximum entropy, for a given covariance, is a Gaussian. The entropy of a distribution $p(\mathbf{x})$ is given by

$$\mathrm{H}[\mathbf{x}] = -\int p(\mathbf{x}) \ln p(\mathbf{x})\, \mathrm{d}\mathbf{x}. \tag{3.200}$$

We wish to maximize $\mathrm{H}[\mathbf{x}]$ over all distributions $p(\mathbf{x})$ subject to the constraints that $p(\mathbf{x})$ is normalized and that it has a specific mean and covariance, so that

$$\int p(\mathbf{x})\, \mathrm{d}\mathbf{x} = 1 \tag{3.201}$$

$$\int p(\mathbf{x})\mathbf{x}\, \mathrm{d}\mathbf{x} = \boldsymbol{\mu} \tag{3.202}$$

$$\int p(\mathbf{x})(\mathbf{x}-\boldsymbol{\mu})(\mathbf{x}-\boldsymbol{\mu})^{\mathrm{T}}\, \mathrm{d}\mathbf{x} = \boldsymbol{\Sigma}. \tag{3.203}$$

By performing a variational maximization of (3.200) and using Lagrange multipliers to enforce the constraints (3.201), (3.202), and (3.203), show that the maximum likelihood distribution is given by the Gaussian (3.26).

Telegram: @uni_k

www.konkur.in

3.9 (⋆⋆⋆) Show that the entropy of the multivariate Gaussian $\mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma})$ is given by

$$\mathrm{H}[\mathbf{x}] = \frac{1}{2}\ln|\boldsymbol{\Sigma}| + \frac{D}{2}(1 + \ln(2\pi)) \tag{3.204}$$

where D is the dimensionality of $\mathbf{x}$.

3.10 (⋆⋆⋆) Consider two random variables x_1 and x_2 having Gaussian distributions with means μ_1 and μ_2 and precisions τ_1 and τ_2, respectively. Derive an expression for the differential entropy of the variable $x = x_1 + x_2$. To do this, first find the distribution of x by using the relation

$$p(x) = \int_{-\infty}^{\infty} p(x|x_2)p(x_2)\,\mathrm{d}x_2 \tag{3.205}$$

and completing the square in the exponent. Then observe that this represents the convolution of two Gaussian distributions, which itself will be Gaussian, and finally make use of the result (2.99) for the entropy of the univariate Gaussian.

3.11 (⋆) Consider the multivariate Gaussian distribution given by (3.26). By writing the precision matrix (inverse covariance matrix) as the sum of a symmetric and an antisymmetric matrix, show that the antisymmetric term does not appear in the exponent of the Gaussian, and hence, that the precision matrix may be taken to be symmetric without loss of generality. Because the inverse of a symmetric matrix is also symmetric (see Exercise 3.16), it follows that the covariance matrix may also be chosen to be symmetric without loss of generality.

3.12 (⋆⋆⋆) Consider a real, symmetric matrix $\boldsymbol{\Sigma}$ whose eigenvalue equation is given by (3.28). By taking the complex conjugate of this equation, subtracting the original equation, and then forming the inner product with eigenvector $\mathbf{u}_i$, show that the eigenvalues λ_i are real. Similarly, use the symmetry property of $\boldsymbol{\Sigma}$ to show that two eigenvectors $\mathbf{u}_i$ and $\mathbf{u}_j$ will be orthogonal provided $\lambda_j \neq \lambda_i$. Finally, show that, without loss of generality, the set of eigenvectors can be chosen to be orthonormal, so that they satisfy (3.29), even if some of the eigenvalues are zero.

3.13 (⋆⋆) Show that a real, symmetric matrix $\boldsymbol{\Sigma}$ having the eigenvector equation (3.28) can be expressed as an expansion in the eigenvectors, with coefficients given by the eigenvalues, of the form (3.31). Similarly, show that the inverse matrix $\boldsymbol{\Sigma}^{-1}$ has a representation of the form (3.32).

3.14 (⋆⋆) A positive definite matrix $\boldsymbol{\Sigma}$ can be defined as one for which the quadratic form

$$\mathbf{a}^{\mathrm{T}}\boldsymbol{\Sigma}\mathbf{a} \tag{3.206}$$

is positive for any real value of the vector $\mathbf{a}$. Show that a necessary and sufficient condition for $\boldsymbol{\Sigma}$ to be positive definite is that all the eigenvalues λ_i of $\boldsymbol{\Sigma}$, defined by (3.28), are positive.

3.15 (⋆) Show that a real, symmetric matrix of size $D \times D$ has $D(D+1)/2$ independent parameters.

Telegram: @uni_k

www.konkur.in

3.16 ($\star$) Show that the inverse of a symmetric matrix is itself symmetric.

3.17 ($\star\star$) By diagonalizing the coordinate system using the eigenvector expansion (3.31), show that the volume contained within the hyperellipsoid corresponding to a constant Mahalanobis distance Δ is given by

$$V_D|\boldsymbol{\Sigma}|^{1/2}\Delta^D \tag{3.207}$$

where V_D is the volume of the unit sphere in D dimensions, and the Mahalanobis distance is defined by (3.27).

3.18 ($\star\star$) Prove the identity (3.60) by multiplying both sides by the matrix

$$\begin{pmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{pmatrix} \tag{3.208}$$

and making use of the definition (3.61).

3.19 ($\star\star\star$) In Sections 3.2.4 and 3.2.5, we considered the conditional and marginal distributions for a multivariate Gaussian. More generally, we can consider a partitioning of the components of $\mathbf{x}$ into three groups $\mathbf{x}_a$, $\mathbf{x}_b$, and $\mathbf{x}_c$, with a corresponding partitioning of the mean vector $\boldsymbol{\mu}$ and of the covariance matrix $\boldsymbol{\Sigma}$ in the form

$$\boldsymbol{\mu} = \begin{pmatrix} \boldsymbol{\mu}_a \\ \boldsymbol{\mu}_b \\ \boldsymbol{\mu}_c \end{pmatrix}, \qquad \boldsymbol{\Sigma} = \begin{pmatrix} \boldsymbol{\Sigma}_{aa} & \boldsymbol{\Sigma}_{ab} & \boldsymbol{\Sigma}_{ac} \\ \boldsymbol{\Sigma}_{ba} & \boldsymbol{\Sigma}_{bb} & \boldsymbol{\Sigma}_{bc} \\ \boldsymbol{\Sigma}_{ca} & \boldsymbol{\Sigma}_{cb} & \boldsymbol{\Sigma}_{cc} \end{pmatrix}. \tag{3.209}$$

By making use of the results of Section 3.2, find an expression for the conditional distribution $p(\mathbf{x}_a|\mathbf{x}_b)$ in which $\mathbf{x}_c$ has been marginalized out.

3.20 ($\star\star$) A very useful result from linear algebra is the *Woodbury* matrix inversion formula given by

$$(\mathbf{A}+\mathbf{BCD})^{-1} = \mathbf{A}^{-1} - \mathbf{A}^{-1}\mathbf{B}(\mathbf{C}^{-1}+\mathbf{DA}^{-1}\mathbf{B})^{-1}\mathbf{DA}^{-1}. \tag{3.210}$$

By multiplying both sides by $(\mathbf{A}+\mathbf{BCD})$, prove the correctness of this result.

3.21 ($\star$) Let $\mathbf{x}$ and $\mathbf{z}$ be two independent random vectors, so that $p(\mathbf{x},\mathbf{z}) = p(\mathbf{x})p(\mathbf{z})$. Show that the mean of their sum $\mathbf{y} = \mathbf{x}+\mathbf{z}$ is given by the sum of the means of each of the variables separately. Similarly, show that the covariance matrix of $\mathbf{y}$ is given by the sum of the covariance matrices of $\mathbf{x}$ and $\mathbf{z}$.

3.22 ($\star\star\star$) Consider a joint distribution over the variable

$$\mathbf{z} = \begin{pmatrix} \mathbf{x} \\ \mathbf{y} \end{pmatrix} \tag{3.211}$$

whose mean and covariance are given by (3.92) and (3.89), respectively. By making use of the results (3.76) and (3.77), show that the marginal distribution $p(\mathbf{x})$ is given by (3.83). Similarly, by making use of the results (3.65) and (3.66), show that the conditional distribution $p(\mathbf{y}|\mathbf{x})$ is given by (3.84).

Telegram: @uni_k

www.konkur.in

3.23 (⋆⋆) Using the partitioned matrix inversion formula (3.60), show that the inverse of the precision matrix (3.88) is given by the covariance matrix (3.89).

3.24 (⋆⋆) By starting from (3.91) and making use of the result (3.89), verify the result (3.92).

3.25 (⋆⋆) Consider two multi-dimensional random vectors $\mathbf{x}$ and $\mathbf{z}$ having Gaussian distributions $p(\mathbf{x}) = \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_{\mathbf{x}}, \boldsymbol{\Sigma}_{\mathbf{x}})$ and $p(\mathbf{z}) = \mathcal{N}(\mathbf{z}|\boldsymbol{\mu}_{\mathbf{z}}, \boldsymbol{\Sigma}_{\mathbf{z}})$, respectively, together with their sum $\mathbf{y} = \mathbf{x} + \mathbf{z}$. By considering the linear-Gaussian model comprising the product of the marginal distribution $p(\mathbf{x})$ and the conditional distribution $p(\mathbf{y}|\mathbf{x})$ and making use of the results (3.93) and (3.94), show that the marginal distribution of $p(\mathbf{y})$ is given by

$$p(\mathbf{y}) = \mathcal{N}(\mathbf{y}|\boldsymbol{\mu}_{\mathbf{x}} + \boldsymbol{\mu}_{\mathbf{z}}, \boldsymbol{\Sigma}_{\mathbf{x}} + \boldsymbol{\Sigma}_{\mathbf{z}}). \tag{3.212}$$

3.26 (⋆⋆⋆) This exercise and the next provide practice at manipulating the quadratic forms that arise in linear-Gaussian models, and they also serve as an independent check of results derived in the main text. Consider a joint distribution $p(\mathbf{x}, \mathbf{y})$ defined by the marginal and conditional distributions given by (3.83) and (3.84). By examining the quadratic form in the exponent of the joint distribution and using the technique of 'completing the square' discussed in Section 3.2, find expressions for the mean and covariance of the marginal distribution $p(\mathbf{y})$ in which the variable $\mathbf{x}$ has been integrated out. To do this, make use of the Woodbury matrix inversion formula (3.210). Verify that these results agree with (3.93) and (3.94).

3.27 (⋆⋆⋆) Consider the same joint distribution as in Exercise 3.26, but now use the technique of completing the square to find expressions for the mean and covariance of the conditional distribution $p(\mathbf{x}|\mathbf{y})$. Again, verify that these agree with the corresponding expressions (3.95) and (3.96).

3.28 (⋆⋆) To find the maximum likelihood solution for the covariance matrix of a multivariate Gaussian, we need to maximize the log likelihood function (3.102) with respect to $\boldsymbol{\Sigma}$, noting that the covariance matrix must be symmetric and positive definite. Here we proceed by ignoring these constraints and doing a straightforward maximization. Using the results (A.21), (A.26), and (A.28) from Appendix A, show that the covariance matrix $\boldsymbol{\Sigma}$ that maximizes the log likelihood function (3.102) is given by the sample covariance (3.106). We note that the final result is necessarily symmetric and positive definite (provided the sample covariance is non-singular).

3.29 (⋆⋆) Use the result (3.42) to prove (3.46). Now, using the results (3.42) and (3.46), show that

$$\mathbb{E}[\mathbf{x}_n \mathbf{x}_m^{\mathrm{T}}] = \boldsymbol{\mu}\boldsymbol{\mu}^{\mathrm{T}} + I_{nm}\boldsymbol{\Sigma} \tag{3.213}$$

where $\mathbf{x}_n$ denotes a data point sampled from a Gaussian distribution with mean $\boldsymbol{\mu}$ and covariance $\boldsymbol{\Sigma}$, and I_{nm} denotes the (n, m) element of the identity matrix. Hence, prove the result (3.108).

3.30 (⋆) The various trigonometric identities used in the discussion of periodic variables in this chapter can be proven easily from the relation

$$\exp(iA) = \cos A + i \sin A \tag{3.214}$$

Telegram: @uni_k

www.konkur.in

in which i is the square root of minus one. By considering the identity

$$\exp(iA)\exp(-iA) = 1 \tag{3.215}$$

prove the result (3.127). Similarly, using the identity

$$\cos(A-B) = \Re\exp\{i(A-B)\} \tag{3.216}$$

where $\Re$ denotes the real part, prove (3.128). Finally, by using $\sin(A-B) = \Im\exp\{i(A-B)\}$, where $\Im$ denotes the imaginary part, prove the result (3.133).

3.31 ($\star\star$) For large m, the von Mises distribution (3.129) becomes sharply peaked around the mode θ_0. By defining $\xi = m^{1/2}(\theta - \theta_0)$ and taking the Taylor expansion of the cosine function given by

$$\cos\alpha = 1 - \frac{\alpha^2}{2} + O(\alpha^4) \tag{3.217}$$

show that as $m \to \infty$, the von Mises distribution tends to a Gaussian.

3.32 ($\star$) Using the trigonometric identity (3.133), show that solution of (3.132) for θ_0 is given by (3.134).

3.33 ($\star$) By computing the first and second derivatives of the von Mises distribution (3.129), and using $I_0(m) > 0$ for $m > 0$, show that the maximum of the distribution occurs when $\theta = \theta_0$ and that the minimum occurs when $\theta = \theta_0 + \pi \pmod{2\pi}$.

3.34 ($\star$) By making use of the result (3.118) together with (3.134) and the trigonometric identity (3.128), show that the maximum likelihood solution m_{ML} for the concentration of the von Mises distribution satisfies $A(m_{\mathrm{ML}}) = \overline{r}$ where $\overline{r}$ is the radius of the mean of the observations viewed as unit vectors in the two-dimensional Euclidean plane, as illustrated in Figure 3.9.

3.35 ($\star$) Verify that the multivariate Gaussian distribution can be cast in exponential family form (3.138), and derive expressions for $\boldsymbol{\eta}$, $\mathbf{u}(\mathbf{x})$, $h(\mathbf{x})$, and $g(\boldsymbol{\eta})$ analogous to (3.164) to (3.167).

3.36 ($\star$) The result (3.172) showed that the negative gradient of $\ln g(\boldsymbol{\eta})$ for the exponential family is given by the expectation of $\mathbf{u}(\mathbf{x})$. By taking the second derivatives of (3.139), show that

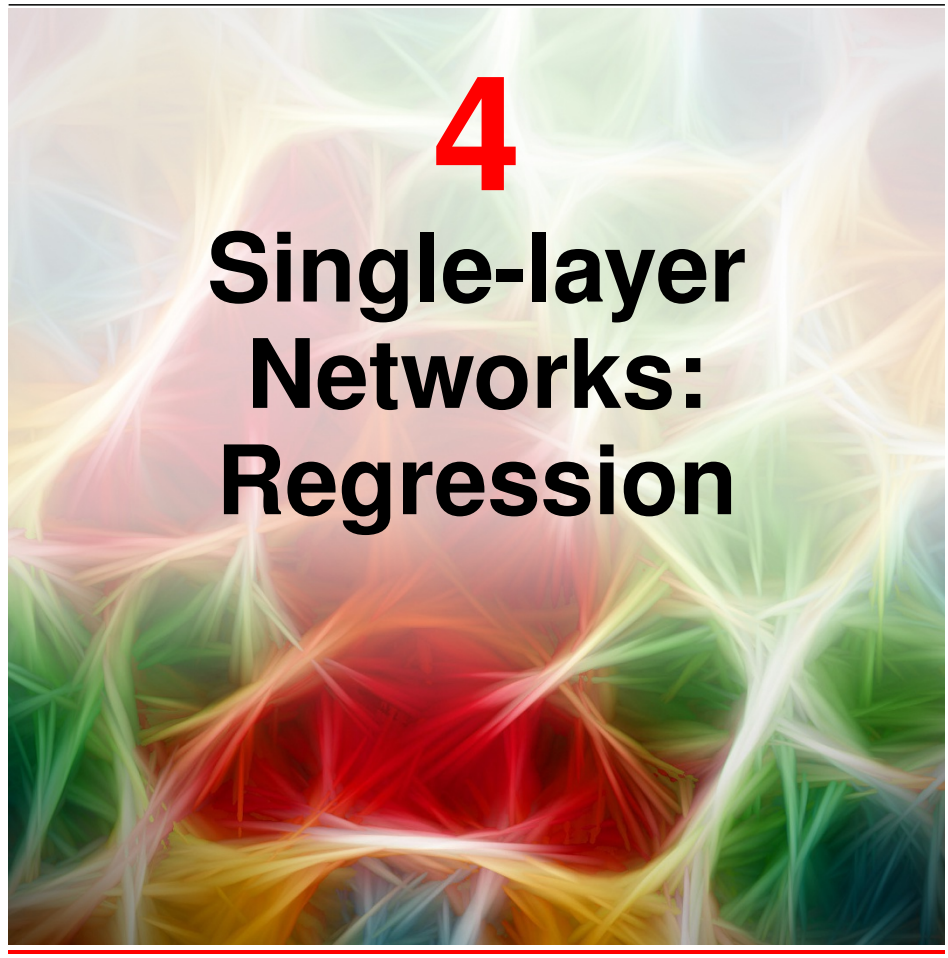
$$-\nabla\nabla\ln g(\boldsymbol{\eta}) = \mathbb{E}[\mathbf{u}(\mathbf{x})\mathbf{u}(\mathbf{x})^{\mathrm{T}}] - \mathbb{E}[\mathbf{u}(\mathbf{x})]\mathbb{E}[\mathbf{u}(\mathbf{x})^{\mathrm{T}}] = \mathrm{cov}[\mathbf{u}(\mathbf{x})]. \tag{3.218}$$

3.37 ($\star\star$) Consider a histogram-like density model in which the space $\mathbf{x}$ is divided into fixed regions for which the density $p(\mathbf{x})$ takes the constant value h_i over the ith region. The volume of region i is denoted Δ_i. Suppose we have a set of N observations of $\mathbf{x}$ such that n_i of these observations fall in region i. Using a Lagrange multiplier to enforce the normalization constraint on the density, derive an expression for the maximum likelihood estimator for the $\{h_i\}$.

3.38 ($\star$) Show that the K-nearest-neighbour density model defines an improper distribution whose integral over all space is divergent.

Telegram: @uni_k

www.konkur.in



4 Single-layer Networks: Regression

Section 1.2

In this chapter we discuss some of the basic ideas behind neural networks using the framework of linear regression, which we encountered briefly in the context of polynomial curve fitting. We will see that a linear regression model corresponds to a simple form of neural network having a single layer of learnable parameters. Although single-layer networks have very limited practical applicability, they have simple analytical properties and provide an excellent framework for introducing many of the core concepts that will lay a foundation for our discussion of deep neural networks in later chapters.

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024
C. M. Bishop, H. Bishop, *Deep Learning*, https://doi.org/10.1007/978-3-031-45468-4_4

Telegram: @uni_k

www.konkur.in

4.1. Linear Regression

The goal of regression is to predict the value of one or more continuous *target* variables t given the value of a D-dimensional vector $\mathbf{x}$ of *input* variables. Typically we are given a training data set comprising N observations $\{\mathbf{x}_n\}$, where $n = 1, \ldots, N$, together with corresponding target values $\{t_n\}$, and the goal is to predict the value of t for a new value of $\mathbf{x}$. To do this, we formulate a function $y(\mathbf{x}, \mathbf{w})$ whose values for new inputs $\mathbf{x}$ constitute the predictions for the corresponding values of t, and where $\mathbf{w}$ represents a vector of parameters that can be learned from the training data.

The simplest model for regression is one that involves a linear combination of the input variables:

$$y(\mathbf{x}, \mathbf{w}) = w_0 + w_1 x_1 + \ldots + w_D x_D \tag{4.1}$$

where $\mathbf{x} = (x_1, \ldots, x_D)^{\mathrm{T}}$. The term *linear regression* sometimes refers specifically to this form of model. The key property of this model is that it is a linear function of the parameters $w_0, \ldots, w_D$. It is also, however, a linear function of the input variables x_i, and this imposes significant limitations on the model.

4.1.1 Basis functions

We can extend the class of models defined by (4.1) by considering linear combinations of fixed nonlinear functions of the input variables, of the form

$$y(\mathbf{x}, \mathbf{w}) = w_0 + \sum_{j=1}^{M-1} w_j \phi_j(\mathbf{x}) \tag{4.2}$$

where $\phi_j(\mathbf{x})$ are known as *basis functions*. By denoting the maximum value of the index j by $M - 1$, the total number of parameters in this model will be M.

The parameter w_0 allows for any fixed offset in the data and is sometimes called a *bias* parameter (not to be confused with bias in a statistical sense). It is often convenient to define an additional dummy basis function $\phi_0(\mathbf{x})$ whose value is fixed at $\phi_0(\mathbf{x}) = 1$ so that (4.2) becomes

Section 4.3

$$y(\mathbf{x}, \mathbf{w}) = \sum_{j=0}^{M-1} w_j \phi_j(\mathbf{x}) = \mathbf{w}^{\mathrm{T}} \boldsymbol{\phi}(\mathbf{x}) \tag{4.3}$$

where $\mathbf{w} = (w_0, \ldots, w_{M-1})^{\mathrm{T}}$ and $\boldsymbol{\phi} = (\phi_0, \ldots, \phi_{M-1})^{\mathrm{T}}$. We can represent the model (4.3) using a neural network diagram, as shown in Figure 4.1.

By using nonlinear basis functions, we allow the function $y(\mathbf{x}, \mathbf{w})$ to be a nonlinear function of the input vector $\mathbf{x}$. Functions of the form (4.2) are called linear models, however, because they are linear in $\mathbf{w}$. It is this linearity in the parameters that will greatly simplify the analysis of this class of models. However, it also leads to some significant limitations.

Section 6.1

Telegram: @uni_k

www.konkur.in

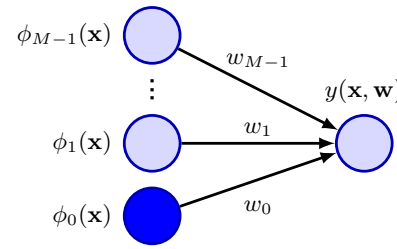


Figure 4.1 The linear regression model (4.3) can be expressed as a simple neural network diagram involving a single layer of parameters. Here each basis function $\phi_j(\mathbf{x})$ is represented by an input node, with the solid node representing the 'bias' basis function ϕ_0, and the function $y(\mathbf{x}, \mathbf{w})$ is represented by an output node. Each of the parameters w_j is shown by a line connecting the corresponding basis function to the output.

Before the advent of deep learning it was common practice in machine learning to use some form of fixed pre-processing of the input variables $\mathbf{x}$, also known as *feature extraction*, expressed in terms of a set of basis functions $\{\phi_j(\mathbf{x})\}$. The goal was to choose a sufficiently powerful set of basis functions that the resulting learning task could be solved using a simple network model. Unfortunately, it is very difficult to hand-craft suitable basis functions for anything but the simplest applications. Deep learning avoids this problem by learning the required nonlinear transformations of the data from the data set itself.

Chapter 1

We have already encountered an example of a regression problem when we discussed curve fitting using polynomials. The polynomial function (1.1) can be expressed in the form (4.3) if we consider a single input variable x and if we choose basis functions defined by $\phi_j(x) = x^j$. There are many other possible choices for the basis functions, for example

$$\phi_j(x) = \exp\left\{-\frac{(x-\mu_j)^2}{2s^2}\right\} \tag{4.4}$$

where the μ_j govern the locations of the basis functions in input space, and the parameter s governs their spatial scale. These are usually referred to as 'Gaussian' basis functions, although it should be noted that they are not required to have a probabilistic interpretation. In particular the normalization coefficient is unimportant because these basis functions will be multiplied by learnable parameters w_j.

Another possibility is the sigmoidal basis function of the form

$$\phi_j(x) = \sigma\left(\frac{x-\mu_j}{s}\right) \tag{4.5}$$

where $\sigma(a)$ is the logistic sigmoid function defined by

$$\sigma(a) = \frac{1}{1+\exp(-a)}. \tag{4.6}$$

Exercise 4.3

Equivalently, we can use the $\tanh$ function because this is related to the logistic sigmoid by $\tanh(a) = 2\sigma(2a) - 1$, and so a general linear combination of logistic sigmoid functions is equivalent to a general linear combination of $\tanh$ functions in the sense that they can represent the same class of input–output functions. These various choices of basis function are illustrated in Figure 4.2.

Telegram: @uni_k

www.konkur.in

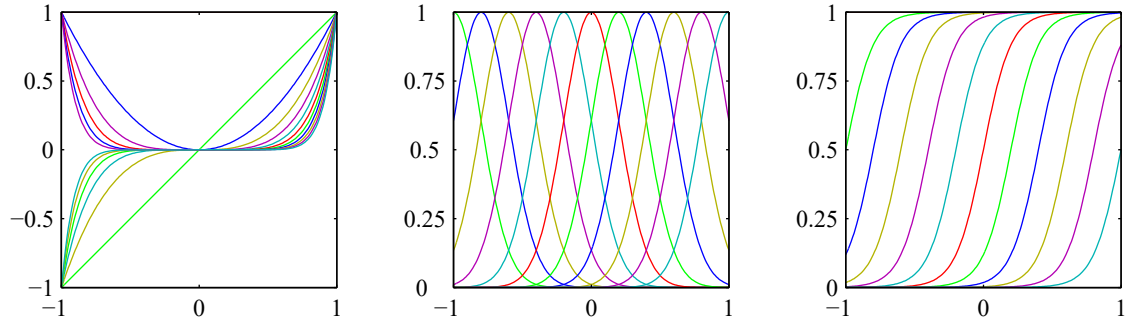


Figure 4.2 Examples of basis functions, showing polynomials on the left, Gaussians of the form (4.4) in the centre, and sigmoidal basis functions of the form (4.5) on the right.

Yet another possible choice of basis function is the Fourier basis, which leads to an expansion in sinusoidal functions. Each basis function represents a specific frequency and has infinite spatial extent. By contrast, basis functions that are localized to finite regions of input space necessarily comprise a spectrum of different spatial frequencies. In signal processing applications, it is often of interest to consider basis functions that are localized in both space and frequency, leading to a class of functions known as *wavelets* (Ogden, 1997; Mallat, 1999; Vidakovic, 1999). These are also defined to be mutually orthogonal, to simplify their application. Wavelets are most applicable when the input values live on a regular lattice, such as the successive time points in a temporal sequence or the pixels in an image.

Most of the discussion in this chapter, however, is independent of the choice of basis function set, and so we will not specify the particular form of the basis functions, except for numerical illustration. Furthermore, to keep the notation simple, we will focus on the case of a single target variable t, although we will briefly outline the modifications needed to deal with multiple target variables.

Section 4.1.7

4.1.2 Likelihood function

We solved the problem of fitting a polynomial function to data by minimizing a sum-of-squares error function, and we also showed that this error function could be motivated as the maximum likelihood solution under an assumed Gaussian noise model. We now return to this discussion and consider the least-squares approach, and its relation to maximum likelihood, in more detail.

Section 1.2

As before, we assume that the target variable t is given by a deterministic function $y(\mathbf{x}, \mathbf{w})$ with additive Gaussian noise so that

$$t = y(\mathbf{x}, \mathbf{w}) + \epsilon \tag{4.7}$$

where ϵ is a zero-mean Gaussian random variable with variance σ^2. Thus, we can write

$$p(t|\mathbf{x}, \mathbf{w}, \sigma^2) = \mathcal{N}(t|y(\mathbf{x}, \mathbf{w}), \sigma^2). \tag{4.8}$$

Telegram: @uni_k

www.konkur.in

Now consider a data set of inputs $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$ with corresponding target values $t_1, \ldots, t_N$. We group the target variables $\{t_n\}$ into a column vector that we denote by $\mathbf{t}$ where the typeface is chosen to distinguish it from a single observation of a multivariate target, which would be denoted $\mathbf{t}$. Making the assumption that these data points are drawn independently from the distribution (4.8), we obtain an expression for the likelihood function, which is a function of the adjustable parameters $\mathbf{w}$ and σ^2:

$$p(\mathbf{t}|\mathbf{X}, \mathbf{w}, \sigma^2) = \prod_{n=1}^{N} \mathcal{N}(t_n|\mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}_n), \sigma^2) \tag{4.9}$$

where we have used (4.3). Taking the logarithm of the likelihood function and making use of the standard form (2.49) for the univariate Gaussian, we have

$$\begin{aligned} \ln p(\mathbf{t}|\mathbf{X}, \mathbf{w}, \sigma^2) &= \sum_{n=1}^{N} \ln \mathcal{N}(t_n|\mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}_n), \sigma^2) \\ &= -\frac{N}{2}\ln\sigma^2 - \frac{N}{2}\ln(2\pi) - \frac{1}{\sigma^2}E_D(\mathbf{w}) \end{aligned} \tag{4.10}$$

where the sum-of-squares error function is defined by

$$E_D(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N}\{t_n - \mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}_n)\}^2. \tag{4.11}$$

Section 2.3.4

The first two terms in (4.10) can be treated as constants when determining $\mathbf{w}$ because they are independent of $\mathbf{w}$. Therefore, as we saw previously, maximizing the likelihood function under a Gaussian noise distribution is equivalent to minimizing the sum-of-squares error function (4.11).

4.1.3 Maximum likelihood

Having written down the likelihood function, we can use maximum likelihood to determine $\mathbf{w}$ and σ^2. Consider first the maximization with respect to $\mathbf{w}$. The gradient of the log likelihood function (4.10) with respect to $\mathbf{w}$ takes the form

$$\nabla_{\mathbf{w}} \ln p(\mathbf{t}|\mathbf{X}, \mathbf{w}, \sigma^2) = \frac{1}{\sigma^2}\sum_{n=1}^{N}\left\{t_n - \mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}_n)\right\}\boldsymbol{\phi}(\mathbf{x}_n)^{\mathrm{T}}. \tag{4.12}$$

Setting this gradient to zero gives

$$0 = \sum_{n=1}^{N} t_n\boldsymbol{\phi}(\mathbf{x}_n)^{\mathrm{T}} - \mathbf{w}^{\mathrm{T}}\left(\sum_{n=1}^{N}\boldsymbol{\phi}(\mathbf{x}_n)\boldsymbol{\phi}(\mathbf{x}_n)^{\mathrm{T}}\right). \tag{4.13}$$

Solving for $\mathbf{w}$ we obtain

$$\mathbf{w}_{\mathrm{ML}} = \left(\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\Phi}\right)^{-1}\boldsymbol{\Phi}^{\mathrm{T}}\mathbf{t}, \tag{4.14}$$

Telegram: @uni_k

www.konkur.in

which are known as the *normal equations* for the least-squares problem. Here $\mathbf{\Phi}$ is an $N \times M$ matrix, called the *design matrix*, whose elements are given by $\Phi_{nj} = \phi_j(\mathbf{x}_n)$, so that

$$\mathbf{\Phi} = \begin{pmatrix} \phi_0(\mathbf{x}_1) & \phi_1(\mathbf{x}_1) & \cdots & \phi_{M-1}(\mathbf{x}_1) \\ \phi_0(\mathbf{x}_2) & \phi_1(\mathbf{x}_2) & \cdots & \phi_{M-1}(\mathbf{x}_2) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_0(\mathbf{x}_N) & \phi_1(\mathbf{x}_N) & \cdots & \phi_{M-1}(\mathbf{x}_N) \end{pmatrix}. \tag{4.15}$$

The quantity

$$\mathbf{\Phi}^\dagger \equiv \left(\mathbf{\Phi}^{\mathrm{T}}\mathbf{\Phi}\right)^{-1}\mathbf{\Phi}^{\mathrm{T}} \tag{4.16}$$

is known as the *Moore–Penrose pseudo-inverse* of the matrix $\mathbf{\Phi}$ (Rao and Mitra, 1971; Golub and Van Loan, 1996). It can be regarded as a generalization of the notion of a matrix inverse to non-square matrices. Indeed, if $\mathbf{\Phi}$ is square and invertible, then using the property $(\mathbf{AB})^{-1} = \mathbf{B}^{-1}\mathbf{A}^{-1}$ we see that $\mathbf{\Phi}^\dagger \equiv \mathbf{\Phi}^{-1}$.

At this point, we can gain some insight into the role of the bias parameter w_0. If we make the bias parameter explicit, then the error function (4.11) becomes

$$E_D(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N}\{t_n - w_0 - \sum_{j=1}^{M-1} w_j\phi_j(\mathbf{x}_n)\}^2. \tag{4.17}$$

Setting the derivative with respect to w_0 equal to zero and solving for w_0, we obtain

$$w_0 = \bar{t} - \sum_{j=1}^{M-1} w_j\overline{\phi_j} \tag{4.18}$$

where we have defined

$$\bar{t} = \frac{1}{N}\sum_{n=1}^{N} t_n, \qquad \overline{\phi_j} = \frac{1}{N}\sum_{n=1}^{N}\phi_j(\mathbf{x}_n). \tag{4.19}$$

Thus, the bias w_0 compensates for the difference between the averages (over the training set) of the target values and the weighted sum of the averages of the basis function values.

We can also maximize the log likelihood function (4.10) with respect to the variance σ^2, giving

$$\sigma^2_{\mathrm{ML}} = \frac{1}{N}\sum_{n=1}^{N}\{t_n - \mathbf{w}_{\mathrm{ML}}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}_n)\}^2, \tag{4.20}$$

and so we see that the maximum likelihood value of the variance parameter is given by the residual variance of the target values around the regression function.

Telegram: @uni_k

www.konkur.in

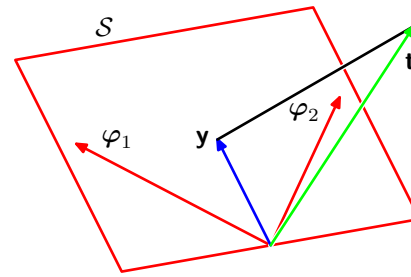


Figure 4.3 Geometrical interpretation of the least-squares solution in an N-dimensional space whose axes are the values of $t_1, \ldots, t_N$. The least-squares regression function is obtained by finding the orthogonal projection of the data vector $\mathbf{t}$ onto the subspace spanned by the basis functions $\phi_j(\mathbf{x})$ in which each basis function is viewed as a vector $\boldsymbol{\varphi}_j$ of length N with elements $\phi_j(\mathbf{x}_n)$.

4.1.4 Geometry of least squares

At this point, it is instructive to consider the geometrical interpretation of the least-squares solution. To do this, we consider an N-dimensional space whose axes are given by the t_n, so that $\mathbf{t} = (t_1, \ldots, t_N)^{\mathrm{T}}$ is a vector in this space. Each basis function $\phi_j(\mathbf{x}_n)$, evaluated at the N data points, can also be represented as a vector in the same space, denoted by $\boldsymbol{\varphi}_j$, as illustrated in Figure 4.3. Note that $\boldsymbol{\varphi}_j$ corresponds to the jth column of $\boldsymbol{\Phi}$, whereas $\boldsymbol{\phi}(\mathbf{x}_n)$ corresponds to the transpose of the nth row of $\boldsymbol{\Phi}$. If the number M of basis functions is smaller than the number N of data points, then the M vectors $\phi_j(\mathbf{x}_n)$ will span a linear subspace $\mathcal{S}$ of dimensionality M. We define $\mathbf{y}$ to be an N-dimensional vector whose nth element is given by $y(\mathbf{x}_n, \mathbf{w})$, where $n = 1, \ldots, N$. Because $\mathbf{y}$ is an arbitrary linear combination of the vectors $\boldsymbol{\varphi}_j$, it can live anywhere in the M-dimensional subspace. The sum-of-squares error (4.11) is then equal (up to a factor of $1/2$) to the squared Euclidean distance between $\mathbf{y}$ and $\mathbf{t}$. Thus, the least-squares solution for $\mathbf{w}$ corresponds to that choice of $\mathbf{y}$ that lies in subspace $\mathcal{S}$ and is closest to $\mathbf{t}$. Intuitively, from Figure 4.3, we anticipate that this solution corresponds to the orthogonal projection of $\mathbf{t}$ onto the subspace $\mathcal{S}$. This is indeed the case, as can easily be verified by noting that the solution for $\mathbf{y}$ is given by $\boldsymbol{\Phi}\mathbf{w}_{\mathrm{ML}}$ and then confirming that this takes the form of an orthogonal projection.

Exercise 4.4

In practice, a direct solution of the normal equations can lead to numerical difficulties when $\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\Phi}$ is close to singular. In particular, when two or more of the basis vectors $\boldsymbol{\varphi}_j$ are co-linear, or nearly so, the resulting parameter values can have large magnitudes. Such near degeneracies will not be uncommon when dealing with real data sets. The resulting numerical difficulties can be addressed using the technique of *singular value decomposition*, or *SVD* (Deisenroth, Faisal, and Ong, 2020). Note that the addition of a regularization term ensures that the matrix is non-singular, even in the presence of degeneracies.

4.1.5 Sequential learning

The maximum likelihood solution (4.14) involves processing the entire training set in one go and is known as a *batch* method. This can become computationally costly for large data sets. If the data set is sufficiently large, it may be worthwhile to use *sequential* algorithms, also known as *online* algorithms, in which the data points are considered one at a time and the model parameters updated after each such presentation. Sequential learning is also appropriate for real-time applications in which the data observations arrive in a continuous stream and predictions must be

Telegram: @uni_k

www.konkur.in

made before all the data points are seen.

We can obtain a sequential learning algorithm by applying the technique of *stochastic gradient descent*, also known as *sequential gradient descent*, as follows. If the error function comprises a sum over data points $E = \sum_n E_n$, then after presentation of data point n, the stochastic gradient descent algorithm updates the parameter vector $\mathbf{w}$ using

Chapter 7

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta \nabla E_n \tag{4.21}$$

where τ denotes the iteration number, and η is a suitably chosen learning rate parameter. The value of $\mathbf{w}$ is initialized to some starting vector $\mathbf{w}^{(0)}$. For the sum-of-squares error function (4.11), this gives

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} + \eta(t_n - \mathbf{w}^{(\tau)\mathrm{T}}\boldsymbol{\phi}_n)\boldsymbol{\phi}_n \tag{4.22}$$

where $\boldsymbol{\phi}_n = \boldsymbol{\phi}(\mathbf{x}_n)$. This is known as the *least-mean-squares* or the *LMS algorithm*.

4.1.6 Regularized least squares

Section 1.2

We have previously introduced the idea of adding a regularization term to an error function to control over-fitting, so that the total error function to be minimized takes the form

$$E_D(\mathbf{w}) + \lambda E_W(\mathbf{w}) \tag{4.23}$$

where λ is the regularization coefficient that controls the relative importance of the data-dependent error $E_D(\mathbf{w})$ and the regularization term $E_W(\mathbf{w})$. One of the simplest forms of regularizer is given by the sum of the squares of the weight vector elements:

$$E_W(\mathbf{w}) = \frac{1}{2}\sum_j w_j^2 = \frac{1}{2}\mathbf{w}^{\mathrm{T}}\mathbf{w}. \tag{4.24}$$

If we also consider the sum-of-squares error function given by

$$E_D(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N}\{t_n - \mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}_n)\}^2, \tag{4.25}$$

then the total error function becomes

$$\frac{1}{2}\sum_{n=1}^{N}\{t_n - \mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}_n)\}^2 + \frac{\lambda}{2}\mathbf{w}^{\mathrm{T}}\mathbf{w}. \tag{4.26}$$

In statistics, this regularizer provides an example of a *parameter shrinkage* method because it shrinks parameter values towards zero. It has the advantage that the error function remains a quadratic function of $\mathbf{w}$, and so its exact minimizer can be found in closed form. Specifically, setting the gradient of (4.26) with respect to $\mathbf{w}$ to zero and solving for $\mathbf{w}$ as before, we obtain

Exercise 4.6

$$\mathbf{w} = \left(\lambda\mathbf{I} + \boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\Phi}\right)^{-1}\boldsymbol{\Phi}^{\mathrm{T}}\mathbf{t}. \tag{4.27}$$

This represents a simple extension of the least-squares solution (4.14).

Telegram: @uni_k

www.konkur.in

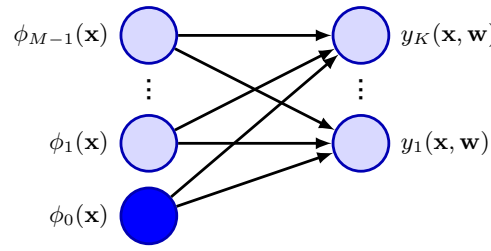


Figure 4.4 Representation of a linear regression model as a neural network having a single layer of connections. Each basis function is represented by a node, with the solid node representing the 'bias' basis function ϕ_0. Likewise each output $y_1, \ldots, y_K$ is represented by a node. The links between the nodes represent the corresponding weight and bias parameters.

4.1.7 Multiple outputs

So far, we have considered situations with a single target variable t. In some applications, we may wish to predict $K > 1$ target variables, which we denote collectively by the target vector $\mathbf{t} = (t_1, \ldots, t_K)^{\mathrm{T}}$. This could be done by introducing a different set of basis functions for each component of $\mathbf{t}$, leading to multiple, independent regression problems. However, a more common approach is to use the same set of basis functions to model all of the components the target vector so that

$$\mathbf{y}(\mathbf{x}, \mathbf{w}) = \mathbf{W}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}) \tag{4.28}$$

where $\mathbf{y}$ is a K-dimensional column vector, $\mathbf{W}$ is an $M \times K$ matrix of parameters, and $\boldsymbol{\phi}(\mathbf{x})$ is an M-dimensional column vector with elements $\phi_j(\mathbf{x})$ with $\phi_0(\mathbf{x}) = 1$ as before. Again, this can be represented as a neural network having a single layer of parameters, as shown in Figure 4.4.

Suppose we take the conditional distribution of the target vector to be an isotropic Gaussian of the form

$$p(\mathbf{t}|\mathbf{x}, \mathbf{W}, \sigma^2) = \mathcal{N}(\mathbf{t}|\mathbf{W}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}), \sigma^2\mathbf{I}). \tag{4.29}$$

If we have a set of observations $\mathbf{t}_1, \ldots, \mathbf{t}_N$, we can combine these into a matrix $\mathbf{T}$ of size $N \times K$ such that the nth row is given by $\mathbf{t}_n^{\mathrm{T}}$. Similarly, we can combine the input vectors $\mathbf{x}_1, \ldots, \mathbf{x}_N$ into a matrix $\mathbf{X}$. The log likelihood function is then given by

$$\begin{aligned}\ln p(\mathbf{T}|\mathbf{X}, \mathbf{W}, \sigma^2) &= \sum_{n=1}^{N} \ln \mathcal{N}(\mathbf{t}_n|\mathbf{W}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}_n), \sigma^2\mathbf{I}) \\ &= -\frac{NK}{2}\ln\left(2\pi\sigma^2\right) - \frac{1}{2\sigma^2}\sum_{n=1}^{N}\left\|\mathbf{t}_n - \mathbf{W}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}_n)\right\|^2. \end{aligned} \tag{4.30}$$

As before, we can maximize this function with respect to $\mathbf{W}$, giving

$$\mathbf{W}_{\mathrm{ML}} = \left(\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\Phi}\right)^{-1}\boldsymbol{\Phi}^{\mathrm{T}}\mathbf{T} \tag{4.31}$$

where we have combined the input feature vectors $\boldsymbol{\phi}(\mathbf{x}_1), \ldots, \boldsymbol{\phi}(\mathbf{x}_N)$ into a matrix $\boldsymbol{\Phi}$. If we examine this result for each target variable t_k, we have

$$\mathbf{w}_k = \left(\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\Phi}\right)^{-1}\boldsymbol{\Phi}^{\mathrm{T}}\mathbf{t}_k = \boldsymbol{\Phi}^{\dagger}\mathbf{t}_k \tag{4.32}$$

Telegram: @uni_k

www.konkur.in

where $\mathbf{t}_k$ is an N-dimensional column vector with components t_{nk} for $n = 1, \ldots N$. Thus, the solution to the regression problem decouples between the different target variables, and we need compute only a single pseudo-inverse matrix $\mathbf{\Phi}^\dagger$, which is shared by all the vectors $\mathbf{w}_k$.

Exercise 4.7

The extension to general Gaussian noise distributions having arbitrary covariance matrices is straightforward. Again, this leads to a decoupling into K independent regression problems. This result is unsurprising because the parameters $\mathbf{W}$ define only the mean of the Gaussian noise distribution, and we know that the maximum likelihood solution for the mean of a multivariate Gaussian is independent of the covariance. From now on, we will therefore consider a single target variable t for simplicity.

Section 3.2.7

4.2. Decision theory

We have formulated the regression task as one of modelling a conditional probability distribution $p(t|\mathbf{x})$, and we have chosen a specific form for the conditional probability, namely a Gaussian (4.8) with an $\mathbf{x}$-dependent mean $y(\mathbf{x}, \mathbf{w})$ governed by parameters $\mathbf{w}$ and with variance given by the parameter σ^2. Both $\mathbf{w}$ and σ^2 can be learned from data using maximum likelihood. The result is a *predictive distribution* given by

$$p(t|\mathbf{x}, \mathbf{w}_{\mathrm{ML}}, \sigma^2_{\mathrm{ML}}) = \mathcal{N}(t|y(\mathbf{x}, \mathbf{w}_{\mathrm{ML}}), \sigma^2_{\mathrm{ML}}). \tag{4.33}$$

The predictive distribution expresses our uncertainty over the value of t for some new input $\mathbf{x}$. However, for many practical applications we need to predict a specific value for t rather than returning an entire distribution, particularly where we must take a specific action. For example, if our goal is to determine the optimal level of radiation to use for treating a tumour and our model predicts a probability distribution over radiation dose, then we must use that distribution to decide the specific dose to be administered. Our task therefore breaks down into two stages. In the first stage, called the *inference* stage, we use the training data to determine a predictive distribution $p(t|\mathbf{x})$. In the second stage, known as the *decision* stage, we use this predictive distribution to determine a specific value $f(\mathbf{x})$, which will be dependent on the input vector $\mathbf{x}$, that is optimal according to some criterion. We can do this by minimizing a *loss function* that depends on both the predictive distribution $p(t|\mathbf{x})$ and on f.

Intuitively we might choose the mean of the conditional distribution, so that we would use $f(\mathbf{x}) = y(\mathbf{x}, \mathbf{w}_{\mathrm{ML}})$. In some cases this intuition will be correct, but in other situations it can give very poor results. It is therefore useful to formalize this so that we can understand when it applies and under what assumptions, and the framework for doing this is called *decision theory*.

Suppose that we choose a value $f(\mathbf{x})$ for our prediction when the true value is t. In doing so, we incur some form of penalty or cost. This is determined by a *loss*, which we denote $L(t, f(\mathbf{x}))$. Of course, we do not know the true value of t, so instead of minimizing L itself, we minimize the average, or expected, loss which is

Telegram: @uni_k

www.konkur.in

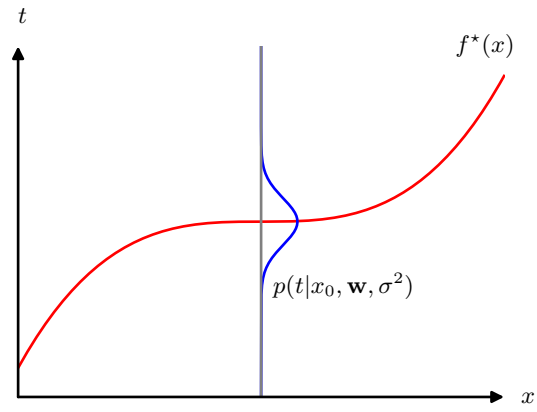


Figure 4.5 The regression function $f^\star(x)$, which minimizes the expected squared loss, is given by the mean of the conditional distribution $p(t|x)$.

given by

$$\mathbb{E}[L] = \iint L(t, f(\mathbf{x}))p(\mathbf{x}, t)\,\mathrm{d}\mathbf{x}\,\mathrm{d}t \tag{4.34}$$

where we are averaging over the distribution of both input and target variables, weighted by their joint distribution $p(\mathbf{x}, t)$. A common choice of loss function in regression problems is the squared loss given by $L(t, f(\mathbf{x})) = \{f(\mathbf{x}) - t\}^2$. In this case, the expected loss can be written

$$\mathbb{E}[L] = \iint \{f(\mathbf{x}) - t\}^2 p(\mathbf{x}, t)\,\mathrm{d}\mathbf{x}\,\mathrm{d}t. \tag{4.35}$$

It is important not to confuse the squared-loss function with the sum-of-squares error function introduced earlier. The error function is used to set the parameters during training in order to determine the conditional probability distribution $p(t|\mathbf{x})$, whereas the loss function governs how the conditional distribution is used to arrive at a predictive function $f(\mathbf{x})$ specifying a prediction for each value of $\mathbf{x}$.

Appendix B

Our goal is to choose $f(\mathbf{x})$ so as to minimize $\mathbb{E}[L]$. If we assume a completely flexible function $f(\mathbf{x})$, we can do this formally using the calculus of variations to give

$$\frac{\delta \mathbb{E}[L]}{\delta f(\mathbf{x})} = 2\int \{f(\mathbf{x}) - t\}p(\mathbf{x}, t)\,\mathrm{d}t = 0. \tag{4.36}$$

Solving for $f(\mathbf{x})$ and using the sum and product rules of probability, we obtain

$$f^\star(\mathbf{x}) = \frac{1}{p(\mathbf{x})}\int tp(\mathbf{x}, t)\,\mathrm{d}t = \int tp(t|\mathbf{x})\,\mathrm{d}t = \mathbb{E}_t[t|\mathbf{x}], \tag{4.37}$$

which is the conditional average of t conditioned on $\mathbf{x}$ and is known as the *regression function*. This result is illustrated in Figure 4.5. It can readily be extended to multiple target variables represented by the vector $\mathbf{t}$, in which case the optimal solution is the conditional average $\mathbf{f}^\star(\mathbf{x}) = \mathbb{E}_t[\mathbf{t}|\mathbf{x}]$. For a Gaussian conditional distribution of the

Exercise 4.8

Telegram: @uni_k

www.konkur.in

form (4.8), the conditional mean will be simply

$$\mathbb{E}[t|\mathbf{x}] = \int tp(t|\mathbf{x})\,\mathrm{d}t = y(\mathbf{x}, \mathbf{w}). \tag{4.38}$$

The use of calculus of variations to derive (4.37) implies that we are optimizing over all possible functions $f(\mathbf{x})$. Although any parametric model that we can implement in practice is limited in the range of functions that it can represent, the framework of deep neural networks, discussed extensively in later chapters, provides a highly flexible class of functions that, for many practical purposes, can approximate any desired function to high accuracy.

We can derive this result in a slightly different way, which will also shed light on the nature of the regression problem. Armed with the knowledge that the optimal solution is the conditional expectation, we can expand the square term as follows

$$\begin{aligned}\{f(\mathbf{x}) - t\}^2 &= \{f(\mathbf{x}) - \mathbb{E}[t|\mathbf{x}] + \mathbb{E}[t|\mathbf{x}] - t\}^2 \\ &= \{f(\mathbf{x}) - \mathbb{E}[t|\mathbf{x}]\}^2 + 2\{f(\mathbf{x}) - \mathbb{E}[t|\mathbf{x}]\}\{\mathbb{E}[t|\mathbf{x}] - t\} + \{\mathbb{E}[t|\mathbf{x}] - t\}^2\end{aligned}$$

where, to keep the notation uncluttered, we use $\mathbb{E}[t|\mathbf{x}]$ to denote $\mathbb{E}_t[t|\mathbf{x}]$. Substituting into the loss function (4.35) and performing the integral over t, we see that the cross-term vanishes and we obtain an expression for the loss function in the form

$$\mathbb{E}[L] = \int \{f(\mathbf{x}) - \mathbb{E}[t|\mathbf{x}]\}^2 p(\mathbf{x})\,\mathrm{d}\mathbf{x} + \int \mathrm{var}\,[t|\mathbf{x}]\, p(\mathbf{x})\,\mathrm{d}\mathbf{x}. \tag{4.39}$$

The function $f(\mathbf{x})$ we seek to determine appears only in the first term, which will be minimized when $f(\mathbf{x})$ is equal to $\mathbb{E}[t|\mathbf{x}]$, in which case this term will vanish. This is simply the result that we derived previously, and shows that the optimal least-squares predictor is given by the conditional mean. The second term is the variance of the distribution of t, averaged over $\mathbf{x}$, and represents the intrinsic variability of the target data and can be regarded as noise. Because it is independent of $f(\mathbf{x})$, it represents the irreducible minimum value of the loss function.

The squared loss is not the only possible choice of loss function for regression. Here we consider briefly one simple generalization of the squared loss, called the *Minkowski* loss, whose expectation is given by

$$\mathbb{E}[L_q] = \iint |f(\mathbf{x}) - t|^q p(\mathbf{x}, t)\,\mathrm{d}\mathbf{x}\,\mathrm{d}t, \tag{4.40}$$

which reduces to the expected squared loss for $q = 2$. The function $|f - t|^q$ is plotted against $f - t$ for various values of q in Figure 4.6. The minimum of $\mathbb{E}[L_q]$ is given by the conditional mean for $q = 2$, the conditional median for $q = 1$, and the conditional mode for $q \to 0$.

Exercise 4.12

Note that the Gaussian noise assumption implies that the conditional distribution of t given $\mathbf{x}$ is unimodal, which may be inappropriate for some applications. In this case a squared loss can lead to very poor results and we need to develop more sophisticated approaches. For example, we can extend this model by using mixtures

Telegram: @uni_k

www.konkur.in

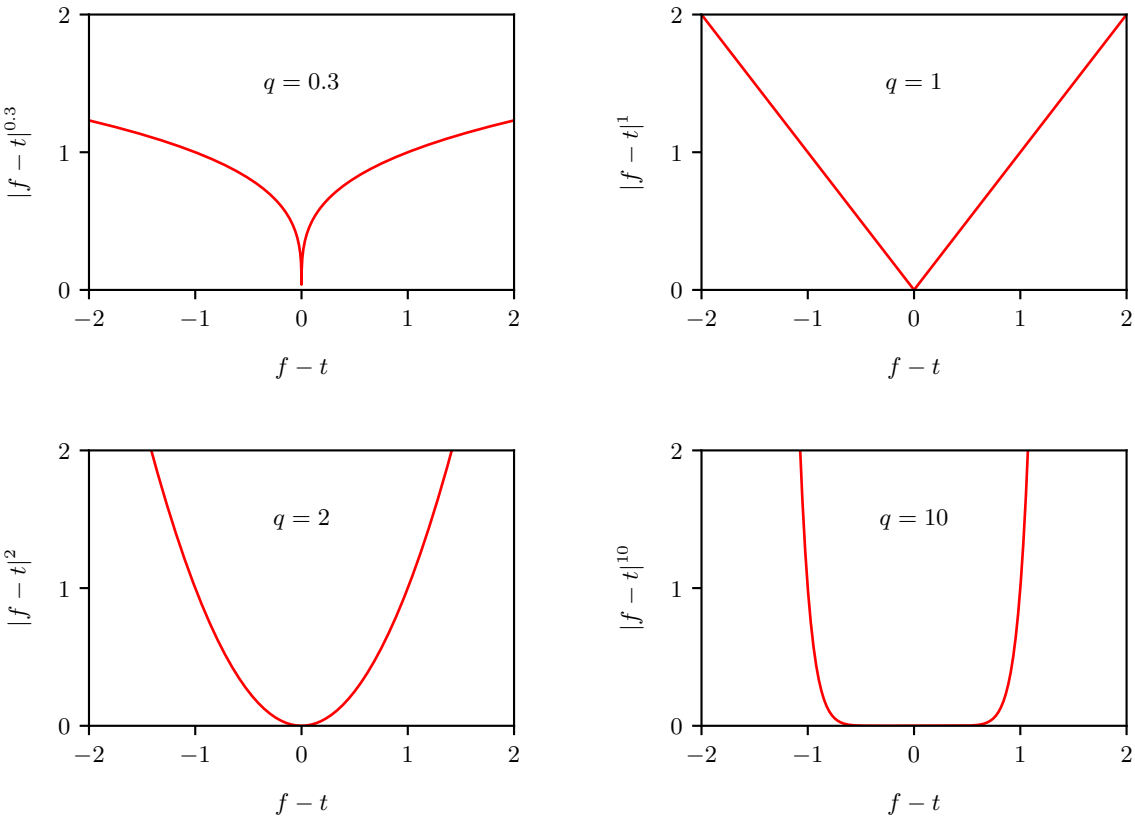


Figure 4.6 Plots of the quantity $L_q = |f - t|^q$ for various values of q.

Section 6.5 of Gaussians to give multimodal conditional distributions, which often arise in the solution of *inverse problems*. Our focus in this section has been on decision theory for regression problems, and in the next chapter we shall develop analogous concepts for classification tasks. *Section 5.2*

4.3. The Bias–Variance Trade-off

Section 1.2 So far in our discussion of linear models for regression, we have assumed that the form and number of basis functions are both given. We have also seen that the use of maximum likelihood can lead to severe over-fitting if complex models are trained using data sets of limited size. However, limiting the number of basis functions to avoid over-fitting has the side effect of limiting the flexibility of the model to capture interesting and important trends in the data. Although a regularization term can control over-fitting for models with many parameters, this raises the question of how to determine a suitable value for the regularization coefficient λ. Seeking the

Telegram: @uni_k

www.konkur.in

solution that minimizes the regularized error function with respect to both the weight vector $\mathbf{w}$ and the regularization coefficient λ is clearly not the right approach, since this leads to the unregularized solution with $\lambda = 0$.

It is instructive to consider a frequentist viewpoint of the model complexity issue, known as the *bias–variance* trade-off. Although we will introduce this concept in the context of linear basis function models, where it is easy to illustrate the ideas using simple examples, the discussion has very general applicability. Note, however, that over-fitting is really an unfortunate property of maximum likelihood and does not arise when we marginalize over parameters in a Bayesian setting (Bishop, 2006).

Section 4.2

When we discussed decision theory for regression problems, we considered various loss functions, each of which leads to a corresponding optimal prediction once we are given the conditional distribution $p(t|\mathbf{x})$. A popular choice is the squared-loss function, for which the optimal prediction is given by the conditional expectation, which we denote by $h(\mathbf{x})$ and is given by

$$h(\mathbf{x}) = \mathbb{E}[t|\mathbf{x}] = \int t p(t|\mathbf{x})\,\mathrm{d}t. \tag{4.41}$$

We have also seen that the expected squared loss can be written in the form

$$\mathbb{E}[L] = \int \{f(\mathbf{x}) - h(\mathbf{x})\}^2 p(\mathbf{x})\,\mathrm{d}\mathbf{x} + \iint \{h(\mathbf{x}) - t\}^2 p(\mathbf{x}, t)\,\mathrm{d}\mathbf{x}\,\mathrm{d}t. \tag{4.42}$$

Recall that the second term, which is independent of $f(\mathbf{x})$, arises from the intrinsic noise on the data and represents the minimum achievable value of the expected loss. The first term depends on our choice for the function $f(\mathbf{x})$, and we will seek a solution for $f(\mathbf{x})$ that makes this term a minimum. Because it is non-negative, the smallest value that we can hope to achieve for this term is zero. If we had an unlimited supply of data (and unlimited computational resources), we could in principle find the regression function $h(\mathbf{x})$ to any desired degree of accuracy, and this would represent the optimal choice for $f(\mathbf{x})$. However, in practice we have a data set $\mathcal{D}$ containing only a finite number N of data points, and consequently, we cannot know the regression function $h(\mathbf{x})$ exactly.

If we were to model $h(\mathbf{x})$ using a function governed by a parameter vector $\mathbf{w}$, then from a Bayesian perspective, the uncertainty in our model would be expressed through a posterior distribution over $\mathbf{w}$. A frequentist treatment, however, involves making a point estimate of $\mathbf{w}$ based on the data set $\mathcal{D}$ and tries instead to interpret the uncertainty of this estimate through the following thought experiment. Suppose we had a large number of data sets each of size N and each drawn independently from the distribution $p(t, \mathbf{x})$. For any given data set $\mathcal{D}$, we can run our learning algorithm and obtain a prediction function $f(\mathbf{x}; \mathcal{D})$. Different data sets from the ensemble will give different functions and consequently different values of the squared loss. The performance of a particular learning algorithm is then assessed by taking the average over this ensemble of data sets.

Consider the integrand of the first term in (4.42), which for a particular data set $\mathcal{D}$ takes the form

$$\{f(\mathbf{x}; \mathcal{D}) - h(\mathbf{x})\}^2. \tag{4.43}$$

Telegram: @uni_k

www.konkur.in

Because this quantity will be dependent on the particular data set $\mathcal{D}$, we take its average over the ensemble of data sets. If we add and subtract the quantity $\mathbb{E}_{\mathcal{D}}[f(\mathbf{x};\mathcal{D})]$ inside the braces, and then expand, we obtain

$$\begin{aligned}&\{f(\mathbf{x};\mathcal{D}) - \mathbb{E}_{\mathcal{D}}[f(\mathbf{x};\mathcal{D})] + \mathbb{E}_{\mathcal{D}}[f(\mathbf{x};\mathcal{D})] - h(\mathbf{x})\}^2 \\ &= \{f(\mathbf{x};\mathcal{D}) - \mathbb{E}_{\mathcal{D}}[f(\mathbf{x};\mathcal{D})]\}^2 + \{\mathbb{E}_{\mathcal{D}}[f(\mathbf{x};\mathcal{D})] - h(\mathbf{x})\}^2 \\ &+ 2\{f(\mathbf{x};\mathcal{D}) - \mathbb{E}_{\mathcal{D}}[f(\mathbf{x};\mathcal{D})]\}\{\mathbb{E}_{\mathcal{D}}[f(\mathbf{x};\mathcal{D})] - h(\mathbf{x})\}. \end{aligned} \tag{4.44}$$

We now take the expectation of this expression with respect to $\mathcal{D}$ and note that the final term will vanish, giving

$$\begin{aligned}&\mathbb{E}_{\mathcal{D}}\left[\{f(\mathbf{x};\mathcal{D}) - h(\mathbf{x})\}^2\right] \\ &= \underbrace{\{\mathbb{E}_{\mathcal{D}}[f(\mathbf{x};\mathcal{D})] - h(\mathbf{x})\}^2}_{(\text{bias})^2} + \underbrace{\mathbb{E}_{\mathcal{D}}\left[\{f(\mathbf{x};\mathcal{D}) - \mathbb{E}_{\mathcal{D}}[f(\mathbf{x};\mathcal{D})]\}^2\right]}_{\text{variance}}. \end{aligned} \tag{4.45}$$

We see that the expected squared difference between $f(\mathbf{x};\mathcal{D})$ and the regression function $h(\mathbf{x})$ can be expressed as the sum of two terms. The first term, called the squared *bias*, represents the extent to which the average prediction over all data sets differs from the desired regression function. The second term, called the *variance*, measures the extent to which the solutions for individual data sets vary around their average, and hence, this measures the extent to which the function $f(\mathbf{x};\mathcal{D})$ is sensitive to the particular choice of data set. We will provide some intuition to support these definitions shortly when we consider a simple example.

So far, we have considered a single input value $\mathbf{x}$. If we substitute this expansion back into (4.42), we obtain the following decomposition of the expected squared loss:

$$\text{expected loss} = (\text{bias})^2 + \text{variance} + \text{noise} \tag{4.46}$$

where

$$(\text{bias})^2 = \int \{\mathbb{E}_{\mathcal{D}}[f(\mathbf{x};\mathcal{D})] - h(\mathbf{x})\}^2 p(\mathbf{x})\,\mathrm{d}\mathbf{x} \tag{4.47}$$

$$\text{variance} = \int \mathbb{E}_{\mathcal{D}}\left[\{f(\mathbf{x};\mathcal{D}) - \mathbb{E}_{\mathcal{D}}[f(\mathbf{x};\mathcal{D})]\}^2\right] p(\mathbf{x})\,\mathrm{d}\mathbf{x} \tag{4.48}$$

$$\text{noise} = \iint \{h(\mathbf{x}) - t\}^2 p(\mathbf{x},t)\,\mathrm{d}\mathbf{x}\,\mathrm{d}t \tag{4.49}$$

and the bias and variance terms now refer to integrated quantities.

Our goal is to minimize the expected loss, which we have decomposed into the sum of a (squared) bias, a variance, and a constant noise term. As we will see, there is a trade-off between bias and variance, with very flexible models having low bias and high variance, and relatively rigid models having high bias and low variance. The model with the optimal predictive capability is the one that leads to the best balance between bias and variance. This is illustrated by considering the sinusoidal data set introduced earlier. Here we independently generate 100 data sets, each containing

Section 1.2

Telegram: @uni_k

www.konkur.in

$N = 25$ data points, from the sinusoidal curve $h(x) = \sin(2\pi x)$. The data sets are indexed by $l = 1, \ldots, L$, where $L = 100$. For each data set $\mathcal{D}^{(l)}$, we fit a model with $M = 24$ Gaussian basis functions along with a constant 'bias' basis function to give a total of 25 parameters. By minimizing the regularized error function (4.26), we obtain a prediction function $f^{(l)}(x)$, as shown in Figure 4.7.

The top row corresponds to a large value of the regularization coefficient λ that gives low variance (because the red curves in the left plot look similar) but high bias (because the two curves in the right plot are very different). Conversely on the bottom row, for which λ is small, there is large variance (shown by the high variability between the red curves in the left plot) but low bias (shown by the good fit between the average model fit and the original sinusoidal function). Note that the result of averaging many solutions for the complex model with $M = 25$ is a very good fit to the regression function, which suggests that averaging may be a beneficial procedure. Indeed, a weighted averaging of multiple solutions lies at the heart of a Bayesian approach, although the averaging is with respect to the posterior distribution of parameters, not with respect to multiple data sets.

We can also examine the bias–variance trade-off quantitatively for this example. The average prediction is estimated from

$$\overline{f}(x) = \frac{1}{L}\sum_{l=1}^{L} f^{(l)}(x), \tag{4.50}$$

and the integrated squared bias and integrated variance are then given by

$$(\text{bias})^2 = \frac{1}{N}\sum_{n=1}^{N}\left\{\overline{f}(x_n) - h(x_n)\right\}^2 \tag{4.51}$$

$$\text{variance} = \frac{1}{N}\sum_{n=1}^{N}\frac{1}{L}\sum_{l=1}^{L}\left\{f^{(l)}(x_n) - \overline{f}(x_n)\right\}^2 \tag{4.52}$$

where the integral over x, weighted by the distribution $p(x)$, is approximated by a finite sum over data points drawn from that distribution. These quantities, along with their sum, are plotted as a function of $\ln \lambda$ in Figure 4.8. We see that small values of λ allow the model to become finely tuned to the noise on each individual data set leading to large variance. Conversely, a large value of λ pulls the weight parameters towards zero leading to large bias.

Note that the bias–variance decomposition is of limited practical value because it is based on averages with respect to ensembles of data sets, whereas in practice we have only the single observed data set. If we had a large number of independent training sets of a given size, we would be better off combining them into a single larger training set, which of course would reduce the level of over-fitting for a given model complexity. Nevertheless, the bias–variance decomposition often provides useful insights into the model complexity issue, and although we have introduced it in this chapter from the perspective of regression problems, the underlying intuition has broad applicability.

Telegram: @uni_k

www.konkur.in

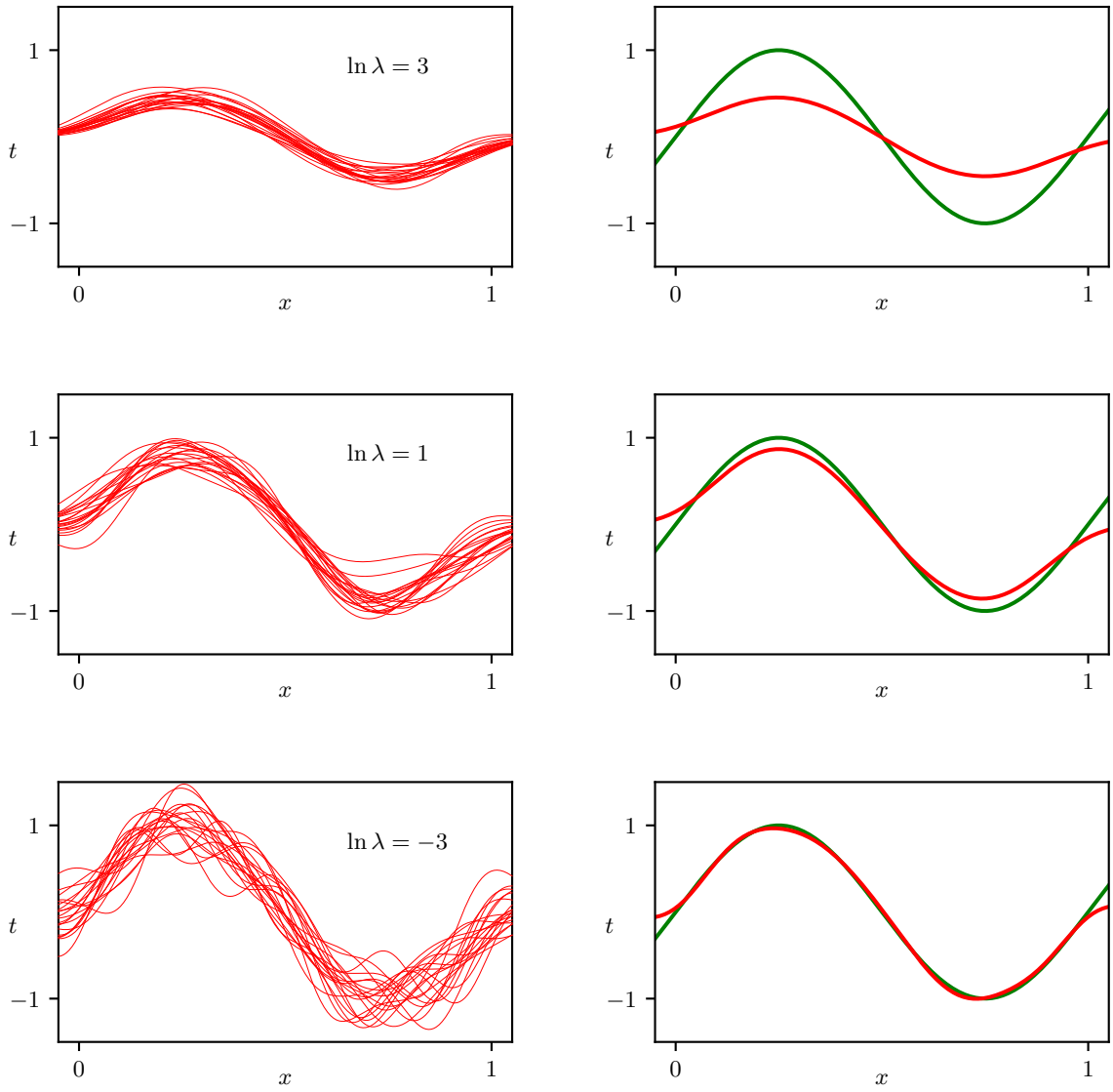


Figure 4.7 Illustration of the dependence of bias and variance on model complexity governed by a regularization parameter λ, using the sinusoidal data from Chapter 1. There are $L = 100$ data sets, each having $N = 25$ data points, and there are 24 Gaussian basis functions in the model so that the total number of parameters is $M = 25$ including the bias parameter. The left column shows the result of fitting the model to the data sets for various values of $\ln \lambda$ (for clarity, only 20 of the 100 fits are shown). The right column shows the corresponding average of the 100 fits (red) along with the sinusoidal function from which the data sets were generated (green).

Telegram: @uni_k

www.konkur.in

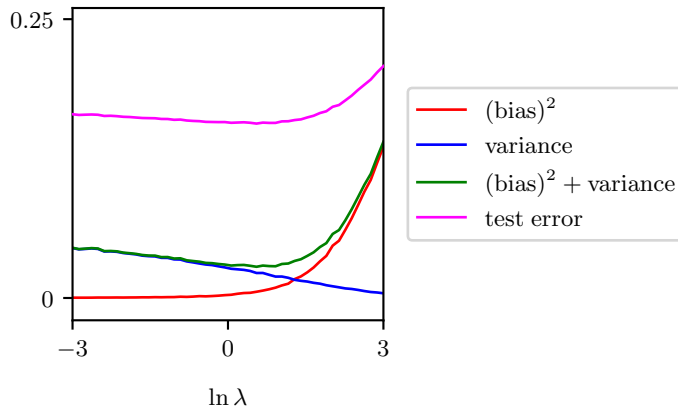


Figure 4.8 Plot of squared bias and variance, together with their sum, corresponding to the results shown in Figure 4.7. Also shown is the average test set error for a test data set size of 1,000 points. The minimum value of $(\text{bias})^2 + \text{variance}$ occurs around $\ln \lambda = 0.43$, which is close to the value that gives the minimum error on the test data.

Exercises

4.1 ($\star$) Consider the sum-of-squares error function given by (1.2) in which the function $y(x, \mathbf{w})$ is given by the polynomial (1.1). Show that the coefficients $\mathbf{w} = \{w_i\}$ that minimize this error function are given by the solution to the following set of linear equations:

$$\sum_{j=0}^{M} A_{ij} w_j = T_i \tag{4.53}$$

where

$$A_{ij} = \sum_{n=1}^{N} (x_n)^{i+j}, \qquad T_i = \sum_{n=1}^{N} (x_n)^i t_n. \tag{4.54}$$

Here a suffix i or j denotes the index of a component, whereas $(x)^i$ denotes x raised to the power of i.

4.2 ($\star$) Write down the set of coupled linear equations, analogous to (4.53), satisfied by the coefficients w_i that minimize the regularized sum-of-squares error function given by (1.4).

4.3 ($\star$) Show that the $\tanh$ function defined by

$$\tanh(a) = \frac{e^a - e^{-a}}{e^a + e^{-a}} \tag{4.55}$$

and the logistic sigmoid function defined by (4.6) are related by

$$\tanh(a) = 2\sigma(2a) - 1. \tag{4.56}$$

Hence, show that a general linear combination of logistic sigmoid functions of the form

$$y(x, \mathbf{w}) = w_0 + \sum_{j=1}^{M} w_j \sigma\left(\frac{x - \mu_j}{s}\right) \tag{4.57}$$

Telegram: @uni_k

www.konkur.in

is equivalent to a linear combination of tanh functions of the form

$$y(x, \mathbf{u}) = u_0 + \sum_{j=1}^{M} u_j \tanh\left(\frac{x - \mu_j}{2s}\right) \tag{4.58}$$

and find expressions to relate the new parameters $\{u_1, \ldots, u_M\}$ to the original parameters $\{w_1, \ldots, w_M\}$.

4.4 ($\star\star\star$) Show that the matrix

$$\boldsymbol{\Phi}(\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\Phi})^{-1}\boldsymbol{\Phi}^{\mathrm{T}} \tag{4.59}$$

takes any vector $\mathbf{v}$ and projects it onto the space spanned by the columns of $\boldsymbol{\Phi}$. Use this result to show that the least-squares solution (4.14) corresponds to an orthogonal projection of the vector $\mathbf{t}$ onto the manifold $\mathcal{S}$, as shown in Figure 4.3.

4.5 ($\star$) Consider a data set in which each data point t_n is associated with a weighting factor $r_n > 0$, so that the sum-of-squares error function becomes

$$E_D(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N} r_n \left\{t_n - \mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}_n)\right\}^2. \tag{4.60}$$

Find an expression for the solution $\mathbf{w}^\star$ that minimizes this error function. Give two alternative interpretations of the weighted sum-of-squares error function in terms of (i) data-dependent noise variance and (ii) replicated data points.

4.6 ($\star$) By setting the gradient of (4.26) with respect to $\mathbf{w}$ to zero, show that the exact minimum of the regularized sum-of-squares error function for linear regression is given by (4.27).

4.7 ($\star\star$) Consider a linear basis function regression model for a multivariate target variable $\mathbf{t}$ having a Gaussian distribution of the form

$$p(\mathbf{t}|\mathbf{W}, \boldsymbol{\Sigma}) = \mathcal{N}(\mathbf{t}|\mathbf{y}(\mathbf{x}, \mathbf{W}), \boldsymbol{\Sigma}) \tag{4.61}$$

where

$$\mathbf{y}(\mathbf{x}, \mathbf{W}) = \mathbf{W}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}) \tag{4.62}$$

together with a training data set comprising input basis vectors $\boldsymbol{\phi}(\mathbf{x}_n)$ and corresponding target vectors $\mathbf{t}_n$, with $n = 1, \ldots, N$. Show that the maximum likelihood solution $\mathbf{W}_{\mathrm{ML}}$ for the parameter matrix $\mathbf{W}$ has the property that each column is given by an expression of the form (4.14), which was the solution for an isotropic noise distribution. Note that this is independent of the covariance matrix $\boldsymbol{\Sigma}$. Show that the maximum likelihood solution for $\boldsymbol{\Sigma}$ is given by

$$\boldsymbol{\Sigma} = \frac{1}{N}\sum_{n=1}^{N} \left(\mathbf{t}_n - \mathbf{W}_{\mathrm{ML}}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}_n)\right)\left(\mathbf{t}_n - \mathbf{W}_{\mathrm{ML}}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}_n)\right)^{\mathrm{T}}. \tag{4.63}$$

Telegram: @uni_k

www.konkur.in

4.8 ($\star$) Consider the generalization of the squared-loss function (4.35) for a single target variable t to multiple target variables described by the vector $\mathbf{t}$ given by

$$\mathbb{E}[L(\mathbf{t}, \mathbf{f}(\mathbf{x}))] = \iint \|\mathbf{f}(\mathbf{x}) - \mathbf{t}\|^2 p(\mathbf{x}, \mathbf{t})\, \mathrm{d}\mathbf{x}\, \mathrm{d}\mathbf{t}. \tag{4.64}$$

Using the calculus of variations, show that the function $\mathbf{f}(\mathbf{x})$ for which this expected loss is minimized is given by

$$\mathbf{f}(\mathbf{x}) = \mathbb{E}_t[\mathbf{t}|\mathbf{x}]. \tag{4.65}$$

4.9 ($\star$) By expansion of the square in (4.64), derive a result analogous to (4.39) and, hence, show that the function $\mathbf{f}(\mathbf{x})$ that minimizes the expected squared loss for a vector $\mathbf{t}$ of target variables is again given by the conditional expectation of $\mathbf{t}$ in the form (4.65).

4.10 ($\star\star$) Rederive the result (4.65) by first expanding (4.64) analogous to (4.39).

4.11 ($\star\star$) The following distribution

$$p(x|\sigma^2, q) = \frac{q}{2(2\sigma^2)^{1/q}\Gamma(1/q)} \exp\left(-\frac{|x|^q}{2\sigma^2}\right) \tag{4.66}$$

is a generalization of the univariate Gaussian distribution. Here $\Gamma(x)$ is the gamma function defined by

$$\Gamma(x) = \int_{-\infty}^{\infty} u^{x-1} e^{-u}\, \mathrm{d}u. \tag{4.67}$$

Show that this distribution is normalized so that

$$\int_{-\infty}^{\infty} p(x|\sigma^2, q)\, \mathrm{d}x = 1 \tag{4.68}$$

and that it reduces to the Gaussian when $q = 2$. Consider a regression model in which the target variable is given by $t = y(\mathbf{x}, \mathbf{w}) + \epsilon$ and ϵ is a random noise variable drawn from the distribution (4.66). Show that the log likelihood function over $\mathbf{w}$ and σ^2, for an observed data set of input vectors $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$ and corresponding target variables $\mathbf{t} = (t_1, \ldots, t_N)^{\mathrm{T}}$, is given by

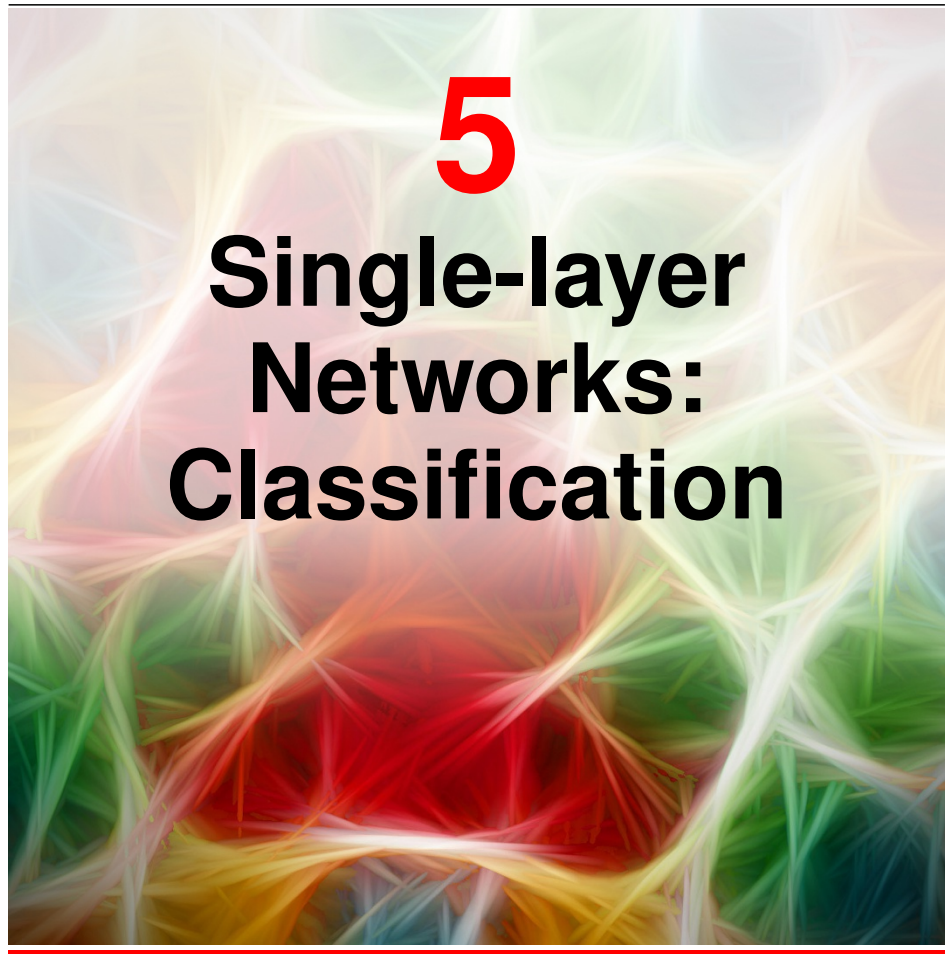
$$\ln p(\mathbf{t}|\mathbf{X}, \mathbf{w}, \sigma^2) = -\frac{1}{2\sigma^2} \sum_{n=1}^{N} |y(\mathbf{x}_n, \mathbf{w}) - t_n|^q - \frac{N}{q} \ln(2\sigma^2) + \text{const} \tag{4.69}$$

where 'const' denotes terms independent of both $\mathbf{w}$ and σ^2. Note that, as a function of $\mathbf{w}$, this is the L_q error function considered in Section 4.2.

4.12 ($\star\star$) Consider the expected loss for regression problems under the L_q loss function given by (4.40). Write down the condition that $y(\mathbf{x})$ must satisfy to minimize $\mathbb{E}[L_q]$. Show that, for $q = 1$, this solution represents the conditional median, i.e., the function $y(\mathbf{x})$ such that the probability mass for $t < y(\mathbf{x})$ is the same as for $t \geqslant y(\mathbf{x})$. Also show that the minimum expected L_q loss for $q \to 0$ is given by the conditional mode, i.e., by the function $y(\mathbf{x})$ being equal to the value of t that maximizes $p(t|\mathbf{x})$ for each $\mathbf{x}$.

Telegram: @uni_k

www.konkur.in



5 Single-layer Networks: Classification

In the previous chapter, we explored a class of regression models in which the output variables were linear functions of the model parameters and which can therefore be expressed as simple neural networks having a single layer of weight and bias parameters. We turn now to a discussion of classification problems, and in this chapter, we will focus on an analogous class of models that again can be expressed as single-layer neural networks. These will allow us to introduce many of the key concepts of classification before dealing with more general deep neural networks in later chapters.

The goal in classification is to take an input vector $\mathbf{x} \in \mathbb{R}^D$ and assign it to one of K discrete classes $\mathcal{C}_k$ where $k = 1, \ldots, K$. In the most common scenario, the classes are taken to be disjoint, so that each input is assigned to one and only one class. The input space is thereby divided into *decision regions* whose boundaries are called *decision boundaries* or *decision surfaces*. In this chapter, we consider linear

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024
C. M. Bishop, H. Bishop, *Deep Learning*, https://doi.org/10.1007/978-3-031-45468-4_5

Telegram: @uni_k

www.konkur.in

models for classification, by which we mean that the decision surfaces are linear functions of the input vector $\mathbf{x}$ and, hence, are defined by $(D-1)$-dimensional hyperplanes within the D-dimensional input space. Data sets whose classes can be separated exactly by linear decision surfaces are said to be *linearly separable*. Linear classification models can be applied to data sets that are not linearly separable, although not all inputs will be correctly classified.

We can broadly identify three distinct approaches to solving classification problems. The simplest involves constructing a *discriminant function* that directly assigns each vector $\mathbf{x}$ to a specific class. A more powerful approach, however, models the conditional probability distributions $p(\mathcal{C}_k|\mathbf{x})$ in an *inference* stage and subsequently uses these distributions to make optimal *decisions*. Separating inference and decision brings numerous benefits. There are two different approaches to determining the conditional probabilities $p(\mathcal{C}_k|\mathbf{x})$. One technique is to model them directly, for example by representing them as parametric models and then optimizing the parameters using a training set. This will be called a *discriminative probabilistic model*. Alternatively, we can model the class-conditional densities $p(\mathbf{x}|\mathcal{C}_k)$, together with the prior probabilities $p(\mathcal{C}_k)$ for the classes, and then compute the required posterior probabilities using Bayes' theorem:

Section 5.2.4

$$p(\mathcal{C}_k|\mathbf{x}) = \frac{p(\mathbf{x}|\mathcal{C}_k)p(\mathcal{C}_k)}{p(\mathbf{x})}. \tag{5.1}$$

This will be called a *generative probabilistic model* because it offers the opportunity to generate samples from each of the class-conditional densities $p(\mathbf{x}|\mathcal{C}_k)$. In this chapter, we will discuss examples of all three approaches: discriminant functions, generative probabilistic models, and discriminative probabilistic models.

5.1. Discriminant Functions

A discriminant is a function that takes an input vector $\mathbf{x}$ and assigns it to one of K classes, denoted $\mathcal{C}_k$. In this chapter, we will restrict attention to *linear discriminants*, namely those for which the decision surfaces are hyperplanes. To simplify the discussion, we consider first two classes and then investigate the extension to $K > 2$ classes.

5.1.1 Two classes

The simplest representation of a linear discriminant function is obtained by taking a linear function of the input vector so that

$$y(\mathbf{x}) = \mathbf{w}^{\mathrm{T}}\mathbf{x} + w_0 \tag{5.2}$$

where $\mathbf{w}$ is called a *weight vector*, and w_0 is a *bias* (not to be confused with bias in the statistical sense). An input vector $\mathbf{x}$ is assigned to class $\mathcal{C}_1$ if $y(\mathbf{x}) \geqslant 0$ and to class $\mathcal{C}_2$ otherwise. The corresponding decision boundary is therefore defined by the relation $y(\mathbf{x}) = 0$, which corresponds to a $(D-1)$-dimensional hyperplane within

Telegram: @uni_k

www.konkur.in

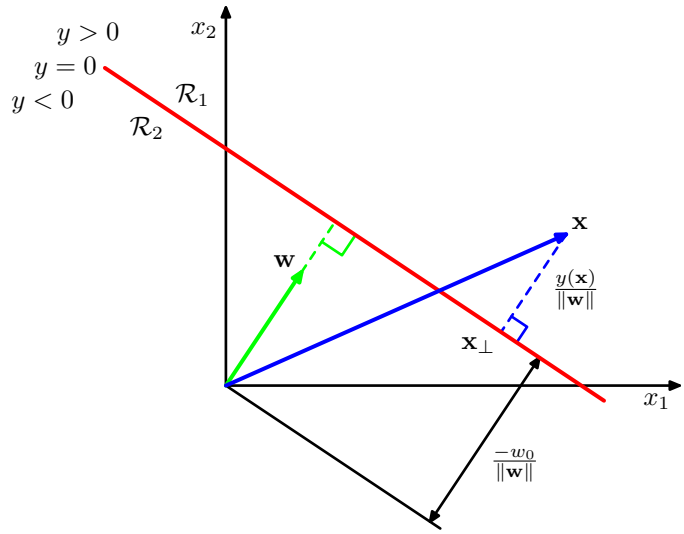


Figure 5.1 Illustration of the geometry of a linear discriminant function in two dimensions. The decision surface, shown in red, is perpendicular to $\mathbf{w}$, and its displacement from the origin is controlled by the bias parameter w_0. Also, the signed orthogonal distance of a general point $\mathbf{x}$ from the decision surface is given by $y(\mathbf{x})/\|\mathbf{w}\|$.

the D-dimensional input space. Consider two points $\mathbf{x}_\mathrm{A}$ and $\mathbf{x}_\mathrm{B}$ both of which lie on the decision surface. Because $y(\mathbf{x}_\mathrm{A}) = y(\mathbf{x}_\mathrm{B}) = 0$, we have $\mathbf{w}^\mathrm{T}(\mathbf{x}_\mathrm{A} - \mathbf{x}_\mathrm{B}) = 0$ and hence the vector $\mathbf{w}$ is orthogonal to every vector lying within the decision surface, and so $\mathbf{w}$ determines the orientation of the decision surface. Similarly, if $\mathbf{x}$ is a point on the decision surface, then $y(\mathbf{x}) = 0$, and so the normal distance from the origin to the decision surface is given by

$$\frac{\mathbf{w}^\mathrm{T}\mathbf{x}}{\|\mathbf{w}\|} = -\frac{w_0}{\|\mathbf{w}\|}. \tag{5.3}$$

We therefore see that the bias parameter w_0 determines the location of the decision surface. These properties are illustrated for the case of $D = 2$ in Figure 5.1.

Furthermore, note that the value of $y(\mathbf{x})$ gives a signed measure of the perpendicular distance r of the point $\mathbf{x}$ from the decision surface. To see this, consider an arbitrary point $\mathbf{x}$ and let $\mathbf{x}_\perp$ be its orthogonal projection onto the decision surface, so that

$$\mathbf{x} = \mathbf{x}_\perp + r\frac{\mathbf{w}}{\|\mathbf{w}\|}. \tag{5.4}$$

Multiplying both sides of this result by $\mathbf{w}^\mathrm{T}$ and adding w_0, and making use of $y(\mathbf{x}) = \mathbf{w}^\mathrm{T}\mathbf{x} + w_0$ and $y(\mathbf{x}_\perp) = \mathbf{w}^\mathrm{T}\mathbf{x}_\perp + w_0 = 0$, we have

$$r = \frac{y(\mathbf{x})}{\|\mathbf{w}\|}. \tag{5.5}$$

This result is illustrated in Figure 5.1.

Section 4.1.1

As with linear regression models, it is sometimes convenient to use a more compact notation in which we introduce an additional dummy 'input' value $x_0 = 1$ and then define $\widetilde{\mathbf{w}} = (w_0, \mathbf{w})$ and $\widetilde{\mathbf{x}} = (x_0, \mathbf{x})$ so that

$$y(\mathbf{x}) = \widetilde{\mathbf{w}}^\mathrm{T}\widetilde{\mathbf{x}}. \tag{5.6}$$

Telegram: @uni_k

www.konkur.in

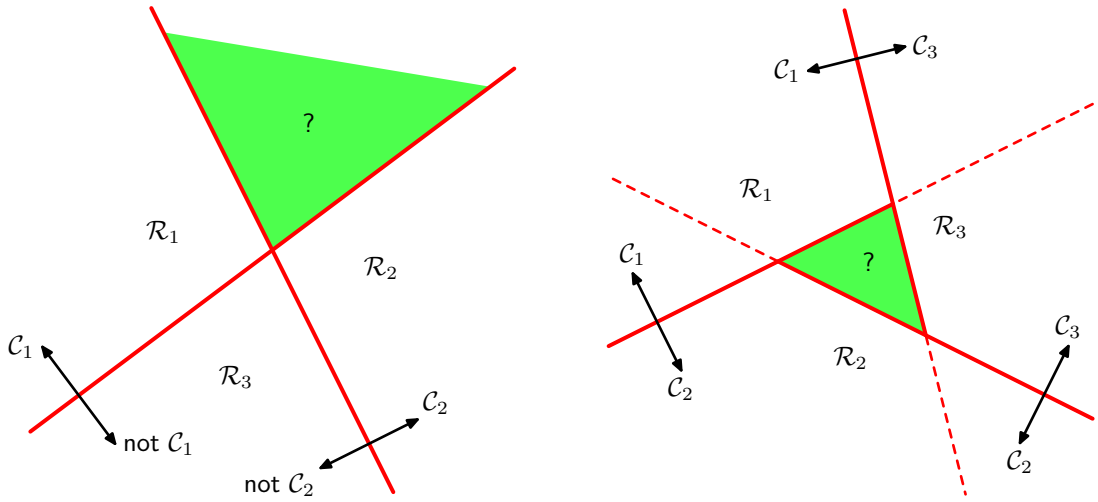


Figure 5.2 Attempting to construct a K-class discriminant from a set of two-class discriminant functions leads to ambiguous regions, as shown in green. On the left is an example with two discriminant functions designed to distinguish points in class $\mathcal{C}_k$ from points not in class $\mathcal{C}_k$. On the right is an example involving three discriminant functions each of which is used to separate a pair of classes $\mathcal{C}_k$ and $\mathcal{C}_j$.

In this case, the decision surfaces are D-dimensional hyperplanes passing through the origin of the $(D+1)$-dimensional expanded input space.

5.1.2 Multiple classes

Now consider the extension of linear discriminant functions to $K > 2$ classes. We might be tempted to build a K-class discriminant by combining a number of two-class discriminant functions. However, this leads to some serious difficulties (Duda and Hart, 1973), as we now show.

Consider a model with $K-1$ classifiers, each of which solves a two-class problem of separating points in a particular class $\mathcal{C}_k$ from points not in that class. This is known as a *one-versus-the-rest* classifier. The left-hand example in Figure 5.2 shows an example involving three classes where this approach leads to regions of input space that are ambiguously classified.

An alternative is to introduce $K(K-1)/2$ binary discriminant functions, one for every possible pair of classes. This is known as a *one-versus-one* classifier. Each point is then classified according to a majority vote amongst the discriminant functions. However, this too runs into the problem of ambiguous regions, as illustrated in the right-hand diagram of Figure 5.2.

We can avoid these difficulties by considering a single K-class discriminant comprising K linear functions of the form

$$y_k(\mathbf{x}) = \mathbf{w}_k^{\mathrm{T}}\mathbf{x} + w_{k0} \tag{5.7}$$

and then assigning a point $\mathbf{x}$ to class $\mathcal{C}_k$ if $y_k(\mathbf{x}) > y_j(\mathbf{x})$ for all $j \neq k$. The decision boundary between class $\mathcal{C}_k$ and class $\mathcal{C}_j$ is therefore given by $y_k(\mathbf{x}) = y_j(\mathbf{x})$ and

Telegram: @uni_k

www.konkur.in

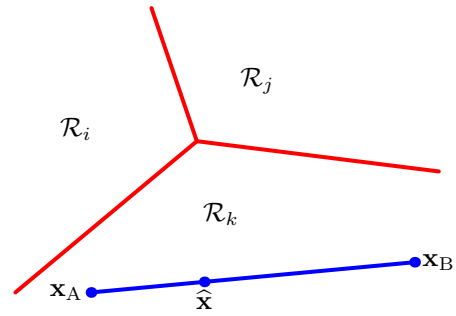


Figure 5.3 Illustration of the decision regions for a multi-class linear discriminant, with the decision boundaries shown in red. If two points $\mathbf{x}_\mathrm{A}$ and $\mathbf{x}_\mathrm{B}$ both lie inside the same decision region $\mathcal{R}_k$, then any point $\widehat{\mathbf{x}}$ that lies on the line connecting these two points must also lie in $\mathcal{R}_k$, and hence, the decision region must be singly connected and convex.

hence corresponds to a $(D-1)$-dimensional hyperplane defined by

$$(\mathbf{w}_k - \mathbf{w}_j)^\mathrm{T}\mathbf{x} + (w_{k0} - w_{j0}) = 0. \tag{5.8}$$

This has the same form as the decision boundary for the two-class case discussed in Section 5.1.1, and so analogous geometrical properties apply.

The decision regions of such a discriminant are always singly connected and convex. To see this, consider two points $\mathbf{x}_\mathrm{A}$ and $\mathbf{x}_\mathrm{B}$ both of which lie inside decision region $\mathcal{R}_k$, as illustrated in Figure 5.3. Any point $\widehat{\mathbf{x}}$ that lies on the line connecting $\mathbf{x}_\mathrm{A}$ and $\mathbf{x}_\mathrm{B}$ can be expressed in the form

$$\widehat{\mathbf{x}} = \lambda\mathbf{x}_\mathrm{A} + (1-\lambda)\mathbf{x}_\mathrm{B} \tag{5.9}$$

where $0 \leqslant \lambda \leqslant 1$. From the linearity of the discriminant functions, it follows that

$$y_k(\widehat{\mathbf{x}}) = \lambda y_k(\mathbf{x}_\mathrm{A}) + (1-\lambda)y_k(\mathbf{x}_\mathrm{B}). \tag{5.10}$$

Because both $\mathbf{x}_\mathrm{A}$ and $\mathbf{x}_\mathrm{B}$ lie inside $\mathcal{R}_k$, it follows that $y_k(\mathbf{x}_\mathrm{A}) > y_j(\mathbf{x}_\mathrm{A})$ and that $y_k(\mathbf{x}_\mathrm{B}) > y_j(\mathbf{x}_\mathrm{B})$, for all $j \neq k$, and hence $y_k(\widehat{\mathbf{x}}) > y_j(\widehat{\mathbf{x}})$, and so $\widehat{\mathbf{x}}$ also lies inside $\mathcal{R}_k$. Thus, $\mathcal{R}_k$ is singly connected and convex.

Note that for two classes, we can either employ the formalism discussed here, based on two discriminant functions $y_1(\mathbf{x})$ and $y_2(\mathbf{x})$, or else use the simpler but essentially equivalent formulation based on a single discriminant function $y(\mathbf{x})$.

Section 5.1.1

5.1.3 1-of-K coding

For regression problems, the target variable $\mathbf{t}$ was simply the vector of real numbers whose values we wish to predict. In classification, there are various ways of using target values to represent class labels. For two-class problems, the most convenient is the binary representation in which there is a single target variable $t \in \{0, 1\}$ such that $t = 1$ represents class $\mathcal{C}_1$ and $t = 0$ represents class $\mathcal{C}_2$. We can interpret the value of t as the probability that the class is $\mathcal{C}_1$, with the probability values taking only the extreme values of 0 and 1. For $K > 2$ classes, it is convenient to use a 1-of-K coding scheme, also known as the one-hot encoding scheme, in which $\mathbf{t}$ is a vector of length K such that if the class is $\mathcal{C}_j$, then all elements t_k of $\mathbf{t}$ are zero

Telegram: @uni_k

www.konkur.in

except element t_j, which takes the value 1. For instance, if we have $K = 5$ classes, then a data point from class 2 would be given the target vector

$$\mathbf{t} = (0, 1, 0, 0, 0)^{\mathrm{T}}. \tag{5.11}$$

Again, we can interpret the value of t_k as the probability that the class is $\mathcal{C}_k$ in which the probabilities take only the values 0 and 1.

5.1.4 Least squares for classification

Section 4.1.3

With linear regression models, the minimization of a sum-of-squares error function leads to a simple closed-form solution for the parameter values. It is therefore tempting to see if we can apply the same least-squares formalism to classification problems. Consider a general classification problem with K classes and a 1-of-K binary coding scheme for the target vector $\mathbf{t}$. One justification for using least squares in such a context is that it approximates the conditional expectation $\mathbb{E}[\mathbf{t}|\mathbf{x}]$ of the target values given the input vector. For a binary coding scheme, this conditional expectation is given by the vector of posterior class probabilities. Unfortunately, these probabilities are typically approximated rather poorly, and indeed the approximations can have values outside the range $(0, 1)$. However, it is instructional to explore these simple models and to understand how these limitations arise.

Exercise 5.1

Each class $\mathcal{C}_k$ is described by its own linear model so that

$$y_k(\mathbf{x}) = \mathbf{w}_k^{\mathrm{T}}\mathbf{x} + w_{k0} \tag{5.12}$$

where $k = 1, \ldots, K$. We can conveniently group these together using vector notation so that

$$\mathbf{y}(\mathbf{x}) = \widetilde{\mathbf{W}}^{\mathrm{T}}\widetilde{\mathbf{x}} \tag{5.13}$$

where $\widetilde{\mathbf{W}}$ is a matrix whose kth column comprises the $(D + 1)$-dimensional vector $\widetilde{\mathbf{w}}_k = (w_{k0}, \mathbf{w}_k^{\mathrm{T}})^{\mathrm{T}}$ and $\widetilde{\mathbf{x}}$ is the corresponding augmented input vector $(1, \mathbf{x}^{\mathrm{T}})^{\mathrm{T}}$ with a dummy input $x_0 = 1$. A new input $\mathbf{x}$ is then assigned to the class for which the output $y_k = \widetilde{\mathbf{w}}_k^{\mathrm{T}}\widetilde{\mathbf{x}}$ is largest.

We now determine the parameter matrix $\widetilde{\mathbf{W}}$ by minimizing a sum-of-squares error function. Consider a training data set $\{\mathbf{x}_n, \mathbf{t}_n\}$ where $n = 1, \ldots, N$, and define a matrix $\mathbf{T}$ whose nth row is the vector $\mathbf{t}_n^{\mathrm{T}}$, together with a matrix $\widetilde{\mathbf{X}}$ whose nth row is $\widetilde{\mathbf{x}}_n^{\mathrm{T}}$. The sum-of-squares error function can then be written as

$$E_D(\widetilde{\mathbf{W}}) = \frac{1}{2}\mathrm{Tr}\left\{(\widetilde{\mathbf{X}}\widetilde{\mathbf{W}} - \mathbf{T})^{\mathrm{T}}(\widetilde{\mathbf{X}}\widetilde{\mathbf{W}} - \mathbf{T})\right\}. \tag{5.14}$$

Setting the derivative with respect to $\widetilde{\mathbf{W}}$ to zero and rearranging, we obtain the solution for $\widetilde{\mathbf{W}}$ in the form

$$\widetilde{\mathbf{W}} = (\widetilde{\mathbf{X}}^{\mathrm{T}}\widetilde{\mathbf{X}})^{-1}\widetilde{\mathbf{X}}^{\mathrm{T}}\mathbf{T} = \widetilde{\mathbf{X}}^{\dagger}\mathbf{T} \tag{5.15}$$

Section 4.1.3

where $\widetilde{\mathbf{X}}^{\dagger}$ is the pseudo-inverse of the matrix $\widetilde{\mathbf{X}}$. We then obtain the discriminant

Telegram: @uni_k

www.konkur.in

function in the form

$$\mathbf{y}(\mathbf{x}) = \widetilde{\mathbf{W}}^{\mathrm{T}}\widetilde{\mathbf{x}} = \mathbf{T}^{\mathrm{T}}\left(\widetilde{\mathbf{X}}^{\dagger}\right)^{\mathrm{T}}\widetilde{\mathbf{x}}. \tag{5.16}$$

An interesting property of least-squares solutions with multiple target variables is that if every target vector in the training set satisfies some linear constraint

$$\mathbf{a}^{\mathrm{T}}\mathbf{t}_n + b = 0 \tag{5.17}$$

for some constants $\mathbf{a}$ and b, then the model prediction for any value of $\mathbf{x}$ will satisfy the same constraint so that

Exercise 5.3

$$\mathbf{a}^{\mathrm{T}}\mathbf{y}(\mathbf{x}) + b = 0. \tag{5.18}$$

Thus, if we use a 1-of-K coding scheme for K classes, then the predictions made by the model will have the property that the elements of $\mathbf{y}(\mathbf{x})$ will sum to 1 for any value of $\mathbf{x}$. However, this summation constraint alone is not sufficient to allow the model outputs to be interpreted as probabilities because they are not constrained to lie within the interval $(0, 1)$.

The least-squares approach gives an exact closed-form solution for the discriminant function parameters. However, even as a discriminant function (where we use it to make decisions directly and dispense with any probabilistic interpretation), it suffers from some severe problems. We have seen that the sum-of-squares error function can be viewed as the negative log likelihood under the assumption of a Gaussian noise distribution. If the true distribution of the data is markedly different from being Gaussian, then least squares can give poor results. In particular, least squares is very sensitive to the presence of *outliers*, which are data points located a long way from the bulk of the data. This is illustrated in Figure 5.4. Here we see that the additional data points in the right-hand figure produce a significant change in the location of the decision boundary, even though these points would be correctly classified by the original decision boundary in the left-hand figure. The sum-of-squares error function gives too much weight to data points that are a long way from the decision boundary, even though they are correctly classified. Outliers can arise due to rare events or may simply be due to mistakes in the data set. Techniques that are sensitive to a very few data points are said to lack *robustness*. For comparison, Figure 5.4 also shows results from a technique called *logistic regression*, which is more robust to outliers.

Section 2.3.4

Section 5.4.3

The failure of least squares should not surprise us when we recall that it corresponds to maximum likelihood under the assumption of a Gaussian conditional distribution, whereas binary target vectors clearly have a distribution that is far from Gaussian. By adopting more appropriate probabilistic models, we can obtain classification techniques with much better properties than least squares, and which can also be generalized to give flexible nonlinear neural network models, as we will see in later chapters.

Telegram: @uni_k

www.konkur.in

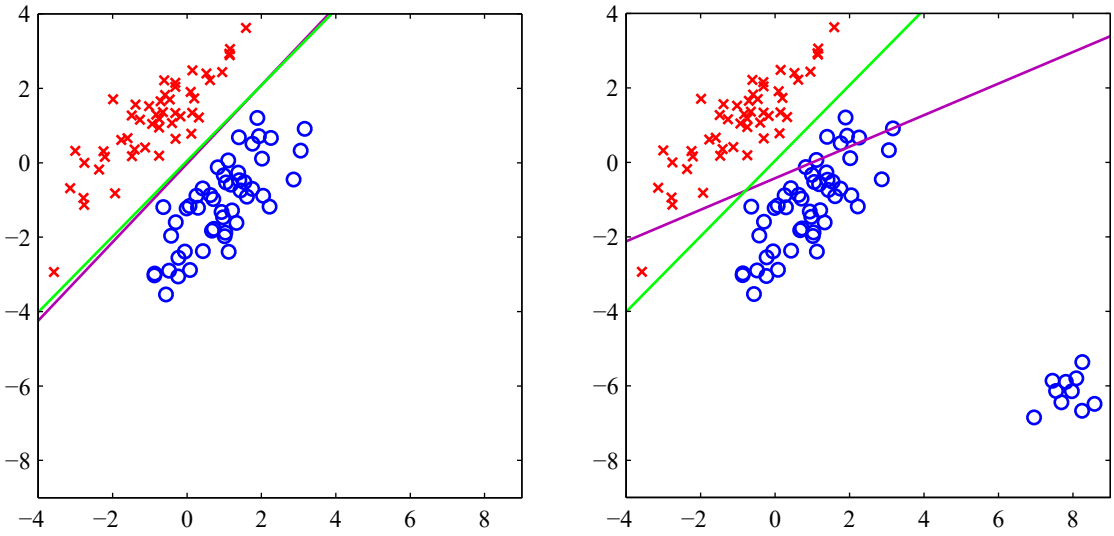


Figure 5.4 The left-hand plot shows data from two classes, denoted by red crosses and blue circles, together with the decision boundaries found by least squares (magenta curve) and by a logistic regression model (green curve). The right-hand plot shows the corresponding results obtained when extra data points are added at the bottom right of the diagram, showing that least squares is highly sensitive to outliers, unlike logistic regression.

5.2. Decision Theory

Section 4.2

When we discussed linear regression we saw how the process of making predictions in machine learning can be broken down into the two stages of inference and decision. We now explore this perspective in much greater depth specifically in the context of classifiers.

Suppose we have an input vector $\mathbf{x}$ together with a corresponding vector $\mathbf{t}$ of target variables, and our goal is to predict $\mathbf{t}$ given a new value for $\mathbf{x}$. For regression problems, $\mathbf{t}$ will comprise continuous variables and in general will be a vector as we may wish to predict several related quantities. For classification problems, $\mathbf{t}$ will represent class labels. Again, $\mathbf{t}$ will in general be a vector if we have more than two classes. The joint probability distribution $p(\mathbf{x}, \mathbf{t})$ provides a complete summary of the uncertainty associated with these variables. Determining $p(\mathbf{x}, \mathbf{t})$ from a set of training data is an example of *inference* and is typically a very difficult problem whose solution forms the subject of much of this book. In a practical application, however, we must often make a specific prediction for the value of $\mathbf{t}$ or more generally take a specific action based on our understanding of the values $\mathbf{t}$ is likely to take, and this aspect is the subject of decision theory.

Consider, for example, our earlier medical diagnosis problem in which we have taken an image of a skin lesion on a patient, and we wish to determine whether the patient has cancer. In this case, the input vector $\mathbf{x}$ is the set of pixel intensities in

Telegram: @uni_k

www.konkur.in

the image, and the output variable t will represent the absence of cancer, which we denote by the class $\mathcal{C}_1$, or the presence of cancer, which we denote by the class $\mathcal{C}_2$. We might, for instance, choose t to be a binary variable such that $t = 0$ corresponds to class $\mathcal{C}_1$ and $t = 1$ corresponds to class $\mathcal{C}_2$. We will see later that this choice of label values is particularly convenient when working with probabilities. The general inference problem then involves determining the joint distribution $p(\mathbf{x}, \mathcal{C}_k)$, or equivalently $p(\mathbf{x}, t)$, which gives us the most complete probabilistic description of the variables. Although this can be a very useful and informative quantity, ultimately, we must decide either to give treatment to the patient or not, and we would like this choice to be optimal according to some appropriate criterion (Duda and Hart, 1973). This is the *decision* step, and the aim of decision theory is that it should tell us how to make optimal decisions given the appropriate probabilities. We will see that the decision stage is generally very simple, even trivial, once we have solved the inference problem. Here we give an introduction to the key ideas of decision theory as required for the rest of the book. Further background, as well as more detailed accounts, can be found in Berger (1985) and Bather (2000).

Before giving a more detailed analysis, let us first consider informally how we might expect probabilities to play a role in making decisions. When we obtain the skin image $\mathbf{x}$ for a new patient, our goal is to decide which of the two classes to assign the image to. We are therefore interested in the probabilities of the two classes, given the image, which are given by $p(\mathcal{C}_k|\mathbf{x})$. Using Bayes' theorem, these probabilities can be expressed in the form

$$p(\mathcal{C}_k|\mathbf{x}) = \frac{p(\mathbf{x}|\mathcal{C}_k)p(\mathcal{C}_k)}{p(\mathbf{x})}. \tag{5.19}$$

Note that any of the quantities appearing in Bayes' theorem can be obtained from the joint distribution $p(\mathbf{x}, \mathcal{C}_k)$ by either marginalizing or conditioning with respect to the appropriate variables. We can now interpret $p(\mathcal{C}_k)$ as the prior probability for the class $\mathcal{C}_k$ and $p(\mathcal{C}_k|\mathbf{x})$ as the corresponding posterior probability. Thus, $p(\mathcal{C}_1)$ represents the probability that a person has cancer, before the image is taken. Similarly, $p(\mathcal{C}_1|\mathbf{x})$ is the posterior probability, revised using Bayes' theorem in light of the information contained in the image. If our aim is to minimize the chance of assigning $\mathbf{x}$ to the wrong class, then intuitively we would choose the class having the higher posterior probability. We now show that this intuition is correct, and we also discuss more general criteria for making decisions.

5.2.1 Misclassification rate

Suppose that our goal is simply to make as few misclassifications as possible. We need a rule that assigns each value of $\mathbf{x}$ to one of the available classes. Such a rule will divide the input space into regions $\mathcal{R}_k$ called *decision regions*, one for each class, such that all points in $\mathcal{R}_k$ are assigned to class $\mathcal{C}_k$. The boundaries between decision regions are called *decision boundaries* or *decision surfaces*. Note that each decision region need not be contiguous but could comprise some number of disjoint regions. To find the optimal decision rule, consider first the case of two classes, as in the cancer problem, for instance. A mistake occurs when an input vector belonging

Telegram: @uni_k

www.konkur.in

to class $\mathcal{C}_1$ is assigned to class $\mathcal{C}_2$ or vice versa. The probability of this occurring is given by

$$\begin{aligned} p(\text{mistake}) &= p(\mathbf{x} \in \mathcal{R}_1, \mathcal{C}_2) + p(\mathbf{x} \in \mathcal{R}_2, \mathcal{C}_1) \\ &= \int_{\mathcal{R}_1} p(\mathbf{x}, \mathcal{C}_2)\, \mathrm{d}\mathbf{x} + \int_{\mathcal{R}_2} p(\mathbf{x}, \mathcal{C}_1)\, \mathrm{d}\mathbf{x}. \end{aligned} \tag{5.20}$$

We are free to choose the decision rule that assigns each point $\mathbf{x}$ to one of the two classes. Clearly, to minimize $p(\text{mistake})$ we should arrange that each $\mathbf{x}$ is assigned to whichever class has the smaller value of the integrand in (5.20). Thus, if $p(\mathbf{x}, \mathcal{C}_1) > p(\mathbf{x}, \mathcal{C}_2)$ for a given value of $\mathbf{x}$, then we should assign that $\mathbf{x}$ to class $\mathcal{C}_1$. From the product rule of probability, we have $p(\mathbf{x}, \mathcal{C}_k) = p(\mathcal{C}_k|\mathbf{x})p(\mathbf{x})$. Because the factor $p(\mathbf{x})$ is common to both terms, we can restate this result as saying that the minimum probability of making a mistake is obtained if each value of $\mathbf{x}$ is assigned to the class for which the posterior probability $p(\mathcal{C}_k|\mathbf{x})$ is largest. This result is illustrated for two classes and a single input variable x in Figure 5.5.

For the more general case of K classes, it is slightly easier to maximize the probability of being correct, which is given by

$$\begin{aligned} p(\text{correct}) &= \sum_{k=1}^{K} p(\mathbf{x} \in \mathcal{R}_k, \mathcal{C}_k) \\ &= \sum_{k=1}^{K} \int_{\mathcal{R}_k} p(\mathbf{x}, \mathcal{C}_k)\, \mathrm{d}\mathbf{x}, \end{aligned} \tag{5.21}$$

which is maximized when the regions $\mathcal{R}_k$ are chosen such that each $\mathbf{x}$ is assigned to the class for which $p(\mathbf{x}, \mathcal{C}_k)$ is largest. Again, using the product rule $p(\mathbf{x}, \mathcal{C}_k) = p(\mathcal{C}_k|\mathbf{x})p(\mathbf{x})$, and noting that the factor of $p(\mathbf{x})$ is common to all terms, we see that each $\mathbf{x}$ should be assigned to the class having the largest posterior probability $p(\mathcal{C}_k|\mathbf{x})$.

5.2.2 Expected loss

For many applications, our objective will be more complex than simply minimizing the number of misclassifications. Let us consider again the medical diagnosis problem. We note that, if a patient who does not have cancer is incorrectly diagnosed as having cancer, the consequences may be that they experience some distress plus there is the need for further investigations. Conversely, if a patient with cancer is diagnosed as healthy, the result may be premature death due to lack of treatment. Thus, the consequences of these two types of mistake can be dramatically different. It would clearly be better to make fewer mistakes of the second kind, even if this was at the expense of making more mistakes of the first kind.

We can formalize such issues through the introduction of a *loss function*, also called a *cost function*, which is a single, overall measure of loss incurred in taking any of the available decisions or actions. Our goal is then to minimize the total loss incurred. Note that some authors consider instead a *utility function*, whose value

Telegram: @uni_k

www.konkur.in

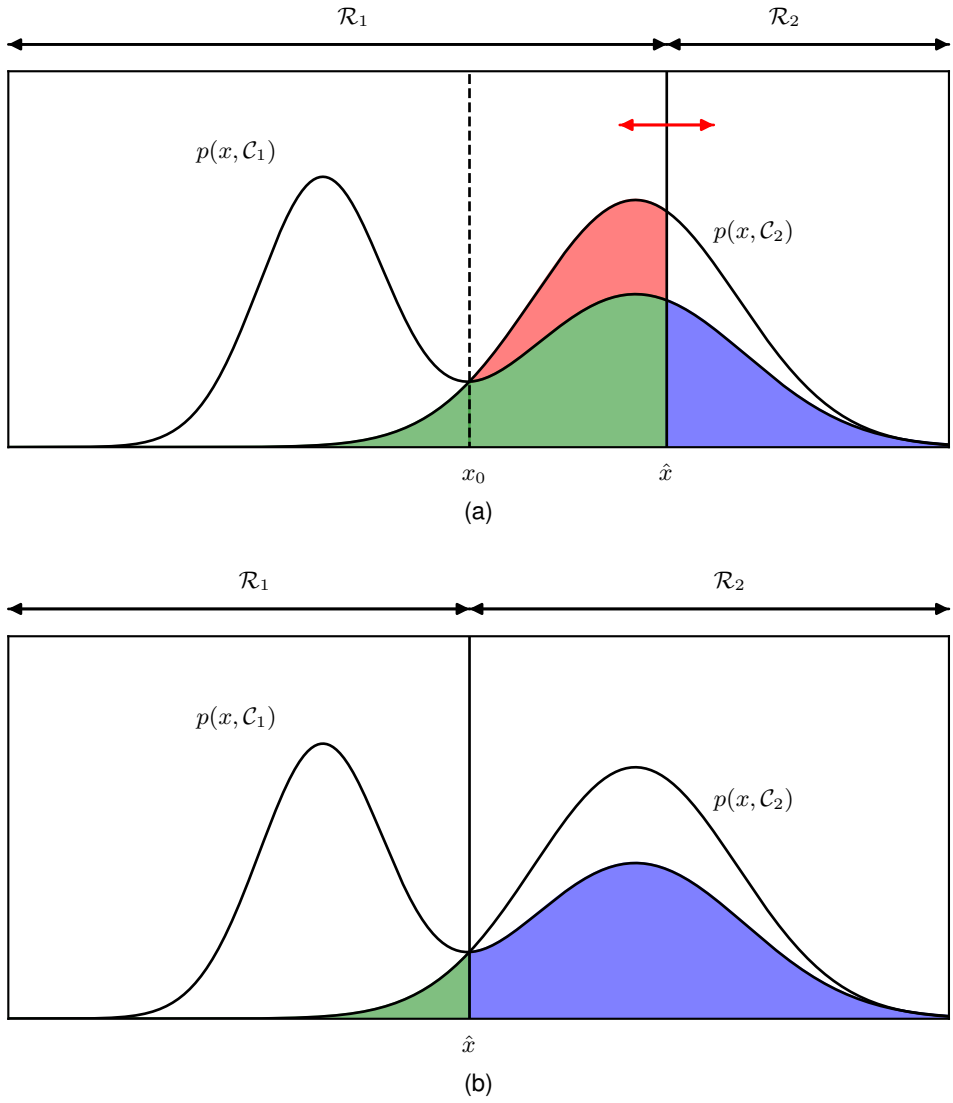


Figure 5.5 Schematic illustration of the joint probabilities $p(x, \mathcal{C}_k)$ for each of two classes plotted against x, together with the decision boundary $x = \widehat{x}$. Values of $x \geqslant \widehat{x}$ are classified as class $\mathcal{C}_2$ and hence belong to decision region $\mathcal{R}_2$, whereas points $x < \widehat{x}$ are classified as $\mathcal{C}_1$ and belong to $\mathcal{R}_1$. Errors arise from the blue, green, and red regions, so that for $x < \widehat{x}$, the errors are due to points from class $\mathcal{C}_2$ being misclassified as $\mathcal{C}_1$ (represented by the sum of the red and green regions). Conversely for points in the region $x \geqslant \widehat{x}$, the errors are due to points from class $\mathcal{C}_1$ being misclassified as $\mathcal{C}_2$ (represented by the blue region). By varying the location $\widehat{x}$ of the decision boundary, as indicated by the red double-headed arrow in (a), the combined areas of the blue and green regions remains constant, whereas the size of the red region varies. The optimal choice for $\widehat{x}$ is where the curves for $p(x, \mathcal{C}_1)$ and $p(x, \mathcal{C}_2)$ cross, as shown in (b) and corresponding to $\widehat{x} = x_0$, because in this case the red region disappears. This is equivalent to the minimum misclassification rate decision rule, which assigns each value of x to the class having the higher posterior probability $p(\mathcal{C}_k|x)$.

Telegram: @uni_k

www.konkur.in

Figure 5.6 An example of a loss matrix with elements L_{kj} for the cancer treatment problem. The rows correspond to the true class, whereas the columns correspond to the assignment of class made by our decision criterion.

	normal	cancer
normal	0	1
cancer	100	0

they aim to maximize. These are equivalent concepts if we take the utility to be simply the negative of the loss. Throughout this text we will use the loss function convention. Suppose that, for a new value of $\mathbf{x}$, the true class is $\mathcal{C}_k$ and that we assign $\mathbf{x}$ to class $\mathcal{C}_j$ (where j may or may not be equal to k). In so doing, we incur some level of loss that we denote by L_{kj}, which we can view as the k, j element of a *loss matrix*. For instance, in our cancer example, we might have a loss matrix of the form shown in Figure 5.6. This particular loss matrix says that there is no loss incurred if the correct decision is made, there is a loss of 1 if a healthy patient is diagnosed as having cancer, whereas there is a loss of 100 if a patient having cancer is diagnosed as healthy.

The optimal solution is the one that minimizes the loss function. However, the loss function depends on the true class, which is unknown. For a given input vector $\mathbf{x}$, our uncertainty in the true class is expressed through the joint probability distribution $p(\mathbf{x}, \mathcal{C}_k)$, and so we seek instead to minimize the average loss, where the average is computed with respect to this distribution and is given by

$$\mathbb{E}[L] = \sum_k \sum_j \int_{\mathcal{R}_j} L_{kj} p(\mathbf{x}, \mathcal{C}_k) \, \mathrm{d}\mathbf{x}. \tag{5.22}$$

Each $\mathbf{x}$ can be assigned independently to one of the decision regions $\mathcal{R}_j$. Our goal is to choose the regions $\mathcal{R}_j$ to minimize the expected loss (5.22), which implies that for each $\mathbf{x}$, we should minimize $\sum_k L_{kj} p(\mathbf{x}, \mathcal{C}_k)$. As before, we can use the product rule $p(\mathbf{x}, \mathcal{C}_k) = p(\mathcal{C}_k|\mathbf{x})p(\mathbf{x})$ to eliminate the common factor of $p(\mathbf{x})$. Thus, the decision rule that minimizes the expected loss assigns each new $\mathbf{x}$ to the class j for which the quantity

$$\sum_k L_{kj} p(\mathcal{C}_k|\mathbf{x}) \tag{5.23}$$

is a minimum. Once we have chosen values for the loss matrix elements L_{kj}, this is clearly trivial to do.

5.2.3 The reject option

We have seen that classification errors arise from the regions of input space where the largest of the posterior probabilities $p(\mathcal{C}_k|\mathbf{x})$ is significantly less than unity or equivalently where the joint distributions $p(\mathbf{x}, \mathcal{C}_k)$ have comparable values. These are the regions where we are relatively uncertain about class membership. In some applications, it will be appropriate to avoid making decisions on the difficult cases in anticipation of obtaining a lower error rate on those examples for which a classification decision is made. This is known as the *reject option*. For example, in our hypothetical cancer screening example, it may be appropriate to use an automatic

Telegram: @uni_k

www.konkur.in

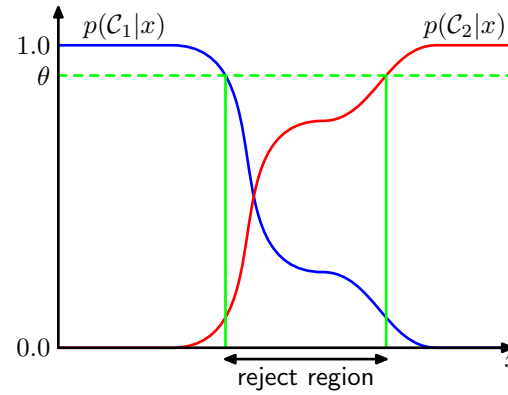


Figure 5.7 Illustration of the reject option. Inputs x such that the larger of the two posterior probabilities is less than or equal to some threshold θ will be rejected.

system to classify those images for which there is little doubt as to the correct class, while requesting a biopsy to classify the more ambiguous cases. We can achieve this by introducing a threshold θ and rejecting those inputs $\mathbf{x}$ for which the largest of the posterior probabilities $p(\mathcal{C}_k|\mathbf{x})$ is less than or equal to θ. This is illustrated for two classes and a single continuous input variable x in Figure 5.7. Note that setting $\theta = 1$ will ensure that all examples are rejected, whereas if there are K classes, then setting $\theta < 1/K$ will ensure that no examples are rejected. Thus, the fraction of examples that are rejected is controlled by the value of θ.

We can easily extend the reject criterion to minimize the expected loss, when a loss matrix is given, by taking account of the loss incurred when a reject decision is made.

Exercise 5.10

5.2.4 Inference and decision

We have broken the classification problem down into two separate stages, the *inference stage* in which we use training data to learn a model for $p(\mathcal{C}_k|\mathbf{x})$ and the subsequent *decision* stage in which we use these posterior probabilities to make optimal class assignments. An alternative possibility would be to solve both problems together and simply learn a function that maps inputs $\mathbf{x}$ directly into decisions. Such a function is called a *discriminant function*.

In fact, we can identify three distinct approaches to solving decision problems, all of which have been used in practical applications. These are, in decreasing order of complexity, as follows:

(a) First, solve the inference problem of determining the class-conditional densities $p(\mathbf{x}|\mathcal{C}_k)$ for each class $\mathcal{C}_k$ individually. Separately infer the prior class probabilities $p(\mathcal{C}_k)$. Then use Bayes' theorem in the form

$$p(\mathcal{C}_k|\mathbf{x}) = \frac{p(\mathbf{x}|\mathcal{C}_k)p(\mathcal{C}_k)}{p(\mathbf{x})} \tag{5.24}$$

to find the posterior class probabilities $p(\mathcal{C}_k|\mathbf{x})$. As usual, the denominator in

Telegram: @uni_k

www.konkur.in

Bayes' theorem can be found in terms of the quantities in the numerator, using

$$p(\mathbf{x}) = \sum_k p(\mathbf{x}|\mathcal{C}_k)p(\mathcal{C}_k). \tag{5.25}$$

Equivalently, we can model the joint distribution $p(\mathbf{x}, \mathcal{C}_k)$ directly and then normalize to obtain the posterior probabilities. Having found the posterior probabilities, we use decision theory to determine the class membership for each new input $\mathbf{x}$. Approaches that explicitly or implicitly model the distribution of inputs as well as outputs are known as *generative models*, because by sampling from them, it is possible to generate synthetic data points in the input space.

(b) First, solve the inference problem of determining the posterior class probabilities $p(\mathcal{C}_k|\mathbf{x})$, and then subsequently use decision theory to assign each new $\mathbf{x}$ to one of the classes. Approaches that model the posterior probabilities directly are called *discriminative models*.

(c) Find a function $f(\mathbf{x})$, called a discriminant function, that maps each input $\mathbf{x}$ directly onto a class label. For instance, for two-class problems, $f(\cdot)$ might be binary valued and such that $f = 0$ represents class $\mathcal{C}_1$ and $f = 1$ represents class $\mathcal{C}_2$. In this case, probabilities play no role.

Let us consider the relative merits of these three alternatives. Approach (a) is the most demanding because it involves finding the joint distribution over both $\mathbf{x}$ and $\mathcal{C}_k$. For many applications, $\mathbf{x}$ will have high dimensionality, and consequently, we may need a large training set to be able to determine the class-conditional densities to reasonable accuracy. Note that the class priors $p(\mathcal{C}_k)$ can often be estimated simply from the fractions of the training set data points in each of the classes. One advantage of approach (a), however, is that it also allows the marginal density of data $p(\mathbf{x})$ to be determined from (5.25). This can be useful for detecting new data points that have low probability under the model and for which the predictions may be of low accuracy, which is known as *outlier detection* or *novelty detection* (Bishop, 1994; Tarassenko, 1995).

However, if we wish only to make classification decisions, then it can be wasteful of computational resources and excessively demanding of data to find the joint distribution $p(\mathbf{x}, \mathcal{C}_k)$ when in fact we really need only the posterior probabilities $p(\mathcal{C}_k|\mathbf{x})$, which can be obtained directly through approach (b). Indeed, the class-conditional densities may contain a significant amount of structure that has little effect on the posterior probabilities, as illustrated in Figure 5.8. There has been much interest in exploring the relative merits of generative and discriminative approaches to machine learning and in finding ways to combine them (Jebara, 2004; Lasserre, Bishop, and Minka, 2006).

An even simpler approach is (c) in which we use the training data to find a discriminant function $f(\mathbf{x})$ that maps each $\mathbf{x}$ directly onto a class label, thereby combining the inference and decision stages into a single learning problem. In the example of Figure 5.8, this would correspond to finding the value of x shown by

Telegram: @uni_k

www.konkur.in

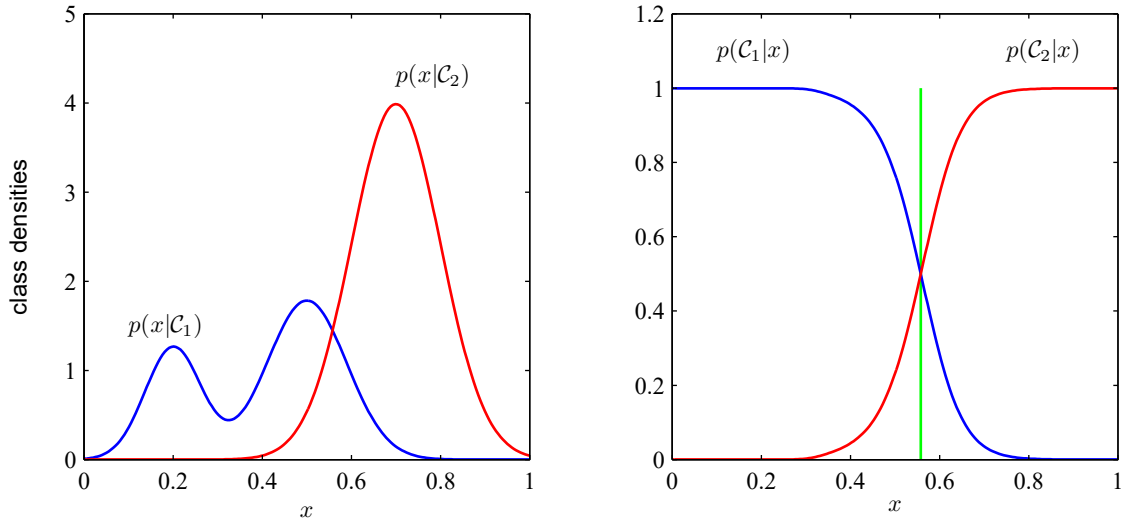


Figure 5.8 Example of the class-conditional densities for two classes having a single input variable x (left plot) together with the corresponding posterior probabilities (right plot). Note that the left-hand mode of the class-conditional density $p(\mathbf{x}|\mathcal{C}_1)$, shown in blue on the left plot, has no effect on the posterior probabilities. The vertical green line in the right plot shows the decision boundary in x that gives the minimum misclassification rate, assuming the prior class probabilities, $p(\mathcal{C}_1)$ and $p(\mathcal{C}_2)$, are equal.

the vertical green line, because this is the decision boundary giving the minimum probability of misclassification.

With option (c), however, we no longer have access to the posterior probabilities $p(\mathcal{C}_k|\mathbf{x})$. There are many powerful reasons for wanting to compute the posterior probabilities, even if we subsequently use them to make decisions. These include:

Minimizing risk. Consider a problem in which the elements of the loss matrix are subjected to revision from time to time (such as might occur in a financial application). If we know the posterior probabilities, we can trivially revise the minimum risk decision criterion by modifying (5.23) appropriately. If we have only a discriminant function, then any change to the loss matrix would require that we return to the training data and solve the inference problem afresh.

Reject option. Posterior probabilities allow us to determine a rejection criterion that will minimize the misclassification rate, or more generally the expected loss, for a given fraction of rejected data points.

Section 2.1.1

Compensating for class priors. Consider our cancer screening example again, and suppose that we have collected a large number of images from the general population for use as training data, which we use to build an automated screening system. Because cancer is rare amongst the general population, we might find that, say, only 1 in every 1,000 examples corresponds to the presence of cancer.

Telegram: @uni_k

www.konkur.in

If we used such a data set to train an adaptive model, we could run into severe difficulties due to the small proportion of those in the cancer class. For instance, a classifier that assigned every point to the normal class would achieve 99.9% accuracy, and it may be difficult to avoid this trivial solution. Also, even a large data set will contain very few examples of skin images corresponding to cancer, and so the learning algorithm will not be exposed to a broad range of examples of such images and hence is not likely to generalize well. A balanced data set with equal numbers of examples from each of the classes would allow us to find a more accurate model. However, we then have to compensate for the effects of our modifications to the training data. Suppose we have used such a modified data set and found models for the posterior probabilities. From Bayes' theorem (5.24), we see that the posterior probabilities are proportional to the prior probabilities, which we can interpret as the fractions of points in each class. We can therefore simply take the posterior probabilities obtained from our artificially balanced data set, divide by the class fractions in that data set, and then multiply by the class fractions in the population to which we wish to apply the model. Finally, we need to normalize to ensure that the new posterior probabilities sum to one. Note that this procedure cannot be applied if we have learned a discriminant function directly instead of determining posterior probabilities.

Combining models. For complex applications, we may wish to break the problem into a number of smaller sub-problems each of which can be tackled by a separate module. For example, in our hypothetical medical diagnosis problem, we may have information available from, say, blood tests as well as skin images. Rather than combine all of this heterogeneous information into one huge input space, it may be more effective to build one system to interpret the images and a different one to interpret the blood data. If each of the two models gives posterior probabilities for the classes, then we can combine the outputs systematically using the rules of probability. One simple way to do this is to assume that, for each class separately, the distributions of inputs for the images, denoted by $\mathbf{x}_{\mathrm{I}}$, and the blood data, denoted by $\mathbf{x}_{\mathrm{B}}$, are independent, so that

$$p(\mathbf{x}_{\mathrm{I}}, \mathbf{x}_{\mathrm{B}}|\mathcal{C}_k) = p(\mathbf{x}_{\mathrm{I}}|\mathcal{C}_k)p(\mathbf{x}_{\mathrm{B}}|\mathcal{C}_k). \tag{5.26}$$

Section 11.2

This is an example of a *conditional independence* property, because the independence holds when the distribution is conditioned on the class $\mathcal{C}_k$. The posterior probability, given both the image and blood data, is then given by

$$\begin{aligned} p(\mathcal{C}_k|\mathbf{x}_{\mathrm{I}}, \mathbf{x}_{\mathrm{B}}) &\propto p(\mathbf{x}_{\mathrm{I}}, \mathbf{x}_{\mathrm{B}}|\mathcal{C}_k)p(\mathcal{C}_k) \\ &\propto p(\mathbf{x}_{\mathrm{I}}|\mathcal{C}_k)p(\mathbf{x}_{\mathrm{B}}|\mathcal{C}_k)p(\mathcal{C}_k) \\ &\propto \frac{p(\mathcal{C}_k|\mathbf{x}_{\mathrm{I}})p(\mathcal{C}_k|\mathbf{x}_{\mathrm{B}})}{p(\mathcal{C}_k)}. \end{aligned} \tag{5.27}$$

Thus, we need the class prior probabilities $p(\mathcal{C}_k)$, which we can easily estimate from the fractions of data points in each class, and then we need to normalize

Telegram: @uni_k

www.konkur.in

$$\begin{array}{c} \\ \text{normal} \\ \text{cancer} \end{array} \begin{array}{c} \begin{array}{cc} \text{normal} & \text{cancer} \end{array} \\ \begin{pmatrix} N_{\text{TN}} & N_{\text{FP}} \\ N_{\text{FN}} & N_{\text{TP}} \end{pmatrix} \end{array}$$

Figure 5.9 The confusion matrix for the cancer treatment problem, in which the rows correspond to the true class and the columns correspond to the assignment of class made by our decision criterion. The elements of the matrix show the numbers of true negatives, false positives, false negatives, and true positives.

the resulting posterior probabilities so they sum to one. The particular conditional independence assumption (5.26) is an example of a *naive Bayes model*. Note that the joint marginal distribution $p(\mathbf{x}_{\text{I}}, \mathbf{x}_{\text{B}})$ will typically not factorize under this model. We will see in later chapters how to construct models for combining data that do not require the conditional independence assumption (5.26). A further advantage of using models that output probabilities rather than decisions is that they can easily be made differentiable with respect to any adjustable parameters (such as the weight coefficients in the polynomial regression example), which allows them to be composed and trained jointly using gradient-based optimization methods.

Section 11.2.3

Chapter 7

5.2.5 Classifier accuracy

The simplest measure of performance for a classifier is the fraction of test set points that are correctly classified. However, we have seen that different types of error can have different consequences, as expressed through the loss matrix, and often we therefore do not simply wish to minimize the number of misclassifications. By changing the location of the decision boundary, we can make trade-offs between different kinds of error, for example with the goal of minimizing an expected loss. Because this is such an important concept, we will introduce some definitions and terminology so that the performance of a classifier can be better characterized.

Section 2.1.1

We will consider again our cancer screening example. For each person tested, there is a 'true label' of whether they have cancer or not, and there is also the prediction made by the classifier. If, for a particular person, the classifier predicts cancer and this is in fact the true label, then the prediction is called a *true positive*. However, if the person does not have cancer it is a *false positive*. Likewise, if the classifier predicts that a person does not have cancer and this is correct, then the prediction is called a *true negative*, otherwise it is a *false negative*. The false positives are also known as *type 1 errors* whereas the false negatives are called *type 2 errors*. If N is the total number of people taking the test, then N_{TP} is the number of true positives, N_{FP} is the number of false positives, N_{TN} is the number of true negatives, and N_{FN} is the number of false negatives, where

$$N = N_{\text{TP}} + N_{\text{FP}} + N_{\text{TN}} + N_{\text{FN}}. \tag{5.28}$$

This can be represented as a *confusion matrix* as shown in Figure 5.9. Accuracy, measured by the fraction of correct classifications, is then given by

$$\text{Accuracy} = \frac{N_{\text{TP}} + N_{\text{TN}}}{N_{\text{TP}} + N_{\text{FP}} + N_{\text{TN}} + N_{\text{FN}}}. \tag{5.29}$$

Telegram: @uni_k

www.konkur.in

We can see that accuracy can be misleading if there are strongly imbalanced classes. In our cancer screening example, for instance, where only 1 person in 1,000 has cancer, a naive classifier that simply decides that nobody has cancer will achieve 99.9% accuracy and yet is completely useless.

Several other quantities can be defined in terms of these numbers, of which the most commonly encountered are

$$\text{Precision} = \frac{N_{\text{TP}}}{N_{\text{TP}} + N_{\text{FP}}} \tag{5.30}$$

$$\text{Recall} = \frac{N_{\text{TP}}}{N_{\text{TP}} + N_{\text{FN}}} \tag{5.31}$$

$$\text{False positive rate} = \frac{N_{\text{FP}}}{N_{\text{FP}} + N_{\text{TN}}} \tag{5.32}$$

$$\text{False discovery rate} = \frac{N_{\text{FP}}}{N_{\text{FP}} + N_{\text{TP}}} \tag{5.33}$$

In our cancer screening example, precision represents an estimate of the probability that a person who has a positive test does indeed have cancer, whereas recall is an estimate of the probability that a person who has cancer is correctly detected by the test. The false positive rate is an estimate of the probability that a person who is normal will be classified as having cancer, whereas the false discovery rate represents the fraction of those testing positive who do not in fact have cancer.

By altering the location of the decision boundary, we can change the trade-offs between the two kinds of errors. To understand this trade-off, we revisit Figure 5.5, but now we label the various regions as shown in Figure 5.10. We can relate the labelled regions to the various true and false rates as follows:

$$N_{\text{FP}}/N = E \tag{5.34}$$

$$N_{\text{TP}}/N = D + E \tag{5.35}$$

$$N_{\text{FN}}/N = B + C \tag{5.36}$$

$$N_{\text{TN}}/N = A + C \tag{5.37}$$

where we are implicitly considering the limit $N \to \infty$ so that we can relate number of observations to probabilities.

5.2.6 ROC curve

A probabilistic classifier will output a posterior probability, which can be converted to a decision by setting a threshold. As the value of the threshold is varied, we can reduce type 1 errors at the expense of increasing type 2 errors, or vice versa. To better understand this trade-off, it is useful to plot the *receiver operating characteristic* or *ROC curve* (Fawcett, 2006), a name that originates from procedures to measure the performance of radar receivers. This is a graph of true positive rate versus false positive rate, as shown in Figure 5.11.

As the decision boundary in Figure 5.10 is moved from $-\infty$ to ∞, the ROC curve is traced out and can then be generated by plotting the cumulative fraction of

Telegram: @uni_k

www.konkur.in

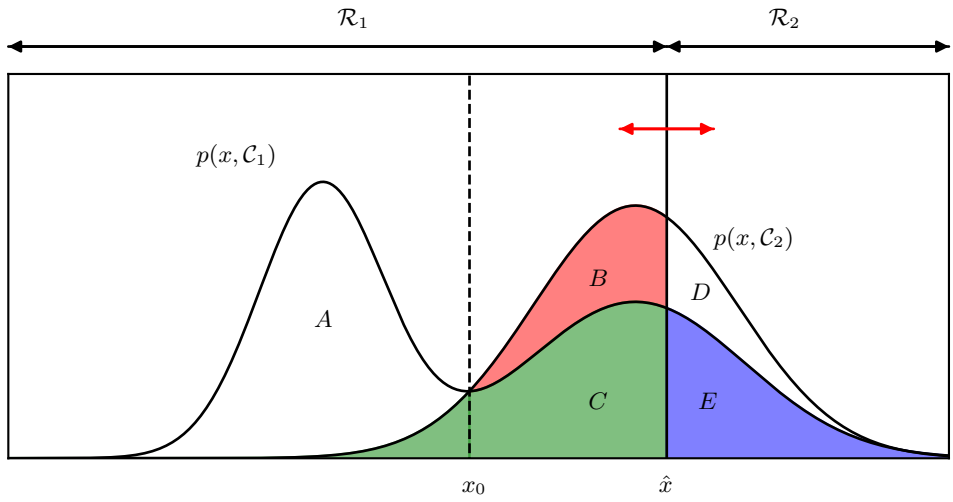


Figure 5.10 As in Figure 5.5, with the various regions labelled. In the cancer classification problem, region $\mathcal{R}_1$ is assigned to the normal class whereas region $\mathcal{R}_2$ is assigned to the cancer class.

correct detection of cancer on the y-axis versus the cumulative fraction of incorrect detection on the x-axis. Note that a specific confusion matrix represents one point along the ROC curve. The best possible classifier would be represented by a point at the top left corner of the ROC diagram. The bottom left corner represents a simple classifier that assigns every point to the normal class and therefore has no true positives but also no false positives. Similarly, the top right corner represents a classifier that assigns everything to the cancer class and therefore has no false negatives but also no true negatives. In Figure 5.11, the classifiers represented by the blue curve are better than those of the red curve for any choice of, say, false positive rate. It is also possible, however, for such curves to cross over, in which case the choice of which is better will depend on the choice of operating point.

As a baseline, we can consider a random classifier that simply assigns each data point to cancer with probability ρ and to normal with probability $1 - \rho$. As we vary the value of ρ it will trace out an ROC curve given by a diagonal straight line, as shown in Figure 5.11. Any classifier below the diagonal line performs worse than random guessing.

Sometimes it is useful to have a single number that characterises the whole ROC curve. One approach is to measure the area under the curve (AUC). A value of 0.5 for the AUC represents random guessing whereas a value of 1.0 represents a perfect classifier.

Another measure is the *F-score*, which is the geometric mean of precision and

Telegram: @uni_k

www.konkur.in

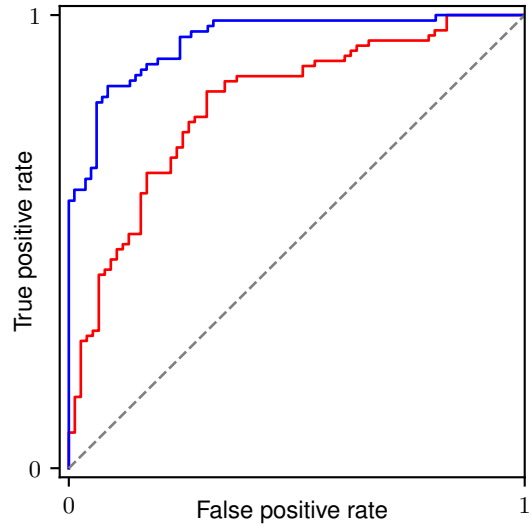


Figure 5.11 The receiver operator characteristic (ROC) curve is a plot of true positive rate against false positive rate, and it characterizes the trade-off between type 1 and type 2 errors in a classification problem. The upper blue curve represents a better classifier than the lower red curve. Here the dashed curve represents the performance of a simple random classifier.

recall, and is therefore defined by

$$F = \frac{2 \times \text{precision} \times \text{recall}}{\text{precision} + \text{recall}} \tag{5.38}$$

$$= \frac{2N_{\text{TP}}}{2N_{\text{TP}} + N_{\text{FP}} + N_{\text{FN}}}. \tag{5.39}$$

Of course, we can also combine the confusion matrix in Figure 5.9 with the loss matrix in Figure 5.6 to compute the expected loss by multiplying the elements pointwise and summing the resulting products.

Although the ROC curve can be extended to more than two classes, it rapidly becomes cumbersome as the number of classes increases.

5.3. Generative Classifiers

Section 5.2.4

We turn next to a probabilistic view of classification and show how models with linear decision boundaries arise from simple assumptions about the distribution of the data. We have already discussed the distinction between the discriminative and the generative approaches to classification. Here we will adopt a generative approach in which we model the class-conditional densities $p(\mathbf{x}|\mathcal{C}_k)$ as well as the class priors $p(\mathcal{C}_k)$ and then use these to compute posterior probabilities $p(\mathcal{C}_k|\mathbf{x})$ through Bayes' theorem.

First, consider problems having two classes. The posterior probability for class

Telegram: @uni_k

www.konkur.in

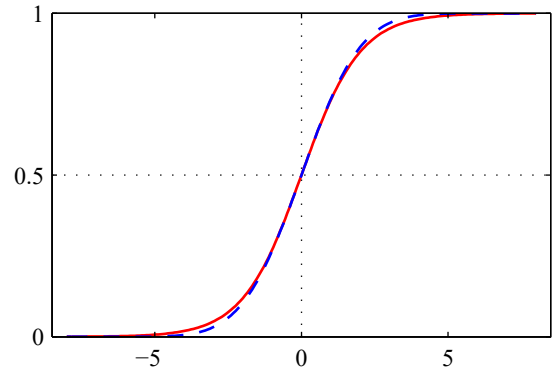


Figure 5.12 Plot of the logistic sigmoid function $\sigma(a)$ defined by (5.42), shown in red, together with the scaled probit function $\Phi(\lambda a)$, for $\lambda^2 = \pi/8$, shown in dashed blue, where $\Phi(a)$ is defined by (5.86). The scaling factor $\pi/8$ is chosen so that the derivatives of the two curves are equal for $a = 0$.

$\mathcal{C}_1$ can be written as

$$\begin{aligned} p(\mathcal{C}_1|\mathbf{x}) &= \frac{p(\mathbf{x}|\mathcal{C}_1)p(\mathcal{C}_1)}{p(\mathbf{x}|\mathcal{C}_1)p(\mathcal{C}_1) + p(\mathbf{x}|\mathcal{C}_2)p(\mathcal{C}_2)} \\ &= \frac{1}{1+\exp(-a)} = \sigma(a) \end{aligned} \tag{5.40}$$

where we have defined

$$a = \ln \frac{p(\mathbf{x}|\mathcal{C}_1)p(\mathcal{C}_1)}{p(\mathbf{x}|\mathcal{C}_2)p(\mathcal{C}_2)} \tag{5.41}$$

and $\sigma(a)$ is the *logistic sigmoid* function defined by

$$\sigma(a) = \frac{1}{1+\exp(-a)}, \tag{5.42}$$

which is plotted in Figure 5.12. The term 'sigmoid' means S-shaped. This type of function is sometimes also called a 'squashing function' because it maps the whole real axis into a finite interval. The logistic sigmoid has been encountered already in earlier chapters and plays an important role in many classification algorithms. It satisfies the following symmetry property:

$$\sigma(-a) = 1 - \sigma(a) \tag{5.43}$$

as is easily verified. The inverse of the logistic sigmoid is given by

$$a = \ln\left(\frac{\sigma}{1-\sigma}\right) \tag{5.44}$$

and is known as the *logit* function. It represents the log of the ratio of probabilities $\ln\left[p(\mathcal{C}_1|\mathbf{x})/p(\mathcal{C}_2|\mathbf{x})\right]$ for the two classes, also known as the *log odds*.

Note that in (5.40), we have simply rewritten the posterior probabilities in an equivalent form, and so the appearance of the logistic sigmoid may seem artificial.

Telegram: @uni_k

www.konkur.in

However, it will have significance provided $a(\mathbf{x})$ has a constrained functional form. We will shortly consider situations in which $a(\mathbf{x})$ is a linear function of $\mathbf{x}$, in which case the posterior probability is governed by a generalized linear model.

If there are $K > 2$ classes, we have

$$\begin{aligned} p(\mathcal{C}_k|\mathbf{x}) &= \frac{p(\mathbf{x}|\mathcal{C}_k)p(\mathcal{C}_k)}{\sum_j p(\mathbf{x}|\mathcal{C}_j)p(\mathcal{C}_j)} \\ &= \frac{\exp(a_k)}{\sum_j \exp(a_j)}, \end{aligned} \tag{5.45}$$

which is known as the *normalized exponential* and can be regarded as a multi-class generalization of the logistic sigmoid. Here the quantities a_k are defined by

$$a_k = \ln\left(p(\mathbf{x}|\mathcal{C}_k)p(\mathcal{C}_k)\right). \tag{5.46}$$

The normalized exponential is also known as the *softmax function*, as it represents a smoothed version of the 'max' function because, if $a_k \gg a_j$ for all $j \neq k$, then $p(\mathcal{C}_k|\mathbf{x}) \simeq 1$, and $p(\mathcal{C}_j|\mathbf{x}) \simeq 0$.

We now investigate the consequences of choosing specific forms for the class-conditional densities, looking first at continuous input variables $\mathbf{x}$ and then discussing briefly discrete inputs.

5.3.1 Continuous inputs

Let us assume that the class-conditional densities are Gaussian. We will then explore the resulting form for the posterior probabilities. To start with, we will assume that all classes share the same covariance matrix $\mathbf{\Sigma}$. Thus, the density for class $\mathcal{C}_k$ is given by

$$p(\mathbf{x}|\mathcal{C}_k) = \frac{1}{(2\pi)^{D/2}} \frac{1}{|\mathbf{\Sigma}|^{1/2}} \exp\left\{-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu}_k)^{\mathrm{T}}\mathbf{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu}_k)\right\}. \tag{5.47}$$

First, suppose that we have two classes. From (5.40) and (5.41), we have

$$p(\mathcal{C}_1|\mathbf{x}) = \sigma(\mathbf{w}^{\mathrm{T}}\mathbf{x} + w_0) \tag{5.48}$$

where we have defined

$$\mathbf{w} = \mathbf{\Sigma}^{-1}(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2) \tag{5.49}$$

$$w_0 = -\frac{1}{2}\boldsymbol{\mu}_1^{\mathrm{T}}\mathbf{\Sigma}^{-1}\boldsymbol{\mu}_1 + \frac{1}{2}\boldsymbol{\mu}_2^{\mathrm{T}}\mathbf{\Sigma}^{-1}\boldsymbol{\mu}_2 + \ln\frac{p(\mathcal{C}_1)}{p(\mathcal{C}_2)}. \tag{5.50}$$

We see that the quadratic terms in $\mathbf{x}$ from the exponents of the Gaussian densities have cancelled (due to the assumption of common covariance matrices), leading to a linear function of $\mathbf{x}$ in the argument of the logistic sigmoid. This result is illustrated for a two-dimensional input space $\mathbf{x}$ in Figure 5.13. The resulting decision boundaries correspond to surfaces along which the posterior probabilities $p(\mathcal{C}_k|\mathbf{x})$

Telegram: @uni_k

www.konkur.in

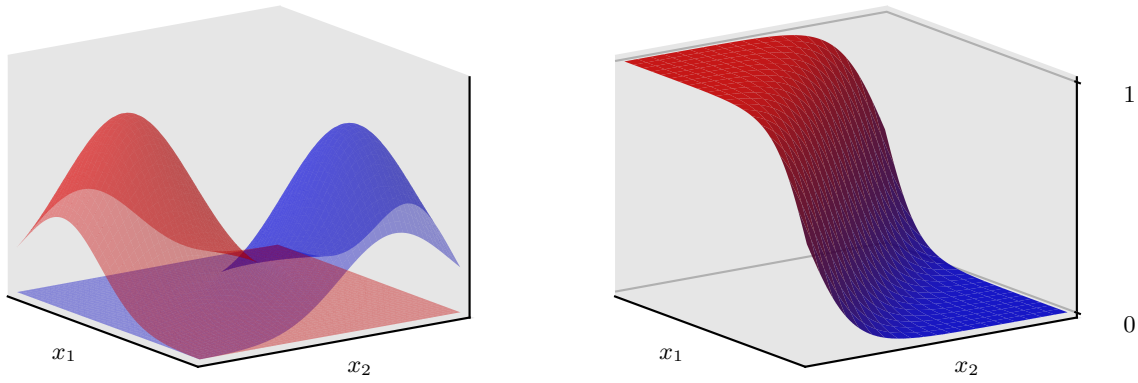


Figure 5.13 The left-hand plot shows the class-conditional densities for two classes, denoted red and blue. On the right is the corresponding posterior probability $p(\mathcal{C}_1|\mathbf{x})$, which is given by a logistic sigmoid of a linear function of $\mathbf{x}$. The surface in the right-hand plot is coloured using a proportion of red ink given by $p(\mathcal{C}_1|\mathbf{x})$ and a proportion of blue ink given by $p(\mathcal{C}_2|\mathbf{x}) = 1 - p(\mathcal{C}_1|\mathbf{x})$.

are constant and so will be given by linear functions of $\mathbf{x}$, and therefore the decision boundaries are linear in input space. The prior probabilities $p(\mathcal{C}_k)$ enter only through the bias parameter w_0, so that changes in the priors have the effect of making parallel shifts of the decision boundary and more generally of the parallel contours of constant posterior probability.

For the general case of K classes, the posterior probabilities are given by (5.45) where, from (5.46) and (5.47), we have

$$a_k(\mathbf{x}) = \mathbf{w}_k^{\mathrm{T}}\mathbf{x} + w_{k0} \tag{5.51}$$

in which we have defined

$$\mathbf{w}_k = \boldsymbol{\Sigma}^{-1}\boldsymbol{\mu}_k \tag{5.52}$$

$$w_{k0} = -\frac{1}{2}\boldsymbol{\mu}_k^{\mathrm{T}}\boldsymbol{\Sigma}^{-1}\boldsymbol{\mu}_k + \ln p(\mathcal{C}_k). \tag{5.53}$$

We see that the $a_k(\mathbf{x})$ are again linear functions of $\mathbf{x}$ as a consequence of the cancellation of the quadratic terms due to the shared covariances. The resulting decision boundaries, corresponding to the minimum misclassification rate, will occur when two of the posterior probabilities (the two largest) are equal, and so will be defined by linear functions of $\mathbf{x}$. Thus, we again have a generalized linear model.

If we relax the assumption of a shared covariance matrix and allow each class-conditional density $p(\mathbf{x}|\mathcal{C}_k)$ to have its own covariance matrix $\boldsymbol{\Sigma}_k$, then the earlier cancellations will no longer occur, and we will obtain quadratic functions of $\mathbf{x}$, giving rise to a *quadratic discriminant*. The linear and quadratic decision boundaries are illustrated in Figure 5.14.

5.3.2 Maximum likelihood solution

Once we have specified a parametric functional form for the class-conditional densities $p(\mathbf{x}|\mathcal{C}_k)$, we can then determine the values of the parameters, together with

Telegram: @uni_k

www.konkur.in

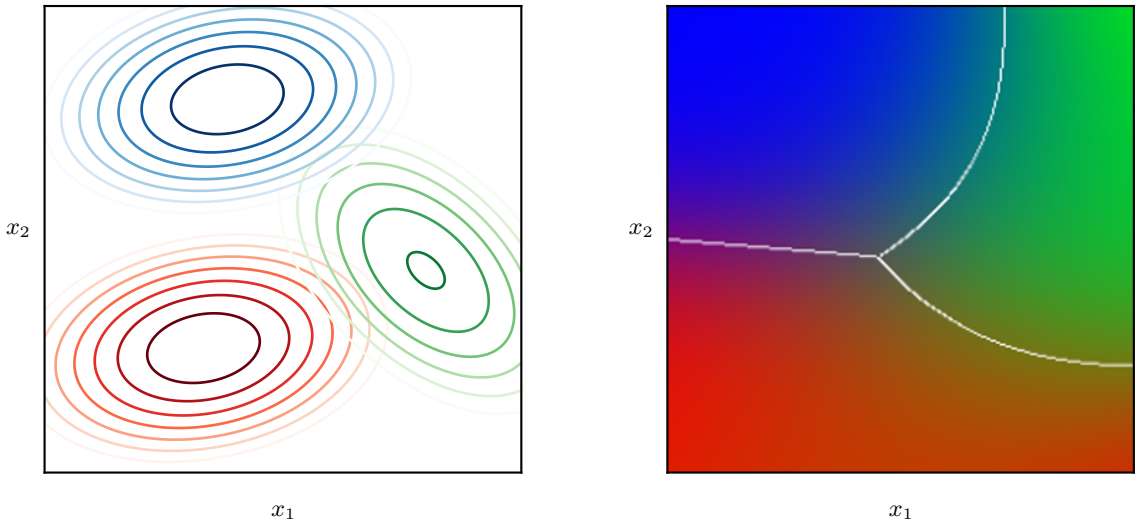


Figure 5.14 The left-hand plot shows the class-conditional densities for three classes each having a Gaussian distribution, coloured red, green, and blue, in which the red and blue classes have the same covariance matrix. The right-hand plot shows the corresponding posterior probabilities, in which each point on the image is coloured using proportions of red, blue, and green ink corresponding to the posterior probabilities for the respective three classes. The decision boundaries are also shown. Notice that the boundary between the red and blue classes, which have the same covariance matrix, is linear, whereas those between the other pairs of classes are quadratic.

the prior class probabilities $p(\mathcal{C}_k)$, using maximum likelihood. This requires a data set comprising observations of $\mathbf{x}$ along with their corresponding class labels.

First, suppose we have two classes, each having a Gaussian class-conditional density with a shared covariance matrix, and suppose we have a data set $\{\mathbf{x}_n, t_n\}$ where $n = 1, \ldots, N$. Here $t_n = 1$ denotes class $\mathcal{C}_1$ and $t_n = 0$ denotes class $\mathcal{C}_2$. We denote the prior class probability $p(\mathcal{C}_1) = \pi$, so that $p(\mathcal{C}_2) = 1 - \pi$. For a data point $\mathbf{x}_n$ from class $\mathcal{C}_1$, we have $t_n = 1$ and hence

$$p(\mathbf{x}_n, \mathcal{C}_1) = p(\mathcal{C}_1)p(\mathbf{x}_n|\mathcal{C}_1) = \pi\mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_1, \boldsymbol{\Sigma}).$$

Similarly for class $\mathcal{C}_2$, we have $t_n = 0$ and hence

$$p(\mathbf{x}_n, \mathcal{C}_2) = p(\mathcal{C}_2)p(\mathbf{x}_n|\mathcal{C}_2) = (1 - \pi)\mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_2, \boldsymbol{\Sigma}).$$

Thus, the likelihood function is given by

$$p(\mathbf{t}, \mathbf{X}|\pi, \boldsymbol{\mu}_1, \boldsymbol{\mu}_2, \boldsymbol{\Sigma}) = \prod_{n=1}^{N} \left[\pi\mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_1, \boldsymbol{\Sigma})\right]^{t_n} \left[(1 - \pi)\mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_2, \boldsymbol{\Sigma})\right]^{1-t_n} \quad (5.54)$$

where $\mathbf{t} = (t_1, \ldots, t_N)^{\mathrm{T}}$. As usual, it is convenient to maximize the log of the likelihood function. Consider first the maximization with respect to π. The terms in

Telegram: @uni_k

www.konkur.in

the log likelihood function that depend on π are

$$\sum_{n=1}^{N} \{t_n \ln \pi + (1 - t_n) \ln(1 - \pi)\} . \tag{5.55}$$

Setting the derivative with respect to π equal to zero and rearranging, we obtain

$$\pi = \frac{1}{N} \sum_{n=1}^{N} t_n = \frac{N_1}{N} = \frac{N_1}{N_1 + N_2} \tag{5.56}$$

where N_1 denotes the total number of data points in class $\mathcal{C}_1$, and N_2 denotes the total number of data points in class $\mathcal{C}_2$. Thus, the maximum likelihood estimate for π is simply the fraction of points in class $\mathcal{C}_1$ as expected. This result is easily generalized to the multi-class case where again the maximum likelihood estimate of the prior probability associated with class $\mathcal{C}_k$ is given by the fraction of the training set points assigned to that class.

Exercise 5.13

Now consider the maximization with respect to $\boldsymbol{\mu}_1$. Again, we can pick out of the log likelihood function those terms that depend on $\boldsymbol{\mu}_1$:

$$\sum_{n=1}^{N} t_n \ln \mathcal{N}(\mathbf{x}_n | \boldsymbol{\mu}_1, \boldsymbol{\Sigma}) = -\frac{1}{2} \sum_{n=1}^{N} t_n (\mathbf{x}_n - \boldsymbol{\mu}_1)^{\mathrm{T}} \boldsymbol{\Sigma}^{-1} (\mathbf{x}_n - \boldsymbol{\mu}_1) + \text{const.} \tag{5.57}$$

Setting the derivative with respect to $\boldsymbol{\mu}_1$ to zero and rearranging, we obtain

$$\boldsymbol{\mu}_1 = \frac{1}{N_1} \sum_{n=1}^{N} t_n \mathbf{x}_n, \tag{5.58}$$

which is simply the mean of all the input vectors $\mathbf{x}_n$ assigned to class $\mathcal{C}_1$. By a similar argument, the corresponding result for $\boldsymbol{\mu}_2$ is given by

$$\boldsymbol{\mu}_2 = \frac{1}{N_2} \sum_{n=1}^{N} (1 - t_n) \mathbf{x}_n, \tag{5.59}$$

which again is the mean of all the input vectors $\mathbf{x}_n$ assigned to class $\mathcal{C}_2$.

Finally, consider the maximum likelihood solution for the shared covariance matrix $\boldsymbol{\Sigma}$. Picking out the terms in the log likelihood function that depend on $\boldsymbol{\Sigma}$, we have

$$\begin{aligned} &-\frac{1}{2} \sum_{n=1}^{N} t_n \ln |\boldsymbol{\Sigma}| - \frac{1}{2} \sum_{n=1}^{N} t_n (\mathbf{x}_n - \boldsymbol{\mu}_1)^{\mathrm{T}} \boldsymbol{\Sigma}^{-1} (\mathbf{x}_n - \boldsymbol{\mu}_1) \\ &-\frac{1}{2} \sum_{n=1}^{N} (1 - t_n) \ln |\boldsymbol{\Sigma}| - \frac{1}{2} \sum_{n=1}^{N} (1 - t_n) (\mathbf{x}_n - \boldsymbol{\mu}_2)^{\mathrm{T}} \boldsymbol{\Sigma}^{-1} (\mathbf{x}_n - \boldsymbol{\mu}_2) \\ &= -\frac{N}{2} \ln |\boldsymbol{\Sigma}| - \frac{N}{2} \mathrm{Tr} \left\{ \boldsymbol{\Sigma}^{-1} \mathbf{S} \right\} \end{aligned} \tag{5.60}$$

Telegram: @uni_k

www.konkur.in

where we have defined

$$\mathbf{S} = \frac{N_1}{N}\mathbf{S}_1 + \frac{N_2}{N}\mathbf{S}_2 \tag{5.61}$$

$$\mathbf{S}_1 = \frac{1}{N_1}\sum_{n\in\mathcal{C}_1}(\mathbf{x}_n - \boldsymbol{\mu}_1)(\mathbf{x}_n - \boldsymbol{\mu}_1)^{\mathrm{T}} \tag{5.62}$$

$$\mathbf{S}_2 = \frac{1}{N_2}\sum_{n\in\mathcal{C}_2}(\mathbf{x}_n - \boldsymbol{\mu}_2)(\mathbf{x}_n - \boldsymbol{\mu}_2)^{\mathrm{T}}. \tag{5.63}$$

Using the standard result for the maximum likelihood solution for a Gaussian distribution, we see that $\boldsymbol{\Sigma} = \mathbf{S}$, which represents a weighted average of the covariance matrices associated with each of the two classes separately.

This result is easily extended to the K-class problem to obtain the corresponding maximum likelihood solutions for the parameters in which each class-conditional density is Gaussian with a shared covariance matrix. Note that the approach of fitting Gaussian distributions to the classes is not robust to outliers, because the maximum likelihood estimation of a Gaussian is not robust.

Exercise 5.14

Section 5.1.4

5.3.3 Discrete features

Let us now consider discrete feature values x_i. For simplicity, we begin by looking at binary feature values $x_i \in \{0, 1\}$ and discuss the extension to more general discrete features shortly. If there are D inputs, then a general distribution would correspond to a table of 2^D numbers for each class and have $2^D - 1$ independent variables (due to the summation constraint). Because this grows exponentially with the number of features, we can seek a more restricted representation. Here we will make the *naive Bayes* assumption in which the feature values are treated as independent and conditioned on the class $\mathcal{C}_k$. Thus, we have class-conditional distributions of the form

Section 11.2.3

$$p(\mathbf{x}|\mathcal{C}_k) = \prod_{i=1}^{D}\mu_{ki}^{x_i}(1 - \mu_{ki})^{1-x_i}, \tag{5.64}$$

which contain D independent parameters for each class. Substituting into (5.46) then gives

$$a_k(\mathbf{x}) = \sum_{i=1}^{D}\{x_i \ln \mu_{ki} + (1 - x_i)\ln(1 - \mu_{ki})\} + \ln p(\mathcal{C}_k), \tag{5.65}$$

which again are linear functions of the input values x_i. For $K = 2$ classes, we can alternatively consider the logistic sigmoid formulation given by (5.40). Analogous results are obtained for discrete variables that take $L > 2$ states.

Exercise 5.16

5.3.4 Exponential family

As we have seen, for both Gaussian distributed and discrete inputs, the posterior class probabilities are given by generalized linear models with logistic sigmoid ($K =$

Telegram: @uni_k

www.konkur.in

2 classes) or softmax ($K \geqslant 2$ classes) activation functions. These are particular cases of a more general result obtained by assuming that the class-conditional densities $p(\mathbf{x}|\mathcal{C}_k)$ are members of the subset of the exponential family of distributions given by

Section 3.4

$$p(\mathbf{x}|\boldsymbol{\lambda}_k, s) = \frac{1}{s} h\left(\frac{1}{s}\mathbf{x}\right) g(\boldsymbol{\lambda}_k) \exp\left\{\frac{1}{s}\boldsymbol{\lambda}_k^{\mathrm{T}}\mathbf{x}\right\}. \tag{5.66}$$

Here the scaling parameter s is shared across all the classes.

For the two-class problem, we substitute this expression for the class-conditional densities into (5.41) and we see that the posterior class probability is again given by a logistic sigmoid acting on a linear function $a(\mathbf{x})$, which is given by

$$a(\mathbf{x}) = (\boldsymbol{\lambda}_1 - \boldsymbol{\lambda}_2)^{\mathrm{T}}\mathbf{x} + \ln g(\boldsymbol{\lambda}_1) - \ln g(\boldsymbol{\lambda}_2) + \ln p(\mathcal{C}_1) - \ln p(\mathcal{C}_2). \tag{5.67}$$

Similarly, for the K-class problem, we substitute the class-conditional density expression into (5.46) to give

$$a_k(\mathbf{x}) = \boldsymbol{\lambda}_k^{\mathrm{T}}\mathbf{x} + \ln g(\boldsymbol{\lambda}_k) + \ln p(\mathcal{C}_k) \tag{5.68}$$

and so again is a linear function of $\mathbf{x}$.

5.4. Discriminative Classifiers

For the two-class classification problem, we have seen that the posterior probability of class $\mathcal{C}_1$ can be written as a logistic sigmoid acting on a linear function of $\mathbf{x}$, for a wide choice of class-conditional distributions $p(\mathbf{x}|\mathcal{C}_k)$ from the exponential family. Similarly, for the multi-class case, the posterior probability of class $\mathcal{C}_k$ is given by a softmax transformation of linear functions of $\mathbf{x}$. For specific choices of the class-conditional densities $p(\mathbf{x}|\mathcal{C}_k)$, we have used maximum likelihood to determine the parameters of the densities as well as the class priors $p(\mathcal{C}_k)$ and then used Bayes' theorem to find the posterior class probabilities. This represents an example of *generative* modelling, because we could take such a model and generate synthetic data by drawing values of $\mathbf{x}$ from the marginal distribution $p(\mathbf{x})$ or from any of the class-conditional densities $p(\mathbf{x}|\mathcal{C}_k)$.

However, an alternative approach is to use the functional form of the generalized linear model explicitly and to determine its parameters directly by using maximum likelihood. In this direct approach, we maximize a likelihood function defined through the conditional distribution $p(\mathcal{C}_k|\mathbf{x})$, which represents a form of *discriminative* probabilistic modelling. One advantage of the discriminative approach is that there will typically be fewer learnable parameters to be determined, as we will see shortly. It may also lead to improved predictive performance, particularly when the assumed forms for the class-conditional densities represent a poor approximation to the true distributions.

Telegram: @uni_k

www.konkur.in

5.4.1 Activation functions

Chapter 4

In linear regression, the model prediction $y(\mathbf{x}, \mathbf{w})$ is given by a linear function of the parameters

$$y(\mathbf{x}, \mathbf{w}) = \mathbf{w}^{\mathrm{T}}\mathbf{x} + w_0, \tag{5.69}$$

which gives a continuous-valued output in the range $(-\infty, \infty)$. For classification problems, however, we wish to predict discrete class labels, or more generally posterior probabilities that lie in the range $(0, 1)$. To achieve this, we consider a generalization of this model in which we transform the linear function of $\mathbf{w}$ and w_0 using a nonlinear function $f(\,\cdot\,)$ so that

$$y(\mathbf{x}, \mathbf{w}) = f\left(\mathbf{w}^{\mathrm{T}}\mathbf{w} + w + 0\right). \tag{5.70}$$

In the machine learning literature, $f(\,\cdot\,)$ is known as an *activation function*, whereas its inverse is called a *link function* in the statistics literature. The decision surfaces correspond to $y(\mathbf{x}) = \text{constant}$, so that $\mathbf{w}^{\mathrm{T}}\mathbf{x} = \text{constant}$, and hence the decision surfaces are linear functions of $\mathbf{x}$, even if the function $f(\cdot)$ is nonlinear. For this reason, the class of models described by (5.70) are called *generalized linear models* (McCullagh and Nelder, 1989). However, in contrast to the models used for regression, they are no longer linear in the parameters due to the nonlinear function $f(\cdot)$. This will lead to more complex analytical and computational properties than for linear regression models. Nevertheless, these models are still relatively simple compared to the much more flexible nonlinear models that will be studied in subsequent chapters.

5.4.2 Fixed basis functions

So far in this chapter, we have considered classification models that work directly with the original input vector $\mathbf{x}$. However, all the algorithms are equally applicable if we first make a fixed nonlinear transformation of the inputs using a vector of basis functions $\boldsymbol{\phi}(\mathbf{x})$. The resulting decision boundaries will be linear in the feature space $\boldsymbol{\phi}$, and these correspond to nonlinear decision boundaries in the original $\mathbf{x}$ space, as illustrated in Figure 5.15. Classes that are linearly separable in the feature space $\boldsymbol{\phi}(\mathbf{x})$ need not be linearly separable in the original observation space $\mathbf{x}$.

Note that as in our discussion of linear models for regression, one of the basis functions is typically set to a constant, say $\phi_0(\mathbf{x}) = 1$, so that the corresponding parameter w_0 plays the role of a bias.

For many problems of practical interest, there is significant overlap in $\mathbf{x}$-space between the class-conditional densities $p(\mathbf{x}|\mathcal{C}_k)$. This corresponds to posterior probabilities $p(\mathcal{C}_k|\mathbf{x})$, which, for at least some values of $\mathbf{x}$, are not 0 or 1. In such cases, the optimal solution is obtained by modelling the posterior probabilities accurately and then applying standard decision theory. Note that nonlinear transformations $\boldsymbol{\phi}(\mathbf{x})$ cannot remove such a class overlap, although they can increase the level of overlap or create an overlap where none existed in the original observation space. However, suitable choices of nonlinearity can make the process of modelling the posterior probabilities easier. However, such fixed basis function models have important limitations, and these will be resolved in later chapters by allowing the basis functions themselves to adapt to the data.

Section 5.2

Section 6.1

Telegram: @uni_k

www.konkur.in

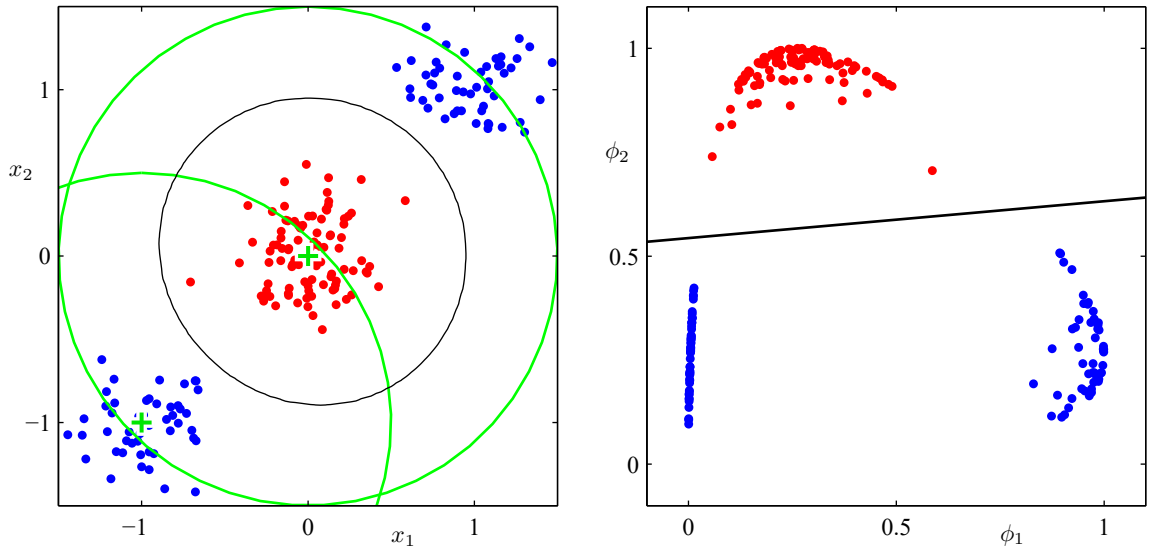


Figure 5.15 Illustration of the role of nonlinear basis functions in linear classification models. The left-hand plot shows the original input space (x_1, x_2) together with data points from two classes labelled red and blue. Two 'Gaussian' basis functions $\phi_1(\mathbf{x})$ and $\phi_2(\mathbf{x})$ are defined in this space with centres shown by the green crosses and with contours shown by the green circles. The right-hand plot shows the corresponding feature space (ϕ_1, ϕ_2) together with the linear decision boundary obtained given by a logistic regression model of the form discussed in Section 5.4.3. This corresponds to a nonlinear decision boundary in the original input space, shown by the black curve in the left-hand plot.

5.4.3 Logistic regression

We first consider the problem of two-class classification. In our discussion of generative approaches in Section 5.3, we saw that under rather general assumptions, the posterior probability of class $\mathcal{C}_1$ can be written as a logistic sigmoid acting on a linear function of the feature vector $\boldsymbol{\phi}$ so that

$$p(\mathcal{C}_1|\boldsymbol{\phi}) = y(\boldsymbol{\phi}) = \sigma\left(\mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}\right) \tag{5.71}$$

with $p(\mathcal{C}_2|\boldsymbol{\phi}) = 1 - p(\mathcal{C}_1|\boldsymbol{\phi})$. Here $\sigma(\cdot)$ is the *logistic sigmoid* function defined by (5.42). In the terminology of statistics, this model is known as *logistic regression*, although it should be emphasized that this is a model for classification rather than for continuous variable.

For an M-dimensional feature space $\boldsymbol{\phi}$, this model has M adjustable parameters. By contrast, if we had fitted Gaussian class-conditional densities using maximum likelihood, we would have used $2M$ parameters for the means and $M(M+1)/2$ parameters for the (shared) covariance matrix. Together with the class prior $p(\mathcal{C}_1)$, this gives a total of $M(M+5)/2+1$ parameters, which grows quadratically with M, in contrast to the linear dependence on M of the number of parameters in logistic regression. For large values of M, there is a clear advantage in working with the logistic regression model directly.

Telegram: @uni_k

www.konkur.in

We now use maximum likelihood to determine the parameters of the logistic regression model. To do this, we will make use of the derivative of the logistic sigmoid function, which can conveniently be expressed in terms of the sigmoid function itself:

Exercise 5.18

$$\frac{\mathrm{d}\sigma}{\mathrm{d}a} = \sigma(1-\sigma). \tag{5.72}$$

For a data set $\{\boldsymbol{\phi}_n, t_n\}$, where $\boldsymbol{\phi}_n = \boldsymbol{\phi}(\mathbf{x}_n)$ and $t_n \in \{0, 1\}$, with $n = 1, \ldots, N$, the likelihood function can be written

$$p(\mathbf{t}|\mathbf{w}) = \prod_{n=1}^{N} y_n^{t_n} \{1 - y_n\}^{1-t_n} \tag{5.73}$$

where $\mathbf{t} = (t_1, \ldots, t_N)^{\mathrm{T}}$ and $y_n = p(\mathcal{C}_1|\boldsymbol{\phi}_n)$. As usual, we can define an error function by taking the negative logarithm of the likelihood, which gives the *cross-entropy* error function:

$$E(\mathbf{w}) = -\ln p(\mathbf{t}|\mathbf{w}) = -\sum_{n=1}^{N} \{t_n \ln y_n + (1 - t_n) \ln(1 - y_n)\} \tag{5.74}$$

where $y_n = \sigma(a_n)$ and $a_n = \mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}_n$. Taking the gradient of the error function with respect to $\mathbf{w}$, we obtain

Exercise 5.19

$$\nabla E(\mathbf{w}) = \sum_{n=1}^{N} (y_n - t_n)\boldsymbol{\phi}_n \tag{5.75}$$

where we have made use of (5.72). We see that the factor involving the derivative of the logistic sigmoid has cancelled, leading to a simplified form for the gradient of the log likelihood. In particular, the contribution to the gradient from data point n is given by the 'error' $y_n - t_n$ between the target value and the prediction of the model times the basis function vector $\boldsymbol{\phi}_n$. Furthermore, comparison with (4.12) shows that this takes precisely the same form as the gradient of the sum-of-squares error function for the linear regression model.

Section 4.1.3

The maximum likelihood solution corresponds to $\nabla E(\mathbf{w}) = 0$. However, from (5.75) we see that this no longer corresponds to a set of linear equations, due to the nonlinearity in $y(\cdot)$, and so this equation does not have a closed-form solution. One approach to finding a maximum likelihood solution would be to use stochastic gradient descent, in which ∇E_n is the nth term on the right-hand side of (5.75).

Chapter 7

Stochastic gradient descent will be the principal approach to training the highly non-linear neural networks discussed in later chapters. However, the maximum likelihood equation is only 'slightly' nonlinear, and in fact the error function (5.74), in which the model is defined by (5.71), is a convex function of the parameters, which allows the error function to be minimized using a simple algorithm called *iterative reweighted least squares* or IRLS (Bishop, 2006). However, this does not easily generalize to more complex models such as deep neural networks.

Telegram: @uni_k

www.konkur.in

Note that maximum likelihood can exhibit severe over-fitting for data sets that are linearly separable. This arises because the maximum likelihood solution occurs when the hyperplane corresponding to $\sigma = 0.5$, equivalent to $\mathbf{w}^{\mathrm{T}}\boldsymbol{\phi} = 0$, separates the two classes and the magnitude of $\mathbf{w}$ goes to infinity. In this case, the logistic sigmoid function becomes infinitely steep in feature space, corresponding to a Heaviside step function, so that every training point from each class k is assigned a posterior probability $p(\mathcal{C}_k|\mathbf{x}) = 1$. Furthermore, there is typically a continuum of such solutions because any separating hyperplane will give rise to the same posterior probabilities at the training data points. Maximum likelihood provides no way to favour one such solution over another, and which solution is found in practice will depend on the choice of optimization algorithm and on the parameter initialization. Note that the problem will arise even if the number of data points is large compared with the number of parameters in the model, so long as the training data set is linearly separable. The singularity can be avoided by adding a regularization term to the error function.

Exercise 5.20

Chapter 9

5.4.4 Multi-class logistic regression

Section 5.3

In our discussion of generative models for multi-class classification, we have seen that, for a large class of distributions from the exponential family, the posterior probabilities are given by a softmax transformation of linear functions of the feature variables, so that

$$p(\mathcal{C}_k|\boldsymbol{\phi}) = y_k(\boldsymbol{\phi}) = \frac{\exp(a_k)}{\sum_j \exp(a_j)} \tag{5.76}$$

where the pre-activations a_k are given by

$$a_k = \mathbf{w}_k^{\mathrm{T}}\boldsymbol{\phi}. \tag{5.77}$$

There we used maximum likelihood to determine separately the class-conditional densities and the class priors and then found the corresponding posterior probabilities using Bayes' theorem, thereby implicitly determining the parameters $\{\mathbf{w}_k\}$. Here we consider the use of maximum likelihood to determine the parameters $\{\mathbf{w}_k\}$ of this model directly. To do this, we will require the derivatives of y_k with respect to all the pre-activations a_j. These are given by

Exercise 5.21

$$\frac{\partial y_k}{\partial a_j} = y_k(I_{kj} - y_j) \tag{5.78}$$

where I_{kj} are the elements of the identity matrix.

Next we write down the likelihood function. This is most easily done using the 1-of-K coding scheme in which the target vector $\mathbf{t}_n$ for a feature vector $\boldsymbol{\phi}_n$ belonging to class $\mathcal{C}_k$ is a binary vector with all elements zero except for element k, which equals one. The likelihood function is then given by

$$p(\mathbf{T}|\mathbf{w}_1, \ldots, \mathbf{w}_K) = \prod_{n=1}^{N}\prod_{k=1}^{K} p(\mathcal{C}_k|\boldsymbol{\phi}_n)^{t_{nk}} = \prod_{n=1}^{N}\prod_{k=1}^{K} y_{nk}^{t_{nk}} \tag{5.79}$$

Telegram: @uni_k

www.konkur.in

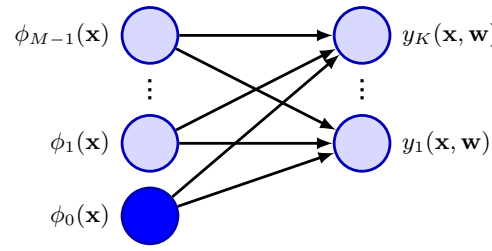


Figure 5.16 Representation of a multi-class linear classification model as a neural network having a single layer of connections. Each basis function is represented by a node, with the solid node representing the 'bias' basis function ϕ_0, whereas each output $y_1, \ldots, y_N$ is also represented by a node. The links between the nodes represent the corresponding weight and bias parameters.

where $y_{nk} = y_k(\boldsymbol{\phi}_n)$, and $\mathbf{T}$ is an $N \times K$ matrix of target variables with elements t_{nk}. Taking the negative logarithm then gives

$$E(\mathbf{w}_1, \ldots, \mathbf{w}_K) = -\ln p(\mathbf{T}|\mathbf{w}_1, \ldots, \mathbf{w}_K) = -\sum_{n=1}^{N}\sum_{k=1}^{K} t_{nk} \ln y_{nk}, \tag{5.80}$$

which is known as the *cross-entropy* error function for the multi-class classification problem.

We now take the gradient of the error function with respect to one of the parameter vectors $\mathbf{w}_j$. Making use of the result (5.78) for the derivatives of the softmax function, we obtain

Exercise 5.22

$$\nabla_{\mathbf{w}_j} E(\mathbf{w}_1, \ldots, \mathbf{w}_K) = \sum_{n=1}^{N} (y_{nj} - t_{nj})\, \boldsymbol{\phi}_n \tag{5.81}$$

where we have made use of $\sum_k t_{nk} = 1$. Again, we could optimize the parameters through stochastic gradient descent.

Chapter 7

Once again, we see the same form arising for the gradient as was found for the sum-of-squares error function with the linear model and for the cross-entropy error with the logistic regression model, namely the product of the error $(y_{nj} - t_{nj})$ times the basis function activation $\boldsymbol{\phi}_n$. These are examples of a more general result that we will explore later.

Section 5.4.6

Linear classification models can be represented as single-layer neural networks as shown in Figure 5.16. If we consider the derivative of the error function with respect to a weight w_{ik}, which links basis function $\phi_i(\mathbf{x})$ to output unit t_k, we have from (5.81)

$$\frac{\partial E(\mathbf{w}_1, \ldots, \mathbf{w}_K)}{\partial w_{ij}} = \sum_{n=1}^{N} (y_{nk} - t_{nk})\, \phi_i(\mathbf{x}_n). \tag{5.82}$$

Comparing this with Figure 5.16, we see that, for each data point n this gradient takes the form of the output of the basis function at the input end of the weight link with the 'error' $(y_{nk} - t_{nk})$ at the output end.

Telegram: @uni_k

www.konkur.in

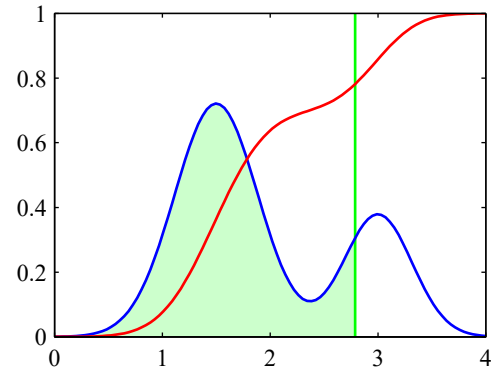


Figure 5.17 Schematic example of a probability density $p(\theta)$ shown by the blue curve, given in this example by a mixture of two Gaussians, along with its cumulative distribution function $f(a)$, shown by the red curve. Note that the value of the blue curve at any point, such as that indicated by the vertical green line, corresponds to the slope of the red curve at the same point. Conversely, the value of the red curve at this point corresponds to the area under the blue curve indicated by the shaded green region. In the stochastic threshold model, the class label takes the value $t = 1$ if the value of $a = \mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}$ exceeds a threshold, otherwise it takes the value $t = 0$. This is equivalent to an activation function given by the cumulative distribution function $f(a)$.

5.4.5 Probit regression

We have seen that, for a broad range of class-conditional distributions described by the exponential family, the resulting posterior class probabilities are given by a logistic (or softmax) transformation acting on a linear function of the feature variables. However, not all choices of class-conditional density give rise to such a simple form for the posterior probabilities, which suggests that it might be worth exploring other types of discriminative probabilistic model. Consider the two-class case, again remaining within the framework of generalized linear models, so that

$$p(t = 1|a) = f(a) \tag{5.83}$$

where $a = \mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}$, and $f(\cdot)$ is the activation function.

One way to motivate an alternative choice for the link function is to consider a noisy threshold model, as follows. For each input $\boldsymbol{\phi}_n$, we evaluate $a_n = \mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}_n$ and then we set the target value according to

$$\begin{cases} t_n = 1, & \text{if } a_n \geqslant \theta, \\ t_n = 0, & \text{otherwise.} \end{cases} \tag{5.84}$$

If the value of θ is drawn from a probability density $p(\theta)$, then the corresponding activation function will be given by the cumulative distribution function

$$f(a) = \int_{-\infty}^{a} p(\theta)\,\mathrm{d}\theta \tag{5.85}$$

as illustrated in Figure 5.17.

As a specific example, suppose that the density $p(\theta)$ is given by a zero-mean, unit-variance Gaussian. The corresponding cumulative distribution function is given by

$$\Phi(a) = \int_{-\infty}^{a} \mathcal{N}(\theta|0, 1)\,\mathrm{d}\theta, \tag{5.86}$$

Telegram: @uni_k

www.konkur.in

which is known as the *probit* function. It has a sigmoidal shape and is compared with the logistic sigmoid function in Figure 5.12. Note that the use of a Gaussian distribution with general mean and variances does not change the model because this is equivalent to a re-scaling of the linear coefficients $\mathbf{w}$. Many numerical packages can evaluate a closely related function defined by

$$\text{erf}(a) = \frac{2}{\sqrt{\pi}} \int_0^a \exp(-\theta^2/2)\,\mathrm{d}\theta \tag{5.87}$$

and known as the *erf function* or *error function* (not to be confused with the error function of a machine learning model). It is related to the probit function by

Exercise 5.23

$$\Phi(a) = \frac{1}{2}\left\{1 + \frac{1}{\sqrt{2}}\text{erf}(a)\right\}. \tag{5.88}$$

The generalized linear model based on a probit activation function is known as *probit regression*. We can determine the parameters of this model using maximum likelihood by a straightforward extension of the ideas discussed earlier. In practice, the results found using probit regression tend to be like those of logistic regression.

One issue that can occur in practical applications is that of *outliers*, which can arise for instance through errors in measuring the input vector $\mathbf{x}$ or through mislabelling of the target value t. Because such points can lie a long way to the wrong side of the ideal decision boundary, they can seriously distort the classifier. The logistic and probit regression models behave differently in this respect because the tails of the logistic sigmoid decay asymptotically like $\exp(-x)$ for $|x| \to \infty$, whereas for the probit activation function, they decay like $\exp(-x^2)$, and so the probit model can be significantly more sensitive to outliers.

5.4.6 Canonical link functions

For the linear regression model with a Gaussian noise distribution, the error function, corresponding to the negative log likelihood, is given by (4.11). If we take the derivative with respect to the parameter vector $\mathbf{w}$ of the contribution to the error function from a data point n, this takes the form of the 'error' $y_n - t_n$ times the feature vector $\boldsymbol{\phi}_n$, where $y_n = \mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}_n$. Similarly, for the combination of the logistic-sigmoid activation function and the cross-entropy error function (5.74) and for the softmax activation function with the multi-class cross-entropy error function (5.80), we again obtain this same simple form. We now show that this is a general result of assuming a conditional distribution for the target variable from the exponential family along with a corresponding choice for the activation function known as the *canonical link function*.

We again make use of the restricted form (3.169) of exponential family distributions. Note that here we are applying the assumption of exponential family distribution to the target variable t, in contrast to Section 5.3.4 where we applied it to the input vector $\mathbf{x}$. We therefore consider conditional distributions of the target variable of the form

$$p(t|\eta, s) = \frac{1}{s}h\left(\frac{t}{s}\right) g(\eta)\exp\left\{\frac{\eta t}{s}\right\}. \tag{5.89}$$

Telegram: @uni_k

www.konkur.in

Using the same line of argument as led to the derivation of the result (3.172), we see that the conditional mean of t, which we denote by y, is given by

$$y \equiv \mathbb{E}[t|\eta] = -s\frac{d}{d\eta}\ln g(\eta). \tag{5.90}$$

Thus, y and η must related, and we denote this relation through $\eta = \psi(y)$.

Following Nelder and Wedderburn (1972), we define a *generalized linear model* to be one for which y is a nonlinear function of a linear combination of the input (or feature) variables so that

$$y = f(\mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}) \tag{5.91}$$

where $f(\cdot)$ is known as the activation function in the machine learning literature, and $f^{-1}(\cdot)$ is known as the link function in statistics.

Now consider the log likelihood function for this model, which, as a function of η, is given by

$$\ln p(\mathbf{t}|\eta, s) = \sum_{n=1}^{N} \ln p(t_n|\eta, s) = \sum_{n=1}^{N} \left\{ \ln g(\eta_n) + \frac{\eta_n t_n}{s} \right\} + \text{const} \tag{5.92}$$

where we are assuming that all observations share a common scale parameter (which corresponds to the noise variance for a Gaussian distribution, for instance) and so s is independent of n. The derivative of the log likelihood with respect to the model parameters $\mathbf{w}$ is then given by

$$\begin{aligned} \nabla_{\mathbf{w}} \ln p(\mathbf{t}|\eta, s) &= \sum_{n=1}^{N} \left\{ \frac{\mathrm{d}}{\mathrm{d}\eta_n} \ln g(\eta_n) + \frac{t_n}{s} \right\} \frac{\mathrm{d}\eta_n}{\mathrm{d}y_n} \frac{\mathrm{d}y_n}{\mathrm{d}a_n} \nabla_{\mathbf{w}} a_n \\ &= \sum_{n=1}^{N} \frac{1}{s} \{t_n - y_n\} \psi'(y_n) f'(a_n) \boldsymbol{\phi}_n \end{aligned} \tag{5.93}$$

where $a_n = \mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}_n$, and we have used $y_n = f(a_n)$ together with the result (5.90) for $\mathbb{E}[t|\eta]$. We now see that there is a considerable simplification if we choose a particular form for the link function $f^{-1}(y)$ given by

$$f^{-1}(y) = \psi(y), \tag{5.94}$$

which gives $f(\psi(y)) = y$ and hence $f'(\psi)\psi'(y) = 1$. Also, because $a = f^{-1}(y)$, we have $a = \psi$ and hence $f'(a)\psi'(y) = 1$. In this case, the gradient of the error function reduces to

$$\nabla \ln E(\mathbf{w}) = \frac{1}{s} \sum_{n=1}^{N} \{y_n - t_n\} \boldsymbol{\phi}_n. \tag{5.95}$$

We have seen that there is a natural pairing between the choice of error function and the choice of output-unit activation function. Although we have derived this result in the context of single-layer network models, the same considerations apply to deep neural networks discussed in later chapters.

Telegram: @uni_k

www.konkur.in

Exercises

5.1 ($\star$) Consider a classification problem with K classes and a target vector $\mathbf{t}$ that uses a 1-of-K binary coding scheme. Show that the conditional expectation $\mathbb{E}[\mathbf{t}|\mathbf{x}]$ is given by the posterior probability $p(\mathcal{C}_k|\mathbf{x})$.

5.2 ($\star\star$) Given a set of data points $\{\mathbf{x}_n\}$, we can define the *convex hull* to be the set of all points $\mathbf{x}$ given by

$$\mathbf{x} = \sum_n \alpha_n \mathbf{x}_n \tag{5.96}$$

where $\alpha_n \geqslant 0$ and $\sum_n \alpha_n = 1$. Consider a second set of points $\{\mathbf{y}_n\}$ together with their corresponding convex hull. By definition, the two sets of points will be linearly separable if there exists a vector $\widehat{\mathbf{w}}$ and a scalar w_0 such that $\widehat{\mathbf{w}}^{\mathrm{T}}\mathbf{x}_n + w_0 > 0$ for all $\mathbf{x}_n$ and $\widehat{\mathbf{w}}^{\mathrm{T}}\mathbf{y}_n + w_0 < 0$ for all $\mathbf{y}_n$. Show that if their convex hulls intersect, the two sets of points cannot be linearly separable, and conversely that if they are linearly separable, their convex hulls do not intersect.

5.3 ($\star\star$) Consider the minimization of a sum-of-squares error function (5.14), and suppose that all the target vectors in the training set satisfy a linear constraint

$$\mathbf{a}^{\mathrm{T}}\mathbf{t}_n + b = 0 \tag{5.97}$$

where $\mathbf{t}_n$ corresponds to the nth row of the matrix $\mathbf{T}$ in (5.14). Show that as a consequence of this constraint, the elements of the model prediction $\mathbf{y}(\mathbf{x})$ given by the least-squares solution (5.16) also satisfy this constraint, so that

$$\mathbf{a}^{\mathrm{T}}\mathbf{y}(\mathbf{x}) + b = 0. \tag{5.98}$$

To do so, assume that one of the basis functions $\phi_0(\mathbf{x}) = 1$ so that the corresponding parameter w_0 plays the role of a bias.

5.4 ($\star\star$) Extend the result of Exercise 5.3 to show that if multiple linear constraints are satisfied simultaneously by the target vectors, then the same constraints will also be satisfied by the least-squares prediction of a linear model.

5.5 ($\star$) Use the definition (5.38), along with (5.30) and (5.31) to derive the result (5.39) for the F-score.

5.6 ($\star\star$) Consider two non-negative numbers a and b, and show that, if $a \leqslant b$, then $a \leqslant (ab)^{1/2}$. Use this result to show that, if the decision regions of a two-class classification problem are chosen to minimize the probability of misclassification, this probability will satisfy

$$p(\text{mistake}) \leqslant \int \{p(\mathbf{x}, \mathcal{C}_1)p(\mathbf{x}, \mathcal{C}_2)\}^{1/2}\, \mathrm{d}\mathbf{x}. \tag{5.99}$$

5.7 ($\star$) Given a loss matrix with elements L_{kj}, the expected risk is minimized if, for each $\mathbf{x}$, we choose the class that minimizes (5.23). Verify that, when the loss matrix

Telegram: @uni_k

www.konkur.in

is given by $L_{kj} = 1 - I_{kj}$, where I_{kj} are the elements of the identity matrix, this reduces to the criterion of choosing the class having the largest posterior probability. What is the interpretation of this form of loss matrix?

5.8 ($\star$) Derive the criterion for minimizing the expected loss when there is a general loss matrix and general prior probabilities for the classes.

5.9 ($\star$) Consider the average of the posterior probabilities over a set of N data points in the form

$$\frac{1}{N}\sum_{N=1}^{N} p(\mathcal{C}_k|\mathbf{x}_n). \tag{5.100}$$

By taking the limit $N \to \infty$, show that this quantity approaches the prior class probability $p(\mathcal{C}_k)$.

5.10 ($\star\star$) Consider a classification problem in which the loss incurred when an input vector from class $\mathcal{C}_k$ is classified as belonging to class $\mathcal{C}_j$ is given by the loss matrix L_{kj} and for which the loss incurred in selecting the reject option is λ. Find the decision criterion that will give the minimum expected loss. Verify that this reduces to the reject criterion discussed in Section 5.2.3 when the loss matrix is given by $L_{kj} = 1 - I_{kj}$. What is the relationship between λ and the rejection threshold θ?

5.11 ($\star$) Show that the logistic sigmoid function (5.42) satisfies the property $\sigma(-a) = 1 - \sigma(a)$ and that its inverse is given by $\sigma^{-1}(y) = \ln\{y/(1-y)\}$.

5.12 ($\star$) Using (5.40) and (5.41), derive the result (5.48) for the posterior class probability in the two-class generative model with Gaussian densities, and verify the results (5.49) and (5.50) for the parameters $\mathbf{w}$ and w_0.

5.13 ($\star$) Consider a generative classification model for K classes defined by prior class probabilities $p(\mathcal{C}_k) = \pi_k$ and general class-conditional densities $p(\boldsymbol{\phi}|\mathcal{C}_k)$ where $\boldsymbol{\phi}$ is the input feature vector. Suppose we are given a training data set $\{\boldsymbol{\phi}_n, \mathbf{t}_n\}$ where $n = 1, \ldots, N$, and $\mathbf{t}_n$ is a binary target vector of length K that uses the 1-of-K coding scheme, so that it has components $t_{nj} = I_{jk}$ if data point n is from class $\mathcal{C}_k$. Assuming that the data points are drawn independently from this model, show that the maximum-likelihood solution for the prior probabilities is given by

$$\pi_k = \frac{N_k}{N} \tag{5.101}$$

where N_k is the number of data points assigned to class $\mathcal{C}_k$.

5.14 ($\star\star$) Consider the classification model of Exercise 5.13 and now suppose that the class-conditional densities are given by Gaussian distributions with a shared covariance matrix, so that

$$p(\boldsymbol{\phi}|\mathcal{C}_k) = \mathcal{N}(\boldsymbol{\phi}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}). \tag{5.102}$$

Telegram: @uni_k

www.konkur.in

Show that the maximum likelihood solution for the mean of the Gaussian distribution for class $\mathcal{C}_k$ is given by

$$\boldsymbol{\mu}_k = \frac{1}{N_k} \sum_{n=1}^{N} t_{nk} \boldsymbol{\phi}_n, \tag{5.103}$$

which represents the mean of those feature vectors assigned to class $\mathcal{C}_k$. Similarly, show that the maximum likelihood solution for the shared covariance matrix is given by

$$\boldsymbol{\Sigma} = \sum_{k=1}^{K} \frac{N_k}{N} \mathbf{S}_k \tag{5.104}$$

where

$$\mathbf{S}_k = \frac{1}{N_k} \sum_{n=1}^{N} t_{nk} (\boldsymbol{\phi}_n - \boldsymbol{\mu}_k)(\boldsymbol{\phi}_n - \boldsymbol{\mu}_k)^{\mathrm{T}}. \tag{5.105}$$

Thus, $\boldsymbol{\Sigma}$ is given by a weighted average of the covariances of the data associated with each class, in which the weighting coefficients are given by the prior probabilities of the classes.

5.15 ($\star\star$) Derive the maximum likelihood solution for the parameters $\{\mu_{ki}\}$ of the probabilistic naive Bayes classifier with discrete binary features described in Section 5.3.3.

5.16 ($\star\star$) Consider a classification problem with K classes for which the feature vector $\boldsymbol{\phi}$ has M components each of which can take L discrete states. Let the values of the components be represented by a 1-of-L binary coding scheme. Further suppose that, conditioned on the class $\mathcal{C}_k$, the M components of $\boldsymbol{\phi}$ are independent, so that the class-conditional density factorizes with respect to the feature vector components. Show that the quantities a_k given by (5.46), which appear in the argument to the softmax function describing the posterior class probabilities, are linear functions of the components of $\boldsymbol{\phi}$. Note that this represents an example of a naive Bayes model.

Section 11.2.3

5.17 ($\star\star$) Derive the maximum likelihood solution for the parameters of the probabilistic naive Bayes classifier described in Exercise 5.16.

5.18 ($\star$) Verify the relation (5.72) for the derivative of the logistic sigmoid function defined by (5.42).

5.19 ($\star$) By making use of the result (5.72) for the derivative of the logistic sigmoid, show that the derivative of the error function (5.74) for the logistic regression model is given by (5.75).

5.20 ($\star$) Show that for a linearly separable data set, the maximum likelihood solution for the logistic regression model is obtained by finding a vector $\mathbf{w}$ whose decision boundary $\mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}) = 0$ separates the classes and then taking the magnitude of $\mathbf{w}$ to infinity.

5.21 ($\star$) Show that the derivatives of the softmax activation function (5.76), where the a_k are defined by (5.77), are given by (5.78).

Telegram: @uni_k

www.konkur.in

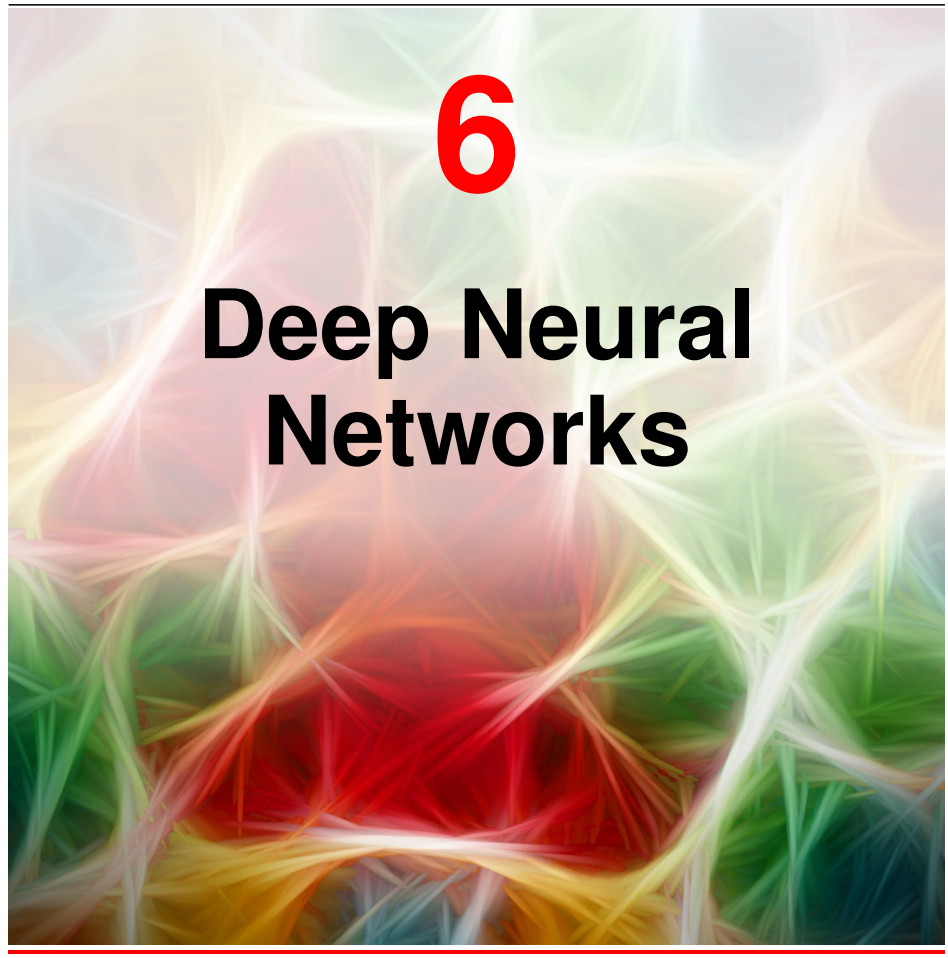
5.22 ($\star$) Using the result (5.78) for the derivatives of the softmax activation function, show that the gradients of the cross-entropy error (5.80) are given by (5.81).

5.23 ($\star$) Show that the probit function (5.86) and the erf function (5.87) are related by (5.88).

5.24 ($\star\star$) Suppose we wish to approximate the logistic sigmoid $\sigma(a)$ defined by (5.42) by a scaled probit function $\Phi(\lambda a)$, where $\Phi(a)$ is defined by (5.86). Show that if λ is chosen so that the derivatives of the two functions are equal at $a = 0$, then $\lambda^2 = \pi/8$.

Telegram: @uni_k

www.konkur.in



6 Deep Neural Networks

In recent years, neural networks have emerged as, by far, the most important machine learning technology for practical applications, and we therefore devote a large fraction of this book to studying them. Previous chapters have already laid many of the foundations we will need. In particular, we have seen that linear regression models that comprise linear combinations of fixed nonlinear basis functions can be expressed as neural networks having a single layer of weight and bias parameters. Likewise, classification models based on linear combinations of basis functions can also be viewed as single-layer neural networks. These allowed us to introduce several important concepts before we embark on a discussion of more complex multilayered networks in this chapter.

Chapter 4

Chapter 5

Given a sufficient number of suitably chosen basis functions, such linear models can approximate any given nonlinear transformation from inputs to outputs to any desired accuracy and might therefore appear to be sufficient to tackle any practical

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024
C. M. Bishop, H. Bishop, *Deep Learning*, https://doi.org/10.1007/978-3-031-45468-4_6

Telegram: @uni_k

www.konkur.in

application. However, these models have some severe limitations, and so we will begin our discussion of neural networks by exploring these limitations and understanding why it is necessary to use basis functions that are themselves learned from data. This leads naturally to a discussion of neural networks having more than one layer of learnable parameters. These are known as *feed-forward networks* or *multilayer perceptrons*. We will also discuss the benefits of having many such layers of processing, leading to the key concept of *deep neural networks* that now dominate the field of machine learning.

Section 6.3.6

6.1. Limitations of Fixed Basis Functions

Chapter 5

Linear basis function models for classification are based on linear combinations of basis functions $\phi_j(\mathbf{x})$ and take the form

$$y(\mathbf{x}, \mathbf{w}) = f\left(\sum_{j=1}^{M} w_j \phi_j(\mathbf{x}) + w_0\right) \tag{6.1}$$

where $f(\cdot)$ is a nonlinear output activation function. Linear models for regression take the same form but with $f(\cdot)$ replaced by the identity. These models allow for an arbitrary set of nonlinear basis functions $\{\phi_i(\mathbf{x})\}$, and because of the generality of these basis functions, such models can in principle provide a solution to any regression or classification problem. This is true in a trivial sense in that if one of the basis functions corresponds to the desired input-to-output transformation, then the learnable linear layer simply has to copy the value of this basis function to the output of the model.

Chapter 4

More generally, we would expect that a sufficiently large and rich set of basis functions would allow any desired function to be approximated to arbitrary accuracy. It would seem therefore that such linear models constitute a general purpose framework for solving problems in machine learning. Unfortunately, there are some significant shortcomings with linear models, which arise from the assumption that the basis functions $\phi_j(\mathbf{x})$ are fixed and independent of the training data. To understand these limitations, we start by looking at the behaviour of linear models as the number of input variables is increased.

6.1.1 The curse of dimensionality

Section 1.2

Consider a simple regression model for a single input variable given by a polynomial of order M in the form

$$y(x, \mathbf{w}) = w_0 + w_1 x + w_2 x^2 + \ldots + w_M x^M \tag{6.2}$$

and let us see what happens if we increase the number of inputs. If we have D input variables $\{x_1, \ldots, x_D\}$, then a general polynomial with coefficients up to order 3

Telegram: @uni_k

www.konkur.in

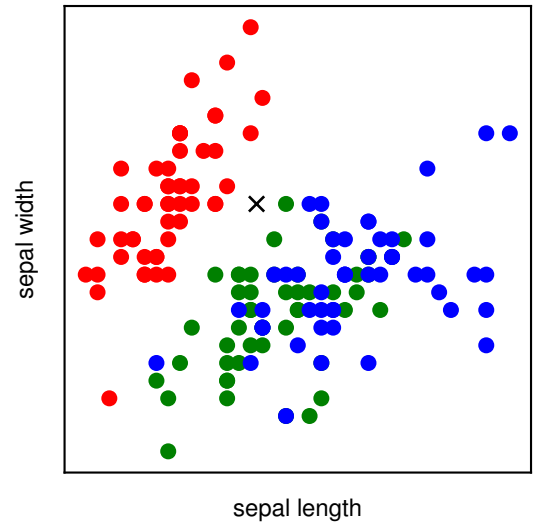


Figure 6.1 Plot of the Iris data in which red, green, and blue points denote three species of iris flower and the axes represent measurements of the length and width of the sepal, respectively. Our goal is to classify a new test point such as the one denoted by $\times$.

would take the form

$$y(\mathbf{x}, \mathbf{w}) = w_0 + \sum_{i=1}^{D} w_i x_i + \sum_{i=1}^{D}\sum_{j=1}^{D} w_{ij} x_i x_j + \sum_{i=1}^{D}\sum_{j=1}^{D}\sum_{k=1}^{D} w_{ijk} x_i x_j x_k. \quad (6.3)$$

As D increases, the growth in the number of independent coefficients is $\mathcal{O}(D^3)$, whereas for a polynomial of order M, the growth in the number of coefficients is $\mathcal{O}(D^M)$ (Bishop, 2006). We see that in spaces of higher dimensionality, polynomials can rapidly become unwieldy and of little practical utility.

The severe difficulties that can arise in spaces of many dimensions is sometimes called the *curse of dimensionality* (Bellman, 1961). It is not limited to polynomial regression but is in fact quite general. Consider the use of linear models for solving classification problems. Figure 6.1 shows a plot of data from the Iris data set comprising 50 observations taken from each of three species of iris flowers. Each observation has four variables representing measurements of the sepal length, sepal width, petal length, and petal width. For this illustration, we consider only the sepal length and sepal width variables. Given these 150 observations as training data, our goal is to classify a new test point, such as the one denoted by the cross in Figure 6.1, by assigning it to one of the three species. We observe that the cross is close to several red points, and so we might suppose that it belongs to the red class. However, there are also some green points nearby, so we might think that it could instead belong to the green class. It seems less likely that it belongs to the blue class. The intuition here is that the identity of the cross should be determined more strongly by nearby points from the training set and less strongly by more distant points, and this intuition turns out to be reasonable.

One very simple way of converting this intuition into a learning algorithm would be to divide the input space into regular cells, as indicated in Figure 6.2. When we are given a test point and we wish to predict its class, we first decide which cell it

Telegram: @uni_k

www.konkur.in

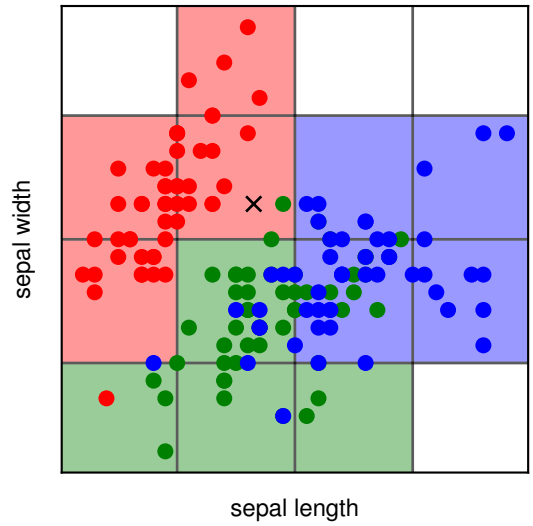


Figure 6.2 Illustration of a simple approach for solving classification problems in which the input space is divided into cells and any new test point is assigned to the class that has the most representatives in the same cell as the test point. As we shall see shortly, this simplistic approach has some severe shortcomings.

belongs to, and then we find all the training data points that fall in the same cell. The identity of the test point is predicted to be the same as the class having the largest number of training points in the same cell as the test point (with ties being broken at random). We can view this as a basis function model in which there is a basis function $\phi_i(\mathbf{x})$ for each grid cell, which simply returns zero if $\mathbf{x}$ lies outside the grid cell, and otherwise returns the majority class of the training data points that fall inside the cell. The output of the model is then given by the sum of the outputs of all the basis functions.

There are numerous problems with this naive approach, but one of the most severe becomes apparent when we consider its extension to problems having larger numbers of input variables, corresponding to input spaces of higher dimensionality. The origin of the problem is illustrated in Figure 6.3, which shows that, if we divide a region of a space into regular cells, then the number of such cells grows exponentially with the dimensionality of the space. The challenge with an exponentially large number of cells is that we would need an exponentially large quantity of training

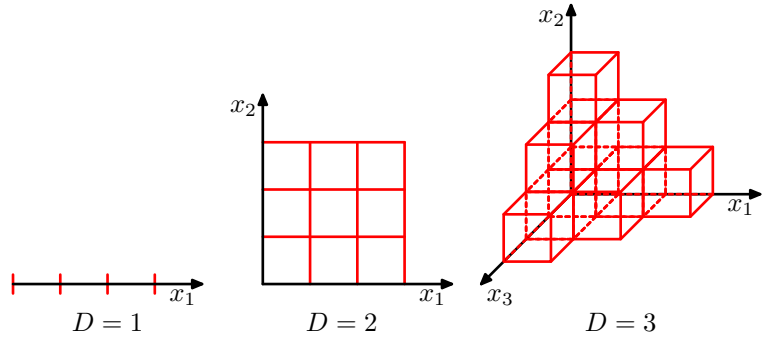


Figure 6.3 Illustration of the curse of dimensionality, showing how the number of regions of a regular grid grows exponentially with the dimensionality D of the space. For clarity, only a subset of the cubical regions are shown for $D = 3$.

Telegram: @uni_k

www.konkur.in

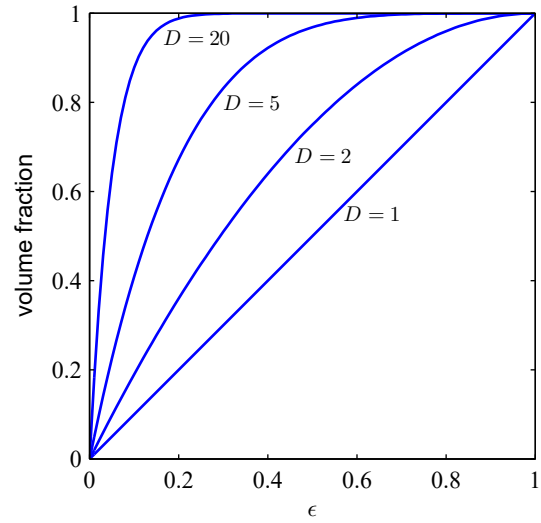


Figure 6.4 Plot of the fraction of the volume of a hypersphere of radius $r = 1$ lying in the range $r = 1 - \epsilon$ to $r = 1$ for various values of the dimensionality D.

data to ensure that the cells are not empty. We have already seen in Figure 6.2 that some cells contain no training points. Hence, a test point in such cells cannot be classified. Clearly, we have no hope of applying such a technique in a space of more than a few variables. The difficulties with both the polynomial regression example and the Iris data classification example arise because the basis functions were chosen independently of the problem being solved. We will need to be more sophisticated in our choice of basis functions if we are to circumvent the curse of dimensionality.

Section 6.1.4

6.1.2 High-dimensional spaces

First, however, we will look more closely at the properties of spaces with higher dimensionality where our geometrical intuitions, formed through a life spent in a space of three dimensions, can fail badly. As a simple example, consider a hypersphere of radius $r = 1$ in a space of D dimensions, and ask what is the fraction of the volume of the hypersphere that lies between radius $r = 1 - \epsilon$ and $r = 1$. We can evaluate this fraction by noting that the volume $V_D(r)$ of a hypersphere of radius r in D dimensions must scale as r^D, and so we write

$$V_D(r) = K_D r^D \tag{6.4}$$

Exercise 6.1

where the constant K_D depends only on D. Thus, the required fraction is given by

$$\frac{V_D(1) - V_D(1-\epsilon)}{V_D(1)} = 1 - (1-\epsilon)^D, \tag{6.5}$$

which is plotted as a function of ϵ for various values of D in Figure 6.4. We see that, for large D, this fraction tends to 1 even for small values of ϵ. Thus, we arrive at the remarkable result that, in spaces of high dimensionality, most of the volume of a hypersphere is concentrated in a thin shell near the surface!

Telegram: @uni_k

www.konkur.in

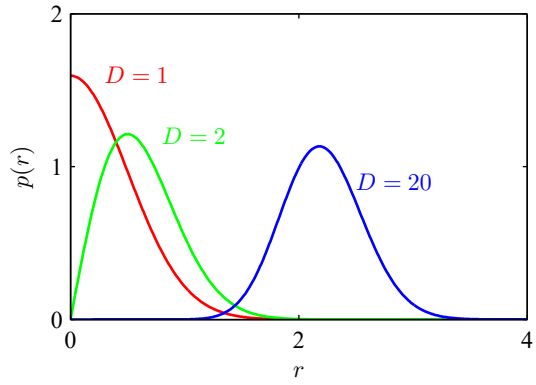


Figure 6.5 Plot of the probability density with respect to radius r of a Gaussian distribution for various values of the dimensionality D. In a high-dimensional space, most of the probability mass of a Gaussian is located within a thin shell at a specific radius.

As a further example of direct relevance to machine learning, consider the behaviour of a Gaussian distribution in a high-dimensional space. If we transform from Cartesian to polar coordinates and then integrate out the directional variables, we obtain an expression for the density $p(r)$ as a function of radius r from the origin. Thus, $p(r)\delta r$ is the probability mass inside a thin shell of thickness δr located at radius r. This distribution is plotted, for various values of D, in Figure 6.5, and we see that for large D, the probability mass of the Gaussian is concentrated in a thin shell at a specific radius.

Exercise 6.3

In this book, we make extensive use of illustrative examples involving one or two variables, because this makes it particularly easy to visualize these spaces graphically. The reader should be warned, however, that not all intuitions developed in spaces of low dimensionality will generalize to situations involving many dimensions.

Finally, although we have talked about the curse of dimensionality, there can also be advantages to working in high-dimensional spaces. Consider the situation shown in Figure 6.6. We see that this data set, in which each data point consists of a pair of values (x_1, x_2), is linearly separable, but when only the value of x_1 is observed, the classes have a strong overlap. The classification problem is therefore much easier in the higher-dimensional space.

6.1.3 Data manifolds

With both the polynomial regression model and the grid-based classifier in Figure 6.2, we saw that the number of basis functions grows rapidly with dimensionality, making such methods impractical for applications involving even a few dozen variables, never mind the millions of inputs that often arise with, say, image processing. The problem is that the basis functions are fixed ahead of time and do not depend on the data, or indeed even on the specific problem being solved. We need to find a way to create basis functions that are tuned to the particular application.

Although the curse of dimensionality certainly raises important issues for machine learning applications, it does not prevent us from finding effective techniques applicable to high-dimensional spaces. One reason for this is that real data will generally be confined to a region of the data space having lower effective dimen-

Telegram: @uni_k

www.konkur.in

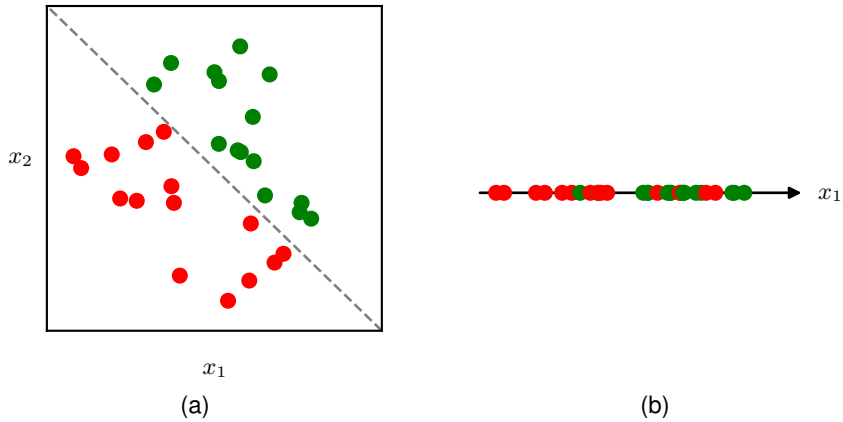


Figure 6.6 Illustration of a data set in two dimensions (x_1, x_2) in which data points from the two classes depicted using green and red circles can be separated by a linear decision surface, as seen in (a). If, however, only the variable x_1 is measured then the classes are no longer separable, as seen in (b).

sionality. Consider the images shown in Figure 6.7. Each image is a point in a high-dimensional space whose dimensionality is determined by the number of pixels. Because the objects can occur at different vertical and horizontal positions within the image and in different orientations, there are three degrees of freedom of variability between images, and a set of images will, to a first approximation, live on a three-dimensional *manifold* embedded within the high-dimensional space. Due to the complex relationships between the object position or orientation and the pixel intensities, this manifold will be highly nonlinear.

In fact, the number of pixels is really an artefact of the image generation process since they represent measurements of a continuous world. Capturing the same image at a higher resolution increases the dimensionality D of the data space without changing the fact that the images still live on a three-dimensional manifold. If we can associate localized basis functions with the data manifold, rather than with the entire high-dimensional data space, we might expect that the number of required basis functions would grow exponentially with the dimensionality of the manifold rather than with the dimensionality of the data space. Since the manifold will typically have a much lower dimensionality than the data space, this represents a huge

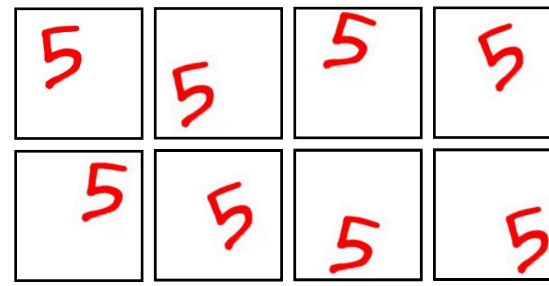


Figure 6.7 Examples of images of a handwritten digit that differ in the location of the digit within the images as well as in their orientation. This data lives on a nonlinear three-dimensional manifold within the high-dimensional image space.

Telegram: @uni_k

www.konkur.in

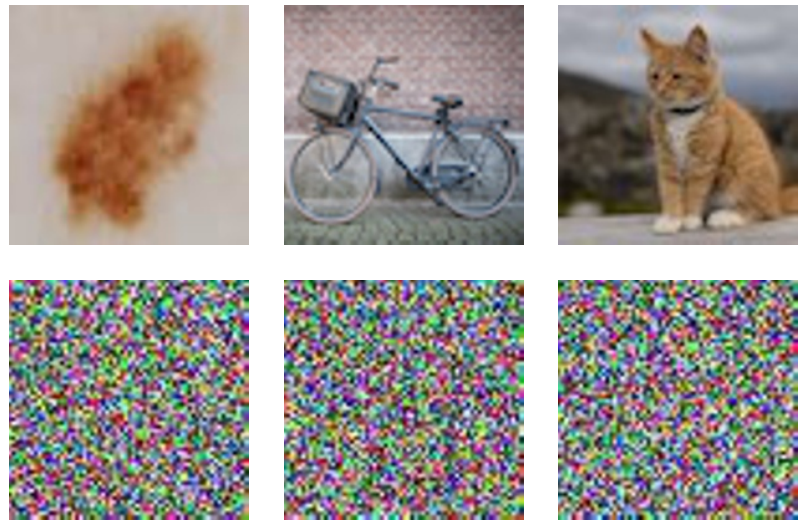


Figure 6.8 The top row shows examples of natural images of size 64×64 pixels, whereas the bottom row shows randomly generated images of the same size obtained by drawing pixel values from a uniform probability distribution over the possible pixel colours.

improvement. Effectively, neural networks learn a set of basis functions that are adapted to data manifolds. Moreover, for a particular application, not all directions within the manifold may be significant. For example, if we wish to determine only the orientation, and not the position, of the object in Figure 6.7, then there is only one relevant degree of freedom on the manifold and not three. Neural networks are also able to learn which directions on the manifold are relevant to predicting the desired outputs.

Another way to see that real data is confined to low-dimensional manifolds is to consider the task of generating random images. In Figure 6.8 we see examples of natural images along with examples of synthetic images of the same resolution generated by sampling each of the red, green, and blue intensities at each pixel independently at random from a uniform distribution. We see that none of the synthetic images look at all like natural images. The reason is that these random images lack the very strong correlations between pixels that natural images exhibit. For example, two adjacent pixels in a natural image have a much higher probability of having the same, or very similar, colour, than would two adjacent images in the random examples. Each of the images in Figure 6.8 corresponds to a point in a high-dimensional space, yet natural images cover only a tiny fraction of this space.

6.1.4 Data-dependent basis functions

We have seen that simple basis functions that are chosen independently of the problem being solved can run into significant limitations, particularly in spaces of high dimensionality. If we want to use basis functions in such situations, then one approach would be to use expert knowledge to hand-craft the basis functions in a

Telegram: @uni_k

www.konkur.in

way that is specific to each application. For many years, this was the mainstream approach in machine learning. Basis functions, often called *features*, would be determined by a combination of domain knowledge and trial-and-error. However, this approach met with limited success and was superseded by data-driven approaches in which basis functions are learned from the training data. Domain knowledge still plays a role in modern machine learning, but at a more qualitative level in designing network architectures where it can capture appropriate *inductive bias*, as we will see in later chapters.

Section 9.1

Since data in a high-dimensional space may be confined to a low-dimensional manifold, we do not need basis functions that densely fill the whole input space, but instead we can use basis functions that are themselves associated with the data manifold. One way to do this is to have one basis function associated with each data point in the training set, which ensures that the basis functions are automatically adapted to the underlying data manifold. An example of such a model is that of *radial basis functions* (Broomhead and Lowe, 1988), which have the property that each basis function depends only on the radial distance (typically Euclidean) from a central vector. If the basis centres are chosen to be the input data values $\{\mathbf{x}_n\}$ then there is one basis function $\phi_n(\mathbf{x})$ for each data point, which will therefore capture the whole of the data manifold. A typical choice for a radial basis function is

$$\phi_n(\mathbf{x}) = \exp\left(-\frac{\|\mathbf{x} - \mathbf{x}_n\|^2}{s^2}\right) \tag{6.6}$$

where s is a parameter controlling the width of the basis function. Although it can be quick to set up such a model, a major problem with this technique is that it becomes computationally unwieldy for large data sets. Moreover, the model needs careful regularization to avoid severe over-fitting.

A related approach, called a *support vector machine* or SVM (Vapnik, 1995; Schölkopf and Smola, 2002; Bishop, 2006), addresses this by again defining basis functions that are centred on each of the training data points and then selecting a subset of these automatically during training. As a result, the effective number of basis functions in the resulting models is generally much smaller than the number of training points, although it is often still relatively large and typically increases with the size of the training set. Support vector machines also do not produce probabilistic outputs, and they do not naturally generalize to more than two classes. Methods such as radial basis functions and support vector machines have been superseded by deep neural networks, which are much better at exploiting very large data sets efficiently. Moreover, as we will see later, neural networks are able to learn deep *hierarchical* representations, which are crucial to achieving high prediction accuracy in more complex applications.

Telegram: @uni_k

www.konkur.in

6.2. Multilayer Networks

In the previous section, we saw that to apply linear models of the form (6.1) to problems involving large-scale data sets and high-dimensional spaces, we need to find a set of basis functions that is tuned to the problem being solved. The key idea behind neural networks is to choose basis functions $\phi_j(\mathbf{x})$ that themselves have learnable parameters and then allow these parameters to be adjusted, along with the coefficients $\{w_j\}$, during training. We then optimize the whole model by minimizing an error function using gradient-based optimization methods, such as stochastic gradient descent, where the error function is defined jointly across all the parameters in the model.

Chapter 7

There are, of course, many ways to construct parametric nonlinear basis functions. One key requirement is that they must be differentiable functions of their learnable parameters so that we can apply gradient-based optimization. The most successful choice has been to use basis functions that follow the same form as (6.1), so that each basis function is itself a nonlinear function of a linear combination of the inputs, where the coefficients in the linear combination are learnable parameters. Note that this construction can clearly be extended recursively to give a hierarchical model with many layers, which forms the basis for deep neural networks.

Section 6.3

Consider a basic neural network model having two layers of learnable parameters. First, we construct M linear combinations of the input variables $x_1, \ldots, x_D$ in the form

$$a_j^{(1)} = \sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)} \tag{6.7}$$

where $j = 1, \ldots, M$, and the superscript (1) indicates that the corresponding parameters are in the first 'layer' of the network. We will refer to the parameters $w_{ji}^{(1)}$ as *weights* and the parameters $w_{j0}^{(1)}$ as *biases*, while the quantities $a_j^{(1)}$ are called *pre-activations*. Each of the quantities a_j is then transformed using a differentiable, nonlinear *activation function* $h(\cdot)$ to give

Chapter 4

$$z_j^{(1)} = h(a_j^{(1)}), \tag{6.8}$$

which represent the outputs of the basis functions in (6.1). In the context of neural networks, these basis functions are called *hidden units*. We will explore various choices for the nonlinear function $h(\cdot)$ shortly, but here we note that provided the derivative $h'(\cdot)$ can be evaluated, then the overall network function will be differentiable. Following (6.1), these values are again linearly combined to give

$$a_k^{(2)} = \sum_{j=1}^{M} w_{kj}^{(2)} z_j^{(1)} + w_{k0}^{(2)} \tag{6.9}$$

where $k = 1, \ldots, K$, and K is the total number of outputs. This transformation corresponds to the second layer of the network, and again the $w_{k0}^{(2)}$ are bias parameters. Finally, the $\{a_k^{(2)}\}$ are transformed using an appropriate output-unit activation

Telegram: @uni_k

www.konkur.in

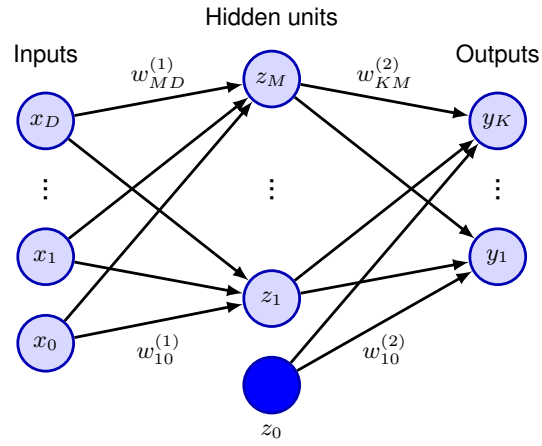


Figure 6.9 Network diagram for a two-layer neural network. The input, hidden, and output variables are represented by nodes, and the weight parameters are represented by links between the nodes. The bias parameters are denoted by links coming from additional input and hidden variables x_0 and z_0 which are themselves denoted by solid nodes. Arrows denote the direction of information flow through the network during forward propagation.

function $f(\cdot)$ to give a set of network outputs y_k. A two-layer neural network can be represented in diagram form as shown in Figure 6.9.

6.2.1 Parameter matrices

Section 4.1.1

As we discussed in the context of linear regression models, the bias parameters in (6.7) can be absorbed into the set of weight parameters by defining an additional input variable x_0 whose value is clamped at $x_0 = 1$, so that (6.7) takes the form

$$a_j = \sum_{i=0}^{D} w_{ji}^{(1)} x_i. \tag{6.10}$$

We can similarly absorb the second-layer biases into the second-layer weights, so that the overall network function becomes

$$y_k(\mathbf{x}, \mathbf{w}) = f\left(\sum_{j=0}^{M} w_{kj}^{(2)} h\left(\sum_{i=0}^{D} w_{ji}^{(1)} x_i \right) \right). \tag{6.11}$$

Another notation that will prove convenient at various points in the book is to represent the inputs as a column vector $\mathbf{x} = (x_1, \ldots, x_N)^{\mathrm{T}}$ and then to gather the weight and bias parameters in (6.11) into matrices to give

$$\mathbf{y}(\mathbf{x}, \mathbf{w}) = f\left(\mathbf{W}^{(2)} h\left(\mathbf{W}^{(1)} \mathbf{x} \right) \right) \tag{6.12}$$

where $f(\cdot)$ and $h(\cdot)$ are evaluated on each vector element separately.

6.2.2 Universal approximation

The capability of a two-layer network to model a broad range of functions is illustrated in Figure 6.10. This figure also shows how individual hidden units work collaboratively to approximate the final function. The role of hidden units in a simple classification problem is illustrated in Figure 6.11.

Telegram: @uni_k

www.konkur.in

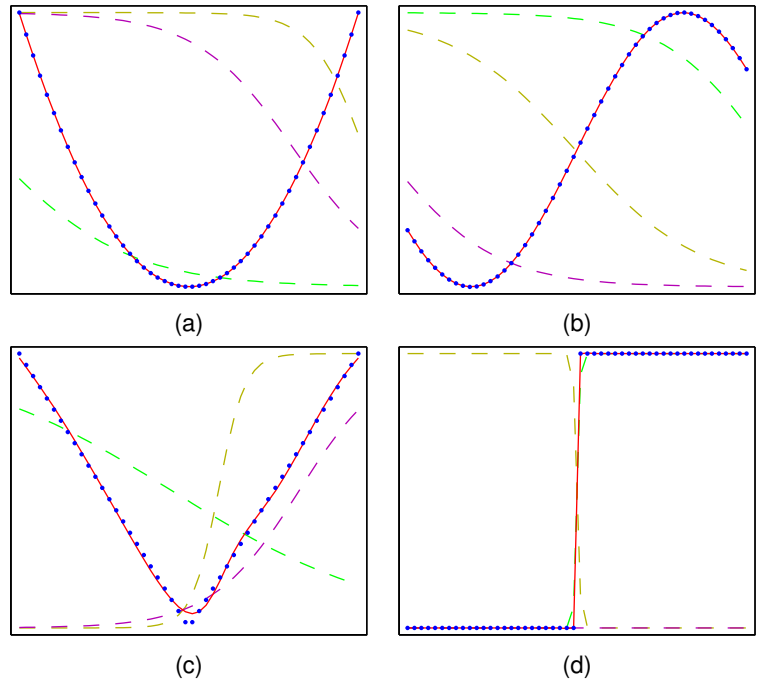


Figure 6.10 Illustration of the capability of a two-layer neural network to approximate four different functions: (a) $f(x) = x^2$, (b) $f(x) = \sin(x)$, (c), $f(x) = |x|$, and (d) $f(x) = H(x)$ where $H(x)$ is the Heaviside step function. In each case, $N = 50$ data points, shown as blue dots, have been sampled uniformly in x over the interval $(-1, 1)$ and the corresponding values of $f(x)$ evaluated. These data points are then used to train a two-layer network having three hidden units with tanh activation functions and linear output units. The resulting network functions are shown by the red curves, and the outputs of the three hidden units are shown by the three dashed curves.

The approximation properties of two-layer feed-forward networks were widely studied in the 1980s, with various theorems showing that, for a wide range of activation functions, such networks can approximate any function defined over a continuous subset of $\mathbb{R}^D$ to arbitrary accuracy (Funahashi, 1989; Cybenko, 1989; Hornik, Stinchcombe, and White, 1989; Leshno *et al.*, 1993). A similar result holds for functions from any finite-dimensional discrete space to any another. Neural networks are therefore said to be *universal approximators*.

Although such theorems are reassuring, they tell us only that there exists a network that can represent the required function. In some cases, they may require networks that have an exponentially large number of hidden units. Moreover, they say nothing about whether such a network can be found by a learning algorithm. Furthermore, we will see later that the *no free lunch theorem* says that we can never find a truly universal machine learning algorithm. Finally, although networks having two layers of weights are universal approximators, in a practical application, there can be huge benefits in considering networks having many more than two layers that can learn hierarchical internal representations. All these points support the drive towards deep learning.

Section 9.1.2

6.2.3 Hidden unit activation functions

We have seen that the activation functions for the output units are determined by the kind of distribution being modelled. For the hidden units, however, the only requirement is that they need to be differentiable, which leaves a wide range of pos-

Telegram: @uni_k

www.konkur.in

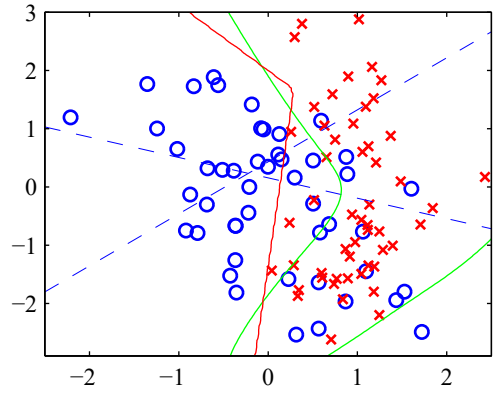


Figure 6.11 Example of the solution of a simple two-class classification problem involving synthetic data using a neural network having two inputs, two hidden units with tanh activation functions, and a single output having a logistic-sigmoid activation function. The dashed blue lines show the $z = 0.5$ contours for each of the hidden units, and the red line shows the $y = 0.5$ decision surface for the network. For comparison, the green lines denote the optimal decision boundary computed from the distributions used to generate the data.

sibilities. In most cases, all the hidden units in a network will be given the same activation function, although in principle there is no reason why different choices could not be applied in different parts of the network.

The simplest option for a hidden unit activation function is the identity function, which means that all the hidden units become linear. However, for any such network, we can always find an equivalent network without hidden units. This follows from the fact that the composition of successive linear transformations is itself a linear transformation, and so its representational capability is no greater than that of a single linear layer. However, if the number of hidden units is smaller than either the number of input or output units, then the transformations that such a network can generate are not the most general possible linear transformation from inputs to outputs because information is lost in the dimensionality reduction at the hidden units. Consider a network with N inputs, M hidden units, and K outputs, and where all activation functions are linear. Such a network has $M(N + K)$ parameters, whereas a linear transformation of inputs directly to outputs would have NK parameters. If M is small relative to N or K, or both, this leads to a two-layer linear network having fewer parameters than the direct linear mapping, corresponding to a rank-deficient transformation. Such 'bottleneck' networks of linear units corresponds to a standard data analysis technique called *principal component analysis*. In general, however, there is limited interest in using multilayer networks of linear units since the overall function computed by such a network is still linear.

Chapter 16

A simple, nonlinear differentiable function is the logistic sigmoid given by

$$\sigma(a) = \frac{1}{1 + \exp(-a)}, \tag{6.13}$$

which is plotted in Figure 5.12. This was widely used in the early years of work on multilayer neural networks and was partly inspired by studies of the properties of

Telegram: @uni_k

www.konkur.in

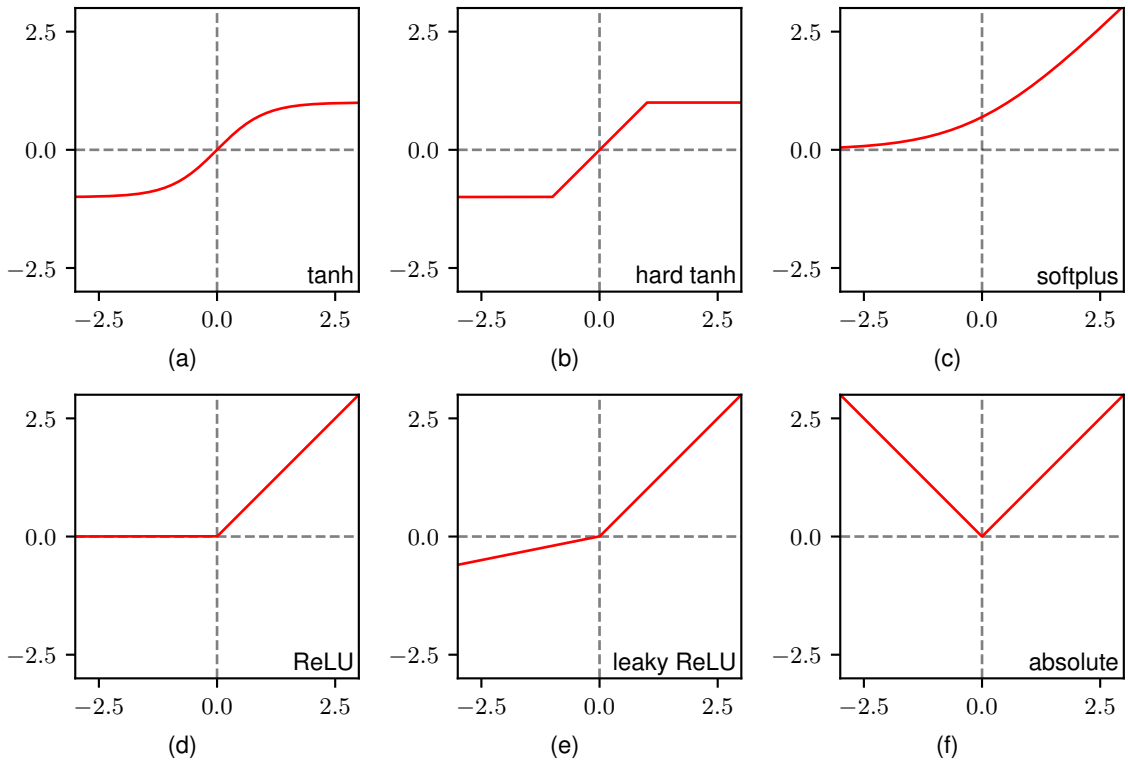


Figure 6.12 A variety of nonlinear activation functions.

biological neurons. A closely related function is tanh, which is defined by

$$\tanh(a) = \frac{e^a - e^{-a}}{e^a + e^{-a}}, \tag{6.14}$$

which is plotted in Figure 6.12(a). This function differs from the logistic sigmoid by a linear transformation of its input and its output values, and so for any network with logistic-sigmoid hidden-unit activation functions, there is an equivalent network with tanh activation functions. However, when training a network, these are not necessarily equivalent because for gradient-based optimization, the network weights and biases need to be initialized, and so if the activation functions are changed, then the initialization scheme must be adjusted accordingly. A 'hard' version of the tanh function (Collobert, 2004) is given by

Exercise 6.4

$$h(a) = \max(-1, \min(1, a)) \tag{6.15}$$

and is plotted in Figure 6.12(b).

A major drawback of both the logistic sigmoid and the tanh activation functions is that the gradients go to zero exponentially when the inputs have either large positive or large negative values. We will discuss this 'vanishing gradients' issue later,

Section 7.4.2

Telegram: @uni_k

www.konkur.in

but for the moment, we note that it will generally be better to use activation functions with non-zero gradients, at least when the input takes a large positive value. One such choice is the *softplus activation function* given by

Exercise 6.7

$$h(a) = \ln\left(1 + \exp(a)\right), \tag{6.16}$$

which is plotted in Figure 6.12(c). For $a \gg 1$, we have $h(a) \simeq a$, and so the gradient remains non-zero even when the input to the activation function is large and positive, thereby helping to alleviate the vanishing gradients problem.

An even simpler choice of activation function is the *rectified linear unit* or *ReLU*, which is defined by

$$h(a) = \max(0, a) \tag{6.17}$$

and which is plotted in Figure 6.12(d). Empirically, this is one of the best-performing activation functions, and it is in widespread use. Note that strictly speaking, the derivative of the ReLU function is not defined when $a = 0$, but in practice this can be safely ignored. The softplus function (6.16) can be viewed as a smoothed version of the ReLU and is therefore also sometimes called *soft ReLU*.

Exercise 6.5

Although the ReLU has a non-zero gradient for positive input values, this is not the case for negative inputs, which can mean that some hidden units receive no 'error signal' during training. A modification of ReLU that seeks to avoid this issue is called a *leaky ReLU* and is defined by

$$h(a) = \max(0, a) + \alpha \min(0, a), \tag{6.18}$$

where $0 < \alpha < 1$. This function is plotted in Figure 6.12(e). Unlike ReLU, this has a nonzero gradient for input values $a < 0$, which ensures that there is a signal to drive training. A variant of this activation function uses $\alpha = -1$, in which case $h(a) = |a|$, which is plotted in Figure 6.12(f). Another variant allows each hidden unit to have its own value α_j, which can be learned during network training by evaluating gradients with respect to the $\{\alpha_j\}$ along with the gradients with respect to the weights and biases.

The introduction of ReLU gave a big improvement in training efficiency over previous sigmoidal activation functions (Krizhevsky, Sutskever, and Hinton, 2012). As well as allowing deeper networks to be trained efficiently, it is much less sensitive to the random initialization of the weights. It is also well suited to a low-precision implementation, such as 8-bit fixed versus 64-bit floating point, and it is computationally cheap to evaluate. Many practical applications simply use ReLU units as the default unless the goal is explicitly to explore the effects of different choices of activation function.

6.2.4 Weight-space symmetries

One property of feed-forward networks is that multiple distinct choices for the weight vector $\mathbf{w}$ can all give rise to the same mapping function from inputs to outputs (Chen, Lu, and Hecht-Nielsen, 1993). Consider a two-layer network of the form shown in Figure 6.9 with M hidden units having tanh activation functions and full connectivity in both layers. If we change the sign of all the weights and the bias

Telegram: @uni_k

www.konkur.in

feeding into a particular hidden unit, then, for a given input data point, the sign of the pre-activation of the hidden unit will be reversed, and therefore so too will the activation, because tanh is an odd function, so that $\tanh(-a) = -\tanh(a)$. This transformation can be exactly compensated for by changing the sign of all the weights leading out of that hidden unit. Thus, by changing the signs of a particular group of weights (and a bias), the input–output mapping function represented by the network is unchanged, and so we have found two different weight vectors that give rise to the same mapping function. For M hidden units, there will be M such 'sign-flip' symmetries, and thus, any given weight vector will be one of a set 2^M equivalent weight vectors .

Similarly, imagine that we interchange the values of all of the weights (and the bias) leading both into and out of a particular hidden unit with the corresponding values of the weights (and bias) associated with a different hidden unit. Again, this clearly leaves the network input–output mapping function unchanged, but it corresponds to a different choice of weight vector. For M hidden units, any given weight vector will belong to a set of $M \times (M-1) \times \cdots \times 2 \times 1 = M!$ equivalent weight vectors associated with this interchange symmetry, corresponding to the $M!$ different orderings of the hidden units. The network will therefore have an overall weight-space symmetry factor of $M!\,2^M$. For networks with more than two layers of weights, the total level of symmetry will be given by the product of such factors, one for each layer of hidden units.

It turns out that these factors account for all the symmetries in weight space (except for possible accidental symmetries due to specific choices for the weight values). Furthermore, the existence of these symmetries is not a particular property of the tanh function but applies to a wide range of activation functions (Kurková and Kainen, 1994). In general, these symmetries in weight space are of little practical consequence, since network training aims to find a specific setting for the parameters, and the existence of other, equivalent, settings is of little consequence. However, weight-space symmetries do play a role when Bayesian methods are used to evaluate the probability distribution over networks of different sizes (Bishop, 2006).

6.3. Deep Networks

We have motivated the development of neural networks by making the basis functions of a linear regression or classification model themselves be governed by learnable parameters, giving rise to the two-layer network model shown in Figure 6.9. For many years, this was the most widely used architecture, primarily because it proved difficult to train networks with more than two layers effectively. However, extending neural networks to have more than two layers, known as deep neural networks, brings many advantages as we will discuss shortly, and recent advances in techniques for training neural networks are effective for networks with many layers.

Chapter 7

We can easily extend the two-layer network architecture (6.12) to any finite number L of layers, in which layer $l = 1, \ldots, L$ computes the following function:

$$\mathbf{z}^{(l)} = h^{(l)}\left(\mathbf{W}^{(l)}\mathbf{z}^{(l-1)}\right) \tag{6.19}$$

Telegram: @uni_k

www.konkur.in

where $h^{(l)}$ denotes the activation function associated with layer l, and $\mathbf{W}^{(l)}$ denotes the corresponding matrix of weight and bias parameters. Also, $\mathbf{z}^{(0)} = \mathbf{x}$ represents the input vector and $\mathbf{z}^{(L)} = \mathbf{y}$ represents the output vector.

Note that there has been some confusion in the literature regarding the terminology for counting the number of layers in such networks. Thus, the network in Figure 6.9 is sometimes described as a three-layer network (which counts the number of layers of units and treats the inputs as units) or sometimes as a single-hidden-layer network (which counts the number of layers of hidden units). We recommend a terminology in which Figure 6.9 is called a two-layer network, because it is the number of layers of learnable weights that is important for determining the network properties.

We have seen that a network of the form shown in Figure 6.9, having two layers of learnable parameters, has universal approximation capabilities. However, networks with more than two layers can sometimes represent a given function with far fewer parameters than a two-layer network. Montúfar *et al.* (2014) show that the network function divides the input space into a number of regions that is exponential in the depth of the network, but which is only polynomial in the width of the hidden layers. To represent the same function using a two-layer network would require an exponential number of hidden units.

6.3.1 Hierarchical representations

Although this is an interesting result, a more compelling reason to explore deep neural networks is that the network architecture encodes a particular form of inductive bias, namely that the outputs are related to the input space through a hierarchical representation. A good example is the task of recognizing objects in images. The relationship between the pixels of an image and a high-level concept such as 'cat' is highly complex and nonlinear, and would be an extremely challenging problem for a two-layer network. However, a deep neural network can learn to detect low-level features, such as edges, in the early layers, and can then combine these in subsequent layers to make higher-level features such as eyes or whiskers, which in turn can be combined in later layers to detect the presence of a cat. This can be viewed as a *compositional* inductive bias, in which higher-level objects, such as a cat, are composed of lower-level objects, such as eyes, which in turn have yet lower-level elements such as edges. We can also think of this in reverse by considering the process of generating an image starting with low-level features such as edges, then combining these to form simple shapes such as circles, and then combining those in turn to form higher-level objects such as cats. At each stage there are many ways to combine different components, giving an exponential gain in the number of possibilities with increasing depth.

Chapter 10

6.3.2 Distributed representations

Neural networks can take advantage of another form of compositionality called a *distributed representation*. Conceptually, each unit in a hidden layer can be thought of as representing a 'feature' at that level of the network, with a high value of the

Telegram: @uni_k

www.konkur.in

activation indicating that the corresponding feature is present and a low value indicating its absence. With M units in a given layer, such a network can represent M different features. However, the network could potentially learn a different representation in which *combinations* of hidden units represent features, thereby potentially allowing a hidden layer with M units to represent 2^M different features, growing exponentially with the number of units. Consider, for example, a network designed to process images of faces. Each particular face image may or may not have glasses, it may or may not have a hat, and it may or may not have a beard, leading to eight different combinations. Although this could be represented by eight units each of which 'turns on' when it detects the corresponding combination, it could also be represented more compactly by just three units, one for each attribute. These can be present independently of each other (although statistically their presence is likely to be correlated to some degree). Later, we will explore in detail the kinds of internal representations that deep learning networks discover for themselves during training.

Chapter 10

6.3.3 Representation learning

We can view the successive layers of a deep neural network as performing transformations of the data, that make it easier to solve the desired task or tasks. For example, a neural network that successfully learns to classify skin lesions as benign or malignant must have learned to transform the original image data into a new space, represented by the outputs of the final layer of hidden units, such that the final layer of the network can distinguish the two classes. This final layer can be viewed as a simple linear classifier, and so in the representation of the last hidden layer, the two classes must be well separated by a linear surface. This ability to discover a nonlinear transformation of the data that makes subsequent tasks easier to solve is called *representation learning* (Bengio, Courville, and Vincent, 2012). The learned representation, sometimes called the *embedding space*, is given by the outputs of one of the hidden layers of the network, so that any input vector, either from the training set or from some new data set, can be transformed into this representation by forward propagation through the network.

Section 1.1.1

Representation learning is especially powerful because it allows us to exploit unlabelled data. Often it is easy to collect a large quantity of unlabelled data, but acquiring the associated labels may be more difficult. For example, a video camera on a vehicle can gather large numbers of images of urban scenes as the vehicle is driven around a city, but taking those images and identifying relevant objects, such as pedestrians and road signs, would require expensive and time-consuming human labelling.

Learning from unlabelled data is called *unsupervised learning*, and many different algorithms have been developed to do this. For example, a neural network can be trained to take images as input and to create the same images as the output. To make this into a non-trivial task, the network may use hidden layers with fewer units than the number of pixels in the image, thereby forcing the network to learn some kind of compression of the images. Only unlabelled data is needed because each image in the training set acts as both the input vector and the target vector. Such networks are known as *autoencoders*. The goal is that this type of training will force the network

Section 19.1

Telegram: @uni_k

www.konkur.in

to discover some internal representation for the data that is useful for solving other tasks, such as image classification.

Historically, unsupervised learning played an important role in enabling the first deep networks (apart from convolutional networks) to be successfully trained. Each layer of the network was first pre-trained using unsupervised learning and then the entire network was trained further using gradient-based supervised training. It was later discovered that the pre-training phase could be omitted and a deep network could be trained from scratch purely using supervised learning given appropriate conditions.

However, pre-training and representation learning remain central to deep learning in other contexts. The most notable example of pre-training is in natural language processing in which transformer models are trained on large quantities of text and are able to learn highly sophisticated internal representations of language that facilitates an impressive range of capabilities at human level and beyond.

Chapter 12

6.3.4 Transfer learning

The internal representation learned for one particular task might also be useful for related tasks. For example, a network trained on a large labelled data set of everyday objects can learn how to transform an image representation into one that is much better suited for classifying objects. Then, the final classification layer of the network can be retrained using a smaller labelled data set of skin lesion images to create a lesion classifier. This is an example of *transfer learning* (Hospedales *et al.*, 2021), which allows higher accuracy to be achieved than if only the lesion image data were used for training, because the network can exploit commonalities shared by natural images in general. Transfer learning is illustrated in Figure 6.13.

Section 1.1.1

In general, transfer learning can be used to improve performance on some task A, for which training data is in short supply, by using data from a related task B, for which data is more plentiful. The two tasks should have the same kind of inputs, and there should be some commonality between the tasks so that low-level features, or internal representations, learned from task B will be useful for task A. When we look at convolutional networks we will see that many image processing tasks require similar low-level features corresponding to the early layers of a deep neural network, whereas later layers are more specialized to a particular task, making such networks well suited to transfer learning applications.

Chapter 10

When data for task A is very scarce, we might simply re-train the final layer of the network. In contrast, if there are more data points, it is feasible to retrain several layers. The process of learning parameters using one task that are then applied to one or more other tasks is called *pre-training*. Note that for the new task, instead of applying stochastic gradient descent to the whole network, it is much more efficient to send the new training data once through the fixed pre-trained network so as to evaluate the training inputs in the new representation. Iterative gradient-based optimization can then be applied just to the smaller network consisting of the final layers. As well as using a pre-trained network as a fixed pre-processor for a different task, it is also possible to apply *fine-tuning* in which the whole network is adapted to the data for task A. This is generally done with a very small learning rate for a lim-

Telegram: @uni_k

www.konkur.in

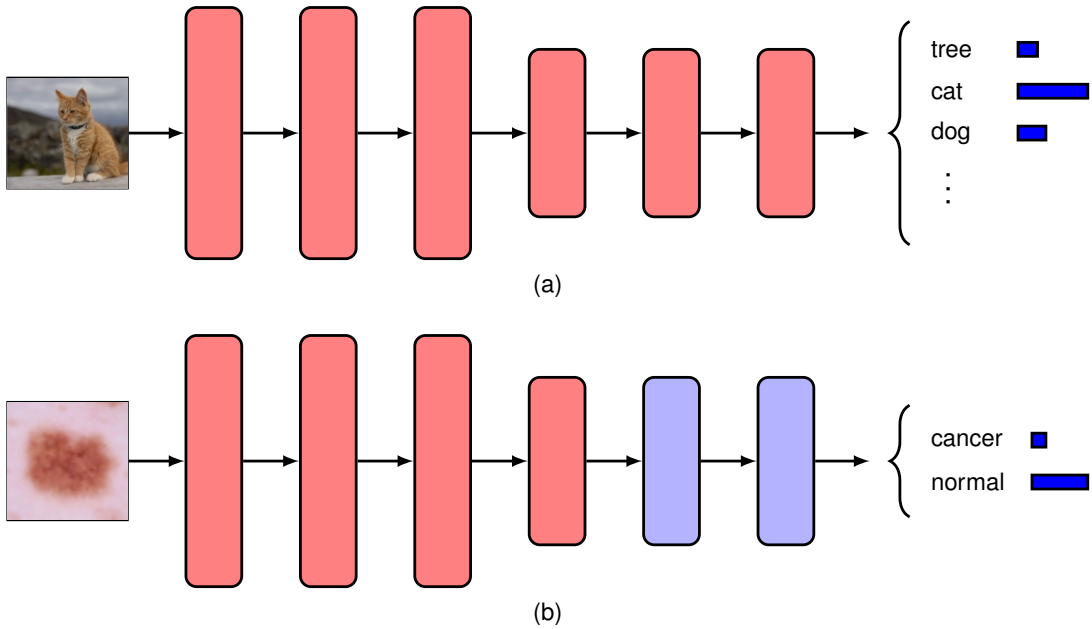


Figure 6.13 Schematic illustration of transfer learning. (a) A network is first trained on a task with abundant data, such as object classification of natural images. (b) The early layers of the network (shown in red) are copied from the first task and the final few layers of the network (shown in blue) are then retrained on a new task such as skin lesion classification for which training data is more scarce.

ited number of iterations to ensure that the network does not over-fit to the relatively small data set available for the new task.

A related approach is *multitask learning* (Caruana, 1997) in which a network jointly learns more than one related task at the same time. For example, we might wish to construct a spam email filter that allows different users to have different classifiers tuned to their particular preferences. The training data may comprise examples of spam email and non-spam email for many different users, but the number of examples for any one user may be quite limited, and therefore training a separate classifier for each user would give poor results. Instead, we can combine the data sets to train a single larger network that might, for example, share early layers but have separate learnable parameters for the different users in later layers. Sharing data across tasks allows the network to exploit commonalities amongst the tasks, thereby improving the accuracy for all users. With a large number of training examples, a deeper network with more parameters can be used, again leading to improved performance.

Learning across multiple tasks can be extended to *meta-learning*, which is also called *learning to learn*. Whereas multitask learning aims to make predictions for a fixed set of tasks, the aim of meta-learning is to make predictions for future tasks that were not seen during training. This can be done by not only learning a shared

Telegram: @uni_k

www.konkur.in

internal representation across tasks but also by learning the learning algorithm itself (Hospedales *et al.*, 2021). Meta-learning can be used to facilitate generalization of, for example, a classification model to new classes when there are very few labelled examples of the new classes. This is referred to as *few-shot learning*. When only a single labelled example is used it is called *one-shot learning*.

6.3.5 Contrastive learning

One of the most common and powerful representation learning methods is *contrastive learning* (Gutmann and Hyvärinen, 2010; Oord, Li, and Vinyals, 2018; Chen, Kornblith, *et al.*, 2020). The idea is to learn a representation such that certain pairs of inputs, referred to as positive pairs, are close in the embedding space, and other pairs of inputs, called negative pairs, are far apart. The intuition is that if we choose our positive pairs in such a way that they are semantically similar and choose negative pairs that are semantically dissimilar, then we will learn a representation space in which similar inputs are close, making downstream tasks, such as classification, much easier. As with other forms of representation learning, the outputs of the trained network are typically not used directly, and instead the activations at some earlier layer are used to form the embedding space. Contrastive learning is unlike most other machine learning tasks, in that the error function for a given input is defined only with respect to other inputs, instead of having a per-input label or target output.

Suppose we have a given data point $\mathbf{x}$ called the *anchor*, for which we have specified another data point $\mathbf{x}^+$ that together with $\mathbf{x}$ makes up a positive pair. We must also specify a set of data points $\{\mathbf{x}_1^-, \ldots, \mathbf{x}_N^-\}$ each of which makes up a negative pair with $\mathbf{x}$. We now need a loss function that will reward close proximity between the representations of $\mathbf{x}$ and $\mathbf{x}^+$ while encouraging a large distance between each pair $\{\mathbf{x}, \mathbf{x}_n^-\}$. One example of such a function, and the most commonly used loss function for contrastive learning, is called the *InfoNCE* loss (Gutmann and Hyvärinen, 2010; Oord, Li, and Vinyals, 2018), where NCE denotes 'noise contrastive estimation'. Suppose we have a neural network function $\mathbf{f}_\mathbf{w}(\mathbf{x})$ that maps points from the input space $\mathbf{x}$ to a representation space, governed by learnable parameters $\mathbf{w}$. This representation is normalized so that $||\mathbf{f}_\mathbf{w}(\mathbf{x})|| = 1$. Then, for a data point $\mathbf{x}$, the InfoNCE loss is defined by

$$E(\mathbf{w}) = -\ln \frac{\exp\{\mathbf{f}_\mathbf{w}(\mathbf{x})^\mathrm{T}\mathbf{f}_\mathbf{w}(\mathbf{x}^+)\}}{\exp\{\mathbf{f}_\mathbf{w}(\mathbf{x})^\mathrm{T}\mathbf{f}_\mathbf{w}(\mathbf{x}^+)\} + \sum_{n=1}^{N}\exp\{\mathbf{f}_\mathbf{w}(\mathbf{x})^\mathrm{T}\mathbf{f}_\mathbf{w}(\mathbf{x}_n^-)\}}. \tag{6.20}$$

We can see that in this function, the cosine similarity $\mathbf{f}_\mathbf{w}(\mathbf{x})^\mathrm{T}\mathbf{f}_\mathbf{w}(\mathbf{x}^+)$ between the representation $\mathbf{f}_\mathbf{w}(\mathbf{x})$ of the anchor and the representation $\mathbf{f}_\mathbf{w}(\mathbf{x}^+)$ of the positive example provides our measure of how close the positive pair examples are in the learned space, and the same measure is used to assess how close the anchor is to the negative examples. Note that the function resembles a classification cross-entropy error function in which the cosine similarity of the positive pair gives the logit for the label class and the cosine similarities for the negative pairs give the logits for the incorrect classes. Also note that the negative pairs are crucial as without them the

Telegram: @uni_k

www.konkur.in

embedding would simply learn the degenerate solution of mapping every point to the same representation.

A particular contrastive learning algorithm is defined predominantly by how the positive and negative pairs are chosen, which is how we use our prior knowledge to specify what a good representation should be. For example, consider the problem of learning representations of images. Here, a common choice is to create positive pairs by corrupting the input images in ways that should preserve the semantic information of the image while greatly altering the image in the pixel space (Wu *et al.*, 2018; He *et al.*, 2019; Chen, Kornblith, *et al.*, 2020). Corruptions are closely related to *data augmentations*, and examples include rotation, translation, and colour shifts. Other images from the data set can then be used to create the negative pairs. This approach to contrastive learning is known as *instance discrimination*.

Section 9.1.3

If, however, we have access to class labels, then we can use images of the same class as positive pairs and images of different classes as negative pairs. This relaxes the reliance on specifying the augmentations that the representation should be invariant to and also avoids treating two semantically similar images as a negative pair. This is referred to as supervised contrastive learning (Khosla *et al.*, 2020) because of the reliance on the class labels, and it can often yield better results than simply learning the representation using cross-entropy classification.

The members of positive and negative pairs do not necessarily have to come from the same data modality. In contrastive-language image pretraining, or CLIP (Radford *et al.*, 2021), a positive pair consists of an image and its corresponding text caption, and two separate functions, one for each modality, are used to map the inputs to the same representation space. Negative pairs are then mismatched images and captions. This is often referred to as *weakly supervised*, as it relies on captioned images, which are often easier to obtain by scraping data from the internet than by manually labelling images with their classes. The loss function in this case is given by

$$E(\mathbf{w}) = -\frac{1}{2}\ln\frac{\exp\{\mathbf{f}_{\mathbf{w}}(\mathbf{x}^+)^{\mathrm{T}}\mathbf{g}_{\boldsymbol{\theta}}(\mathbf{y}^+)\}}{\exp\{\mathbf{f}_{\mathbf{w}}(\mathbf{x}^+)^{\mathrm{T}}\mathbf{g}_{\boldsymbol{\theta}}(\mathbf{y}^+)\} + \sum_{n=1}^{N}\exp\{\mathbf{f}_{\mathbf{w}}(\mathbf{x}_n^-)^{\mathrm{T}}\mathbf{g}_{\boldsymbol{\theta}}(\mathbf{y}^+)\}} - \frac{1}{2}\ln\frac{\exp\{\mathbf{f}_{\mathbf{w}}(\mathbf{x}^+)^{\mathrm{T}}\mathbf{g}_{\boldsymbol{\theta}}(\mathbf{y}^+)\}}{\exp\{\mathbf{f}_{\mathbf{w}}(\mathbf{x}^+)^{\mathrm{T}}\mathbf{g}_{\boldsymbol{\theta}}(\mathbf{y}^+)\} + \sum_{m=1}^{M}\exp\{\mathbf{f}_{\mathbf{w}}(\mathbf{x}^+)^{\mathrm{T}}\mathbf{g}_{\boldsymbol{\theta}}(\mathbf{y}_m^-)\}} \tag{6.21}$$

where $\mathbf{x}^+$ and $\mathbf{y}^+$ represent a positive pair in which $\mathbf{x}$ is an image and $\mathbf{y}$ is its corresponding text caption, $\mathbf{f}_{\mathbf{w}}$ represents the mapping from images to the representation space, and $\mathbf{g}_{\boldsymbol{\theta}}$ is the mapping from text input to the representation space. We also require a set $\{\mathbf{x}_1^-, \ldots, \mathbf{x}_N^-\}$ of other images from the data set, for which we can assume the text caption $\mathbf{y}^+$ is inappropriate, and a set $\{\mathbf{y}_1^-, \ldots, \mathbf{y}_M^-\}$ of text captions that are similarly mismatched to the input image $\mathbf{x}$. The two terms in the loss function ensure that (a) the representation of the image is close to its text caption representation relative to other image representations and (b) the text caption representation is close to the representation of the image it describes relative to other representations of text captions. Although CLIP uses text and image pairs, any data

Telegram: @uni_k

www.konkur.in

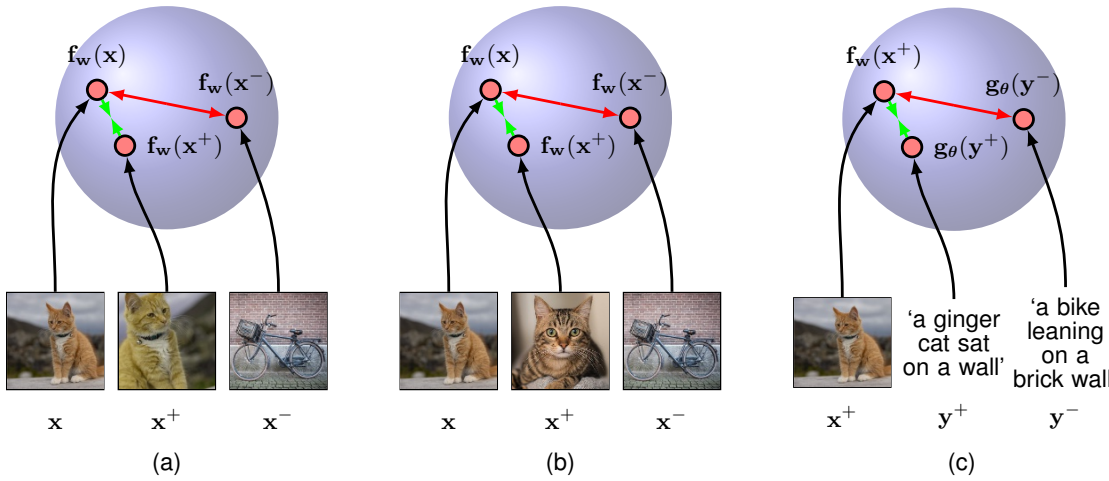


Figure 6.14 Illustration of three different contrastive learning paradigms. (a) The instance discrimination approach, where the positive pair is made up of the anchor and an augmented version of the same image. These are mapped to points in a normalized space that can be thought of as a unit hypersphere. The coloured arrows show that the loss encourages the representations of the positive pair to be closer together but pushes negative pairs further apart. (b) Supervised contrastive learning in which the positive pair consists of two different images from the same class. (c) The CLIP model in which the positive pair is made up of an image and an associated text snippet.

set with paired modalities can be used to learn representations. A comparison of the different contrastive learning methods we have discussed is shown in Figure 6.14.

6.3.6 General network architectures

So far, we have explored neural network architectures that are organized into a sequence of fully-connected layers. However, because there is a direct correspondence between a network diagram and its mathematical function, we can develop more general network mappings by considering more complex network diagrams. These must be restricted to a *feed-forward* architecture, in other words to one having no closed directed cycles, to ensure that the outputs are deterministic functions of the inputs. This is illustrated with a simple example in Figure 6.15. Each (hidden or output) unit in such a network computes a function given by

$$z_k = h\left(\sum_{j \in \mathcal{A}(k)} w_{kj} z_j + b_k\right) \tag{6.22}$$

where $\mathcal{A}(k)$ denotes the set of *ancestors* of node k, in other words the set of units that send connections to unit k, and b_k denotes the associated bias parameter. For a given set of values applied to the inputs of the network, successive application of (6.22) allows the activations of all units in the network to be evaluated including those of the output units.

Telegram: @uni_k

www.konkur.in

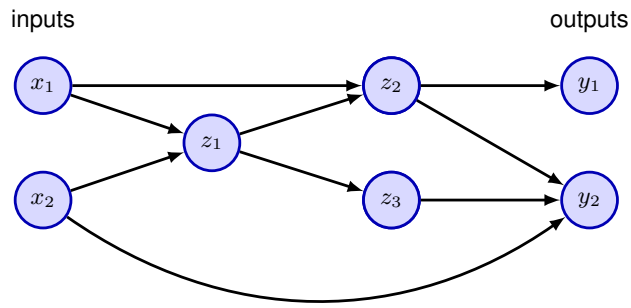


Figure 6.15 Example of a neural network having a general feed-forward topology. Note that each hidden and output unit has an associated bias parameter (omitted for clarity).

6.3.7 Tensors

We see that linear algebra plays a central role in neural networks, with quantities such as data sets, activations, and network parameters represented as scalars, vectors, and matrices. However, we also encounter variables of higher dimensionality. Consider, for example, a data set of N colour images each of which is I pixels high and J pixels wide. Each pixel is indexed by its row and column within the image and has red, green, and blue values. We have one such value for each image in the data set, and so we can represent a particular intensity value by a four-dimensional array $\mathbf{X}$ with elements x_{ijkn} where $i \in \{1, \ldots, I\}$ and $j \in \{1, \ldots, J\}$ index the row and column within the image, $k \in \{1, 2, 3\}$ indexes the red, green, and blue intensities, and $n \in \{1, \ldots, N\}$ indexes the particular image within the data set. These higher-dimensional arrays are called *tensors* and include scalars, vectors, and matrices as special cases. We will see many examples of such tensors when we discuss more sophisticated neural network architectures later in the book. Massively parallel processors such as GPUs are especially well suited to processing tensors.

6.4. Error Functions

Chapter 4
Chapter 5

In earlier chapters, we explored linear models for regression and classification, and in the process we derived suitable forms for the error functions along with corresponding choices for the output-unit activation function. The same considerations for choosing an error function apply for multilayer neural networks, and so for convenience, we will summarize the key points here.

6.4.1 Regression

Section 2.3.4

We start by discussing regression problems, and for the moment we consider a single target variable t that can take any real value. Following the discussion of regression in single-layer networks, we assume that t has a Gaussian distribution with an $\mathbf{x}$-dependent mean, which is given by the output of the neural network, so that

$$p(t|\mathbf{x}, \mathbf{w}) = \mathcal{N}\left(t|y(\mathbf{x}, \mathbf{w}), \sigma^2\right) \tag{6.23}$$

Telegram: @uni_k

www.konkur.in

where σ^2 is the variance of the Gaussian noise. Of course this is a somewhat restrictive assumption, and in some applications we will need to extend this approach to allow for more general distributions. For the conditional distribution given by (6.23), it is sufficient to take the output-unit activation function to be the identity, because such a network can approximate any continuous function from $\mathbf{x}$ to y. Given a data set of N i.i.d. observations $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$, along with corresponding target values $\mathbf{t} = \{t_1, \ldots, t_N\}$, we can construct the corresponding likelihood function:

Section 6.5

$$p(\mathbf{t}|\mathbf{X}, \mathbf{w}, \sigma^2) = \prod_{n=1}^{N} p(t_n|y(\mathbf{x}_n, \mathbf{w}), \sigma^2). \tag{6.24}$$

Note that in the machine learning literature, it is usual to consider the minimization of an error function rather than the maximization of the likelihood, and so here we will follow this convention. Taking the negative logarithm of the likelihood function (6.24), we obtain the error function

$$\frac{1}{2\sigma^2} \sum_{n=1}^{N} \{y(\mathbf{x}_n, \mathbf{w}) - t_n\}^2 + \frac{N}{2} \ln \sigma^2 + \frac{N}{2} \ln(2\pi), \tag{6.25}$$

which can be used to learn the parameters $\mathbf{w}$ and σ^2. Consider first the determination of $\mathbf{w}$. Maximizing the likelihood function is equivalent to minimizing the sum-of-squares error function given by

$$E(\mathbf{w}) = \frac{1}{2} \sum_{n=1}^{N} \{y(\mathbf{x}_n, \mathbf{w}) - t_n\}^2 \tag{6.26}$$

where we have discarded additive and multiplicative constants. The value of $\mathbf{w}$ found by minimizing $E(\mathbf{w})$ will be denoted $\mathbf{w}^\star$. Note that this will typically not correspond to the global maximum of the likelihood function because the nonlinearity of the network function $y(\mathbf{x}_n, \mathbf{w})$ causes the error $E(\mathbf{w})$ to be non-convex, and so finding the global optimum is generally infeasible. Moreover, regularization terms may be added to the error function and other modifications may be made to the training process, so that the resulting solution for the network parameters may differ significantly from the maximum likelihood solution.

Chapter 9

Having found $\mathbf{w}^\star$, the value of σ^2 can be found by minimizing the error function (6.25) to give

Exercise 6.8

$$\sigma^{2\star} = \frac{1}{N} \sum_{n=1}^{N} \{y(\mathbf{x}_n, \mathbf{w}^\star) - t_n\}^2. \tag{6.27}$$

Note that this can be evaluated once the iterative optimization required to find $\mathbf{w}^\star$ is completed.

If we have multiple target variables, and we assume that they are independent, conditional on $\mathbf{x}$ and $\mathbf{w}$, with shared noise variance σ^2, then the conditional distribution of the target values is given by

$$p(\mathbf{t}|\mathbf{x}, \mathbf{w}) = \mathcal{N}\left(\mathbf{t}|\mathbf{y}(\mathbf{x}, \mathbf{w}), \sigma^2 \mathbf{I}\right). \tag{6.28}$$

Telegram: @uni_k

www.konkur.in

Following the same argument as for a single target variable, we see that maximizing the likelihood function with respect to the weights is equivalent to minimizing the sum-of-squares error function:

Exercise 6.9

$$E(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N} \|\mathbf{y}(\mathbf{x}_n, \mathbf{w}) - \mathbf{t}_n\|^2. \tag{6.29}$$

The noise variance is then given by

$$\sigma^{2\star} = \frac{1}{NK}\sum_{n=1}^{N} \|\mathbf{y}(\mathbf{x}_n, \mathbf{w}^\star) - \mathbf{t}_n\|^2 \tag{6.30}$$

where K is the dimensionality of the target variable. The assumption of conditional independence of the target variables can be dropped at the expense of a slightly more complex optimization problem.

Exercise 6.10

Recall that there is a natural pairing of the error function (given by the negative log likelihood) and the output-unit activation function. In regression, we can view the network as having an output activation function that is the identity, so that $y_k = a_k$. The corresponding sum-of-squares error function then has the property

Section 5.4.6

$$\frac{\partial E}{\partial a_k} = y_k - t_k. \tag{6.31}$$

6.4.2 Binary classification

Now consider binary classification in which we have a single target variable t such that $t = 1$ denotes class $\mathcal{C}_1$ and $t = 0$ denotes class $\mathcal{C}_2$. Following the discussion of canonical link functions, we consider a network having a single output whose activation function is a logistic sigmoid (6.13) so that $0 \leqslant y(\mathbf{x}, \mathbf{w}) \leqslant 1$. We can interpret $y(\mathbf{x}, \mathbf{w})$ as the conditional probability $p(\mathcal{C}_1|\mathbf{x})$, with $p(\mathcal{C}_2|\mathbf{x})$ given by $1 - y(\mathbf{x}, \mathbf{w})$. The conditional distribution of targets given inputs is then a Bernoulli distribution of the form

Section 5.4.6

$$p(t|\mathbf{x}, \mathbf{w}) = y(\mathbf{x}, \mathbf{w})^t \left\{1 - y(\mathbf{x}, \mathbf{w})\right\}^{1-t}. \tag{6.32}$$

If we consider a training set of independent observations, then the error function, which is given by the negative log likelihood, is then a *cross-entropy* error of the form

$$E(\mathbf{w}) = -\sum_{n=1}^{N} \{t_n \ln y_n + (1 - t_n) \ln(1 - y_n)\} \tag{6.33}$$

where y_n denotes $y(\mathbf{x}_n, \mathbf{w})$. Simard, Steinkraus, and Platt (2003) found that using the cross-entropy error function instead of the sum-of-squares for a classification problem leads to faster training as well as improved generalization.

Note that there is no analogue of the noise variance σ^2 in (6.32) because the target values are assumed to be correctly labelled. However, the model is easily extended to allow for labelling errors by introducing a probability ϵ that the target

Exercise 6.11

Telegram: @uni_k

www.konkur.in

value t has been flipped to the wrong value (Opper and Winther, 2000). Here ϵ may be set in advance, or it may be treated as a hyperparameter whose value is inferred from the data.

If we have K separate binary classifications to perform, then we can use a network having K outputs each of which has a logistic-sigmoid activation function. Associated with each output is a binary class label $t_k \in \{0, 1\}$, where $k = 1, \ldots, K$. If we assume that the class labels are independent, given the input vector, then the conditional distribution of the targets is

$$p(\mathbf{t}|\mathbf{x}, \mathbf{w}) = \prod_{k=1}^{K} y_k(\mathbf{x}, \mathbf{w})^{t_k} \left[1 - y_k(\mathbf{x}, \mathbf{w})\right]^{1-t_k}. \tag{6.34}$$

Exercise 6.13

Taking the negative logarithm of the corresponding likelihood function then gives the following error function:

$$E(\mathbf{w}) = -\sum_{n=1}^{N}\sum_{k=1}^{K} \{t_{nk} \ln y_{nk} + (1 - t_{nk}) \ln(1 - y_{nk})\} \tag{6.35}$$

Exercise 6.14

where y_{nk} denotes $y_k(\mathbf{x}_n, \mathbf{w})$. Again, the derivative of the error function with respect to the pre-activation for a particular output unit takes the form (6.31), just as in the regression case.

6.4.3 multiclass classification

Finally, we consider the standard multiclass classification problem in which each input is assigned to one of K mutually exclusive classes. The binary target variables

Section 5.1.3

$t_k \in \{0, 1\}$ have a 1-of-K coding scheme indicating the class, and the network outputs are interpreted as $y_k(\mathbf{x}, \mathbf{w}) = p(t_k = 1|\mathbf{x})$, leading to the error function (5.80), which we reproduce here:

$$E(\mathbf{w}) = -\sum_{n=1}^{N}\sum_{k=1}^{K} t_{kn} \ln y_k(\mathbf{x}_n, \mathbf{w}). \tag{6.36}$$

Section 5.4.4

The output-unit activation function, which corresponds to the canonical link, is given by the softmax function:

$$y_k(\mathbf{x}, \mathbf{w}) = \frac{\exp(a_k(\mathbf{x}, \mathbf{w}))}{\sum_j \exp(a_j(\mathbf{x}, \mathbf{w}))}, \tag{6.37}$$

which satisfies $0 \leqslant y_k \leqslant 1$ and $\sum_k y_k = 1$. Note that the $y_k(\mathbf{x}, \mathbf{w})$ are unchanged if a constant is added to all of the $a_k(\mathbf{x}, \mathbf{w})$, causing the error function to be constant for some directions in weight space. This degeneracy is removed if an appropriate

Chapter 9

regularization term is added to the error function. Once again, the derivative of the error function with respect to the pre-activation for a particular output unit takes the

Exercise 6.15

familiar form (6.31).

Telegram: @uni_k

www.konkur.in

In summary, there is a natural choice of both output-unit activation function and matching error function according to the type of problem being solved. For regression, we use linear outputs and a sum-of-squares error, for multiple independent binary classifications, we use logistic sigmoid outputs and a cross-entropy error function, and for multi-class classification, we use softmax outputs with the corresponding multi-class cross-entropy error function. For classification problems involving two classes, we can use a single logistic sigmoid output, or alternatively, we can use a network with two outputs having a softmax output activation function.

This procedure is quite general, and by considering other forms of conditional distribution, we can derive the associated error functions as the corresponding negative log likelihood. We will see an example of this in the next section when we consider multimodal network outputs.

6.5. Mixture Density Networks

So far in this chapter we have discussed neural networks whose outputs represent simple probability distributions comprising either a Gaussian for continuous variables or a binary distribution for discrete variables. We close the chapter by showing how a neural network can represent more general conditional probabilities by treating the outputs of the network as the parameters of a more complex distribution, in this case a Gaussian mixture model. This is known as a *mixture density network*, and we will see how to define the associated error function and the corresponding output-unit activation functions.

6.5.1 Robot kinematics example

The goal of supervised learning is to model a conditional distribution $p(\mathbf{t}|\mathbf{x})$, which for many simple regression problems is chosen to be Gaussian. However, practical machine learning problems can often have significantly non-Gaussian distributions. These can arise, for example, with *inverse problems* in which the distribution can be multimodal, in which case the Gaussian assumption can lead to very poor predictions.

Exercise 6.16

As a simple illustration of an inverse problem, consider the kinematics of a robot arm, as illustrated in Figure 6.16. The *forward problem* involves finding the end effector position given the joint angles and has a unique solution. However, in practice we wish to move the end effector of the robot to a specific position, and to do this we must set appropriate joint angles. We therefore need to solve the inverse problem, which has two solutions, as seen in Figure 6.16.

Forward problems often correspond to causality in a physical system and generally have a unique solution. For instance, a specific pattern of symptoms in the human body may be caused by the presence of a particular disease. In machine learning, however, we typically have to solve an inverse problem, such as trying to predict the presence of a disease given a set of symptoms. If the forward problem involves a many-to-one mapping, then the inverse problem will have multiple solutions. For instance, several different diseases may result in the same symptoms.

Telegram: @uni_k

www.konkur.in

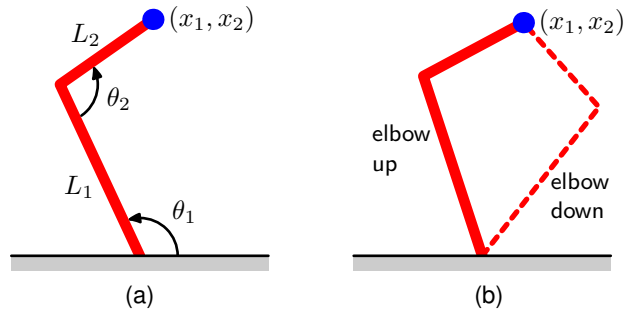


Figure 6.16 (a) A two-link robot arm, in which the Cartesian coordinates (x_1, x_2) of the end effector are determined uniquely by the two joint angles θ_1 and θ_2 and the (fixed) lengths L_1 and L_2 of the arms. This is known as the *forward kinematics* of the arm. (b) In practice, we have to find the joint angles that will give rise to a desired end effector position. This *inverse kinematics* has two solutions corresponding to 'elbow up' and 'elbow down'.

In the robotics example, the kinematics is defined by geometrical equations, and the multimodality is readily apparent. However, in many machine learning problems the presence of multimodality, particularly in problems involving spaces of high dimensionality, can be less obvious. For tutorial purposes, however, we will consider a simple toy problem for which we can easily visualize the multimodality. The data for this problem is generated by sampling a variable x uniformly over the interval $(0, 1)$, to give a set of values $\{x_n\}$, and the corresponding target values t_n are obtained by computing the function $x_n + 0.3\sin(2\pi x_n)$ and then adding uniform noise over the interval $(-0.1, 0.1)$. The inverse problem is then obtained by keeping the same data points but exchanging the roles of x and t. Figure 6.17 shows the data sets for the forward and inverse problems, along with the results of fitting two-layer neural networks having six hidden units and a single linear output unit by minimizing a sum-of-squares error function. Least squares corresponds to maximum likelihood under a Gaussian assumption. We see that this leads to a good model for the forward problem but a very poor model for the highly non-Gaussian inverse problem.

6.5.2 Conditional mixture distributions

We therefore seek a general framework for modelling conditional probability distributions. This can be achieved by using a mixture model for $p(\mathbf{t}|\mathbf{x})$ in which

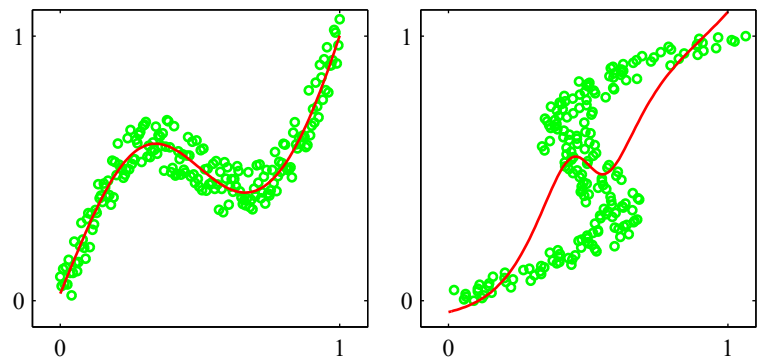


Figure 6.17 On the left is the data set for a simple forward problem in which the red curve shows the result of fitting a two-layer neural network by minimizing the sum-of-squares error function. The corresponding inverse problem, shown on the right, is obtained by exchanging the roles of x and t. Here the same network, again trained by minimizing the sum-of-squares error function, gives a poor fit to the data due to the multimodality of the data set.

Telegram: @uni_k

www.konkur.in

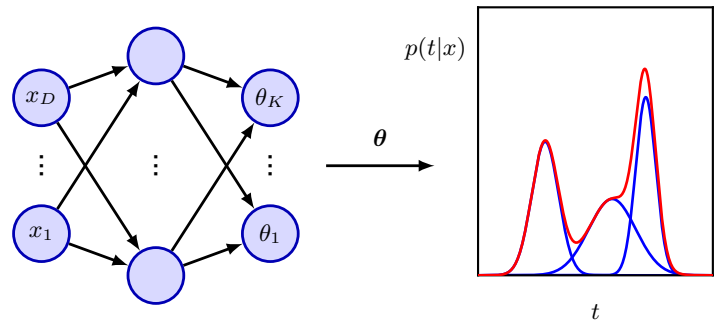


Figure 6.18 The *mixture density network* can represent general conditional probability densities $p(\mathbf{t}|\mathbf{x})$ by considering a parametric mixture model for the distribution of $\mathbf{t}$ whose parameters are determined by the outputs of a neural network that takes $\mathbf{x}$ as its input vector.

both the mixing coefficients as well as the component densities are flexible functions of the input vector $\mathbf{x}$, giving rise to a *mixture density network*. For any given value of $\mathbf{x}$, the mixture model provides a general formalism for modelling an arbitrary conditional density function $p(\mathbf{t}|\mathbf{x})$. Provided we consider a sufficiently flexible network, we then have a framework for approximating arbitrary conditional distributions.

Here we will develop the model explicitly for Gaussian components, so that

$$p(\mathbf{t}|\mathbf{x}) = \sum_{k=1}^{K} \pi_k(\mathbf{x}) \mathcal{N}\left(\mathbf{t}|\boldsymbol{\mu}_k(\mathbf{x}), \sigma_k^2(\mathbf{x})\right). \tag{6.38}$$

This is an example of a *heteroscedastic* model in which the noise variance on the data is a function of the input vector $\mathbf{x}$. Instead of Gaussians, we can use other distributions for the components, such as Bernoulli distributions if the target variables are binary rather than continuous. We have also specialized to the case of isotropic covariances for the components, although the mixture density network can readily be extended to allow for general covariance matrices by representing the covariances using a Cholesky factorization (Williams, 1996). Even with isotropic components, the conditional distribution $p(\mathbf{t}|\mathbf{x})$ does not assume factorization with respect to the components of $\mathbf{t}$ (in contrast to the standard sum-of-squares regression model) as a consequence of the mixture distribution.

We now take the various parameters of the mixture model, namely the mixing coefficients $\pi_k(\mathbf{x})$, the means $\boldsymbol{\mu}_k(\mathbf{x})$, and the variances $\sigma_k^2(\mathbf{x})$, to be governed by the outputs of a neural network that takes $\mathbf{x}$ as its input. The structure of this mixture density network is illustrated in Figure 6.18. The mixture density network is closely related to the mixture-of-experts model (Jacobs *et al.*, 1991). The principal difference is that a mixture of experts has independent parameters for each component model in the mixture, whereas in a mixture density network, the same function is used to predict the parameters of all the component densities as well as the mixing coefficients, and so the nonlinear hidden units are shared amongst the input-dependent functions.

The neural network in Figure 6.18 can, for example, be a two-layer network having sigmoidal (tanh) hidden units. If there are K components in the mixture model (6.38), and if $\mathbf{t}$ has L components, then the network will have K output-

Telegram: @uni_k

www.konkur.in

unit pre-activations denoted by a_k^π that determine the mixing coefficients $\pi_k(\mathbf{x})$, K outputs denoted by a_k^σ that determine the Gaussian standard deviations $\sigma_k(\mathbf{x})$, and $K \times L$ outputs denoted by a_{kj}^μ that determine the components $\mu_{kj}(\mathbf{x})$ of the Gaussian means $\boldsymbol{\mu}_k(\mathbf{x})$. The total number of network outputs is given by $(L+2)K$, unlike the usual L outputs for a network that simply predicts the conditional means of the target variables.

The mixing coefficients must satisfy the constraints

$$\sum_{k=1}^{K} \pi_k(\mathbf{x}) = 1, \qquad 0 \leqslant \pi_k(\mathbf{x}) \leqslant 1, \tag{6.39}$$

which can be achieved using a set of softmax outputs:

$$\pi_k(\mathbf{x}) = \frac{\exp(a_k^\pi)}{\sum_{l=1}^{K} \exp(a_l^\pi)}. \tag{6.40}$$

Similarly, the variances must satisfy $\sigma_k^2(\mathbf{x}) \geqslant 0$ and so can be represented in terms of the exponentials of the corresponding network pre-activations using

$$\sigma_k(\mathbf{x}) = \exp(a_k^\sigma). \tag{6.41}$$

Finally, because the means $\boldsymbol{\mu}_k(\mathbf{x})$ have real components, they can be represented directly by the network outputs:

$$\mu_{kj}(\mathbf{x}) = a_{kj}^\mu \tag{6.42}$$

in which the output-unit activation functions are given by the identity $f(a) = a$.

The learnable parameters of the mixture density network comprise the vector $\mathbf{w}$ of weights and biases in the neural network, which can be set by maximum likelihood or equivalently by minimizing an error function defined to be the negative logarithm of the likelihood. For independent data, this error function takes the form

$$E(\mathbf{w}) = -\sum_{n=1}^{N} \ln \left\{ \sum_{k=1}^{K} \pi_k(\mathbf{x}_n, \mathbf{w}) \mathcal{N}\left(\mathbf{t}_n | \boldsymbol{\mu}_k(\mathbf{x}_n, \mathbf{w}), \sigma_k^2(\mathbf{x}_n, \mathbf{w})\right) \right\} \tag{6.43}$$

where we have made the dependencies on $\mathbf{w}$ explicit.

6.5.3 Gradient optimization

To minimize the error function, we need to calculate the derivatives of the error $E(\mathbf{w})$ with respect to the components of $\mathbf{w}$. We will see later how to compute these derivatives automatically. It is instructive, however, to derive suitable expressions for the derivatives of the error with respect to the output-unit pre-activations explicitly as this highlights the probabilistic interpretation of these quantities. Because the error function (6.43) is composed of a sum of terms, one for each training data point, we can consider the derivatives for a particular input vector $\mathbf{x}_n$ with associated target vector $\mathbf{t}_n$. The derivatives of the total error E are obtained by summing over all

Chapter 8

Telegram: @uni_k

www.konkur.in

data points, or the individual gradients for each data point can be used directly in gradient-based optimization algorithms.

Chapter 7

It is convenient to introduce the following variables:

$$\gamma_{nk} = \gamma_k(\mathbf{t}_n|\mathbf{x}_n) = \frac{\pi_k \mathcal{N}_{nk}}{\sum_{l=1}^{K} \pi_l \mathcal{N}_{nl}} \tag{6.44}$$

where $\mathcal{N}_{nk}$ denotes $\mathcal{N}\left(\mathbf{t}_n|\boldsymbol{\mu}_k(\mathbf{x}_n), \sigma_k^2(\mathbf{x}_n)\right)$. These quantities have a natural interpretation as posterior probabilities for the components of the mixture in which the mixing coefficients $\pi_k(\mathbf{x})$ are viewed as $\mathbf{x}$-dependent prior probabilities.

Exercise 6.17

The derivatives of the error function with respect to the network output pre-activations governing the mixing coefficients are given by

Exercise 6.18

$$\frac{\partial E_n}{\partial a_k^{\pi}} = \pi_k - \gamma_{nk}. \tag{6.45}$$

Similarly, the derivatives with respect to the output pre-activations controlling the component means are given by

Exercise 6.19

$$\frac{\partial E_n}{\partial a_{kl}^{\mu}} = \gamma_{nk} \left\{ \frac{\mu_{kl} - t_{nl}}{\sigma_k^2} \right\}. \tag{6.46}$$

Finally, the derivatives with respect to the output pre-activations controlling the component variances are given by

Exercise 6.20

$$\frac{\partial E_n}{\partial a_k^{\sigma}} = \gamma_{nk} \left\{ L - \frac{\|\mathbf{t}_n - \boldsymbol{\mu}_k\|^2}{\sigma_k^2} \right\}. \tag{6.47}$$

6.5.4 Predictive distribution

We illustrate the use of a mixture density network by returning to the toy example of an inverse problem shown in Figure 6.17. Plots of the mixing coefficients $\pi_k(x)$, the means $\mu_k(x)$, and the conditional density contours corresponding to $p(t|x)$, are shown in Figure 6.19. The outputs of the neural network, and hence the parameters in the mixture model, are necessarily continuous single-valued functions of the input variables. However, we see from Figure 6.19(c) that the model is able to produce a conditional density that is unimodal for some values of x and trimodal for other values by modulating the amplitudes of the mixing components $\pi_k(\mathbf{x})$.

Once a mixture density network has been trained, it can predict the conditional density function of the target data for any given value of the input vector. This conditional density represents a complete description of the generator of the data, so far as the problem of predicting the value of the output vector is concerned. From this density function, we can calculate more specific quantities that may be of interest in different applications. One of the simplest of these is the mean, corresponding to the conditional average of the target data, and is given by

$$\mathbb{E}\left[\mathbf{t}|\mathbf{x}\right] = \int \mathbf{t} p(\mathbf{t}|\mathbf{x}) \, \mathrm{d}\mathbf{t} = \sum_{k=1}^{K} \pi_k(\mathbf{x}) \boldsymbol{\mu}_k(\mathbf{x}) \tag{6.48}$$

Telegram: @uni_k

www.konkur.in

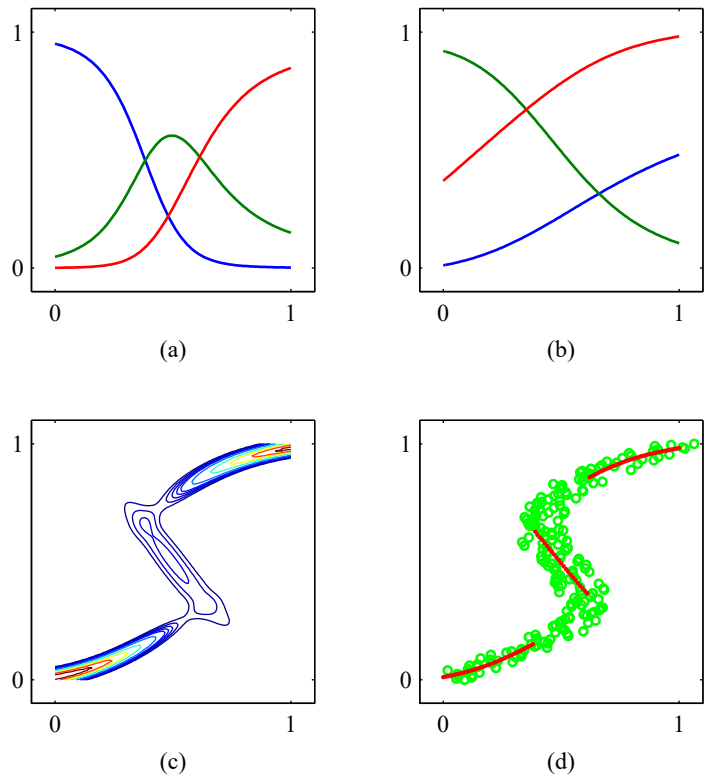


Figure 6.19 (a) Plot of the mixing coefficients $\pi_k(x)$ as a function of x for the three mixture components in a mixture density network trained on the data shown in Figure 6.17. The model has three Gaussian components and uses a two-layer neural network with five $\tanh$ sigmoidal units in the hidden layer and nine outputs (corresponding to the three means and three variances of the Gaussian components and the three mixing coefficients). At both small and large values of x, where the conditional probability density of the target data is unimodal, only one of the Gaussian components has a high value for its prior probability, whereas at intermediate values of x, where the conditional density is trimodal, the three mixing coefficients have comparable values. (b) Plots of the means $\mu_k(x)$ using the same colour coding as for the mixing coefficients. (c) Plot of the contours of the corresponding conditional probability density of the target data for the same mixture density network. (d) Plot of the approximate conditional mode, shown by the red points, of the conditional density.

where we have used (6.38). Because a standard network trained by least squares approximates the conditional mean, we see that a mixture density network can reproduce the conventional least-squares result as a special case. Of course, as we have already noted, for a multimodal distribution the conditional mean is of limited value.

Exercise 6.21

We can similarly evaluate the variance of the density function about the conditional average, to give

$$s^2(\mathbf{x}) = \mathbb{E}\left[\|\mathbf{t} - \mathbb{E}[\mathbf{t}|\mathbf{x}]\|^2 \,|\mathbf{x}\right] \tag{6.49}$$

$$= \sum_{k=1}^{K} \pi_k(\mathbf{x}) \left\{ \sigma_k^2(\mathbf{x}) + \left\| \boldsymbol{\mu}_k(\mathbf{x}) - \sum_{l=1}^{K} \pi_l(\mathbf{x}) \boldsymbol{\mu}_l(\mathbf{x}) \right\|^2 \right\} \tag{6.50}$$

where we have used (6.38) and (6.48). This is more general than the corresponding least-squares result because the variance is a function of $\mathbf{x}$.

We have seen that for multimodal distributions, the conditional mean can give a poor representation of the data. For instance, in controlling the simple robot arm shown in Figure 6.16, we need to pick one of the two possible joint angle settings

Telegram: @uni_k

www.konkur.in

to achieve the desired end-effector location, but the average of the two solutions is not itself a solution. In such cases, the conditional mode may be of more value. Because the conditional mode for the mixture density network does not have a simple analytical solution, a numerical iteration is required. A simple alternative is to take the mean of the most probable component (i.e., the one with the largest mixing coefficient) at each value of $\mathbf{x}$. This is shown for the toy data set in Figure 6.19(d).

Exercises

6.1 ($\star\star\star$) Use the result (2.126) to derive an expression for the surface area S_D and the volume V_D of a hypersphere of unit radius in D dimensions. To do this, consider the following result, which is obtained by transforming from Cartesian to polar coordinates:

$$\prod_{i=1}^{D} \int_{-\infty}^{\infty} e^{-x_i^2} \, \mathrm{d}x_i = S_D \int_0^{\infty} e^{-r^2} r^{D-1} \, \mathrm{d}r. \tag{6.51}$$

Using the gamma function, defined by

$$\Gamma(x) = \int_0^{\infty} t^{x-1} e^{-t} \, \mathrm{d}t \tag{6.52}$$

together with (2.126), evaluate both sides of this equation, and hence show that

$$S_D = \frac{2\pi^{D/2}}{\Gamma(D/2)}. \tag{6.53}$$

Next, by integrating with respect to the radius from 0 to 1, show that the volume of the unit hypersphere in D dimensions is given by

$$V_D = \frac{S_D}{D}. \tag{6.54}$$

Finally, use the results $\Gamma(1) = 1$ and $\Gamma(3/2) = \sqrt{\pi}/2$ to show that (6.53) and (6.54) reduce to the usual expressions for $D = 2$ and $D = 3$.

6.2 ($\star\star\star$) Consider a hypersphere of radius a in D dimensions together with the concentric hypercube of side $2a$, so that the hypersphere touches the hypercube at the centres of each of its sides. By using the results of Exercise 6.1, show that the ratio of the volume of the hypersphere to the volume of the cube is given by

$$\frac{\text{volume of hypersphere}}{\text{volume of cube}} = \frac{\pi^{D/2}}{D 2^{D-1} \Gamma(D/2)}. \tag{6.55}$$

Now make use of Stirling's formula in the form

$$\Gamma(x+1) \simeq (2\pi)^{1/2} e^{-x} x^{x+1/2}, \tag{6.56}$$

which is valid for $x \gg 1$, to show that, as $D \to \infty$, the ratio (6.55) goes to zero. Show also that the distance from the centre of the hypercube to one of the corners

Telegram: @uni_k

www.konkur.in

divided by the perpendicular distance to one of the sides is $\sqrt{D}$, which therefore goes to ∞ as $D \to \infty$. From these results, we see that, in a space of high dimensionality, most of the volume of a cube is concentrated in the large number of corners, which themselves become very long 'spikes'!

6.3 ($\star\star\star$) In this exercise, we explore the behaviour of the Gaussian distribution in high-dimensional spaces. Consider a Gaussian distribution in D dimensions given by

$$p(\mathbf{x}) = \frac{1}{(2\pi\sigma^2)^{D/2}} \exp\left(-\frac{\|\mathbf{x}\|^2}{2\sigma^2}\right). \tag{6.57}$$

We wish to find the density as a function of the radius in polar coordinates in which the direction variables have been integrated out. To do this, show that the integral of the probability density over a thin shell of radius r and thickness ϵ, where $\epsilon \ll 1$, is given by $p(r)\epsilon$ where

$$p(r) = \frac{S_D r^{D-1}}{(2\pi\sigma^2)^{D/2}} \exp\left(-\frac{r^2}{2\sigma^2}\right) \tag{6.58}$$

where S_D is the surface area of a unit hypersphere in D dimensions. Show that the function $p(r)$ has a single stationary point located, for large D, at $\widehat{r} \simeq \sqrt{D}\sigma$. By considering $p(\widehat{r} + \epsilon)$ where $\epsilon \ll \widehat{r}$, show that for large D,

$$p(\widehat{r} + \epsilon) = p(\widehat{r}) \exp\left(-\frac{3\epsilon^2}{2\sigma^2}\right), \tag{6.59}$$

which shows that $\widehat{r}$ is a maximum of the radial probability density and also that $p(r)$ decays exponentially away from its maximum at $\widehat{r}$ with length scale σ. We have already seen that $\sigma \ll \widehat{r}$ for large D, and so we see that most of the probability mass is concentrated in a thin shell at large radius. Finally, show that the probability density $p(\mathbf{x})$ is larger at the origin than at the radius $\widehat{r}$ by a factor of $\exp(D/2)$. We therefore see that most of the probability mass in a high-dimensional Gaussian distribution is located at a different radius from the region of high probability density.

6.4 ($\star\star$) Consider a two-layer network function of the form (6.11) in which the hidden-unit nonlinear activation functions $h(\cdot)$ are given by logistic sigmoid functions of the form

$$\sigma(a) = \{1 + \exp(-a)\}^{-1}. \tag{6.60}$$

Show that there exists an equivalent network, which computes exactly the same function, but with hidden-unit activation functions given by $\tanh(a)$ where the tanh function is defined by (6.14). Hint: first find the relation between $\sigma(a)$ and $\tanh(a)$, and then show that the parameters of the two networks differ by linear transformations.

6.5 ($\star\star$) The *swish* activation function (Ramachandran, Zoph, and Le, 2017) is defined by

$$h(x) = x\sigma(\beta x) \tag{6.61}$$

Telegram: @uni_k

www.konkur.in

where $\sigma(x)$ is the logistic-sigmoid activation function defined by (6.13). When used in a neural network, β can be treated as a learnable parameter. Either sketch or plot using software graphs of the swish activation function as well as its first derivative for $\beta = 0.1$, $\beta = 1.0$, and $\beta = 10$. Show that when $\beta \to \infty$, the swish function becomes the ReLU function.

6.6 ($\star$) We saw in (5.72) that the derivative of the logistic-sigmoid activation function can be expressed in terms of the function value itself. Derive the corresponding result for the $\tanh$ activation function defined by (6.14).

6.7 ($\star\star$) Show that the softplus activation function $\zeta(a)$ given by (6.16) satisfies the properties:

$$\zeta(a) - \zeta(-a) = a \tag{6.62}$$

$$\ln \sigma(a) = -\zeta(-a) \tag{6.63}$$

$$\frac{\mathrm{d}\zeta(a)}{\mathrm{d}a} = \sigma(a) \tag{6.64}$$

$$\zeta^{-1}(a) = \ln\left(\exp(a) - 1\right) \tag{6.65}$$

where $\sigma(a)$ is the logistic-sigmoid activation function given by (6.13).

6.8 ($\star$) Show that minimization of the error function (6.25) with respect to the variance σ^2 gives the result (6.27).

6.9 ($\star$) Show that maximizing the likelihood function under the conditional distribution (6.28) for a multioutput neural network is equivalent to minimizing the sum-of-squares error function (6.29). Also, show that the noise variance that minimizes this error function is given by (6.30).

6.10 ($\star\star$) Consider a regression problem involving multiple target variables in which it is assumed that the distribution of the targets, conditioned on the input vector $\mathbf{x}$, is a Gaussian of the form

$$p(\mathbf{t}|\mathbf{x}, \mathbf{w}) = \mathcal{N}(\mathbf{t}|\mathbf{y}(\mathbf{x}, \mathbf{w}), \boldsymbol{\Sigma}) \tag{6.66}$$

where $\mathbf{y}(\mathbf{x}, \mathbf{w})$ is the output of a neural network with input vector $\mathbf{x}$ and weight vector $\mathbf{w}$, and $\boldsymbol{\Sigma}$ is the covariance of the assumed Gaussian noise on the targets. Given a set of independent observations of $\mathbf{x}$ and $\mathbf{t}$, write down the error function that must be minimized to find the maximum likelihood solution for $\mathbf{w}$, if we assume that $\boldsymbol{\Sigma}$ is fixed and known. Now assume that $\boldsymbol{\Sigma}$ is also to be determined from the data, and write down an expression for the maximum likelihood solution for $\boldsymbol{\Sigma}$. Note that the optimizations of $\mathbf{w}$ and $\boldsymbol{\Sigma}$ are now coupled, in contrast to the case of independent target variables discussed in Section 6.4.1.

6.11 ($\star\star$) Consider a binary classification problem in which the target values are $t \in \{0, 1\}$, with a network output $y(\mathbf{x}, \mathbf{w})$ that represents $p(t = 1|\mathbf{x})$, and suppose that there is a probability ϵ that the class label on a training data point has been incorrectly set. Assuming i.i.d. data, write down the error function corresponding to the negative log likelihood. Verify that the error function (6.33) is obtained when $\epsilon = 0$. Note that

Telegram: @uni_k

www.konkur.in

this error function makes the model robust to incorrectly labelled data, in contrast to the usual cross-entropy error function.

6.12 ($\star\star$) The error function (6.33) for binary classification problems was derived for a network having a logistic-sigmoid output activation function, so that $0 \leqslant y(\mathbf{x}, \mathbf{w}) \leqslant 1$, and data having target values $t \in \{0, 1\}$. Derive the corresponding error function if we consider a network having an output $-1 \leqslant y(\mathbf{x}, \mathbf{w}) \leqslant 1$ and target values $t = 1$ for class $\mathcal{C}_1$ and $t = -1$ for class $\mathcal{C}_2$. What would be the appropriate choice of output-unit activation function?

6.13 ($\star$) Show that maximizing the likelihood for a multi-class neural network model in which the network outputs have the interpretation $y_k(\mathbf{x}, \mathbf{w}) = p(t_k = 1|\mathbf{x})$ is equivalent to minimizing the cross-entropy error function (6.36).

6.14 ($\star$) Show that the derivative of the error function (6.33) with respect to the pre-activation a_k for an output unit having a logistic-sigmoid activation function $y_k = \sigma(a_k)$, where $\sigma(a)$ is given by (6.13), satisfies (6.31).

6.15 ($\star$) Show that the derivative of the error function (6.36) with respect to the pre-activation a_k for output units having a softmax activation function (6.37) satisfies (6.31).

6.16 ($\star\star$) Write down a pair of equations that express the Cartesian coordinates (x_1, x_2) for the robot arm shown in Figure 6.16 in terms of the joint angles θ_1 and θ_2 and the lengths L_1 and L_2 of the links. Assume the origin of the coordinate system is given by the attachment point of the lower arm. These equations define the forward kinematics of the robot arm.

6.17 ($\star\star$) Show that the variable γ_{nk} defined by (6.44) can be viewed as the posterior probabilities $p(k|\mathbf{t})$ for the components of the mixture distribution (6.38) in which the mixing coefficients $\pi_k(\mathbf{x})$ are viewed as $\mathbf{x}$-dependent prior probabilities $p(k)$.

6.18 ($\star\star$) Derive the result (6.45) for the derivative of the error function with respect to the network output pre-activations controlling the mixing coefficients in the mixture density network.

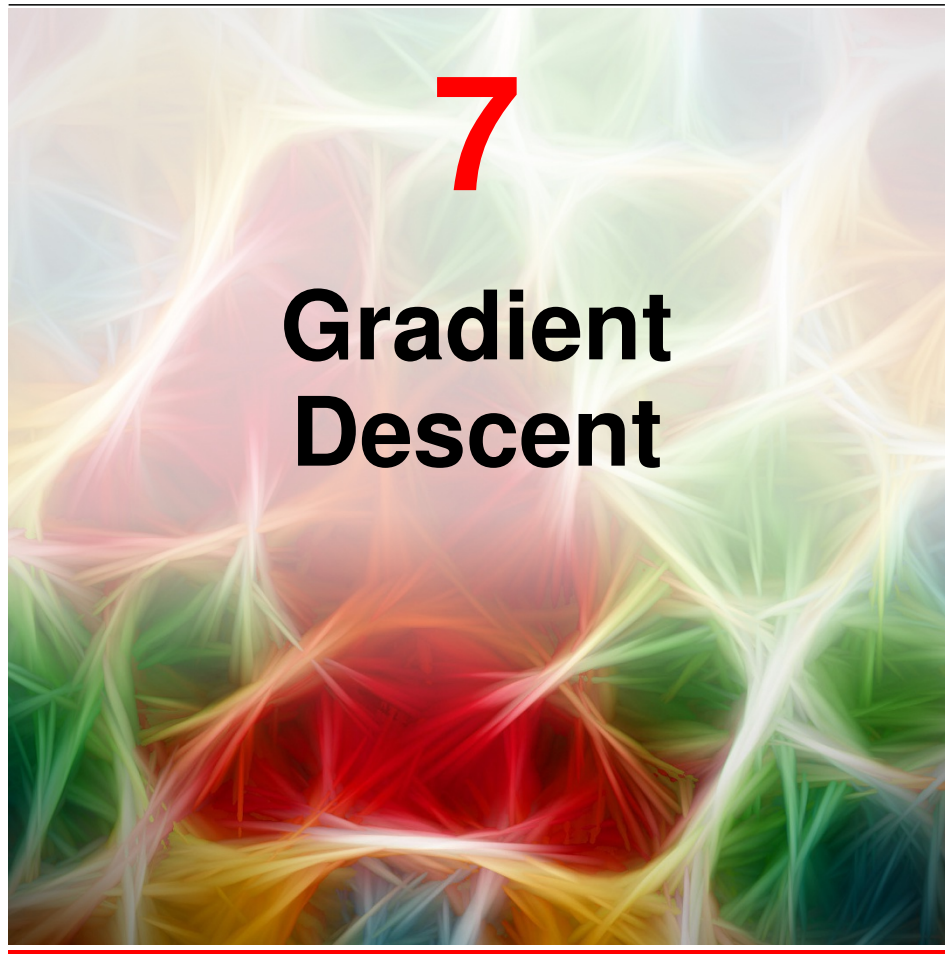
6.19 ($\star\star$) Derive the result (6.46) for the derivative of the error function with respect to the network output pre-activations controlling the component means in the mixture density network.

6.20 ($\star\star$) Derive the result (6.47) for the derivative of the error function with respect to the network output pre-activations controlling the component variances in the mixture density network.

6.21 ($\star\star\star$) Verify the results (6.48) and (6.50) for the conditional mean and variance of the mixture density network model.

Telegram: @uni_k

www.konkur.in



7 Gradient Descent

In the previous chapter we saw that neural networks are a very broad and flexible class of functions and are able in principle to approximate any desired function to arbitrarily high accuracy given a sufficiently large number of hidden units. Moreover, we saw that deep neural networks can encode inductive biases corresponding to hierarchical representations, which prove valuable in a wide range of practical applications. We now turn to the task of finding a suitable setting for the network parameters (weights and biases), based on a set of training data.

As with the regression and classification models discussed in earlier chapters, we choose the model parameters by optimizing an error function. We have seen how to define a suitable error function for a particular application by using maximum likelihood. Although in principle the error function could be minimized numerically through a series of direct error function evaluations, this turns out to be very inefficient. Instead, we turn to another core concept that is used in deep learning, which

Section 6.4

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024
C. M. Bishop, H. Bishop, *Deep Learning*, https://doi.org/10.1007/978-3-031-45468-4_7

Telegram: @uni_k

www.konkur.in

is that optimizing the error function can be done much more efficiently by making use of gradient information, in other words by evaluating the derivatives of the error function with respect to the network parameters. This is why we took care to ensure that the function represented by the neural network is differentiable by design. Likewise, the error function itself also needs to be differentiable.

The required derivatives of the error function with respect to each of the parameters in the network can be evaluated efficiently using a technique called *back-propagation*, which involves successive computations that flow backwards through the network in a way that is analogous to the forward flow of function computations during the evaluation of the network outputs.

Chapter 8

Although the likelihood is used to define an error function, the goal when optimizing the error function in a neural network is to achieve good generalization on test data. In classical statistics, maximum likelihood is used to fit a parametric model to a finite data set, in which the number of data points typically far exceeds the number of parameters in the model. The optimal solution has the maximum value of the likelihood function, and the values found for the fitted parameters are of direct interest. By contrast, modern deep learning works with very rich models containing huge numbers of learnable parameters, and the goal is never simply exact optimization. Instead, the properties and behaviour of the learning algorithm itself, along with various methods for regularization, are important in determining how well the solution generalizes to new data.

Section 9.3.2

Chapter 9

7.1. Error Surfaces

Our goal during training is to find values for the weights and biases in the neural network that will allow it to make effective predictions. For convenience we will group these parameters into a single vector $\mathbf{w}$, and we will optimize $\mathbf{w}$ by using a chosen error function $E(\mathbf{w})$. At this point, it is useful to have a geometrical picture of the error function, which we can view as a surface sitting over 'weight space', as shown in Figure 7.1.

First note that if we make a small step in weight space from $\mathbf{w}$ to $\mathbf{w} + \delta\mathbf{w}$ then the change in the error function is given by

$$\delta E \simeq \delta\mathbf{w}^{\mathrm{T}} \nabla E(\mathbf{w}) \tag{7.1}$$

where the vector $\nabla E(\mathbf{w})$ points in the direction of the greatest rate of increase of the error function. Provided the error $E(\mathbf{w})$ is a smooth, continuous function of $\mathbf{w}$, its smallest value will occur at a point in weight space such that the gradient of the error function vanishes, so that

$$\nabla E(\mathbf{w}) = 0 \tag{7.2}$$

as otherwise we could make a small step in the direction of $-\nabla E(\mathbf{w})$ and thereby further reduce the error. Points at which the gradient vanishes are called stationary points and may be further classified into minima, maxima, and saddle points.

Section 7.1.1

Telegram: @uni_k

www.konkur.in

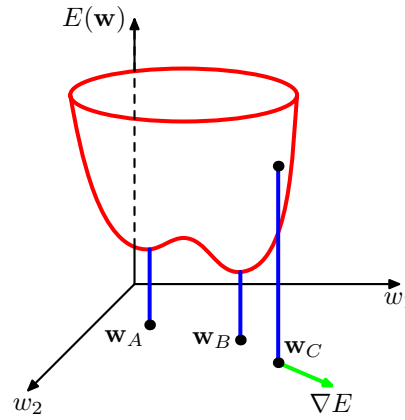


Figure 7.1 Geometrical view of the error function $E(\mathbf{w})$ as a surface sitting over weight space. Point $\mathbf{w}_A$ is a local minimum and $\mathbf{w}_B$ is the global minimum, so that $E(\mathbf{w}_\mathrm{A}) > E(\mathbf{w}_\mathrm{B})$. At any point $\mathbf{w}_C$, the local gradient of the error surface is given by the vector ∇E.

We will aim to find a vector $\mathbf{w}$ such that $E(\mathbf{w})$ takes its smallest value. However, the error function typically has a highly nonlinear dependence on the weights and bias parameters, and so there will be many points in weight space at which the gradient vanishes (or is numerically very small). Indeed, for any point $\mathbf{w}$ that is a local minimum, there will generally be other points in weight space that are equivalent minima. For instance, in a two-layer network of the kind shown in Figure 6.9, with M hidden units, each point in weight space is a member of a family of $M!\,2^M$ equivalent points.

Section 6.2.4

Furthermore, there may be multiple non-equivalent stationary points and in particular multiple non-equivalent minima. A minimum that corresponds to the smallest value of the error function across the whole of $\mathbf{w}$-space is said to be a *global minimum*. Any other minima corresponding to higher values of the error function are said to be *local minima*. The error surfaces for deep neural networks can be very complex, and it was thought that gradient-based methods might become trapped in poor local minima. In practice, this seems not to be the case, and large networks can reach solutions with similar performance under a variety of initial conditions.

Section 9.3.2

7.1.1 Local quadratic approximation

Insight into the optimization problem and into the various techniques for solving it can be obtained by considering a local quadratic approximation to the error function. The Taylor expansion of $E(\mathbf{w})$ around some point $\widehat{\mathbf{w}}$ in weight space is given by

$$E(\mathbf{w}) \simeq E(\widehat{\mathbf{w}}) + (\mathbf{w} - \widehat{\mathbf{w}})^{\mathrm{T}}\mathbf{b} + \frac{1}{2}(\mathbf{w} - \widehat{\mathbf{w}})^{\mathrm{T}}\mathbf{H}(\mathbf{w} - \widehat{\mathbf{w}}) \tag{7.3}$$

where cubic and higher terms have been omitted. Here $\mathbf{b}$ is defined to be the gradient of E evaluated at $\widehat{\mathbf{w}}$

$$\mathbf{b} \equiv \nabla E|_{\mathbf{w}=\widehat{\mathbf{w}}}\,. \tag{7.4}$$

The *Hessian* is defined to be the corresponding matrix of second derivatives

$$\mathbf{H}(\widehat{\mathbf{w}}) = \nabla\nabla E(\mathbf{w})|_{\mathbf{w}=\widehat{\mathbf{w}}}\,. \tag{7.5}$$

Telegram: @uni_k

www.konkur.in

If there is a total of W weights and biases in the network, then $\mathbf{w}$ and $\mathbf{b}$ have length W and $\mathbf{H}$ has dimensionality $W \times W$. From (7.3), the corresponding local approximation to the gradient is given by

$$\nabla E(\mathbf{w}) = \mathbf{b} + \mathbf{H}(\mathbf{w} - \widehat{\mathbf{w}}). \tag{7.6}$$

For points $\mathbf{w}$ that are sufficiently close to $\widehat{\mathbf{w}}$, these expressions will give reasonable approximations for the error and its gradient.

Consider the particular case of a local quadratic approximation around a point $\mathbf{w}^\star$ that is a minimum of the error function. In this case there is no linear term, because $\nabla E = 0$ at $\mathbf{w}^\star$, and (7.3) becomes

$$E(\mathbf{w}) = E(\mathbf{w}^\star) + \frac{1}{2}(\mathbf{w} - \mathbf{w}^\star)^{\mathrm{T}}\mathbf{H}(\mathbf{w} - \mathbf{w}^\star) \tag{7.7}$$

where the Hessian $\mathbf{H}$ is evaluated at $\mathbf{w}^\star$. To interpret this geometrically, consider the eigenvalue equation for the Hessian matrix:

$$\mathbf{H}\mathbf{u}_i = \lambda_i \mathbf{u}_i \tag{7.8}$$

Appendix A where the eigenvectors $\mathbf{u}_i$ form a complete orthonormal set so that

$$\mathbf{u}_i^{\mathrm{T}}\mathbf{u}_j = \delta_{ij}. \tag{7.9}$$

We now expand $(\mathbf{w} - \mathbf{w}^\star)$ as a linear combination of the eigenvectors in the form

$$\mathbf{w} - \mathbf{w}^\star = \sum_i \alpha_i \mathbf{u}_i. \tag{7.10}$$

This can be regarded as a transformation of the coordinate system in which the origin is translated to the point $\mathbf{w}^\star$ and the axes are rotated to align with the eigenvectors *Appendix A* through the orthogonal matrix whose columns are $\{\mathbf{u}_1, \ldots, \mathbf{u}_W\}$. By substituting (7.10) into (7.7) and using (7.8) and (7.9), the error function can be written in the *Exercise 7.1* form

$$E(\mathbf{w}) = E(\mathbf{w}^\star) + \frac{1}{2}\sum_i \lambda_i \alpha_i^2. \tag{7.11}$$

Suppose we set all $\alpha_i = 0$ for $i \neq j$ and then vary α_j, corresponding to moving $\mathbf{w}$ away from $\mathbf{w}^\star$ in the direction of $\mathbf{u}_j$. We see from (7.11) that the error function will increase if the corresponding eigenvalue λ_j is positive and will decrease if it is negative. If all eigenvalues are positive then $\mathbf{w}^\star$ corresponds to a local minimum of the error function, whereas if they are all negative then $\mathbf{w}^\star$ corresponds to a local maximum. If we have a mix of positive and negative eigenvalues then $\mathbf{w}^\star$ represents a saddle point.

A matrix $\mathbf{H}$ is said to be *positive definite* if, and only if,

$$\mathbf{v}^{\mathrm{T}}\mathbf{H}\mathbf{v} > 0, \qquad \text{for all } \mathbf{v}. \tag{7.12}$$

Telegram: @uni_k

www.konkur.in

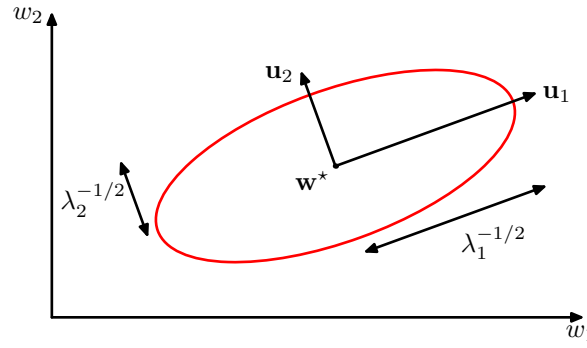


Figure 7.2 In the neighbourhood of a minimum $\mathbf{w}^\star$, the error function can be approximated by a quadratic. Contours of constant error are then ellipses whose axes are aligned with the eigenvectors $\mathbf{u}_i$ of the Hessian matrix, with lengths that are inversely proportional to the square roots of the corresponding eigenvectors λ_i.

Because the eigenvectors $\{\mathbf{u}_i\}$ form a complete set, an arbitrary vector $\mathbf{v}$ can be written in the form

$$\mathbf{v} = \sum_i c_i \mathbf{u}_i. \tag{7.13}$$

From (7.8) and (7.9), we then have

$$\mathbf{v}^{\mathrm{T}}\mathbf{H}\mathbf{v} = \sum_i c_i^2 \lambda_i \tag{7.14}$$

Exercise 7.2 and so $\mathbf{H}$ will be positive definite if, and only if, all its eigenvalues are positive. Thus, a necessary and sufficient condition for $\mathbf{w}^\star$ to be a local minimum is that the gradient of the error function should vanish at $\mathbf{w}^\star$ and the Hessian matrix evaluated *Exercise 7.3* at $\mathbf{w}^\star$ should be positive definite. In the new coordinate system, whose basis vectors are given by the eigenvectors $\{\mathbf{u}_i\}$, the contours of constant $E(\mathbf{w})$ are axis-aligned *Exercise 7.6* ellipses centred on the origin, as illustrated in Figure 7.2.

7.2. Gradient Descent Optimization

There is little hope of finding an analytical solution to the equation $\nabla E(\mathbf{w}) = 0$ for an error function as complex as one defined by a neural network, and so we resort to iterative numerical procedures. The optimization of continuous nonlinear functions is a widely studied problem, and there exists an extensive literature on how to solve it efficiently. Most techniques involve choosing some initial value $\mathbf{w}^{(0)}$ for the weight vector and then moving through weight space in a succession of steps of the form

$$\mathbf{w}^{(\tau)} = \mathbf{w}^{(\tau-1)} + \Delta\mathbf{w}^{(\tau-1)} \tag{7.15}$$

where τ labels the iteration step. Different algorithms involve different choices for the weight vector update $\Delta\mathbf{w}^{(\tau)}$.

Because of the complex shape of the error surface for all but the simplest neural networks, the solution found will depend, among other things, on the particular choice of initial parameter values $\mathbf{w}^{(0)}$. To find a sufficiently good solution, it may

Telegram: @uni_k

www.konkur.in

be necessary to run a gradient-based algorithm multiple times, each time using a different randomly chosen starting point, and comparing the resulting performance on an independent validation set.

7.2.1 Use of gradient information

The gradient of an error function for a deep neural network can be evaluated efficiently using the technique of error backpropagation, and applying this gradient information can lead to significant improvements in the speed of network training. We can see why this is so, as follows.

Chapter 8

In the quadratic approximation to the error function given by (7.3), the error surface is specified by the quantities $\mathbf{b}$ and $\mathbf{H}$, which contain a total of $W(W+3)/2$ independent elements (because the matrix $\mathbf{H}$ is symmetric), where W is the dimensionality of $\mathbf{w}$ (i.e., the total number of learnable parameters in the network). The location of the minimum of this quadratic approximation therefore depends on $\mathcal{O}(W^2)$ parameters, and we should not expect to be able to locate the minimum until we have gathered $\mathcal{O}(W^2)$ independent pieces of information. If we do not make use of gradient information, we would expect to have to perform $\mathcal{O}(W^2)$ function evaluations, each of which would require $\mathcal{O}(W)$ steps. Thus, the computational effort needed to find the minimum using such an approach would be $\mathcal{O}(W^3)$.

Exercise 7.7

Now compare this with an algorithm that makes use of the gradient information. Because ∇E is a vector of length W, each evaluation of ∇E brings W pieces of information, and so we might hope to find the minimum of the function in $\mathcal{O}(W)$ gradient evaluations. As we shall see, by using error backpropagation, each such evaluation takes only $\mathcal{O}(W)$ steps and so the minimum can now be found in $\mathcal{O}(W^2)$ steps. Although the quadratic approximation only holds in the neighbourhood of a minimum, the efficiency gains are generic. For this reason, the use of gradient information forms the basis of all practical algorithms for training neural networks.

Chapter 8

7.2.2 Batch gradient descent

The simplest approach to using gradient information is to choose the weight update in (7.15) such that there is a small step in the direction of the negative gradient, so that

$$\mathbf{w}^{(\tau)} = \mathbf{w}^{(\tau-1)} - \eta \nabla E(\mathbf{w}^{(\tau-1)}) \tag{7.16}$$

where the parameter $\eta > 0$ is known as the *learning rate*. After each such update, the gradient is re-evaluated for the new weight vector $\mathbf{w}^{(\tau+1)}$ and the process repeated. At each step, the weight vector is moved in the direction of the greatest rate of decrease of the error function, and so this approach is known as *gradient descent* or *steepest descent*. Note that the error function is defined with respect to a training set, and so to evaluate ∇E, each step requires that the entire training set be processed. Techniques that use the whole data set at once are called *batch* methods.

7.2.3 Stochastic gradient descent

Deep learning methods benefit greatly from very large data sets. However, batch methods can become extremely inefficient if there are many data points in the training set because each error function or gradient evaluation requires the entire data set

Telegram: @uni_k

www.konkur.in

Algorithm 7.1: Stochastic gradient descent

Input: Training set of data points indexed by $n \in \{1, \ldots, N\}$
Error function per data point $E_n(\mathbf{w})$
Learning rate parameter η
Initial weight vector $\mathbf{w}$

Output: Final weight vector $\mathbf{w}$

$n \leftarrow 1$

repeat

  $\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla E_n(\mathbf{w})$ `// update weight vector`

  $n \leftarrow n + 1 (\text{mod } N)$ `// iterate over data`

until convergence

return $\mathbf{w}$

to be processed. To find a more efficient approach, note that error functions based on maximum likelihood for a set of independent observations comprise a sum of terms, one for each data point:

$$E(\mathbf{w}) = \sum_{n=1}^{N} E_n(\mathbf{w}). \tag{7.17}$$

The most widely used training algorithms for large data sets are based on a sequential version of gradient descent known as *stochastic gradient descent* (Bottou, 2010), or SGD, which updates the weight vector based on one data point at a time, so that

$$\mathbf{w}^{(\tau)} = \mathbf{w}^{(\tau-1)} - \eta \nabla E_n(\mathbf{w}^{(\tau-1)}). \tag{7.18}$$

This update is repeated by cycling through the data. A complete pass through the whole training set is known as a training *epoch*. This technique is also known as *online gradient descent*, especially if the data arises from a continuous stream of new data points. Stochastic gradient descent is summarized in Algorithm 7.1.

A further advantage of stochastic gradient descent, compared to batch gradient descent, is that it handles redundancy in the data much more efficiently. To see this, consider an extreme example in which we take a data set and double its size by duplicating every data point. Note that this simply multiplies the error function by a factor of 2 and so is equivalent to using the original error function, if the value of the learning rate is adjusted to compensate. Batch methods will require double the computational effort to evaluate the batch error function gradient, whereas stochastic gradient descent will be unaffected. Another property of stochastic gradient descent is the possibility of escaping from local minima, since a stationary point with respect to the error function for the whole data set will generally not be a stationary point for each data point individually.

Telegram: @uni_k

www.konkur.in

7.2.4 Mini-batches

A downside of stochastic gradient descent is that the gradient of the error function computed from a single data point provides a very noisy estimate of the gradient of the error function computed on the full data set. We can consider an intermediate approach in which a small subset of data points, called a *mini-batch*, is used to evaluate the gradient at each iteration. In determining the optimum size for the mini-batch, note that the error in computing the mean from N samples is given by $\sigma/\sqrt{N}$ where σ is the standard deviation of the distribution generating the data. This indicates that there are diminishing returns in estimating the true gradient from increasing the batch size. If we increase the size of the mini-batch by a factor of 100 then the error only reduces by a factor of 10. Another consideration in choosing the mini-batch size is the desire to make efficient use of the hardware architecture on which the code is running. For example, on some hardware platforms, mini-batch sizes that are powers of 2 (for example, 64, 128, 256, ...) work well.

Exercise 7.8

One important consideration when using mini-batches is that the constituent data points should be chosen randomly from the data set, since in raw data sets there may be correlations between successive data points arising from the way the data was collected (for example, if the data points have been ordered alphabetically or by date). This is often handled by randomly shuffling the entire data set and then subsequently drawing mini-batches as successive blocks of data. The data set can also be reshuffled between iterations through the data set, so that each mini-batch is unlikely to have been used before, which can help escape local minima. The variant of stochastic gradient descent with mini-batches is summarized in Algorithm 7.2. Note that the learning algorithm is often still called 'stochastic gradient descent' even when mini-batches are used.

7.2.5 Parameter initialization

Iterative algorithms such as gradient descent require that we choose some initial setting for the parameters being learned. The specific initialization can have a significant effect on how long it takes to reach a solution and on the generalization performance of the resulting trained network. Unfortunately, there is relatively little theory to guide the initialization strategy.

One key consideration, however, is *symmetry breaking*. Consider a set of hidden units or output units that take the same inputs. If the parameters were all initialized with the same value, for example if they were all set to zero, the parameters of these units would all be updated in unison and the units would each compute the same function and hence be redundant. This problem can be addressed by initializing parameters randomly from some distribution to break symmetry. If computational resources permit, the network might be trained multiple times starting from different random initializations and the results compared on held-out data.

The distribution used to initialize the weights is typically either a uniform distribution in the range $[-\epsilon, \epsilon]$ or a zero-mean Gaussian of the form $\mathcal{N}(0, \epsilon^2)$. The choice of the value of ϵ is important, and various heuristics to select it have been proposed. One widely used approach is called *He initialization* (He *et al.*, 2015b). Consider a

Telegram: @uni_k

www.konkur.in

Algorithm 7.2: Mini-batch stochastic gradient descent

Input: Training set of data points indexed by $n \in \{1, \ldots, N\}$
Batch size B
Error function per mini-batch $E_{n:n+B-1}(\mathbf{w})$
Learning rate parameter η
Initial weight vector $\mathbf{w}$

Output: Final weight vector $\mathbf{w}$

$n \leftarrow 1$
repeat
  $\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla E_{n:n+B-1}(\mathbf{w})$ `// weight vector update`
  $n \leftarrow n + B$
  if $n > N$ **then**
    shuffle data
    $n \leftarrow 1$
  end if
until convergence
return $\mathbf{w}$

network in which layer l evaluates the following transformations

$$a_i^{(l)} = \sum_{j=1}^{M} w_{ij} z_j^{(l-1)} \tag{7.19}$$

$$z_i^{(l)} = \mathrm{ReLU}(a_i^{(l)}) \tag{7.20}$$

where M is the number of units that send connections to unit i, and the ReLU activation function is given by (6.17). Suppose we initialize the weights using a Gaussian $\mathcal{N}(0, \epsilon^2)$, and suppose that the outputs $z_j^{(l-1)}$ of the units in layer $l-1$ have variance λ^2. Then we can easily show that

Exercise 7.9

$$\mathbb{E}[a_i^{(l)}] = 0 \tag{7.21}$$

$$\mathrm{var}[z_j^{(l)}] = \frac{M}{2} \epsilon^2 \lambda^2 \tag{7.22}$$

where the factor of $1/2$ arises from the ReLU activation function. Ideally we want to ensure that the variance of the pre-activations neither decays to zero nor grows significantly as we propagate from one layer to the next. If we therefore require that the units at layer l also have variance λ^2 then we arrive at the following choice for the standard deviation of the Gaussian used to initialize the weights that feed into a

Telegram: @uni_k

www.konkur.in

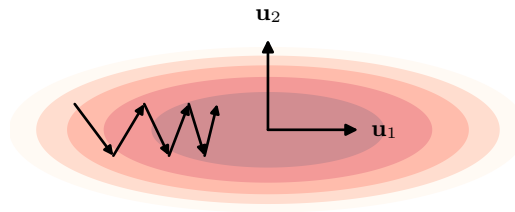


Figure 7.3 Schematic illustration of fixed-step gradient descent for an error function that has substantially different curvatures along different directions. The error surface E has the form of a long valley, as depicted by the ellipses. Note that, for most points in weight space, the local negative gradient vector $-\nabla E$ does not point towards the minimum of the error function. Successive steps of gradient descent can therefore oscillate across the valley, leading to very slow progress along the valley towards the minimum. The vectors $\mathbf{u}_1$ and $\mathbf{u}_2$ are the eigenvectors of the Hessian matrix.

unit with M inputs:

$$\epsilon = \sqrt{\frac{2}{M}}. \tag{7.23}$$

It is also possible to treat the scale ϵ of the initialization distribution as a hyperparameter and to explore different values across multiple training runs. The bias parameters are typically set to small positive values to ensure that most pre-activations are initially active during learning. This is particularly helpful with ReLU units, where we want the pre-activations to be positive so that there is a non-zero gradient to drive learning.

Another important class of techniques for initializing the parameters of a neural network is by using the values that result from training the network on a different task or by exploiting various forms of unsupervised training. These techniques fall into the broad class of *transfer learning* techniques.

Section 6.3.4

7.3. Convergence

When applying gradient descent in practice, we need to choose a value for the learning rate parameter η. Consider the simple error surface depicted in Figure 7.3 for a hypothetical two-dimensional weight space in which the curvature of E varies significantly with direction, creating a 'valley'. At most points on the error surface, the local gradient vector for batch gradient descent, which is perpendicular to the local contour, does not point directly towards the minimum. Intuitively we might expect that increasing the value of η should lead to bigger steps through weight space and hence faster convergence. However, the successive steps oscillate back and forth across the valley, and if we increase η too much, those oscillations will become divergent. Because η must be kept sufficiently small to avoid divergent oscillations across the valley, progress along the valley is very slow. Gradient descent then takes many small steps to reach the minimum and is a very inefficient procedure.

We can gain deeper insight into the nature of this problem by considering the quadratic approximation to the error function in the neighbourhood of the minimum. From (7.7), (7.8), and (7.10), the gradient of the error function in this approximation

Section 7.1.1

Telegram: @uni_k

www.konkur.in

can be written as

$$\nabla E = \sum_i \alpha_i \lambda_i \mathbf{u}_i. \tag{7.24}$$

Again using (7.10) we can express the change in the weight vector in terms of corresponding changes in the coefficients $\{\alpha_i\}$:

$$\Delta \mathbf{w} = \sum_i \Delta\alpha_i \mathbf{u}_i. \tag{7.25}$$

Combining (7.24) with (7.25) and the gradient descent formula (7.16) and using the orthonormality relation (7.9) for the eigenvectors of the Hessian, we obtain the following expression for the change in α_i at each step of the gradient descent algorithm:

$$\Delta\alpha_i = -\eta\lambda_i\alpha_i \tag{7.26}$$

Exercise 7.10

from which it follows that

$$\alpha_i^{\text{new}} = (1 - \eta\lambda_i)\alpha_i^{\text{old}} \tag{7.27}$$

where 'old' and 'new' denote values before and after a weight update. Using the orthonormality relation (7.9) for the eigenvectors together with (7.10), we have

$$\mathbf{u}_i^{\mathrm{T}}(\mathbf{w} - \mathbf{w}^\star) = \alpha_i \tag{7.28}$$

and so α_i can be interpreted as the distance to the minimum along the direction $\mathbf{u}_i$. From (7.27) we see that these distances evolve independently such that, at each step, the distance along the direction of $\mathbf{u}_i$ is multiplied by a factor $(1-\eta\lambda_i)$. After a total of T steps we have

$$\alpha_i^{(T)} = (1 - \eta\lambda_i)^T \alpha_i^{(0)}. \tag{7.29}$$

It follows that, provided $|1 - \eta\lambda_i| < 1$, the limit $T \to \infty$ leads to $\alpha_i = 0$, which from (7.28) shows that $\mathbf{w} = \mathbf{w}^\star$ and so the weight vector has reached the minimum of the error.

Note that (7.29) demonstrates that gradient descent leads to linear convergence in the neighbourhood of a minimum. Also, convergence to the stationary point requires that all the λ_i be positive, which in turn implies that the stationary point is indeed a minimum. By making η larger we can make the factor $(1 - \eta\lambda_i)$ smaller and hence improve the speed of convergence. There is a limit to how large η can be made, however. We can permit $(1 - \eta\lambda_i)$ to go negative (which gives oscillating values of α_i), but we must ensure that $|1 - \eta\lambda_i| < 1$ otherwise the α_i values will diverge. This limits the value of η to $\eta < 2/\lambda_{\max}$ where $\lambda_{\max}$ is the largest of the eigenvalues. The rate of convergence, however, is dominated by the smallest eigenvalue, so with η set to its largest permitted value, the convergence along the direction corresponding to the smallest eigenvalue (the long axis of the ellipse in Figure 7.3) will be governed by

$$\left(1 - \frac{2\lambda_{\min}}{\lambda_{\max}}\right) \tag{7.30}$$

Telegram: @uni_k

www.konkur.in

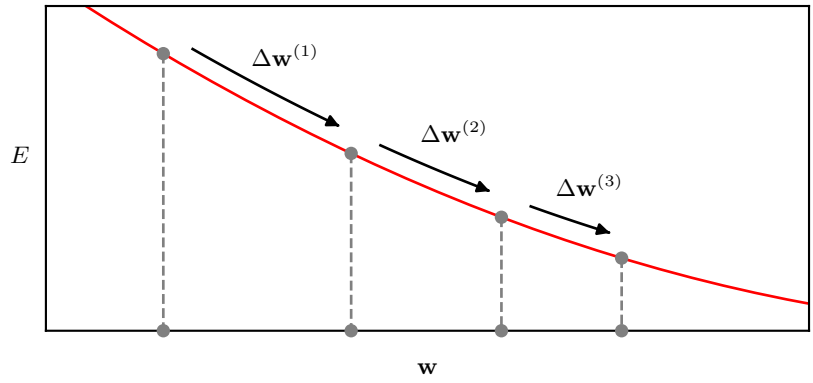


Figure 7.4 With a fixed learning rate parameter, gradient descent down a surface with low curvature leads to successively smaller steps corresponding to linear convergence. In such a situation, the effect of a momentum term is like an increase in the effective learning rate parameter.

where $\lambda_{\min}$ is the smallest eigenvalue. If the ratio $\lambda_{\min}/\lambda_{\max}$ (whose reciprocal is known as the *condition number* of the Hessian) is very small, corresponding to highly elongated elliptical error contours as in Figure 7.3, then progress towards the minimum will be extremely slow.

7.3.1 Momentum

One simple technique for dealing with the problem of widely differing eigenvalues is to add a *momentum* term to the gradient descent formula. This effectively adds inertia to the motion through weight space and smooths out the oscillations depicted in Figure 7.3. The modified gradient descent formula is given by

$$\Delta\mathbf{w}^{(\tau-1)} = -\eta\nabla E\left(\mathbf{w}^{(\tau-1)}\right) + \mu\Delta\mathbf{w}^{(\tau-2)} \tag{7.31}$$

where μ is called the momentum parameter. The weight vector is then updated using (7.15).

To understand the effect of the momentum term, consider first the motion through a region of weight space for which the error surface has relatively low curvature, as indicated in Figure 7.4. If we make the approximation that the gradient is unchanging, then we can apply (7.31) iteratively to a long series of weight updates, and then sum the resulting arithmetic series to give

$$\Delta\mathbf{w} = -\eta\nabla E\{1 + \mu + \mu^2 + \ldots\} \tag{7.32}$$

$$= -\frac{\eta}{1-\mu}\nabla E \tag{7.33}$$

and we see that the result of the momentum term is to increase the effective learning rate from η to $\eta/(1-\mu)$.

By contrast, in a region of high curvature in which gradient descent is oscillatory, as indicated in Figure 7.5, successive contributions from the momentum term will

Telegram: @uni_k

www.konkur.in

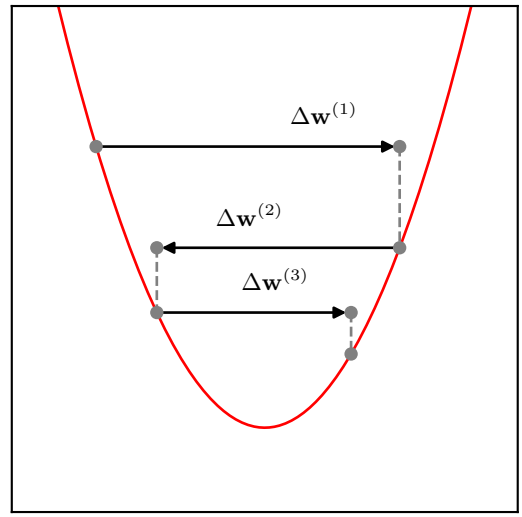


Figure 7.5 For a situation in which successive steps of gradient descent are oscillatory, a momentum term has little influence on the effective value of the learning rate parameter.

tend to cancel and the effective learning rate will be close to η. Thus, the momentum term can lead to faster convergence towards the minimum without causing divergent oscillations. A schematic illustration of the effect of a momentum term is shown in Figure 7.6.

Although the inclusion of momentum can lead to an improvement in the performance of gradient descent, it also introduces a second parameter μ whose value needs to be chosen, in addition to that of the learning rate parameter η. From (7.33) we see that μ should be in the range $0 \leqslant \mu \leqslant 1$. A typical value used in practice is $\mu = 0.9$. Stochastic gradient descent with momentum is summarized in Algorithm 7.3.

The convergence can be further accelerated using a modified version of momentum called *Nesterov momentum* (Nesterov, 2004; Sutskever *et al.*, 2013). In conventional stochastic gradient descent with momentum, we first compute the gradient at the current location then take a step that is amplified by adding momentum from the previous step. With the Nesterov method, we change the order of these and first compute a step based on the previous momentum, then calculate the gradient at this

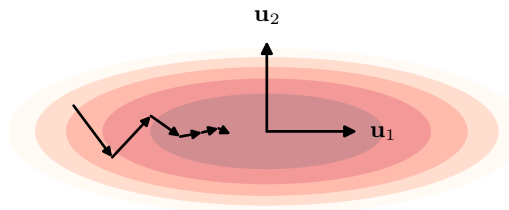


Figure 7.6 Illustration of the effect of adding a momentum term to the gradient descent algorithm, showing the more rapid progress along the valley of the error function, compared with the unmodified gradient descent shown in Figure 7.3.

Telegram: @uni_k

www.konkur.in

Algorithm 7.3: Stochastic gradient descent with momentum

Input: Training set of data points indexed by $n \in \{1, \ldots, N\}$
Batch size B
Error function per mini-batch $E_{n:n+B-1}(\mathbf{w})$
Learning rate parameter η
Momentum parameter μ
Initial weight vector $\mathbf{w}$

Output: Final weight vector $\mathbf{w}$

$n \leftarrow 1$
$\Delta\mathbf{w} \leftarrow \mathbf{0}$
repeat
  $\Delta\mathbf{w} \leftarrow -\eta\nabla E_{n:n+B-1}(\mathbf{w}) + \mu\Delta\mathbf{w}$ `// calculate update term`
  $\mathbf{w} \leftarrow \mathbf{w} + \Delta\mathbf{w}$ `// weight vector update`
  $n \leftarrow n + B$
  if $n > N$ **then**
    shuffle data
    $n \leftarrow 1$
  end if
until convergence
return $\mathbf{w}$

new location to find the update, so that

$$\Delta\mathbf{w}^{(\tau-1)} = -\eta\nabla E\left(\mathbf{w}^{(\tau-1)} + \mu\Delta\mathbf{w}^{(\tau-2)}\right) + \mu\Delta\mathbf{w}^{(\tau-2)}. \tag{7.34}$$

For batch gradient descent, Nesterov momentum can improve the rate of convergence, although for stochastic gradient descent it can be less effective.

7.3.2 Learning rate schedule

Section 7.3.1

In the stochastic gradient descent learning algorithm (7.18), we need to specify a value for the learning rate parameter η. If η is very small then learning will proceed slowly. However, if η is increased too much it can lead to instability. Although some oscillation can be tolerated, it should not be divergent. In practice, the best results are obtained by using a larger value for η at the start of training and then reducing the learning rate over time, so that the value of η becomes a function of the step index τ:

$$\mathbf{w}^{(\tau)} = \mathbf{w}^{(\tau-1)} - \eta^{(\tau-1)}\nabla E_n(\mathbf{w}^{(\tau-1)}). \tag{7.35}$$

Telegram: @uni_k

www.konkur.in

Examples of learning rate schedules include linear, power law, and exponential decay:

$$\eta^{(\tau)} = (1 - \tau/K)\,\eta^{(0)} + (\tau/K)\eta^{(K)} \tag{7.36}$$

$$\eta^{(\tau)} = \eta^{(0)}\,(1 + \tau/s)^{c} \tag{7.37}$$

$$\eta^{(\tau)} = \eta^{(0)} c^{\tau/s} \tag{7.38}$$

where in (7.36) the value of η reduces linearly over K steps, after which its value is held constant at $\eta^{(K)}$. Good values for the hyperparameters $\eta^{(0)}$, $\eta^{(K)}$, K, S, and c must be found empirically. It can be very helpful in practice to monitor the *learning curve* showing how the error function evolves during the gradient descent iteration to ensure that it is decreasing at a suitable rate.

7.3.3 RMSProp and Adam

Section 7.3

We saw that the optimal learning rate depends on the local curvature of the error surface, and moreover that this curvature can vary according to the direction in parameter space. This motivates several algorithms that use different learning rates for each parameter in the network. The values of these learning rates are adjusted automatically during training. Here we review some of the most widely used examples. Note, however, that this intuition really applies only if the principal curvature directions are aligned with the axes in weight space, corresponding to a locally diagonal Hessian matrix, which is unlikely to be the case in practice. Nevertheless, these types of algorithms can be effective and are widely used.

The key idea behind *AdaGrad*, short for 'adaptive gradient', is to reduce each learning rate parameter over time by using the accumulated sum of squares of all the derivatives calculated for that parameter (Duchi, Hazan, and Singer, 2011). Thus, parameters associated with high curvature are reduced most rapidly. Specifically,

$$r_i^{(\tau)} = r_i^{(\tau-1)} + \left(\frac{\partial E(\mathbf{w})}{\partial w_i}\right)^2 \tag{7.39}$$

$$w_i^{(\tau)} = w_i^{(\tau-1)} - \frac{\eta}{\sqrt{r_i^{\tau}} + \delta}\left(\frac{\partial E(\mathbf{w})}{\partial w_i}\right) \tag{7.40}$$

where η is the learning rate parameter, and δ is a small constant, say 10^{-8}, that ensures numerical stability in the event that r_i is close to zero. The algorithm is initialized with $r_i^{(0)} = 0$. Here $E(\mathbf{w})$ is the error function for a particular mini-batch, and the update (7.40) is standard stochastic gradient descent but with a modified learning rate that is specific to each parameter.

One problem with AdaGrad is that it accumulates the squared gradients from the very start of training, and so the associated weight updates can become very small, which can slow down training too much in the later phases. The idea behind the *RMSProp* algorithm, which is short for 'root mean square propagation', is to replace the sum of squared gradients of AdaGrad with an exponentially weighted average

Telegram: @uni_k

www.konkur.in

(Hinton, 2012), giving

$$r_i^{(\tau)} = \beta r_i^{(\tau-1)} + (1-\beta)\left(\frac{\partial E(\mathbf{w})}{\partial w_i}\right)^2 \tag{7.41}$$

$$w_i^{(\tau)} = w_i^{(\tau-1)} - \frac{\eta}{\sqrt{r_i^\tau} + \delta}\left(\frac{\partial E(\mathbf{w})}{\partial w_i}\right) \tag{7.42}$$

where $0 < \beta < 1$ and a typical value is $\beta = 0.9$.

If we combine RMSProp with momentum, we obtain the *Adam* optimization method (Kingma and Ba, 2014) where the name is derived from 'adaptive moments'. Adam stores the momentum for each parameter separately using update equations that consist of exponentially weighted moving averages for both the gradients and the squared gradients in the form

$$s_i^{(\tau)} = \beta_1 s_i^{(\tau-1)} + (1-\beta_1)\left(\frac{\partial E(\mathbf{w})}{\partial w_i}\right) \tag{7.43}$$

$$r_i^{(\tau)} = \beta_2 r_i^{(\tau-1)} + (1-\beta_2)\left(\frac{\partial E(\mathbf{w})}{\partial w_i}\right)^2 \tag{7.44}$$

$$\widehat{s}_i^{(\tau)} = \frac{s_i^{(\tau)}}{1-\beta_1^\tau} \tag{7.45}$$

$$\widehat{r}_i^{\,\tau} = \frac{r_i^\tau}{1-\beta_2^\tau} \tag{7.46}$$

$$w_i^{(\tau)} = w_i^{(\tau-1)} - \eta \frac{\widehat{s}_i^{\,\tau}}{\sqrt{\widehat{r}_i^{\,\tau}} + \delta}. \tag{7.47}$$

Exercise 7.12

Here the factors $1/(1-\beta_1^\tau)$ and $1/(1-\beta_2^\tau)$ correct for a bias introduced by initializing $s_i^{(0)}$ and $r_i^{(0)}$ to zero. Note that the bias goes to zero as τ becomes large, since $\beta_i < 1$, and so in practice this bias correction is sometimes omitted. Typical values for the weighting parameters are $\beta_1 = 0.9$ and $\beta_2 = 0.99$. Adam is the most widely adopted learning algorithm in deep learning and is summarized in Algorithm 7.4.

7.4. Normalization

Normalization of the variables computed during the forward pass through a neural network removes the need for the network to deal with extremely large or extremely small values. Although in principle the weights and biases in a neural network can adapt to whatever values the input and hidden variables take, in practice normalization can be crucial for ensuring effective training. Here we consider three kinds of normalization according to whether we are normalizing across the input data, across mini-batches, or across layers.

Telegram: @uni_k

www.konkur.in

Algorithm 7.4: Adam optimization

Input: Training set of data points indexed by $n \in \{1, \ldots, N\}$
Batch size B
Error function per mini-batch $E_{n:n+B-1}(\mathbf{w})$
Learning rate parameter η
Decay parameters β_1 and β_2
Stabilization parameter δ

Output: Final weight vector $\mathbf{w}$

$n \leftarrow 1$
$\mathbf{s} \leftarrow \mathbf{0}$
$\mathbf{r} \leftarrow \mathbf{0}$
repeat
  Choose a mini-batch at random from $\mathcal{D}$
  $\mathbf{g} = -\nabla E_{n:n+B-1}(\mathbf{w})$ `// evaluate gradient vector`
  $\mathbf{s} \leftarrow \beta_1 \mathbf{s} + (1 - \beta_1)\mathbf{g}$
  $\mathbf{r} \leftarrow \beta_2 \mathbf{r} + (1 - \beta_2)\mathbf{g} \odot \mathbf{g}$ `// element-wise multiply`
  $\widehat{\mathbf{s}} \leftarrow \mathbf{s}/(1 - \beta_1^\tau)$ `// bias correction`
  $\widehat{\mathbf{r}} \leftarrow \mathbf{r}/(1 - \beta_2^\tau)$ `// bias correction`
  $\Delta\mathbf{w} \leftarrow -\eta \dfrac{\widehat{\mathbf{s}}}{\sqrt{\widehat{\mathbf{r}}} + \delta}$ `// element-wise operations`
  $\mathbf{w} \leftarrow \mathbf{w} + \Delta\mathbf{w}$ `// weight vector update`
  $n \leftarrow n + B$
  if $n + B > N$ **then**
    shuffle data
    $n \leftarrow 1$
  end if
until convergence
return $\mathbf{w}$

Telegram: @uni_k

www.konkur.in

7.4.1 Data normalization

Sometimes we encounter data sets in which different input variables span very different ranges. For example, in health data, a patient's height might be measured in meters, such as 1.8m, whereas their blood platelet count might be measured in platelets per microliter, such as 300,000 platelets per μL. Such variations can make gradient descent training much more challenging. Consider a single-layer regression network with two weights in which the two corresponding input variables have very different ranges. Changes in the value of one of the weights produce much larger changes in the output, and hence in the error function, than would similar changes in the other weight. This corresponds to an error surface with very different curvatures along different axes as illustrated in Figure 7.3.

For continuous input variables, it can therefore be very beneficial to re-scale the input values so that they span similar ranges. This is easily done by first evaluating the mean and variance of each input:

$$\mu_i = \frac{1}{N}\sum_{n=1}^{N} x_{ni} \tag{7.48}$$

$$\sigma_i^2 = \frac{1}{N}\sum_{n=1}^{N}(x_{ni} - \mu_i)^2, \tag{7.49}$$

which is a calculation that is performed once, before any training is started. The input values are then re-scaled using

$$\widetilde{x}_{ni} = \frac{x_{ni} - \mu_i}{\sigma_i} \tag{7.50}$$

Exercise 7.14

so that the re-scaled values $\{\widetilde{x}_{ni}\}$ have zero mean and unit variance. Note that the same values of μ_i and σ_i must be used to pre-process any development, validation, or test data to ensure that all inputs are scaled in the same way. Input data normalization is illustrated in Figure 7.7.

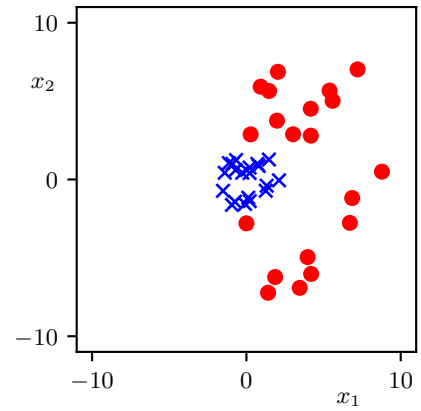


Figure 7.7 Illustration of the effect of input data normalization. The red circles show the original data points for a data set with two variables. The blue crosses show the data set after normalization such that each variable now has zero mean and unit variance across the data set.

Telegram: @uni_k

www.konkur.in

7.4.2 Batch normalization

We have seen the importance of normalizing the input data, and we can apply similar reasoning to the variables in each hidden layer of a deep network. If there is wide variation in the range of activation values in a particular hidden layer, then normalizing those values to have zero mean and unit variance should make the learning problem easier for the next layer. However, unlike normalization of the input values, which can be done once prior to the start of training, normalization of the hidden-unit values will need to be repeated during training every time the weight values are updated. This is called *batch normalization* (Ioffe and Szegedy, 2015).

A further motivation for batch normalization arises from the phenomena of *vanishing gradients* and *exploding gradients*, which occur when we try to train very deep neural networks. From the chain rule of calculus, the gradient of an error function E with respect to a parameter in the first layer of the network is given by

Section 8.1.5

$$\frac{\partial E}{\partial w_i} = \sum_m \cdots \sum_l \sum_j \frac{\partial z_m^{(1)}}{\partial w_i} \cdots \frac{\partial z_j^{(K)}}{\partial z_l^{(K-1)}} \frac{\partial E}{\partial z_j^{(K)}} \tag{7.51}$$

where $z_j^{(k)}$ denotes the activation of node j in layer k, and each of the partial derivatives on the right-hand side of (7.51) represents the elements of the Jacobian matrix for that layer. The product of a large number of such terms will tend towards 0 if most of them have a magnitude < 1 and will tend towards ∞ if most of them have a magnitude > 1. Consequently, as the depth of a network increases, error function gradients can tend to become either very large or very small. Batch normalization largely resolves this issue.

Section 8.1.5

To see how batch normalization is defined, consider a specific layer within a multi-layer network. Each hidden unit in that layer computes a nonlinear function of its input pre-activation $z_i = h(a_i)$, and so we have a choice of whether to normalize the pre-activation values a_i or the activation values z_i. In practice, either approach may be used, and here we illustrate the procedure by normalizing the pre-activations. Because weight values are updated after each mini-batch of examples, we apply the normalization to each mini-batch. Specifically, for a mini-batch of size K, we define

$$\mu_i = \frac{1}{K} \sum_{n=1}^{K} a_{ni} \tag{7.52}$$

$$\sigma_i^2 = \frac{1}{K} \sum_{n=1}^{K} (a_{ni} - \mu_i)^2 \tag{7.53}$$

$$\widehat{a}_{ni} = \frac{a_{ni} - \mu_i}{\sqrt{\sigma_i^2 + \delta}} \tag{7.54}$$

where the summations over $n = 1, \ldots, K$ are taken over the elements of the mini-batch. Here δ is a small constant, introduced to avoid numerical issues in situations where σ_i^2 is small.

By normalizing the pre-activations in a given layer of the network, we reduce the number of degrees of freedom in the parameters of that layer and hence we

Telegram: @uni_k

www.konkur.in

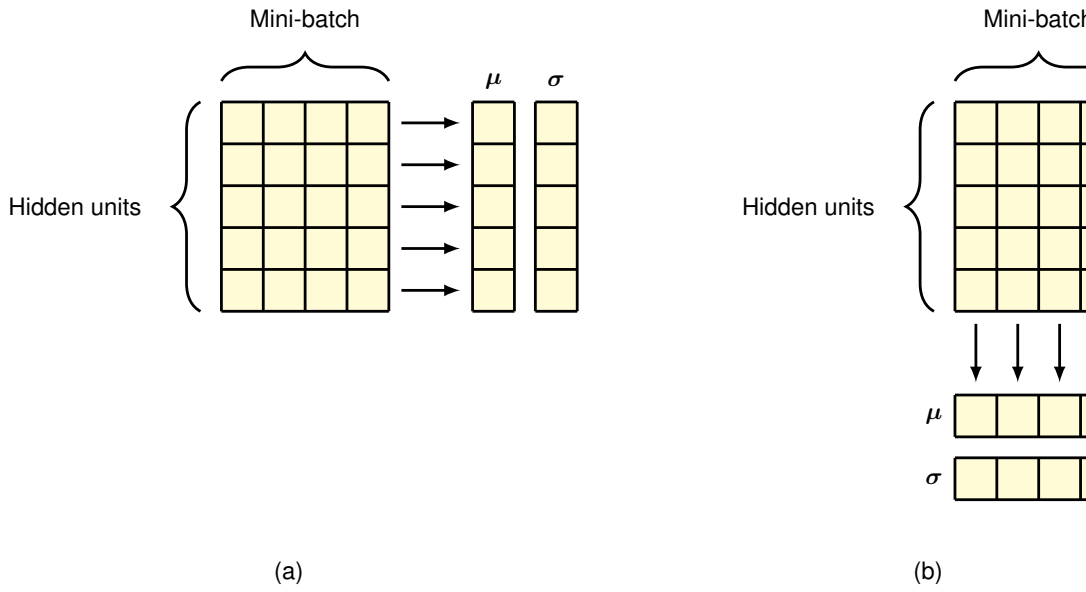


Figure 7.8 Illustration of batch normalization and layer normalization in a neural network. In batch normalization, shown in (a), the mean and variance are computed across the mini-batch separately for each hidden unit. In layer normalization, shown in (b), the mean and variance are computed across the hidden units separately for each data point.

reduce its representational capability. We can compensate for this by re-scaling the pre-activations of the batch to have mean β_i and standard deviation γ_i using

$$\widetilde{a}_{ni} = \gamma_i \widehat{a}_{ni} + \beta_i \tag{7.55}$$

where β_i and γ_i are adaptive parameters that are learned by gradient descent jointly with the weights and biases of the network. These learnable parameters represent a key difference compared to input data normalization.

Section 7.4.1

It might appear that the transformation (7.55) has simply undone the effect of the batch normalization since the mean and variance can now adapt to arbitrary values again. However, the crucial difference is in the way the parameters evolve during training. For the original network, the mean and variance across a mini-batch are determined by a complex function of all the weights and biases in the layer, whereas in the representation given by (7.55), they are determined directly by independent parameters β_i and γ_i, which turn out to be much easier to learn during gradient descent.

Equations (7.52) – (7.55) describe a transformation of the variables that is differentiable with respect to the learnable parameters β_i and γ_i. This can be viewed as an additional layer in the neural network, and so each standard hidden layer can be followed by a batch normalization layer. The structure of the batch-normalization process is illustrated in Figure 7.8.

Once the network is trained and we want to make predictions on new data, we

Telegram: @uni_k

www.konkur.in

no longer have the training mini-batches available, and we cannot determine a mean and variance from just one data example. To solve this, we could in principle evaluate μ_i and σ_i^2 for each layer across the whole training set after we have made the final update to the weights and biases. However, this would involve processing the whole data set just to evaluate these quantities and is therefore usually too expensive. Instead, we compute moving averages throughout the training phase:

$$\overline{\mu}_i^{(\tau)} = \alpha\overline{\mu}_i^{(\tau-1)} + (1-\alpha)\mu_i \tag{7.56}$$

$$\overline{\sigma}_i^{(\tau)} = \alpha\overline{\sigma}_i^{(\tau-1)} + (1-\alpha)\sigma_i \tag{7.57}$$

where $0 \leqslant \alpha \leqslant 1$. These moving averages play no role during training but are used to process new data points during the inference phase.

Although batch normalization is very effective in practice, there is uncertainty as to why it works so well. Batch normalization was originally motivated by noting that updates to weights in earlier layers of the network change the distribution of values seen by later layers, a phenomenon called *internal covariate shift*. However, later studies (Santurkar *et al.*, 2018) suggest that covariate shift is not a significant factor and that the improved training results from an improvement in the smoothness of the error function landscape.

7.4.3 Layer normalization

With batch normalization, if the batch size is too small then the estimates of the mean and variance become too noisy. Also, for very large training sets, the mini-batches may be split across different GPUs, making global normalization across the mini-batch inefficient. An alternative to normalizing across examples within a mini-batch for each hidden unit separately is to normalize across the hidden-unit values for each data point separately. This is known as *layer normalization* (Ba, Kiros, and Hinton, 2016). It was introduced in the context of recurrent neural networks where the distributions change after each time step making batch normalization infeasible. However, it is useful in other architectures such as transformer networks.

Section 12.2.5

Chapter 12

By analogy with batch normalization, we therefore make the following transformation:

$$\mu_n = \frac{1}{M}\sum_{i=1}^{M} a_{ni} \tag{7.58}$$

$$\sigma_n^2 = \frac{1}{M}\sum_{i=1}^{M} (a_{ni} - \mu_i)^2 \tag{7.59}$$

$$\widehat{a}_{ni} = \frac{a_{ni} - \mu_n}{\sqrt{\sigma_n^2 + \delta}} \tag{7.60}$$

where the sums $i = 1, \ldots, M$ are taken over all hidden units in the layer. As with batch normalization, additional learnable mean and standard deviation parameters are introduced for each hidden unit separately in the form (7.55). Note that the same normalization function can be employed during training and during inference, and

Telegram: @uni_k

www.konkur.in

so there is no need to store moving averages. Layer normalization is compared with batch normalization in Figure 7.8.

Exercises

7.1 (⋆) By substituting (7.10) into (7.7) and using (7.8) and (7.9), show that the error function (7.7) can be written in the form (7.11).

7.2 (⋆) Consider a Hessian matrix $\mathbf{H}$ with eigenvector equation (7.8). By setting the vector $\mathbf{v}$ in (7.14) equal to each of the eigenvectors $\mathbf{u}_i$ in turn, show that $\mathbf{H}$ is positive definite if, and only if, all its eigenvalues are positive.

7.3 (⋆⋆) By considering the local Taylor expansion (7.7) of an error function about a stationary point $\mathbf{w}^\star$, show that the necessary and sufficient condition for the stationary point to be a local minimum of the error function is that the Hessian matrix $\mathbf{H}$, defined by (7.5) with $\widehat{\mathbf{w}} = \mathbf{w}^\star$, is positive definite.

7.4 (⋆⋆) Consider a linear regression model with a single input variable x and a single output variable y of the form

$$y(x, w, b) = wx + b \tag{7.61}$$

together with a sum-of-squares error function given by

$$E(w, b) = \frac{1}{2}\sum_{n=1}^{N} \{y(x_n, w, b) - t_n\}^2 . \tag{7.62}$$

Derive expressions for the elements of the 2×2 Hessian matrix given by the second derivatives of the error function with respect to the weight parameter w and bias parameter b. Show that the trace and the determinant of this Hessian are both positive. Since the trace represents the sum of the eigenvalue and the determinant corresponds to the product of the eigenvalues, then both eigenvalues are positive and hence the stationary point of the error function is a minimum.

Appendix A

7.5 (⋆⋆) Consider a single-layer classification model with a single input variable x and a single output variable y of the form

$$y(x, w, b) = \sigma(wx + b) \tag{7.63}$$

where $\sigma(\cdot)$ is the logistic sigmoid function defined by (5.42) together with a cross-entropy error function given by

$$E(w, b) = \sum_{n=1}^{N} \{t_n \ln y(x_n, w, b) + (1 - t_n) \ln(1 - y(x_n, w, b))\} . \tag{7.64}$$

Derive expressions for the elements of the 2×2 Hessian matrix given by the second derivatives of the error function with respect to the weight parameter w and bias parameter b. Show that the trace and the determinant of this Hessian are both positive.

Telegram: @uni_k

www.konkur.in

Appendix A

Since the trace represents the sum of the eigenvalue and the determinant corresponds to the product of the eigenvalues, then both eigenvalues are positive and hence the stationary point of the error function is a minimum.

7.6 (⋆⋆) Consider a quadratic error function defined by (7.7) in which the Hessian matrix $\mathbf{H}$ has an eigenvalue equation given by (7.8). Show that the contours of constant error are ellipses whose axes are aligned with the eigenvectors $\mathbf{u}_i$ with lengths that are inversely proportional to the square roots of the corresponding eigenvalues λ_i.

7.7 (⋆) Show that, as a consequence of the symmetry of the Hessian matrix $\mathbf{H}$, the number of independent elements in the quadratic error function (7.3) is given by $W(W+3)/2$.

7.8 (⋆) Consider a set of values $x_1, \ldots, x_N$ drawn from a distribution with mean μ and variance σ^2, and define the sample mean to be

$$\overline{x} = \frac{1}{N}\sum_{n=1}^{N} x_n. \tag{7.65}$$

Show that the expectation of the squared error $(\overline{x} - \mu)^2$ with respect to the distribution from which the data is drawn is given by σ^2/N. This shows that the RMS error in the sample mean is given by $\sigma/\sqrt{N}$, which decreases relatively slowly as the sample size N increases.

7.9 (⋆⋆) Consider a layered network that computes the functions (7.19) and (7.20) in layer l. Suppose we initialize the weights using a Gaussian $\mathcal{N}(0, \epsilon^2)$, and suppose that the outputs $z_j^{(l-1)}$ of the units in layer $l-1$ have variance λ^2. By using the form of the ReLU activation function, show that the mean and variance of the outputs in layer l are given by (7.21) and (7.22), respectively. Hence, show that if we want the units in layer l also to have pre-activations with variance λ^2 then the value of ϵ should be given by (7.23).

7.10 (⋆⋆) By making use of (7.7), (7.8), and (7.10), derive the results (7.24) and (7.25), which express the gradient vector and a general weight update, as expansions in the eigenvectors of the Hessian matrix. Use these results, together with the eigenvector orthonormality relation (7.9) and the batch gradient descent formula (7.16), to derive the result (7.26) for the batch gradient descent update expressed in terms of the coefficients $\{\alpha_i\}$.

7.11 (⋆) Consider a smoothly varying error surface with low curvature such that the gradient varies only slowly with position. Show that, for small values of the learning rate and momentum parameters, the Nesterov momentum gradient update defined by (7.34) is equivalent to the standard gradient descent with momentum defined by (7.31).

7.12 (⋆⋆) Consider a sequence of values $\{x_1, \ldots, x_N\}$ of some variable x, and suppose we compute an exponentially weighted moving average using the formula

$$\mu_n = \beta\mu_{n-1} + (1-\beta)x_n \tag{7.66}$$

Telegram: @uni_k

www.konkur.in

where $0 \leqslant \beta \leqslant 1$. By making use of the following result for the sum of a finite geometric series

$$\sum_{k=1}^{n} \beta^{k-1} = \frac{1-\beta^n}{1-\beta} \tag{7.67}$$

show that if the sequence of averages is initialized using $\mu_0 = 0$, then the estimators are biased and that the bias can be corrected using

$$\widehat{\mu}_n = \frac{\mu_n}{1-\beta^n}. \tag{7.68}$$

7.13 (⋆) In gradient descent, the weight vector $\mathbf{w}$ is updated by taking a step in weight space in the direction of the negative gradient governed by a learning rate parameter η. Suppose instead that we choose a direction $\mathbf{d}$ in weight space along which we minimize the error function, given the current weight vector $\mathbf{w}^{(\tau)}$. This involves minimizing the quantity

$$E(\mathbf{w}^{(\tau)} + \lambda \mathbf{d}) \tag{7.69}$$

as a function of λ to give a value $\lambda^\star$ corresponding to a new weight vector $\mathbf{w}^{(\tau+1)}$. Show that the gradient of $E(\mathbf{w})$ at $\mathbf{w}^{(\tau+1)}$ is orthogonal to the vector $\mathbf{d}$. This is known as a 'line search' method and it forms the basis for a variety of numerical optimization algorithms (Bishop, 1995b).

7.14 (⋆) Show that the renormalized input variables defined by (7.50), where μ_i is defined by (7.48) and σ_i^2 is defined by (7.49), have zero mean and unit variance.

Telegram: @uni_k

www.konkur.in



8 Backpropagation

Our goal in this chapter is to find an efficient technique for evaluating the gradient of an error function $E(\mathbf{w})$ for a feed-forward neural network. We will see that this can be achieved using a local message-passing scheme in which information is sent backwards through the network and is known as *error backpropagation*, or sometimes simply as *backprop*.

Historically, the backpropagation equations would have been derived by hand and then implemented in software alongside the forward propagation equations, with both steps taking time and being prone to mistakes. Modern neural network software environments, however, allow virtually any derivatives of interest to be calculated efficiently with only minimal effort beyond that of coding up the original network function. This idea, called *automatic differentiation*, plays a key role in modern deep learning. However, it is valuable to understand how the calculations are performed so that we are not relying on 'black box' software solutions. In this chapter we

Section 8.2

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024
C. M. Bishop, H. Bishop, *Deep Learning*, https://doi.org/10.1007/978-3-031-45468-4_8

Telegram: @uni_k

www.konkur.in

therefore explain the key concepts of backpropagation, and explore the framework of automatic differentiation in detail.

Note that the term 'backpropagation' is used in the neural computing literature in a variety of different ways. For instance, a feed-forward architecture may be called a backpropagation network. Also the term 'backpropagation' is sometimes used to describe the end-to-end training procedure for a neural network including the gradient descent parameter updates. In this book we will use 'backpropagation' specifically to describe the computational procedure used in the numerical evaluation of derivatives such as the gradient of the error function with respect to the weights and biases of a network. This procedure can also be applied to the evaluation of other important derivatives such as the Jacobian and Hessian matrices.

8.1. Evaluation of Gradients

We now derive the backpropagation algorithm for a general network having arbitrary feed-forward topology, arbitrary differentiable nonlinear activation functions, and a broad class of error function. The resulting formulae will then be illustrated using a simple layered network structure having a single layer of sigmoidal hidden units together with a sum-of-squares error.

Many error functions of practical interest, for instance those defined by maximum likelihood for a set of i.i.d. data, comprise a sum of terms, one for each data point in the training set, so that

$$E(\mathbf{w}) = \sum_{n=1}^{N} E_n(\mathbf{w}). \tag{8.1}$$

Here we will consider the problem of evaluating $\nabla E_n(\mathbf{w})$ for one such term in the error function. This may be used directly for stochastic gradient descent, or the results could be accumulated over a set of training data points for batch or mini-batch methods.

8.1.1 Single-layer networks

Consider first a simple linear model in which the outputs y_k are linear combinations of the input variables x_i so that

$$y_k = \sum_i w_{ki} x_i \tag{8.2}$$

together with a sum-of-squares error function that, for a particular input data point n, takes the form

$$E_n = \frac{1}{2} \sum_k (y_{nk} - t_{nk})^2 \tag{8.3}$$

Telegram: @uni_k

www.konkur.in

where $y_{nk} = y_k(\mathbf{x}_n, \mathbf{w})$, and t_{nk} is the associated target value. The gradient of this error function with respect to a weight w_{ji} is given by

$$\frac{\partial E_n}{\partial w_{ji}} = (y_{nj} - t_{nj})x_{ni}. \tag{8.4}$$

This can be interpreted as a 'local' computation involving the product of an 'error signal' $y_{nj} - t_{nj}$ associated with the output end of the link w_{ji} and the variable x_{ni} associated with the input end of the link. In Section 5.4.3, we saw how a similar formula arises with the logistic-sigmoid activation function together with the cross-entropy error function and similarly for the softmax activation function together with its matching multivariate cross-entropy error function. We will now see how this simple result extends to the more complex setting of multilayer feed-forward networks.

8.1.2 General feed-forward networks

In general, a feed-forward network consists of a set of units each of which computes a weighted sum of its inputs:

$$a_j = \sum_i w_{ji} z_i \tag{8.5}$$

where z_i is either the activation of another unit or an input unit that sends a connection to unit j, and w_{ji} is the weight associated with that connection. Biases can be included in this sum by introducing an extra unit, or input, with activation fixed at $+1$, and so we do not need to deal with biases explicitly. The sum in (8.5), known as a pre-activation, is transformed by a nonlinear activation function $h(\cdot)$ to give the activation z_j of unit j in the form

Section 6.2

$$z_j = h(a_j). \tag{8.6}$$

Note that one or more of the variables z_i in the sum in (8.5) could be an input, and similarly, the unit j in (8.6) could be an output.

For each data point in the training set, we will suppose that we have supplied the corresponding input vector to the network and calculated the activations of all the hidden and output units in the network by successive application of (8.5) and (8.6). This process is called *forward propagation* because it can be regarded as a forward flow of information through the network.

Now consider the evaluation of the derivative of E_n with respect to a weight w_{ji}. The outputs of the various units will depend on the particular input data point n. However, to keep the notation uncluttered, we will omit the subscript n from the network variables. First note that E_n depends on the weight w_{ji} only via the summed input a_j to unit j. We can therefore apply the chain rule for partial derivatives to give

$$\frac{\partial E_n}{\partial w_{ji}} = \frac{\partial E_n}{\partial a_j} \frac{\partial a_j}{\partial w_{ji}}. \tag{8.7}$$

We now introduce a useful notation:

$$\delta_j \equiv \frac{\partial E_n}{\partial a_j} \tag{8.8}$$

Telegram: @uni_k

www.konkur.in

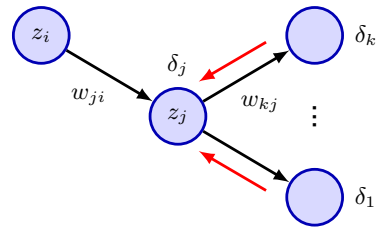


Figure 8.1 Illustration of the calculation of δ_j for hidden unit j by backpropagation of the δ's from those units k to which unit j sends connections. The black arrows denote the direction of information flow during forward propagation, and the red arrows indicate the backward propagation of error information.

where the δ's are often referred to as *errors* for reasons we will see shortly. Using (8.5), we can write

$$\frac{\partial a_j}{\partial w_{ji}} = z_i. \tag{8.9}$$

Substituting (8.8) and (8.9) into (8.7), we then obtain

$$\frac{\partial E_n}{\partial w_{ji}} = \delta_j z_i. \tag{8.10}$$

Equation (8.10) tells us that the required derivative is obtained simply by multiplying the value of δ for the unit at the output end of the weight by the value of z for the unit at the input end of the weight (where $z = 1$ for a bias). Note that this takes the same form as that found for the simple linear model in (8.4). Thus, to evaluate the derivatives, we need calculate only the value of δ_j for each hidden and output unit in the network and then apply (8.10).

As we have seen already, for the output units, we have

$$\delta_k = y_k - t_k \tag{8.11}$$

Section 5.4.6 provided we are using the canonical link as the output-unit activation function. To evaluate the δ's for hidden units, we again make use of the chain rule for partial derivatives:

$$\delta_j \equiv \frac{\partial E_n}{\partial a_j} = \sum_k \frac{\partial E_n}{\partial a_k} \frac{\partial a_k}{\partial a_j} \tag{8.12}$$

where the sum runs over all units k to which unit j sends connections. The arrangement of units and weights is illustrated in Figure 8.1. Note that the units labelled k include other hidden units and/or output units. In writing down (8.12), we are making use of the fact that variations in a_j give rise to variations in the error function only through variations in the variables a_k.

If we now substitute the definition of δ_j given by (8.8) into (8.12) and make use *Exercise 8.1* of (8.5) and (8.6), we obtain the following *backpropagation* formula:

$$\delta_j = h'(a_j) \sum_k w_{kj} \delta_k, \tag{8.13}$$

which tells us that the value of δ for a particular hidden unit can be obtained by propagating the δ's backwards from units higher up in the network, as illustrated

Telegram: @uni_k

www.konkur.in

Algorithm 8.1: Backpropagation

Input: Input vector $\mathbf{x}_n$
Network parameters $\mathbf{w}$
Error function $E_n(\mathbf{w})$ for input x_n
Activation function $h(a)$

Output: Error function derivatives $\{\partial E_n/\partial w_{ji}\}$

`// Forward propagation`

for $j \in$ all hidden and output units **do**

$a_j \leftarrow \sum_i w_{ji} z_i$ `// {`z_i`} includes inputs {`x_i`}`

$z_j \leftarrow h(a_j)$ `// activation function`

end for

`// Error evaluation`

for $k \in$ all output units **do**

$\delta_k \leftarrow \dfrac{\partial E_n}{\partial a_k}$ `// compute errors`

end for

`// Backward propagation, in reverse order`

for $j \in$ all hidden units **do**

$\delta_j \leftarrow h'(a_j) \sum_k w_{kj} \delta_k$ `// recursive backward evaluation`

$\dfrac{\partial E_n}{\partial w_{ji}} \leftarrow \delta_j z_i$ `// evaluate derivatives`

end for

return $\left\{ \dfrac{\partial E_n}{\partial w_{ji}} \right\}$

in Figure 8.1. Note that the summation in (8.13) is taken over the first index on w_{kj} (corresponding to backward propagation of information through the network), whereas in the forward propagation equation (8.5), it is taken over the second index. Because we already know the values of the δ's for the output units, it follows that by recursively applying (8.13), we can evaluate the δ's for all the hidden units in a feed-forward network, regardless of its topology. The backpropagation procedure is summarized in Algorithm 8.1.

Exercise 8.2

For batch methods, the derivative of the total error E can then be obtained by repeating the above steps for each data point in the training set and then summing over all data points in the batch or mini-batch:

$$\frac{\partial E}{\partial w_{ji}} = \sum_n \frac{\partial E_n}{\partial w_{ji}}. \tag{8.14}$$

Telegram: @uni_k

www.konkur.in

In the above derivation we have implicitly assumed that each hidden or output unit in the network has the same activation function $h(\cdot)$. However, the derivation is easily generalized to allow different units to have individual activation functions, simply by keeping track of which form of $h(\cdot)$ goes with which unit.

8.1.3 A simple example

The above derivation of the backpropagation procedure allowed for general forms for the error function, the activation functions, and the network topology. To illustrate the application of this algorithm, we consider a two-layer network of the form illustrated in Figure 6.9, together with a sum-of-squares error. The output units have linear activation functions, so that $y_k = a_k$, and the hidden units have sigmoidal activation functions given by

$$h(a) \equiv \tanh(a) \tag{8.15}$$

where $\tanh(a)$ is defined by (6.14). A useful feature of this function is that its derivative can be expressed in a particularly simple form:

$$h'(a) = 1 - h(a)^2. \tag{8.16}$$

We also consider a sum-of-squares error function, so that for data point n the error is given by

$$E_n = \frac{1}{2}\sum_{k=1}^{K}(y_k - t_k)^2 \tag{8.17}$$

where y_k is the activation of output unit k, and t_k is the corresponding target value for a particular input vector $\mathbf{x}_n$.

For each data point in the training set in turn, we first perform a forward propagation using

$$a_j = \sum_{i=0}^{D} w_{ji}^{(1)} x_i \tag{8.18}$$

$$z_j = \tanh(a_j) \tag{8.19}$$

$$y_k = \sum_{j=0}^{M} w_{kj}^{(2)} z_j \tag{8.20}$$

where D is the dimensionality of the input vector $\mathbf{x}$ and M is the total number of hidden units. Also we have used $x_0 = z_0 = 1$ to allow bias parameters to be included in the weights. Next we compute the δ's for each output unit using

$$\delta_k = y_k - t_k. \tag{8.21}$$

Then, we backpropagate these errors to obtain δ's for the hidden units using

$$\delta_j = (1 - z_j^2)\sum_{k=1}^{K} w_{kj}^{(2)}\delta_k, \tag{8.22}$$

Telegram: @uni_k

www.konkur.in

which follows from (8.13) and (8.16). Finally, the derivatives with respect to the first-layer and second-layer weights are given by

$$\frac{\partial E_n}{\partial w_{ji}^{(1)}} = \delta_j x_i, \qquad \frac{\partial E_n}{\partial w_{kj}^{(2)}} = \delta_k z_j. \tag{8.23}$$

8.1.4 Numerical differentiation

One of the most important aspects of backpropagation is its computational efficiency. To understand this, let us examine how the number of compute operations required to evaluate the derivatives of the error function scales with the total number W of weights and biases in the network.

A single evaluation of the error function (for a given input data point) would require $\mathcal{O}(W)$ operations, for sufficiently large W. This follows because, except for a network with very sparse connections, the number of weights is typically much greater than the number of units, and so the bulk of the computational effort in forward propagation arises from evaluation of the sums in (8.5), with the evaluation of the activation functions representing a small overhead. Each term in the sum in (8.5) requires one multiplication and one addition, leading to an overall computational cost that is $\mathcal{O}(W)$.

An alternative approach to backpropagation for computing the derivatives of the error function is to use finite differences. This can be done by perturbing each weight in turn and approximating the derivatives by using the expression

$$\frac{\partial E_n}{\partial w_{ji}} = \frac{E_n(w_{ji} + \epsilon) - E_n(w_{ji})}{\epsilon} + \mathcal{O}(\epsilon) \tag{8.24}$$

where $\epsilon \ll 1$. In a software simulation, the accuracy of the approximation to the derivatives can be improved by making ϵ smaller, until numerical round-off problems arise. The accuracy of the finite differences method can be improved significantly by using symmetrical *central differences* of the form

$$\frac{\partial E_n}{\partial w_{ji}} = \frac{E_n(w_{ji} + \epsilon) - E_n(w_{ji} - \epsilon)}{2\epsilon} + \mathcal{O}(\epsilon^2). \tag{8.25}$$

Exercise 8.3

In this case, the $\mathcal{O}(\epsilon)$ corrections cancel, as can be verified by a Taylor expansion of the right-hand side of (8.25), and so the residual corrections are $\mathcal{O}(\epsilon^2)$. Note, however, that the number of computational steps is roughly doubled compared with (8.24). Figure 8.2 shows a plot of the error between a numerical evaluation of a gradient using both finite differences (8.24) and central differences (8.25) versus the analytical result, as a function of the value of the step size ϵ.

The main problem with numerical differentiation is that the highly desirable $\mathcal{O}(W)$ scaling has been lost. Each forward propagation requires $\mathcal{O}(W)$ steps, and there are W weights in the network each of which must be perturbed individually, so that the overall computational cost is $\mathcal{O}(W^2)$.

However, numerical differentiation can play a useful role in practice, because a comparison of the derivatives calculated from a direct implementation of backpropagation, or from automatic differentiation, with those obtained using central differences provides a powerful check on the correctness of the software.

Telegram: @uni_k

www.konkur.in

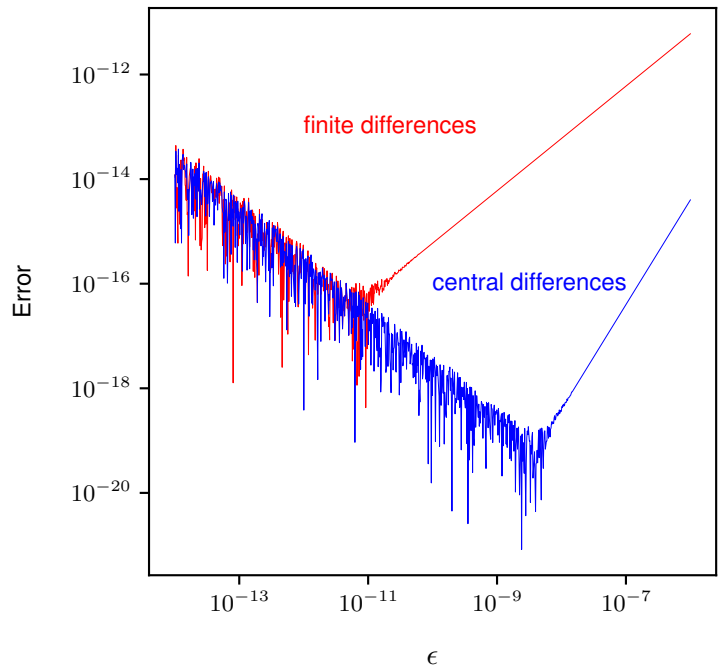


Figure 8.2 The red curve shows a plot of the error between the numerical evaluation of a gradient using finite differences (8.24) and the analytical result, as a function of ϵ. As ϵ decreases, the plot initially shows a linear decrease in error, and this represents a power law behaviour since the axes are logarithmic. The slope of this line is 1 which shows that this error behaves like $\mathcal{O}(\epsilon)$. At some point the evaluated gradient reaches the limit of numerical round-off and further reduction in ϵ leads to a noisy line, which again follows a power law but where the error now increases with decreasing ϵ. The blue curve shows the corresponding result for central differences (8.25). We see a much smaller error compared to finite differences, and the slope of the line is 2 which shows that the error is $\mathcal{O}(\epsilon^2)$.

8.1.5 The Jacobian matrix

We have seen how the derivatives of an error function with respect to the weights can be obtained by propagating errors backwards through the network. Backpropagation can also be used to calculate other derivatives. Here we consider the evaluation of the *Jacobian* matrix, whose elements are given by the derivatives of the network outputs with respect to the inputs:

$$J_{ki} \equiv \frac{\partial y_k}{\partial x_i} \tag{8.26}$$

where each such derivative is evaluated with all other inputs held fixed. Jacobian matrices play a useful role in systems built from a number of distinct modules, as illustrated in Figure 8.3. Each module can comprise a fixed or learnable function, which can be linear or nonlinear, so long as it is differentiable.

Suppose we wish to minimize an error function E with respect to the parameter w in Figure 8.3. The derivative of the error function is given by

$$\frac{\partial E}{\partial w} = \sum_{k,j} \frac{\partial E}{\partial y_k} \frac{\partial y_k}{\partial z_j} \frac{\partial z_j}{\partial w} \tag{8.27}$$

in which the Jacobian matrix for the red module in Figure 8.3 appears as the middle term on the right-hand side.

Because the Jacobian matrix provides a measure of the local sensitivity of the outputs to changes in each of the input variables, it also allows any known errors Δx_i

Telegram: @uni_k

www.konkur.in

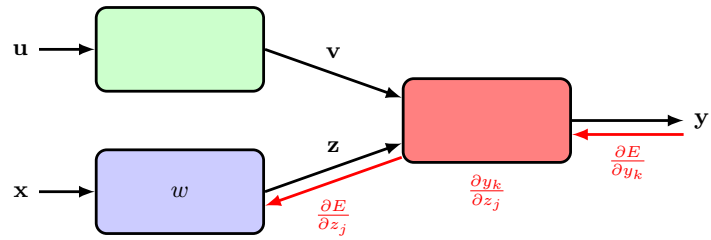


Figure 8.3 Illustration of a modular deep learning architecture in which the Jacobian matrix can be used to back-propagate error signals from the outputs through to earlier modules in the system.

associated with the inputs to be propagated through the trained network to estimate their contribution Δy_k to the errors at the outputs, through the relation

$$\Delta y_k \simeq \sum_i \frac{\partial y_k}{\partial x_i} \Delta x_i, \tag{8.28}$$

which assumes that the $|\Delta x_i|$ are small. In general, the network mapping represented by a trained neural network will be nonlinear, and so the elements of the Jacobian matrix will not be constants but will depend on the particular input vector used. Thus, (8.28) is valid only for small perturbations of the inputs, and the Jacobian itself must be re-evaluated for each new input vector.

The Jacobian matrix can be evaluated using a backpropagation procedure that is like the one derived earlier for evaluating the derivatives of an error function with respect to the weights. We start by writing the element J_{ki} in the form

$$\begin{aligned} J_{ki} = \frac{\partial y_k}{\partial x_i} &= \sum_j \frac{\partial y_k}{\partial a_j} \frac{\partial a_j}{\partial x_i} \\ &= \sum_j w_{ji} \frac{\partial y_k}{\partial a_j} \end{aligned} \tag{8.29}$$

where we have made use of (8.5). The sum in (8.29) runs over all units j to which the input unit i sends connections (for example, over all units in the first hidden layer in the layered topology considered earlier). We now write down a recursive backpropagation formula for the derivatives $\partial y_k / \partial a_j$:

$$\begin{aligned} \frac{\partial y_k}{\partial a_j} &= \sum_l \frac{\partial y_k}{\partial a_l} \frac{\partial a_l}{\partial a_j} \\ &= h'(a_j) \sum_l w_{lj} \frac{\partial y_k}{\partial a_l} \end{aligned} \tag{8.30}$$

where the sum runs over all units l to which unit j sends connections (corresponding to the first index of w_{lj}). Again, we have made use of (8.5) and (8.6). This back-propagation starts at the output units, for which the required derivatives can be found directly from the functional form of the output-unit activation function. For linear output units, we have

$$\frac{\partial y_k}{\partial a_l} = \delta_{kl} \tag{8.31}$$

Telegram: @uni_k

www.konkur.in

where δ_{kl} are the elements of the identity matrix and are defined by

$$\delta_{kl} = \begin{cases} 1, & \text{if } k = l, \\ 0, & \text{otherwise.} \end{cases} \tag{8.32}$$

Section 3.4 If we have individual logistic sigmoid activation functions at each output unit, then

$$\frac{\partial y_k}{\partial a_l} = \delta_{kl}\sigma'(a_l) \tag{8.33}$$

Section 3.4 whereas for softmax outputs, we have

$$\frac{\partial y_k}{\partial a_l} = \delta_{kl}y_k - y_k y_l. \tag{8.34}$$

We can summarize the procedure for calculating the Jacobian matrix as follows. Apply the input vector corresponding to the point in input space at which the Jacobian matrix is to be evaluated, and forward propagate in the usual way to obtain the states of all the hidden and output units in the network. Next, for each row k of the Jacobian matrix, corresponding to the output unit k, backpropagate using the recursive relation (8.30), starting with (8.31), (8.33) or (8.34), for all the hidden units in the network. Finally, use (8.29) for the backpropagation to the inputs. The Jacobian can also be evaluated using an alternative *forward* propagation formalism, which can be derived in an analogous way to the backpropagation approach given here.

Exercise 8.5

Again, the implementation of such algorithms can be checked using numerical differentiation in the form

$$\frac{\partial y_k}{\partial x_i} = \frac{y_k(x_i + \epsilon) - y_k(x_i - \epsilon)}{2\epsilon} + \mathcal{O}(\epsilon^2), \tag{8.35}$$

which involves $2D$ forward propagation passes for a network having D inputs and therefore requires $\mathcal{O}(DW)$ steps in total.

8.1.6 The Hessian matrix

We have shown how backpropagation can be used to obtain the first derivatives of an error function with respect to the weights in the network. Backpropagation can also be used to evaluate the second derivatives of the error, which are given by

$$\frac{\partial^2 E}{\partial w_{ji}\partial w_{lk}}. \tag{8.36}$$

It is often convenient to consider all the weight and bias parameters as elements w_i of a single vector, denoted $\mathbf{w}$, in which case the second derivatives form the elements H_{ij} of the *Hessian* matrix $\mathbf{H}$:

$$H_{ij} = \frac{\partial^2 E}{\partial w_i \partial w_j} \tag{8.37}$$

Telegram: @uni_k

www.konkur.in

where $i, j \in \{1, \ldots, W\}$ and W is the total number of weights and biases. The Hessian matrix arises in several nonlinear optimization algorithms used for training neural networks based on considerations of the second-order properties of the error surface (Bishop, 2006). It also plays a role in some Bayesian treatments of neural networks (MacKay, 1992; Bishop, 2006) and has been used to reduce the precision of the weights in large language models to lessen their memory footprint (Shen *et al.*, 2019).

An important consideration for many applications of the Hessian is the efficiency with which it can be evaluated. If there are W parameters (weights and biases) in the network, then the Hessian matrix has dimensions $W \times W$ and so the computational effort needed to evaluate the Hessian will scale like $\mathcal{O}(W^2)$ for each point in the data set. Extension of the backpropagation procedure (Bishop, 1992) allows the Hessian matrix to be evaluated efficiently with a scaling that is indeed $\mathcal{O}(W^2)$. Sometimes, we do not need the Hessian matrix explicitly but only the product $\mathbf{v}^{\mathrm{T}}\mathbf{H}$ of the Hessian with some vector $\mathbf{v}$, and this product can be calculated efficiently in $\mathcal{O}(W)$ steps using an extension of backpropagation (Møller, 1993; Pearlmutter, 1994).

Exercise 8.6

Since neural networks may contain millions or even billions of parameters, evaluating, or even just storing, the full Hessian matrix for many models is infeasible. Evaluating the inverse of the Hessian is even more demanding as this has $\mathcal{O}(W^3)$ computational scaling. Consequently there is interest in finding effective approximations to the full Hessian.

One approximation involves simply evaluating only the diagonal elements of the Hessian and implicitly setting the off-diagonal elements to zero. This requires $\mathcal{O}(W)$ storage and allows the inverse to be evaluated in $\mathcal{O}(W)$ steps but still requires $\mathcal{O}(W^2)$ computation (Ricotti, Ragazzini, and Martinelli, 1988), although with further approximation this can be reduced to $\mathcal{O}(W)$ steps (Becker and LeCun, 1989; LeCun, Denker, and Solla, 1990). In practice, however, the Hessian generally has significant off-diagonal terms, and so this approximation must be treated with care.

A more convincing approach, known as the *outer product approximation*, is obtained as follows. Consider a regression application using a sum-of-squares error function of the form

$$E = \frac{1}{2}\sum_{n=1}^{N}(y_n - t_n)^2 \tag{8.38}$$

where we have considered a single output to keep the notation simple (the extension to several outputs is straightforward). We can then write the Hessian matrix in the form

Exercise 8.8

$$\mathbf{H} = \nabla\nabla E = \sum_{n=1}^{N}\nabla y_n(\nabla y_n)^{\mathrm{T}} + \sum_{n=1}^{N}(y_n - t_n)\nabla\nabla y_n \tag{8.39}$$

where ∇ denotes the gradient with respect to $\mathbf{w}$. If the network has been trained on the data set and its outputs y_n are very close to the target values t_n, then the final term in (8.39) will be small and can be neglected. More generally, however, it may be appropriate to neglect this term based on the following argument. Recall from Section 4.2 that the optimal function that minimizes a sum-of-squares loss is

Telegram: @uni_k

www.konkur.in

the conditional average of the target data. The quantity $(y_n - t_n)$ is then a random variable with zero mean. If we assume that its value is uncorrelated with the value of the second derivative term on the right-hand side of (8.39), then the whole term will average to zero in the summation over n.

Exercise 8.9

By neglecting the second term in (8.39), we arrive at the *Levenberg–Marquardt* approximation, also known as the *outer product* approximation because the Hessian matrix is built up from a sum of outer products of vectors, given by

$$\mathbf{H} \simeq \sum_{n=1}^{N} \nabla a_n \nabla a_n^{\mathrm{T}}. \tag{8.40}$$

Evaluating the outer product approximation for the Hessian is straightforward as it involves only first derivatives of the error function, which can be evaluated efficiently in $\mathcal{O}(W)$ steps using standard backpropagation. The elements of the matrix can then be found in $\mathcal{O}(W^2)$ steps by simple multiplication. It is important to emphasize that this approximation is likely to be valid only for a network that has been trained appropriately, and that for a general network mapping, the second derivative terms on the right-hand side of (8.39) will typically not be negligible.

Exercise 8.10

For a cross-entropy error function for a network with logistic-sigmoid output-unit activation functions, the corresponding approximation is given by

$$\mathbf{H} \simeq \sum_{n=1}^{N} y_n(1 - y_n) \nabla a_n \nabla a_n^{\mathrm{T}}. \tag{8.41}$$

Exercise 8.11
Exercise 8.12

An analogous result can be obtained for multi-class networks having softmax output-unit activation functions. The outer product approximation can also be used to develop an efficient sequential procedure for approximating the inverse of a Hessian (Hassibi and Stork, 1993).

8.2. Automatic Differentiation

Section 7.2.1

We have seen the importance of using gradient information to train neural networks efficiently. There are essentially four ways in which the gradient of a neural network error function can be evaluated.

The first approach, which formed the mainstay of neural networks for many years, is to derive the backpropagation equations by hand and then to implement them explicitly in software. If this is done carefully it results in efficient code that gives precise results that are accurate to numerical precision. However, the process of deriving the equations as well as the process of coding them both take time and are prone to errors. It also results in some redundancy in the code because the forward propagation equations are coded separately from the backpropagation equations. As these often involve duplicated calculations, then if the model is altered, both the forward and backward implementations need to be changed in unison. This effort

Telegram: @uni_k

www.konkur.in

can easily become a limitation on how quickly and effectively different architectures can be explored empirically.

Section 8.1.4

A second approach is to evaluate the gradients numerically using finite differences. This requires only a software implementation of the forward propagation equations. One problem with numerical differentiation is that it has limited computational accuracy, although this is unlikely to be an issue for network training as we may be using stochastic gradient descent in which each evaluation is only a very noisy estimate of the local gradient. The main drawback of this approach is that it scales poorly with the size of the network. However, the technique is useful for debugging other approaches, because the gradients are evaluated using only the forward propagation code and so can be used to confirm the correctness of backpropagation or other code used to evaluate gradients.

A third approach is called *symbolic differentiation* and makes use of specialist software to automate the analytical manipulations that are done by hand in the first approach. This process is an example of *computer algebra* or *symbolic computation* and involves the automatic application of the rules of calculus, such as the chain rule, in a completely mechanistic process. The resulting expressions are then implemented in standard software. An obvious advantage of this approach is that it avoids human error in the manual derivation of the backpropagation equations. Moreover, the gradients are again calculated to machine precision, and the poor scaling seen with numerical differentiation is avoided. The major downside of symbolic differentiation, however, is that the resulting expressions for derivatives can become exponentially longer than the original function, with correspondingly long evaluation times. Consider a function $f(x)$ given by the product of $u(x)$ and $v(x)$. The function and its derivative are given by

$$f(x) = u(x)v(x) \tag{8.42}$$

$$f'(x) = u'(x)v(x) + u(x)v'(x). \tag{8.43}$$

We see that there is redundant computation in that $u(x)$ and $v(x)$ must be evaluated both for the calculation of $f(x)$ and for $f'(x)$. If the factors $u(x)$ and $v(x)$ themselves involve factors, then we end up with a nested duplication of expressions, which rapidly grow in complexity. This problem is called *expression swell*.

As a further illustration, consider a function that is structured like two layers of a neural network (Grosse, 2018) with a single input x, a hidden unit with activation z, and an output y in which

$$z = h(w_1 x + b_1) \tag{8.44}$$

$$y = h(w_2 z + b_2) \tag{8.45}$$

where $h(a)$ is the soft ReLU:

$$\zeta(a) = \ln\left(1 + \exp(a)\right). \tag{8.46}$$

The overall function is therefore given by

$$y(x) = h\left(w_2 h(w_1 x + b_1) + b_2\right) \tag{8.47}$$

Telegram: @uni_k

www.konkur.in

Exercise 8.13

and the derivative of the network output with respect to w_1, evaluated symbolically, is given by

$$\frac{\partial y}{\partial w_1} = \frac{w_2 x \exp\left(w_1 x + b_1 + b_2 + w_2 \ln[1 + e^{w_1 x + b_1}]\right)}{(1 + e^{w_1 x + b_1})\,(1 + \exp(b_2 + w_2 \ln[1 + e^{w_1 x + b_1}]))}. \tag{8.48}$$

As well as being significantly more complex than the original function, we also see redundant computation where expressions such as $w_1 x + b_1$ occur in several places.

A further major drawback with symbolic differentiation is that it requires that the expression to be differentiated is expressed in closed form. It therefore excludes important control flow operations such as loops, recursions, conditional execution, and procedure calls, which are valuable constructs that we might wish to use when defining the network function.

We therefore turn to the fourth technique for evaluating derivatives in neural networks called *automatic differentiation*, also known as 'autodiff' or 'algorithmic differentiation' (Baydin *et al.*, 2018). Unlike symbolic differentiation, the goal of automatic differentiation is not to find a mathematical expression for the derivatives but to have the computer automatically generate the code that implements the gradient calculations given only the code for the forward propagation equations. It is accurate to machine precision, just as with symbolic differentiation, but is more efficient because it is able to exploit intermediate variables used in the definition of the forward propagation equations and thereby avoid redundant evaluations. It is important to note that not only can automatic differentiation handle conventional closed-form mathematical expressions but it can also deal with flow control elements such as branches, loops, recursion, and procedure calls, and is therefore significantly more powerful than symbolic differentiation. Automatic differentiation is a well-established field with broad applicability that was developed largely outside of the machine learning community. Modern deep learning is a largely empirical process, involving evaluating and comparing different architectures, and automatic differentiation therefore plays a key role in enabling this experimentation to be done accurately and efficiently.

The key idea of automatic differentiation is to take the code that evaluates a function, for example the forward propagation equations that evaluate the error function for a neural network, and augment the code with additional variables whose values are accumulated during code execution to obtain the required derivatives. There are two principal forms of automatic differentiation, known as forward mode and reverse mode. We start by looking at forward mode, which is conceptually somewhat simpler.

8.2.1 Forward-mode automatic differentiation

In forward-mode automatic differentiation, we augment each intermediate variable z_i, known as a 'primal' variable, involved in the evaluation of a function, such as the error function of a neural network, with an additional variable representing the value of some derivative of that variable, which we can denote $\dot{z}_i$, known as a 'tangent' variable. The tangent variables and their associated code are generated

Telegram: @uni_k

www.konkur.in

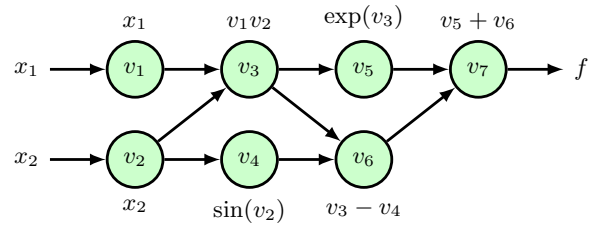


Figure 8.4 Evaluation trace diagram showing the steps involved in the numerical evaluation of the function (8.49) using the primal equations (8.50) to (8.56).

automatically by the software environment. Instead of simply doing forward propagation to compute $\{z_i\}$, the code now propagates tuples $(z_i, \dot{z}_i)$ so that variables and derivatives are evaluated in parallel. The original function is generally defined in terms of elementary operators consisting of arithmetic operations and negation as well as transcendental functions such as exponential, logarithm, and trigonometric functions, all of which have simple formulae for their derivatives. Using these derivatives in combination with the chain rule of calculus allows the code used to evaluate gradients to be constructed automatically.

As an example, consider the following function, which has two input variables:

$$f(x_1, x_2) = x_1 x_2 + \exp(x_1 x_2) - \sin(x_2). \tag{8.49}$$

When implemented in software, the code consists of a sequence of operations that can be expressed as an *evaluation trace* of the underlying elementary operations. This trace can be visualized in the form of a graph, as shown in Figure 8.4. Here we have defined the following primal variables

$$v_1 = x_1 \tag{8.50}$$
$$v_2 = x_2 \tag{8.51}$$
$$v_3 = v_1 v_2 \tag{8.52}$$
$$v_4 = \sin(v_2) \tag{8.53}$$
$$v_5 = \exp(v_3) \tag{8.54}$$
$$v_6 = v_3 - v_4 \tag{8.55}$$
$$v_7 = v_5 + v_6. \tag{8.56}$$

Now suppose we wish to evaluate the derivative $\partial f / \partial x_1$. We define the tangent variables by $\dot{v}_i = \partial v_i / \partial x_1$. Expressions for evaluating these can be constructed automatically using the chain rule of calculus:

$$\dot{v}_i = \frac{\partial v_i}{\partial x_1} = \sum_{j \in \mathrm{pa}(i)} \frac{\partial v_j}{\partial x_1} \frac{\partial v_i}{\partial v_j} = \sum_{j \in \mathrm{pa}(i)} \dot{v}_j \frac{\partial v_i}{\partial v_j}. \tag{8.57}$$

where $\mathrm{pa}(i)$ denotes the set of *parents* of the node i in the evaluation trace diagram, that is the set of variables with arrows pointing to node i. For example, in Figure 8.4 the parents of node v_3 are nodes v_1 and v_2. Applying (8.57) to the evaluation trace

Telegram: @uni_k

www.konkur.in

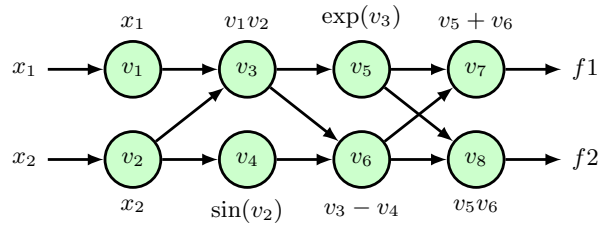


Figure 8.5 Extension of the example shown in Figure 8.4 to a function with two outputs f_1 and f_2.

equations (8.50) to (8.56), we obtain the following evaluation trace equations for the tangent variables

$$\dot{v}_1 = 1 \tag{8.58}$$
$$\dot{v}_2 = 0 \tag{8.59}$$
$$\dot{v}_3 = v_1\dot{v}_2 + \dot{v}_1v_2 \tag{8.60}$$
$$\dot{v}_4 = \dot{v}_2\cos(v_2) \tag{8.61}$$
$$\dot{v}_5 = \dot{v}_3\exp(v_3) \tag{8.62}$$
$$\dot{v}_6 = \dot{v}_3 - \dot{v}_4 \tag{8.63}$$
$$\dot{v}_7 = \dot{v}_5 + \dot{v}_6. \tag{8.64}$$

We can summarize automatic differentiation for this example as follows. We first write code to implement the evaluation of the primal variables, given by (8.50) to (8.56). The associated equations and corresponding code for evaluating the tangent variables (8.58) to (8.64) are generated automatically. To evaluate the derivative $\partial f/\partial x_1$, we input specific values of x_1 and x_2 and the code then executes the primal and tangent equations, numerically evaluating the tuples $(v_i, \dot{v}_i)$ in sequence until we obtain $\dot{v}_5$, which is the required derivative.

Exercise 8.17

Now consider an example with two outputs $f_1(x_1, x_2)$ and $f_2(x_1, x_2)$ where $f_1(x_1, x_2)$ is defined by (8.49) and

$$f_2(x_1, x_2) = (x_1x_2 - \sin(x_2))\exp(x_1x_2) \tag{8.65}$$

as illustrated by the evaluation trace diagram in Figure 8.5. We see that this involves only a small extension to the evaluation equations for the primal and tangent variables, and so both $\partial f_1/\partial x_1$ and $\partial f_2/\partial x_1$ can be evaluated together in a single forward pass. The downside, however, is that if we wish to evaluate derivatives with respect to a different input variable x_2 then we have to run a separate forward pass. In general, if we have a function with D inputs and K outputs then a single pass of forward-mode automatic differentiation produces a single column of the $K \times D$ Jacobian matrix:

$$\mathbf{J} = \begin{bmatrix} \dfrac{\partial f_1}{\partial x_1} & \cdots & \dfrac{\partial f_1}{\partial x_D} \\ \vdots & \ddots & \vdots \\ \dfrac{\partial f_K}{\partial x_1} & \cdots & \dfrac{\partial f_K}{\partial x_D} \end{bmatrix}. \tag{8.66}$$

Telegram: @uni_k

www.konkur.in

To compute column j of the Jacobian, we need to initialize the forward pass of the tangent equations by setting $\dot{x}_j = 1$ and $\dot{x}_i = 0$ for $i \neq j$. We can write this in vector form as $\dot{\mathbf{x}} = \mathbf{e}_i$ where $\mathbf{e}_i$ is the ith unit vector. To compute the full Jacobian matrix we need D forward-mode passes. However, if we wish to evaluate the product of the Jacobian with a vector $\mathbf{r} = (r_1, \ldots, r_D)^{\mathrm{T}}$:

$$\mathbf{J} = \begin{bmatrix} \dfrac{\partial f_1}{\partial x_1} & \cdots & \dfrac{\partial f_1}{\partial x_D} \\ \vdots & \ddots & \vdots \\ \dfrac{\partial f_K}{\partial x_1} & \cdots & \dfrac{\partial f_K}{\partial x_D} \end{bmatrix} \begin{bmatrix} r_1 \\ \\ \vdots \\ \\ r_D \end{bmatrix} \tag{8.67}$$

Exercise 8.18 then this can be done in single forward pass by setting $\dot{\mathbf{x}} = \mathbf{r}$.

We see that forward-mode automatic differentiation can evaluate the full $K \times D$ Jacobian matrix of derivatives using D forward passes. This is very efficient for networks with a few inputs and many outputs, such that $K \gg D$. However, we often operate in a regime where we often have just one function, namely the error function used for training, and large numbers of variables that we want to differentiate with respect to, comprising the weights and biases in the network, of which there may be millions or billions. In such situations, forward-mode automatic differentiation is extremely inefficient. We therefore turn to an alternative version of automatic differentiation based on the a backwards flow of derivative data through the evaluation trace graph.

8.2.2 Reverse-mode automatic differentiation

We can think of reverse-mode automatic differentiation as a generalization of the error backpropagation procedure. As with forward mode, we augment each intermediate variable v_i with additional variables, in this case called *adjoint* variables, denoted $\overline{v}_i$. Consider again a situation with a single output function f for which the adjoint variables are defined by

$$\overline{v}_i = \frac{\partial f}{\partial v_i}. \tag{8.68}$$

These can be evaluated sequentially starting with the output and working backwards by using the chain rule of calculus:

$$\overline{v}_i = \frac{\partial f}{\partial v_i} = \sum_{j \in \mathrm{ch}(i)} \frac{\partial f}{\partial v_j} \frac{\partial v_j}{\partial v_i} = \sum_{j \in \mathrm{ch}(i)} \overline{v}_j \frac{\partial v_j}{\partial v_i}. \tag{8.69}$$

Here $\mathrm{ch}(i)$ denotes the *children* of node i in the evaluation trace graph, in other words the set of nodes that have arrows pointing into them from node i. The successive evaluation of the adjoint variables represents a flow of information backwards through the graph, as we saw previously.

Figure 8.1

If we again consider the specific example function given by (8.50) to (8.56), we

Exercise 8.16 obtain the following evaluation equations for the evaluation of the adjoint variables

Telegram: @uni_k

www.konkur.in

$$\overline{v}_7 = 1 \tag{8.70}$$
$$\overline{v}_6 = \overline{v}_7 \tag{8.71}$$
$$\overline{v}_5 = \overline{v}_7 \tag{8.72}$$
$$\overline{v}_4 = -\overline{v}_6 \tag{8.73}$$
$$\overline{v}_3 = \overline{v}_5 v_5 + \overline{v}_6 \tag{8.74}$$
$$\overline{v}_2 = \overline{v}_2 v_1 + \overline{v}_4 \cos(v_2) \tag{8.75}$$
$$\overline{v}_1 = \overline{v}_3 v_2. \tag{8.76}$$

Note that these start at the output and then flow backwards through the graph to the inputs. Even with multiple inputs, only a single backward pass is required to evaluate the derivatives. For a neural network error function, the derivatives of E with respect to the weight and biases are obtained as the corresponding adjoint variables. However, if we now have more than one output then we need to run a separate backward pass for each output variable.

Figure 8.5

Reverse mode is often more memory intensive than forward mode because all of the intermediate primal variables must be stored so that they will be available as needed when evaluating the adjoint variables during the backward pass. By contrast, with forward mode, the primal and tangent variables are computed together during the forward pass, and therefore variables can be discarded once they have been used. It is therefore also generally easier to implement forward mode compared to reverse mode.

For both forward-mode and reverse-mode automatic differentiation, a single pass through the network is guaranteed to take no more than 6 times the computational cost of a single function evaluation. In practice, the overhead is typically closer to a factor of 2 or 3 (Griewank and Walther, 2008). Hybrids of forward and reverse modes are also of interest. One situation in which this arises is in the evaluation of the product of a Hessian matrix with a vector, which can be calculated without explicit evaluation of the full Hessian (Pearlmutter, 1994). Here we can use reverse mode to calculate the gradient of code, which itself has been generated by the forward model. We start from a vector $\mathbf{b}$ and a point $\mathbf{x}$ at which the Hessian–vector product is to be evaluated. By setting $\dot{\mathbf{x}} = \mathbf{v}$ and using forward mode, we obtain the directional derivative $\mathbf{v}^{\mathrm{T}}\nabla f$. This is then differentiated using reverse mode to obtain $\nabla^2 f\mathbf{v} = \mathbf{H}\mathbf{v}$. If W is the number of parameters in the neural network then this evaluation has $\mathcal{O}(W)$ complexity even though the Hessian is of size $W \times W$. The Hessian itself can also be evaluated explicitly using automatic differentiation but this has $\mathcal{O}(W^2)$ complexity.

Exercises

8.1 ($\star$) By making use of (8.5), (8.6), (8.8), and (8.12), verify the backpropagation formula (8.13) for evaluating the derivatives of an error function.

8.2 ($\star\star$) Consider a network that consists of layers and rewrite the backpropagation formula (8.13) in matrix notation by starting with the forward propagation equation (6.19). Note that the result involves multiplication by the transposes of the matrices.

Telegram: @uni_k

www.konkur.in

8.3 ($\star$) By using a Taylor expansion, verify that the terms that are $\mathcal{O}(\epsilon)$ cancel on the right-hand side of (8.25).

8.4 ($\star\star$) Consider a two-layer network of the form shown in Figure 6.9 with the addition of extra parameters corresponding to skip-layer connections that go directly from the inputs to the outputs. By extending the discussion of Section 8.1.3, write down the equations for the derivatives of the error function with respect to these additional parameters.

8.5 ($\star\star\star$) In Section 8.1.5, we derived a procedure for evaluating the Jacobian matrix of a neural network using a backpropagation procedure. Derive an alternative formalism for finding the Jacobian based on *forward propagation* equations.

8.6 ($\star\star\star$) Consider a two-layer neural network, and define the quantities

$$\delta_k = \frac{\partial E_n}{\partial a_k}, \qquad M_{kk'} \equiv \frac{\partial^2 E_n}{\partial a_k \partial a_{k'}}. \tag{8.77}$$

Derive expressions for the elements of the Hessian matrix expressed in terms of δ_k and $M_{kk'}$ for elements in which (i) both weights are in the second layer, (ii) both weights are in the first layer, and (iii) one weight is in each layer.

8.7 ($\star\star\star$) Extend the results of Exercise 8.6 for the exact Hessian of a two-layer network to include skip-layer connections that go directly from inputs to outputs.

8.8 ($\star\star$) The outer product approximation to the Hessian matrix for a neural network using a sum-of-squares error function is given by (8.40). Extend this result for multiple outputs.

8.9 ($\star\star$) Consider a squared-loss function of the form

$$E(\mathbf{w}) = \frac{1}{2} \iint \{y(\mathbf{x}, \mathbf{w}) - t\}^2 \, p(\mathbf{x}, t) \, \mathrm{d}\mathbf{x} \, \mathrm{d}t \tag{8.78}$$

where $y(\mathbf{x}, \mathbf{w})$ is a parametric function such as a neural network. The result (4.37) shows that the function $y(\mathbf{x}, \mathbf{w})$ that minimizes this error is given by the conditional expectation of t given $\mathbf{x}$. Use this result to show that the second derivative of E with respect to two elements w_r and w_s of the vector $\mathbf{w}$, is given by

$$\frac{\partial^2 E}{\partial w_r \partial w_s} = \int \frac{\partial y}{\partial w_r} \frac{\partial y}{\partial w_s} p(\mathbf{x}) \, \mathrm{d}\mathbf{x}. \tag{8.79}$$

Note that, for a finite sample from $p(\mathbf{x})$, we obtain (8.40).

8.10 ($\star\star$) Derive the expression (8.41) for the outer product approximation of a Hessian matrix for a network having a single output with a logistic-sigmoid output-unit activation function and a cross-entropy error function, corresponding to the result (8.40) for the sum-of-squares error function.

Telegram: @uni_k

www.konkur.in

8.11 (⋆⋆) Derive an expression for the outer product approximation of a Hessian matrix for a network having K outputs with a softmax output-unit activation function and a cross-entropy error function, corresponding to the result (8.40) for the sum-of-squares error function.

8.12 (⋆⋆⋆) Consider the matrix identity

$$\left(\mathbf{M}+\mathbf{v}\mathbf{v}^{\mathrm{T}}\right)^{-1}=\mathbf{M}^{-1}-\frac{\left(\mathbf{M}^{-1}\mathbf{v}\right)\left(\mathbf{v}^{\mathrm{T}}\mathbf{M}^{-1}\right)}{1+\mathbf{v}^{\mathrm{T}}\mathbf{M}^{-1}\mathbf{v}}, \tag{8.80}$$

which is simply a special case of the Woodbury identity (A.7). By applying (8.80) to the outer product approximation (8.40) for a Hessian matrix, derive a formula that allows the inverse of the Hessian matrix to be computed by making one pass through the training data and updating the inverse Hessian with each data point. Note that the algorithm can initialized using $\mathbf{H}=\alpha\mathbf{I}$ where α is a small constant, and that the results are not particularly sensitive to the precise value of α.

8.13 (⋆⋆) Verify that the derivative of (8.47) is given by (8.48).

8.14 (⋆⋆) The logistic map is a function defined by the iterative relation $L_{n+1}(x)=4L_n(x)\left(1-L_n(x)\right)$ with $L_1(x)=x$. Write down the evaluation trace equations for $L_2(x)$, $L_3(x)$, and $L_4(x)$, and then write down expressions for the corresponding derivatives $L_1'(x)$, $L_2'(x)$, $L_3'(x)$, and $L_4'(x)$. Do not simplify the expressions but instead simply note how the complexity of the formulae for the derivatives grows much more rapidly than the expressions for the functions themselves.

8.15 (⋆⋆) Starting from the evaluation trace equations (8.50) to (8.56) for the example function (8.49), use (8.57) to derive the forward-mode tangent variable evaluation equations (8.58) to (8.64).

8.16 (⋆⋆) Starting from the evaluation trace equations (8.50) to (8.56) for the example function (8.49), and referring to Figure 8.4, use (8.69) to derive the reverse-mode adjoint variable evaluation equations (8.70) to (8.76).

8.17 (⋆⋆⋆) Consider the example function (8.49). Write down an expression for $\partial f/\partial x_1$ and evaluate this function for $x_1=1$ and $x_2=2$. Now use the evaluation trace equations (8.50) to (8.56) to evaluate the variables v_1 to v_7 and then use the evaluation trace equations of forward-mode automatic differentiation to evaluate the tangent variables $\dot{v}_1$ to $\dot{v}_7$ and to confirm that the resulting value of $\partial f/\partial x_1$ agrees with that found directly. Similarly, use the evaluation trace equations of reverse-mode automatic differentiation (8.70) to (8.76) to evaluate the adjoint variables $\overline{v}_7$ to $\overline{v}_1$ and again confirm that the resulting value of $\partial f/\partial x_1$ agrees with that found directly.

8.18 (⋆⋆) By expressing an arbitrary vector $\mathbf{r}=(r_1,\ldots,r_D)^{\mathrm{T}}$ as a linear combination of the unit vectors $\mathbf{e}_i$, where $i=1,\ldots,D$, show that the product of the Jacobian of a function with $\mathbf{r}$ in the form (8.67) can be evaluated using a single pass of forward-mode automatic differentiation by setting $\dot{\mathbf{x}}=\mathbf{r}$.

Telegram: @uni_k

www.konkur.in



9 Regularization

Section 1.2

We introduced the concept of regularization when discussing polynomial curve fitting as a way to reduce over-fitting by discouraging the parameters of the model from taking values with a large magnitude. This involved adding a simple penalty term to the error function to give a regularized error function in the form

$$\widetilde{E}(\mathbf{w}) = E(\mathbf{w}) + \frac{\lambda}{2}\mathbf{w}^{\mathrm{T}}\mathbf{w} \tag{9.1}$$

where $\mathbf{w}$ is the vector of model parameters, $E(\mathbf{w})$ is the unregularized error function, and the regularization hyperparameter λ controls the strength of the regularization effect. An improvement in predictive accuracy with such a regularizer can be understood in terms of the bias–variance trade-off through the reduction in the variance of the solution at the expense of some increase in bias. In this chapter we will explore regularization in depth and will discuss several different approaches to regulariza-

Section 4.3

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024
C. M. Bishop, H. Bishop, *Deep Learning*, https://doi.org/10.1007/978-3-031-45468-4_9

Telegram: @uni_k

www.konkur.in

tion. We will also look more broadly at the important role of bias in achieving good generalization from finite training data sets.

In a practical application, it is very unlikely that the process that generates the data will correspond precisely to a particular neural network architecture, and so any given neural network will only ever represent an approximation to the true data generator. Larger networks can provide closer approximations, but this comes at the risk of over-fitting. In practice, we find that the best generalization results are almost always obtained by using a larger network combined with some form of regularization. In this chapter we explore several alternative regularization techniques including early stopping, model averaging, dropout, data augmentation, and parameter sharing. Multiple forms of regularization can be used together if desired. For example, error function regularization of the form (9.1) is often used alongside dropout.

9.1. Inductive Bias

Section 1.2.4

Section 1.2.5

Section 4.3

When we compared the predictive error of polynomials of various orders for the sinusoidal synthetic data problem, we saw that the smallest generalization error was achieved using a polynomial of intermediate complexity, being neither too simple nor too flexible. A similar result was found when we used a regularization term of the form (9.1) to control model complexity, as an intermediate value of the regularization coefficient λ gave the best predictions for new input values. Insight into this result came from the bias–variance decomposition, where we saw that an appropriate level of bias in the model was important to allow generalization from finite data sets. Simple models with high bias are unable to capture the variation in the underlying data generation process, whereas highly flexible models with low bias are prone to over-fitting leading to poor generalization. As the size of the data set grows, we can afford to use more flexible models having less bias without incurring excessive variance, thereby leading to improved generalization. Note that in a practical setting, our choice of model might also be influenced by factors such as memory usage or speed of execution. Here we ignore such ancillary considerations and focus on the core goal of achieving good predictive performance, in other words good generalization.

9.1.1 Inverse problems

This issue of model choice lies at the heart of machine learning and can be traced to the fact that most machine learning tasks are examples of *inverse problems*. Given a conditional distribution $p(t|\mathbf{x})$ along with a finite set of input points $\{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$, it is straightforward, at least in principle, to sample corresponding values $\{t_1, \ldots, t_N\}$ from that distribution. In machine learning, however, we have to solve the inverse of this problem, namely to infer an entire distribution given only a finite number of samples. This is intrinsically *ill-posed*, as there are infinitely many distributions which could potentially have been responsible for generating the observed training data. In fact any distribution that has a non-zero probability density at the observed target values is a candidate.

Telegram: @uni_k

www.konkur.in

For machine learning to be useful, however, we need to make predictions for new values of $\mathbf{x}$, and therefore we need a way to choose a specific distribution from amongst the infinitely many possibilities. The preference for one choice over others is called *inductive bias*, or *prior knowledge*, and plays a central role in machine learning. Prior knowledge may come from background information that helps constrain the space of solutions. For many applications, small changes in the input values should lead to small changes in the output values, and so we should bias our solutions towards those with smoothly varying functions. Regularization terms of the form (9.1) encourage the model weights to have a smaller magnitude and hence introduce a bias towards functions that vary more slowly with changes in the inputs. Likewise, when detecting objects in images, we can introduce prior knowledge that the identity of an object is generally independent of its location within the image. This is known as translation invariance, and incorporating this into our solution can greatly simplify the task of building a system with good generalization. However, care must be taken not to incorporate biases or constraints that are inconsistent with the underlying process that generates the data. For example, assuming that the relationship between outputs and inputs is linear when in fact there are significant nonlinearities can lead to a system that produces inaccurate answers.

Chapter 10

Techniques such as transfer learning and multi-task learning can also be viewed from the perspective of regularization. When training data for a particular task is limited, additional data from a different, but related, task can be used to help determine the learnable parameters in a neural network. The assumption of similarity between the tasks represents a more sophisticated form of inductive bias compared to simple regularization, and this explains the improved performance resulting from the use of the additional data.

Section 6.3.4

9.1.2 No free lunch theorem

The core focus of this book is on the important class of machine learning models called deep neural networks. These are highly flexible models and have revolutionized many fields including computer vision, speech recognition, and natural language processing. In fact, they have become the framework of choice for the great majority of machine learning applications. It might appear, therefore, that they represent a 'universal' learning algorithm able to solve all tasks. However, even very flexible neural networks contain important inductive biases. For example, convolutional neural networks encode specific forms of inductive bias, including translation equivariance, that are especially useful in applications involving images.

Chapter 10

The *no free lunch theorem* (Wolpert, 1996), named from the expression 'There's no such thing as a free lunch,' states that every learning algorithm is as good as any other when averaged over all possible problems. If a particular model or algorithm is better than average on some problems, it must be worse than average on others. However, this is a rather theoretical notion as the space of possible problems here includes relationships between input and output that would be very uncharacteristic of any plausible practical application. For example, we have already noted that most examples of practical interest exhibit some degree of smoothness, in which small changes in the input values are associated, for the most part, with small changes in

Telegram: @uni_k

www.konkur.in

the target values. Models such as neural networks, and indeed most widely used machine learning techniques, exhibit this form of inductive bias, and therefore to some degree, they have broad applicability.

Although the no free lunch theorem is somewhat theoretical, it does highlight the central importance of bias in determining the performance of a machine learning algorithm. It is not possible to learn 'purely from data' in the absence of any bias. In practice, bias may be implicit. For example, every neural network has a finite number of parameters, which therefore limits the functions that it can represent. However, bias may also be encoded explicitly as a reflection of prior knowledge relating to the specific type of problem being solved.

In trying to find general-purpose learning algorithms, we are really seeking inductive biases that are appropriate to the broad classes of applications that will be encountered in practice. For any given application, however, better results can be obtained if it is possible to incorporate stronger inductive biases that are specific to that application. The perspective of *model-based machine learning* (Winn *et al.*, 2023) advocates making all the assumptions in machine learning models explicit so that the appropriate choices can be made for inductive biases.

We have seen that inductive bias can be incorporated through the form of distribution, for example by specifying that the output is a linear function of a fixed set of specific basis functions. It can also be incorporated through the addition of a regularization term to the error function used during training. Yet another way to control the complexity of a neural network is through the training process itself. We will see that deep neural networks can give good generalization even when the number of adjustable parameters exceeds the number of training data points, provided the training process is set up correctly. Part of the skill in applying deep learning to real-world problems is in the careful design of inductive bias and the incorporation of prior knowledge.

Chapter 7

9.1.3 Symmetry and invariance

In many applications of machine learning, the predictions should be unchanged, or *invariant*, under one or more transformations of the input variables. For example, when classifying an object in two-dimensional images, such as a cat or dog, a particular object should be assigned the same classification irrespective of its position within the image. This is known as *translation invariance*. Likewise changes to the size of the object within the image should also leave its classification unchanged. This is called *scale invariance*. Exploiting such symmetries to create inductive biases can greatly improve the performance of machine learning models and forms the subject of *geometric deep learning* (Bronstein *et al.*, 2021).

Transformations, such as a translation or scaling, that leave particular properties unchanged, represent *symmetries*. The set of all transformations corresponding to a particular symmetry form a mathematical structure called a *group*. A group consists of a set of elements $\mathcal{A}, \mathcal{B}, \mathcal{C}, \ldots$ together with a binary operation for composing pairs of elements together, which we denote using the notation $\mathcal{A} \circ \mathcal{B}$. The following four axioms hold for a group:

Telegram: @uni_k

www.konkur.in

1. **Closed**: For any two elements $\mathcal{A}, \mathcal{B}$ in the set, $\mathcal{A} \circ \mathcal{B}$ must also be in the set.
2. **Associative**: For any three elements $\mathcal{A}, \mathcal{B}, \mathcal{C}$ in the set, $(\mathcal{A} \circ \mathcal{B}) \circ \mathcal{C} = \mathcal{A} \circ (\mathcal{B} \circ \mathcal{C})$.
3. **Identity**: There exists an element $\mathcal{I}$ of the set, called the identity, with the property: $\mathcal{A} \circ \mathcal{I} = \mathcal{I} \circ \mathcal{A} = \mathcal{A}$ for every element $\mathcal{A}$ in the set.
4. **Inverse**: For each element $\mathcal{A}$ in the set, there exists another element in the set, which we denote by $\mathcal{A}^{-1}$, called the inverse, which has the property: $\mathcal{A} \circ \mathcal{A}^{-1} = \mathcal{A}^{-1} \circ \mathcal{A} = \mathcal{I}$.

Simple examples of groups include the set of rotations of a square through multiples of $90°$ or the set of continuous translations of an object in a two-dimensional plane.

Exercise 9.1

In principle, invariance of the predictions made by a neural network to transformations of the input space could be learned from data, without any special modifications to the network or the training procedure. In practice, however, this can prove to be extremely challenging because such transformations can produce substantial changes in the raw data. For example, relatively small translations of an object within an image, even by a few pixels, can cause pixel values to change significantly. Furthermore, multiple invariances must often hold at the same time, for example invariance to translations in two dimensions as well as scaling, rotation, changes of intensity, changes of colour balance, and many others. There are exponentially many possible *combinations* of such transformations, making the size of the required training set needed to learn all of these invariances prohibitive.

We therefore seek more efficient approaches for encouraging an adaptive model to exhibit the required invariances. These can broadly be divided into four categories:

1. **Pre-processing.** Invariances are built into a pre-processing stage by computing features of the data that are invariant under the required transformations. Any subsequent regression or classification system that uses such features as inputs will necessarily also respect these invariances.
2. **Regularized error function.** A regularization term is added to the error function to penalize changes in the model output when the input is subject to one of the invariant transformations.
3. **Data augmentation.** The training set is expanded using replicas of the training data points, transformed according to the desired invariances and carrying the same output target values as the untransformed examples.
4. **Network architecture.** The invariance properties are built into the structure of a neural network through an appropriate choice of network architecture.

One challenge with approach 1 is to design features that exhibit the required invariances without also discarding information that can be useful for determining the network outputs. We have already seen that fixed, hand-crafted features have limited capabilities and have been superseded by learned representations obtained using deep neural networks.

Chapter 6

Telegram: @uni_k

www.konkur.in

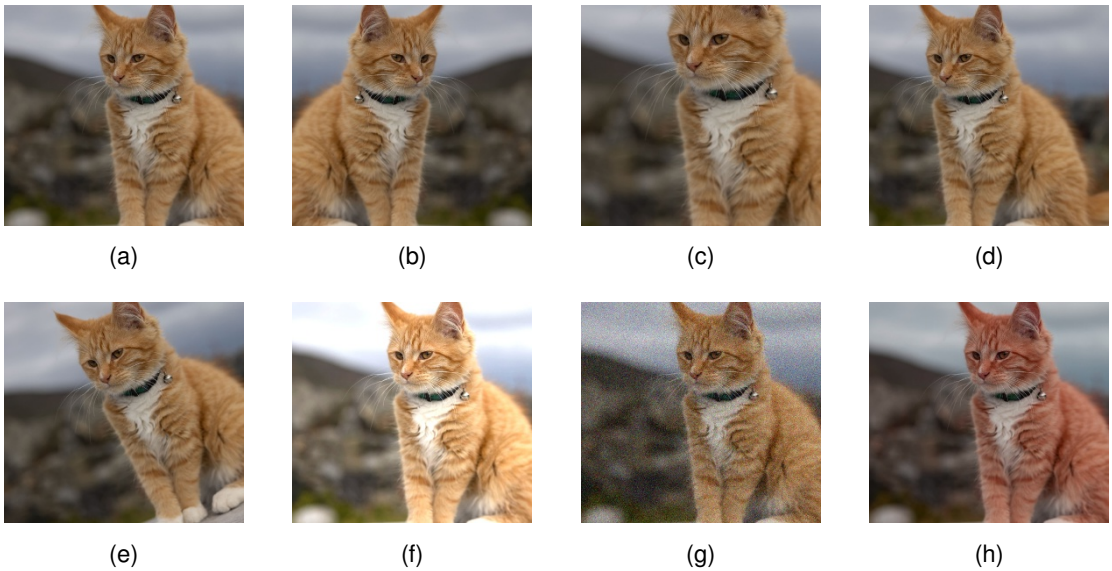


Figure 9.1 Illustration of data set augmentation, showing (a) the original image, (b) horizontal inversion, (c) scaling, (d) translation, (e) rotation, (f) brightness and contrast change, (g) additive noise, and (h) colour shift.

An example of approach 2 is the technique of *tangent propagation* (Simard *et al.*, 1992) in which a regularisation term is added to the error function during training. This term directly penalizes changes in the output resulting from changes in the input variables that correspond to one of the invariant transformations. A limitation of this technique, in addition to the extra complexity of training, is can only cope with small transformations (e.g., translations by less than a pixel).

Approach 3 is known as *data set augmentation*. It is often relatively easy to implement and can prove to be very effective in practice. It is often applied in the context of image analysis as it straightforward to create the transformed training data. Figure 9.1 shows examples of such transformations applied to an image of a cat. For medical images of soft tissue, data augmentation could also include continuous 'rubber sheet' deformations (Ronneberger, Fischer, and Brox, 2015).

For sequential training algorithms, such as stochastic gradient descent, the data set can be augmented by transforming each input data point before it is presented to the model so that, if the data points are being recycled, a different transformation (drawn from an appropriate distribution) is applied each time. For batch methods, a similar effect can be achieved by replicating each data point a number of times and transforming each copy independently.

We can analyse the effect of using augmented data by considering transformations that represent small changes to the original examples and then making a Taylor expansion of the error function in powers of the magnitude of the transformation (Bishop, 1995c; Leen, 1995; Bishop, 2006). This leads to a regularized error function in which the regularizer penalizes the gradient of the network output with respect

Telegram: @uni_k

www.konkur.in

to the input variables projected onto the direction of transformation. This is related to the technique of tangent propagation discussed above. A special case arises when the transformation of the input variables consists simply of the addition of random noise, in which case the regularizer penalizes the derivatives of the network outputs with respect to the inputs. Again, this is intuitively reasonable, since we are encouraging the network outputs to remain unchanged despite the addition of noise to the input variables.

Exercise 9.2

Finally, approach 4, in which we build invariances into the structure of the network, has proven to be very powerful and effective and leads to other key benefits. We will discuss this approach at length in the context of convolutional neural networks for computer vision.

Chapter 10

9.1.4 Equivariance

An important generalization of invariance is called *equivariance* in which the output of the network, instead of remaining constant when the input is transformed, is itself transformed in a specific way. For example, consider a network that takes an image as input and returns a segmentation of that image in which each pixel is classified as belonging either to a foreground object or to the background. In this case, if the location of the object within the image is translated, we want the corresponding segmentation of the object to be similarly translated. Suppose we denote the image by $\mathbf{I}$, and the operation of the segmentation network by $\mathcal{S}$, then for a translation operation $\mathcal{T}$ we have

$$\mathcal{S}(\mathcal{T}(\mathbf{I})) = \mathcal{T}(\mathcal{S}(\mathbf{I})), \tag{9.2}$$

which says that the segmentation of the translated image is given by the translation of the segmentation of the original image. This is illustrated in Figure 9.2

More generally, equivariance can hold if the transformation applied to the output is different to that applied to the input:

$$\mathcal{S}(\mathcal{T}(\mathbf{I})) = \widetilde{\mathcal{T}}(\mathcal{S}(\mathbf{I})). \tag{9.3}$$

For example, if the segmented image has a lower resolution than the original image, then if $\mathcal{T}$ is a translation in the original image space, $\widetilde{\mathcal{T}}$ represents the corresponding translation in the lower-dimensional segmentation space. Similarly, if $\mathcal{S}$ is an operator that measures the orientation of an object within an image, and $\mathcal{T}$ represents a rotation (which is a complex nonlinear transformation of all of the pixel values in the image) then $\widetilde{\mathcal{T}}$ will increment or decrement the scalar orientation value generated by $\mathcal{S}$.

We also see that invariance is a special case of equivariance in which the output transformation is simply the identity. For example, if $\mathcal{C}$ is a neural network that classifies an object within an image and $\mathcal{T}$ is a translation operator then

$$\mathcal{C}(\mathcal{T}(\mathbf{I})) = \mathcal{C}(\mathbf{I}), \tag{9.4}$$

which says that the class of the object does not depend on its position within the image.

Telegram: @uni_k

www.konkur.in

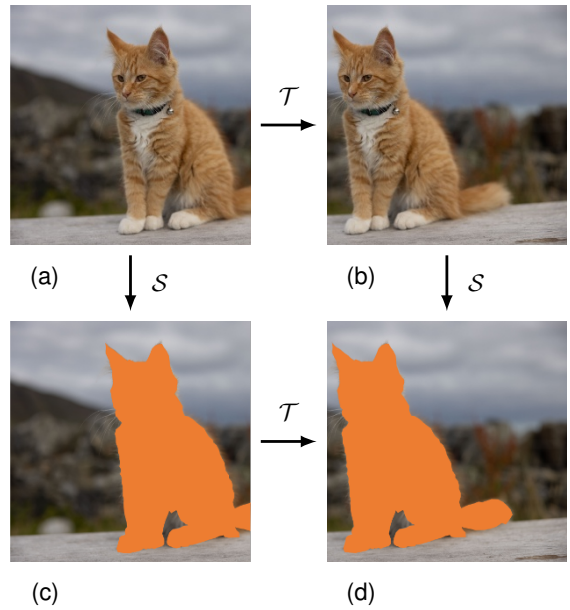


Figure 9.2 Illustration of equivariance, corresponding to (9.2). If an image (a) is first translated to give (b) and then segmented to give (d), the result is the same as if the image is first segmented to give (c) and then translated to give (d).

9.2. Weight Decay

Section 1.2.5

We introduced regularization in the context of linear regression to control model complexity, as an alternative to limiting the number of parameters in the model. The simplest regularizer comprises the sum of the squares of the model parameters to give a regularized error function of the form (9.1), which penalizes parameter values with large magnitude. The effective model complexity is then determined by the choice of the regularization coefficient λ.

We have also seen that this additive regularization term can be interpreted as the negative logarithm of a zero-mean Gaussian prior distribution over the weight vector $\mathbf{w}$. This provides a probabilistic perspective on the inclusion of prior knowledge into the model training process. Unfortunately, this prior is expressed over the model parameters, whereas any domain knowledge we might possess regarding the problem to be solved is more likely to be expressed in terms of the network function from inputs to outputs. The relationship between the parameters and the network function is, however, extremely complex, and therefore only very limited kinds of prior knowledge can easily be expressed directly as priors over model parameters.

Section 2.6.2

The sum-of-squares regularizer in (9.1) is known in the machine learning literature as *weight decay* because in sequential learning algorithms, it encourages weight values to decay towards zero, unless supported by the data. One advantage of this kind of regularizer is that it is trivial to evaluate its derivatives for use in gradient descent training. Specifically for (9.1) the gradient is given by

Exercise 9.3

$$\nabla \widetilde{E}(\mathbf{w}) = \nabla E(\mathbf{w}) + \lambda \mathbf{w}. \tag{9.5}$$

Telegram: @uni_k

www.konkur.in

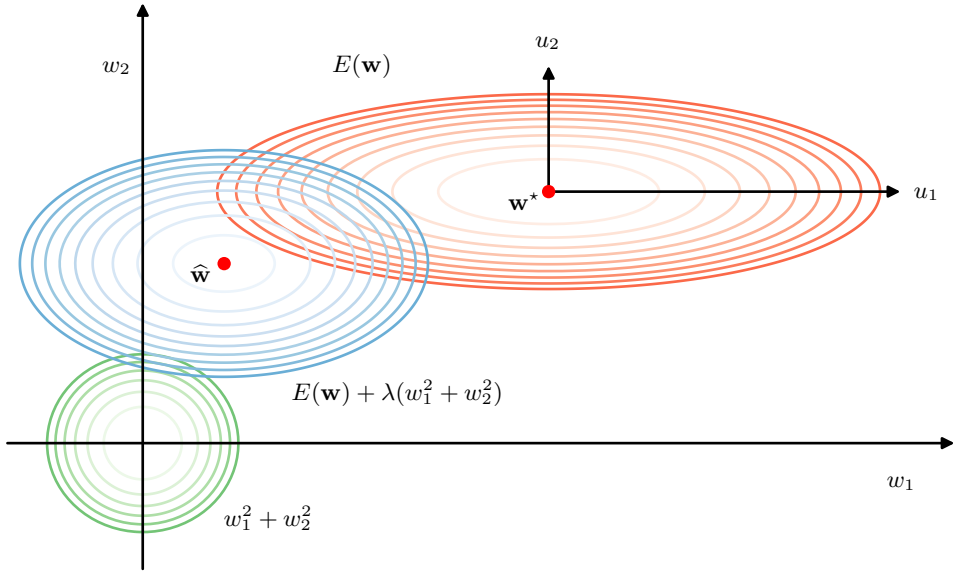


Figure 9.3 Contours of the error function (red), the regularization term (green), and a linear combination of the two (blue) for a quadratic error function and a sum-of-squares regularizer $\lambda(w_1^2 + w_2^2)$. Here the axes in parameter space have been rotated to align with the axes of the elliptical contour of the unregularized error function. For $\lambda = 0$, the minimum error is indicated by $\mathbf{w}^\star$. When $\lambda > 0$, the minimum of the regularized error function $E(\mathbf{w}) + \lambda(w_1^2 + w_2^2)$ is shifted towards the origin. This shift is greater in the direction of w_1 because the unregularized error is relatively insensitive to the parameter value, and less in direction w_2 where the error is more strongly dependent on the parameter value. The regularization term is effectively suppressing parameters that have only a small effect on the accuracy of the network predictions.

We see that the factor of $1/2$ in (9.1), which is often included by convention, disappears when we take the derivative.

Section 4.1.2
Section 8.1.6

The general effect of a quadratic regularizer can be seen by considering a two-dimensional parameter space along with an unregularized error function $E(\mathbf{w})$ that is a quadratic function of $\mathbf{w}$ (corresponding to a simple linear regression model with a sum-of-squares error function), as illustrated in Figure 9.3. The axes in parameter space have been rotated to align with the eigenvectors of the Hessian matrix, corresponding to the axes of the elliptical error function contours. We see that the effect of the regularization term is to shrink the magnitudes of the weight parameters. However, the effect is much larger for parameter w_1 because the unregularized error is much less sensitive to the value of w_1 compared to that of w_2. Intuitively only the parameter w_2 is 'active' because the output is relatively insensitive to w_1, and hence the regularizer pushes w_1 close to zero. The *effective number of parameters* is the number that remain active after regularization, and this concept can be formalized either from a Bayesian or from a frequentist perspective (Bishop, 2006; Hastie, Tibshirani, and Friedman, 2009). For $\lambda \to \infty$, all the parameters are driven to zero and the effective number of parameters is then zero. As λ is reduced, the number of parameters increases until for $\lambda = 0$ it equals the total number of actual param-

Telegram: @uni_k

www.konkur.in

eters in the model. We see that controlling model complexity by regularization has similarities to controlling model complexity by limiting the number of parameters.

9.2.1 Consistent regularizers

One of the limitations of simple weight decay in the form (9.1) is that it breaks certain desirable transformation properties of network mappings. To illustrate this, consider a multilayer perceptron network having two layers of weights and linear output units that performs a mapping from a set of input variables $\{x_i\}$ to a set of output variables $\{y_k\}$. The activations of the hidden units in the first hidden layer take the form

$$z_j = h\left(\sum_i w_{ji}x_i + w_{j0}\right) \tag{9.6}$$

whereas the activations of the output units are given by

$$y_k = \sum_j w_{kj}z_j + w_{k0}. \tag{9.7}$$

Suppose we perform a linear transformation of the input data:

$$x_i \to \widetilde{x}_i = ax_i + b. \tag{9.8}$$

Then we can arrange for the mapping performed by the network to be unchanged by making a corresponding linear transformation of the weights and biases from the inputs to the units in the hidden layer:

Exercise 9.4

$$w_{ji} \to \widetilde{w}_{ji} = \frac{1}{a}w_{ji} \tag{9.9}$$

$$w_{j0} \to \widetilde{w}_{j0} = w_{j0} - \frac{b}{a}\sum_i w_{ji}. \tag{9.10}$$

Similarly, a linear transformation of the output variables of the network of the form

$$y_k \to \widetilde{y}_k = cy_k + d \tag{9.11}$$

can be achieved transforming the second-layer weights and biases using

$$w_{kj} \to \widetilde{w}_{kj} = cw_{kj} \tag{9.12}$$

$$w_{k0} \to \widetilde{w}_{k0} = cw_{k0} + d. \tag{9.13}$$

If we train one network using the original data and one network using data for which the input and/or target variables have been transformed by one of the above linear transformations, then consistency requires that we should obtain equivalent networks that differ only by the linear transformation of the weights as given. Any regularizer should be consistent with this property, otherwise it would arbitrarily favour one solution over another, equivalent one. Clearly, simple weight decay (9.1), which treats all weights and biases on an equal footing, does not satisfy this property.

Telegram: @uni_k

www.konkur.in

We therefore look for a regularizer that is invariant under the linear transformations (9.9), (9.10), (9.12), and (9.13). These require that the regularizer should be invariant to re-scaling of the weights and to shifts of the biases. Such a regularizer is given by

$$\frac{\lambda_1}{2}\sum_{w\in\mathcal{W}_1} w^2 + \frac{\lambda_2}{2}\sum_{w\in\mathcal{W}_2} w^2 \tag{9.14}$$

where $\mathcal{W}_1$ denotes the set of weights in the first layer, $\mathcal{W}_2$ denotes the set of weights in the second layer, and biases are excluded from the summations. This regularizer will remain unchanged under the weight transformations provided the regularization parameters are re-scaled using $\lambda_1 \to a^{1/2}\lambda_1$ and $\lambda_2 \to c^{-1/2}\lambda_2$.

The regularizer (9.14) corresponds to a prior distribution over the parameters of the form:

$$p(\mathbf{w}|\alpha_1,\alpha_2) \propto \exp\left(-\frac{\alpha_1}{2}\sum_{w\in\mathcal{W}_1} w^2 - \frac{\alpha_2}{2}\sum_{w\in\mathcal{W}_2} w^2\right). \tag{9.15}$$

Note that priors of this form are *improper* (they cannot be normalized) because the bias parameters are unconstrained. Using improper priors can lead to difficulties in selecting regularization coefficients and in model comparison within the Bayesian framework. It is therefore common to include separate priors for the biases (which then break the shift invariance) that have their own hyperparameters.

We can illustrate the effect of the resulting four hyperparameters by drawing samples from the prior and plotting the corresponding network functions, as shown in Figure 9.4. The priors are governed by four hyperparameters, α_1^{b}, α_1^{w}, α_2^{b}, and α_2^{w}, which represent the precisions of the Gaussian distributions of the first-layer biases, first-layer weights, second-layer biases, and second-layer weights, respectively. We see that the parameter α_2^{w} governs the vertical scale of the functions (note the different vertical axis ranges on the top two diagrams), α_1^{w} governs the horizontal scale of variations in the function values, and α_1^{b} governs the horizontal range over which variations occur. The parameter α_2^{b}, whose effect is not illustrated here, governs the range of the vertical offsets of the functions

More generally, we can consider regularizers in which the weights are divided into any number of groups $\mathcal{W}_k$ so that

$$\Omega(\mathbf{w}) = \frac{1}{2}\sum_k \alpha_k \|\mathbf{w}\|_k^2 \tag{9.16}$$

where

$$\|\mathbf{w}\|_k^2 = \sum_{j\in\mathcal{W}_k} w_j^2. \tag{9.17}$$

For example, we could use a different regularizer for each layer in a multilayer network.

Telegram: @uni_k

www.konkur.in

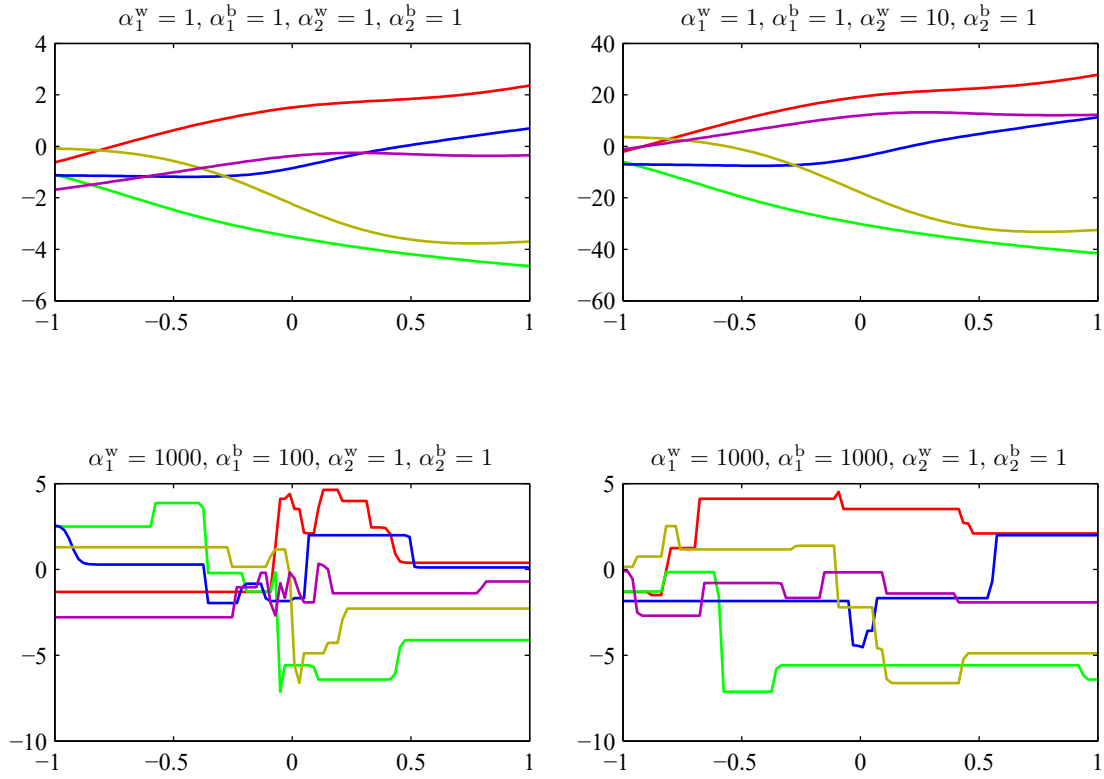


Figure 9.4 Illustration of the effect of the hyperparameters governing the prior distribution over weights and biases in a two-layer network having a single input, a single linear output, and 12 hidden units with $\tanh$ activation functions.

9.2.2 Generalized weight decay

A generalization of the simple quadratic regularizer is sometimes used:

$$\Omega(\mathbf{w}) = \frac{\lambda}{2} \sum_{j=1}^{M} |w_j|^q \tag{9.18}$$

where $q = 2$ corresponds to the quadratic regularizer in (9.1). Figure 9.5 shows contours of the regularization function for different values of q.

A regularizer of the form (9.18) with $q = 1$ is known as a *lasso* in the statistics literature (Tibshirani, 1996). For quadratic error functions, it has the property that if λ is sufficiently large, some of the coefficients w_j are driven to zero, leading to a *sparse* model in which the corresponding basis functions play no role. To see this,

Telegram: @uni_k

www.konkur.in

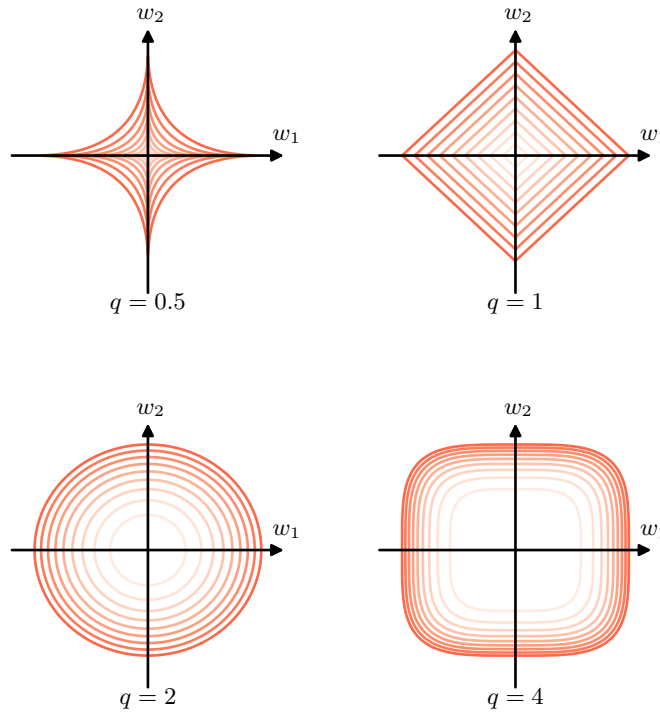


Figure 9.5 Contours of the regularization term in (9.18) for various values of the parameter q.

we first note that minimizing the regularized error function given by

$$E(\mathbf{w}) + \frac{\lambda}{2}\sum_{j=1}^{M}|w_j|^q \tag{9.19}$$

Exercise 9.5

is equivalent to minimizing the unregularized error function $E(\mathbf{w})$ subject to the constraint

$$\sum_{j=1}^{M}|w_j|^q \leqslant \eta \tag{9.20}$$

Appendix C

for an appropriate value of the parameter η, where the two approaches can be related using Lagrange multipliers. The origin of the sparsity can be seen in Figure 9.6, which shows the minimum of the error function, subject to the constraint (9.20). As λ is increased, more parameters will be driven to zero. By comparison, a quadratic regularizer leaves both weight parameters with non-zero values.

Regularization allows complex models to be trained on data sets of limited size without severe over-fitting, essentially by limiting the effective model complexity. However, the problem of determining the optimal model complexity is then shifted from one of finding the appropriate number of learnable parameters to one of determining a suitable value of the regularization coefficient λ. We will discuss the issue of model complexity in the next section.

Telegram: @uni_k

www.konkur.in

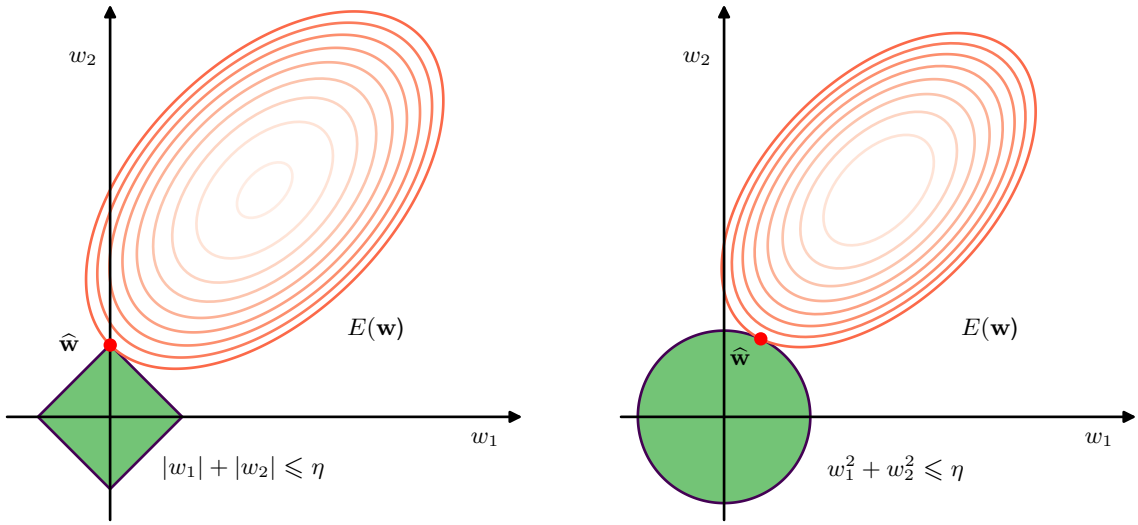


Figure 9.6 Plot of the contours of the unregularized error function (red) along with the constraint region (9.20) for the lasso regularizer $q = 1$ on the left, and the quadratic regularizer $q = 2$ on the right, in which the optimum value for the parameter vector $\mathbf{w}$ is denoted by $\widehat{\mathbf{w}}$. The lasso gives a sparse solution in which $\widehat{w}_1 = 0$, whereas the quadratic regularizer simply reduces w_1 to a smaller value.

9.3. Learning Curves

We have already explored how the generalization performance of a model varies as we change the number of parameters in the model, the size of the data set, and the coefficient of a weight-decay regularization term. Each of these allows for a trade-off between bias and variance to minimize the generalization error. Another factor that influences this trade-off is the learning process itself. During optimization of the error function through gradient descent, the training error typically decreases as the model parameters are updated, whereas the error for hold-out data may be non-monotonic. This behaviour can be visualized using *learning curves*, which plot performance measures such as training set and validation set error as a function of iteration number during an iterative learning process such as stochastic gradient descent. These curves provide insight into the progress of training and also offer a practical methodology for controlling the effective model complexity.

9.3.1 Early stopping

An alternative to regularization as a way of controlling the effective complexity of a network is *early stopping*. The training of deep learning models involves an iterative reduction of the error function defined with respect to a set of training data. Although the error function evaluated using the training set often shows a broadly monotonic decrease as a function of the iteration number, the error measured with respect to held-out data, generally called a validation set, often shows a decrease at

Telegram: @uni_k

www.konkur.in

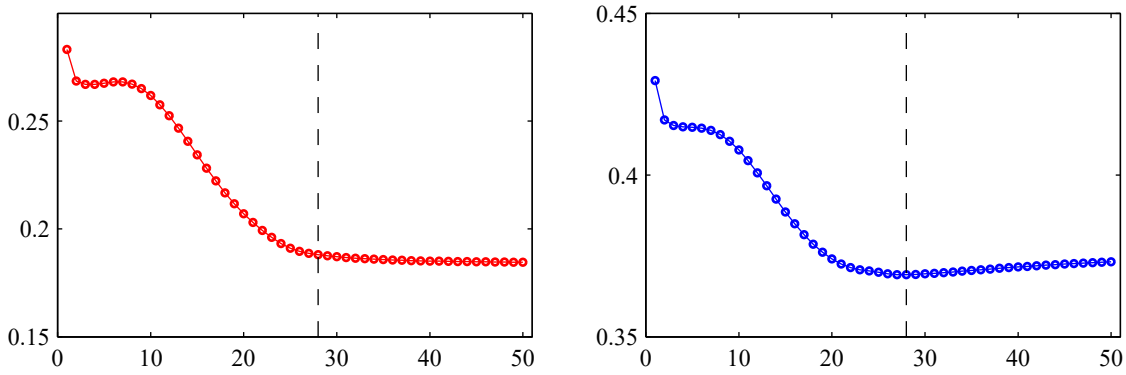


Figure 9.7 An illustration of the behaviour of training set error (left) and validation set error (right) during a typical training session, as a function of the iteration step, for the sinusoidal data set. To achieve the best generalization performance , the training should be stopped at the point shown by the vertical dashed lines, corresponding to the minimum of the validation set error.

first, followed by an increase as the network starts to over-fit. Therefore, to obtain a network with good generalization performance, training should be stopped at the point of smallest error with respect to the validation data set, as indicated in Figure 9.7.

This behaviour of the learning curves is sometimes explained qualitatively in terms of the effective number of parameters in the network. This number starts out small and then grows during training, corresponding to a steady increase in the effective complexity of the model. Stopping training before a minimum of the training error has been reached is a way to limit the effective network complexity.

We can verify this insight for a quadratic error function and show that early stopping should exhibit similar behaviour to regularization using a simple weight-decay term (Bishop, 1995a). This can be understood from Figure 9.8, in which the axes in weight space have been rotated to be parallel to the eigenvectors of the Hessian matrix. If, in the absence of weight decay, the weight vector starts at the origin and proceeds during training along a path that follows the local negative gradient vector, then the weight vector will move initially parallel to the w_2 axis through a point corresponding roughly to $\widehat{\mathbf{w}}$ and then move towards the minimum of the error function $\mathbf{w}_{\mathrm{ML}}$. This follows from the shape of the error surface and the widely differing eigenvalues of the Hessian. Stopping at a point near $\widehat{\mathbf{w}}$ is therefore similar to weight decay. The relationship between early stopping and weight decay can be made quantitative, thereby showing that the quantity $\tau\eta$ (where τ is the iteration index and η is the learning rate parameter) acts like the reciprocal of the regularization parameter λ. The effective number of parameters in the network therefore grows during training.

Section 7.1.1

Exercise 9.6

Telegram: @uni_k

www.konkur.in

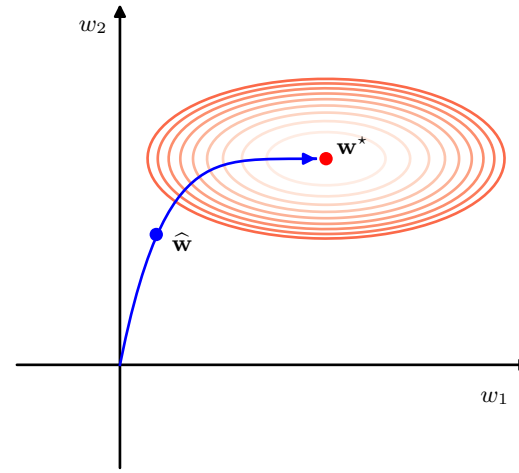


Figure 9.8 A schematic illustration of why early stopping can give similar results to weight decay for a quadratic error function. The ellipses show contours of constant error, and $\mathbf{w}^\star$ denotes the maximum likelihood solution corresponding to the minimum of the unregularized error function. If the weight vector starts at the origin and moves according to the local negative gradient direction, then it will follow the path shown by the curve. By stopping training early, a weight vector $\widehat{\mathbf{w}}$ is found that is qualitatively like that obtained with a simple weight-decay regularizer along with training to the minimum of the regularized error, as can be seen by comparing with Figure 9.3.

9.3.2 Double descent

Section 4.3

The bias–variance trade-off provides insight into the generalization performance of a learnable model as the number of parameters in the model is varied. Models with too few parameters will have a high test set error due to the limited representational capacity (high bias), and as the number of parameters increases, the test error is expected to fall. However, as the number of parameters is increased further, the test error increases again due to over-fitting (high variance). This leads to the conventional belief, widespread in classical statistics, that the number of parameters in the model needs to be limited according to the size of the data set and that for a given training data set, very large models are expected to have poor performance.

Contrary to this expectation, however, modern deep neural networks can have excellent performance even when the number of parameters far exceeds that required to achieve a perfect fit to the training data (Zhang *et al.*, 2016), and the general wisdom in the deep learning community is that bigger models are better. Although early stopping is sometimes used, models may also be trained to zero error and yet still have good performance on test data.

These seemingly contradictory perspectives can be reconciled by examining learning curves and other plots of generalization performance versus model complexity, which reveal a more subtle phenomenon called *double descent* (Belkin *et al.*, 2019). This is illustrated in Figure 9.9, which shows training set and test set errors versus model complexity, as determined by the number of learnable parameters, for a large neural network called ResNet18 (He *et al.*, 2015a), which has 18 layers of parameters trained on an image classification task. The number of weights and biases in the network is varied by changing the 'width parameter', which governs the number of hidden units in each layer. We see that the training error decreases monotonically with increasing complexity of the model, as expected. However, the

Telegram: @uni_k

www.konkur.in

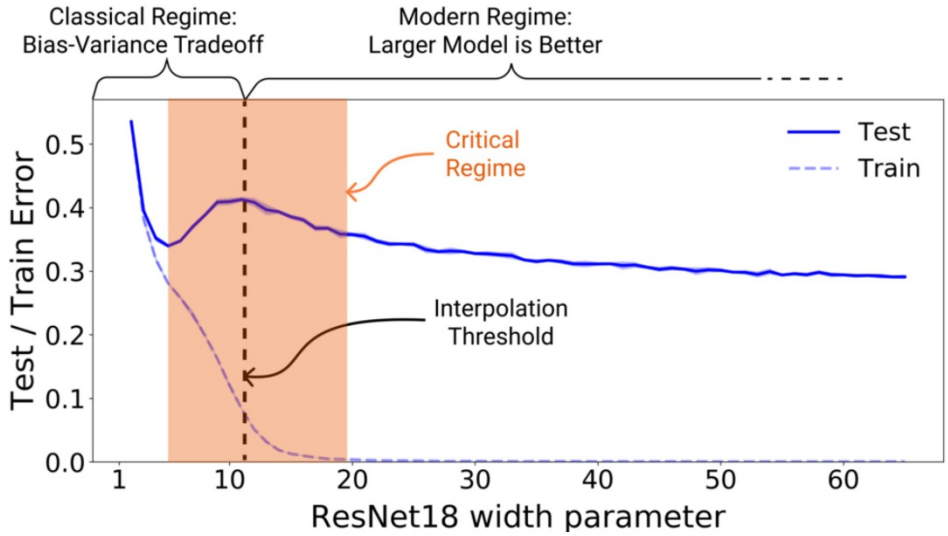


Figure 9.9 Plot of training set and test set errors for a large neural network model called ResNet18 trained on an image classification problem versus the complexity of a model. The horizontal axis represents a hyperparameter governing the number of hidden units and hence the overall number of weights and biases in the network. The vertical dashed line, labelled 'interpolation threshold' indicates the level of model complexity at which the model is capable, in principle, of achieving zero error on the training set. [From Nakkiran *et al.* (2019) with permission.]

test set error decreases at first then increases again and then finally decreases again. This reduction in test set error for very large models continues even after the training set error has reached zero.

This surprising behaviour is more complex than we would expect from the usual bias–variance discussion of classical statistics and exhibits two different regimes of model fitting, as shown schematically in Figure 9.9, corresponding to the classical bias–variance trade-off for small to medium complexity, followed by a further reduction in test set error as we enter a regime of very large models. The transition between the two regimes occurs roughly when the number of parameters in the model is sufficiently large that the model is able to fit the training data exactly (Belkin *et al.*, 2019). Nakkiran *et al.* (2019) define the *effective model complexity* to be the maximum size of training data set on which a model can achieve zero training error, and so double descent arises when the effective model complexity exceeds the number of data points in the training set.

We see similar behaviour if we control model complexity using early stopping, as seen in Figure 9.10. Increasing the number of training epochs increases the effective model complexity, and for a sufficiently large model, double descent is again observed. For such models there are many possible solutions including those that over-fit to the data. It therefore seems to be a property of stochastic gradient descent that the implicit biases that it introduces lead to good generalization performance.

Analogous results are also obtained when a regularization term in the error func-

Telegram: @uni_k

www.konkur.in

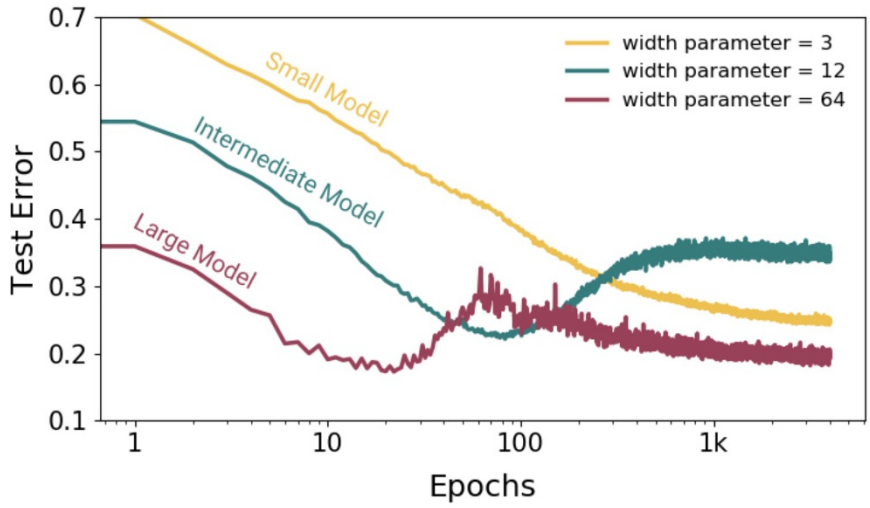


Figure 9.10 Plot of test set error versus number of epochs of gradient descent training for ResNet18 models of various sizes. The effective model complexity increases with the number of training epochs, and the double descent phenomenon is observed for a sufficiently large model. [From Nakkiran *et al.* (2019) with permission.]

tion is used to control complexity. Here the test set error of a large model trained to convergence shows double descent with respect to $1/\lambda$, the inverse regularization parameter, since high λ corresponds to low complexity (Yilmaz and Heckel, 2022).

One ironic consequence of double descent is that it possible to operate in a regime where increasing the size of the training data set could actually reduce performance, contrary to the conventional view that more data is always a good thing. For a model in the critical regime shown in Figure 9.9, an increase in the size of the training set can push the interpolation threshold to the right, leading to a higher test set error. This is confirmed in Figure 9.11, which shows the test set error for a transformer model as a function of the dimensionality of the input space, known as the embedding dimension. Increasing the embedding dimension increases the number of weights and biases in the model and hence increases the model complexity. We see that increasing the training set size from 4,000 to 18,000 data points leads to a curve that is overall much lower. However, for a range of embedding dimensions that correspond to models in the critical complexity regime, increasing the size of the data set can actually reduce generalization performance.

Chapter 12

9.4. Parameter Sharing

Regularization terms, such as the L_2 regularizer $\|\mathbf{w}\|^2$, help to reduce over-fitting by encouraging weight values to be close to zero. Another way to reduce network complexity is to impose hard constraints on the weights by forming them into groups and requiring that all weights within each group share the same value, in which the shared

Telegram: @uni_k

www.konkur.in

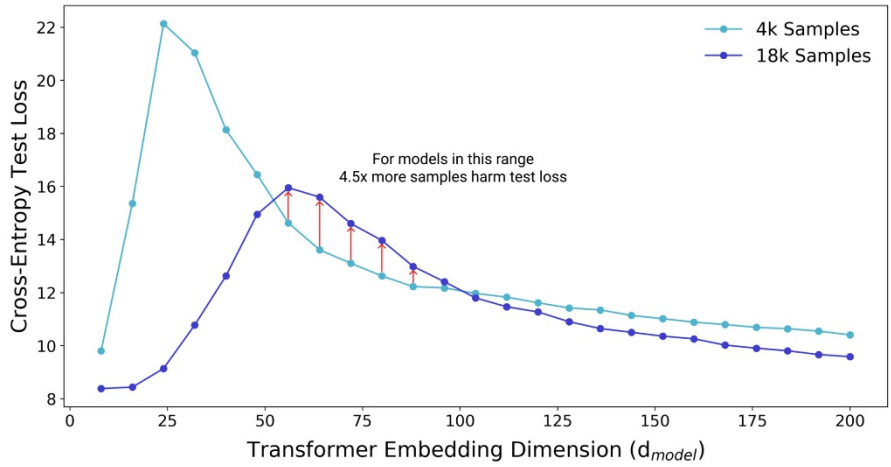


Figure 9.11 Plot of test set error for a large transformer model versus the embedding dimension, which controls the number of parameters in the model. Increasing the size of the training set from 4,000 to 18,000 samples generally leads to a lower test set error, but for some intermediate values of model complexity, there can be an increase in the error, as shown by the vertical red arrows. [From Nakkiran *et al.* (2019) with permission.]

value is learned from data. This is known as *weight sharing* or *parameter sharing* or *parameter tying*. It means that the number of degrees of freedom is smaller than the number of connections in the network. Usually this is introduced as a way to encode inductive bias into a network to express some known invariances. Evaluating the error function gradients for such networks can be done using a small modification to backpropagation although in practice this is handled implicitly through automatic differentiation. We will make extensive use of parameter sharing when we discuss convolutional neural networks. Parameter sharing is applicable, however, only to particular problems in which the form of the constraints can be specified in advance.

Exercise 9.7

Chapter 10

9.4.1 Soft weight sharing

Instead of using a hard constraint that forces sets of model parameters to be equal, Nowlan and Hinton (1992) introduced a form of *soft weight sharing* in which a regularization term encourages groups of weights to have similar values. Furthermore, the division of weights into groups, the mean weight value for each group, and the spread of values within the groups are all determined as part of the learning process.

Recall that the simple-weight decay regularizer in (9.1) can be viewed as the negative log of a Gaussian prior distribution over the weights. This encourages all the weights to converge towards a single value of zero. We can instead encourage the weight values to form several groups, rather than just one group, by considering a probability distribution that is a *mixture* of Gaussians. The means $\{\mu_j\}$ and variances $\{\sigma_j^2\}$ of the Gaussian components, as well as the mixing coefficients $\{\pi_j\}$, will be considered as adjustable parameters to be determined as part of the learning process.

Section 3.2.9

Telegram: @uni_k

www.konkur.in

Thus, we have a probability density of the form

$$p(\mathbf{w}) = \prod_i \left\{ \sum_{j=1}^{K} \pi_j \mathcal{N}(w_i|\mu_j, \sigma_j^2) \right\} \tag{9.21}$$

where K is the number of components in the mixture. Taking the negative logarithm then leads to a regularization function of the form

$$\Omega(\mathbf{w}) = -\sum_i \ln \left(\sum_{j=1}^{K} \pi_j \mathcal{N}(w_i|\mu_j, \sigma_j^2) \right). \tag{9.22}$$

The total error function is then given by

$$\widetilde{E}(\mathbf{w}) = E(\mathbf{w}) + \lambda \Omega(\mathbf{w}) \tag{9.23}$$

where λ is the regularization coefficient.

This error is minimized jointly with respect to the weights $\{w_i\}$ and with respect to the parameters $\{\pi_j, \mu_j, \sigma_j\}$ of the mixture model. This can be done using gradient descent, which requires that we evaluate the derivatives of $\Omega(\mathbf{w})$ with respect to all the learnable parameters. To do this, it is convenient to regard the $\{\pi_j\}$ as *prior* probabilities for each component to have generated a weight value, and to introduce the corresponding posterior probabilities, which are given by Bayes' theorem:

Exercise 9.8

$$\gamma_j(w_i) = \frac{\pi_j \mathcal{N}(w_i|\mu_j, \sigma_j^2)}{\sum_k \pi_k \mathcal{N}(w_i|\mu_k, \sigma_k^2)}. \tag{9.24}$$

The derivatives of the total error function with respect to the weights are then given by

Exercise 9.9

$$\frac{\partial \widetilde{E}}{\partial w_i} = \frac{\partial E}{\partial w_i} + \lambda \sum_j \gamma_j(w_i) \frac{(w_i - \mu_j)}{\sigma_j^2}. \tag{9.25}$$

The effect of the regularization term is therefore to pull each weight towards the centre of the jth Gaussian, with a force proportional to the posterior probability of that Gaussian for the given weight. This is precisely the kind of effect that we are seeking.

Derivatives of the error with respect to the centres of the Gaussians are also easily computed to give

Exercise 9.10

$$\frac{\partial \widetilde{E}}{\partial \mu_j} = \lambda \sum_i \gamma_j(w_i) \frac{(\mu_j - w_i)}{\sigma_j^2} \tag{9.26}$$

which has a simple intuitive interpretation, because it pushes μ_j towards an average of the weight values, weighted by the posterior probabilities that the respective weight parameters were generated by component j.

Telegram: @uni_k

www.konkur.in

To ensure that the variances $\{\sigma_j^2\}$ remain positive, we introduce new variables $\{\xi_j\}$ defined by

$$\sigma_j^2 = \exp(\xi_j) \tag{9.27}$$

and an unconstrained minimization is performed with respect to the $\{\xi_j\}$. The associated derivatives are then given by

Exercise 9.11

$$\frac{\partial \widetilde{E}}{\partial \xi} = \frac{\lambda}{2} \sum_i \gamma_j(w_i) \left(1 - \frac{(w_i - \mu_j)^2}{\sigma_j^2} \right). \tag{9.28}$$

This process drives σ_j towards a weighted average of the squared deviations of the weights around the corresponding centre μ_j, where the weighting coefficients are again given by the posterior probability that each weight is generated by component j.

For the derivatives with respect to the mixing coefficients π_j, we need to take account of the constraints

$$\sum_j \pi_j = 1, \qquad 0 \leqslant \pi_i \leqslant 1, \tag{9.29}$$

which follow from the interpretation of the π_j as prior probabilities. This can be done by expressing the mixing coefficients in terms of a set of auxiliary variables $\{\eta_j\}$ using the *softmax* function given by

$$\pi_j = \frac{\exp(\eta_j)}{\sum_{k=1}^{K} \exp(\eta_k)}. \tag{9.30}$$

The derivatives of the regularized error function with respect to the $\{\eta_j\}$ then take the form

Exercise 9.12

$$\frac{\partial \widetilde{E}}{\partial \eta_j} = \lambda \sum_i \{\pi_j - \gamma_j(w_i)\}. \tag{9.31}$$

We see that π_j is therefore driven towards the average posterior probability for mixture component j.

A different application of soft weight sharing (Lasserre, Bishop, and Minka, 2006) introduces a principled approach that combines the unsupervised training of a generative model with the supervised training of a corresponding discriminative model. This is useful in situations where we have a significant amount of unlabelled data but where labelled data is in short supply. The generative model has the advantage that all of the data can be used to determine its parameters, whereas only the labelled examples directly inform the parameters of the discriminative model. However, a discriminative model can achieve better generalization when there is model mis-specification, in other words when the model does not exactly describe the true distribution that generates the data, as is typically the case. By introducing a soft tying of the parameters of the two models, we obtain a well-defined hybrid of generative and discriminative approaches that can be robust to model mis-specification while also benefiting from being trained on unlabelled data.

Telegram: @uni_k

www.konkur.in

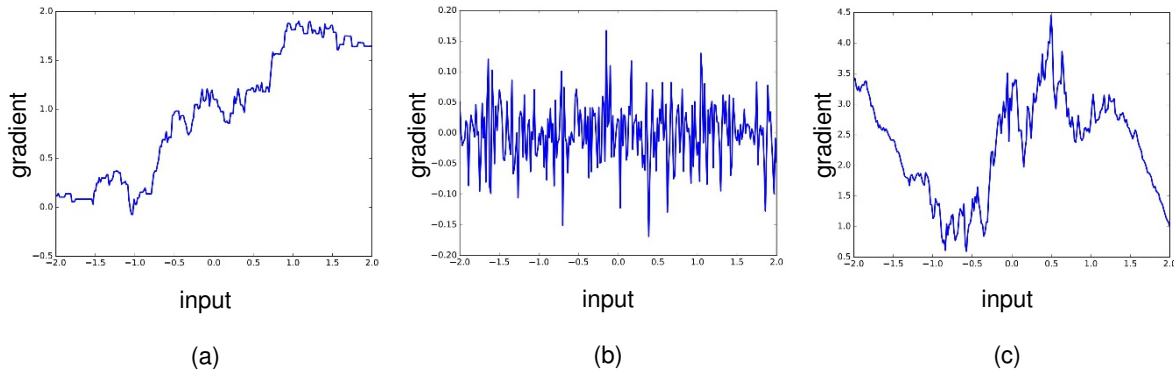


Figure 9.12 Plots of the Jacobian for networks with a single input and a single output, showing (a) a network with two layers of weights, (b) a network with 25 layers of weights, and (c) a network with 51 layers of weights together with residual connections. [From Balduzzi *et al.* (2017) with permission.]

9.5. Residual Connections

The representational power of deep neural networks stems in large part from the use of multiple layers of processing, and it has been observed that increasing the number of layers in a network can increase generalization performance significantly. We have also seen how batch normalization, along with careful initialization of the weights and biases, can help address the problem of vanishing or exploding gradients in deep networks. However, even with batch normalization, it becomes increasingly difficult to train networks with a large number of layers.

Section 7.4.2
Section 7.2.5

One explanation for this phenomenon is called *shattered gradients* (Balduzzi *et al.*, 2017). We have seen that the representational capabilities of neural networks increase exponentially with depth. With ReLU activation functions, there is an exponential increase in the number of linear regions that the network can represent. However, a consequence of this is a proliferation of discontinuities in the gradient of the error function. This is illustrated for networks with a single input variable and a single output variable in Figure 9.12. Here the derivative of the output variable with respect to the input variable (the Jacobian of the network) is plotted as a function of the input variable. From the chain rule of calculus, these derivatives determine the gradients of the error function surface. We see that for deep networks, extremely small changes in the weight parameters in the early layers of the network can produce significant changes in the gradient. Iterative gradient-based optimization algorithms assume that the gradient varies smoothly across parameter space, and hence this 'shattered gradient' effect can render training ineffective in very deep networks.

Section 6.3

An important modification to the architecture of neural networks that greatly assists in training very deep networks is that of *residual connections* (He *et al.*, 2015a), which are a particular form of *skip-layer connections*. Consider a neural network

Telegram: @uni_k

www.konkur.in

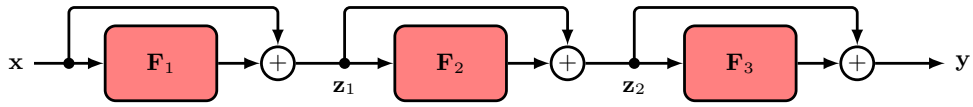


Figure 9.13 A residual network consisting of three residual blocks, corresponding to the sequence of transformations (9.35) to (9.37).

that consists of a sequence of three layers of the form

$$\mathbf{z}_1 = \mathbf{F}_1(\mathbf{x}) \tag{9.32}$$
$$\mathbf{z}_2 = \mathbf{F}_2(\mathbf{z}_1) \tag{9.33}$$
$$\mathbf{y} = \mathbf{F}_3(\mathbf{z}_2). \tag{9.34}$$

Here the functions $\mathbf{F}_l(\cdot)$ might simply consist of a linear transformation followed by a ReLU activation function or they might be more complex with multiple linear, activation function, and normalization layers. A residual connection consists simply of adding the input to each function back onto the output to give

$$\mathbf{z}_1 = \mathbf{F}_1(\mathbf{x}) + \mathbf{x} \tag{9.35}$$
$$\mathbf{z}_2 = \mathbf{F}_2(\mathbf{z}_1) + \mathbf{z}_1 \tag{9.36}$$
$$\mathbf{y} = \mathbf{F}_3(\mathbf{z}_2) + \mathbf{z}_2. \tag{9.37}$$

Each combination of a function and a residual connection, such as $\mathbf{F}_1(\mathbf{x}) + \mathbf{x}$, is called a *residual block*. A *residual network*, also known as a *ResNet*, consists of multiple layers of such blocks in sequence. A modified network with residual connections is illustrated in Figure 9.13. A residual block can easily generate the identity transformation, if the parameters in the nonlinear function are small enough for the function outputs to become close to zero.

The term 'residual' refers to the fact that in each block the function learns the residual between the identity map and the desired output, which we can see by rearranging the residual transformation:

$$\mathbf{F}_l(\mathbf{z}_{l-1}) = \mathbf{z}_l - \mathbf{z}_{l-1}. \tag{9.38}$$

The gradients in a network with residual connections are much less sensitive to input values compared to a standard deep network, as seen in Figure 9.12(c).

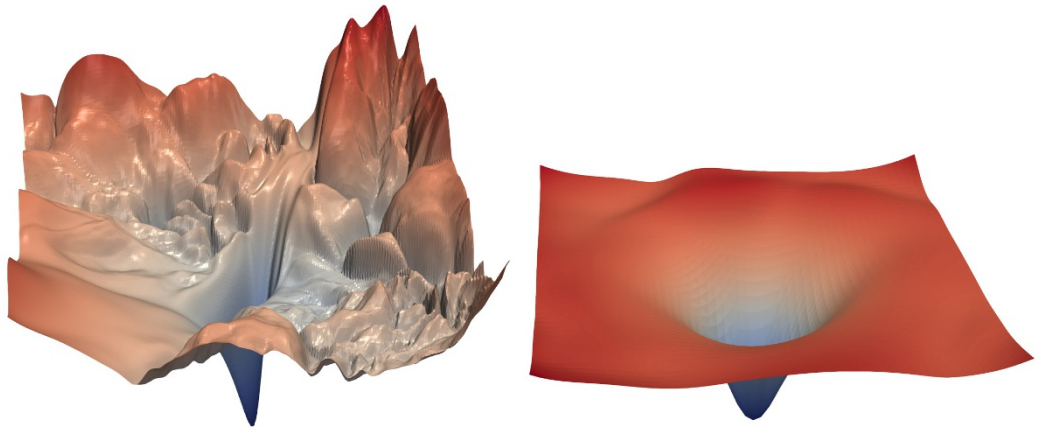
Li *et al.* (2017) developed a way to visualize error surfaces directly, which showed that the effect of the residual connections is to create smoother error function surfaces, as shown in Figure 9.14. It is usual to include batch normalization layers in a residual network, as together they significantly reduce the issue of vanishing and exploding gradients. He *et al.* (2015a) showed that including residual connections allows very deep networks, potentially having hundreds of layers, to be trained effectively.

Further insight into the way residual connections encourage smooth error surfaces can be obtained if we combine (9.35), (9.36), and (9.37) to give a single overall equation for the whole network:

$$\mathbf{y} = \mathbf{F}_3(\mathbf{F}_2(\mathbf{F}_1(\mathbf{x}) + \mathbf{x}) + \mathbf{z}_1) + \mathbf{z}_2. \tag{9.39}$$

Telegram: @uni_k

www.konkur.in



(a) (b)

Figure 9.14 (a) A visualization of the error surface for a network with 56 layers. (b) The same network with the inclusion of residual connections, showing the smoothing effect that comes from the residual connections. [From Li *et al.* (2017) with permission.]

Exercise 9.13

We can now substitute for the intermediate variables $\mathbf{z}_1$ and $\mathbf{z}_2$ to give an expression for the network output as a function of the input $\mathbf{x}$:

$$\begin{aligned}\mathbf{y} = &\mathbf{F}_3(\mathbf{F}_2(\mathbf{F}_1(\mathbf{x}) + \mathbf{x}) + \mathbf{F}_1(\mathbf{x}) + \mathbf{x}) \\ &+ \mathbf{F}_2(\mathbf{F}_1(\mathbf{x}) + \mathbf{x})) \\ &+ \mathbf{F}_1(\mathbf{x}) + \mathbf{x}. \end{aligned} \tag{9.40}$$

This expanded form of the residual network is depicted in Figure 9.15. We see that the overall function consists of multiple networks acting in parallel and that these include networks with fewer layers. The network has the representational capability of a deep network, since it contains such a network as a special case. However, the error surface is moderated by a combination of shallow and deep sub-networks.

Note that the skip-layer connections defined by (9.40) require the input and all the intermediate variables to have the same dimensionality so that they can be added. We can change the dimensionality at some point in the network by including a non-square matrix $\mathbf{W}$ of learnable parameters in the form

$$\mathbf{z}_l = \mathbf{F}_l(\mathbf{z}_{l-1}) + \mathbf{W}\mathbf{z}_{l-1}. \tag{9.41}$$

So far we have not been specific about the form of the learnable nonlinear functions $\mathbf{F}_l(\cdot)$. The simplest choice would be a standard neural network that alternates between layers consisting of a learnable linear transformation and a fixed nonlinear activation function such as ReLU. This opens two possibilities for placing the residual connections, as shown in Figure 9.16. In version (a) the quantities being added are always non-negative since they are given by the outputs of ReLU layers, and so to allow for both positive and negative values, version (b) is more commonly used.

Telegram: @uni_k

www.konkur.in

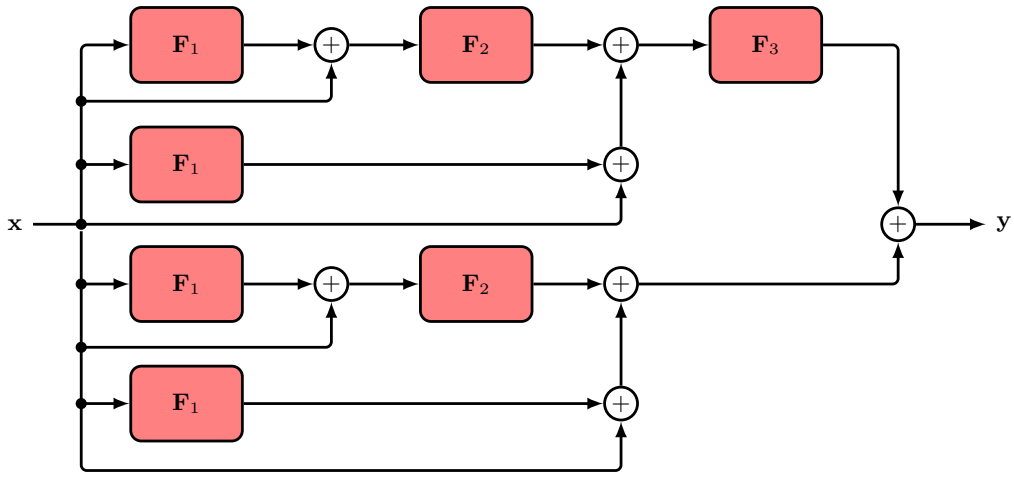


Figure 9.15 The same network as in Figure 9.13, shown here in expanded form.

9.6. Model Averaging

If we have several different models trained to solve the same problem then instead of trying to select the single best model, we can often improve generalization by averaging the predictions made by the individual models. Such combinations of models are sometimes called *committees* or *ensembles*. For models that produce probabilistic outputs, the predicted distribution is the average of the predictions from each model:

$$p(\mathbf{y}|\mathbf{x}) = \frac{1}{L}\sum_{l=1}^{L} p_l(\mathbf{y}|\mathbf{x}) \tag{9.42}$$

where $p_l(\mathbf{y}|\mathbf{x})$ is the output of model l and L is the total number of models.

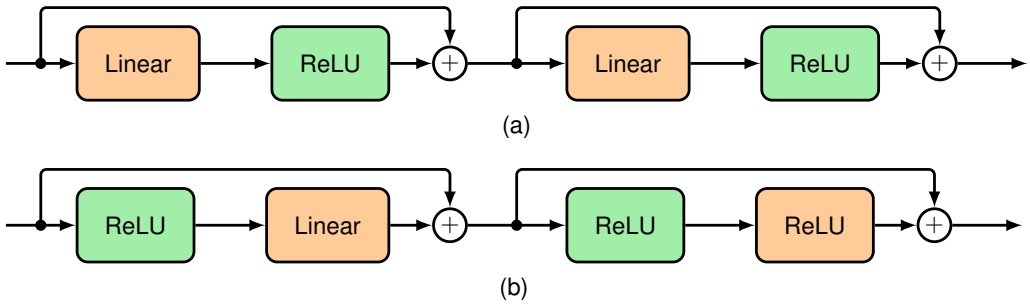


Figure 9.16 Two alternative ways to include residual network connections into a standard feed-forward network that alternates between learnable linear layers and nonlinear ReLU activation functions.

Telegram: @uni_k

www.konkur.in

Section 4.3

This averaging process can be motivated by considering the trade-off between bias and variance. Recall from Figure 4.7 that when we trained multiple polynomials using the sinusoidal data and then averaged the resulting functions, the contribution arising from the variance term tended to cancel, leading to improved predictions.

In practice, of course, we have only a single data set, and so we have to find a way to introduce variability between the different models within the committee. One approach is to use *bootstrap* data sets, in which multiple data sets are created as follows. Suppose our original data set consists of N data points $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$. We can create a new data set $\mathbf{X}_{\mathrm{B}}$ by drawing N points at random from $\mathbf{X}$, with replacement, so that some points in $\mathbf{X}$ may be replicated in $\mathbf{X}_{\mathrm{B}}$, whereas other points in $\mathbf{X}$ may be absent from $\mathbf{X}_{\mathrm{B}}$. This process can be repeated L times to generate L data sets each of size N and each obtained by sampling from the original data set $\mathbf{X}$. Each data set can then be used to train a model, and the predictions of the resulting models are averaged. This procedure is known as bootstrap aggregation or *bagging* (Breiman, 1996). An alternative approach to forming an ensemble is to use the original data set to train multiple different models having different architectures.

We can analyse the benefits of ensemble predictions by considering a regression problem with an input vector $\mathbf{x}$ and a single output variable y. Suppose we have a set of trained models $y_1(\mathbf{x}), \ldots, y_M(\mathbf{x})$, and we form a committee prediction given by

$$y_{\mathrm{COM}}(\mathbf{x}) = \frac{1}{M}\sum_{m=1}^{M} y_m(\mathbf{x}). \tag{9.43}$$

If the true function that we are trying to predict is given by $h(\mathbf{x})$, then the output of each of the models can be written as the true value plus an error:

$$y_m(\mathbf{x}) = h(\mathbf{x}) + \epsilon_m(\mathbf{x}). \tag{9.44}$$

The average sum-of-squares error then takes the form

$$\mathbb{E}_{\mathbf{x}}\left[\{y_m(\mathbf{x}) - h(\mathbf{x})\}^2\right] = \mathbb{E}_{\mathbf{x}}\left[\epsilon_m(\mathbf{x})^2\right] \tag{9.45}$$

where $\mathbb{E}_{\mathbf{x}}[\cdot]$ denotes a frequentist expectation with respect to the distribution of the input vector $\mathbf{x}$. The average error made by the models acting individually is therefore

$$E_{\mathrm{AV}} = \frac{1}{M}\sum_{m=1}^{M} \mathbb{E}_{\mathbf{x}}\left[\epsilon_m(\mathbf{x})^2\right]. \tag{9.46}$$

Similarly, the expected error from the committee (9.43) is given by

$$\begin{aligned} E_{\mathrm{COM}} &= \mathbb{E}_{\mathbf{x}}\left[\left\{\frac{1}{M}\sum_{m=1}^{M} y_m(\mathbf{x}) - h(\mathbf{x})\right\}^2\right] \\ &= \mathbb{E}_{\mathbf{x}}\left[\left\{\frac{1}{M}\sum_{m=1}^{M} \epsilon_m(\mathbf{x})\right\}^2\right]. \end{aligned} \tag{9.47}$$

Telegram: @uni_k

www.konkur.in

If we assume that the errors have zero mean and are uncorrelated, so that

$$\mathbb{E}_{\mathbf{x}}\left[\epsilon_m(\mathbf{x})\right] = 0 \tag{9.48}$$

$$\mathbb{E}_{\mathbf{x}}\left[\epsilon_m(\mathbf{x})\epsilon_l(\mathbf{x})\right] = 0, \qquad m \neq l \tag{9.49}$$

Exercise 9.14 then we obtain

$$E_{\mathrm{COM}} = \frac{1}{M} E_{\mathrm{AV}}. \tag{9.50}$$

This apparently dramatic result suggests that the average error of a model can be reduced by a factor of M simply by averaging M versions of the model. Unfortunately, it depends on the key assumption that the errors due to the individual models are uncorrelated. In practice, the errors are typically highly correlated, and the reduction in the overall error is generally much smaller. It can, however, be shown that the expected committee error will not exceed the expected error of the constituent models, so that $E_{\mathrm{COM}} \leqslant E_{\mathrm{AV}}$. *Exercise 9.15*

A somewhat different approach to model combination, known as *boosting* (Freund and Schapire, 1996), combines multiple 'base' classifiers to produce a form of committee whose performance can be significantly better than that of any of the base classifiers. Boosting can give good results even if the base classifiers perform only slightly better than random. The principal difference between boosting and the committee methods, such as bagging as discussed above, is that the base classifiers are trained in sequence and each base classifier is trained using a weighted form of the data set in which the weighting coefficient associated with each data point depends on the performance of the previous classifiers. In particular, points that are misclassified by one of the base classifiers are given a greater weight when used to train the next classifier in the sequence. Once all the classifiers have been trained, their predictions are then combined through a weighted majority voting scheme.

In practice, the major drawback of all model combination methods is that multiple models have to be trained and then predictions have to be evaluated for all the models, thereby increasing the computational cost of both training and inference. How significant this depends on the specific application scenario.

9.6.1 Dropout

A widely used and very effective form of regularization known as *dropout* (Srivastava *et al.*, 2014) can be viewed as an implicit way to perform approximate model averaging over exponentially many models without having to train multiple models individually. It has broad applicability and is computationally cheap. Dropout is one of the most effective forms of regularization and is widely used in applications.

The central idea of dropout is to delete nodes from the network, including their connections, at random during training. Each time a data point is presented to the network, a new random choice is made for which nodes to omit. Figure 9.17 shows a simple network along with examples of pruned networks in which subsets of nodes have been omitted.

Dropout is applied to both hidden nodes and input nodes, but not outputs, and is equivalent to setting the output of a dropped node to zero. It can be implemented by defining a mask vector $R_i \in \{0, 1\}$ which multiplies the activation of the non-output

Telegram: @uni_k

www.konkur.in

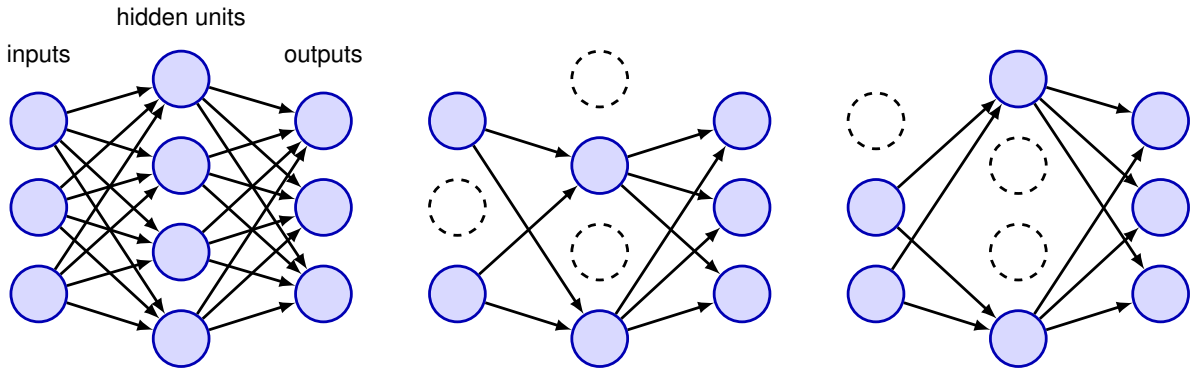


Figure 9.17 A neural network on the left along with two examples of pruned networks in which a random subset of nodes have been omitted.

node i for data point n, whose values are set to 1 with probability ρ. A value of $\rho = 0.5$ seems to work well for the hidden nodes, whereas for the inputs a value of $\rho = 0.8$ is typically used.

During training, as each data point is presented to the network, a new mask is created, and the forward and backward propagation steps are applied on that pruned network to create error function gradients, which are then used to update the weights, for example by stochastic gradient descent. If the data points are grouped into mini-batches then the gradients are averaged over the data points in each mini-batch before applying the weight update. For a network with M non-output nodes, there are 2^M pruned networks, and so only a small fraction of these networks will ever be considered during training. This differs from conventional ensemble methods in which each of the networks in the ensemble is independently trained to convergence. Another difference is that the exponentially many networks that are implicitly being trained with dropout are not independent but share their parameter values with the full network, and hence with each other. Note that training can take longer with dropout since the individual parameter updates are very noisy. Also, because the error function is intrinsically noisy, it is harder to confirm that the optimization algorithm is working correctly just by looking for a decreasing error function during training.

Section 7.2.4

Once training is complete, predictions can in principle be made by applying the ensemble rule (9.42), which in this case takes the form

$$p(\mathbf{y}|\mathbf{x}) = \sum_{\mathbf{R}} p(\mathbf{R}) p(\mathbf{y}|\mathbf{x}, \mathbf{R}) \tag{9.51}$$

where the sum is over the exponentially large space of masks, and $p(\mathbf{y}|\mathbf{x}, \mathbf{R})$ is the predictive distribution from the network with mask $\mathbf{R}$. Because this summation is intractable, it can be approximated by sampling a small number of masks, and in practice, as few as 10 or 20 masks can be sufficient to obtain good results. This procedure is known as *Monte Carlo dropout*.

Telegram: @uni_k

www.konkur.in

An even simpler approach is to make predictions using the trained network with no nodes masked out, and to re-scale the weights in the network so that the expected input to each node is roughly the same during testing as it would be during training, compensating for the fact that in training a proportion of the nodes would be missing. Thus, if a node is present with probability ρ during training, then during testing the output weights from that node would be multiplied by ρ before using the network to make predictions.

A different motivation for dropout comes from the Bayesian perspective. In a fully Bayesian treatment, we would make predictions by averaging over all possible 2^M network models, with each network weighted by its posterior probability. Computationally, this would be prohibitively expensive, both during training when evaluating the posterior probabilities and during testing when computing the weighted predictions. Dropout approximates this model averaging by giving an equal weight to each possible model.

Section 2.6

Further intuition behind dropout comes from its role in reducing over-fitting. In a standard network, the parameters can become tuned to noise on individual data points, with hidden nodes becoming over-specialized. Each node adjusts its weights to minimize the error, given the outputs of other nodes, leading to co-adaptation of nodes in a way that might not generalize to new data. With dropout, each node cannot rely on the presence of other specific nodes and must instead make useful contributions in a broad range of contexts, thereby reducing co-adaptation and specialization. For a simple linear regression model trained using least squares, dropout regularization is equivalent to a modified form of quadratic regularization.

Exercise 9.18

Exercises

9.1 ($\star$) By considering each of the four group axioms in turn, show that the set of all possible rotations of a square through (positive or negative) multiples of $90°$, together with the binary operation of composing rotations, forms a group. Similarly, show that the set of all continuous translations of an object in a two-dimensional plane also forms a group.

Section 9.1.3

9.2 ($\star\star$) Consider a linear model of the form

$$y(\mathbf{x}, \mathbf{w}) = w_0 + \sum_{i=1}^{D} w_i x_i \tag{9.52}$$

together with a sum-of-squares error function of the form

$$E_D(\mathbf{w}) = \frac{1}{2} \sum_{n=1}^{N} \{y(\mathbf{x}_n, \mathbf{w}) - t_n\}^2 . \tag{9.53}$$

Now suppose that Gaussian noise ϵ_i with zero mean and variance σ^2 is added independently to each of the input variables x_i. By making use of $\mathbb{E}[\epsilon_i] = 0$ and $\mathbb{E}[\epsilon_i \epsilon_j] = \delta_{ij}\sigma^2$, show that minimizing E_D averaged over the noise distribution is

Telegram: @uni_k

www.konkur.in

equivalent to minimizing the sum-of-squares error for noise-free input variables with the addition of a weight-decay regularization term, in which the bias parameter w_0 is omitted from the regularizer.

9.3 ($\star\star$) Consider an error function that consists simply of the quadratic regularizer

$$\Omega(\mathbf{w}) = -\frac{1}{2}\mathbf{w}^{\mathrm{T}}\mathbf{w} \tag{9.54}$$

together with the gradient descent update formula

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau+1)} - \eta\nabla\Omega(\mathbf{w}). \tag{9.55}$$

By considering the limit of infinitesimal updates, write down a corresponding differential equation for the evolution of $\mathbf{w}$. Write down the solution of this equation starting from an initial value $\mathbf{w}_0$, and show that the elements of $\mathbf{w}$ decay exponentially to zero.

9.4 ($\star$) Verify that the network function defined by (9.6) and (9.7) is invariant under the transformation (9.8) applied to the inputs, provided the weights and biases are simultaneously transformed using (9.9) and (9.10). Similarly, show that the network outputs can be transformed according to (9.11) by applying the transformation (9.12) and (9.13) to the second-layer weights and biases.

Appendix C **9.5** ($\star\star$) By using Lagrange multipliers, show that minimizing the regularized error function given by (9.19) is equivalent to minimizing the unregularized error function $E(\mathbf{w})$ subject to the constraint (9.20). Discuss the relationship between the parameters η and λ.

9.6 ($\star\star\star$) Consider a quadratic error function of the form

$$E = E_0 + \frac{1}{2}(\mathbf{w} - \mathbf{w}^\star)^{\mathrm{T}}\mathbf{H}(\mathbf{w} - \mathbf{w}^\star) \tag{9.56}$$

where $\mathbf{w}^\star$ represents the minimum, and the Hessian matrix $\mathbf{H}$ is positive definite and constant. Suppose the initial weight vector $\mathbf{w}^{(0)}$ is chosen to be at the origin and is updated using simple gradient descent:

$$\mathbf{w}^{(\tau)} = \mathbf{w}^{(\tau-1)} - \rho\nabla E \tag{9.57}$$

where τ denotes the step number, and ρ is the learning rate (which is assumed to be small). Show that, after τ steps, the components of the weight vector parallel to the eigenvectors of $\mathbf{H}$ can be written

$$w_j^{(\tau)} = \{1 - (1 - \rho\eta_j)^\tau\}\, w_j^\star \tag{9.58}$$

where $w_j = \mathbf{w}^{\mathrm{T}}\mathbf{u}_j$, and $\mathbf{u}_j$ and η_j are the eigenvectors and eigenvalues of $\mathbf{H}$, respectively, defined by

$$\mathbf{H}\mathbf{u}_j = \eta_j\mathbf{u}_j. \tag{9.59}$$

Telegram: @uni_k

www.konkur.in

Show that as $\tau \to \infty$, this gives $\mathbf{w}^{(\tau)} \to \mathbf{w}^\star$ as expected, provided $|1 - \rho\eta_j| < 1$. Now suppose that training is halted after a finite number τ of steps. Show that the components of the weight vector parallel to the eigenvectors of the Hessian satisfy

$$w_j^{(\tau)} \simeq w_j^\star \quad \text{when} \quad \eta_j \gg (\rho\tau)^{-1} \tag{9.60}$$

$$|w_j^{(\tau)}| \ll |w_j^\star| \quad \text{when} \quad \eta_j \ll (\rho\tau)^{-1}. \tag{9.61}$$

This result shows that $(\rho\tau)^{-1}$ plays an analogous role to the regularization parameter λ in weight decay.

9.7 ($\star\star$) Consider a neural network in which multiple weights are constrained to have the same value. Discuss how the standard backpropagation algorithm must be modified to ensure that such constraints are satisfied when evaluating the derivatives of an error function with respect to the adjustable parameters in the network.

9.8 ($\star$) Consider a mixture distribution defined by

$$p(w) = \sum_{j=1}^{M} \pi_j \mathcal{N}(w|\mu_j, \sigma_j^2) \tag{9.62}$$

in which $\{\pi_j\}$ can be viewed as prior probabilities $p(j)$ for the corresponding Gaussian components. Using Bayes' theorem, show that the corresponding posterior probabilities $p(j|w)$ are given by (9.24).

9.9 ($\star\star$) Using (9.21), (9.22), (9.23), and (9.24) verify the result (9.25).

9.10 ($\star\star$) Using (9.21), (9.22), (9.23), and (9.24) verify the result (9.26).

9.11 ($\star\star$) Using (9.21), (9.22), (9.23), and (9.24) verify the result (9.28).

9.12 ($\star\star$) Show that the derivatives of the mixing coefficients $\{\pi_k\}$ defined by (9.30) with respect to the auxiliary parameters $\{\eta_j\}$ are given by

$$\frac{\partial \pi_k}{\partial \eta_j} = \delta_{jk}\pi_j - \pi_j\pi_k. \tag{9.63}$$

Hence, by making use of the constraint $\sum_k \gamma_k(w_i) = 1$ for all i, derive the result (9.31).

9.13 ($\star$) Verify that combining (9.35), (9.36), and (9.37) gives a single overall equation for the whole network in the form (9.40).

9.14 ($\star\star$) The expected sum-of-squares error E_{AV} for a simple committee model can be defined by (9.46), and the expected error of the committee itself is given by (9.47). Assuming that the individual errors satisfy (9.48) and (9.49), derive the result (9.50).

Telegram: @uni_k

www.konkur.in

9.15 (⋆⋆) By making use of Jensen's inequality (2.102) for the special case of the convex function $f(x) = x^2$, show that the average expected sum-of-squares error E_{AV} of the members of a simple committee model, given by (9.46), and the expected error E_{COM} of the committee itself, given by (9.47), satisfy

$$E_{\mathrm{COM}} \leqslant E_{\mathrm{AV}}. \tag{9.64}$$

9.16 (⋆⋆) By making use of Jensen's in equality (2.102), show that the result (9.64) derived in the previous exercise holds for any error function $E(y)$, not just sum-of-squares, provided it is a convex function of y.

9.17 (⋆⋆) Consider a committee in which we allow unequal weighting of the constituent models, so that

$$y_{\mathrm{COM}}(\mathbf{x}) = \sum_{m=1}^{M} \alpha_m y_m(\mathbf{x}). \tag{9.65}$$

To ensure that the predictions $y_{\mathrm{COM}}(\mathbf{x})$ remain within sensible limits, suppose that we require that they be bounded at each value of $\mathbf{x}$ by the minimum and maximum values given by any of the members of the committee, so that

$$y_{\min}(\mathbf{x}) \leqslant y_{\mathrm{COM}}(\mathbf{x}) \leqslant y_{\max}(\mathbf{x}). \tag{9.66}$$

Show that a necessary and sufficient condition for this constraint is that the coefficients α_m satisfy

$$\alpha_m \geqslant 0, \qquad \sum_{m=1}^{M} \alpha_m = 1. \tag{9.67}$$

9.18 (⋆⋆⋆) Here we explore the effect of dropout regularization on a simple linear regression model trained using least squares. Consider a model of the form

$$y_k = \sum_{i=1}^{D} w_{ki} x_i \tag{9.68}$$

along with a sum-of-squares error function given by

$$E(\mathbf{W}) = \sum_{n=1}^{N} \sum_{k=1}^{K} \left\{ t_{nk} - \sum_{i=1}^{D} w_{ki} R_{ni} x_{ni} \right\}^2 \tag{9.69}$$

where the elements $R_{ni} \in \{0, 1\}$ of the dropout matrix are chosen randomly from a Bernoulli distribution with parameter ρ. We now take an expectation over the distribution of random dropout parameters. Show that

$$\mathbb{E}[R_{ni}] = \rho \tag{9.70}$$

$$\mathbb{E}[R_{ni} R_{nj}] = \delta_{ij}\rho + (1 - \delta_{ij})\rho^2. \tag{9.71}$$

Telegram: @uni_k

www.konkur.in

Hence, show that the expected error function for this dropout model is given by

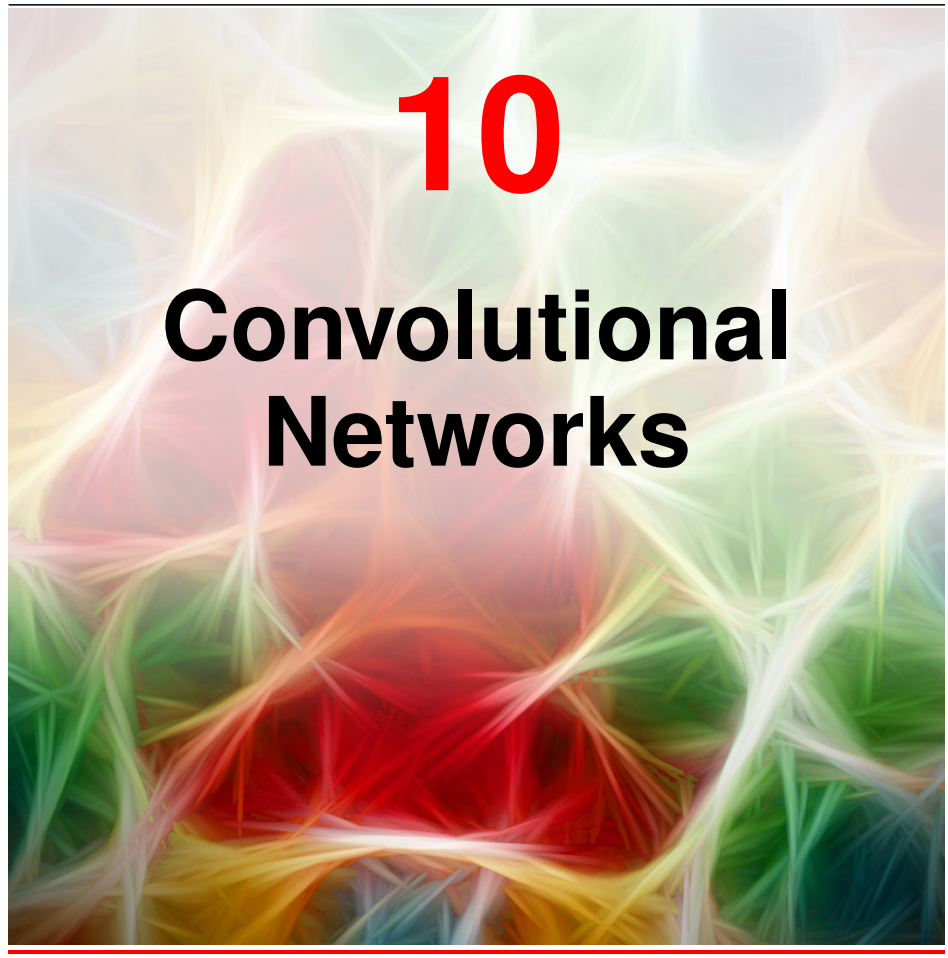
$$\mathbb{E}\left[E(\mathbf{W})\right] = \sum_{n=1}^{N}\sum_{k=1}^{K}\left\{y_{nk} - \rho\sum_{i=1}^{D} w_{ki}x_{ni}\right\}^2 \tag{9.72}$$

$$+ \rho(1-\rho)\sum_{n=1}^{N}\sum_{k=1}^{K}\sum_{i=1}^{D} w_{ki}^2 x_{ni}^2. \tag{9.73}$$

Thus, we see that the expected error function corresponds to a sum-of-squares error with a quadratic regularizer in which the regularization coefficient is scaled separately for each input variable according to the data values seen by that input. Finally, write down a closed-form solution for the weight matrix that minimizes this regularized error function.

Telegram: @uni_k

www.konkur.in



10 Convolutional Networks

The simplest machine learning models assume that the observed data values are unstructured, meaning that the elements of the data vectors $\mathbf{x} = (x_1, \ldots, x_D)$ are treated as if we do not know anything in advance about how the individual elements might relate to each other. If we were to make a random permutation of the ordering of these variables and apply this fixed permutation consistently on all training and test data, there would be no difference in the performance for the models considered so far.

Many applications of machine learning, however, involve *structured data* in which there are additional relationships between input variables. For example, the words in natural language form a *sequence*, and if we were to model language as a generative autoregressive process then we would expect each word to depend more strongly on the immediately preceding words and less so on words much earlier in the sequence. Likewise, the pixels of an image have a well-defined spatial relation-

Chapter 12

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024
C. M. Bishop, H. Bishop, *Deep Learning*, https://doi.org/10.1007/978-3-031-45468-4_10

Telegram: @uni_k

www.konkur.in

ship to each other in which the input variables are arranged in a two-dimensional grid, and nearby pixels have highly correlated values.

We have already seen that our knowledge of the structure of specific data modalities can be utilized through the addition of a regularization term to the error function in the training objective, through data augmentation, or through modifications to the model architecture. These approaches can help guide the model to respect certain properties such as invariance and equivariance with respect to transformations of the input data. In this chapter we will take a look at an architectural approach called a *convolutional neural network* (CNN), which we will see can be viewed as a sparsely connected multilayer network with parameter sharing, and designed to encode invariances and equivariances specific to image data.

Section 9.1

Section 9.1.4

10.1. Computer Vision

The automatic analysis and interpretation of image data form the focus of the field of computer vision and represent a major application area for machine learning (Szeliski, 2022). Historically, computer vision was based largely on 3-dimensional projective geometry. Hand-crafted features were constructed and used as input to simple learning algorithms (Hartley and Zisserman, 2004). However, it was one of the first fields to be transformed by the deep learning revolution, predominantly thanks to the CNN architecture. Although the architecture was originally developed in the context of image analysis, it has also been applied in other domains such as the analysis of sequential data. Recently alternative architectures based on transformers have become competitive with convolutional networks in some applications.

Chapter 12

There are many applications for machine learning in computer vision, of which some of the most commonly encountered are the following:

Figure 1.1

1. **Classification** of images, for example classifying an image of a skin lesion as benign or malignant. This is sometimes called 'image recognition'.

Figure 10.19

2. **Detection** of objects in an image and determining their locations within the image, for example detecting pedestrians from camera data collected by an autonomous vehicle.

Figure 10.26

3. **Segmentation** of images, in which each pixel is classified individually thereby dividing the image into regions sharing a common label. For example, a natural scene might be segmented into sky, grass, trees, and buildings, whereas a medical scan image could be segmented into cancerous tissue and normal tissue.

Figure 12.27

4. **Caption generation** in which a textual description is generated automatically from an image.

Figure 1.3

5. **Synthesis** of new images, for example generating images of human faces. Images can also be synthesized based on a text input describing the desired image content.

Telegram: @uni_k

www.konkur.in

Figure 20.9

6. **Inpainting** in which a region of an image is replaced with synthesized pixels that are consistent with the rest of the image. This is used, for example, to remove unwanted objects during image editing.

Figure 10.32

7. **Style transfer** in which an input image in one style, for example a photograph, is transformed into a corresponding image in a different style, for example an oil painting.

Figure 20.8

8. **Super-resolution** in which the resolution of an image is improved by increasing the number of pixels and synthesizing associated high-frequency information.

9. **Depth prediction** in which one or more views are used to predict the distance of the scene from the camera at each pixel in a target image.

10. **Scene reconstruction** in which one or more two-dimensional images of a scene are used to reconstruct a three-dimensional representation.

10.1.1 Image data

An image comprises a rectangular array of pixels, in which each pixel has either a grey-scale intensity or more commonly a triplet of red, green, and blue *channels* each with its own intensity value. These intensities are non-negative numbers that also have some maximum value corresponding to the limits of the camera or other hardware device used to capture the image. For the most part, we will view the intensities as continuous variables, but in practice they are represented with finite precision, for example as 8-bit numbers represented as integers in the range $0, \ldots, 255$. Some images, such as the magnetic resonance imaging (MRI) scans used in medical diagnosis, comprise three-dimensional grids of *voxels*. Similarly, videos comprise a sequence of two-dimensional images and therefore can also be viewed as three-dimensional structures in which successive frames are stacked through time.

Now consider the challenge of applying neural networks to image data to address some of the applications highlighted above. Images generally have a high dimensionality, with typical cameras capturing images comprising tens of megapixels. Treating the image data as unstructured may therefore require a model with a vast number of parameters that would be infeasible to train. More significantly, such an approach fails to take account of the highly structured nature of image data, in which the relative positions of different pixels play a crucial role. We can see this because if we take the pixels of an image and randomly permute them, then the result no longer looks like a natural image. Similarly, if we generate a synthetic image by drawing random values for the pixel intensities independently for each pixel, there

Figure 6.8

is essentially zero chance of generating something that looks like a natural image. Local correlations are important, and in a natural image there is a much higher probability that two nearby pixels will have similar colours and intensities compared to two pixels that are far apart. This represents powerful prior knowledge and can be used to encode strong inductive biases into a neural network, leading to models with far fewer parameters and with much better generalization accuracy.

Telegram: @uni_k

www.konkur.in

10.2. Convolutional Filters

One motivation for the introduction of convolutional networks is that for image data, which is the modality for which CNNs were designed, a standard fully connected architecture would require vast numbers of parameters due to the high-dimensional nature of images. To see this, consider a colour image with $10^3 \times 10^3$ pixels, each with three values corresponding to red, green, and blue intensities. If the first hidden layer of the network has, say, 1,000 hidden units, then we already have 3×10^9 weights in the first layer. Furthermore, such a network would have to learn any invariances and equivariances by example, which would require huge data sets. By designing an architecture that incorporates our inductive bias about the structure of images, we can reduce the data set requirements dramatically and also improve generalization with respect to symmetries in the image space.

To exploit the two-dimensional structure of image data to create inductive biases, we can use four interrelated concepts: hierarchy, locality, equivariance, and invariance. Consider the task of detecting faces in images. There is a natural hierarchical structure because one image may contain several faces, and each face includes elements such as eyes, and each eye has structure such as an iris, which itself has structure such as edges. At the lowest level of the hierarchy, a node in a neural network could detect the presence of a feature such as an edge using information that is *local* to a small region of an image, and therefore it would only need to see a small subset of the image pixels. More complex structures further up the hierarchy can be detected by composing multiple features found at previous levels. A key point, however, is that although we want to build the general concept of hierarchy into the model, we want the details of the hierarchy, including the type of features extracted at each level, to be learned from data, not hand-coded. Hierarchical models fit naturally within the deep learning framework, which already allows very complex concepts to be extracted from raw data through a succession of, possibly very many, 'layers' of processing, in which the whole system is trained end-to-end.

10.2.1 Feature detectors

For simplicity we will initially restrict our attention to grey-scale images (i.e., ones having a single channel). Consider a single unit in the first layer of a neural network that takes as input just the pixel values from a small rectangular region, or patch, from the image, as illustrated in Figure 10.1(a). This patch is referred to as the *receptive field* of that unit, and it captures the notion of locality. We would like weight values associated with this unit to learn to detect some useful low-level feature. The output of this unit is given by the usual functional form comprising a weighted linear combination of the input values, which is subsequently transformed using a nonlinear activation function:

$$z = \mathrm{ReLU}(\mathbf{w}^{\mathrm{T}}\mathbf{x} + w_0) \tag{10.1}$$

where $\mathbf{x}$ is a vector of pixel values for the receptive field, and we have assumed a ReLU activation function. Because there is one weight associated with each input

Telegram: @uni_k

www.konkur.in

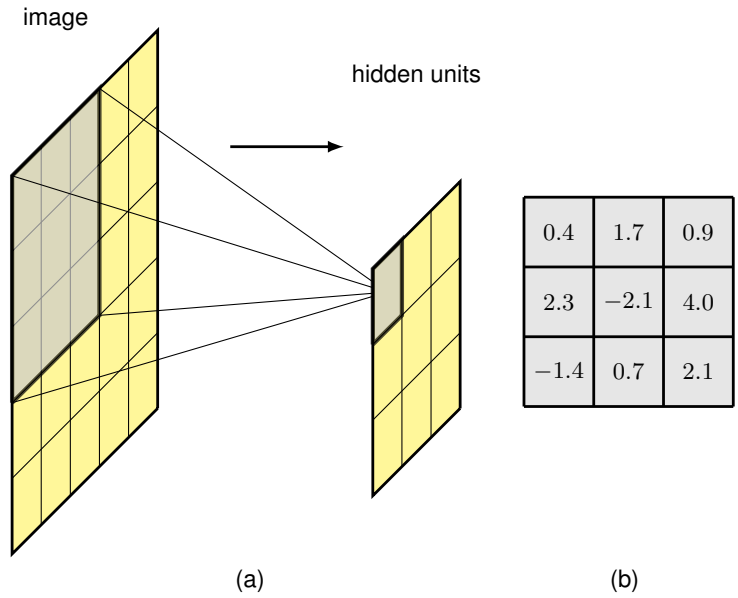


Figure 10.1 (a) Illustration of a receptive field, showing a unit in a hidden layer of a network that receives input from pixels in a 3×3 patch of the image. Pixels in this patch form the receptive field for this unit. (b) The weight values associated with this hidden unit can be visualized as a small 3×3 matrix, known as a kernel. There is also an additional bias parameter that is not shown here.

pixel, the weights themselves form a small two-dimensional grid known as a *filter*, sometimes also called a *kernel*, which itself can be visualized as an image. This is illustrated in Figure 10.1(b).

Suppose that $\mathbf{w}$ and w_0 in (10.1) are fixed and we ask for which value of the input image patch $\mathbf{x}$ will this hidden unit give the largest output response. To answer this we need to constrain $\mathbf{x}$ in some way, so let us suppose that its norm $\|\mathbf{x}\|^2$ is fixed. Then the solution for $\mathbf{x}$ that maximizes $\mathbf{w}^{\mathrm{T}}\mathbf{x}$, and hence maximizes the response of the hidden unit, is of the form $\mathbf{x} = \alpha\mathbf{w}$ for some coefficient α. This says that the maximum output response from this hidden unit occurs when it detects a patch of image that, up to an overall scaling, looks like the kernel image. Note that the ReLU generates a non-zero output only when $\mathbf{w}^{\mathrm{T}}\mathbf{x}$ exceeds a threshold of $-w_0$, and therefore the unit acts as a feature detector that signals when it finds a sufficiently good match to its kernel.

Exercise 10.1

10.2.2 Translation equivariance

Next note that if a small patch in a face image represents, for example, an eye at that location, then the same set of pixel values in a different part of the image must represent an eye at the new location. Our neural network needs to be able to generalize what it has learned in one location to all possible locations in the image, without needing to see examples in the training set of the corresponding feature at every possible location. To achieve this, we can simply replicate the *same* hidden-unit weight values at multiple locations across the image, as illustrated for a one-dimensional input space in Figure 10.2.

The units of the hidden layer form a *feature map* in which all the units share the same weights. Consequently if a local patch of an image produces a particular

Telegram: @uni_k

www.konkur.in

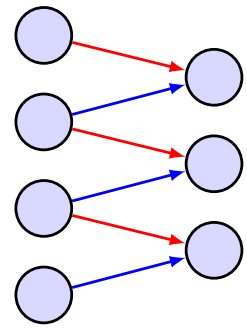


Figure 10.2 Illustration of convolution for a one-dimensional array of input values and a kernel of width 2. The connections are sparse and are shared by the hidden units, as shown by the red and blue arrows in which links with the same colour have the same weight values. This network therefore has six connections but only two independent learnable parameters.

response in the unit connected to that patch, then the same set of pixel values at a different location will produce the same response in the corresponding translated location in the feature map. This is an example of *equivariance*. We see that the connections in this network are *sparse* in that most connections are absent. Also, the values of the weights are *shared* by all the hidden units, as indicated by the colours of the connections. This transformation is an example of a *convolution*.

Section 9.1.3

We can extend the idea of convolution to two-dimensional images as follows (Dumoulin and Visin, 2016). For an image $\mathbf{I}$ with pixel intensities $I(j,k)$, and a filter $\mathbf{K}$ with pixel values $K(l,m)$, the feature map $\mathbf{C}$ has activation values given by

$$C(j,k) = \sum_l \sum_m I(j+l, k+m) K(l,m) \tag{10.2}$$

where we have omitted the nonlinear activation function for clarity. This again is an example of a *convolution* and is sometimes expressed as $\mathbf{C} = \mathbf{I} * \mathbf{K}$. Note that strictly speaking (10.2) is called a *cross-correlation*, which differs slightly from the conventional mathematical definition of convolution, but here we will follow common practice in the machine learning literature and refer to (10.2) as a convolution. The relationship (10.2) is illustrated in Figure 10.3 for a 3×3 image and a 2×2 filter. Importantly, when using batch normalization in a convolutional network, the same value of mean and variance must be used at every spatial location within a feature map when normalizing the states of the units to ensure that the statistics of the feature map are independent of location.

Exercise 10.4

Section 7.4.2

As an example of the application of convolution, we consider the problem of detecting edges in images using a fixed, hand-crafted convolutional filter. Intuitively, we can think of a vertical edge as occurring when there is a significant local change in the intensity between pixels as we move horizontally across the image. We can measure this by convolving the image with a 3×3 filter of the form

$$\begin{array}{|c|c|c|} \hline -1 & 0 & 1 \\ \hline -1 & 0 & 1 \\ \hline -1 & 0 & 1 \\ \hline \end{array} \tag{10.3}$$

Telegram: @uni_k

www.konkur.in

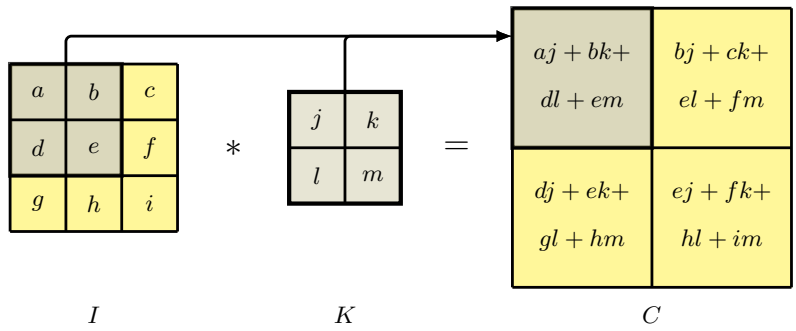


Figure 10.3 Example of a 3×3 image convolved with a 2×2 filter to give a resulting 2×2 feature map.

Similarly we can detect horizontal edges by convolving with the transpose of this filter:

$$\begin{array}{|c|c|c|}\hline -1 & -1 & -1 \\ \hline 0 & 0 & 0 \\ \hline 1 & 1 & 1 \\ \hline \end{array} \tag{10.4}$$

Figure 10.4 shows the results of applying these two convolutional filters to a sample image. Note that in Figure 10.4(b) if a vertical edge corresponds to an increase in pixel intensity, the corresponding point on the feature map is positive (indicated by a light colour), whereas if the vertical edge corresponds to a decrease in pixel intensity, the corresponding point on the feature map is negative (indicated by a dark colour), with analogous properties for Figure 10.4(c) for horizontal edges.

Comparing this convolutional structure with a standard fully connected network, we see several advantages: (i) the connections are sparse, leading to far fewer weights even with large images, (ii) the weight values are shared, greatly reducing the number of independent parameters and consequently reducing the required size



(a) (b) (c)

Figure 10.4 Illustration of edge detection using convolutional filters showing (a) the original image, (b) the result of convolving with the filter (10.3) that detects vertical edges, and (c) the result of convolving with the filter (10.4) that detects horizontal edges.

Telegram: @uni_k

www.konkur.in

of the training set needed to learn those parameters, and (iii) the same network can be applied to images of different sizes without the need for retraining. We will return to this final point later, but for the moment, simply note that changing the size of the input image simply changes the size of the feature map but does not change the number of weights, or the number of independent learnable parameters, in the model. One final observation regarding convolutional networks is that, by exploiting the massive parallelism of graphics processing units (GPUs) to achieve high computational throughput, convolutions can be implemented very efficiently.

Section 10.4.3

10.2.3 Padding

We see from Figure 10.3 that the convolution map is smaller than the original image. If the image has dimensionality $J \times K$ pixels and we convolve with a kernel of dimensionality $M \times M$ (filters are typically chosen to be square) the resulting feature map has dimensionality $(J - M + 1) \times (K - M + 1)$. In some cases we want the feature map to have the same dimensions as the original image. This can be achieved by *padding* the original image with additional pixels around the outside, as illustrated in Figure 10.5. If we pad with P pixels then the output map has dimensionality $(J+2P-M+1)\times(K+2P-M+1)$. If there is no padding, so that $P = 0$, this is called a *valid* convolution. When the value of P is chosen such that the output array has the same size as the input, corresponding to $P = (M - 1)/2$, this is called a *same* convolution, because the image and the feature map have the same dimensions. In computer vision, filters generally use odd values of M, so that the padding can be symmetric on all sides of the image and that there is a well-defined central pixel associated with the location of the filter. Finally, we have to choose a suitable value for the intensities associated with the padding pixels. A typical choice is to set the padding values to zero, after first subtracting the mean from each image so that zero represents the average value of the pixel intensity. Padding can also be applied to feature maps for processing by subsequent convolutional layers.

Exercise 10.6

10.2.4 Strided convolutions

In typical image processing applications, the images can have very large numbers of pixels, and since the kernels are often relatively small, so that $M \ll J, K$, the convolutional feature map will be of a similar size to the original image and will be the same size if *same* padding is used. Sometimes we wish to use feature maps that are significantly smaller than the original image to provide flexibility in the design of convolutional network architectures. One way to achieve this is to use *strided convolutions* in which, instead of stepping the filter over the image one pixel at a time, it is moved in larger steps of size S, called the *stride*. If we use the same stride horizontally and vertically, then the number of elements in the feature map will be

Exercise 10.7

$$\left\lfloor \frac{J + 2P - M}{S} - 1 \right\rfloor \times \left\lfloor \frac{K + 2P - M}{S} - 1 \right\rfloor \tag{10.5}$$

where $\lfloor x \rfloor$ denotes the 'floor' of x, i.e., the largest integer that is less than or equal to x. For large images and small filter sizes, the image map will be roughly a factor of $1/S$ smaller than the original image.

Telegram: @uni_k

www.konkur.in

Figure 10.5 Illustration of a 4×4 image that has been padded with additional pixels to create a 6×6 image.

0	0	0	0	0	0
0	X_{11}	X_{12}	X_{13}	X_{14}	0
0	X_{21}	X_{22}	X_{23}	X_{24}	0
0	X_{31}	X_{32}	X_{33}	X_{34}	0
0	X_{41}	X_{42}	X_{43}	X_{44}	0
0	0	0	0	0	0

10.2.5 Multi-dimensional convolutions

So far we have considered convolutions over a single grey-scale image. For a colour image there will be three channels corresponding to the red, green, and blue colours. We can easily extend convolutions to cover multiple channels by extending the dimensionality of the filter. An image with $J \times K$ pixels and C channels will be described by a *tensor* of dimensionality $J \times K \times C$. We can introduce a filter described by a tensor of dimensionality $M \times M \times C$ comprising a separate $M \times M$ filter for each of the C channels. Assuming no padding and a stride of 1, this again gives a feature map of size $(J-M+1) \times (K-M+1)$, as is illustrated in Figure 10.6.

Section 6.3.7

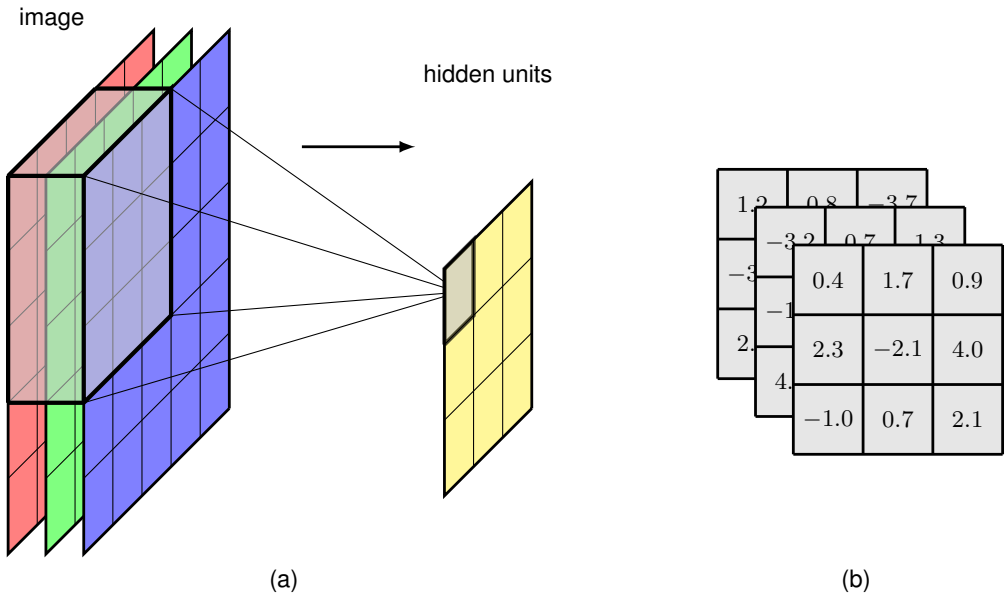


Figure 10.6 (a) Illustration of a multi-dimensional filter that takes input from across the R, G, and B channels. (b) The kernel here has 27 weights (plus a bias parameter not shown) and can be visualized as a $3 \times 3 \times 3$ tensor.

Telegram: @uni_k

www.konkur.in

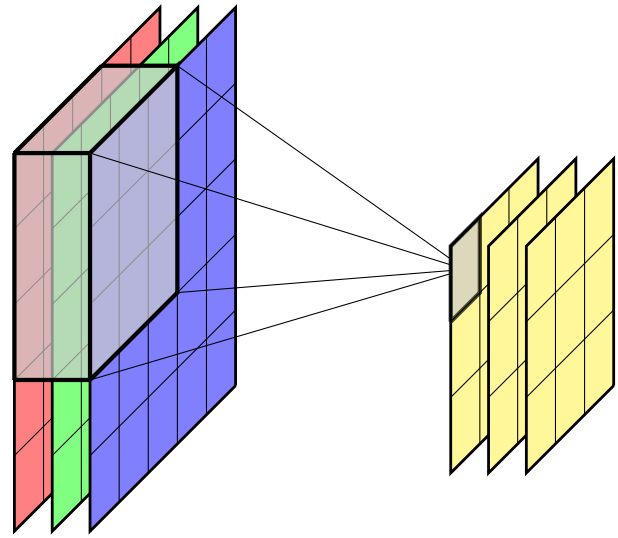


Figure 10.7 The multi-dimensional convolutional filter layer shown in Figure 10.6 can be extended to include multiple independent filter channels.

We now make a further important extension to convolutions. Up to now we have created a single feature map in which all the points in the feature map share the same set of learnable parameters. For a filter of dimensionality $M \times M \times C$, this will have M^2C weight parameters, irrespective of the size of the image. In addition there will be a bias parameter associated with this unit. Such a filter is analogous to a single hidden node in a fully connected network, and it can learn to detect only one kind of feature and is therefore very limited. To build more flexible models, we simply include multiple such filters, in which each filter has its own independent set of parameters giving rise to its own independent feature map, as illustrated in Figure 10.7. We will again refer to these separate feature maps as *channels*. The filter tensor now has dimensionality $M \times M \times C \times C_{\text{OUT}}$ where C is the number of input channels and C_{OUT} is the number of output channels. Each output channel will have its own associated bias parameter, so the total number of parameters will be $(M^2C + 1)C_{\text{OUT}}$.

A useful concept in designing convolutional networks is the 1×1 *convolution* (Lin, Chen, and Yan, 2013), which is simply a convolutional layer in which the filter size is a single pixel. The filters have C weights, one for each input channel, plus a bias. One application for 1×1 convolutions is simply to change the number of channels (typically to reduce the number of channels) without changing the size of the feature maps, by setting the number of output channels to be different to the number of input channels. It is therefore complementary to strided convolutions or pooling in that it reduces the number of channels rather than the dimensionality of the channels.

10.2.6 Pooling

A convolutional layer encodes translation *equivariance*, whereby if a small patch of pixels, representing the receptive field of a hidden unit, is moved to a different

Telegram: @uni_k

www.konkur.in

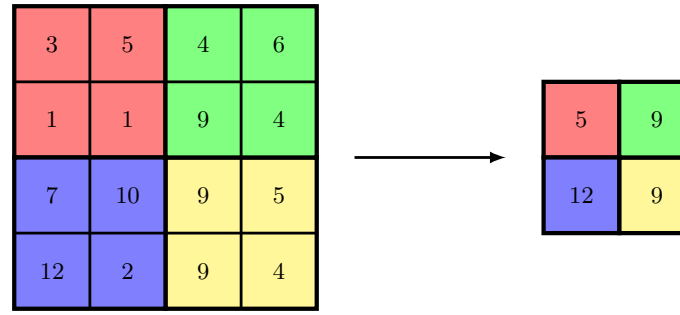


Figure 10.8 Illustration of max-pooling in which blocks of 2×2 pixels in a feature map are combined using the 'max' operator to generate a new feature map of smaller dimensionality.

location in the image, the associated outputs of the feature map will move to the corresponding location in the feature map. This is valuable for applications such as finding the location of an object within an image. For other applications, such as classifying an image, we want the output to be *invariant* to translations of the input. In all cases, however, we want the network to be able to learn hierarchical structure in which complex features at a particular level are built up from simpler features at the previous level. In many cases the spatial relationship between those simpler features will be important. For example, it is the relative positions of the eyes, nose, and mouth that help determine the presence of a face and not just the presence of these features in arbitrary locations within the image. However, small changes in the relative locations do not affect the classification, and we want to be invariant to such small translations of individual features. This can be achieved using *pooling* applied to the output of the convolutional layer.

Pooling has similarities to using a convolutional layer in that an array of units is arranged in a grid, with each unit taking input from a receptive field in the previous feature map layer. Again, there is a choice of filter size and of stride length. The difference, however, is that the output of a pooling unit is a simple, fixed function of its inputs, and so there are no learnable parameters in pooling. A common example of a pooling function is *max-pooling* (Zhou and Chellappa, 1988) in which each unit simply outputs the $\max$ function applied to the input values. This is illustrated with a simple example in Figure 10.8. Here the stride length is equal to the filter width, and so there is no overlap of the receptive fields.

As well as building in some local translation invariance, pooling can also be used to reduce the dimensionality of the representation by down-sampling the feature map. Note that using strides greater than 1 in a convolutional layer also has the effect of down-sampling the feature maps.

We can interpret the activation of a unit in a feature map as a measure of the strength of detection of a corresponding feature, so that the max-pooling preserves information on whether the feature is present and with what strength but discards some positional information. There are many other choices of pooling function, for example *average pooling* in which the pooling function computes the average of the values from the corresponding receptive field in the feature map. These all introduce some degree of local translation invariance.

Pooling is usually applied to each channel of a feature map independently. For

Telegram: @uni_k

www.konkur.in

example, if we have a feature map with 8 channels, each of dimensionality 64×64, and we apply max-pooling with a receptive field of size 2×2 and a stride of 2, the output of the pooling operation will be a tensor of dimensionality $32 \times 32 \times 8$.

We can also apply pooling across multiple channels of a feature map, which gives the network the potential to *learn* other invariances beyond simple translation invariance. For example, if several channels in a convolutional layer learn to detect the same feature but at different orientations, then max-pooling across those feature maps will be approximately invariant to rotations.

Pooling also allows a convolutional network to process images of varying sizes. Ultimately, the output, and generally some of the intermediate layers, of a convolutional network must have a fixed size. Variable-sized inputs can be accommodated by varying the stride length of the pooling according to the size of the image such that the number of pooled outputs remains constant.

10.2.7 Multilayer convolutions

The convolutional network structure described so far is analogous to a single layer in a standard fully connected neural network. To allow the network to discover and represent hierarchical structure in the data, we now extend the architecture by considering multiple layers of the kind described above. Each convolutional layer is described by a filter tensor of dimensionality $M \times M \times C_{\text{IN}} \times C_{\text{OUT}}$ in which the number of independent weight and bias parameters is $(M^2 C_{\text{IN}} + 1)C_{\text{OUT}}$. Each such convolutional layer can optionally be followed by a pooling layer. We can now apply multiple such layers of convolution and pooling in succession, in which the C_{OUT} output channels of a particular layer, analogous to the RGB channels of the input image, form the input channels of the next layer. Note that the number of channels in a feature map is sometimes called the 'depth' of the feature map, but we prefer to reserve the term depth to mean the number of layers in a multilayer network.

A key property that we built into the convolutional framework is that of locality, in which a given unit in a feature map takes information only from a small patch, the *receptive field*, in the previous layer. When we construct a deep neural network in which each layer is convolutional then the effective receptive field of a unit in later layers in the network becomes much larger than those in earlier layers, as seen in Figure 10.9.

In many applications, the output units of the network need to make predictions about the image as a whole, for example in a classification task, and so they need to combine information from across the whole of the input image. This is typically achieved by introducing one or two standard fully connected layers as the final stages of the network, in which each unit is connected to every unit in the previous layer. The number of parameters in such an architecture can be manageable because the final convolutional layer generally has much lower dimensionality than the input layer due to the intermediate pooling layers. Nevertheless, the final fully connected layers may contain the majority of the independent degrees of freedom in the network even if the number of (shared) connections in the network is larger in the convolutional layers.

A complete CNN therefore comprises multiple layers of convolutions inter-

Telegram: @uni_k

www.konkur.in

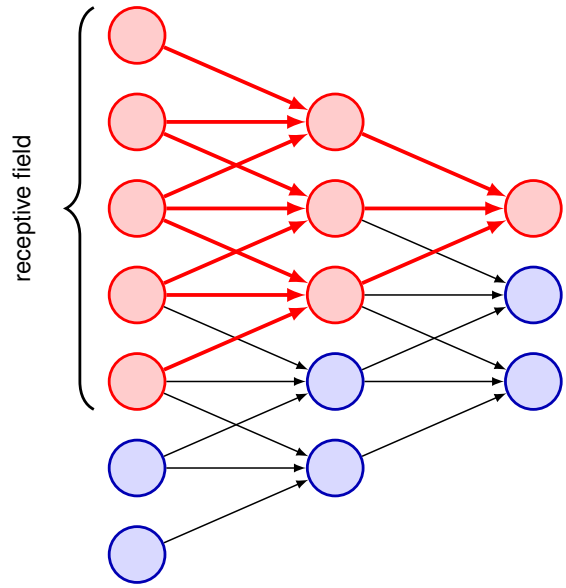


Figure 10.9 Illustration of how the effective receptive field grows with depth in a multilayer convolutional network. Here we see that the red unit at the top of the output layer takes inputs from a receptive field in the middle layer of units, each of which has a receptive field in the first layer of units. Thus, the activation of the red unit in the output layer depends on the outputs of 3 units in the middle layer and 5 units in the input layer.

spersed with pooling operations, and often with conventional fully connected layers in the final stages of the network. There are many choices to be made in designing such an architecture including the number of layers, the number of channels in each layer, the filter sizes, the stride widths, and multiple other such hyperparameters. A wide variety of different architectures have been explored, although in practice it is difficult to make a systematic comparison of hyperparameter values using hold-out data due to the high computational cost of training each candidate configuration.

10.2.8 Example network architectures

Convolutional networks were the first deep neural networks (i.e., ones with more than two learnable layers of parameters) to be successfully deployed in applications. An early example was *LeNet*, which was used to classify low-resolution monochrome images of handwritten digits (LeCun *et al.*, 1989; LeCun *et al.*, 1998). The development of more powerful convolutional networks was accelerated through the introduction of a large-scale benchmark data set called *ImageNet* (Deng *et al.*, 2009) comprising some 14 million natural images each of which has been hand labelled into one of nearly 22,000 categories. This was a much larger data set than had been used previously, and the advances in the field driven by ImageNet served to emphasize the importance of large-scale data, alongside well-designed models having appropriate inductive biases, in building successful deep learning solutions.

A subset of images comprising 1,000 non-overlapping categories formed the basis for the annual *ImageNet Large Scale Visual Recognition Challenge*. Again, this was a much larger number of categories than the typically few dozen classes previously considered. Having so many categories made the problem much more challenging because, if the classes were distributed uniformly, random guessing would

Telegram: @uni_k

www.konkur.in

have an error rate of 99.9%. The data set has just over 1.28 million training images, 50,000 validation images, and 100,000 test images. The classifiers are designed to produce a ranked list of predicted output classes on test images, and results are reported in terms of top-1 and top-5 error rates, meaning an image is deemed to be correctly classified if the true class appears at the top of the list or if it is in one of the five highest-ranked class predictions. Early results with this data set achieved a top-5 error rate of around 25.5%. An important advance was made by the *AlexNet* convolutional network architecture (Krizhevsky, Sutskever, and Hinton, 2012), which won the 2012 competition and reduced the top-5 error rate to a new record of 15.3%. Key aspects of this model were the use of the ReLU activation function, the application of GPUs to train the network, and the use of dropout regularization. Subsequent years saw further advances, leading to error rates of around 3%, which is somewhat better than human-level performance for the same data, which is around 5% (Dodge and Karam, 2017). This can be attributed to the difficulty humans have in distinguishing subtly different classes (for example multiple varieties of mushrooms).

Section 9.6.1

As an example of a typical convolutional network architecture, we look in detail at the VGG-16 model (Simonyan and Zisserman, 2014), where VGG stands for the Visual Geometry Group, who developed the model, and 16 refers to the number of learnable layers in the model. VGG-16 has some simple design principles leading to a relatively uniform architecture, shown in Figure 10.10, that minimizes the number of hyperparameter choices that need to be made. It takes an input image having 224×224 pixels and three colour channels, followed by sets of convolutional layers interspersed with down-sampling. All convolutional layers have filters of size 3×3 with a stride of 1, same padding, and a ReLU activation function, whereas the max-pooling operations all use stride 2 and filter size 2×2 thereby down-sampling the number of units by a factor of 4. The first learnable layer is a convolutional layer in which each unit takes input from a $3 \times 3 \times 3$ 'cube' from the stack of input channels and so has 28 parameters including the bias. These parameters are shared across all units in the feature map for that channel. There are 64 such feature channels in the first layer, giving an output tensor of size $224 \times 224 \times 64$. The second layer is also convolutional and again has 64 channels. This is followed by max-pooling giving feature maps of size 112×112. Layers 3 and 4 are again convolutional, of dimensionality 112×112, and each was chosen to have 128 channels. This increase in the number of channels offsets to some extent the down-sampling in the max-pooling layer to ensure that the number of variables in the representation at each layer does not decrease too rapidly through the network. Again, this is followed by a max-pooling operation to give a feature map size of 56×56. Next come three more convolutional layers each with 256 channels, thereby again doubling the number of channels in association with the down-sampling. This is followed by another max-pooling to give feature maps of size 28×28 followed by three more convolutional layers each having 512 channels, followed by another max-pooling, which down-samples to feature maps of size 14×14. This is followed by three more convolutional layers, although the number of feature maps in these layers is kept at 512, followed by another max-pooling, which brings the size of the feature maps down to 7×7. Finally there are three more layers that are *fully connected* meaning that they are

Telegram: @uni_k

www.konkur.in

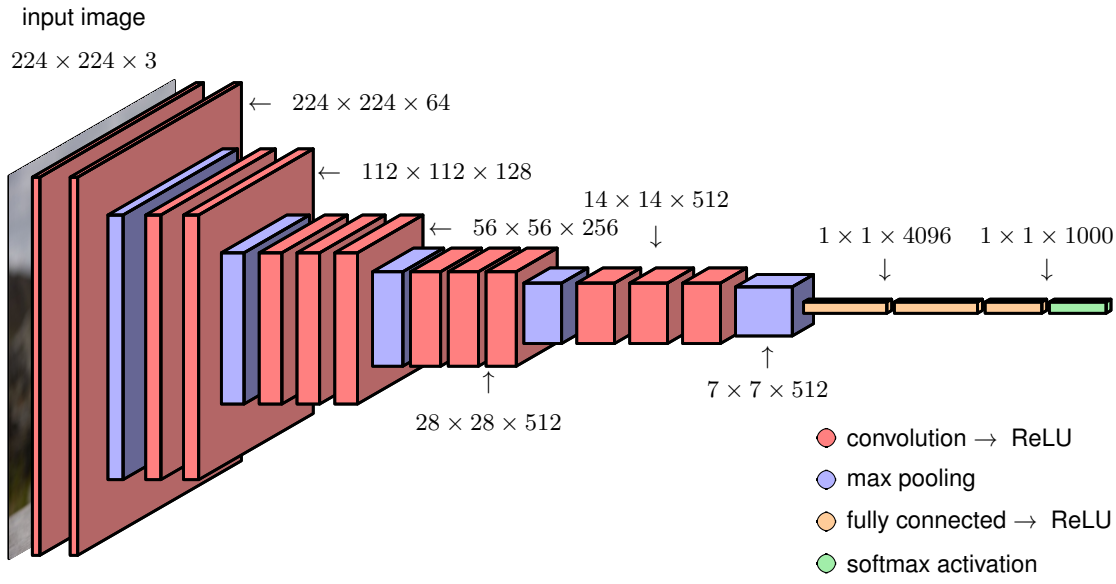


Figure 10.10 The architecture of a typical convolutional network, in this case a model called VGG-16.

standard neural network layers with full connectivity and no sharing of parameters. The final max-pooling layer has 512 channels each of size 7×7 giving 25,088 units in total. The first fully connected layer has 4,096 units, each of which is connected to each of the max-pooling units. This is followed by a second fully connected layer again with 4,096 units, and finally there is a third fully connected layer with 1,000 units so that the network can be applied to a classification problem involving 1,000 classes. All the learnable layers in the network have nonlinear ReLU activation functions, except for the output layer, which has a softmax activation function. In total there are roughly 138 million independently learnable parameters in VGG-16, the majority of which (nearly 103 million) are in the first fully connected layer, whereas most of the connections are in the first convolutional layer.

Exercise 10.8

Earlier CNNs typically had fewer convolutional layers, as they had larger receptive fields. For example, Alexnet (Krizhevsky, Sutskever, and Hinton, 2012) has 11×11 receptive fields with a stride of 4. We saw in Figure 10.9 that larger receptive fields can also be achieved implicitly by using multiple layers each having smaller receptive fields. The advantage of the latter approach is that it requires significantly fewer parameters, effectively imposing an inductive bias on the larger filters as they must be composed of convolutional sub-filters. Although this is a highly complex architecture, only the network function itself needs to be coded explicitly since the derivatives of the cost function can be evaluated using automatic differentiation and the cost function optimized using stochastic gradient descent.

Section 8.2

Telegram: @uni_k

www.konkur.in

10.3. Visualizing Trained CNNs

We turn now to an exploration of the features learned by modern deep CNNs, and we will see some remarkable similarities to the properties of the mammalian visual cortex.

10.3.1 Visual cortex

Historically, much of the motivation for CNNs came from pioneering research in neuroscience, which gave insights into the nature of visual processing in mammals including humans. Electrical signals from the retina are transformed through a series of processing layers in the visual cortex, which is at the back of the brain, where the neurons are organized into two-dimensional sheets each of which forms a map of the two-dimensional visual field. In their pioneering work, Hubel and Wiesel (1959) measured the electrical responses of individual neurons in the visual cortex of cats while presenting visual stimuli to the cats' eyes. They discovered that some neurons, called 'simple cells', have a strong response to visual inputs with a simple edge oriented at a particular angle and located at a particular position within the visual field, whereas other stimuli generated relatively little response in those neurons. More detailed studies showed that the response of these simple cells can be modelled using *Gabor filters*, which are two-dimensional functions defined by

$$G(x, y) = A \exp\left(-\alpha \widetilde{x}^2 - \beta \widetilde{y}^2\right) \sin\left(\omega \widetilde{x} + \phi\right) \tag{10.6}$$

where

$$\widetilde{x} = (x - x_0)\cos(\theta) + (y - y_0)\sin(\theta) \tag{10.7}$$
$$\widetilde{y} = -(x - x_0)\sin(\theta) + (y - y_0)\cos(\theta). \tag{10.8}$$

Equations (10.7) and (10.8) represent a rotation of the coordinate system through an angle θ and therefore the $\sin(\cdot)$ term in (10.6) represents a sinusoidal spatial oscillation oriented in a direction defined by the polar angle θ, with frequency ω and phase angle ϕ . The exponential factor in (10.6) creates a decay envelope that localizes the filter in the neighbourhood of position (x_0, y_0) and with decay rates governed by α and β. Example Gabor filters are shown in Figure 10.11.

Hubel and Wiesel also discovered the presence of 'complex cells', which respond to more complex stimuli and which seem to be derived by combining and processing the output of simple cells. These responses exhibit some degree of invariance to small changes in the input such as shifts in location, analogous to the pooling units in a convolutional deep network. Deeper levels of the mammalian visual processing system have even more specific responses and even greater invariance to transformations of the visual input. Such cells have been termed 'grandmother cells' because such a cell could notionally respond if, and only if, the visual input corresponds to a person's grandmother, irrespective of location, scale, lighting, or other transformations of the scene. This work directly inspired an early form of deep neural network called the *neocognitron* (Fukushima, 1980), which was the forerunner of

Telegram: @uni_k

www.konkur.in

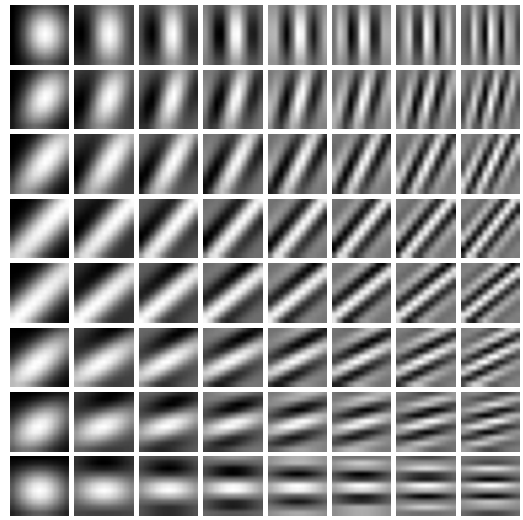


Figure 10.11 Examples of Gabor filters defined by (10.6). The orientation angle θ varies from 0 in the top row to $\pi/2$ in the bottom row, whereas the frequency varies from $\omega = 1$ in the left column to $\omega = 10$ in the right column.

convolutional neural networks. The neocognitron had multiple layers of processing comprising local receptive fields with shared weights followed by local averaging or max-pooling to confer positional invariance. However, it lacked an end-to-end training procedure since it predated the development of backpropagation, relying instead on greedy layer-wise learning through an unsupervised clustering algorithm.

10.3.2 Visualizing trained filters

Suppose we have a trained deep CNN and we wish to explore what the hidden units have learned to detect. For the filters in the first convolutional layer this is relatively straightforward, as they correspond to small patches in the original input image space, and so we can visualize the network weights associated with these filters directly as small images. The first convolutional layer computes inner products between the filters and the corresponding image patches, and so the unit will have a large activation when the inner product has a large magnitude.

Figure 10.12 shows some example filters from the first layer of a CNN trained on the ImageNet data set. We see a remarkable similarity between these filters and the Gabor filters of Figure 10.11. However, this does not imply that a convolutional neural network is a good model of how the brain works, because very similar results can be obtained from a wide variety of statistical methods (Hyvärinen, Hurri, and Hoyer, 2009). This is because these characteristic filters are a general property of the statistics of natural images and therefore prove useful for image understanding in both natural and artificial systems.

Although we can visualize the filters in the first layer directly, the subsequent layers in the network are harder to interpret because their inputs are not patches of images but groups of filter responses. One approach, analogous to that used by Hubel and Wiesel, is to present a large number of image patches to the network and

Telegram: @uni_k

www.konkur.in

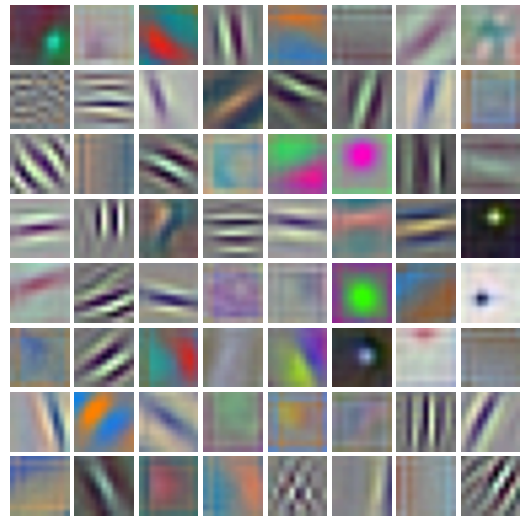


Figure 10.12 Examples of learned filters from the first layer of AlexNet. Note the remarkable similarity of many of the learned filters to the Gabor filters in Figure 10.11, which correspond to features detected by living neurons in the visual cortex of mammals.

see which produce the highest activation value in any particular hidden unit. Figure 10.13 shows examples obtained using a network with five convolutional layers, followed by two fully connected layers, trained on 1.3 million ImageNet data points spanning 1,000 classes. We see a natural hierarchical structure, with the first layer responding to edges, the second layer responding to textures and simple shapes, layer 3 showing components of objects (such as wheels), and layer 5 showing entire objects.

We can extend this technique to go beyond simply selecting image patches from the validation set and instead perform a numerical optimization over the input variables to maximize the activation of a particular unit (Zeiler and Fergus, 2013; Simonyan, Vedaldi, and Zisserman, 2013; Yosinski *et al.*, 2015). If we chose the unit to be one of the outputs then we can look for an image that is most representative of the corresponding class label. Because the output units generally have a softmax activation function, it is better to maximise the pre-activation value that feeds into the softmax rather than the class probability directly, as this ensures the optimization depends on only one class. For example, if we seek the image that produces the strongest response to the class 'dog', then if we optimize the softmax output it could drive the image to be, say, less like a cat because of the denominator in the softmax. This approach is related to adversarial training. Unconstrained optimization of the output-unit activation, however, leads to individual pixel values being driven to infinity and also creates high-frequency structure that is difficult to interpret, and so some form of regularization is required to find solutions that are closer to natural images. Yosinski *et al.* (2015) used a regularization function comprising the sum of squares of the pixel values along with a procedure that alternates gradient-based updates to the image pixel values with a blurring operation to remove high-frequency structure and a clipping operation that sets to zero those pixel values that make only small contributions to the class label. Example results are shown in Figure 10.14.

Chapter 17

Telegram: @uni_k

www.konkur.in

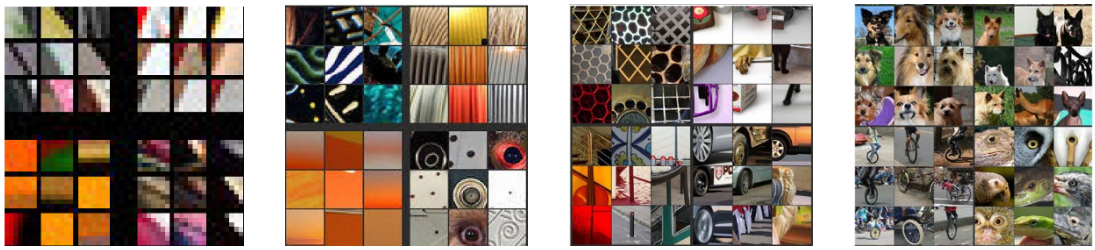


Figure 10.13 Examples of image patches (taken from a validation set) that produce the strongest activation in the hidden units in a network having five convolutional layers trained on ImageNet data. The top nine activations in each feature map are arranged as a 3×3 grid for four randomly chosen channels in each of the corresponding layers. We see a steady progression in complexity with depth, from simple edges in layer 1 to complete objects in layer 5. [From Zeiler and Fergus (2013) with permission.]

10.3.3 Saliency maps

Another way to gain insight into the features used by a convolutional network is through *saliency maps*, which aim to identify those regions of an image that are most significant in determining the class label. This is best done by investigating the final convolutional layer because this still retains spatial localization, which becomes lost in the subsequent fully connected layers, and yet it has the highest level of semantic representation. The Grad-CAM (gradient class activation mapping) method (Selvaraju *et al.*, 2016) first computes, for a given input image, the derivatives of the output-unit pre-activation $a^{(c)}$ for a given class c, before the softmax, with respect to the pre-activations $a_{ij}^{(k)}$ of all the units in the final convolutional layer for channel k. For each channel in that layer, the average of those derivatives is evaluated to give

$$\alpha_k = \frac{1}{M_k} \sum_i \sum_j \frac{\partial a^{(c)}}{\partial a_{ij}^{(k)}} \tag{10.9}$$

where i and j index the rows and columns of channel k, and M_k is the total number of units in that channel. These averages are then used to form a weighted sum of the form:

$$\mathbf{L} = \sum_k \alpha_k \mathbf{A}^{(k)} \tag{10.10}$$

in which $\mathbf{A}^{(k)}$ is a matrix with elements $a_{ij}^{(k)}$. The resulting array has the same dimensionality as the final convolutional layer, for example 14×14 for the VGG network shown in Figure 10.10, and can be superimposed on the original image in the form of a 'heat map' as seen in Figure 10.15.

Telegram: @uni_k

www.konkur.in

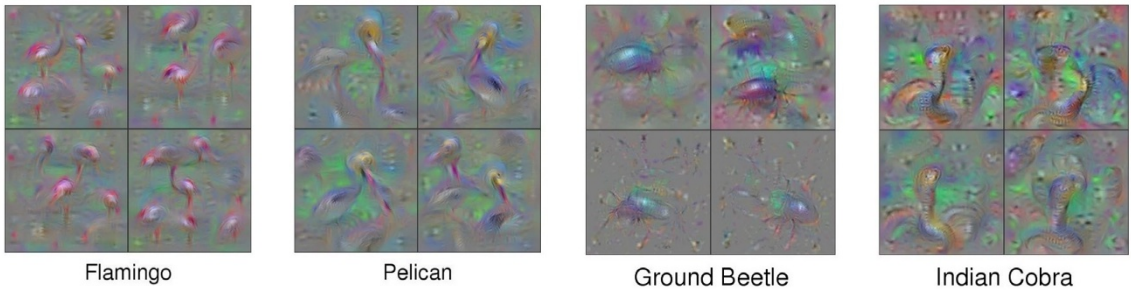


Figure 10.14 Examples of synthetic images generated by maximizing the class probability with respect to the image pixel channel values for a trained convolutional classifier. Four different solutions, obtained with different settings of the regularization parameters, are shown for each of four object classes. [From Yosinski *et al.* (2015) with permission.]

10.3.4 Adversarial attacks

Gradients with respect to changes in the input image pixel values can also be used to create *adversarial attacks* against convolutional networks (Szegedy *et al.*, 2013). These attacks involve making very small modifications to an image, at a level that is imperceptible to a human, which cause the image to be misclassified by the neural network. One simple approach to creating adversarial images is called the *fast gradient sign* method (Goodfellow, Shlens, and Szegedy, 2014). This involves changing each pixel value in an image $\mathbf{x}$ by a fixed amount ϵ with a sign determined by the gradient of an error function $E(\mathbf{x}, t)$ with respect to the pixel values. This gives a modified image defined by

$$\mathbf{x}' = \mathbf{x} + \epsilon \, \text{sign}(\nabla_{\mathbf{x}} E(\mathbf{x}, t)). \tag{10.11}$$

Here t is the true label of $\mathbf{x}$, and the error $E(\mathbf{x}, t)$ could, for example, be the negative log likelihood of $\mathbf{x}$. The required gradient can be computed efficiently using backpropagation. During conventional training of a neural network, the network parameters are adjusted to minimize this error, whereas the modification defined by (10.11) alters the image (while keeping the trained network parameters fixed) so as

Chapter 8

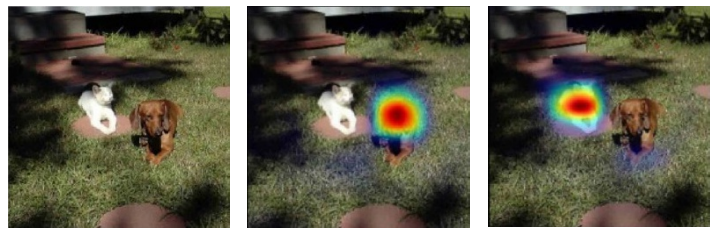


Figure 10.15 Saliency maps for the VGG-16 network with respect to the 'dog' and 'cat' categories. [From Selvaraju *et al.* (2016) with permission.]

Telegram: @uni_k

www.konkur.in

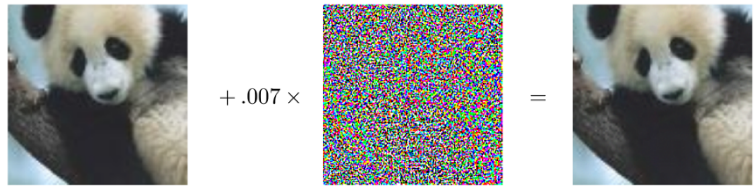


Figure 10.16 Example of an adversarial attack against a trained convolutional network. The image on the left is classified as a panda with confidence 57.7%. The addition of a small level of a random-looking perturbation (that itself is classified as a nematode with confidence 8.2%) results in the image on the right, which is classified as a gibbon with confidence 99.3%. [From Goodfellow, Shlens, and Szegedy (2014) with permission.]

to increase the error. By keeping ϵ small, we ensure that the changes to the image are undetectable to the human eye. Remarkably, this can give images that are misclassified by the network with high confidence, as seen in the example in Figure 10.16.

The ability to fool neural networks in this way raises potential security concerns as it creates opportunities for attacking trained classifiers. It might appear that this issue arises from over-fitting, in which a high-capacity model has adapted precisely to the specific image such that small changes in the input produce large changes in the predicted class probabilities. However, it turns out that an image that has been adapted to give a spurious output for a particular trained network can give similarly spurious outputs when fed to other networks (Goodfellow, Shlens, and Szegedy, 2014). Moreover, a similar adversarial result can be obtained with much less flexible linear models. It is even possible to create physical artefacts such that a regular, uncorrupted image of the artefact will give erroneous predictions when presented to a trained neural network, as seen in Figure 10.17. Although these basic kinds of adversarial attacks can be addressed by simple modifications to the network training process, more sophisticated approaches are harder to defeat. Understanding the implications of these results and mitigating their potential pitfalls remain open areas of research.



Figure 10.17 Two examples of physical stop signs that have been modified. Images of these objects are robustly classified as 45 mph speed-limit signs by CNNs. [From Eykholt *et al.* (2018) with permission.]

Telegram: @uni_k

www.konkur.in

10.3.5 Synthetic images

As a final example of image modification that provides additional insights into the operation of a trained convolutional network, we consider a technique called *DeepDream* (Mordvintsev, Olah, and Tyka, 2015). The goal is to generate a synthetic image with exaggerated characteristics. We do this by determining which nodes in a particular hidden layer of the network respond strongly to a particular image and then modifying the image to amplify those responses. For example, if we present an image of some clouds to a network trained on object recognition and a particular node detects a cat-like pattern at a particular region of the image, then we modify the image to be more 'cat like' in that region. To do this, we apply an image to the input of the network and forward propagate through to some particular layer. We then set the backpropagation δ variables for that layer equal to the pre-activations of the nodes and run backpropagation to the input layer to get a gradient vector over the pixels of the image. Finally, we modify the image by taking a small step in the direction of the gradient vector. This procedure can be viewed as a gradient-based method for increasing the function

Section 8.1.2

Exercise 10.10

$$F(\mathbf{I}) = \sum_{i,j,k} a_{ijk}(\mathbf{I})^2 \tag{10.12}$$

where $a_{ijk}(\mathbf{I})$ is the pre-activation of the unit in row i and column j of channel k in the chosen layer when the input image is $\mathbf{I}$, and the sum is over all units and over all channels in that layer. To generate smooth-looking images, some regularization is applied in the form of spatial smoothing and pixel clipping. This process can then be repeated multiple times if stronger enhancements are desired. Examples of the resulting image are shown in Figure 10.18. It is interesting that even though convolutional networks are trained to discriminate between object classes, they seem able to capture at least some of the information needed to generate images from those classes.

This technique can be applied to a photograph, or we can start with inputs consisting of random noise to obtain an image generated entirely from the trained network. Although DeepDream provides some insights into the operation of the trained network, it has primarily been used to generate interesting looking images as a form of artwork.

10.4. Object Detection

We have motivated the design of CNNs primarily by the image classification problem, in which an entire image is assigned to a single class, for example 'cat' or 'bicycle'. This is reasonable for data sets such as ImageNet where, by design, each image is dominated by a single object. However, there are many other applications for CNNs that are able to exploit the inbuilt inductive biases. More generally, the convolutional layers of a CNN trained on a large image data base for a particular task can learn internal representations that have broad applicability, and therefore a

Telegram: @uni_k

www.konkur.in

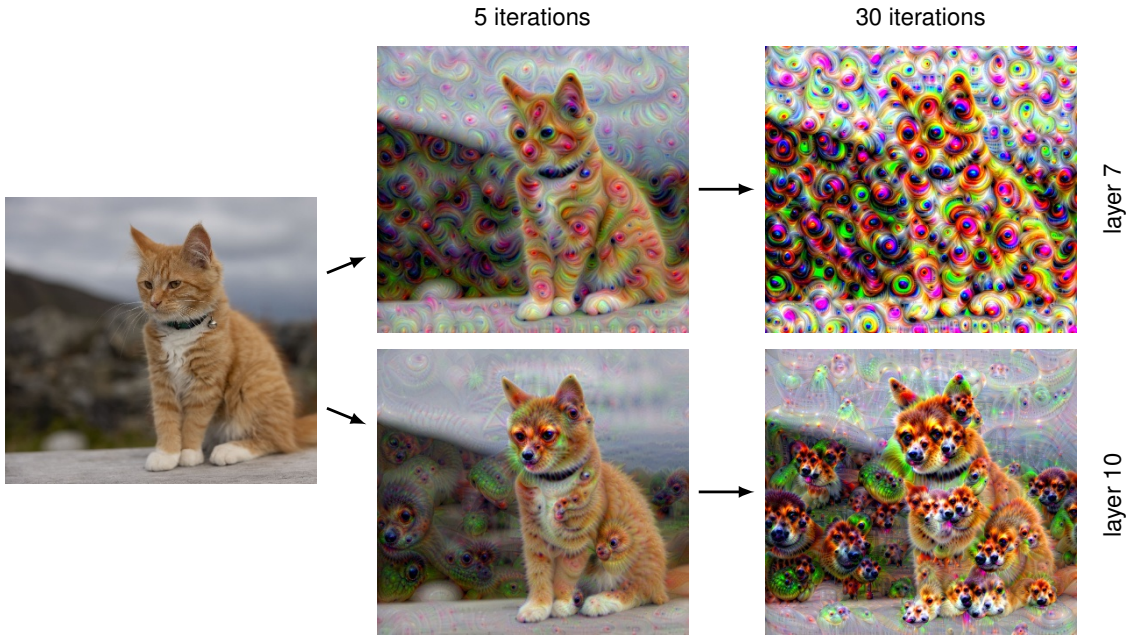


Figure 10.18 Examples of DeepDream applied to an image. The top row shows outputs when the algorithm is applied using the activations from the 7th convolutional layer of the VGG-16 network after five iterations and after 30 iterations. Similarly, the bottom row shows examples using the 10th layer, again after five iterations and after 30 iterations.

Section 6.3.4

Section 1.1.1

CNN can be fine-tuned for a wide range of specific tasks. We have already seen an example of a convolutional network trained on ImageNet data, which through transfer learning was able to achieve human-level performance on skin lesion classification.

10.4.1 Bounding boxes

Many images have multiple objects belonging to one or more classes, and we may wish to detect the presence and class of each object. Moreover, in many applications of computer vision we also need to determine the locations within the image of any objects that are detected. For example, an autonomous vehicle that uses RGB cameras may need to detect the presence and location of pedestrians and also identify road signs, other vehicles, etc.

Consider the problem of specifying the location of an object in an image. A widely used approach is to define a *bounding box*, which consists of a rectangle that fits closely to the boundary of the object, as illustrated in Figure 10.19. The bounding box can be defined by the coordinates of its centre along with its width and height in the form of a vector $\mathbf{b} = (b_x, b_y, b_{\mathrm{W}}, b_{\mathrm{H}})$. Here the elements of $\mathbf{b}$ can be specified in terms of pixels or as continuous numbers where, by convention, the top left of the image is given coordinates $(0, 0)$ and the bottom right is given coordinates $(1, 1)$.

Telegram: @uni_k

www.konkur.in

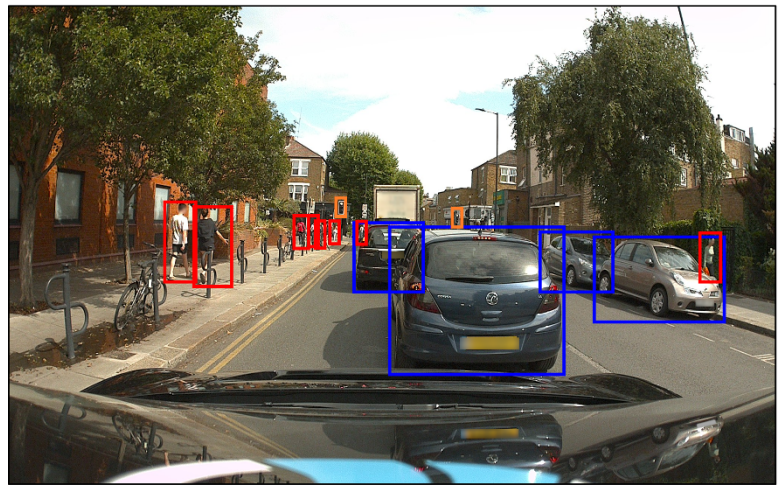


Figure 10.19 An image containing several objects from different classes in which the location of each object is labelled by a close-fitting rectangle known as a bounding box. Here blue boxes correspond to the class 'car', red boxes to the class 'pedestrian', and orange boxes to the class 'traffic light'. [Original image courtesy of Wayve Technologies Ltd.]

Section 5.3

When images are assumed to contain one, and only one, object drawn from a predefined set of C classes, a CNN will generally have C output units whose activation functions are defined by the softmax function. An object can be localized by using an additional four outputs, with linear activation functions trained to predict the bounding box coordinates $(b_x, b_y, b_{\mathrm{W}}, b_{\mathrm{H}})$. Since these quantities are continuous, a sum-of-squares error function over the corresponding outputs may be appropriate. This is used for example by Redmon *et al.* (2015), who first divide the image into a 7×7 grid. For each grid cell, they use a convolutional network to output the class and bounding box coordinates of any object associated with that grid cell, based on features taken from the whole image.

10.4.2 Intersection-over-union

We need a meaningful way to measure the performance of a trained network that can predict bounding boxes. In image classification the output of the network is a probability distribution over class labels, and we can measure performance by looking at the log likelihood for the true class labels on a test set. For object localization, however, we need some way to measure the accuracy of a bounding box relative to some ground truth, where the latter could, for example, be obtained by human labelling. The extent to which the predicted and target boxes overlap can be used as the basis for such a measure, but the area of the overlap will depend on the size of the object within the image. Also a predicted bounding box should be be penalized for the region of the prediction that lies outside the ground truth bounding box. A better metric that addresses both of these issues is called *intersection-over-union*, or IoU, and is simply the ratio of the area of the intersection of the two bounding boxes

Telegram: @uni_k

www.konkur.in

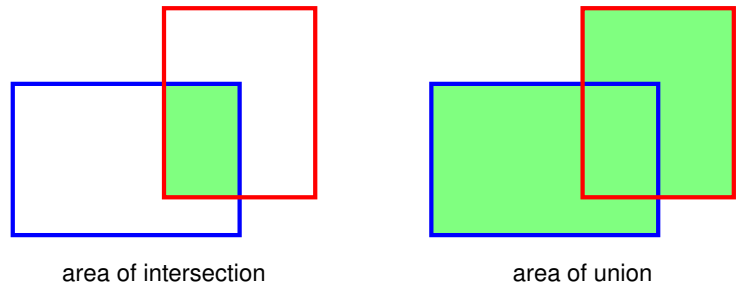


Figure 10.20 Illustration of the intersection-over-union metric for quantifying the accuracy of a bounding box prediction. If the predicted bounding box is shown by the blue rectangle and the ground truth by the red rectangle, then the intersection-over-union is defined as the ratio of the area of the intersection of the boxes, shown in green on the left, divided by the area of their union, shown in green on the right.

to that of their union, as illustrated in Figure 10.20. Note that the IoU measure lies in the range 0 to 1. Predictions can be labelled as correct if the IoU measure exceeds a threshold, which is typically set at 0.5. Note that IoU is not generally used directly as a loss function for training as it is hard to optimize by gradient descent, and so training is typically performed using centred objects, and the IoU score is mainly used an evaluation metric.

10.4.3 Sliding windows

One approach to object detection and object localization starts by creating a training set consisting of tightly cropped examples of the object to be detected, as well as examples of similarly cropped sections of images that do not contain any object (the 'background' class). This data set is used to train a classifier, such as a deep CNN, whose outputs represent the probability of there being an object of each particular class in the input window. The trained model is then used to detect objects in a new image by 'scanning' an input window across the image and, for each location, taking the resulting subset of the image as input to the classifier. This is called a *sliding window*. When an object is detected with high probability, the associated window location then defines the corresponding bounding box.

One obvious drawback of this approach is that it can be computationally very costly due to the large number of potential window positions in the image. Furthermore, the process may have to be repeated using windows of various scales to allow for different sizes of object within the image. A cost saving can be made by moving the input window in strides across the image, both horizontally and vertically, which are larger than one pixel. However, there is a trade-off between precision of location using a small stride and reducing the computational cost by using a larger stride. The computational cost of a sliding window approach may be reasonable for simple classifiers, but for deep neural networks potentially containing millions of parameters, the cost of a naive implementation can be prohibitive.

Fortunately, the convolutional structure of the neural network allows for a dramatic improvement in efficiency (Sermanet *et al.*, 2013). We note that a convolutional layer within such a network itself involves sliding a feature detector, with shared weights, across the input image in strides. Consequently, when a sliding window is used to generate multiple forward passes through a convolutional network

Telegram: @uni_k

www.konkur.in

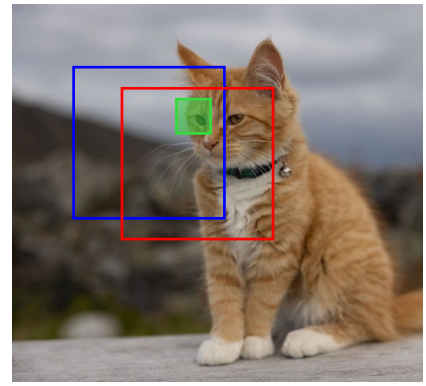


Figure 10.21 Illustration of replicated calculations when a CNN is used to process data from a sliding input window, in which the red and blue boxes show two overlapping locations for the input window. The green box represents one of the locations for the receptive field of a hidden unit in the first convolutional layer, and the evaluation of the corresponding hidden-unit activation is shared across the two window positions.

there is substantial redundancy in the computation, as illustrated in Figure 10.21.

Because the computational structure of sliding windows mirrors that of convolutions, it turns out to be remarkably simple to implement sliding windows efficiently in a convolutional network. Consider the simplified convolutional network in Figure 10.22, which consists of a convolutional layer followed by a max-pooling layer followed by a fully connected layer. For simplicity we have shown only a single channel in each layer, but the extension to multiple channels is straightforward. The input image to the network has size 6×6, the filters in the convolutional layer have size 3×3 with stride 1, and the max-pooling layer has non-overlapping receptive fields of size 2×2 with stride 1. This is followed by a fully connected layer with a single output unit. Note that we can also view this final layer as another convolutional layer with a filter size that is 2×2, so that there is only a single position for the filter and hence a single output.

Now suppose this network is trained on centred images of objects and then applied to a larger image of size 8×8, as shown in Figure 10.23 in which we simply enlarge the network by increasing the size of the convolutional and max-pooling layers. The convolution layer now has size 6×6 and the pooling layer has size

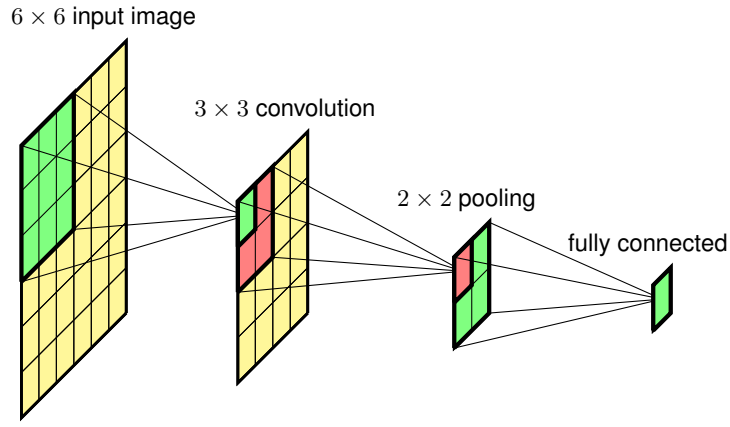


Figure 10.22 Example of a simple convolutional network having a single channel at each layer used to illustrate the concept of a sliding window for detecting objects in images.

Telegram: @uni_k

www.konkur.in

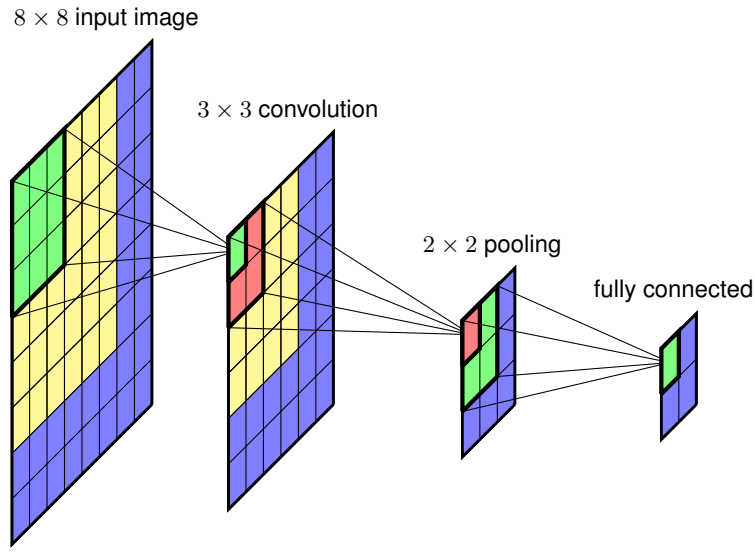


Figure 10.23 Application of the network shown in Figure 10.22 to a larger image in which the additional computation required corresponds to the blue regions.

3×3. There are now four output units each of which has its own softmax function. The weights into this unit are shared across the four units. We see that the calculations needed to process the input corresponding to a window position in the top left corner of the input image are the same as those used to process the original 6×6 inputs used in training. For the remaining window positions, only a small amount of additional computation is needed, as indicated by the blue squares, leading to a significant increase in efficiency compared to a naive repeated application of the full convolutional network. Note that the fully connected layers themselves now have a convolutional structure.

Exercise 10.12

10.4.4 Detection across scales

As well as looking for objects in different positions in the image, we also need to look for objects at different scales and at different aspect ratios. For example, a tight bounding box drawn around a cat will have a different aspect ratio when the cat is sitting upright compared to when it is lying down. Instead of using multiple detectors with different sizes and shapes of input window, it is simpler but equivalent to use a fixed input window and to make multiple copies of the input image each with a different pair of horizontal and vertical scaling factors. The input window is then scanned over each of the image copies to detect objects, and the associated scaling factors are then used to transform the bounding box coordinates back into the original image space, as illustrated in Figure 10.24.

Telegram: @uni_k

www.konkur.in

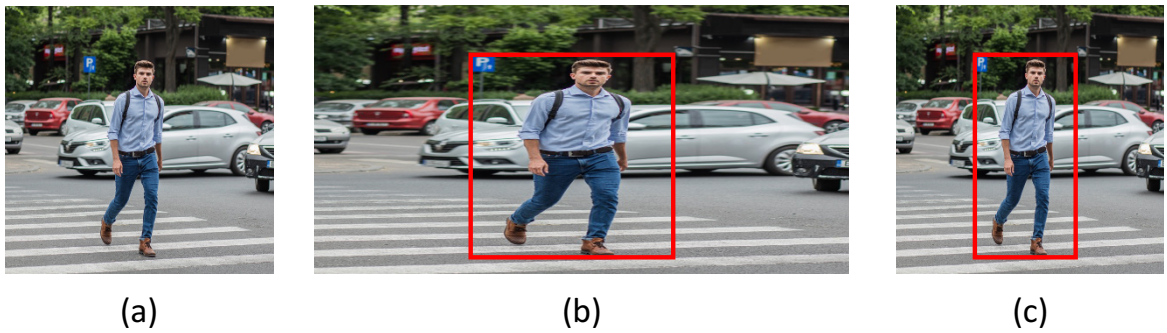


Figure 10.24 Illustration of the detection and localization of objects at multiple scales and aspect ratios using a fixed input window. The original image (a) is replicated multiple times and each copy is scaled in the horizontal and/or vertical directions, as illustrated for a horizontal scaling in (b). A fixed-sized window is then scanned over the scaled images. When an object is detected with high probability, as illustrated by the red box in (b), the corresponding window coordinates can be projected back into the original image space to determine the corresponding bounding box as shown in (c).

10.4.5 Non-max suppression

By scanning a trained convolutional network over an image, it is possible to detect multiple instances of the same class of object within the image as well as instances of objects from other classes. However, this also tends to produce multiple detections of the same object at similar locations, as illustrated in Figure 10.25. This can be addressed using *non-max suppression*, which, for each object class in turn, works as follows. It first runs the sliding window over the whole image and evaluates the probability of an object of that class being present at each location. Next it eliminates all the associated bounding boxes whose probability is below some threshold, say 0.7, giving a result of the kind illustrated in Figure 10.25. The box with the highest probability is considered to be a successful detection, and the corresponding bounding box is recorded as a prediction. Next, any other boxes whose IoU with the successful detection box exceeds some threshold, say 0.5, is discarded. This is intended to eliminate multiple nearby detections of the same object. Then of the remaining boxes, the one with the highest probability is declared to be another successful detection, and the elimination step is repeated. The process continues until all bounding boxes have either been discarded or declared as successful detections.

10.4.6 Fast region CNNs

Another way to speed up object detection and localization is to note that a scanning window approach applies the full power of a deep convolutional network to all areas of the image, even though some areas may be unlikely to contain an object. Instead, we can apply some form of computationally cheaper technique, for example a segmentation algorithm, to identify parts of the image where there is a higher probability of finding an object, and then apply the full network only to these areas, leading to techniques such as *fast region proposals with CNN* or *fast R-CNN*

Telegram: @uni_k

www.konkur.in

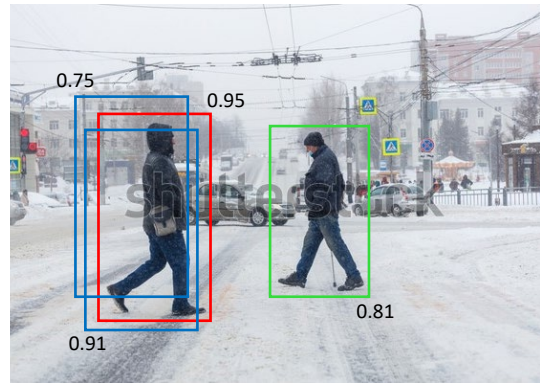


Figure 10.25 Schematic illustration of multiple detections of the same object at nearby locations, along with their associated probabilities. The red bounding box corresponds to the highest overall probability. Non-max suppression eliminates the other overlapping candidate bounding boxes shown in blue, while preserving the detection of another instance of the same object class shown by the bounding box in green.

(Girshick, 2015). It is also possible to use a *region proposal* convolutional network to identify the most promising regions, leading to *faster R-CNN* (Ren *et al.*, 2015), which allows end-to-end training of both the region proposal network and the detection and localization network.

A disadvantage of the sliding window approach is that if we want a very precise localization the objects then we need to consider large numbers of finely spaced window positions, which becomes computationally costly. A more efficient approach is to combine sliding windows with the direct bounding box predictions that we discussed at the start of this section (Sermanet *et al.*, 2013). In this case, the continuous outputs predict the position of the bounding box relative to the window position and therefore provide some fine-tuning to the predicted position.

10.5. Image Segmentation

Section 10.4

In an image classification problem, an entire image is assigned to a single class label. We have seen that more detailed information is provided if multiple objects are detected and their positions recorded using bounding boxes. An even more detailed analysis is obtained with *semantic segmentation* in which every pixel of an image is assigned to one of a predefined set of classes. This means that the output space will have the same dimensionality as the input image and can therefore be conveniently represented as an image with the same number of pixels. Although the input image will generally have three channels for R, G, and B, the output array will have C channels, if there are C classes, representing the probability for each class. If we associate a different (arbitrarily chosen) colour with each class, then the prediction of a segmentation network can be represented as an image in which each pixel is coloured according to the class having the highest probability, as illustrated in Figure 10.26.

10.5.1 Convolutional segmentation

A simple way to approach a semantic segmentation problem would be to construct a convolutional network that takes as input a rectangular section of the image

Telegram: @uni_k

www.konkur.in



Figure 10.26 Example of an image and its corresponding semantic segmentation in which each pixel is coloured according to its class. For example, blue pixels correspond to the class 'car', red pixels to the class 'pedestrian', and orange pixels to the class 'traffic light'. [Courtesy of Wayve Technologies Ltd.]

Figure 10.21

Figure 10.23

centred on a pixel and that has a single softmax output that classifies that pixel. By applying such a network to each pixel in turn, the entire image can be segmented (this would require edge padding around the image depending on the size of the input window). However, this approach would be extremely inefficient due to redundant calculations caused by overlapping patches. As we have seen, we can remove this inefficiency by grouping together the forward-pass calculations for different input locations into a single network, which results in a model in which the final fully connected layers are also convolutional. We could therefore create a CNN in which each layer has the same dimensionality as the input image, by having stride 1 at each layer with same padding and no pooling. Each output unit has a softmax activation function with weights that are shared across all outputs. Although this could work, such a network would still need many layers, with multiple channels in each layer, to learn the complex internal representations needed to achieve high accuracy, and overall this would be prohibitively costly for images of reasonable resolution.

10.5.2 Up-sampling

As we have already seen, most convolutional networks use several levels of down-sampling so that as the number of channels increases, the size of the feature maps decreases, keeping the overall size and cost of the network tractable, while allowing the network to extract semantically meaningful high-order features from the image. We can use this concept to create a more efficient architecture for semantic segmentation by taking a standard deep convolutional network and adding additional learnable layers that take the low-dimensional internal representation and transform it back up to the original image resolution (Long, Shelhamer, and Darrell, 2014; Noh, Hong, and Han, 2015; Badrinarayanan, Kendall, and Cipolla, 2015), as illustrated in Figure 10.27.

To do this we need a way to reverse the down-sampling effects of strided convolutions and pooling operations. Consider first the up-sampling analogue of pooling, where the output layer has a larger number of units than the input layer, for example

Telegram: @uni_k

www.konkur.in

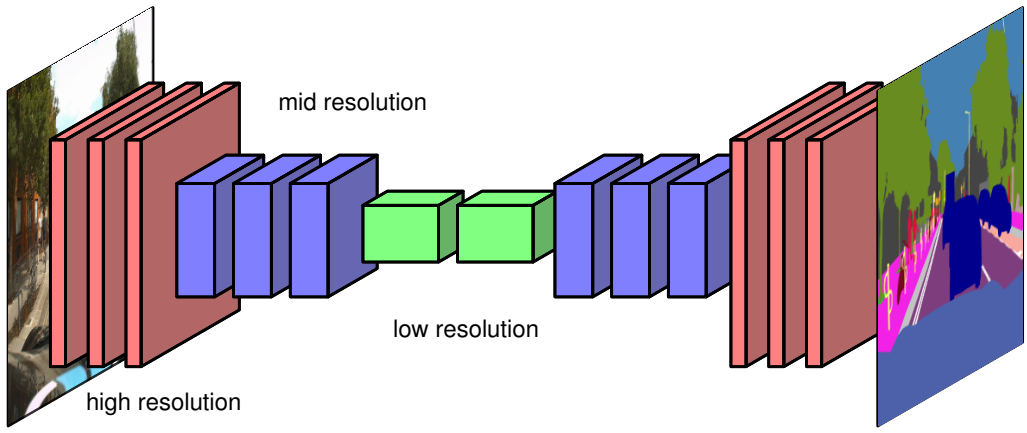


Figure 10.27 Illustration of a convolutional neural network used for semantic image segmentation, showing the reduction in the dimensionality of the feature maps through a series of strided convolutions and/or pooling operations, followed by a series of transpose convolutions and/or unpooling which increase the dimensionality back up to that of the original image.

with each input unit corresponding to a 2×2 block of output units. The question is then what values to use for the outputs. To find an up-sampling analogue of average pooling, we can simply copy over each input value into all the corresponding output units, as shown in Figure 10.28(a). We see that applying average pooling to the output of this operation regenerates the input.

For max-pooling, we can consider the operation shown in Figure 10.28(b) in which each input value is copied into the first unit of the corresponding output block, and the remaining values in each block are set to zero. Again we see that applying a max-pooling operation to the output layer regenerates the input layer. This is sometimes called *max-unpooling*. Assigning the non-zero value to the first element of the output block seems arbitrary, and so a modified approach can be used that also preserves more of the spatial information from the down-sampling layers (Badrinarayanan, Kendall, and Cipolla, 2015). This is done by choosing a network architecture in which each max-pooling down-sampling layer has a corresponding

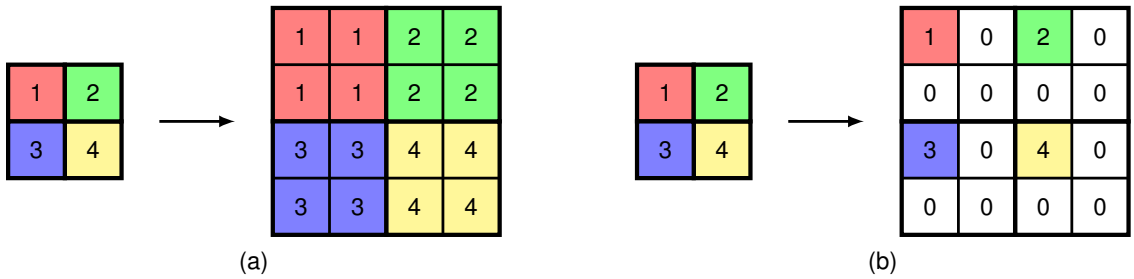


Figure 10.28 Illustration of unpooling operations showing (a) an analogue of average pooling and (b) an analogue of max-pooling.

Telegram: @uni_k

www.konkur.in

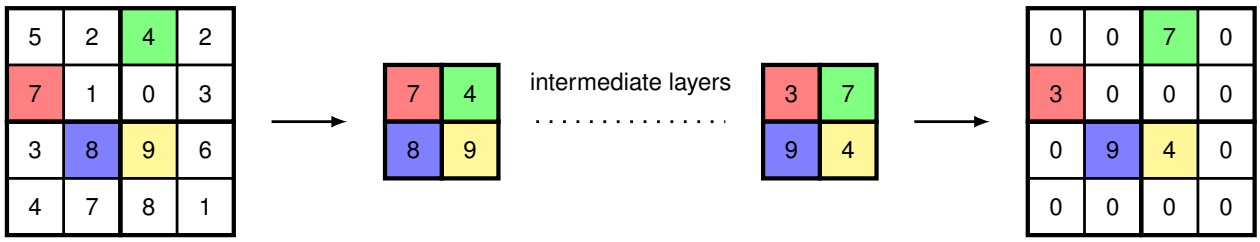


Figure 10.29 Some of the spatial information from a max-pooling layer, shown on the left, can be preserved by noting the location of the maximum value for each 2×2 block in the input array, and then in the corresponding up-sampling layer, placing the non-zero entry at the corresponding location in the output array.

up-sampling layer later in the network. Then during down-sampling, a record is kept of which element in each block had the maximum value, and then in the corresponding up-sampling layer, the non-zero element is chosen to have the same location, as illustrated for 2×2 max-pooling in Figure 10.29.

10.5.3 Fully convolutional networks

The up-sampling methods considered above are fixed functions, much like the average-pooling and max-pooling down-sampling operations. We can also use a *learned* up-sampling that is analogous to strided convolution for down-sampling. In strided convolution, each unit on the output map is connected via shared learnable weights to a small patch on the input map, and as we move one step through the output array, the filter is moved two or more steps across the input array, and hence the output array has lower dimensionality than the input array. For up-sampling, we use a filter that connects one pixel in the input array to a patch in the output array, and then chose the architecture so that as we move one step across the input array, we move two or more steps across the output array (Dumoulin and Visin, 2016). This is illustrated for 3×3 filters, and an output stride of 2, in Figure 10.30. Note that there are output cells for which multiple filter positions overlap, and the corresponding output values can be found either by summing or by averaging the contributions from the individual filter positions.

Exercise 10.13

This up-sampling is called *transpose convolution* because, if the down-sampling convolution is expressed in matrix form, the corresponding up-sampling is given by the transpose matrix. It is also called 'fractionally strided convolution' because the stride of a standard convolution is the ratio of the step size in the output layer to the step size in the input layer. In Figure 10.30, for example, this ratio is $1/2$. Note that this is sometimes also referred to as 'deconvolution', but it is better to avoid this term since deconvolution is widely used in mathematics to mean the inverse of the operation of convolution used in functional analysis, which is a different concept. If we have a network architecture with no pooling layers, so that the down-sampling and up-sampling are done purely using convolutions, then the architecture is known as a *fully convolutional network* (Long, Shelhamer, and Darrell, 2014). It can take an arbitrarily sized image and will output a segmentation map of the same size.

Telegram: @uni_k

www.konkur.in

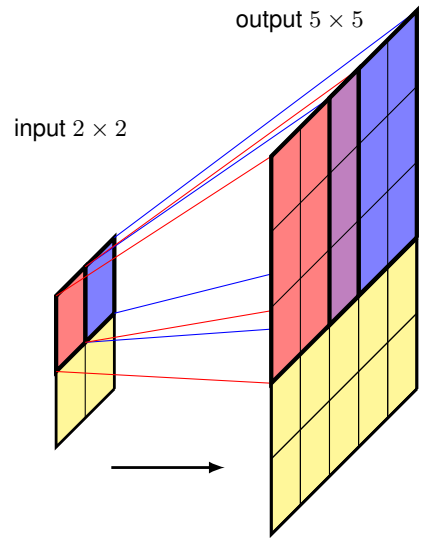


Figure 10.30 Illustration of transpose convolution for a 3×3 filter with an output stride of 2. This can be thought of as the inverse operation to a 3×3 convolution. The red output patch is given by multiplying the kernel by the activation of the red unit in the input layer, and similarly for the blue output patch. The activations of cells for which patches overlap are calculated by summing or averaging the contributions from the individual patches.

10.5.4 The U-net architecture

We have seen that the down-sampling associated with strided convolutions and pooling allows the number of channels to be increased without the size of the network becoming prohibitive. This also has the effect of reducing the spatial resolution and hence discarding positional information as the signals flow through the network. Although this is fine for image classification, the loss of spatial information is a problem for semantic segmentation as we want to classify each pixel. One approach for addressing this is the *U-net* architecture (Ronneberger, Fischer, and Brox, 2015) illustrated in Figure 10.31, where the name comes from the U-shape of the diagram.

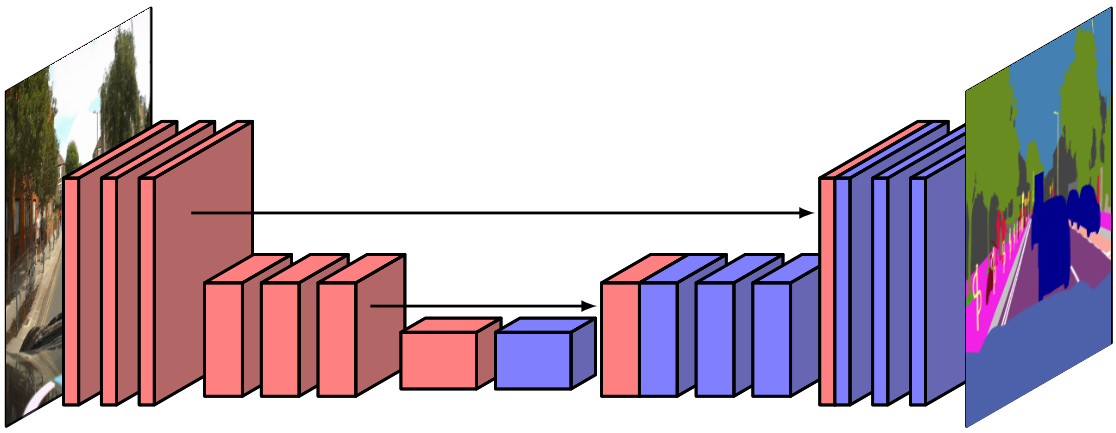


Figure 10.31 The U-net architecture has a symmetrical arrangement of down-sampling and up-sampling layers, and the output from each down-sampling layer is concatenated with the corresponding up-sampling layer.

Telegram: @uni_k

www.konkur.in



Figure 10.32 An example of neural style transfer showing a photograph of a canal scene (left) that has been rendered in the style of *The Wreck of a Transport Ship* by J. M. W. Turner (centre) and in the style of *The Starry Night* by Vincent van Gogh (right). In each case the image used to provide the style is shown in the inset. [From Gatys, Ecker, and Bethge (2015) with permission.]

The core concept is that for each down-sampling layer there is a corresponding up-sampling layer, and the final set of channel activations at each down-sampling layer is concatenated with the corresponding first set of channels in the up-sampling layer, thereby giving those layers access to higher-resolution spatial information. Note that 1×1 convolutions may be used in the final layer of a U-net to reduce the number of channels down to the number of classes, which is then followed by a softmax activation function.

10.6. Style Transfer

Section 10.3

As we have seen, early layers in a deep convolutional network learn to detect simple features such as edges and textures whereas later layers learn to detect more complex entities such as objects. We can exploit this property to re-render an image in the style of a different image using a process called *neural style transfer* (Gatys, Ecker, and Bethge, 2015). This is illustrated in Figure 10.32.

Our goal is to generate a synthetic image $\mathbf{G}$ whose 'content' is defined by an image $\mathbf{C}$ and whose 'style' is taken from some other image $\mathbf{S}$. This is achieved by defining an error function $E(\mathbf{G})$ given by the sum of two terms, one of which encourages $\mathbf{G}$ to have a similar content to $\mathbf{C}$ whereas the other encourages $\mathbf{G}$ to have a similar style to $\mathbf{S}$:

$$E(\mathbf{G}) = E_{\text{content}}(\mathbf{G}, \mathbf{C}) + E_{\text{style}}(\mathbf{G}, \mathbf{S}). \tag{10.13}$$

The concepts of content and style are defined implicitly by the functional forms of these two terms. We can then find $\mathbf{G}$ by starting from a randomly initialized image and using gradient descent to minimize $E(\mathbf{G})$.

To define $E_{\text{content}}(\mathbf{G}, \mathbf{C})$, we can pick a particular convolutional layer in the network and measure the activations of the units in that layer when image $\mathbf{G}$ is used as input and also when image $\mathbf{C}$ is used as input. We can then encourage the

Telegram: @uni_k

www.konkur.in

corresponding pre-activations to be similar by using a sum-of-squares error function of the form

$$E_{\text{content}}(\mathbf{G}, \mathbf{C}) = \sum_{i,j,k} \{a_{ijk}(\mathbf{G}) - a_{ijk}(\mathbf{C})\}^2 \tag{10.14}$$

where $a_{ijk}(\mathbf{G})$ denotes the pre-activation of the unit at position (i, j) in channel k of that layer when the input image is $\mathbf{G}$, and similarly for $a_{ijk}(\mathbf{C})$. The choice of which layer to use in defining the pre-activations will influence the final result, with earlier layers aiming to match low-level features like edges and later layers matching more complex structures or even entire objects.

In defining $E_{\text{style}}(\mathbf{G}, \mathbf{C})$, the intuition is that style is determined by the co-occurrence of features from different channels within a convolutional layer. For example, if the style image $\mathbf{S}$ is such that vertical edges are generally associated with orange blobs, then we would like the same to be true for the generated image $\mathbf{G}$. However, although $E_{\text{content}}(\mathbf{G}, \mathbf{C})$ tries to match features in $\mathbf{G}$ at the same locations as corresponding features in $\mathbf{C}$, for the style error $E_{\text{style}}(\mathbf{G}, \mathbf{S})$ we want $\mathbf{G}$ to have characteristics that match those of $\mathbf{S}$ but taken from any location, and so we take an average over locations in a feature map. Again, consider a particular convolutional layer. We can measure the extent to which a feature in channel k co-occurs with the corresponding feature in channel k' for input image $\mathbf{G}$ by forming the cross-correlation matrix

$$F_{kk'}(\mathbf{G}) = \sum_{i=1}^{I} \sum_{j=1}^{J} a_{ijk}(\mathbf{G}) a_{ijk'}(\mathbf{G}) \tag{10.15}$$

where I and J are the dimensions of the feature maps in this particular convolutional layer, and the product $a_{ijk}a_{ijk'}$ will be large if both features are activated. If there are K channels in this layer, then $F_{kk'}$ form the elements of a $K \times K$ matrix, called the *style matrix*. We can measure the extent to which the two images $\mathbf{G}$ and $\mathbf{S}$ have the same style by comparing their style matrices using

$$E_{\text{style}}(\mathbf{G}, \mathbf{S}) = \frac{1}{(2IJK)^2} \sum_{k=1}^{K} \sum_{k'=1}^{K} \{F_{kk'}(\mathbf{G}) - F_{kk'}(\mathbf{S})\}^2 . \tag{10.16}$$

Although we could again make use of a single layer, more pleasing results are obtained by using contributions from multiple layers in the form

$$E_{\text{style}}(\mathbf{G}, \mathbf{S}) = \sum_{l} \lambda_l E^{(l)}_{\text{style}}(\mathbf{G}, \mathbf{S}) \tag{10.17}$$

where l denotes the convolutional layer. The coefficients λ_l determine the relative weighting between the different layers and also the weighting relative to the content error term. These weighting coefficients are adjusted empirically using subjective judgement.

Telegram: @uni_k

www.konkur.in

Exercises

10.1 ($\star$) Consider a fixed weight vector $\mathbf{w}$ and show that the input vector $\mathbf{x}$ that maximizes the scalar product $\mathbf{w}^{\mathrm{T}}\mathbf{x}$, subject to the constraint that $\|\mathbf{x}\|^2$ is constant, is given by $\mathbf{x} = \alpha\mathbf{w}$ for some scalar α. This can most easily be done using a Lagrange multiplier.

Appendix C

10.2 ($\star\star$) Consider a convolutional network layer with a one-dimensional input array and a one-dimensional feature map as shown in Figure 10.2, in which the input array has dimensionality 5 and the filters have width 3 with a stride of 1. Show that this can be expressed as a special case of a fully connected layer by writing down the weight matrix in which missing connections are replaced by zeros and where shared parameters are indicated by using replicated entries. Ignore any bias parameters.

10.3 ($\star$) Explicitly calculate the output of the following convolution of a 4×4 input matrix with a 2×2 filter:

$$\begin{array}{|c|c|c|c|} \hline 2 & 5 & -3 & 0 \\ \hline 0 & 6 & 0 & -4 \\ \hline -1 & -3 & 0 & 2 \\ \hline 5 & 0 & 0 & 3 \\ \hline \end{array} \quad * \quad \begin{array}{|c|c|} \hline -2 & 0 \\ \hline 4 & 6 \\ \hline \end{array} \quad = \quad \begin{array}{|c|c|c|} \hline ? & ? & ? \\ \hline ? & ? & ? \\ \hline ? & ? & ? \\ \hline \end{array} \tag{10.18}$$

10.4 ($\star\star$) If an image $\mathbf{I}$ has $J \times K$ pixels and a filter $\mathbf{K}$ has $L \times M$ elements, write down the limits for the two summations in (10.2). In the mathematics literature, the operation (10.2) would be called a *cross-correlation*, whereas a *convolution* would be defined by

$$C(j,k) = \sum_{l}\sum_{m} I(j-l, k-m)K(l,m). \tag{10.19}$$

Write down the limits for the summations in (10.19). Show that (10.19) can be written in the equivalent 'flipped' form

$$C(j,k) = \sum_{l}\sum_{m} I(j+l, k+m)K(l,m) \tag{10.20}$$

and again write down the limits for the summations.

10.5 ($\star$) In mathematics, a convolution for a continuous variable x is defined by

$$F(x) = \int_{-\infty}^{\infty} G(y)k(x-y)\,\mathrm{d}y \tag{10.21}$$

where $k(x-y)$ is the kernel function. By considering a discrete approximation to the integral, explain the relationship to a convolutional layer, defined by (10.19), in a CNN.

Telegram: @uni_k

www.konkur.in

10.6 ($\star$) Consider an image of size $J \times K$ that is padded with an additional P pixels on all sides and which is then convolved using a kernel of size $M \times M$ where M is an odd number. Show that if we choose $P = (M-1)/2$, then the resulting feature map will have size $J \times K$ and hence will be the same size as the original image.

10.7 ($\star$) Show that if a kernel of size $M \times M$ is convolved with an image of size $J \times K$ with padding of depth P and strides of length S then the dimensionality of the resulting feature map is given by (10.5)

10.8 ($\star\star$) For each of the 16 layers in the VGG-16 CNN shown in Figure 10.10, evaluate (i) the number of weights (i.e., connections) including biases and (ii) the number of independently learnable parameters. Confirm that the total number of learnable parameters in the network is approximately 138 million.

10.9 ($\star\star$) Consider a convolution of the form (10.2) and suppose that the kernel is separable so that

$$K(l, m) = F(l)G(m) \tag{10.22}$$

for some functions $F(\cdot)$ and $G(\cdot)$. Show that instead of performing a single two-dimensional convolution it is now possible to compute the same answer using two one-dimensional convolutions thereby resulting in a significant improvement in efficiency.

10.10 ($\star$) The DeepDream update procedure involves setting the δ variables for backpropagation equal to the pre-activations of the nodes in the chosen layer and then running backpropagation to the input layer to get a gradient vector over the pixels of the image. Show that this can be derived as a gradient optimization with respect to the pixels of an image $\mathbf{I}$ applied to the function (10.12).

10.11 ($\star\star$) When designing a neural network to detect objects from C different classes in an image, we can use a 1-of-$(C+1)$ class label with one variable for each object class and one additional variable representing a 'background' class, i.e., an input image region that does not contain an object belonging to any of the defined classes. The network will then output a vector of probabilities of length $(C + 1)$. Alternatively, we can use a single binary variable to denote the presence or absence of an object and then use a separate 1-of-C vector to denote the specific object class. In this case, the network outputs a single probability representing the presence of an object and a separate set of probabilities over the class label. Write down the relationship between these two sets of probabilities.

10.12 ($\star\star$) Calculate the number of computational steps required to make one forward pass through the convolutional network shown in Figure 10.22, ignoring biases and ignoring the evaluation of activation functions. Similarly, calculate the total number of computational steps for a single forward pass through the expanded network shown in Figure 10.23. Finally, evaluate ratio of nine repeated naive applications of the network in Figure 10.22 to an 8×8 image compared to a single application of the network in Figure 10.23. This ratio indicates the improvement in efficiency from using a convolutional implementation of the sliding window technique.

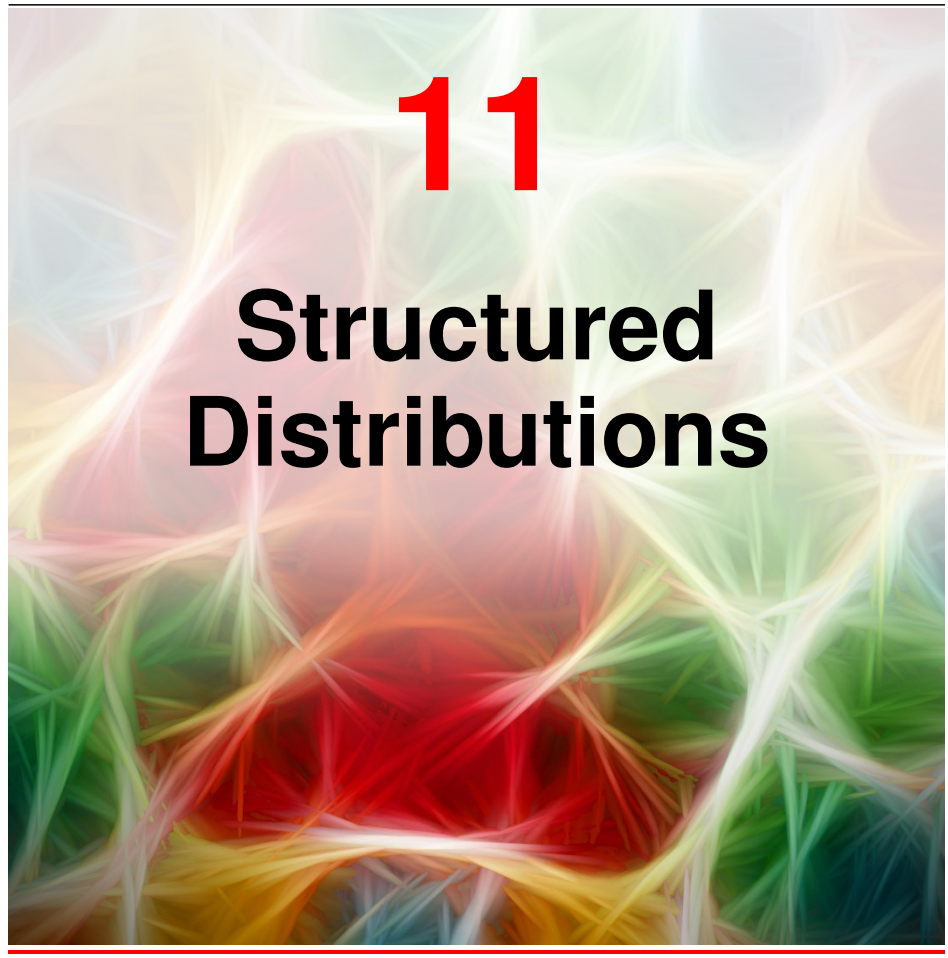
Telegram: @uni_k

www.konkur.in

10.13 ($\star\star$) In this exercise we use one-dimensional vectors to demonstrate why a convolutional up-sampling is sometimes called a transpose convolution. Consider a one-dimensional strided convolutional layer with an input having four units with activations (x_1, x_2, x_3, x_4), which is padded with zeros to give $(0, x_1, x_2, x_3, x_4, 0)$, and a filter with parameters (w_1, w_2, w_3). Write down the one-dimensional activation vector of the output layer assuming a stride of 2. Express this output in the form of a matrix $\mathbf{A}$ multiplied by the vector $(0, x_1, x_2, x_3, x_4, 0)$. Now consider an up-sampling convolution in which the input layer has activations (z_1, z_2) with a filter having values (w_1, w_2, w_3) and an output stride of 2. Write down the six-dimensional output vector assuming that overlapping filter values are summed and that the activation function is just the identity. Show that this can be expressed as a matrix multiplication using the transpose matrix $\mathbf{A}^{\mathrm{T}}$.

Telegram: @uni_k

www.konkur.in



11 Structured Distributions

We have seen that probability forms one of the most important foundational concepts for deep learning. For example, a neural network used for binary classification is described by a conditional probability distribution of the form

$$p(t|\mathbf{x}, \mathbf{w}) = y(\mathbf{x}, \mathbf{w})^t \{1 - y(\mathbf{x}, \mathbf{w})\}^{(1-t)} \tag{11.1}$$

where $y(\mathbf{x}, \mathbf{w})$ represents a neural network function that takes a vector $\mathbf{x}$ as input and is governed by a vector $\mathbf{w}$ of learnable parameters. The corresponding cross-entropy likelihood forms the basis for defining an error function used to train the neural network. Although the network function might be extremely complex, the conditional distribution in (11.1) has a simple form. However, there are many important deep learning models that have a much richer probabilistic structure, such as large language models, normalizing flows, variational autoencoders, diffusion models, and many others. To describe and exploit this structure, we introduce a powerful

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024
C. M. Bishop, H. Bishop, *Deep Learning*, https://doi.org/10.1007/978-3-031-45468-4_11

Telegram: @uni_k

www.konkur.in

framework called *probabilistic graphical models*, or simply *graphical models*, which allows structured probability distributions to be expressed in graphical form. When combined with neural networks to define associated probability distributions, graphical models offer huge flexibility when creating sophisticated models that can be trained end to end using stochastic gradient descent in which gradients are evaluated efficiently using auto-differentiation. In this chapter, we will focus on the core concepts of graphical models needed for applications in deep learning, whereas a more comprehensive treatment of graphical models for machine learning can be found in Bishop (2006).

11.1. Graphical Models

Section 2.1

Probability theory can be expressed in terms of two simple equations known as the *sum rule* and the *product rule*. All of the probabilistic manipulations discussed in this book, no matter how complex, amount to repeated application of these two equations. In principle, we could therefore formulate and use complex probabilistic models purely by using algebraic manipulation. However, we will find it advantageous to augment the analysis using diagrammatic representations of probability distributions, as these offer several useful properties:

1. They provide a simple way to visualize the structure of a probabilistic model and can be used to design and motivate new models.
2. Insights into the properties of the model, including conditional independence properties, can be obtained by inspecting the graph.
3. The complex computations required to perform inference and learning in sophisticated models can be expressed in terms of graphical operations, such as message-passing, in which the underlying mathematical operations are carried out implicitly.

Although such graphical models have nodes and edges much like neural network diagrams, their interpretation is specifically probabilistic and carries a richer semantics. To help avoid confusion, in this book we denote neural network diagrams in blue and probabilistic graphical models in red.

11.1.1 Directed graphs

A graph comprises *nodes*, also called *vertices*, connected by *links*, also known as *edges*. In a probabilistic graphical model, each node represents a random variable, and the links express probabilistic relationships between these variables. The graph then captures the way in which the joint distribution over all the random variables can be decomposed into a product of factors each depending only on a subset of the variables. In this chapter we will focus on graphical models in which the links of the graphs have a particular direction indicated by arrows. These are known as *directed graphical models* and are also called *Bayesian networks* or *Bayes nets*.

Telegram: @uni_k

www.konkur.in

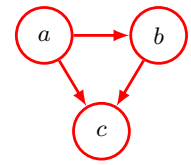


Figure 11.1 A directed graphical model representing the joint probability distribution over three variables a, b, and c, corresponding to the decomposition on the right-hand side of (11.3).

The other major class of graphical models are *Markov random fields*, also known as *undirected graphical models*, in which the links do not carry arrows and have no directional significance. Directed graphs are useful for expressing causal relationships between random variables, whereas undirected graphs are better suited to expressing soft constraints between random variables. Both directed and undirected graphs can be viewed as special cases of a representation called *factor graphs*. From now on we focus our attention on directed graphical models. Note, however, that undirected graphs, without the probabilistic interpretation, will also arise in our discussion of *graph neural networks* in which the nodes represent deterministic variables as in standard neural networks.

Chapter 13

11.1.2 Factorization

To motivate the use of directed graphs to describe probability distributions, consider first an arbitrary joint distribution $p(a, b, c)$ over three variables a, b, and c. Note that at this stage, we do not need to specify anything further about these variables, such as whether they are discrete or continuous. Indeed, one of the powerful aspects of graphical models is that a specific graph can make probabilistic statements for a broad class of distributions. By application of the product rule of probability (2.9), we can write the joint distribution in the form

$$p(a, b, c) = p(c|a, b)p(a, b). \tag{11.2}$$

A second application of the product rule, this time to the second term on the right-hand side of (11.2), gives

$$p(a, b, c) = p(c|a, b)p(b|a)p(a). \tag{11.3}$$

Note that this decomposition holds for any choice of the joint distribution. We now represent the right-hand side of (11.3) in terms of a simple graphical model as follows. First we introduce a node for each of the random variables a, b, and c and associate each node with the corresponding conditional distribution on the right-hand side of (11.3). Then, for each conditional distribution we add directed links (depicted as arrows) from the nodes corresponding to the variables on which the distribution is conditioned. Thus, for the factor $p(c|a, b)$, there will be links from nodes a and b to node c, whereas for the factor $p(a)$, there will be no incoming links. The result is the graph shown in Figure 11.1. If there is a link going from node a to node b, then we say that node a is the *parent* of node b, and we say that node b is the *child* of node a. Note that we will not make any formal distinction between a node and the variable to which it corresponds but will simply use the same symbol to refer to both.

Telegram: @uni_k

www.konkur.in

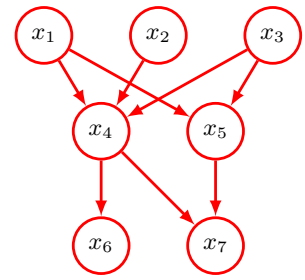


Figure 11.2 Example of a directed graph describing the joint distribution over variables $x_1, \ldots, x_7$. The corresponding decomposition of the joint distribution is given by (11.5).

An important point to note about (11.3) is that the left-hand side is symmetrical with respect to the three variables a, b, and c, whereas the right-hand side is not. In making the decomposition in (11.3), we have implicitly chosen a particular ordering, namely a, b, c, and had we chosen a different ordering we would have obtained a different decomposition and hence a different graphical representation.

For the moment let us extend the example of Figure 11.1 by considering the joint distribution over K variables given by $p(x_1, \ldots, x_K)$. By repeated application of the product rule of probability, this joint distribution can be written as a product of conditional distributions, one for each of the variables:

$$p(x_1, \ldots, x_K) = p(x_K|x_1, \ldots, x_{K-1}) \ldots p(x_2|x_1)p(x_1). \tag{11.4}$$

For a given choice of K, we can again represent this as a directed graph having K nodes, one for each conditional distribution on the right-hand side of (11.4), with each node having incoming links from all lower numbered nodes. We say that this graph is *fully connected* because there is a link between every pair of nodes.

So far, we have worked with completely general joint distributions, and so their factorization, and associated representation as fully connected graphs, will be applicable to any choice of distribution. As we will see shortly, it is the *absence* of links in the graph that conveys interesting information about the properties of the class of distributions that the graph represents. Consider the graph shown in Figure 11.2. Note that it is not a fully connected graph because, for instance, there is no link from x_1 to x_2 or from x_3 to x_7. We take this graph and extract the corresponding representation of the joint probability distribution written in terms of the product of a set of conditional distributions, one for each node in the graph. Each such conditional distribution will be conditioned only on the parents of the corresponding node in the graph. For instance, x_5 will be conditioned on x_1 and x_3. The joint distribution of all seven variables is therefore given by

$$p(x_1)p(x_2)p(x_3)p(x_4|x_1, x_2, x_3)p(x_5|x_1, x_3)p(x_6|x_4)p(x_7|x_4, x_5). \tag{11.5}$$

The reader should take a moment to study carefully the correspondence between (11.5) and Figure 11.2.

We can now state in general terms the relationship between a given directed graph and the corresponding distribution over the variables. The joint distribution defined by a graph is given by the product, over all of the nodes of the graph, of

Telegram: @uni_k

www.konkur.in

a conditional distribution for each node conditioned on the variables corresponding to the parents of that node in the graph. Thus, for a graph with K nodes, the joint distribution is given by

$$p(x_1, \ldots, x_K) = \prod_{k=1}^{K} p(x_k|\mathrm{pa}(k)) \tag{11.6}$$

where $\mathrm{pa}(k)$ denotes the set of parents of x_k. This key equation expresses the *factorization* properties of the joint distribution for a directed graphical model. Although we have considered each node to correspond to a single variable, we can equally well associate sets of variables and vector-valued or tensor-valued variables with the nodes of a graph. It is easy to show that the representation on the right-hand side of (11.6) is always correctly normalized provided the individual conditional distributions are normalized.

Exercise 11.1

The directed graphs that we are considering are subject to an important restriction, namely that there must be no *directed cycles*. In other words, there are no closed paths within the graph such that we can move from node to node along links following the direction of the arrows and end up back at the starting node. Such graphs are also called *directed acyclic graphs*, or *DAGs*. This is equivalent to the statement that there exists an ordering of the nodes such that there are no links that go from any node to any lower-numbered node.

Exercise 11.2

11.1.3 Discrete variables

We have discussed the importance of probability distributions that are members of the exponential family, and we have seen that this family includes many well-known distributions as special cases. Although such distributions are relatively simple, they form useful building blocks for constructing more complex probability distributions, and the framework of graphical models is very useful in expressing the way in which these building blocks are linked together. There are two particular choices for the component distributions that are widely used, corresponding to discrete variables and to Gaussian variables. We begin by examining the discrete case.

Section 3.4

The probability distribution $p(\mathbf{x}|\boldsymbol{\mu})$ for a single discrete variable $\mathbf{x}$ having K possible states (using the 1-of-K representation) is given by

$$p(\mathbf{x}|\boldsymbol{\mu}) = \prod_{k=1}^{K} \mu_k^{x_k} \tag{11.7}$$

and is governed by the parameters $\boldsymbol{\mu} = (\mu_1, \ldots, \mu_K)^{\mathrm{T}}$. Due to the constraint $\sum_k \mu_k = 1$, only $K - 1$ values for μ_k need to be specified to define the distribution.

Now suppose that we have two discrete variables, $\mathbf{x}_1$ and $\mathbf{x}_2$, each of which has K states, and we wish to model their joint distribution. We denote the probability of observing both $x_{1k} = 1$ and $x_{2l} = 1$ by the parameter μ_{kl}, where x_{1k} denotes the

Telegram: @uni_k

www.konkur.in

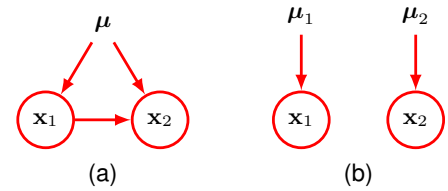


Figure 11.3 (a) This fully connected graph describes a general distribution over two K-state discrete variables having a total of $K^2 - 1$ parameters. (b) By dropping the link between the nodes, the number of parameters is reduced to $2(K - 1)$.

kth component of $\mathbf{x}_1$, and similarly for x_{2l}. The joint distribution can be written

$$p(\mathbf{x}_1, \mathbf{x}_2|\boldsymbol{\mu}) = \prod_{k=1}^{K}\prod_{l=1}^{K} \mu_{kl}^{x_{1k}x_{2l}}.$$

Because the parameters μ_{kl} are subject to the constraint $\sum_k \sum_l \mu_{kl} = 1$, this distribution is governed by $K^2 - 1$ parameters. It is easily seen that the total number of parameters that must be specified for an arbitrary joint distribution over M variables is $K^M - 1$ and therefore grows exponentially with the number M of variables.

Using the product rule, we can factor the joint distribution $p(\mathbf{x}_1, \mathbf{x}_2)$ in the form $p(\mathbf{x}_2|\mathbf{x}_1)p(\mathbf{x}_1)$, which corresponds to a two-node graph with a link going from the $\mathbf{x}_1$ node to the $\mathbf{x}_2$ node as shown in Figure 11.3(a). The marginal distribution $p(\mathbf{x}_1)$ is governed by $K - 1$ parameters, as before. Similarly, the conditional distribution $p(\mathbf{x}_2|\mathbf{x}_1)$ requires the specification of $K - 1$ parameters for each of the K possible values of $\mathbf{x}_1$. The total number of parameters that must be specified in the joint distribution is therefore $(K - 1) + K(K - 1) = K^2 - 1$ as before.

Now suppose that the variables $\mathbf{x}_1$ and $\mathbf{x}_2$ are independent, corresponding to the graphical model shown in Figure 11.3(b). Each variable is then described by a separate discrete distribution, and the total number of parameters would be $2(K-1)$. For a distribution over M independent discrete variables, each having K states, the total number of parameters would be $M(K - 1)$, which therefore grows linearly with the number of variables. From a graphical perspective, we have reduced the number of parameters by dropping links in the graph, at the expense of having a more restricted class of distributions.

More generally, if we have M discrete variables $\mathbf{x}_1, \ldots, \mathbf{x}_M$, we can model the joint distribution using a directed graph with one variable for each node. The conditional distribution at each node is given by a set of non-negative parameters subject to the usual normalization constraint. If the graph is fully connected, then we have a completely general distribution having $K^M - 1$ parameters, whereas if there are no links in the graph, the joint distribution factorizes into the product of the marginal distributions, and the total number of parameters is $M(K - 1)$. Graphs having intermediate levels of connectivity allow for more general distributions than the fully factorized one while requiring fewer parameters than the general joint distribution. As an illustration, consider the chain of nodes shown in Figure 11.4. The marginal distribution $p(\mathbf{x}_1)$ requires $K - 1$ parameters, whereas each of the $M - 1$ conditional distributions $p(\mathbf{x}_i|\mathbf{x}_{i-1})$, for $i = 2, \ldots, M$, requires $K(K - 1)$ parameters.

Telegram: @uni_k

www.konkur.in

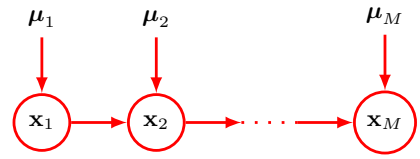


Figure 11.4 This chain of M discrete nodes, each having K states, requires the specification of $K-1+(M-1)K(K-1)$ parameters, which grows linearly with the length M of the chain. In contrast, a fully connected graph of M nodes would have K^M-1 parameters, which grows exponentially with M.

This gives a total parameter count of $K-1+(M-1)K(K-1)$, which is quadratic in K and which grows linearly (rather than exponentially) with the length M of the chain.

An alternative way to reduce the number of independent parameters in a model is by *sharing* parameters (also known as *tying* of parameters). For instance, in the chain example of Figure 11.4, we can arrange that all the conditional distributions $p(\mathbf{x}_i|\mathbf{x}_{i-1})$, for $i=2,\ldots,M$, are governed by the same set of $K(K-1)$ parameters, giving the model shown in Figure 11.5. Together with the $K-1$ parameters governing the distribution of $\mathbf{x}_1$, this gives a total of K^2-1 parameters that must be specified to define the joint distribution.

Another way of controlling the exponential growth of the number of parameters in models of discrete variables is to use parameterized representations for the conditional distributions instead of complete tables of conditional probability values. To illustrate this idea, consider the graph in Figure 11.6 in which all the nodes represent binary variables. Each of the parent variables x_i is governed by a single parameter μ_i representing the probability $p(x_i=1)$, giving M parameters in total for the parent nodes. The conditional distribution $p(y|x_1,\ldots,x_M)$, however, would require 2^M parameters representing the probability $p(y=1)$ for each of the 2^M possible settings of the parent variables. Thus, in general the number of parameters required to specify this conditional distribution will grow exponentially with M. We can obtain a more parsimonious form for the conditional distribution by using a logistic sigmoid function acting on a linear combination of the parent variables, giving

Section 3.4

$$p(y=1|x_1,\ldots,x_M)=\sigma\left(w_0+\sum_{i=1}^{M}w_ix_i\right)=\sigma(\mathbf{w}^{\mathrm{T}}\mathbf{x}) \tag{11.8}$$

where $\sigma(a)=(1+\exp(-a))^{-1}$ is the logistic sigmoid, $\mathbf{x}=(x_0,x_1,\ldots,x_M)^{\mathrm{T}}$ is an $(M+1)$-dimensional vector of parent states augmented with an additional variable x_0 whose value is clamped to 1, and $\mathbf{w}=(w_0,w_1,\ldots,w_M)^{\mathrm{T}}$ is a vector of $M+1$ parameters. This is a more restricted form of conditional distribution than the general case but is now governed by a number of parameters that grows linearly with M. In

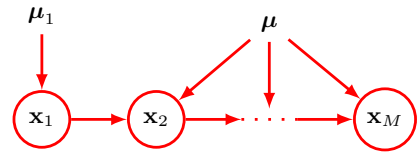


Figure 11.5 As in Figure 11.4 but with a single set of parameters $\boldsymbol{\mu}$ shared amongst all the conditional distributions $p(\mathbf{x}_i|\mathbf{x}_{i-1})$.

Telegram: @uni_k

www.konkur.in

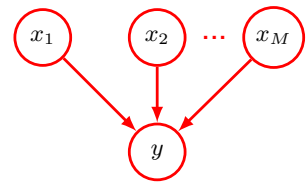


Figure 11.6 A graph comprising M parents $x_1, \ldots, x_M$ and a single child y, used to illustrate the idea of parameterized conditional distributions for discrete variables.

this sense, it is analogous to the choice of a restrictive form of covariance matrix (for example, a diagonal matrix) in a multivariate Gaussian distribution.

11.1.4 Gaussian variables

We now turn to graphical models in which the nodes represent continuous variables having Gaussian distributions. Each distribution is conditioned on the state of its parents in the graph. That dependence could take many forms, and here we focus on a specific choice in which the mean of each Gaussian is some linear function of the states of the Gaussian parent variables. This leads to a class of models called *linear-Gaussian models*, which include many cases of practical interest such as probabilistic principal component analysis, factor analysis, and linear dynamical systems (Roweis and Ghahramani, 1999).

Section 16.2

Consider an arbitrary directed acyclic graph over D variables in which node i represents a single continuous random variable x_i having a Gaussian distribution. The mean of this distribution is taken to be a linear combination of the states of its parent nodes $\mathrm{pa}(i)$ of node i:

$$p(x_i|\mathrm{pa}(i)) = \mathcal{N}\left(x_i \,\middle|\, \sum_{j\in\mathrm{pa}(i)} w_{ij}x_j + b_i, v_i\right) \tag{11.9}$$

where w_{ij} and b_i are parameters governing the mean and v_i is the variance of the conditional distribution for x_i. The log of the joint distribution is then the log of the product of these conditionals over all nodes in the graph and hence takes the form

$$\ln p(\mathbf{x}) = \sum_{i=1}^{D} \ln p(x_i|\mathrm{pa}(i)) \tag{11.10}$$

$$= -\sum_{i=1}^{D} \frac{1}{2v_i}\left(x_i - \sum_{j\in\mathrm{pa}(i)} w_{ij}x_j - b_i\right)^2 + \text{const} \tag{11.11}$$

where $\mathbf{x} = (x_1, \ldots, x_D)^{\mathrm{T}}$ and 'const' denotes terms independent of $\mathbf{x}$. We see that this is a quadratic function of the components of $\mathbf{x}$, and hence the joint distribution $p(\mathbf{x})$ is a multivariate Gaussian.

We can find the mean and covariance of this joint distribution as follows. The mean of each variable is given by the recursion relation

Exercise 11.6

$$\mathbb{E}[x_i] = \sum_{j\in\mathrm{pa}(i)} w_{ij}\mathbb{E}[x_j] + b_i. \tag{11.12}$$

Telegram: @uni_k

www.konkur.in



Figure 11.7 A directed graph over three Gaussian variables with one missing link.

and so we can find the components of $\mathbb{E}[\mathbf{x}] = (\mathbb{E}[x_1], \ldots, \mathbb{E}[x_D])^{\mathrm{T}}$ by starting at the lowest numbered node and working recursively through the graph, where we assume that the nodes are numbered such that each node has a higher number than its parents. Similarly, the elements of the covariance matrix of the joint distribution satisfy a recursion relation of the form

Exercise 11.7

$$\mathrm{cov}[x_i, x_j] = \sum_{k \in \mathrm{pa}(j)} w_{jk} \mathrm{cov}[x_i, x_k] + I_{ij} v_j \tag{11.13}$$

and so the covariance can similarly be evaluated recursively starting from the lowest numbered node.

We now consider two extreme cases of possible graph structures. First, suppose that there are no links in the graph, which therefore comprises D isolated nodes. In this case, there are no parameters w_{ij} and so there are just D parameters b_i and D parameters v_i. From the recursion relations (11.12) and (11.13), we see that the mean of $p(\mathbf{x})$ is given by $(b_1, \ldots, b_D)^{\mathrm{T}}$ and the covariance matrix is diagonal of the form $\mathrm{diag}(v_1, \ldots, v_D)$. The joint distribution has a total of $2D$ parameters and represents a set of D independent univariate Gaussian distributions.

Now consider a fully connected graph in which each node has all lower numbered nodes as parents. In this case the total number of independent parameters $\{w_{ij}\}$ and $\{v_i\}$ in the covariance matrix is $D(D+1)/2$ corresponding to a general symmetric covariance.

Exercise 11.8

Graphs having some intermediate level of complexity correspond to joint Gaussian distributions with partially constrained covariance matrices. Consider for example the graph shown in Figure 11.7, which has a link missing between variables x_1 and x_3. Using the recursion relations (11.12) and (11.13), we see that the mean and covariance of the joint distribution are given by

Exercise 11.9

$$\boldsymbol{\mu} = (b_1, b_2 + w_{21}b_1, b_3 + w_{32}b_2 + w_{32}w_{21}b_1)^{\mathrm{T}} \tag{11.14}$$

$$\boldsymbol{\Sigma} = \begin{pmatrix} v_1 & w_{21}v_1 & w_{32}w_{21}v_1 \\ w_{21}v_1 & v_2 + w_{21}^2 v_1 & w_{32}(v_2 + w_{21}^2 v_1) \\ w_{32}w_{21}v_1 & w_{32}(v_2 + w_{21}^2 v_1) & v_3 + w_{32}^2(v_2 + w_{21}^2 v_1) \end{pmatrix}. \tag{11.15}$$

We can readily extend the linear-Gaussian graphical model to a situation in which the nodes of the graph represent multivariate Gaussian variables. In this case, we can write the conditional distribution for node i in the form

$$p(\mathbf{x}_i | \mathrm{pa}(i)) = \mathcal{N}\left(\mathbf{x}_i \,\middle|\, \sum_{j \in \mathrm{pa}(i)} \mathbf{W}_{ij}\mathbf{x}_j + \mathbf{b}_i, \boldsymbol{\Sigma}_i\right) \tag{11.16}$$

where now $\mathbf{W}_{ij}$ is a matrix (which is non-square if $\mathbf{x}_i$ and $\mathbf{x}_j$ have different dimensionality). Again it is easy to verify that the joint distribution over all variables is Gaussian.

Exercise 11.10

Telegram: @uni_k

www.konkur.in

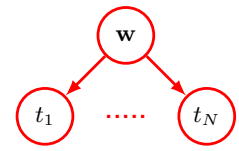


Figure 11.8 Directed graphical model representing the binary classifier model described by the joint distribution (11.17) showing only the stochastic variables $\{t_1, \ldots, t_N\}$ and $\mathbf{w}$.

11.1.5 Binary classifier

Section 2.6.2

We can illustrate the use of directed graphs to describe probability distributions using a two-class classifier model with Gaussian prior over the learnable parameters. We can write this in the form

$$p(\mathbf{t}, \mathbf{w}|\mathbf{X}, \lambda) = p(\mathbf{w}|\lambda) \prod_{n=1}^{N} p(t_n|\mathbf{w}, \mathbf{x}_n) \tag{11.17}$$

where $\mathbf{t} = (t_1, \ldots, t_N)^{\mathrm{T}}$ is the vector of target values, $\mathbf{X}$ is the data matrix with rows $\mathbf{x}_1^{\mathrm{T}}, \ldots \mathbf{x}_N^{\mathrm{T}}$, and the distribution $p(t|\mathbf{x}, \mathbf{w})$ is given by (11.1). We also assume a Gaussian prior over the parameter vector $\mathbf{w}$ given by

$$p(\mathbf{w}|\lambda) = \mathcal{N}(\mathbf{w}|\mathbf{0}, \lambda\mathbf{I}). \tag{11.18}$$

The stochastic variables in this model are $\{t_1, \ldots, t_N\}$ and $\mathbf{w}$. In addition, this model contains the noise variance σ^2 and the hyperparameter λ, both of which are parameters of the model rather than stochastic variables. If we consider for a moment only the stochastic variables, then the distribution given by (11.17) can be represented by the graphical model shown in Figure 11.8.

When we start to deal with more complex models, it becomes inconvenient to have to write out multiple nodes of the form $t_1, \ldots, t_N$ explicitly as in Figure 11.8. We therefore introduce a graphical notation that allows such multiple nodes to be expressed more compactly. We draw a single representative node t_n and then surround this with a box, called a *plate*, labelled with N to indicate that there are N nodes of this kind. Rewriting the graph of Figure 11.8 in this way, we obtain the graph shown in Figure 11.9.

11.1.6 Parameters and observations

We will sometimes find it helpful to make the parameters of a model, as well as its stochastic variables, explicit in the graphical representation. To do this, we will adopt the convention that random variables are denoted by open circles and deterministic parameters are denoted by floating variables. If we take the graph of

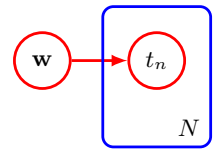


Figure 11.9 An alternative, more compact, representation of the graph shown in Figure 11.8 in which we have introduced a *plate* (the box labelled N) that represents N nodes of which only a single example t_n is shown explicitly.

Telegram: @uni_k

www.konkur.in

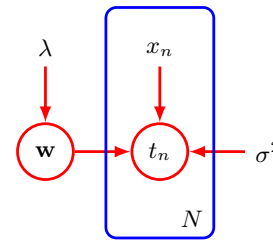


Figure 11.10 The same model as in Figure 11.9 but with the deterministic parameters shown explicitly by the floating variables.

Figure 11.9 and include the deterministic parameters, we obtain the graph shown in Figure 11.10.

When we apply a graphical model to a problem in machine learning, we will typically set some of the random variables to specific observed values. For example, the stochastic variables $\{t_n\}$ in the linear regression model will be set equal to the specific values given in the training set. In a graphical model, we denote such *observed variables* by shading the corresponding nodes. Thus, the graph corresponding to Figure 11.10 in which the variables $\{t_n\}$ are observed is shown in Figure 11.11.

Note that the value of $\mathbf{w}$ is not observed, and so $\mathbf{w}$ is an example of a *latent* variable, also known as a *hidden* variable. Such variables play a crucial role in many of the models discussed in this book. We therefore have three kinds of variables in a directed graphical model. First, there are unobserved (also called latent, or hidden) stochastic variables, which are denoted by open red circles. Second, when stochastic variables are observed, so that that they are set to specific values, they are denoted by red circles shaded with blue. Finally, non-stochastic parameters are denoted by floating variables, as seen in Figure 11.11.

Note that model parameters such as $\mathbf{w}$ are generally of little direct interest in themselves, because our ultimate goal is to make predictions for new input values. Suppose we are given a new input value $\widehat{x}$ and we wish to find the corresponding probability distribution for $\widehat{t}$ conditioned on the observed data. The joint distribution of all the random variables in this model, conditioned on the deterministic parameters, is given by

$$p(\widehat{t}, \mathbf{t}, \mathbf{w}|\widehat{x}, \mathbf{X}, \lambda) = p(\mathbf{w}|\lambda)p(\widehat{t}|\mathbf{w}, \widehat{x}) \prod_{n=1}^{N} p(t_n|\mathbf{w}, \mathbf{x}_n) \tag{11.19}$$

and the corresponding graphical model is shown in Figure 11.12.

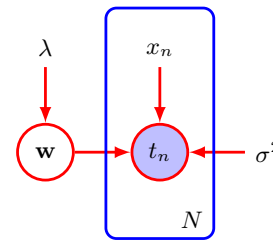


Figure 11.11 As in Figure 11.10 but with the nodes $\{t_n\}$ shaded to indicate that the corresponding random variables have been set to their observed values given by the training set.

Telegram: @uni_k

www.konkur.in

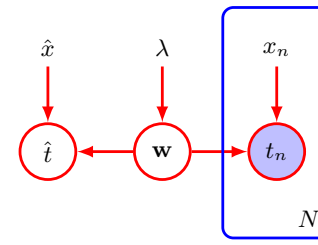


Figure 11.12 The classification model, corresponding to Figure 11.11, showing a new input value $\widehat{x}$ together with the corresponding model prediction $\widehat{t}$.

The required predictive distribution for $\widehat{t}$ is then obtained from the sum rule of probability by integrating out the model parameters $\mathbf{w}$. This integration over parameters represents a fully Bayesian treatment, which is rarely used in practice, especially with deep neural networks. Instead, we approximate this integral by first finding the most probable value $\mathbf{w}_{\text{MAP}}$ that maximizes the posterior distribution and then using just this single value to make predictions using $p(\widehat{t}|\mathbf{w}_{\text{MAP}}, \widehat{x})$.

11.1.7 Bayes' theorem

When stochastic variables in a probabilistic model are set equal to observed values, the distributions over other unobserved stochastic variables change accordingly. The process of calculating these updated distributions is known as *inference*. We can illustrate this by considering the graphical interpretation of Bayes' theorem. Suppose we decompose the joint distribution $p(x, y)$ over two variables x and y into a product of factors in the form $p(x, y) = p(x)p(y|x)$. This can be represented by the directed graph shown in Figure 11.13(a). Now suppose we observe the value of y, as indicated by the shaded node in Figure 11.13(b). We can view the marginal distribution $p(x)$ as a prior over the latent variable x, and our goal is to infer the corresponding posterior. Using the sum and product rules of probability we can evaluate

$$p(y) = \sum_{x'} p(y|x')p(x'), \tag{11.20}$$

which can then be used in Bayes' theorem to calculate

$$p(x|y) = \frac{p(y|x)p(x)}{p(y)}. \tag{11.21}$$

Thus, the joint distribution is now expressed in terms of $p(x|y)$ and $p(y)$. From a graphical perspective, the joint distribution $p(x, y)$ is represented by the graph shown in Figure 11.13(c), in which the direction of the arrow is reversed. This is the simplest example of an inference problem for a graphical model.

For complex graphical models that capture rich probabilistic structure, the process of calculating posterior distributions once some of the stochastic variables are observed can be complex and subtle. Conceptually, it simply involves the systematic application of the sum and product rules of probability, or equivalently Bayes' theorem. In practice, however, managing these calculations efficiently can benefit greatly from an exploitation of the graphical structure. These calculations can be expressed

Telegram: @uni_k

www.konkur.in

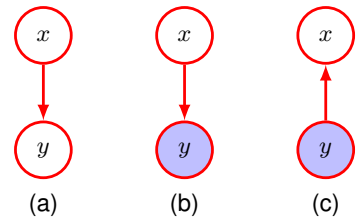


Figure 11.13 A graphical representation of Bayes' theorem showing (a) a joint distribution over two variables x and y expressed in factorized form, (b) the case with y set to an observed value, and (c) the resulting posterior distribution over x, given by Bayes' theorem.

in terms of elegant calculations on the graph that involve sending local messages between nodes. Such methods give exact answers for tree-structured graphs and give approximate iterative algorithms for graphs with loops. Since we will not discuss these further here, see Bishop (2006) for a more comprehensive discussion in the context of machine learning.

11.2. Conditional Independence

An important concept for probability distributions over multiple variables is that of *conditional independence* (Dawid, 1980). Consider three variables a, b, and c, and suppose that the conditional distribution of a given b and c is such that it does not depend on the value of b, so that

$$p(a|b,c) = p(a|c). \tag{11.22}$$

We say that a is conditionally independent of b given c. This can be expressed in a slightly different way if we consider the joint distribution of a and b conditioned on c, which we can write in the form

$$\begin{aligned} p(a,b|c) &= p(a|b,c)p(b|c) \\ &= p(a|c)p(b|c) \end{aligned} \tag{11.23}$$

where we have used the product rule of probability together with (11.22). We see that, conditioned on c, the joint distribution of a and b factorizes into the product of the marginal distribution of a and the marginal distribution of b (again both conditioned on c). This says that the variables a and b are statistically independent, given c. Note that our definition of conditional independence will require that (11.22), or equivalently (11.23), must hold for every possible value of c, and not just for some values. We will sometimes use a shorthand notation for conditional independence (Dawid, 1979) in which

$$a \perp\!\!\!\perp b \mid c \tag{11.24}$$

denotes that a is conditionally independent of b given c. Conditional independence properties play an important role in probabilistic models for machine learning because they simplify both the structure of a model and the computations needed to perform inference and learning under that model.

Telegram: @uni_k

www.konkur.in

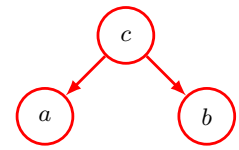


Figure 11.14 The first of three examples of graphs over three variables a, b, and c used to discuss conditional independence properties of directed graphical models.

If we are given an expression for the joint distribution over a set of variables in terms of a product of conditional distributions (i.e., the mathematical representation underlying a directed graph), then we could in principle test whether any potential conditional independence property holds by repeated application of the sum and product rules of probability. In practice, such an approach would be very time-consuming. An important and elegant feature of graphical models is that conditional independence properties of the joint distribution can be read directly from the graph without having to perform any analytical manipulations. The general framework for achieving this is called *d-separation*, where the 'd' stands for 'directed' (Pearl, 1988). Here we will motivate the concept of d-separation and give a general statement of the d-separation criterion. A formal proof can be found in Lauritzen (1996).

11.2.1 Three example graphs

We begin our discussion of the conditional independence properties of directed graphs by considering three simple examples each involving graphs having just three nodes. Together, these will motivate and illustrate the key concepts of d-separation. The first of the three examples is shown in Figure 11.14, and the joint distribution corresponding to this graph is easily written down using the general result (11.6) to give

$$p(a, b, c) = p(a|c)p(b|c)p(c). \tag{11.25}$$

If none of the variables are observed, then we can investigate whether a and b are independent by marginalizing both sides of (11.25) with respect to c to give

$$p(a, b) = \sum_c p(a|c)p(b|c)p(c). \tag{11.26}$$

In general, this does not factorize into the product $p(a)p(b)$, and so

$$a \not\perp\!\!\!\perp b \mid \emptyset \tag{11.27}$$

where $\emptyset$ denotes the empty set, and the symbol $\not\perp\!\!\!\perp$ means that the conditional independence property does not hold in general. Of course, it may hold for a particular distribution by virtue of the specific numerical values associated with the various conditional probabilities, but it does not follow in general from the structure of the graph.

Now suppose we condition on the variable c, as represented by the graph of Figure 11.15. From (11.25), we can easily write down the conditional distribution of

Telegram: @uni_k

www.konkur.in

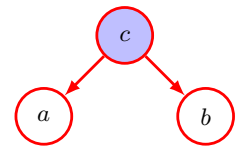


Figure 11.15 As in Figure 11.14 but where we have conditioned on the value of variable c.

a and b, given c, in the form

$$\begin{aligned} p(a,b|c) &= \frac{p(a,b,c)}{p(c)} \\ &= p(a|c)p(b|c) \end{aligned}$$

and so we obtain the conditional independence property

$$a \perp\!\!\!\perp b \mid c.$$

We can provide a simple graphical interpretation of this result by considering the path from node a to node b via c. The node c is said to be *tail-to-tail* with respect to this path because the node is connected to the tails of the two arrows, and the presence of such a path connecting nodes a and b causes these nodes to be dependent. However, when we condition on node c, as in Figure 11.15, the conditioned node 'blocks' the path from a to b and causes a and b to become (conditionally) independent.

We can similarly consider the graph shown in Figure 11.16. The joint distribution corresponding to this graph is again obtained from our general formula (11.6) to give

$$p(a,b,c) = p(a)p(c|a)p(b|c). \tag{11.28}$$

First, suppose that none of the variables are observed. Again, we can test to see if a and b are independent by marginalizing over c to give

$$p(a,b) = p(a)\sum_c p(c|a)p(b|c) = p(a)p(b|a)$$

which in general does not factorize into $p(a)p(b)$, and so

$$a \not\perp\!\!\!\perp b \mid \emptyset \tag{11.29}$$

as before.

Now suppose we condition on node c, as shown in Figure 11.17. Using Bayes'

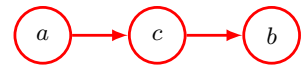
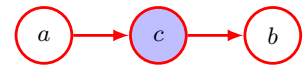


Figure 11.16 The second of our three examples of three-node graphs used to motivate the conditional independence framework for directed graphical models.

Telegram: @uni_k

www.konkur.in

Figure 11.17 As in Figure 11.16 but now conditioning on node c.



theorem together with (11.28), we obtain

$$\begin{aligned} p(a,b|c) &= \frac{p(a,b,c)}{p(c)} \\ &= \frac{p(a)p(c|a)p(b|c)}{p(c)} \\ &= p(a|c)p(b|c) \end{aligned}$$

and so again we obtain the conditional independence property

$$a \perp\!\!\!\perp b \mid c.$$

As before, we can interpret these results graphically. The node c is said to be *head-to-tail* with respect to the path from node a to node b. Such a path connects nodes a and b and renders them dependent. If we now observe c, as in Figure 11.17, then this observation 'blocks' the path from a to b and so we obtain the conditional independence property $a \perp\!\!\!\perp b \mid c$.

Finally, we consider the third of our three-node examples, shown by the graph in Figure 11.18. As we will see, this has a more subtle behaviour than the two previous graphs. The joint distribution can again be written down using our general result (11.6) to give

$$p(a,b,c) = p(a)p(b)p(c|a,b). \tag{11.30}$$

Consider first the case where none of the variables are observed. Marginalizing both sides of (11.30) over c we obtain

$$p(a,b) = p(a)p(b)$$

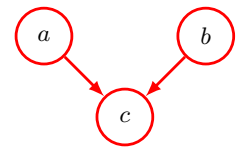
and so a and b are independent with no variables observed, in contrast to the two previous examples. We can write this result as

$$a \perp\!\!\!\perp b \mid \emptyset. \tag{11.31}$$

Now suppose we condition on c, as indicated in Figure 11.19. The conditional distribution of a and b is then given by

$$\begin{aligned} p(a,b|c) &= \frac{p(a,b,c)}{p(c)} \\ &= \frac{p(a)p(b)p(c|a,b)}{p(c)}, \end{aligned}$$

Figure 11.18 The last of our three examples of three-node graphs used to explore conditional independence properties in graphical models. This graph has rather different properties from the two previous examples.



Telegram: @uni_k

www.konkur.in

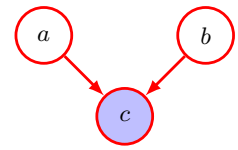


Figure 11.19 As in Figure 11.18 but conditioning on the value of node c. In this graph, the act of conditioning induces a dependence between a and b.

which in general does not factorize into the product $p(a|c)p(b|c)$, and so

$$a \not\perp\!\!\!\perp b \mid c.$$

Thus, our third example has the opposite behaviour from the first two. Graphically, we say that node c is *head-to-head* with respect to the path from a to b because it connects to the heads of the two arrows. The node c is sometimes called a *collider node*. When node c is unobserved, it 'blocks' the path, and the variables a and b are independent. However, conditioning on c 'unblocks' the path and renders a and b dependent.

There is one more subtlety associated with this third example that we need to consider. First we introduce some more terminology. We say that node y is a *descendant* of node x if there is a path from x to y in which each step of the path follows the directions of the arrows. Then it can be shown that a head-to-head path will become unblocked if either the node, *or any of its descendants*, is observed.

Exercise 11.13

In summary, a tail-to-tail node or a head-to-tail node leaves a path unblocked unless it is observed, in which case it blocks the path. By contrast, a head-to-head node blocks a path if it is unobserved, but once the node and/or at least one of its descendants is observed the path becomes unblocked.

11.2.2 Explaining away

It is worth spending a moment to understand further the unusual behaviour of the graph in Figure 11.19. Consider a particular instance of such a graph corresponding to a problem with three binary random variables relating to the fuel system on a car, as shown in Figure 11.20. The variables are B, which represents the state of a battery that is either charged ($B = 1$) or flat ($B = 0$), F which represents the state of the fuel tank that is either full of fuel ($F = 1$) or empty ($F = 0$), and G, which is the state of an electric fuel gauge and which indicates that the fuel tank is either full ($G = 1$) or empty ($G = 0$). The battery is either charged or flat, and independently,

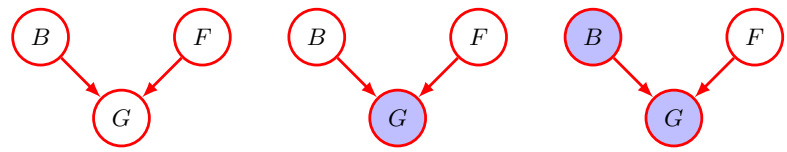


Figure 11.20 An example of a three-node graph used to illustrate 'explaining away'. The three nodes represent the state of the battery (B), the state of the fuel tank (F), and the reading on the electric fuel gauge (G). See the text for details.

Telegram: @uni_k

www.konkur.in

the fuel tank is either full or empty, with prior probabilities

$$\begin{aligned} p(B=1) &= 0.9 \\ p(F=1) &= 0.9. \end{aligned}$$

Given the state of the fuel tank and the battery, the fuel gauge reads full with probabilities given by

$$\begin{aligned} p(G=1|B=1,F=1) &= 0.8 \\ p(G=1|B=1,F=0) &= 0.2 \\ p(G=1|B=0,F=1) &= 0.2 \\ p(G=1|B=0,F=0) &= 0.1 \end{aligned}$$

so this is a rather unreliable fuel gauge! All remaining probabilities are determined by the requirement that probabilities sum to one, and so we have a complete specification of the probabilistic model.

Before we observe any data, the prior probability of the fuel tank being empty is $p(F=0)=0.1$. Now suppose that we observe the fuel gauge and discover that it reads empty, i.e., $G=0$, corresponding to the middle graph in Figure 11.20. We can use Bayes' theorem to evaluate the posterior probability of the fuel tank being empty. First we evaluate the denominator for Bayes' theorem:

$$p(G=0) = \sum_{B\in\{0,1\}} \sum_{F\in\{0,1\}} p(G=0|B,F)p(B)p(F) = 0.315 \tag{11.32}$$

and similarly we evaluate

$$p(G=0|F=0) = \sum_{B\in\{0,1\}} p(G=0|B,F=0)p(B) = 0.81 \tag{11.33}$$

and using these results, we have

$$p(F=0|G=0) = \frac{p(G=0|F=0)p(F=0)}{p(G=0)} \simeq 0.257 \tag{11.34}$$

and so $p(F=0|G=0) > p(F=0)$. Thus, observing that the gauge reads empty makes it more likely that the tank is indeed empty, as we would intuitively expect. Next suppose that we also check the state of the battery and find that it is flat, i.e., $B=0$. We have now observed the states of both the fuel gauge and the battery, as shown by the right-hand graph in Figure 11.20. The posterior probability that the fuel tank is empty given the observations of both the fuel gauge and the battery state is then given by

$$p(F=0|G=0,B=0) = \frac{p(G=0|B=0,F=0)p(F=0)}{\sum_{F\in\{0,1\}} p(G=0|B=0,F)p(F)} \simeq 0.111 \tag{11.35}$$

where the prior probability $p(B=0)$ has cancelled between the numerator and denominator. Thus, the probability that the tank is empty has *decreased* (from 0.257

Telegram: @uni_k

www.konkur.in

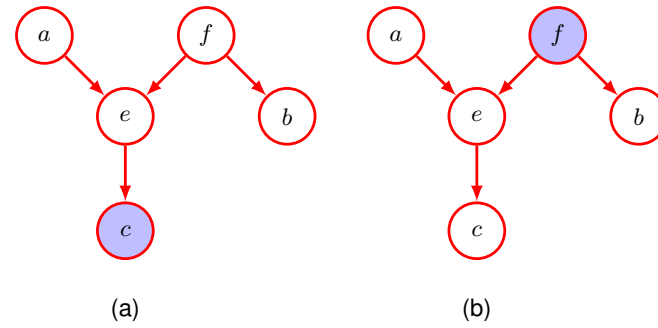


Figure 11.21 Illustration of d-separation. See the text for details.

to 0.111) as a result of the observation of the state of the battery. This accords with our intuition that finding that the battery is flat *explains away* the observation that the fuel gauge reads empty. We see that the state of the fuel tank and that of the battery have indeed become dependent on each other as a result of observing the reading on the fuel gauge. In fact, this would also be the case if, instead of observing the fuel gauge directly, we observed the state of some descendant of G, for example a rather unreliable witness who reports seeing that the gauge was reading empty. Note that the probability $p(F = 0|G = 0, B = 0) \simeq 0.111$ is greater than the prior probability $p(F = 0) = 0.1$ because the observation that the fuel gauge reads zero still provides some evidence in favour of an empty fuel tank.

Exercise 11.14

11.2.3 D-separation

We now give a general statement of the d-separation property (Pearl, 1988) for directed graphs. Consider a general directed graph in which A, B, and C are arbitrary non-intersecting sets of nodes (whose union may be smaller than the complete set of nodes in the graph). We wish to ascertain whether a particular conditional independence statement $A \perp\!\!\!\perp B \mid C$ is implied by a given directed acyclic graph. To do so, we consider all possible paths from any node in A to any node in B. Any such path is said to be *blocked* if it includes a node such that either

(a) the arrows on the path meet either head-to-tail or tail-to-tail at the node, and the node is in the set C, or

(b) the arrows meet head-to-head at the node and neither the node, nor any of its descendants is in the set C.

If all paths are blocked, then A is said to be d-separated from B by C, and the joint distribution over all the variables in the graph will satisfy $A \perp\!\!\!\perp B \mid C$.

D-separation is illustrated in Figure 11.21. In graph (a), the path from a to b is not blocked by node f because it is a tail-to-tail node for this path and is not observed, nor is it blocked by node e because, although the latter is a head-to-head node, it has a descendant c in the conditioning set. Thus, the conditional independence statement $a \perp\!\!\!\perp b \mid c$ does *not* follow from this graph. In graph (b), the path from a to b is blocked by node f because this is a tail-to-tail node that is observed, and so the conditional independence property $a \perp\!\!\!\perp b \mid f$ will be satisfied by any distribution that factorizes

Telegram: @uni_k

www.konkur.in

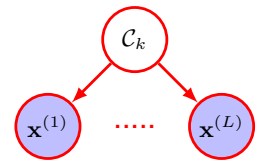


Figure 11.22 A graphical representation of the naive Bayes model for classification. Conditioned on the class label $\mathcal{C}_k$, the elements of the observed vector $\mathbf{x} = (\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(L)})$ are assumed to be independent.

according to this graph. Note that this path is also blocked by node e because e is a head-to-head node and neither it nor its descendant are in the conditioning set. In d-separation, parameters such as λ in Figure 11.12, which are indicated by floating variables, behave in the same way as observed nodes. However, there are no marginal distributions associated with such nodes, and consequently parameter nodes never themselves have parents and so all paths through these nodes will always be tail-to-tail and hence blocked. Consequently they play no role in d-separation.

Section 2.3.2

Another example of conditional independence and d-separation is provided by i.i.d. (independent and identically distributed) data. Consider the binary classification model shown in Figure 11.12. Here the stochastic nodes correspond to $\{t_n\}$, $\mathbf{w}$, and $\widehat{t}$. We see that the node for $\mathbf{w}$ is tail-to-tail with respect to the path from $\widehat{t}$ to any one of the nodes t_n, and so we have the following conditional independence property:

$$\widehat{t} \perp\!\!\!\perp t_n \mid \mathbf{w}. \tag{11.36}$$

Thus, conditioned on the network parameters $\mathbf{w}$, the predictive distribution for $\widehat{t}$ is independent of the training data $\{t_1, \ldots, t_N\}$. We can therefore first use the training data to determine the posterior distribution (or some approximation to the posterior distribution) over the coefficients $\mathbf{w}$ and then we can discard the training data and use the posterior distribution for $\mathbf{w}$ to make predictions of $\widehat{t}$ for new input observations $\widehat{x}$.

11.2.4 Naive Bayes

A related graphical structure arises in an approach to classification called the *naive Bayes* model, in which we use conditional independence assumptions to simplify the model structure. Suppose our data consists of observations of a vector $\mathbf{x}$, and we wish to assign values of $\mathbf{x}$ to one of K classes. We can define a class-conditional density $p(\mathbf{x}|\mathcal{C}_k)$ for each of the classes, along with prior class probabilities $p(\mathcal{C}_k)$. The key assumption of the naive Bayes model is that, conditioned on the class $\mathcal{C}_k$, the distribution of the input variable factorizes into the product of two or more densities. Suppose we partition $\mathbf{x}$ into L elements $\mathbf{x} = (\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(L)})$. Naive Bayes then takes the form

$$p(\mathbf{x}|\mathcal{C}_k) = \prod_{l=1}^{L} p(\mathbf{x}^{(l)}|\mathcal{C}_k) \tag{11.37}$$

where it is assumed that (11.37) holds for each of the classes $\mathcal{C}_k$ separately. The graphical representation of this model is shown in Figure 11.22. We see that an observation of $\mathcal{C}_k$ would block the path between $\mathbf{x}^{(i)}$ and $\mathbf{x}^{(j)}$ for $j \neq i$ because such

Telegram: @uni_k

www.konkur.in

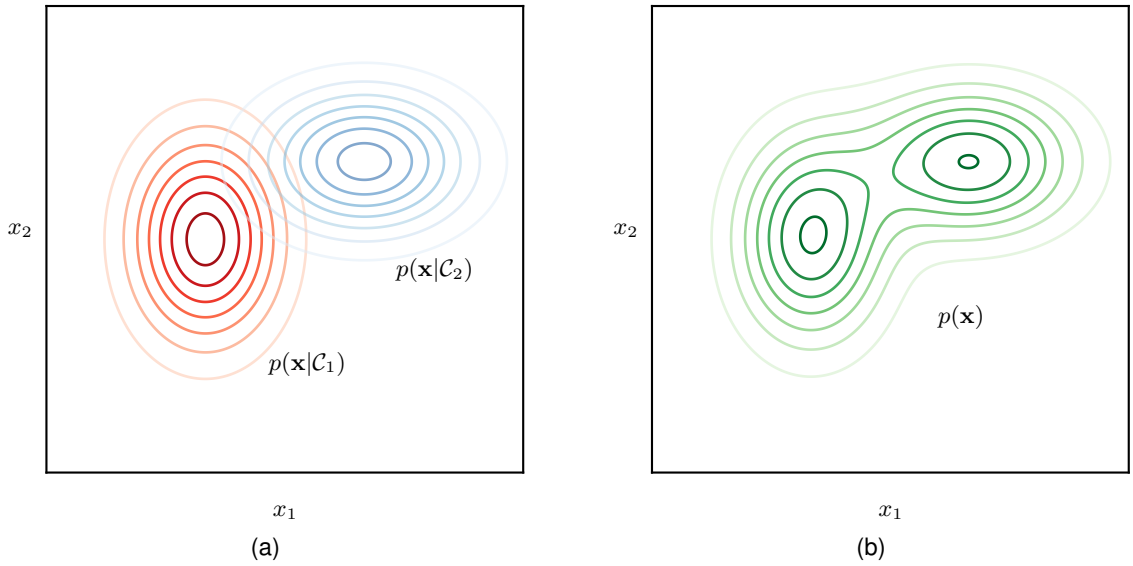


Figure 11.23 Illustration of a naive Bayes classifier for a two-dimensional data space, showing (a) the conditional distributions $p(\mathbf{x}|\mathcal{C}_k)$ for each of the two classes and (b) the marginal distribution $p(\mathbf{x})$ in which we have assumed equal class priors $p(\mathcal{C}_1) = p(\mathcal{C}_2) = 0.5$. Note that the conditional distributions factorize with respect to x_1 and x_2, whereas the marginal distribution does not.

paths are tail-to-tail at the node $\mathcal{C}_k$, and so $\mathbf{x}^{(i)}$ and $\mathbf{x}^{(j)}$ are conditionally independent given $\mathcal{C}_k$. If, however, we marginalize out $\mathcal{C}_k$, the tail-to-tail path from $\mathbf{x}^{(i)}$ to $\mathbf{x}^{(j)}$ is no longer blocked, which tells us that in general the marginal density $p(\mathbf{x})$ will not factorize with respect to the elements $\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(L)}$.

If we are given a labelled training set, comprising observations $\{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$ together with their class labels, then we can fit the naive Bayes model to the training data using maximum likelihood assuming that the data are drawn independently from the model. The solution is obtained by fitting the model for each class separately using the corresponding labelled data and then setting the class priors $p(\mathcal{C}_k)$ equal to the fraction of training data points in each class. The probability that a vector $\mathbf{x}$ belongs to class $\mathcal{C}_k$ is then given by Bayes' theorem in the form

Exercise 11.15

$$p(\mathcal{C}_k|\mathbf{x}) = \frac{p(\mathbf{x}|\mathcal{C}_k)p(\mathcal{C}_k)}{p(\mathbf{x})} \tag{11.38}$$

where $p(\mathbf{x}|\mathcal{C}_k)$ is given by (11.37), and $p(\mathbf{x})$ can be evaluated using

$$p(\mathbf{x}) = \sum_{k=1}^{K} p(\mathbf{x}|\mathcal{C}_k)p(\mathcal{C}_k). \tag{11.39}$$

The naive Bayes model is illustrated for a two-dimensional data space in Figure 11.23 in which $\mathbf{x} = (x_1, x_2)$. Here we assume that the conditional densities

Telegram: @uni_k

www.konkur.in

$p(\mathbf{x}|\mathcal{C}_k)$ for each of the two classes are axis-aligned Gaussians, and hence that they each factorize with respect to x_1 and x_2 so that

$$p(\mathbf{x}|\mathcal{C}_k) = p(x_1|\mathcal{C}_k)p(x_2|\mathcal{C}_k). \tag{11.40}$$

However, the marginal density $p(\mathbf{x})$ given by

$$p(\mathbf{x}) = \sum_{k=1}^{K} p(\mathbf{x}|\mathcal{C}_k)p(\mathcal{C}_k) \tag{11.41}$$

is now a mixture of Gaussians and does not factorize with respect to x_1 and x_2. We have already encountered a simple application of the naive Bayes model in the context of fusing data from different sources, such as blood tests and skin images for medical diagnosis.

Section 5.2.4

The naive Bayes assumption is helpful when the dimensionality D of the input space is high, making density estimation in the full D-dimensional space more challenging. It is also useful if the input vector contains both discrete and continuous variables, since each can be represented separately using appropriate models (e.g., Bernoulli distributions for binary observations or Gaussians for real-valued variables). The conditional independence assumption of this model is clearly a strong one that may lead to rather poor representations of the class-conditional densities. Nevertheless, even if this assumption is not precisely satisfied, the model may still give good classification performance in practice because the decision boundaries can be insensitive to some of the details in the class-conditional densities, as illustrated in Figure 5.8.

11.2.5 Generative models

Many applications of machine learning can be viewed as examples of *inverse problems* in which there is an underlying, often physical, process that generates data, and the goal is to learn now to invert this process. For example, an image of an object can be viewed as the output of a generative process in which the type of object is selected from some distribution of possible object classes. The position and orientation of the object are also chosen from some prior distributions, and then the resulting image is created. Given a large data set of images labelled with the type, position, and scale of the objects they contain, the goal is to train a machine learning model that can take new, unlabelled images and detect the presence of an object including its location within the image and its size. The machine learning solution therefore represents the inverse of the process that generated the data.

One approach would be to train a deep neural network, such as a convolutional network, to take an image as input and to generate outputs that describe the object's type, position, and scale. This approach therefore tries to solve the inverse problem directly and is an example of a *discriminative model*. It can achieve high accuracy provided ample examples of labelled images are available. In practice, unlabelled images are often plentiful, and much of the effort in obtaining a training set goes into proving the labels, which may be done by hand. Our simple discriminative model cannot directly make use of unlabelled images during training.

Telegram: @uni_k

www.konkur.in

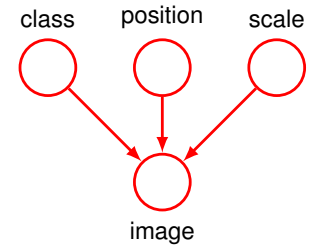


Figure 11.24 A graphical model representing the process by which images of objects are created. The identity of an object (a discrete variable) and the position and orientation of that object (continuous variables) have independent prior probabilities. The image (an array of pixel intensities) has a probability distribution that is dependent on the identity of the object as well as on its position and orientation.

An alternative approach is to model the generative process and then subsequently to invert it computationally. In our image example, if we assume that the object's class, position, and scale are all chosen independently, then we can represent the generative process using a directed graphical model as shown in Figure 11.24. Note that the directions of the arrows correspond to the sequence of generative steps, and so the model represents the *causal* process (Pearl, 1988) by which the observed data is generated. This is an example of a *generative model* because once it is trained, it can be used to generate synthetic images by first selecting values for object's class, position, and scale from the learned prior distributions and then subsequently sampling an image from the learned conditional distribution. We will later see how diffusion models and other generative models can synthesize impressive high-resolution images based on a textual description of the desired content and style of the image.

Chapter 20

The graph in Figure 11.24 assumes that, when no image is observed, the class, position, and scale variables are independent. This follows because every path between any two of these variables is head-to-head with respect to the image variable, which is unobserved. However, when we observe an image, those paths become unblocked, and the class, position, and scale variables are no longer independent. Intuitively this is reasonable because being told the identity of the object within the image provides us with very relevant information to assist us with determining its location.

The hidden variables in a probabilistic model need not, however, have any explicit physical interpretation but may be introduced simply to allow a more complex joint distribution to be constructed from simpler components. For example, models such as normalizing flows, variational autoencoders, and diffusion models all use deep neural networks to create complex distributions in the data space by transforming hidden variables having a simple Gaussian distribution.

11.2.6 Markov blanket

A conditional independence property that is helpful when discussing more complex directed graphs is called the *Markov blanket* or *Markov boundary*. Consider a joint distribution $p(\mathbf{x}_1, \ldots, \mathbf{x}_D)$ represented by a directed graph having D nodes, and consider the conditional distribution of a particular node with variables $\mathbf{x}_i$ conditioned on all the remaining variables $\mathbf{x}_{j \neq i}$. Using the factorization property (11.6),

Telegram: @uni_k

www.konkur.in

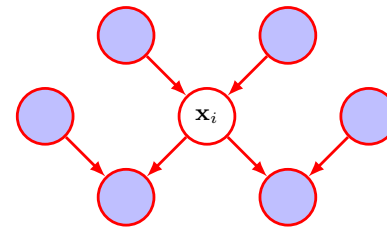


Figure 11.25 The Markov blanket of a node $\mathbf{x}_i$ comprises the set of parents, children, and co-parents of the node. It has the property that the conditional distribution of $\mathbf{x}_i$, conditioned on all the remaining variables in the graph, is dependent only on the variables in the Markov blanket.

we can express this conditional distribution in the form

$$p(\mathbf{x}_i|\mathbf{x}_{\{j\neq i\}}) = \frac{p(\mathbf{x}_1,\ldots,\mathbf{x}_D)}{\int p(\mathbf{x}_1,\ldots,\mathbf{x}_D)\,\mathrm{d}\mathbf{x}_i} = \frac{\prod_k p(\mathbf{x}_k|\mathrm{pa}(k))}{\int \prod_k p(\mathbf{x}_k|\mathrm{pa}(k))\,\mathrm{d}\mathbf{x}_i}$$

in which the integral is replaced by a summation for discrete variables. We now observe that any factor $p(\mathbf{x}_k|\mathrm{pa}(k))$ that does not have any functional dependence on $\mathbf{x}_i$ can be taken outside the integral over $\mathbf{x}_i$ and will therefore cancel between numerator and denominator. The only factors that remain will be the conditional distribution $p(\mathbf{x}_i|\mathrm{pa}(i))$ for node $\mathbf{x}_i$ itself, together with the conditional distributions for any nodes $\mathbf{x}_k$ such that node $\mathbf{x}_i$ is in the conditioning set of $p(\mathbf{x}_k|\mathrm{pa}(k))$, in other words for which $\mathbf{x}_i$ is a parent of $\mathbf{x}_k$. The conditional $p(\mathbf{x}_i|\mathrm{pa}(i))$ will depend on the parents of node $\mathbf{x}_i$, whereas the conditionals $p(\mathbf{x}_k|\mathrm{pa}(k))$ will depend on the children of $\mathbf{x}_i$ as well as on the *co-parents*, in other words variables corresponding to parents of node $\mathbf{x}_k$ other than node $\mathbf{x}_i$. The set of nodes comprising the parents, the children, and the co-parents is called the *Markov blanket* and is illustrated in Figure 11.25.

We can think of the Markov blanket of a node $\mathbf{x}_i$ as being the minimal set of nodes that isolates $\mathbf{x}_i$ from the rest of the graph. Note that it is not sufficient to include only the parents and children of node $\mathbf{x}_i$ because explaining away means that observations of the child nodes will not block paths to the co-parents. We must therefore observe the co-parent nodes as well.

11.2.7 Graphs as filters

We have seen that a particular directed graph represents a specific decomposition of a joint probability distribution into a product of conditional probabilities, and it also expresses a set of conditional independence statements obtained through the d-separation criterion. The d-separation theorem is really an expression of the equivalence of these two properties. To make this clear, it is helpful to think of a directed graph as a filter. Suppose we consider a particular joint probability distribution $p(\mathbf{x})$ over the variables $\mathbf{x}$ corresponding to the (unobserved) nodes of the graph. The filter will allow this distribution to pass through if, and only if, it can be expressed in

Telegram: @uni_k

www.konkur.in

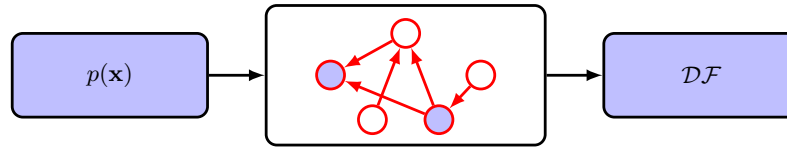


Figure 11.26 We can view a graphical model (in this case a directed graph) as a filter in which a probability distribution $p(\mathbf{x})$ is allowed through the filter if, and only if, it satisfies the directed factorization property (11.6). The set of all possible probability distributions $p(\mathbf{x})$ that pass through the filter is denoted $\mathcal{DF}$. We can alternatively use the graph to filter distributions according to whether they respect all the conditional independence properties implied by the d-separation properties of the graph. The d-separation theorem says the same set of distributions $\mathcal{DF}$ will be allowed through this second kind of filter.

terms of the factorization (11.6) implied by the graph. If we present to the filter the set of all possible distributions $p(\mathbf{x})$ over the set of variables $\mathbf{x}$, then the subset of distributions that are passed by the filter is denoted $\mathcal{DF}$, for *directed factorization*. This is illustrated in Figure 11.26.

Alternatively, we can use the graph as a different kind of filter by first listing all the conditional independence properties obtained by applying the d-separation criterion to the graph and then allowing a distribution to pass only if it satisfies all of these properties. If we present all possible distributions $p(\mathbf{x})$ to this second kind of filter, then the d-separation theorem tells us that the set of distributions that will be allowed through is precisely the set $\mathcal{DF}$.

It should be emphasized that the conditional independence properties obtained from d-separation apply to any probabilistic model described by that particular directed graph. This will be true, for instance, whether the variables are discrete or continuous or a combination of these. Again, we see that a particular graph describes a whole family of probability distributions.

At one extreme, we have a fully connected graph that exhibits no conditional independence properties at all and which can represent any possible joint probability distribution over the given variables. The set $\mathcal{DF}$ will contain all possible distributions $p(\mathbf{x})$. At the other extreme, we have a fully disconnected graph, i.e., one having no links at all. This corresponds to joint distributions that factorize into the product of the marginal distributions over the variables comprising the nodes of the graph. Note that for any given graph, the set of distributions $\mathcal{DF}$ will include any distributions that have additional independence properties beyond those described by the graph. For instance, a fully factorized distribution will always be passed through the filter implied by any graph over the corresponding set of variables.

11.3. Sequence Models

There are many important applications of machine learning in which the data consists of a *sequence* of values. For example, text comprises a sequence of words, whereas a protein comprises a sequence of amino acids. Many sequences are ordered by

Telegram: @uni_k

www.konkur.in

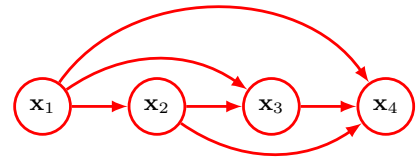


Figure 11.27 An illustration of a general autoregressive model of the form (11.42) with four nodes.

time, such as the audio signals from a microphone or daily rainfall measurements at a particular location. Sometimes the terminology of 'time' as well as 'past' and 'future' are used when referring to other types of sequential data, not just temporal sequences. Applications involving sequences include speech recognition, automatic translation between languages, detecting genes in DNA, synthesizing music, writing computer code, holding a conversation with a modern search engine, and many others.

We will denote a data sequence by $\mathbf{x}_1, \ldots, \mathbf{x}_N$ where each element $\mathbf{x}_n$ of the sequence comprises a vector of values. Note that we might have several such sequences drawn independently from the same distribution, in which case the joint distribution over all the sequences factorizes into the product of the distributions over each sequence individually. From now on, we focus on modelling just one of those sequences.

We have already seen in (11.4) that by repeated application of the product rule of probability, a general distribution over N variables can be written as the product of conditional distributions, and that the form of this decomposition depends on a specific of ordering for the variables. For vector-valued variables, and if we chose an ordering that corresponds to the order of the variables in the sequence, then we can write

$$p(\mathbf{x}_1, \ldots, \mathbf{x}_N) = \prod_{n=1}^{N} p(\mathbf{x}_n | \mathbf{x}_1, \ldots, \mathbf{x}_{n-1}). \tag{11.42}$$

This corresponds to a directed graph in which each node receives a link from every previous node in the sequence, as illustrated using four variables in Figure 11.27. This is known as an *autoregressive* model.

This representation has complete generality and therefore from a modelling perspective adds no value since it encodes no assumptions. We can constrain the space of models by introducing conditional independence properties by removing links from the graph, or equivalently by removing variables from the conditioning set of the factors on the right-hand-side of (11.42).

The strongest assumption would be to remove all conditioning variables, giving a joint distribution of the form

$$p(\mathbf{x}_1, \ldots, \mathbf{x}_N) = \prod_{n=1}^{N} p(\mathbf{x}_n), \tag{11.43}$$

which treats the variables as independent and therefore completely ignores the ordering information. This corresponds to a probabilistic graphical model without links, as shown in Figure 11.28.

Telegram: @uni_k

www.konkur.in

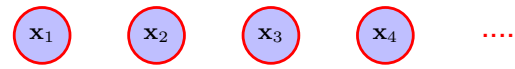


Figure 11.28 The simplest approach to modelling a sequence of observations is to treat them as independent, corresponding to a probabilistic graphical model without links.

Interesting models that capture sequential properties while introducing modelling assumptions lie between these two extremes. One strong assumption would be to assume that each conditional distribution depends only on the immediately preceding variable in the sequence, giving a joint distribution of the form

$$p(\mathbf{x}_1, \ldots, \mathbf{x}_N) = p(\mathbf{x}_1) \prod_{n=2}^{N} p(\mathbf{x}_n|\mathbf{x}_{n-1}). \tag{11.44}$$

Note that the first variable in the sequence is treated slightly differently since it has no conditioning variable. The functional form (11.44) is known as a *Markov model*, or *Markov chain*, and is represented by a graph consisting of a simple chain of nodes, as seen in Figure 11.29. Using d-separation, we see that the conditional distribution for observation $\mathbf{x}_n$, given all of the observations up to time n, is given by

Section 11.2.3

$$p(\mathbf{x}_n|\mathbf{x}_1, \ldots, \mathbf{x}_{n-1}) = p(\mathbf{x}_n|\mathbf{x}_{n-1}), \tag{11.45}$$

which is easily verified by direct evaluation starting from (11.44) and using the product rule of probability. Thus, if we use such a model to predict the next observation in a sequence, the distribution of predictions will depend only on the value of the immediately preceding observation and will be independent of all earlier observations.

Exercise 11.16

More specifically, (11.44) is known as a first-order Markov model because only one conditioning variable appears in each conditional distribution. We can extend the model by allowing each conditional distribution to depend on the two preceding variables, giving a second-order Markov model of the form

$$p(\mathbf{x}_1, \ldots, \mathbf{x}_N) = p(\mathbf{x}_1)p(\mathbf{x}_2|\mathbf{x}_1) \prod_{n=3}^{N} p(\mathbf{x}_n|\mathbf{x}_{n-1}, \mathbf{x}_{n-2}). \tag{11.46}$$

Note that the first two variables are treated differently as they have fewer than two conditioning variables. This model is shown as a directed graph in Figure 11.30.

Exercise 11.17

By using d-separation (or by direct evaluation using the rules of probability), we see that in the second-order Markov model, the conditional distribution of $\mathbf{x}_n$ given all previous observations $\mathbf{x}_1, \ldots, \mathbf{x}_{n-1}$ is independent of the observations $\mathbf{x}_1, \ldots \mathbf{x}_{n-3}$. We can similarly consider extensions to an M^{th} order Markov chain in

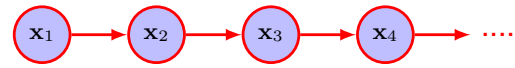


Figure 11.29 A first-order Markov chain of observations in which the distribution of a particular observation $\mathbf{x}_n$ is conditioned on the value of the previous observation $\mathbf{x}_{n-1}$.

Telegram: @uni_k

www.konkur.in

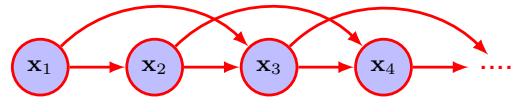


Figure 11.30 A second-order Markov chain in which the conditional distribution of a particular observation $\mathbf{x}_n$ depends on the values of the two previous observations $\mathbf{x}_{n-1}$ and $\mathbf{x}_{n-2}$.

which the conditional distribution for a particular variable depends on the previous M variables. However, we have paid a price for this increased flexibility because the number of parameters in the model is now much larger. Suppose the observations are discrete variables having K states. Then the conditional distribution $p(\mathbf{x}_n|\mathbf{x}_{n-1})$ in a first-order Markov chain will be specified by a set of $K-1$ parameters for each of the K states of $\mathbf{x}_{n-1}$ giving a total of $K(K-1)$ parameters. Now suppose we extend the model to an M^{th} order Markov chain, so that the joint distribution is built up from conditionals $p(\mathbf{x}_n|\mathbf{x}_{n-M},\ldots,\mathbf{x}_{n-1})$. If the variables are discrete and if the conditional distributions are represented by general conditional probability tables, then such a model will have $K^{M-1}(K-1)$ parameters. Thus, the number of parameters grows exponentially with M, which will generally render this approach impractical for larger values of M.

11.3.1 Hidden variables

Suppose we wish to build a model for sequences that is not limited by the Markov assumption to any order and yet can be specified using a limited number of free parameters. We can achieve this by introducing additional latent variables, thus permitting a rich class of models to be constructed out of simple components. For each observation $\mathbf{x}_n$, we introduce a corresponding latent variable $\mathbf{z}_n$ (which may be of different type or dimensionality to the observed variable). We now assume that it is the latent variables that form a Markov chain, giving rise to the graphical structure known as a *state-space model*, which is shown in Figure 11.31. It satisfies the key conditional independence property that $\mathbf{z}_{n-1}$ and $\mathbf{z}_{n+1}$ are independent given $\mathbf{z}_n$, so that

$$\mathbf{z}_{n+1} \perp\!\!\!\perp \mathbf{z}_{n-1} \mid \mathbf{z}_n. \tag{11.47}$$

The joint distribution for this model is given by

$$p(\mathbf{x}_1,\ldots,\mathbf{x}_N,\mathbf{z}_1,\ldots,\mathbf{z}_N) = p(\mathbf{z}_1)\left[\prod_{n=2}^{N} p(\mathbf{z}_n|\mathbf{z}_{n-1})\right]\prod_{n=1}^{N} p(\mathbf{x}_n|\mathbf{z}_n). \tag{11.48}$$

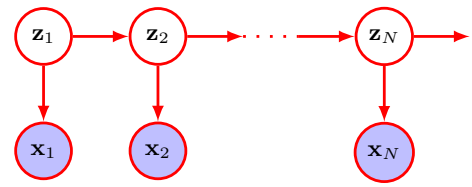


Figure 11.31 A state-space model expresses the joint probability distribution over a sequence of observed states $\mathbf{x}_1,\ldots,\mathbf{x}_N$ in terms of a Markov chain of hidden states $\mathbf{z}_1,\ldots,\mathbf{z}_N$ in the form (11.48).

Telegram: @uni_k

www.konkur.in

Using the d-separation criterion, we see that in the state-space model there is always a path connecting any two observed variables $\mathbf{x}_n$ and $\mathbf{x}_m$ via the latent variables and that this path is never blocked. Thus, the predictive distribution $p(\mathbf{x}_{n+1}|\mathbf{x}_1, \ldots, \mathbf{x}_n)$ for observation $\mathbf{x}_{n+1}$ given all previous observations does not exhibit any conditional independence properties, and so our predictions for $\mathbf{x}_{n+1}$ depend on all previous observations. The observed variables, therefore, do not satisfy the Markov property at any order.

There are two important models for sequential data that are described by this graph. If the latent variables are discrete, then we obtain a *hidden Markov model* (Elliott, Aggoun, and Moore, 1995). Note that the observed variables in a hidden Markov model may be discrete or continuous, and a variety of different conditional distributions can be used to model them. If both the latent and the observed variables are Gaussian (with a linear-Gaussian dependence of the conditional distributions on their parents), then we obtain a *linear dynamical system*, also known as a *Kalman filter* (Zarchan and Musoff, 2005). Both hidden Markov models and Kalman filters are discussed at length, along with algorithms for training them, in Bishop (2006). Such models can be made considerably more flexible by replacing the simple discrete probability tables, or linear-Gaussian distributions, used to define $p(\mathbf{x}_n|\mathbf{z}_n)$ with deep neural networks.

Exercises

11.1 ($\star$) By marginalizing out the variables in order, show that the representation (11.6) for the joint distribution of a directed graph is correctly normalized, provided each of the conditional distributions is normalized.

11.2 ($\star$) Show that the property of there being no directed cycles in a directed graph follows from the statement that there exists an ordered numbering of the nodes such that for each node there are no links going to a lower-numbered node.

11.3 ($\star\star$) Consider three binary variables $a, b, c \in \{0, 1\}$ having the joint distribution given in Table 11.1. Show by direct evaluation that this distribution has the property that a and b are marginally dependent, so that $p(a, b) \neq p(a)p(b)$, but that they become independent when conditioned on c, so that $p(a, b|c) = p(a|c)p(b|c)$ for both $c = 0$ and $c = 1$.

Table 11.1 The joint distribution over three binary variables.

a	b	c	$p(a, b, c)$
0	0	0	0.192
0	0	1	0.144
0	1	0	0.048
0	1	1	0.216
1	0	0	0.192
1	0	1	0.064
1	1	0	0.048
1	1	1	0.096

Telegram: @uni_k

www.konkur.in

11.4 (⋆⋆) Evaluate the distributions $p(a)$, $p(b|c)$, and $p(c|a)$ corresponding to the joint distribution given in Table 11.1. Hence, show by direct evaluation that $p(a, b, c) = p(a)p(c|a)p(b|c)$. Draw the corresponding directed graph.

11.5 (⋆) For the model shown in Figure 11.6, we have seen that the number of parameters required to specify the conditional distribution $p(y|x_1, \ldots, x_M)$, where $x_i \in \{0, 1\}$, could be reduced from 2^M to $M + 1$ by making use of the logistic sigmoid representation (11.8). An alternative representation (Pearl, 1988) is given by

$$p(y = 1|x_1, \ldots, x_M) = 1 - (1 - \mu_0) \prod_{i=1}^{M} (1 - \mu_i)^{x_i} \tag{11.49}$$

where the parameters μ_i represent the probabilities $p(x_i = 1)$ and μ_0 is an additional parameter satisfying $0 \leqslant \mu_0 \leqslant 1$. The conditional distribution (11.49) is known as the *noisy-OR*. Show that this can be interpreted as a 'soft' (probabilistic) form of the logical OR function (i.e., the function that gives $y = 1$ whenever at least one of the $x_i = 1$). Discuss the interpretation of μ_0.

11.6 (⋆⋆) Starting from the definition (11.9) for the conditional distributions, derive the recursion relation (11.12) for the mean of the joint distribution for a linear-Gaussian model.

11.7 (⋆⋆) Starting from the definition (11.9) for the conditional distributions, derive the recursion relation (11.13) for the covariance matrix of the joint distribution for a linear-Gaussian model.

11.8 (⋆⋆) Show that the number of parameters in the covariance matrix of a fully connected linear-Gaussian graphical model over D variables defined by (11.9) is $D(D + 1)/2$.

11.9 (⋆⋆) Using the recursion relations (11.12) and (11.13), show that the mean and covariance of the joint distribution for the graph shown in Figure 11.7 are given by (11.14) and (11.15), respectively.

11.10 (⋆) Verify that the joint distribution over a set of vector-valued variables defined by a linear-Gaussian model in which each node corresponds to a distribution of the form (11.16) is itself a Gaussian.

11.11 (⋆) Show that $a \perp\!\!\!\perp b, c \mid d$ implies $a \perp\!\!\!\perp b \mid d$.

11.12 (⋆) Using the d-separation criterion, show that the conditional distribution for a node $\mathbf{x}$ in a directed graph, conditioned on all the nodes in the Markov blanket, is independent of the remaining variables in the graph.

11.13 (⋆) Consider the directed graph shown in Figure 11.32 in which none of the variables is observed. Show that $a \perp\!\!\!\perp b \mid \emptyset$. Suppose we now observe the variable d. Show that in general $a \not\perp\!\!\!\perp b \mid d$.

Telegram: @uni_k

www.konkur.in

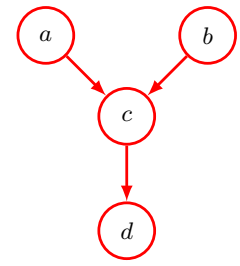


Figure 11.32 Example of a graphical model used to explore the conditional independence properties of the head-to-head path a–c–b when a descendant of c, namely the node d, is observed.

11.14 ($\star\star$) Consider the example of the car fuel system shown in Figure 11.20, and suppose that instead of observing the state of the fuel gauge G directly, the gauge is seen by the driver D, who reports to us the reading on the gauge. This report says that the gauge shows either that the tank is full $D = 1$ or that it is empty $D = 0$. Our driver is a bit unreliable, as expressed through the following probabilities:

$$p(D = 1|G = 1) = 0.9 \tag{11.50}$$

$$p(D = 0|G = 0) = 0.9. \tag{11.51}$$

Suppose that the driver tells us that the fuel gauge shows empty, in other words that we observe $D = 0$. Evaluate the probability that the tank is empty given only this observation. Similarly, evaluate the corresponding probability given also the observation that the battery is flat, and note that this second probability is lower. Discuss the intuition behind this result, and relate the result to Figure 11.32.

11.15 ($\star\star$) Suppose we train a naive Bayes model, with the assumption (11.37), using maximum likelihood. Assume that each of the class-conditional densities $p(\mathbf{x}^{(l)}|\mathcal{C}_k)$ is governed by its own independent parameters $\mathbf{w}^{(l)}$. Show that the maximum likelihood solution involves fitting each of the class-conditional densities using the corresponding observed data vectors $\mathbf{x}_1^{(l)}, \ldots, \mathbf{x}_N^{(l)}$ by maximizing the likelihood with respect to the corresponding class label data, and then setting the class priors $p(\mathcal{C}_k)$ to the fraction of training data points in each class.

11.16 ($\star\star$) Consider the joint probability distribution (11.44) corresponding to the directed graph of Figure 11.29. Using the sum and product rules of probability, verify that this joint distribution satisfies the conditional independence property (11.45) for $n = 2, \ldots, N$. Similarly, show that the second-order Markov model described by the joint distribution (11.46) satisfies the conditional independence property

$$p(\mathbf{x}_n|\mathbf{x}_1, \ldots, \mathbf{x}_{n-1}) = p(\mathbf{x}_n|\mathbf{x}_{n-1}, \mathbf{x}_{n-2}) \tag{11.52}$$

for $n = 3, \ldots, N$.

11.17 ($\star$) Use d-separation, as discussed in Section 11.2, to verify that the Markov model shown in Figure 11.29 having N nodes in total satisfies the conditional independence properties (11.45) for $n = 2, \ldots, N$. Similarly, show that a model described by the graph in Figure 11.30 in which there are N nodes in total satisfies the conditional independence properties (11.52) for $n = 3, \ldots, N$.

Telegram: @uni_k

www.konkur.in

11.18 ($\star$) Consider a second-order Markov process described by the graph in Figure 11.30. By combining adjacent pairs of variables, show that this can be expressed as a first-order Markov process over the new variables.

11.19 ($\star$) By using d-separation, show that the distribution $p(\mathbf{x}_1, \ldots, \mathbf{x}_N)$ of the observed data for the state-space model represented by the directed graph in Figure 11.31 does not satisfy any conditional independence properties and hence does not exhibit the Markov property at any finite order.

Telegram: @uni_k

www.konkur.in



12 Transformers

Transformers represent one of the most important developments in deep learning. They are based on a processing concept called *attention*, which allows a network to give different weights to different inputs, with weighting coefficients that themselves depend on the input values, thereby capturing powerful inductive biases related to sequential and other forms of data.

These models are known as transformers because they transform a set of vectors in some representation space into a corresponding set of vectors, having the same dimensionality, in some new space. The goal of the transformation is that the new space will have a richer internal representation that is better suited to solving downstream tasks. Inputs to a transformer can take the form of unstructured sets of vectors, ordered sequences, or more general representations, giving transformers broad applicability.

Transformers were originally introduced in the context of natural language pro-

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024
C. M. Bishop, H. Bishop, *Deep Learning*, https://doi.org/10.1007/978-3-031-45468-4_12

Telegram: @uni_k

www.konkur.in

cessing, or NLP (where a 'natural' language is one such as English or Mandarin) and have greatly surpassed the previous state-of-the-art approaches based on recurrent neural networks (RNNs). Transformers have subsequently been found to achieve excellent results in many other domains. For example, vision transformers often outperform CNNs in image processing tasks, whereas multimodal transformers that combine multiple types of data, such as text, images, audio, and video, are amongst the most powerful deep learning models.

One major advantage of transformers is that transfer learning is very effective, so that a transformer model can be trained on a large body of data and then the trained model can be applied to many downstream tasks using some form of fine-tuning. A large-scale model that can subsequently be adapted to solve multiple different tasks is known as a *foundation model*. Furthermore, transformers can be trained in a self-supervised way using unlabelled data, which is especially effective with language models since transformers can exploit vast quantities of text available from the internet and other sources. The *scaling hypothesis* asserts that simply by increasing the scale of the model, as measured by the number of learnable parameters, and training on a commensurately large data set, significant improvements in performance can be achieved, even with no architectural changes. Moreover, the transformer is especially well suited to massively parallel processing hardware such as *graphical processing units*, or GPUs, allowing exceptionally large neural network language models having of the order of a trillion (10^{12}) parameters to be trained in reasonable time. Such models have extraordinary capabilities and show clear indications of emergent properties that have been described as the early signs of artificial general intelligence (Bubeck *et al.*, 2023).

The architecture of a transformer can seem complex, or even daunting, to a newcomer as it involves multiple different components working together, in which the various design choices can seem arbitrary. In this chapter we therefore aim to give a comprehensive step-by-step introduction to all the key ideas behind transformers and to provide clear intuition to motivate the design of the various elements. We first describe the transformer architecture and then focus on natural language processing, before exploring other application domains.

12.1. Attention

Section 12.2.5

The fundamental concept that underpins a transformer is *attention*. This was originally developed as an enhancement to RNNs for machine translation (Bahdanau, Cho, and Bengio, 2014). However, Vaswani *et al.* (2017) later showed that significantly improved performance could be obtained by eliminating the recurrence structure and instead focusing exclusively on the attention mechanism. Today, transformers based on attention have completely superseded RNNs in almost all applications.

We will motivate the use of attention using natural language as an example,

Telegram: @uni_k

www.konkur.in

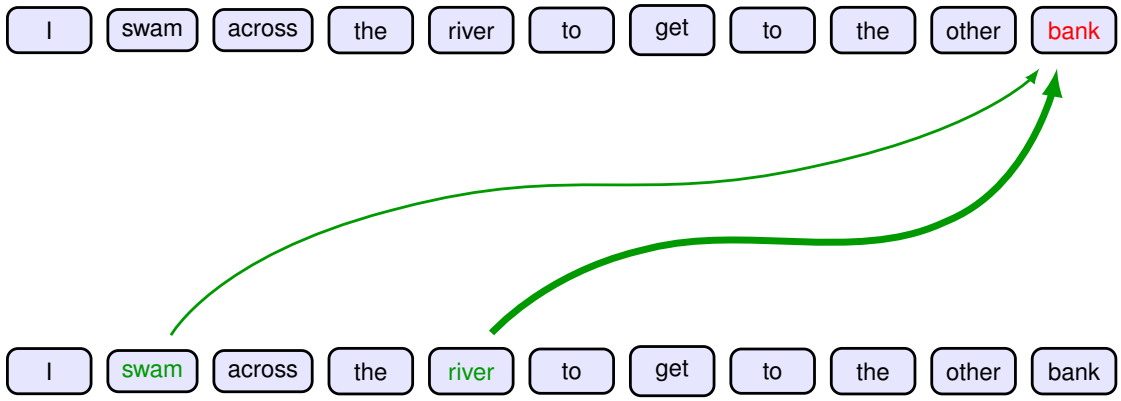


Figure 12.1 Schematic illustration of attention in which the interpretation of the word 'bank' is influenced by the words 'river' and 'swam', with the thickness of each line being indicative of the strength of its influence.

although it has much broader applicability. Consider the following two sentences:

I swam across the river to get to the other bank.

I walked across the road to get cash from the bank.

Here the word 'bank' has different meanings in the two sentences. However, this can be detected only by looking at the context provided by other words in the sequence. We also see that some words are more important than others in determining the interpretation of 'bank'. In the first sentence, the words 'swam' and 'river' most strongly indicate that 'bank' refers to the side of a river, whereas in the second sentence, the word 'cash' is a strong indicator that 'bank' refers to a financial institution. We see that to determine the appropriate interpretation of 'bank', a neural network processing such a sentence should *attend* to, in other words rely more heavily on, specific words from the rest of the sequence. This concept of attention is illustrated in Figure 12.1.

Moreover, we also see that the particular locations that should receive more attention depend on the input sequence itself: in the first sentence it is the second and fifth words that are important whereas in the second sentence it is the eighth word. In a standard neural network, different inputs will influence the output to different extents according to the values of the weights that multiply those inputs. Once the network is trained, however, those weights, and their associated inputs, are fixed. By contrast, attention uses weighting factors whose values depend on the specific input data. Figure 12.2 shows the attention weights from a section of a transformer network trained on natural language.

When we discuss natural language processing, we will see how word embedding can be used to map words into vectors in an embedding space. These vectors can then be used as inputs for subsequent neural network processing. These embeddings capture elementary semantic properties, for example by mapping words with similar meanings to nearby locations in the embedding space. One characteristic of such embeddings is that a given word always maps to the same embedding vector.

Telegram: @uni_k

www.konkur.in

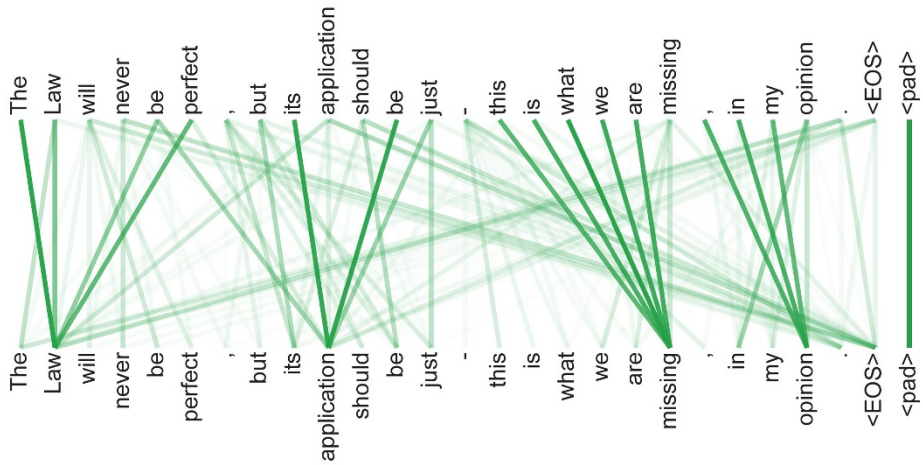


Figure 12.2 An example of learned attention weights. [From Vaswani *et al.* (2017) with permission.]

A transformer can be viewed as a richer form of embedding in which a given vector is mapped to a location that depends on the other vectors in the sequence. Thus, the vector representing 'bank' in our example above could map to different places in a new embedding space for the two different sentences. For example, in the first sentence the transformed representation might put 'bank' close to 'water' in the embedding space, whereas in the second sentence the transformed representation might put it close to 'money'.

As an example of attention, consider the modelling of proteins. We can view a protein as a one-dimensional sequence of molecular units called amino acids. A protein can comprise potentially hundreds or thousands of such units, each of which is given by one of 22 possibilities. In a living cell, a protein folds up into a three-dimensional structure in which amino acids that are widely separated in the one-dimensional sequence can become physically close in three-dimensional space and thereby interact. Transformer models allows these distant amino acids to 'attend' to each other thereby greatly improving the accuracy with which their 3-dimensional structure can be modelled (Vig *et al.*, 2020).

Figure 1.2

12.1.1 Transformer processing

The input data to a transformer is a set of vectors $\{\mathbf{x}_n\}$ of dimensionality D, where $n = 1, \ldots, N$. We refer to these data vectors as *tokens*, where a token might, for example, correspond to a word within a sentence, a patch within an image, or an amino acid within a protein. The elements x_{ni} of the tokens are called *features*. Later we will see how to construct these token vectors for natural language data and for images. A powerful property of transformers is that we do not have to design a new neural network architecture to handle a mix of different data types but instead can simply combine the data variables into a joint set of tokens.

Before we can gain a clear understanding of the operation of a transformer, it

Telegram: @uni_k

www.konkur.in

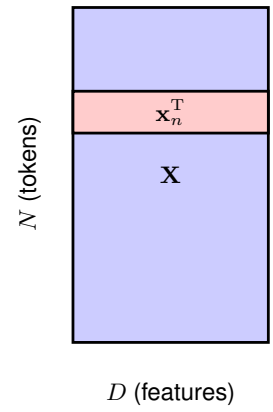


Figure 12.3 The structure of the data matrix $\mathbf{X}$, of dimension $N \times D$, in which row n represents the transposed data vector $\mathbf{x}_n^{\mathrm{T}}$.

is important to be precise about notation. We will follow the standard convention and combine the data vectors into a matrix $\mathbf{X}$ of dimensions $N \times D$ in which the nth row comprises the token vector $\mathbf{x}_n^{\mathrm{T}}$, and where $n = 1, \ldots, N$ labels the rows, as illustrated in Figure 12.3. Note that this matrix represents one set of input tokens, and that for most applications, we will require a data set containing many sets of tokens, such as independent passages of text where each word is represented as one token. The fundamental building block of a transformer is a function that takes a data matrix as input and creates a transformed matrix $\widetilde{\mathbf{X}}$ of the same dimensionality as the output. We can write this function in the form

$$\widetilde{\mathbf{X}} = \mathrm{TransformerLayer}\left[\mathbf{X}\right]. \tag{12.1}$$

We can then apply multiple transformer layers in succession to construct deep networks capable of learning rich internal representations. Each transformer layer contains its own weights and biases, which can be learned using gradient descent using an appropriate cost function, as we will discuss in detail later in the chapter.

Section 12.3

A single transformer layer itself comprises two stages. The first stage, which implements the attention mechanism, mixes together the corresponding features from different token vectors across the columns of the data matrix, whereas the second stage then acts on each row independently and transforms the features within each token vector. We start by looking at the attention mechanism.

12.1.2 Attention coefficients

Suppose that we have a set of input tokens $\mathbf{x}_1, \ldots, \mathbf{x}_N$ in an embedding space and we want to map this to another set $\mathbf{y}_1, \ldots, \mathbf{y}_N$ having the same number of tokens but in a new embedding space that captures a richer semantic structure. Consider a particular output vector $\mathbf{y}_n$. The value of $\mathbf{y}_n$ should depend not just on the corresponding input vector $\mathbf{x}_n$ but on all the vectors $\mathbf{x}_1, \ldots, \mathbf{x}_N$ in the set. With attention, this dependence should be stronger for those inputs $\mathbf{x}_m$ that are particularly important for determining the modified representation of $\mathbf{y}_n$. A simple way to achieve this is to define each output vector $\mathbf{y}_n$ to be a linear combination of the input vectors

Telegram: @uni_k

www.konkur.in

$\mathbf{x}_1, \ldots, \mathbf{x}_N$ with weighting coefficients a_{nm}:

$$\mathbf{y}_n = \sum_{m=1}^{N} a_{nm}\mathbf{x}_m \tag{12.2}$$

where a_{nm} are called *attention weights*. The coefficients should be close to zero for input tokens that have little influence on the output $\mathbf{y}_n$ and largest for inputs that have most influence. We therefore constrain the coefficients to be non-negative to avoid situations in which one coefficient can become large and positive while another coefficient compensates by becoming large and negative. We also want to ensure that if an output pays more attention to a particular input, this will be at the expense of paying less attention to the other inputs, and so we constrain the coefficients to sum to unity. Thus, the weighting coefficients must satisfy the following two constraints:

$$a_{nm} \geqslant 0 \tag{12.3}$$

$$\sum_{m=1}^{N} a_{nm} = 1. \tag{12.4}$$

Exercise 12.1

Together these imply that each coefficient lies in the range $0 \leqslant a_{nm} \leqslant 1$ and so the coefficients define a 'partition of unity'. For the special case $a_{mm} = 1$, it follows that $a_{nm} = 0$ for $n \neq m$, and therefore $\mathbf{y}_m = \mathbf{x}_m$ so that the input vector is unchanged by the transformation. More generally, the output $\mathbf{y}_m$ is a blend of the input vectors with some inputs given more weight than others.

Note that we have a different set of coefficients for each output vector $\mathbf{y}_n$, and the constraints (12.3) and (12.4) apply separately for each value of n. These coefficients a_{nm} depend on the input data, and we will shortly see how to calculate them.

12.1.3 Self-attention

The next question is how to determine the coefficients a_{nm}. Before we discuss this in detail, it is useful to first introduce some terminology taken from the field of information retrieval. Consider the problem of choosing which movie to watch in an online movie streaming service. One approach would be to associate each movie with a list of attributes describing things such as the genre (comedy, action, etc.), the names of the leading actors, the length of the movie, and so on. The user could then search through a catalogue to find a movie that matches their preferences. We could automate this by encoding the attributes of each movie in a vector called the *key*. The corresponding movie file itself is called a *value*. Similarly, the user could then provide their own personal vector of values for the desired attributes, which we call the *query*. The movie service could then compare the query vector with all the key vectors to find the best match and send the corresponding movie to the user in the form of the value file. We can think of the user 'attending' to the particular movie whose key most closely matches their query. This would be considered a form of *hard attention* in which a single value vector is returned. For the transformer, we generalize this to *soft attention* in which we use continuous variables to measure

Telegram: @uni_k

www.konkur.in

the degree of match between queries and keys and we then use these variables to weight the influence of the value vectors on the outputs. This will also ensure that the transformer function is differentiable and can therefore be trained by gradient descent.

Following the analogy with information retrieval, we can view each of the input vectors $\mathbf{x}_n$ as a value vector that will be used to create the output tokens. We also use the vector $\mathbf{x}_n$ directly as the key vector for input token n. That would be analogous to using the movie itself to summarize the characteristics of the movie. Finally, we can use $\mathbf{x}_m$ as the query vector for output $\mathbf{y}_m$, which can then be compared to each of the key vectors. To see how much the token represented by $\mathbf{x}_n$ should attend to the token represented by $\mathbf{x}_m$, we need to work out how similar these vectors are. One simple measure of similarity is to take their dot product $\mathbf{x}_n^{\mathrm{T}}\mathbf{x}_m$. To impose the constraints (12.3) and (12.4), we can define the weighting coefficients a_{nm} by using the *softmax* function to transform the dot products:

Section 5.3

$$a_{nm} = \frac{\exp(\mathbf{x}_n^{\mathrm{T}}\mathbf{x}_m)}{\sum_{m'=1}^{N}\exp(\mathbf{x}_n^{\mathrm{T}}\mathbf{x}_{m'})}. \tag{12.5}$$

Note that in this case there is no probabilistic interpretation of the softmax function and it is simply being used to normalize the attention weights appropriately.

So in summary, each input vector $\mathbf{x}_n$ is transformed to a corresponding output vector $\mathbf{y}_n$ by taking a linear combination of input vectors of the form (12.2) in which the weight a_{nm} applied to input vector $\mathbf{x}_m$ is given by the softmax function (12.5) defined in terms of the dot product $\mathbf{x}_n^{\mathrm{T}}\mathbf{x}_m$ between the query $\mathbf{x}_n$ for input n and the key $\mathbf{x}_m$ associated with input m. Note that, if all the input vectors are orthogonal, then each output vector is simply equal to the corresponding input vector so that $\mathbf{y}_m = \mathbf{x}_m$ for $m = 1, \ldots, N$.

Exercise 12.3

We can write (12.2) in matrix notation by using the data matrix $\mathbf{X}$, along with the analogous $N \times D$ output matrix $\mathbf{Y}$, whose rows are given by $\mathbf{y}_m$, so that

$$\mathbf{Y} = \mathrm{Softmax}\left[\mathbf{X}\mathbf{X}^{\mathrm{T}}\right]\mathbf{X} \tag{12.6}$$

where $\mathrm{Softmax}[\mathbf{L}]$ is an operator that takes the exponential of every element of a matrix $\mathbf{L}$ and then normalizes each row independently to sum to one. From now on, we will focus on matrix notation for clarity.

This process is called *self-attention* because we are using the same sequence to determine the queries, keys, and values. We will encounter variants of this attention mechanism later in this chapter. Also, because the measure of similarity between query and key vectors is given by a dot product, this is known as *dot-product self-attention*.

12.1.4 Network parameters

As it stands, the transformation from input vectors $\{\mathbf{x}_n\}$ to output vectors $\{\mathbf{y}_n\}$ is fixed and has no capacity to learn from data because it has no adjustable parameters. Furthermore, each of the feature values within a token vector $\mathbf{x}_n$ plays an equal role in determining the attention coefficients, whereas we would like the network to

Telegram: @uni_k

www.konkur.in

have the flexibility to focus more on some features than others when determining token similarity. We can address both issues if we define modified feature vectors given by a linear transformation of the original vectors in the form

$$\widetilde{\mathbf{X}} = \mathbf{X}\mathbf{U} \tag{12.7}$$

where $\mathbf{U}$ is a $D \times D$ matrix of learnable weight parameters, analogous to a 'layer' in a standard neural network. This gives a modified transformation of the form

$$\mathbf{Y} = \text{Softmax}\left[\mathbf{X}\mathbf{U}\mathbf{U}^{\mathrm{T}}\mathbf{X}^{\mathrm{T}}\right]\mathbf{X}\mathbf{U}. \tag{12.8}$$

Although this has much more flexibility, it has the property that the matrix

$$\mathbf{X}\mathbf{U}\mathbf{U}^{\mathrm{T}}\mathbf{X}^{\mathrm{T}} \tag{12.9}$$

is symmetric, whereas we would like the attention mechanism to support significant asymmetry. For example, we might expect that 'chisel' should be strongly associated with 'tool' since every chisel is a tool, whereas 'tool' should only be weakly associated with 'chisel' because there are many other kinds of tools besides chisels. Although the softmax function means the resulting matrix of attention weights is not itself symmetric, we can create a much more flexible model by allowing the queries and the keys to have independent parameters. Furthermore, the form (12.8) uses the same parameter matrix $\mathbf{U}$ to define both the value vectors and the attention coefficients, which again seems like an undesirable restriction.

We can overcome these limitations by defining separate query, key, and value matrices each having their own independent linear transformations:

$$\mathbf{Q} = \mathbf{X}\mathbf{W}^{(\mathrm{q})} \tag{12.10}$$

$$\mathbf{K} = \mathbf{X}\mathbf{W}^{(\mathrm{k})} \tag{12.11}$$

$$\mathbf{V} = \mathbf{X}\mathbf{W}^{(\mathrm{v})} \tag{12.12}$$

where the weight matrices $\mathbf{W}^{(\mathrm{q})}$, $\mathbf{W}^{(\mathrm{k})}$, and $\mathbf{W}^{(\mathrm{v})}$ represent parameters that will be learned during the training of the final transformer architecture. Here the matrix $\mathbf{W}^{(\mathrm{k})}$ has dimensionality $D \times D_{\mathrm{k}}$ where D_{k} is the length of the key vector. The matrix $\mathbf{W}^{(\mathrm{q})}$ must have the same dimensionality $D \times D_{\mathrm{k}}$ as $\mathbf{W}^{(\mathrm{k})}$ so that we can form dot products between the query and key vectors. A typical choice is $D_{\mathrm{k}} = D$. Similarly, $\mathbf{W}^{(\mathrm{v})}$ is a matrix of size $D \times D_{\mathrm{v}}$, where D_{v} governs the dimensionality of the output vectors. If we set $D_{\mathrm{v}} = D$, so that the output representation has the same dimensionality as the input, this will facilitate the inclusion of residual connections, which we discuss later. Also, multiple transformer layers can be stacked on top of each other if each layer has the same dimensionality. We can then generalize (12.6) to give

Section 12.1.7

$$\mathbf{Y} = \text{Softmax}\left[\mathbf{Q}\mathbf{K}^{\mathrm{T}}\right]\mathbf{V} \tag{12.13}$$

where $\mathbf{Q}\mathbf{K}^{\mathrm{T}}$ has dimension $N \times N$, and the matrix $\mathbf{Y}$ has dimension $N \times D_{\mathrm{v}}$. The calculation of the matrix $\mathbf{Q}\mathbf{K}^{\mathrm{T}}$ is illustrated in Figure 12.4, whereas the evaluation of the matrix $\mathbf{Y}$ is illustrated in Figure 12.5.

Telegram: @uni_k

www.konkur.in

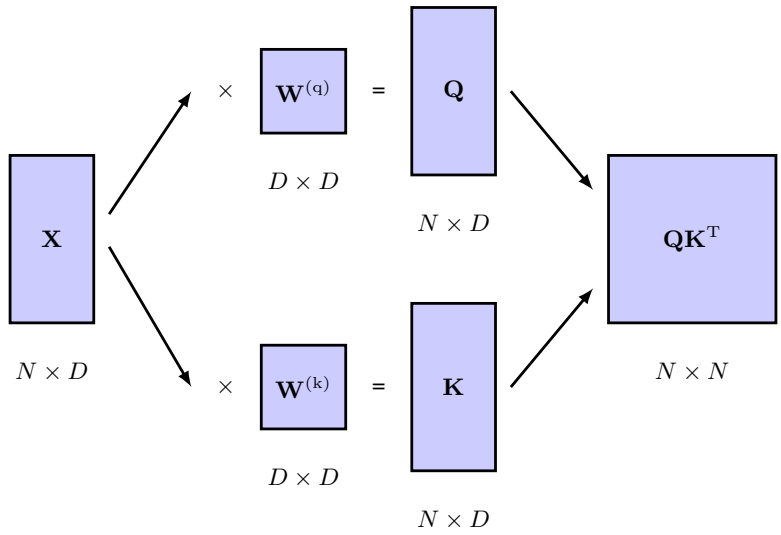


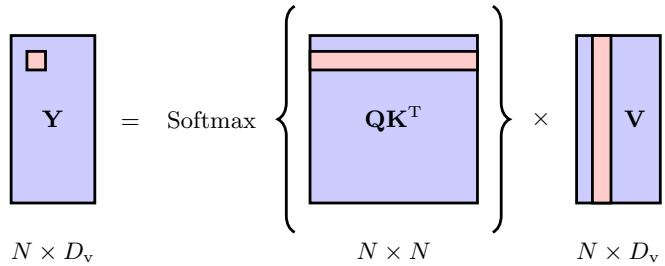
Figure 12.4 Illustration of the evaluation of the matrix $\mathbf{QK}^{\mathrm{T}}$, which determines the attention coefficients in a transformer. The input $\mathbf{X}$ is separately transformed using (12.10) and (12.11) to give the query matrix $\mathbf{Q}$ and key matrix $\mathbf{K}$, respectively, which are then multiplied together.

Section 6.2.1

In practice we can also include bias parameters in these linear transformations. However, the bias parameters can be absorbed into the weight matrices, as we did with standard neural networks, by augmenting the data matrix $\mathbf{X}$ with an additional column of 1's and by augmenting the weight matrices with an additional row of parameters to represent the biases. From now on we will treat the bias parameters as implicit to avoid cluttering the notation.

Compared to a conventional neural network, the signal paths have multiplicative relations between activation values. Whereas standard networks multiply activations by fixed weights, here the activations are multiplied by the data-dependent attention coefficients. This means, for example, that if one of the attention coefficients is close to zero for a particular choice of input vector, the resulting signal path will ignore the corresponding incoming signal, which will therefore have no influence

Figure 12.5 Illustration of the evaluation of the output from an attention layer given the query, key, and value matrices $\mathbf{Q}$, $\mathbf{K}$, and $\mathbf{V}$, respectively. The entry at the position highlighted in the output matrix $\mathbf{Y}$ is obtained from the dot product of the highlighted row and column of the $\mathrm{Softmax}\left[\mathbf{QK}^{\mathrm{T}}\right]$ and $\mathbf{V}$ matrices, respectively.



Telegram: @uni_k

www.konkur.in

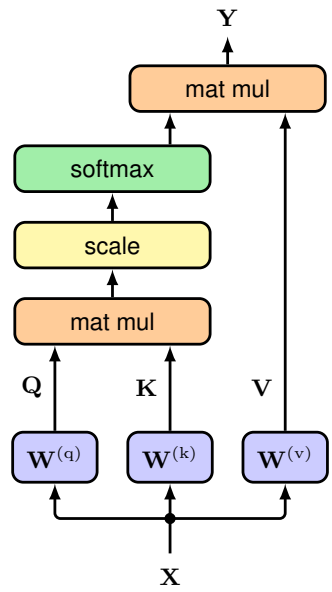


Figure 12.6 Information flow in a scaled dot-product self-attention neural network layer. Here 'mat mul' denotes matrix multiplication, and 'scale' refers to the normalization of the argument to the softmax using $\sqrt{D_\mathrm{k}}$. This structure constitutes a single attention 'head'.

on the network outputs. By contrast, if a standard neural network learns to ignore a particular input or hidden-unit variable, it does so for all input vectors.

12.1.5 Scaled self-attention

There is one final refinement we can make to the self-attention layer. Recall that the gradients of the softmax function become exponentially small for inputs of high magnitude, just as happens with $\tanh$ or logistic-sigmoid activation functions. To help prevent this from happening, we can re-scale the product of the query and key vectors before applying the softmax function. To derive a suitable scaling, note that if the elements of the query and key vectors were all independent random numbers with zero mean and unit variance, then the variance of the dot product would be D_k. We therefore normalize the argument to the softmax using the standard deviation given by the square root of D_k, so that the output of the attention layer takes the form

Exercise 12.4

$$\mathbf{Y} = \mathrm{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) \equiv \mathrm{Softmax}\left[\frac{\mathbf{Q}\mathbf{K}^\mathrm{T}}{\sqrt{D_\mathrm{k}}}\right]\mathbf{V}. \tag{12.14}$$

This is called *scaled dot-product self-attention*, and is the final form of our self-attention neural network layer. The structure of this layer is summarized in Figure 12.6 and in Algorithm 12.1.

12.1.6 Multi-head attention

The attention layer described so far allows the output vectors to attend to data-dependent patterns of input vectors and is called an *attention head*. However, there

Telegram: @uni_k

www.konkur.in

Algorithm 12.1: Scaled dot-product self-attention

Input: Set of tokens $\mathbf{X} \in \mathbb{R}^{N \times D} : \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$
Weight matrices $\{\mathbf{W}^{(\mathrm{q})}, \mathbf{W}^{(\mathrm{k})}\} \in \mathbb{R}^{D \times D_\mathrm{k}}$ and $\mathbf{W}^{(\mathrm{v})} \in \mathbb{R}^{D \times D_\mathrm{v}}$
Output: $\mathrm{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) \in \mathbb{R}^{N \times D_\mathrm{v}} : \{\mathbf{y}_1, \ldots, \mathbf{y}_N\}$

$\mathbf{Q} = \mathbf{X}\mathbf{W}^{(\mathrm{q})}$ // compute queries $\mathbf{Q} \in \mathbb{R}^{N \times D_\mathrm{k}}$
$\mathbf{K} = \mathbf{X}\mathbf{W}^{(\mathrm{k})}$ // compute keys $\mathbf{K} \in \mathbb{R}^{N \times D_\mathrm{k}}$
$\mathbf{V} = \mathbf{X}\mathbf{W}^{(\mathrm{v})}$ // compute values $\mathbf{V} \in \mathbb{R}^{N \times D}$
return $\mathrm{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \mathrm{Softmax}\left[\frac{\mathbf{Q}\mathbf{K}^{\mathrm{T}}}{\sqrt{D_\mathrm{k}}}\right]\mathbf{V}$

might be multiple patterns of attention that are relevant at the same time. In natural language, for example, some patterns might be relevant to tense whereas others might be associated with vocabulary. Using a single attention head can lead to averaging over these effects. Instead we can use multiple attention heads in parallel. These consist of identically structured copies of the single head, with independent learnable parameters that govern the calculation of the query, key, and value matrices. This is analogous to using multiple different filters in each layer of a convolutional network.

Suppose we have H heads indexed by $h = 1, \ldots, H$ of the form

$$\mathbf{H}_h = \mathrm{Attention}(\mathbf{Q}_h, \mathbf{K}_h, \mathbf{V}_h) \tag{12.15}$$

where $\mathrm{Attention}(\cdot, \cdot, \cdot)$ is given by (12.14), and we have defined separate query, key, and value matrices for each head using

$$\mathbf{Q}_h = \mathbf{X}\mathbf{W}_h^{(\mathrm{q})} \tag{12.16}$$

$$\mathbf{K}_h = \mathbf{X}\mathbf{W}_h^{(\mathrm{k})} \tag{12.17}$$

$$\mathbf{V}_h = \mathbf{X}\mathbf{W}_h^{(\mathrm{v})}. \tag{12.18}$$

The heads are first concatenated into a single matrix, and the result is then linearly transformed using a matrix $\mathbf{W}^{(\mathrm{o})}$ to give a combined output in the form

$$\mathbf{Y}(\mathbf{X}) = \mathrm{Concat}\left[\mathbf{H}_1, \ldots, \mathbf{H}_H\right]\mathbf{W}^{(\mathrm{o})}. \tag{12.19}$$

This is illustrated in Figure 12.7.

Each matrix $\mathbf{H}_h$ has dimension $N \times D_\mathrm{v}$, and so the concatenated matrix has dimension $N \times HD_\mathrm{v}$. This is transformed by the linear matrix $\mathbf{W}^{(\mathrm{o})}$ of dimension $HD_\mathrm{v} \times D$ to give the final output matrix $\mathbf{Y}$ of dimension $N \times D$, which is the same as the original input matrix $\mathbf{X}$. The elements of the matrix $\mathbf{W}^{(\mathrm{o})}$ are learned during the training phase along with the query, key, and value matrices. Typically D_v is

Telegram: @uni_k

www.konkur.in

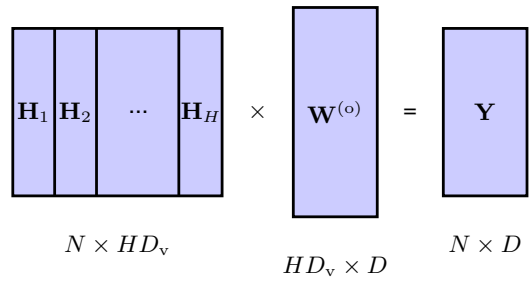


Figure 12.7 Network architecture for multi-head attention. Each head comprises the structure shown in Figure 12.6, and has its own key, query, and value parameters. The outputs of the heads are concatenated and then linearly projected back to the input data dimensionality.

chosen to be equal to D/H so that the resulting concatenated matrix has dimension $N \times D$. Multi-head attention is summarized in Algorithm 12.2, and the information flow in a multi-head attention layer is illustrated in Figure 12.8.

Note that the formulation of multi-head attention given above, which follows that used in the research literature, includes some redundancy in the successive multiplication of the $\mathbf{W}^{(\mathrm{v})}$ matrix for each head and the output matrix $\mathbf{W}^{(\mathrm{o})}$. Removing this redundancy allows a multi-head self-attention layer to be written as a sum over contributions from each of the heads separately.

Exercise 12.5

12.1.7 Transformer layers

Multi-head self-attention forms the core architectural element in a transformer network. We know that neural networks benefit greatly from depth, and so we would like to stack multiple self-attention layers on top of each other. To improve training

Algorithm 12.2: Multi-head attention

Input: Set of tokens $\mathbf{X} \in \mathbb{R}^{N \times D} : \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$
Query weight matrices $\{\mathbf{W}_1^{(\mathrm{q})}, \ldots, \mathbf{W}_H^{(\mathrm{q})}\} \in \mathbb{R}^{D \times D}$
Key weight matrices $\{\mathbf{W}_1^{(\mathrm{k})}, \ldots, \mathbf{W}_H^{(\mathrm{k})}\} \in \mathbb{R}^{D \times D}$
Value weight matrices $\{\mathbf{W}_1^{(\mathrm{v})}, \ldots, \mathbf{W}_H^{(\mathrm{v})}\} \in \mathbb{R}^{D \times D_{\mathrm{v}}}$
Output weight matrix $\mathbf{W}^{(\mathrm{o})} \in \mathbb{R}^{HD_{\mathrm{v}} \times D}$

Output: $\mathbf{Y} \in \mathbb{R}^{N \times D} : \{\mathbf{y}_1, \ldots, \mathbf{x}_N\}$

```
// compute self-attention for each head (Algorithm 12.1)
```

for $h = 1, \ldots, H$ **do**
  $\mathbf{Q}_h = \mathbf{X}\mathbf{W}_h^{(\mathrm{q})}, \quad \mathbf{K}_h = \mathbf{X}\mathbf{W}_h^{(\mathrm{k})}, \quad \mathbf{V}_h = \mathbf{X}\mathbf{W}_h^{(\mathrm{v})}$
  $\mathbf{H}_h = \mathrm{Attention}\left(\mathbf{Q}_h, \mathbf{K}_h, \mathbf{V}_h\right)$ // $\mathbf{H}_h \in \mathbb{R}^{N \times D_{\mathrm{v}}}$
end for
$\mathbf{H} = \mathrm{Concat}\left[\mathbf{H}_1, \ldots, \mathbf{H}_N\right]$ // concatenate heads
return $\mathbf{Y}(\mathbf{X}) = \mathbf{H}\mathbf{W}^{(\mathrm{o})}$

Telegram: @uni_k

www.konkur.in

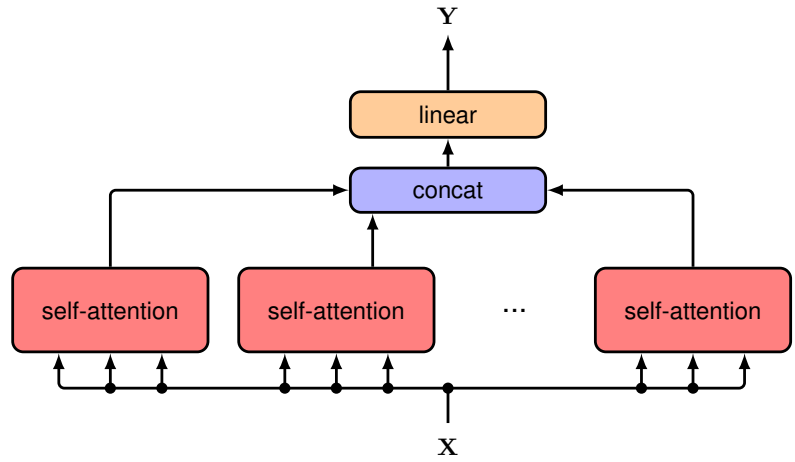


Figure 12.8 Information flow in a multi-head attention layer. The associated computation, given by Algorithm 12.2, is illustrated in Figure 12.7.

Section 9.5

Section 7.4.3

efficiency, we can introduce residual connections that bypass the multi-head structure. To do this we require that the output dimensionality is the same as the input dimensionality, namely $N \times D$. This is then followed by *layer normalization* (Ba, Kiros, and Hinton, 2016), which improves training efficiency. The resulting transformation can be written as

$$\mathbf{Z} = \text{LayerNorm}\left[\mathbf{Y}(\mathbf{X}) + \mathbf{X}\right] \tag{12.20}$$

where $\mathbf{Y}$ is defined by (12.19). Sometimes the layer normalization is replaced by *pre-norm* in which the normalization layer is applied before the multi-head self-attention instead of after, as this can result in more effective optimization, in which case we have

$$\mathbf{Z} = \mathbf{Y}(\mathbf{X}') + \mathbf{X}, \quad \text{where} \quad \mathbf{X}' = \text{LayerNorm}\left[\mathbf{X}\right]. \tag{12.21}$$

In each case, $\mathbf{Z}$ again has the same dimensionality $N \times D$ as the input matrix $\mathbf{X}$.

We have seen that the attention mechanism creates linear combinations of the value vectors, which are then linearly combined to produce the output vectors. Also, the values are linear functions of the input vectors, and so we see that the outputs of an attention layer are constrained to be linear combinations of the inputs. Nonlinearity does enter through the attention weights, and so the outputs will depend nonlinearly on the inputs via the softmax function, but the output vectors are still constrained to lie in the subspace spanned by the input vectors and this limits the expressive capabilities of the attention layer. We can enhance the flexibility of the transformer by post-processing the output of each layer using a standard nonlinear neural network with D inputs and D outputs, denoted $\text{MLP}[\cdot]$ for 'multilayer perceptron'. For example, this might consist of a two-layer fully connected network with ReLU hidden units. This needs to be done in a way that preserves the ability

Telegram: @uni_k

www.konkur.in

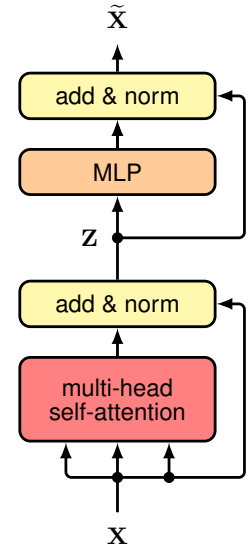


Figure 12.9 One layer of the transformer architecture that implements the transformation (12.1). Here 'MLP' stands for multilayer perceptron, while 'add and norm' denotes a residual connection followed by layer normalization.

of the transformer to process sequences of variable length. To achieve this, the same shared network is applied to each of the output vectors, corresponding to the rows of $\mathbf{Z}$. Again, this neural network layer can be improved by using a residual connection. It also includes layer normalization so that the final output from the transformer layer has the form

$$\widetilde{\mathbf{X}} = \text{LayerNorm}\left[\text{MLP}\left[\mathbf{Z}\right] + \mathbf{Z}\right]. \tag{12.22}$$

This leads to an overall architecture for a transformer layer shown in Figure 12.9 and summarized in Algorithm 12.3. Again, we can use a pre-norm instead, in which case the final output is given by

$$\widetilde{\mathbf{X}} = \text{MLP}(\mathbf{Z}') + \mathbf{Z}, \quad \text{where} \quad \mathbf{Z}' = \text{LayerNorm}\left[\mathbf{Z}\right]. \tag{12.23}$$

In a typical transformer there are multiple such layers stacked on top of each other. The layers generally have identical structures, although there is no sharing of weights and biases between different layers.

12.1.8 Computational complexity

The attention layer discussed so far takes a set of N vectors each of length D and maps them into another set of N vectors having the same dimensionality. Thus, the inputs and outputs each have overall dimensionality ND. If we had used a standard fully connected neural network to map the input values to the output values, it would have $\mathcal{O}(N^2D^2)$ independent parameters. Likewise the computational cost of evaluating one forward pass through such a network would also be $\mathcal{O}(N^2D^2)$.

In the attention layer, the matrices $\mathbf{W}^{(\text{q})}$, $\mathbf{W}^{(\text{k})}$, and $\mathbf{W}^{(\text{v})}$ are shared across input tokens, and therefore the number of independent parameters is $\mathcal{O}(D^2)$, assuming $D_{\text{k}} \simeq D_{\text{v}} \simeq D$. Since there are N input tokens, the number of computational steps

Telegram: @uni_k

www.konkur.in

Algorithm 12.3: Transformer layer

Input: Set of tokens $\mathbf{X} \in \mathbb{R}^{N \times D} : \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$
Multi-head self-attention layer parameters
Feed-forward network parameters
Output: $\widetilde{\mathbf{X}} \in \mathbb{R}^{N \times D} : \{\widetilde{\mathbf{x}}_1, \ldots, \widetilde{\mathbf{x}}_N\}$

$\mathbf{Z} = \text{LayerNorm}\left[\mathbf{Y}(\mathbf{X}) + \mathbf{X}\right]$ `// Y(X) from Algorithm 12.2`
$\widetilde{\mathbf{X}} = \text{LayerNorm}\left[\text{MLP}\left[\mathbf{Z}\right] + \mathbf{Z}\right]$ `// shared neural network`
return $\widetilde{\mathbf{X}}$

in evaluating the dot products in a self-attention layer is $\mathcal{O}(N^2D)$. We can think of a self-attention layer as a sparse matrix in which parameters are shared between specific blocks of the matrix. The subsequent neural network layer, which has D inputs and D outputs, has a cost that is $\mathcal{O}(D^2)$. Since it is shared across tokens, it has a complexity that is linear in N, and therefore overall this layer has a cost that is $\mathcal{O}(ND^2)$. Depending on the relative sizes of N and D, either the transformer layer or the MLP layer may dominate the computational cost. Compared to a fully connected network, a transformer layer is computationally more efficient. Many variants of the transformer architecture have been proposed (Lin *et al.*, 2021; Phuong and Hutter, 2022) including modifications aimed at improving efficiency (Tay *et al.*, 2020).

Exercise 12.6

12.1.9 Positional encoding

In the transformer architecture, the matrices $\mathbf{W}_h^{(\text{q})}$, $\mathbf{W}_h^{(\text{k})}$, and $\mathbf{W}_h^{(\text{v})}$ are shared across the input tokens, as is the subsequent neural network. As a consequence, the transformer has the property that permuting the order of the input tokens, i.e., the rows of $\mathbf{X}$, results in the same permutation of the rows of the output matrix $\widetilde{\mathbf{X}}$. In other words a transformer is *equivariant* with respect to input permutations. The sharing of parameters in the network architecture facilitates the massively parallel processing of the transformer, and also allows the network to learn long-range dependencies just as effectively as short-range dependencies. However, the lack of dependence on token order becomes a major limitation when we consider sequential data, such as the words in a natural language, because the representation learned by a transformer will be independent of the input token ordering. The two sentences '*The food was bad, not good at all.*' and '*The food was good, not bad at all.*' contain the same tokens but they have very different meanings because of the different token ordering. Clearly token order is crucial for most sequential processing tasks including natural language processing, and so we need to find a way to inject token order information into the network.

Exercise 12.7
Section 10.2

Since we wish to retain the powerful properties of the attention layers that we have carefully constructed, we aim to encode the token order in the data itself in-

Telegram: @uni_k

www.konkur.in

stead of having to be represented in the network architecture. We will therefore construct a position encoding vector $\mathbf{r}_n$ associated with each input position n and then combine this with the associated input token embedding $\mathbf{x}_n$. One obvious way to combine these vectors would be to concatenate them, but this would increase the dimensionality of the input space and hence of all subsequent attention spaces, creating a significant increase in computational cost. Instead, we can simply add the position vectors onto the token vectors to give

$$\widetilde{\mathbf{x}}_n = \mathbf{x}_n + \mathbf{r}_n. \tag{12.24}$$

This requires that the positional encoding vectors have the same dimensionality as the token-embedding vectors.

At first it might seem that adding position information onto the token vector would corrupt the input vectors and make the task of the network much more difficult. However, some intuition as to why this can work well comes from noting that two randomly chosen uncorrelated vectors tend to be nearly orthogonal in spaces of high dimensionality, indicating that the network is able to process the token identity information and the position information relatively separately. Note also that, because of the residual connections across every layer, the position information does not get lost in going from one transformer layer to the next. Moreover, due to the linear processing layers in the transformer, a concatenated representation has similar properties to an additive one.

Exercise 12.8

Exercise 12.9

The next task is to construct the embedding vectors $\{\mathbf{r}_n\}$. A simple approach would be to associate an integer $1, 2, 3, \ldots$ with each position. However, this has the problem that the magnitude of the value increases without bound and therefore may start to corrupt the embedding vector significantly. Also it may not generalize well to new input sequences that are longer than those used in training, since these will involve coding values that lie outside the range of those used in training. Alternatively we could assign a number in the range $(0, 1)$ to each token in the sequence, which keeps the representation bounded. However, this representation is not unique for a given position as it depends on the overall sequence length.

An ideal positional encoding should provide a unique representation for each position, it should be bounded, it should generalize to longer sequences, and it should have a consistent way to express the number of steps between any two input vectors irrespective of their absolute position because the relative position of tokens is often more important than the absolute position.

There are many approaches to positional encoding (Dufter, Schmitt, and Schütze, 2021). Here we describe a technique based on sinusoidal functions introduced by Vaswani *et al.* (2017). For a given position n the associated position-encoding vector has components r_{ni} given by

$$r_{ni} = \begin{cases} \sin\left(\dfrac{n}{L^{i/D}}\right), & \text{if } i \text{ is even,} \\ \cos\left(\dfrac{n}{L^{(i-1)/D}}\right), & \text{if } i \text{ is odd.} \end{cases} \tag{12.25}$$

We see that the elements of the embedding vector $\mathbf{r}_n$ are given by a series of sine and cosine functions of steadily increasing wavelength, as illustrated in Figure 12.10(a).

Telegram: @uni_k

www.konkur.in

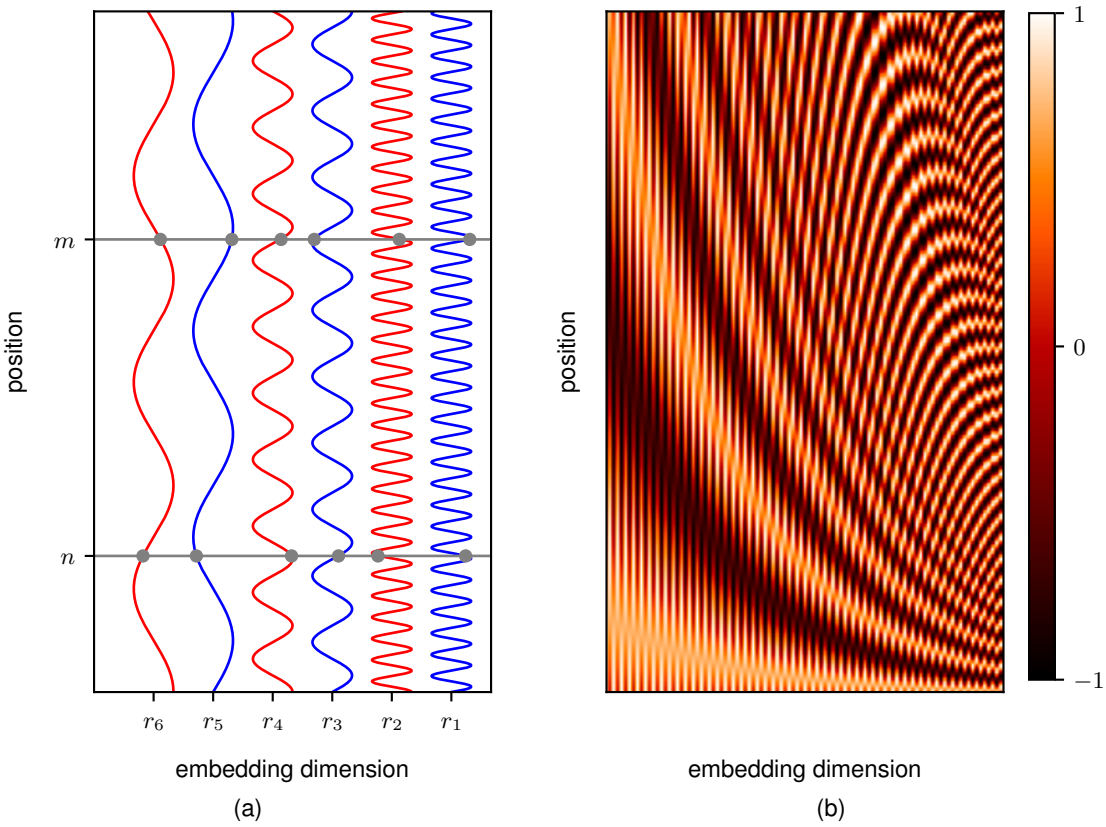


Figure 12.10 Illustrations of the functions defined by (12.25) and used to construct position-encoding vectors. (a) A plot in which the horizontal axis shows the different components of the embedding vector $\mathbf{r}$ whereas the vertical axis shows the position in the sequence. The values of the vector elements for two positions n and m are shown by the intersections of the sine and cosine curves with the horizontal grey lines. (b) A heat map illustration of the position-encoding vectors defined by (12.25) for dimension $D = 100$ with $L = 30$ for the first $N = 200$ positions.

Telegram: @uni_k

www.konkur.in

This encoding has the property that the elements of the vector $\mathbf{r}_n$ all lie in the range $(-1, 1)$. It is reminiscent of the way binary numbers are represented, with the lowest order bit alternating with high frequency, and subsequent bits alternating with steadily decreasing frequencies:

$$\begin{array}{ll} \mathbf{1}: & 0\ 0\ 0\ 1 \\ \mathbf{2}: & 0\ 0\ 1\ 0 \\ \mathbf{3}: & 0\ 0\ 1\ 1 \\ \mathbf{4}: & 0\ 1\ 0\ 0 \\ \mathbf{5}: & 0\ 1\ 0\ 1 \\ \mathbf{6}: & 0\ 1\ 1\ 0 \\ \mathbf{7}: & 0\ 1\ 1\ 1 \\ \mathbf{8}: & 1\ 0\ 0\ 0 \\ \mathbf{9}: & 1\ 0\ 0\ 1 \end{array}$$

For the encoding given by (12.25), however, the vector elements are continuous variables rather than binary. A plot of the position-encoding vectors is shown in Figure 12.10(b).

One nice property of the sinusoidal representation given by (12.25) is that, for any fixed offset k, the encoding at position $n + k$ can be represented as a linear combination of the encoding at position n, in which the coefficients do not depend on the absolute position but only on the value of k. The network should therefore be able to learn to attend to relative positions. Note that this property requires that the encoding makes use of both sine and cosine functions.

Exercise 12.10

Another popular approach to positional representation is to use learned position encodings. This is done by having a vector of weights at each token position that can be learned jointly with the rest of the model parameters during training, and avoids using hand-crafted representations. Because the parameters are not shared between the token positions, the tokens are no longer invariant under a permutation, which is the purpose of a positional encoding. However, this approach does not meet the criteria we mentioned earlier of generalizing to longer input sequences, as the encoding will be untrained for positional encodings not seen during training. Therefore, this approach is generally most suitable when the input length is relatively constant during both training and inference.

12.2. Natural Language

Now that we have studied the architecture of the transformer, we will explore how this can be used to process language data consisting of words, sentences, and paragraphs. Although this is the modality that transformers were originally developed to operate on, they have proved to be a very general class of models and have become the state-of-the-art for most input data types. Later in this chapter we will look at their use in other domains.

Section 12.4

Many languages, including English, comprise a series of words separated by white space, along with punctuation symbols, and therefore represent an example of

Telegram: @uni_k

www.konkur.in

Section 11.3
Section 12.2.2

sequential data. For the moment we will focus on the words, and we will return to punctuation later.

The first challenge is to convert the words into a numerical representation that is suitable for use as the input to a deep neural network. One simple approach is to define a fixed dictionary of words and then introduce vectors of length equal to the size of the dictionary along with a 'one hot' representation for each word, in which the kth word in the dictionary is encoded with a vector having a 1 in position k and 0 in all other positions. For example if 'aardwolf' is the third word in our dictionary then its vector representation would be $(0, 0, 1, 0, \ldots, 0)$.

An obvious problem with a one-hot representation is that a realistic dictionary might have several hundred thousand entries leading to vectors of very high dimensionality. Also, it does not capture any similarities or relationships that might exist between words. Both issues can be addressed by mapping the words into a lower-dimensional space through a process called *word embedding* in which each word is represented as a dense vector in a space of typically a few hundred dimensions.

12.2.1 Word embedding

The embedding process can be defined by a matrix $\mathbf{E}$ of size $D \times K$ where D is the dimensionality of the embedding space and K is the dimensionality of the dictionary. For each one-hot encoded input vector $\mathbf{x}_n$ we can then calculate the corresponding embedding vector using

$$\mathbf{v}_n = \mathbf{E}\mathbf{x}_n. \tag{12.26}$$

Because $\mathbf{x}_n$ has a one-hot encoding, the vector $\mathbf{v}_n$ is simply given by the corresponding column of the matrix $\mathbf{E}$.

We can learn the matrix $\mathbf{E}$ from a corpus (i.e., a large data set) of text, and there are many approaches to doing this. Here we look at a popular technique called *word2vec* (Mikolov *et al.*, 2013), which can be viewed as a simple two-layer neural network. A training set is constructed in which each sample is obtained by considering a 'window' of M adjacent words in the text, where a typical value might be $M = 5$. The samples are considered to be independent, and the error function is defined as the sum of the error functions for each sample. There are two variants of this approach. In *continuous bag of words*, the target variable for network training is the middle word, and the remaining *context* words form the inputs, so that the network is being trained to 'fill in the blank'. A closely related approach, called *skip-grams*, reverses the inputs and outputs, so that the centre word is presented as the input and the target values are the context words. These models are illustrated in Figure 12.11.

This training procedure can be viewed as a form of *self-supervised* learning since the data consists simply of a large corpus of unlabelled text from which many small windows of word sequences are drawn at random. Labels are obtained from the text itself by 'masking' out those words whose values the network is trying to predict.

Once the model is trained, the embedding matrix $\mathbf{E}$ is given by the transpose of the second-layer weight matrix for the continuous bag-of-words approach and by the first-layer weight matrix for skip-grams. Words that are semantically related are mapped to nearby positions in the embedding space. This is to be expected

Telegram: @uni_k

www.konkur.in

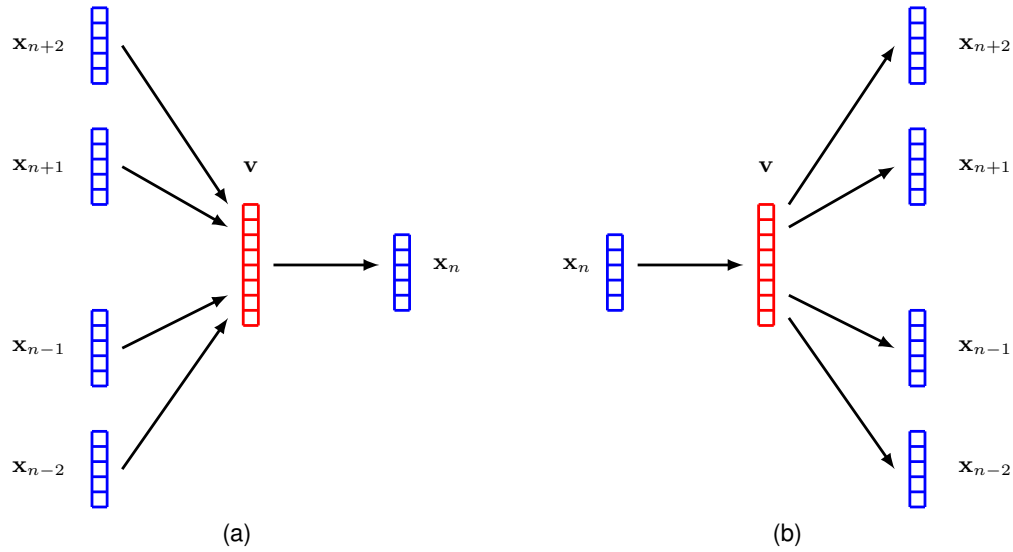


Figure 12.11 Two-layer neural networks used to learn word embeddings, where (a) shows the continuous bag-of-words approach, and (b) shows the skip-grams approach.

since related words are more likely to occur with similar context words compared to unrelated words. For example, the words 'city' and 'capital' might occur with higher frequency as context for target words such as 'Paris' or 'London' and less frequently as context for 'orange' or 'polynomial'. The network can more easily predict the probability of the missing words if 'Paris' and 'London' are mapped to nearby embedding vectors.

It turns out that the learned embedding space often has an even richer semantic structure than just the proximity of related words, and that this allows for simple vector arithmetic. For example, the concept that 'Paris is to France as Rome is to Italy' can be expressed through operations on the embedding vectors. If we use $\mathbf{v}(\text{word})$ to denote the embedding vector for 'word', then we find

$$\mathbf{v}(\text{Paris}) - \mathbf{v}(\text{France}) + \mathbf{v}(\text{Italy}) \simeq \mathbf{v}(\text{Rome}). \tag{12.27}$$

Word embeddings were originally developed as natural language processing tools in their own right. Today, they are more likely to be used as pre-processing steps for deep neural networks. In this regard they can be viewed as the first layer in a deep neural network. They can be fixed using some standard pre-trained embedding matrix, or they can be treated as an adaptive layer that is learned as part of the overall end-to-end training of the system. In the latter case the embedding layer can be initialized either using random weight values or using a standard embedding matrix.

Telegram: @uni_k

www.konkur.in

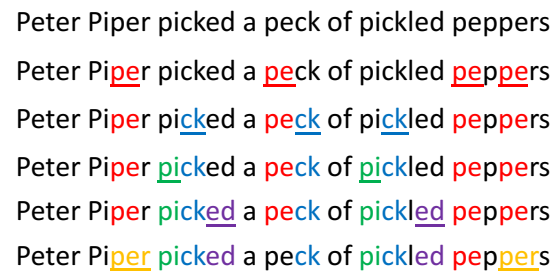


Figure 12.12 An illustration of the process of tokenizing natural language by analogy with byte pair encoding. In this example, the most frequently occurring pair of characters is 'pe', which occurs four times, and so these form a new token that replaces all the occurrences of 'pe'. Note that 'Pe' is not included in this since upper-case 'P' and lower-case 'p' are distinct characters. Next the pair 'ck' is added since this occurs three times. This is followed by tokens such as 'pi', 'ed', and 'per', all of which occur twice, and so on.

12.2.2 Tokenization

One problem with using a fixed dictionary of words is that it cannot cope with words not in the dictionary or which are misspelled. It also does not take account of punctuation symbols or other character sequences such as computer code. An alternative approach that addresses these problems would be to work at the level of characters instead of using words, so that our dictionary comprises upper-case and lower-case letters, numbers, punctuation, and white-space symbols such as spaces and tabs. A disadvantage of this approach, however, is that it discards the semantically important word structure of language, and the subsequent neural network would have to learn to reassemble words from elementary characters. It would also require a much larger number of sequential steps for a given body of text, thereby increasing the computational cost of processing the sequence.

We can combine the benefits of character-level and word-level representations by using a pre-processing step that converts a string of words and punctuation symbols into a string of *tokens*, which are generally small groups of characters and might include common words in their entirety, along with fragments of longer words as well as individual characters that can be assembled into less common words (Schuster and Nakajima, 2012). This tokenization also allows the system to process other kinds of sequences such as computer code or even other modalities such as images. It also means that variations of the same word can have related representations. For example, 'cook', 'cooks', 'cooked', 'cooking', and 'cooker' are all related and share the common element 'cook', which itself could be represented as one of the tokens.

Section 12.4.1

There are many approaches to tokenization. As an example, a technique called *byte pair encoding* that is used for data compression, can be adapted to text tokenization by merging characters instead of bytes (Sennrich, Haddow, and Birch, 2015). The process starts with the individual characters and iteratively merges them into longer strings. The list of tokens is first initialized with the list of individual characters. Then a body of text is searched for the most frequently occurring adjacent pairs of tokens and these are replaced with a new token. To ensure that words are not merged, a new token is not formed from two tokens if the second token starts with a white space. The process is repeated iteratively as illustrated in Figure 12.12.

Initially the number of tokens is equal to the number of characters, which is relatively small. As tokens are formed, the total number of tokens increases, and

Telegram: @uni_k

www.konkur.in

if this is continued long enough, the tokens will eventually correspond to the set of words in the text. The total number of tokens is generally fixed in advance, as a compromise between character-level and word-level representations. The algorithm is stopped when this number of tokens is reached.

In practical applications of deep learning to natural language, the input text is typically first mapped into a tokenized representation. However, for the remainder of this chapter, we will use word-level representations as this makes it easier to illustrate and motivate key concepts.

12.2.3 Bag of words

We now turn to the task of modelling the joint distribution $p(\mathbf{x}_1, \ldots, \mathbf{x}_N)$ of an ordered sequence of vectors, such as words (or tokens) in a natural language. The simplest approach is to assume that the words are drawn independently from the same distribution and hence that the joint distribution is fully factorized in the form

$$p(\mathbf{x}_1, \ldots, \mathbf{x}_N) = \prod_{n=1}^{N} p(\mathbf{x}_n). \tag{12.28}$$

This can be expressed as a probabilistic graphical model in which the nodes are isolated with no interconnecting links.

Figure 11.28

The distribution $p(\mathbf{x})$ is shared across the variables and can be represented, without loss of generality, as a simple table listing the probabilities of each of the possible states of $\mathbf{x}$ (corresponding to the dictionary of words or tokens). The maximum likelihood solution for this model is obtained simply by setting each of these probabilities to the fraction of times that the word occurs in the training set. This is known as a *bag-of-words* model because it completely ignores the ordering of the words.

Exercise 12.11

We can use the bag-of-words approach to construct a simple text classifier. This could be used for example in sentiment analysis in which a passage of text representing a restaurant review is to be classified as positive or negative. The *naive Bayes* classifier assumes that the words are independent within each class $\mathcal{C}_k$, but with a different distribution for each class, so that

$$p(\mathbf{x}_1, \ldots, \mathbf{x}_N | \mathcal{C}_k) = \prod_{n=1}^{N} p(\mathbf{x}_n | \mathcal{C}_k). \tag{12.29}$$

Given prior class probabilities $p(\mathcal{C}_k)$, the posterior class probabilities for a new sequence are given by:

$$p(\mathcal{C}_k | \mathbf{x}_1, \ldots, \mathbf{x}_N) \propto p(\mathcal{C}_k) \prod_{n=1}^{N} p(\mathbf{x}_n | \mathcal{C}_k). \tag{12.30}$$

Both the class-conditional densities $p(\mathbf{x}|\mathcal{C}_k)$ and the prior probabilities $p(\mathcal{C}_k)$ can be estimated using frequencies from the training data set. For a new sequence, the table entries are multiplied together to get the desired posterior probabilities. Note that if a word occurs in the test set that was not present in the training set then the

Telegram: @uni_k

www.konkur.in

corresponding probability estimate will be zero, and so these estimates are typically 'smoothed' after training by reassigning a small level of probability uniformly across all entries to avoid zero values.

12.2.4 Autoregressive models

One obvious major limitation of the bag-of-words model is that it completely ignores word order. To address this we can take an autoregressive approach. Without loss of generality we can decompose the distribution over the sequence of words into a product of conditional distributions in the form

$$p(\mathbf{x}_1, \ldots, \mathbf{x}_N) = \prod_{n=1}^{N} p(\mathbf{x}_n | \mathbf{x}_1, \ldots, \mathbf{x}_{n-1}). \tag{12.31}$$

Figure 11.27

This can be represented as a probabilistic graphical model in which each node in the sequence receives a link from every previous node. We could represent each term on the right-hand side of (12.31) by a table whose entries are once again estimated using simple frequency counts from the training set. However, the size of these tables grows exponentially with the length of the sequence, and so this approach would become prohibitively expensive.

Exercise 12.12

We can simplify the model dramatically by assuming that each of the conditional distributions on the right-hand side of (12.31) is independent of all previous observations except the L most recent words. For example, if $L = 2$ then the joint distribution for a sequence of N observations under this model is given by

$$p(\mathbf{x}_1, \ldots, \mathbf{x}_N) = p(\mathbf{x}_1)p(\mathbf{x}_2|\mathbf{x}_1) \prod_{n=3}^{N} p(\mathbf{x}_n | \mathbf{x}_{n-1}, \mathbf{x}_{n-2}). \tag{12.32}$$

Figure 11.30

In the corresponding graphical model each node has links from the two previous nodes. Here we assume that the conditional distributions $p(\mathbf{x}_n|\mathbf{x}_{n-1})$ are shared across all variables. Again each of the distributions on the right-hand side of (12.32) can be represented as tables whose values are estimated from the statistics of triplets of successive words drawn from a training corpus.

The case with $L = 1$ is known as a *bi-gram* model because it depends on pairs of adjacent words. Similarly $L = 2$, which involves triplets of adjacent words, is called a *tri-gram* model, and in general these are called *n-gram* models.

All the models discussed so far in this section can be run *generatively* to synthesize novel text. For example, if we provide the first and second words in a sequence, then we can sample from the tri-gram statistics $p(\mathbf{x}_n|\mathbf{x}_{n-1}, \mathbf{x}_{n-2})$ to generate the third word, and then we can use the second and third words to sample the fourth word, and so on. The resulting text, however, will be incoherent because each word is predicted only on the basis of the two previous words. High-quality text models must take account of the long-range dependencies in language. On the other hand, we cannot simply increase the value of L because the size of the probability tables grows exponentially in L so that it is prohibitively expensive to go much beyond tri-gram models. However, the autoregressive representation will play a central role

Telegram: @uni_k

www.konkur.in

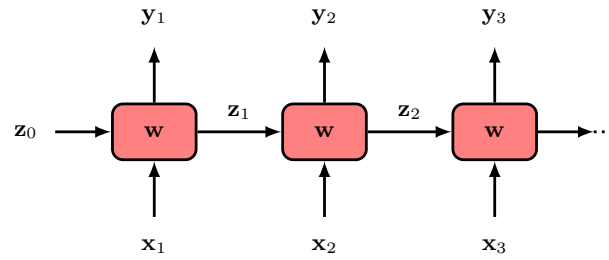


Figure 12.13 A general RNN with parameters $\mathbf{w}$. It takes a sequence $\mathbf{x}_1, \ldots, \mathbf{x}_N$ as input and generates a sequence $\mathbf{y}_1, \ldots, \mathbf{y}_N$ as output. Each of the boxes corresponds to a multilayer network with nonlinear hidden units.

when we consider modern language models based not on probability tables but on deep neural networks configured as transformers.

One way to allow longer-range dependencies, while avoiding the exponential growth in the number of parameters of an n-gram model, is to use a *hidden Markov model* whose graphical structure is shown in Figure 11.31. The number of learnable parameters is governed by the dimensionality of the latent variables whereas the distribution over a given observation $\mathbf{x}_n$ depends, in principle, on all previous observations. However, the influence of more distant observations is still very limited since their effect must be carried through the chain of latent states which are themselves being updated by more recent observations.

Section 11.3.1

12.2.5 Recurrent neural networks

Techniques such as n-grams have very poor scaling with sequence length because they store completely general tables of conditional distributions. We can achieve much better scaling by using parameterized models based on neural networks. Suppose we simply apply a standard feed-forward neural network to sequences of words in natural language. One problem that arises is that the network has a fixed number of inputs and outputs, whereas we need to be able to handle sequences in the training and test sets that have variable length. Furthermore, if a word, or group of words, at a particular location in a sequence represents some concept then the same word, or group of words, at a different location is likely to represent the same concept at that new location. This is reminiscent of the equivariance property we encountered in processing image data. If we can construct a network architecture that is able to share parameters across the sequence then not only can we capture this equivariance property but we can greatly reduce the number of free parameters in the model as well as handle sequences having different lengths.

Chapter 10

To address this we can borrow inspiration from the hidden Markov model and introduce an explicit hidden variable $\mathbf{z}_n$ associated with each step n in the sequence. The neural network takes as input both the current word $\mathbf{x}_n$ and the current hidden state $\mathbf{z}_{n-1}$ and produces an output word $\mathbf{y}_n$ as well as the next state $\mathbf{z}_n$ of the hidden variable. We can then chain together copies of this network, in which the weight values are shared across the copies. The resulting architecture is called a *recurrent neural network* (RNN) and is illustrated in Figure 12.13. Here the initial value of the hidden state may be initialized for example to some default value such as $\mathbf{z}_0 =$

Telegram: @uni_k

www.konkur.in

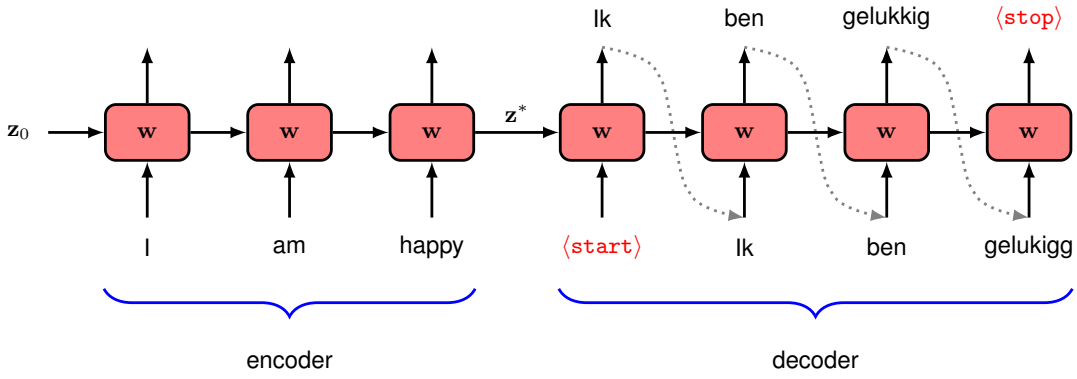


Figure 12.14 An example of a recurrent neural network used for language translation. See the text for details.

$(0, 0, \ldots, 0)^{\mathrm{T}}$.

As an example of how an RNN might be used in practice, consider the specific task of translating sentences from English into Dutch. The sentences can have variable length, and each output sentence might have a different length from the corresponding input sentence. Furthermore, the network may need to see the whole of the input sentence before it can even start to generate the output sentence. We can address this using an RNN by feeding in the complete English sentence followed by a special input token, which we denote by ⟨start⟩, to trigger the start of translation. During training the network learns to associate ⟨start⟩ with the beginning of the output sentence. We also take each successively generated word and feed it into the input at the next time step, as shown in Figure 12.14. The network can be trained to generate a specific ⟨stop⟩ token to signify the completion of the translation. The first few stages of the network are used to absorb the input sequence, and the associated output vectors are simply ignored. This part of the network can be viewed as an 'encoder' in which the entire input sentence has been compressed into the state $\mathbf{z}^\star$ of the hidden variable. The remaining network stages function as the 'decoder', which generates the translated sentence as output one word at a time. Notice that each output word is fed as input to the next stage of the network, and so this approach has an autoregressive structure analogous to (12.31).

12.2.6 Backpropagation through time

RNNs can be trained by stochastic gradient descent using gradients calculated by backpropagation and evaluated through automatic differentiation, just as with regular neural networks. The error function consists of a sum over all output units of the error for each unit, in which each output unit has a softmax activation function along with an associated cross-entropy error function. During forward propagation, the activation values are propagated all the way from the first input in the sequence through to all the output nodes in the sequence, and error signals are then backpropagated along the same paths. This process is called *backpropagation through time*

Section 5.4.4

Telegram: @uni_k

www.konkur.in

and in principle is straightforward. However, in practice, for very long sequences, training can be difficult due to the problems of *vanishing gradients* or *exploding gradients* that arise with very deep network architectures.

Section 7.4.2

Another problem with standard RNNs is that they deal poorly with long-range dependencies. This is especially problematic for natural language where such dependencies are widespread. In a long passage of text, a concept might be introduced that plays an important role in predicting words occurring much later in the text. In the architecture shown in Figure 12.14, the entire concept of the English sentence must be captured in the single hidden vector $\mathbf{z}^\star$ of fixed length, and this becomes increasingly problematic with longer sequences. This is known as the *bottleneck problem* because a sequence of arbitrary length has to be summarized in a single hidden vector of activations and the network can start to generate the output translation only once the full input sequence has been processed.

One approach for addressing both the vanishing and exploding gradients problems and the limited long-range dependencies is to modify the architecture of the neural network to allow additional signal paths that bypass many of the processing steps within each stage of the network and hence allow information to be remembered over a larger number of time steps. *Long short-term memory* (LSTM) models (Hochreiter and Schmidhuber, 1997) and *gated recurrent unit* (GRU) models (Cho *et al.*, 2014) are the most widely known examples. Although they improve performance compared to standard RNNs, they still have a limited ability to model long-range dependencies. Also, the additional complexity of each cell means that LSTMs are even slower to train than standard RNNs. Furthermore, all recurrent networks have signal paths that grow linearly with the number of steps in the sequence. Moreover, they do not support parallel computation within a single training example due to the sequential nature of the processing. In particular, this means that RNNs struggle to make efficient use of modern highly parallel hardware based on GPUs. These problems are addressed by replacing RNNs with transformers.

12.3. Transformer Language Models

The transformer processing layer is a highly flexible component for building powerful neural network models with broad applicability. In this section we explore the application of transformers to natural language. This has given rise to the development of massive neural networks known as *large language models* (LLMs), which have proven to be exceptionally capable (Zhao *et al.*, 2023).

Transformers can be applied to many different kinds of language processing task, and can be grouped into three categories according to the form of the input and output data. In a problem such as sentiment analysis, we take a sequence of words as input and provide a single variable representing the sentiment of the text, for example happy or sad, as output. Here a transformer is acting as an 'encoder' of the sequence. Other problems might take a single vector as input and generate a word sequence as output, for example if we wish to generate a text caption given an input image. In such cases the transformer functions as a 'decoder', generating

Telegram: @uni_k

www.konkur.in

a sequence as output. Finally, in sequence-to-sequence processing tasks, both the input and the output comprise a sequence of words, for example if our goal is to translate from one language to another. In this case, transformers are used in both encoder and decoder roles. We discuss each of these classes of language model in turn, using illustrative examples of model architectures.

12.3.1 Decoder transformers

We start by considering decoder-only transformer models. These can be used as *generative models* that create output sequences of tokens. As an illustrative example, we will focus on a class of models called *GPT* which stands for *generative pre-trained transformer* (Radford *et al.*, 2019; Brown *et al.*, 2020; OpenAI, 2023). The goal is to use the transformer architecture to construct an autoregressive model of the form defined by (12.31) in which the conditional distributions $p(\mathbf{x}_n|\mathbf{x}_1, \ldots, \mathbf{x}_{n-1})$ are expressed using a transformer neural network that is learned from data.

The model takes as input a sequence consisting of the first $n-1$ tokens, and its corresponding output represents the conditional distribution for token n. If we draw a sample from this distribution then we have extended the sequence to n tokens and this new sequence can be fed back through the model to give a distribution over token $n+1$, and so on. The process can be repeated to generate sequences up to a maximum length determined by the number of inputs to the transformer. We will shortly discuss strategies for sampling from the conditional distributions, but for the moment we focus on how to construct and train the network.

Section 12.3.2

The architecture of a GPT model consists of a stack of transformer layers that take a sequence $\mathbf{x}_1, \ldots, \mathbf{x}_N$ of tokens, each of dimensionality D, as input and produce a sequence $\widetilde{\mathbf{x}}_1, \ldots, \widetilde{\mathbf{x}}_N$ of tokens, again of dimensionality D, as output. Each output needs to represent a probability distribution over the dictionary of tokens at that time step, and this dictionary has dimensionality K whereas the tokens have a dimensionality of D. We therefore make a linear transformation of each output token using a matrix $\mathbf{W}^{(\mathrm{p})}$ of dimensionality $D \times K$ followed by a softmax activation function in the form

$$\mathbf{Y} = \mathrm{Softmax}\left(\widetilde{\mathbf{X}}\mathbf{W}^{(\mathrm{p})}\right) \tag{12.33}$$

where $\mathbf{Y}$ is a matrix whose nth row is $\mathbf{y}_n^{\mathrm{T}}$, and $\widetilde{\mathbf{X}}$ is a matrix whose nth row is $\widetilde{\mathbf{x}}_n^{\mathrm{T}}$. Each softmax output unit has an associated cross-entropy error function. The architecture of the model is shown in Figure 12.15.

Section 5.4.4

The model can be trained using a large corpus of unlabelled natural language by taking a self-supervised approach. Each training sample consists of a sequence of tokens $\mathbf{x}_1, \ldots, \mathbf{x}_n$, which form the input to the network, along with an associated target value $\mathbf{x}_{n+1}$ consisting of the next token in the sequence. The sequences are considered to be independent and identically distributed so that the error function used for training is the sum of the cross-entropy error values summed over the training set, grouped into appropriate mini-batches. Naively we could process each such training sample independently using a forward pass through the model. However, we can achieve much greater efficiency by processing an entire sequence at once so that each token acts both as a target value for the sequence of previous tokens and as

Telegram: @uni_k

www.konkur.in

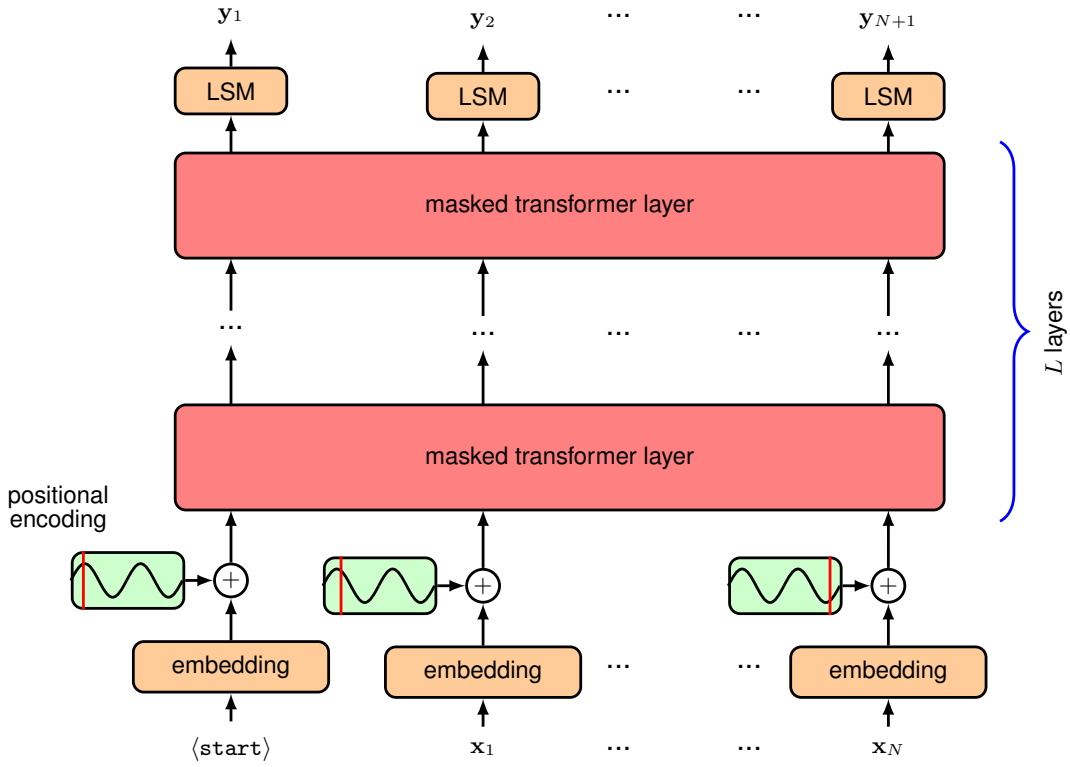


Figure 12.15 Architecture of a GPT decoder transformer network. Here 'LSM' stands for linear-softmax and denotes a linear transformation whose learnable parameters are shared across the token positions, followed by a softmax activation function. Masking is explained in the text.

an input value for subsequent tokens. For example, consider the word sequence

I swam across the river to get to the other bank.

We can use 'I swam across' as an input sequence with an associated target of 'the', and also use 'I swam across the' as an input sequence with an associated target of 'river', and so on. However, to process these in parallel we have to ensure that the network is not able to 'cheat' by looking ahead in the sequence, otherwise it will simply learn to copy the next input directly to the output. If it did this, it would then be unable to generate new sequences since the subsequent token by definition is not available at test time. To address this problem we do two things. First, we shift the input sequence to the right by one step, so that input $\mathbf{x}_n$ corresponds to output $\mathbf{y}_{n+1}$, with target $\mathbf{x}_{n+1}$, and an additional special token denoted $\langle\texttt{start}\rangle$ is prepended in the first position of the input sequence. Second, note that the tokens in a transformer are processed independently, except when they are used to compute the attention weights, when they interact in pairs through the dot product. We therefore introduce *masked attention*, sometimes called *causal attention*, into each of the at-

Telegram: @uni_k

www.konkur.in

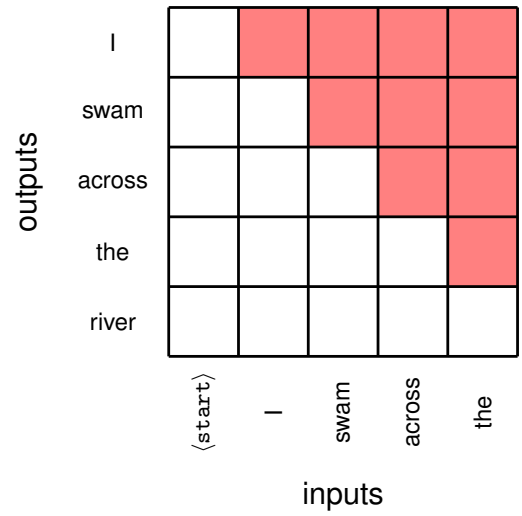


Figure 12.16 An illustration of the mask matrix for masked self-attention. Attention weights corresponding to the red elements are set to zero. Thus, in predicting the token 'across', the output can depend only on the input tokens '⟨start⟩' 'I' and 'swam'.

tention layers, in which we set to zero all of the attention weights that correspond to a token attending to any later token in the sequence. This simply involves setting to zero all the corresponding elements of the attention matrix $\mathrm{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V})$ defined by (12.14) and then normalizing the remaining elements so that each row once again sums to one. In practice, this can be achieved by setting the corresponding pre-activation values to $-\infty$ so that the softmax evaluates to zero for the associated outputs and also takes care of the normalization across the non-zero outputs. The structure of the masked attention matrix is illustrated in Figure 12.16.

In practice, we wish to make efficient use of the massive parallelism of GPUs, and hence multiple sequences may be stacked together into an input tensor for parallel processing in a single batch. However, this requires the sequences to be of the same length, whereas text sequences naturally have variable length. This can be addressed by introducing a specific token, which we denote by ⟨pad⟩, that is used to fill unused positions to bring all sequences up to the same length so that they can be combined into a single tensor. An additional mask is then used in the attention weights to ensure that the output vectors do not pay attention to any inputs occupied by the ⟨pad⟩ token. Note that the form of this mask depends on the particular input sequence.

The output of the trained model is a probability distribution over the space of tokens, given by the softmax output activation function, which represents the probability of the next token given the current token sequence. Once this next word is chosen, the token sequence with the new token included can then be fed through the model again to generate the subsequent token in the sequence, and this process can be repeated indefinitely or until an end-of-sequence token is generated. This may appear to be quite inefficient since data must be fed through the whole model for each new generated token. However, note that due to the masked attention, the embedding learned for a particular token depends only on that token itself and on earlier tokens

Telegram: @uni_k

www.konkur.in

and hence does not change when a new, later token is generated. Consequently, much of the computation can be recycled when processing a new token.

12.3.2 Sampling strategies

We have seen that the output of a decoder transformer is a probability distribution over values for the next token in the sequence, from which a particular value for that token must be chosen to extend the sequence. There are several options for selecting the value of the token based on the computed probabilities (Holtzman *et al.*, 2019). One obvious approach, called greedy search, is simply to select the token with the highest probability. This has the effect of making the model deterministic, in that a given input sequence always generates the same output sequence. Note that simply choosing the highest probability token at each stage is not the same as selecting the highest probability sequence of tokens. To find the most probable sequence, we would need to maximize the joint distribution over all tokens, which is given by

Exercise 12.15

$$p(\mathbf{y}_1, \ldots, \mathbf{y}_N) = \prod_{n=1}^{N} p(\mathbf{y}_n | \mathbf{y}_1, \ldots, \mathbf{y}_{n-1}). \tag{12.34}$$

If there are N steps in the sequence and the number of token values in the dictionary is K then the total number of sequences is $\mathcal{O}(K^N)$, which grows exponentially with the length of the sequence, and hence finding the single most probable sequence is infeasible. By comparison, greedy search has cost $\mathcal{O}(KN)$, which is linear in the sequence length.

One technique that has the potential to generate higher probability sequences than greedy search is called *beam search*. Instead of choosing the single most probable token value at each step, we maintain a set of B hypotheses, where B is called the *beam width*, each consisting of a sequence of token values up to step n. We then feed all these sequences through the network, and for each sequence we find the B most probable token values, thereby creating B^2 possible hypotheses for the extended sequence. This list is then pruned by selecting the most probable B hypotheses according to the total probability of the extended sequence. Thus, the beam search algorithm maintains B alternative sequences and keeps track of their probabilities, finally selecting the most probable sequence amongst those considered. Because the probability of a sequence is obtained by multiplying the probabilities at each step of the sequence and since these probability are always less than or equal to one, a long sequence will generally have a lower probability than a short one, biasing the results towards short sequences. For this reason the sequence probabilities are generally normalized by the corresponding lengths of the sequence before making comparisons. Beam search has cost $\mathcal{O}(BKN)$, which is again linear in the sequence length. However, the cost of generating a sequence is increased by a factor of B, and so for very large language models, where the cost of inference can become significant, this makes beam search much less attractive.

One problem with approaches such as greedy search and beam search is that they limit the diversity of potential outputs and can even cause the generation process to become stuck in a loop, where the same sub-sequence of words is repeated over and

Telegram: @uni_k

www.konkur.in

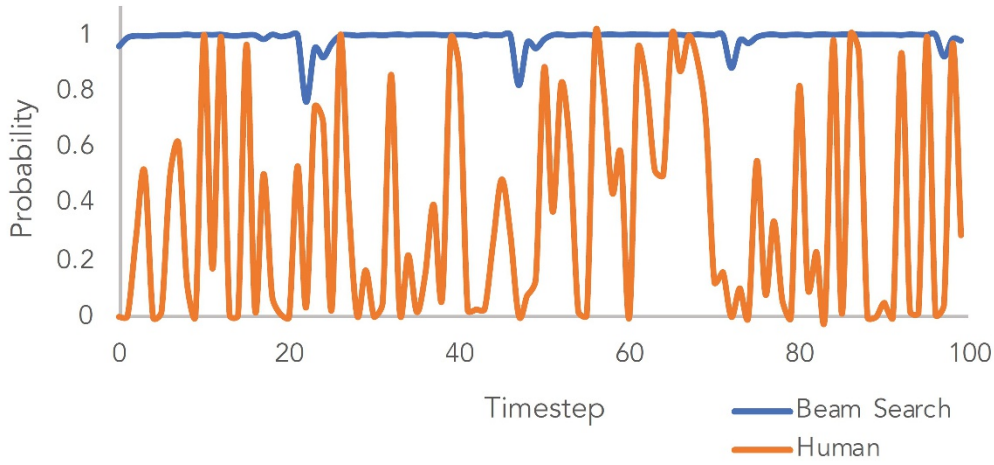


Figure 12.17 A comparison of the token probabilities from beam search and human text for a given trained transformer language model and a given initial input sequence, showing how the human sequence has much lower token probabilities. [From Holtzman *et al.* (2019) with permission.]

over. As can be seen in Figure 12.17, human-generated text may have lower probability and hence be more surprising with respect to a given model than automatically generated text.

Instead of trying to find a sequence with the highest probability, we can instead generate successive tokens simply by sampling from the softmax distribution at each step. However, this can lead to sequences that are nonsensical. This arises from the typically very large size of the token dictionary, in which there is a long tail of many token states each of which has a very small probability but which in aggregate account for a significant fraction of the total probability mass. This leads to the problem in which there is a significant chance that the system will make a bad choice for the next token.

As a balance between these extremes, we can consider only the states having the top K probabilities, for some choice of K, and then sample from these according to their renormalized probabilities. A variant of this approach, called top-p sampling or *nucleus sampling*, calculates the cumulative probability of the top outputs until a threshold is reached and then samples from this restricted set of token states.

A 'softer' version of top-K sampling is to introduce a parameter T called *temperature* into the definition of the softmax function (Hinton, Vinyals, and Dean, 2015) so that

$$y_i = \frac{\exp(a_i/T)}{\sum_j \exp(a_j/T)} \tag{12.35}$$

and then sample the next token from this modified distribution. When $T = 0$, the probability mass is concentrated on the most probable state, with all other states having zero probability, and hence this becomes greedy selection. For $T = 1$, we

Telegram: @uni_k

www.konkur.in

recover the unmodified softmax distribution, and as $T \to \infty$, the distribution becomes uniform across all states. By choosing a value in the range $0 < T < 1$, the probability is concentrated towards the higher values.

One challenge with sequence generation is that during the learning phase, the model is trained on a human-generated input sequence, whereas when it is running generatively, the input sequence is itself generated from the model. This means that the model can drift away from the distribution of sequences seen during training.

12.3.3 Encoder transformers

We next consider transformer language models based on encoders, which are models that take sequences as input and produce fixed-length vectors, such as class labels, as output. An example of such a model is *BERT*, which stands for *bidirectional encoder representations from transformers* (Devlin *et al.*, 2018). The goal is to pre-train a language model using a large corpus of text and then to fine-tune the model using *transfer learning* for a broad range of downstream tasks each of which requires a smaller application-specific training data set. The architecture of an encoder transformer is illustrated in Figure 12.18. This approach is a straightforward application of the transformer layers discussed previously.

Section 12.1.7

The first token of every input string is given by a special token ⟨`class`⟩, and the corresponding output of the model is ignored during pre-training. Its role will become apparent when we discuss fine-tuning. The model is pre-trained by presenting token sequences at the input. A randomly chosen subset of the tokens, say 15%, are replaced with a special token denoted ⟨`mask`⟩. The model is trained to predict the missing tokens at the corresponding output nodes. This is analogous to the masking used in word2vec to learn word embeddings. For example, an input sequence might be

Section 12.2.1

I ⟨`mask`⟩ across the river to get to the ⟨`mask`⟩ bank.

and the network should predict ‘swam’ at output node 2 and ‘other’ at output node 10. In this case only two of the outputs contribute to the error function and the other outputs are ignored.

The term ‘bidirectional’ refers to the fact that the network sees words both before and after the masked word and can use both sources of information to make a prediction. As a consequence, unlike decoder models, there is no need to shift the inputs to the right by one place, and there is no need to mask the outputs of each layer from seeing input tokens occurring later in the sequence. Compared to the decoder model, an encoder is less efficient since only a fraction of the sequence tokens are used as training labels. Moreover, an encoder model is unable to generate sequences.

The procedure of replacing randomly selected tokens with ⟨`mask`⟩ means the training set has a mismatch compared to subsequent fine-tuning sets in that the latter will not contain any ⟨`mask`⟩ tokens. To mitigate any problems this might cause, Devlin *et al.* (2018) modified the procedure slightly, so that of the 15% of randomly selected tokens, 80% are replaced with ⟨`mask`⟩, 10% are replaced with a word selected at random from the vocabulary, and in 10% of the cases, the original words are retained at the input, but they still have to be correctly predicted at the output.

Telegram: @uni_k

www.konkur.in

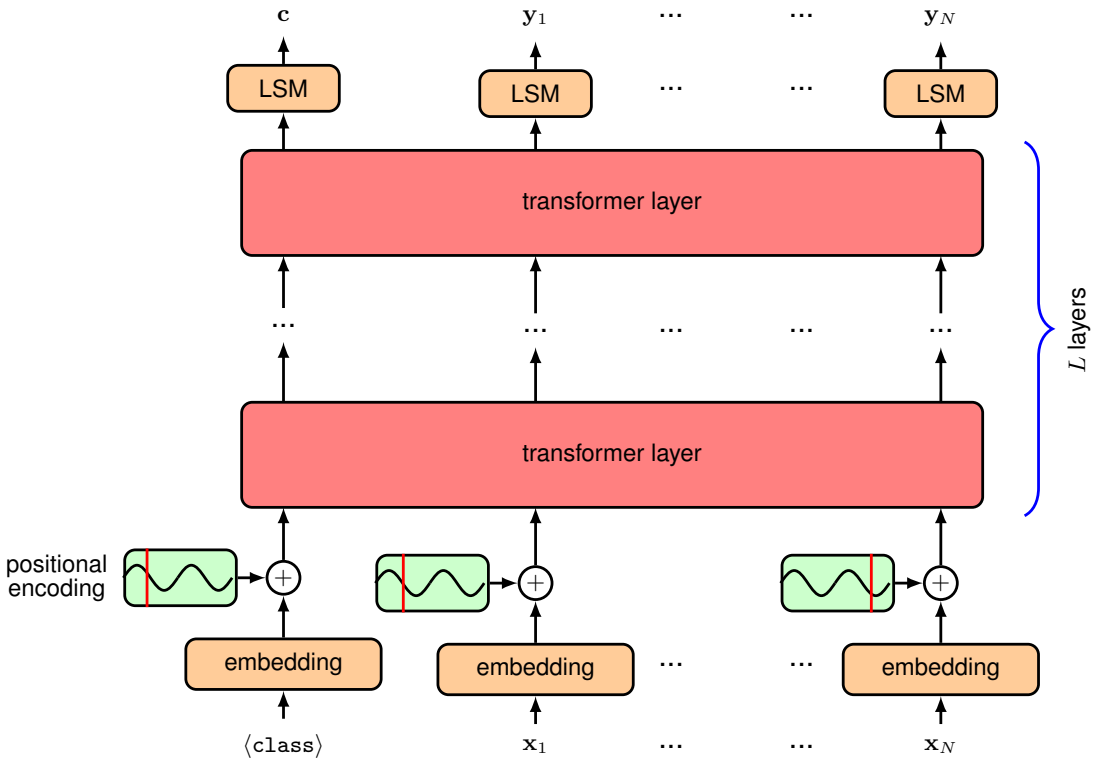


Figure 12.18 Architecture of an encoder transformer model. The boxes labelled 'LSM' denote a linear transformation whose learnable parameters are shared across the token positions, followed by a softmax activation function. The main differences compared to the decoder model are that the input sequence is not shifted to the right, and the 'look ahead' masking matrix is omitted and therefore, within each self-attention layer, every output token can attend to any of the input tokens.

Once the encoder model is trained it can then be fine-tuned for a variety of different tasks. To do this a new output layer is constructed whose form is specific to the task being solved. For a text classification task, only the first output position is used, which corresponds to the ⟨class⟩ token that always appears in the first position of the input sequence. If this output has dimension D then a matrix of parameters of dimension $D \times K$, where K is the number of classes, is appended to the first output node and this in turn feeds into a K-dimensional softmax function or a vector of dimension $D \times 1$ followed by a logistic sigmoid for $K = 2$. The linear output transformation could alternatively be replaced with a more complex differentiable model such as an MLP. If the goal is to classify each token of the input string, for example to assign each token to a category (such as person, place, colour, etc) then the first output is ignored and the subsequent outputs have a shared linear-plus-softmax layer. During fine-tuning all model parameters including the new output matrix are learned by stochastic gradient descent using the log probability

Telegram: @uni_k

www.konkur.in

of the correct label. Alternatively the output of a pre-trained model might feed into a sophisticated generative deep learning model for applications such as text-to-image synthesis.

Chapter 20

12.3.4 Sequence-to-sequence transformers

For completeness, we discuss briefly the third category of transformer model, which combines an encoder with a decoder, as discussed in the original transformer paper of Vaswani *et al.* (2017). Consider the task of translating an English sentence into a Dutch sentence. We can use a decoder model to generate the token sequence corresponding to the Dutch output, token by token, as discussed previously. The main difference is that this output needs to be conditioned on the entire input sequence corresponding to the English sentence. An encoder transformer can be used to map the input token sequence into a suitable internal representation, which we denote by $\mathbf{Z}$. To incorporate $\mathbf{Z}$ into the generative process for the output sequence, we use a modified form of the attention mechanism called *cross attention*. This is the same as self-attention except that although the query vectors come from the sequence being generated, in this case the Dutch output sequence, the key and value vectors come from the sequence represented by $\mathbf{Z}$, as illustrated in Figure 12.19. Returning to our analogy with a video streaming service, this would be like the user sending their query vector to a different streaming company who then compares it with their own set of key vectors to find the best match and then returns the associated value vector in the form of a movie.

Section 12.3.1

When we combine the encoder and decoder modules, we obtain the architecture of the model shown in Figure 12.20. The model can be trained using paired input and output sentences.

12.3.5 Large language models

The most important recent development in the field of machine learning has been the creation of very large transformer-based neural networks for natural language processing, known as *large language models* or LLMs. Here 'large' refers to the number of weight and bias parameters in the network, which can number up to around one trillion (10^{12}) at the time of writing. Such models are expensive to train, and the motivation for building them comes from their extraordinary capabilities.

In addition to the availability of large data sets, the training of ever larger models has been facilitated by the advent of massively parallel training hardware based on GPUs (graphics processing units) and similar processors tightly coupled in large clusters equipped fast interconnect and lots of onboard memory. The transformer architecture has played a key role in the development of these models because it is able to make very efficient use of such hardware. Very often, increasing the size of the training data set, along with a commensurate increase in the number of model parameters, leads to improvements in performance that outpace architectural improvements or other ways to incorporate more domain knowledge (Sutton, 2019; Kaplan *et al.*, 2020). For example, the impressive increase in performance of the GPT series of models (Radford *et al.*, 2019; Brown *et al.*, 2020; OpenAI, 2023) through successive generations has come primarily from an increase in scale. These kinds

Telegram: @uni_k

www.konkur.in

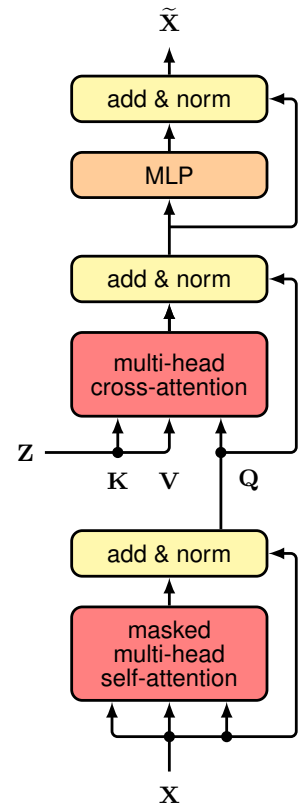


Figure 12.19 Schematic illustration of one cross-attention layer as used in the decoder section of a sequence-to-sequence transformer. Here $\mathbf{Z}$ denotes the output from the encoder section. $\mathbf{Z}$ determines the key and value vectors for the cross-attention layer, whereas the query vectors are determined within the decoder section.

of performance improvements have driven a new kind of Moore's law in which the number of compute operations required to train a state-of-the-art machine learning model has grown exponentially since about 2012 with a doubling time of around 3.4 months.

Figure 1.16

Early language models were trained using supervised learning. For example, to build a translation system, the training set would consist of matched pairs of sentences in two languages. A major limitation of supervised learning, however, is that the data typically has to be human-curated to provide labelled examples, and this severely limits the quantity of data available, thereby requiring heavy use of inductive biases such as feature engineering and architecture constraints to achieve reasonable performance.

Large language models are trained instead by self-supervised learning on very large data sets of text, along with potentially other token sequences such as computer code. We have seen how a decoder transformer can be trained on token sequences in which each token acts as a labelled target example, with the preceding sequence as input, to learn a conditional probability distribution. This 'self-labelling' hugely expands the quantity of training data available and therefore allows exploitation of deep neural networks having large numbers of parameters.

Section 12.3.1

Telegram: @uni_k

www.konkur.in

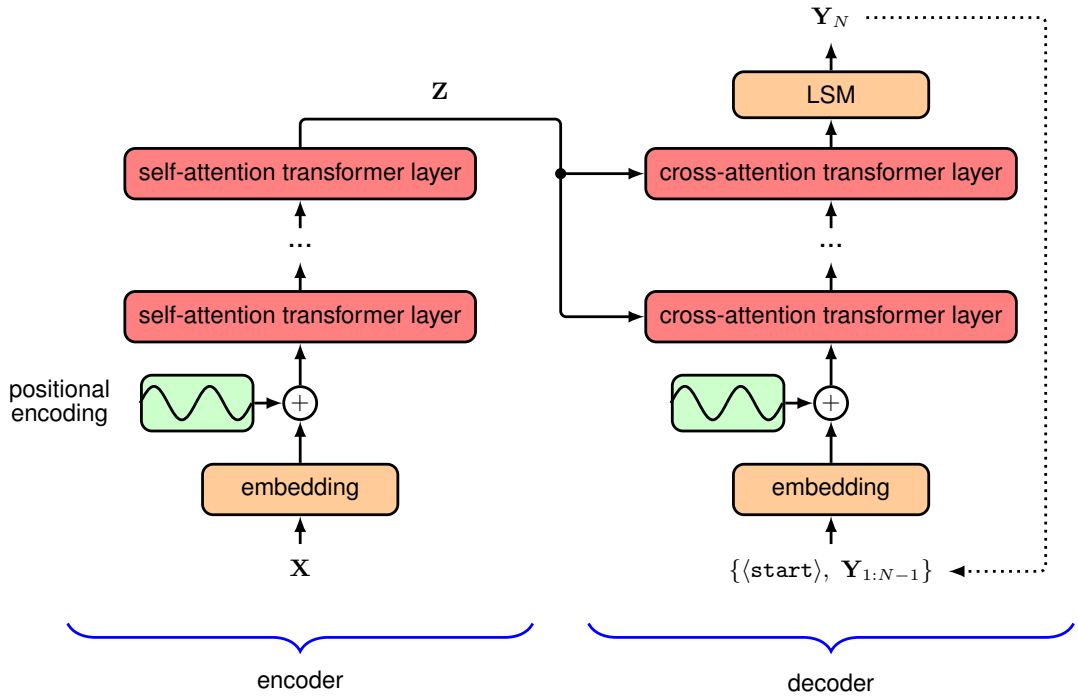


Figure 12.20 Schematic illustration of a sequence-to-sequence transformer. To keep the diagram uncluttered the input tokens are collectively shown as a single box, and likewise for the output tokens. Positional-encoding vectors are added to the input tokens for both the encoder and decoder sections. Each layer in the encoder corresponds to the structure shown in Figure 12.9, and each cross-attention layer is of the form shown in Figure 12.19.

This use of self-supervised learning led to a paradigm shift in which a large model is first *pre-trained* using unlabelled data and then subsequently *fine-tuned* using supervised learning based on a much smaller set of labelled data. This is effectively a form of transfer learning, and the same pre-trained model can be used for multiple 'downstream' applications. A model with broad capabilities that can be subsequently fine-tuned for specific tasks is called a *foundation model* (Bommasani *et al.*, 2021).

The fine-tuning can be done by adding extra layers to the outputs of the network or by replacing the last few layers with fresh parameters and then using the labelled data to train these final layers. During the fine-tuning stage, the weights and biases in the main model can either be left unchanged or be allowed to undergo small levels of adaptation. Typically the cost of the fine-tuning is small compared to that of pre-training.

One very efficient approach to fine-tuning is called *low-rank adaptation* or LoRA (Hu *et al.*, 2021). This approach is inspired by results which show that a trained over-parameterized model has a low intrinsic dimensionality with respect to fine-tuning, meaning that changes in the model parameters during fine-tuning lie on a manifold

Telegram: @uni_k

www.konkur.in

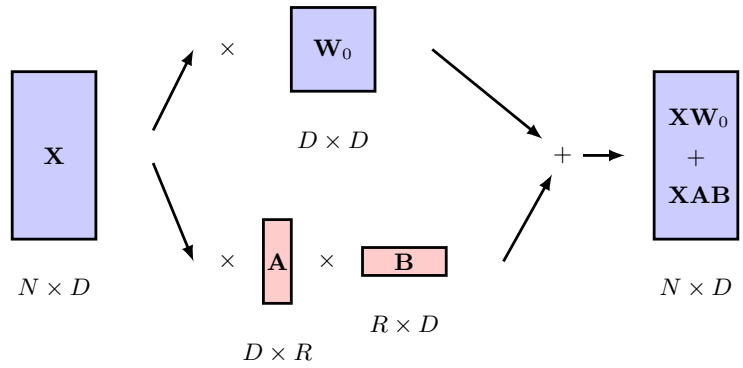


Figure 12.21 Schematic illustration low-rank adaptation showing a weight matrix $\mathbf{W}_0$ from one of the attention layers in a pre-trained transformer. Additional weights given by matrices $\mathbf{A}$ and $\mathbf{B}$ are adapted during fine-tuning and their product $\mathbf{AB}$ is then added to the original matrix for subsequent inference.

whose dimensionality is much smaller than the total number of learnable parameters in the model (Aghajanyan, Zettlemoyer, and Gupta, 2020). LoRa exploits this by freezing the weights of the original model and adding additional learnable weight matrices into each layer of the transformer in the form of low-rank products. Typically only attention-layer weights are modified, whereas MLP-layer weights are kept fixed. Consider a weight matrix $\mathbf{W}_0$ having dimension $D \times D$, which might represent a query, key, or value matrix in which the matrices from multiple attention heads are treated together as a single matrix. We introduce a parallel set of weights defined by the product of two matrices $\mathbf{A}$ and $\mathbf{B}$ with dimensions $D \times R$ and $R \times D$, respectively, as shown schematically in Figure 12.21. This layer then generates an output given by $\mathbf{XW}_0 + \mathbf{XAB}$. The number of parameters in the additional weight matrix $\mathbf{AB}$ is $2RD$ compared to the D^2 parameters in the original weight matrix $\mathbf{W}_0$, and so if $R \ll D$ then the number of parameters that need to be adapted during fine-tuning is much smaller than the number in the original transformer. In practice, this can reduce the number of parameters that need to be trained by a factor of 10,000. Once the fine-tuning is complete, the additional weights can be added to the original weight matrices to give a new weight matrix

$$\widehat{\mathbf{W}} = \mathbf{W}_0 + \mathbf{AB} \tag{12.36}$$

so that during inference there is no additional computational overhead compared to running the original model since the updated model has the same size as the original.

As language models have become larger and more powerful, the need for fine-tuning has diminished, with generative language models now able to solve a broad range of tasks simply through text-based interaction. For example, if a text string

English: the cat sat on the mat. French:

is given as the input sequence, an autoregressive language model can continue to generate subsequent tokens until a ⟨stop⟩ token is generated, in which the newly gen-

Telegram: @uni_k

www.konkur.in

erated tokens represent the French translation. Note that the model was not trained specifically to do translation but has learned to do so as a result of being trained on a vast corpus of data that includes multiple languages.

A user can interact with such models using a natural language dialogue, making them very accessible to broad audiences. To improve the user experience and the quality of the generated outputs, techniques have been developed for fine-tuning large language models through human evaluation of generated output, using methods such as *reinforcement learning through human feedback* or RLHF (Christiano *et al.*, 2017). Such techniques have helped to create large language models with impressively easy-to-use conversational interfaces, most notably the system from OpenAI called *ChatGPT*.

The sequence of input tokens given by the user is called a *prompt*. For example, it might consist of the opening words of a story, which the model is required to complete. Or it might comprise a question, and the model should provide the answer. By using different prompts, the same trained neural network may be capable of solving a broad range of tasks such as generating computer code from a simple text request or writing rhyming poetry on demand. The performance of the model now depends on the form of the prompt, leading to a new field called *prompt engineering* (Liu *et al.*, 2021), which aims to design a good form for a prompt that results in high-quality output for the downstream task. The behaviour of the model can also be modified by adapting the user's prompt before feeding it into the language model by pre-pending an additional token sequence called a *prefix prompt* to the user prompt to modify the form of the output. For example, the pre-prompt might consist of instructions, expressed in standard English, to tell the network not to include offensive language in its output.

This allows the model to solve new tasks simply by providing some examples within the prompt, without needing to adapt the parameters of the model. This is an example of *few-shot learning*.

Current state-of-the-art models such as GPT-4 have become so powerful that they are exhibiting remarkable properties which have been described as the first indications of artificial general intelligence (Bubeck *et al.*, 2023) and are driving a new wave of technological innovation. Moreover, the capabilities of these models continue to improve at an impressive pace.

12.4. Multimodal Transformers

Although transformers were initially developed as an alternative to recurrent networks for processing sequential language data, they have become prevalent in nearly all areas of deep learning. They have proved to be general-purpose models, as they make very few assumptions about the input data, in contrast, for example, to convolutional networks, which make strong assumptions about equivariances and locality. Due to their generality, transformers have become the state-of-the-art for many different modalities, including text, image, video, point cloud, and audio data, and have been used for both discriminative and generative applications within each of these

Chapter 10

Telegram: @uni_k

www.konkur.in

domains. The core architecture of the transformer layer has remained relatively constant, both over time and across applications. Therefore, the key innovations that enabled the use of transformers in areas other than natural language have largely focused on the representation and encoding of the inputs and outputs.

One big advantage of a single architecture that is capable of processing many different kinds of data is that it makes *multimodal* computation relatively straightforward. In this context, multimodal refers to applications that combine two or more different types of of data, either in the inputs or outputs or both. For example, we may wish to generate an image from a text prompt or design a robot that can combine information from multiple sensors such as cameras, radar, and microphones. The important thing to note is that if we can tokenize the inputs and decode the output tokens, then it is likely that we can use a transformer.

12.4.1 Vision transformers

Transformers have been applied with great success to computer vision and have achieved state-of-the-art performance on many tasks. The most common choice for discriminative tasks is a standard transformer encoder, and this approach in the vision domain is known as a *vision transformer*, or ViT (Dosovitskiy *et al.*, 2020). When using a transformer, we need to decide how to convert an input image into tokens, and the simplest choice is to use each pixel as a token, following a linear projection. However, the memory required by a standard transformer implementation grows quadratically with the number of input tokens, and so this approach is generally infeasible. Instead, the most common approach to tokenization is to split the image into a set of patches of the same size. Suppose the images have dimension $\mathbf{x} \in \mathbb{R}^{H \times W \times C}$ where H and W are the height and width of the image in pixels and C is the number of channels (where typically $C = 3$ for R, G, and B colours). Each image is split into non-overlapping patches of size $P \times P$ (where $P = 16$ is a common choice) and then 'flattened' into a one-dimensional vector, which gives a representation $\mathbf{x}_p \in \mathbb{R}^{N \times (P^2 C)}$ where $N = HW/P^2$ is the total number of patches for one image. The ViT architecture is shown in Figure 12.22.

Another approach to tokenization is to feed the image through a small convolutional neural network (CNN). This can down-sample the image to give a manageable number of tokens each represented by one of the network outputs. For example a typical ResNet18 encoder architecture down-samples an image by a factor of 8 in both the height and width dimensions, giving 64 times fewer tokens than pixels.

Chapter 10

We also need a way to encode positional information in the tokens. It is possible to construct explicit positional embeddings that encode the two-dimensional positional information of the image patches, but in practice this does not generally improve performance, and so it is most common to just use learned positional embeddings. In contrast to the transformers used for natural language, vision transformers generally take a fixed number of tokens as input, which avoids the problem of learned positional encodings not generalizing to inputs of a different size.

A vision transformer has a very different architectural design compared to a CNN. Although strong inductive biases are baked into a CNN model, the only two-dimensional inductive bias in a vision transformer is due to the patches used to tok-

Telegram: @uni_k

www.konkur.in

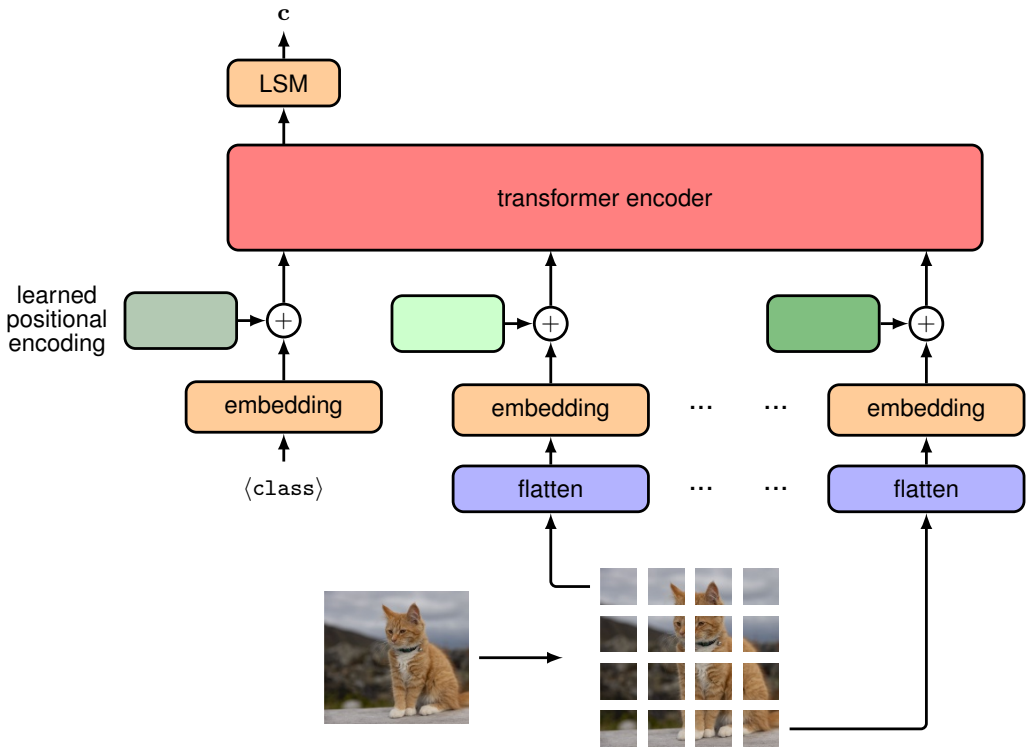


Figure 12.22 Illustration of the vision transformer architecture for a classification task. Here a learnable ⟨class⟩ token is included as an additional input, and the associated output is transformed by a linear layer with a softmax activation, denoted by LSM, to give the final class-vector output $\mathbf{c}$.

enize the input. A transformer therefore generally requires more training data than a comparable CNN as it has to learn the geometrical properties of images from scratch. However, because there are no strong assumptions about the structure of the inputs, transformers are often able to converge to a higher accuracy. This provides another illustration of the trade-off between inductive bias and the scale of the training data (Sutton, 2019).

12.4.2 Generative image transformers

In the language domain, the most impressive results have come when transformers are used as an autoregressive generative model for synthesizing text. It is therefore natural to ask whether we can also use transformers to synthesize realistic images. Since natural language is inherently sequential, it fits neatly into the autoregressive framework, whereas images have no natural ordering of their pixels so that it is not as intuitive that decoding them autoregressively would be useful. However, any distribution can be decomposed into a product of conditionals, provided we first define some ordering of the variables. Thus, the joint distribution over ordered

Section 11.1.2

Telegram: @uni_k

www.konkur.in

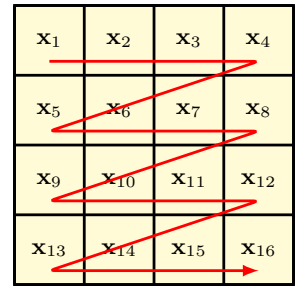


Figure 12.23 Illustration of a *raster scan* that defines a specific linear ordering of the pixels in a two-dimensional image.

variables $\mathbf{x}_1, \ldots, \mathbf{x}_N$ can be written

$$p(\mathbf{x}_1, \ldots, \mathbf{x}_N) = \prod_{n=1}^{N} p(\mathbf{x}_n | \mathbf{x}_1, \ldots, \mathbf{x}_{n-1}). \tag{12.37}$$

This factorization is completely general and makes no restrictions on the form of the individual conditional distributions $p(\mathbf{x}_n|\mathbf{x}_1, \ldots, \mathbf{x}_{n-1})$.

For an image we can choose $\mathbf{x}_n$ to represent the nth pixel as a three-dimensional vector of the RGB values. We now need to decide on an ordering for the pixels, and one widely used choice is called a *raster scan* as illustrated in Figure 12.23. A schematic illustration of an image being generated using an autoregressive model, based on a raster-scan ordering, is shown in Figure 12.24.

Note that the use of autoregressive generative models of images predates the introduction of transformers. For example, PixelCNN (Oord *et al.*, 2016) and PixelRNN (Oord, Kalchbrenner, and Kavukcuoglu, 2016) used bespoke masked convolution layers that preserve the conditional independence defined for each pixel by the corresponding term on the right-hand side of 12.37.

Representations of an image using continuous values can work well in discriminative tasks. However, much better results are obtained for image generation by using discrete representations. Continuous conditional distributions learned by maximum likelihood, such as Gaussians for which the negative log likelihood function is a sum-of-squares error function, tend to learn averages of the training data, leading to blurry images. Conversely, discrete distributions can handle multimodality with ease. For example, one of the conditional distributions $p(\mathbf{x}_n|\mathbf{x}_1, \ldots, \mathbf{x}_{n-1})$ in

Section 4.2

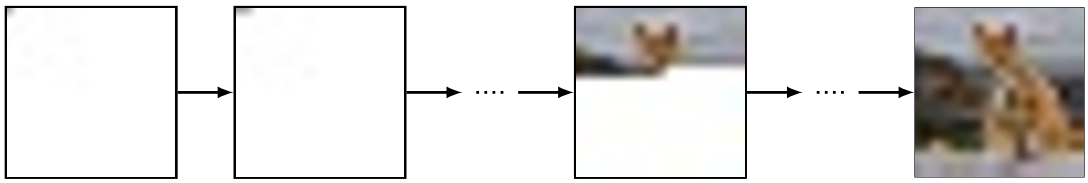


Figure 12.24 An illustration of how an image can be sampled from an autoregressive model. The first pixel is sampled from the marginal distribution $p(\mathbf{x}_{11})$, the second pixel from the conditional distribution $p(\mathbf{x}_{12}|\mathbf{x}_{11})$, and so on in raster scan order until we have a complete image.

Telegram: @uni_k

www.konkur.in

(12.37) might learn that a pixel could be either black or white, whereas a regression model might learn that the pixel should be grey.

However, working with discrete spaces also brings its challenges. The R, G, and B values of image pixels are typically represented with at least 8 bits of precision, so that each pixel has $2^{24} \simeq 16\text{M}$ possible values. Learning a conditional softmax distribution over a such a high-dimensional space is infeasible.

One way to address the problem of the high dimensionality is to use the technique of *vector quantization*, which can be viewed as a form of data compression. Suppose we have a set of data vectors $\mathbf{x}_1, \ldots, \mathbf{x}_N$ each of dimensionality D, which might, for example, represent image pixels, and we then introduce a set of K *codebook vectors* $\mathcal{C} = \mathbf{c}_1, \ldots, \mathbf{c}_K$ also of dimensionality D, where typically $K \ll D$. We now approximate each data vector by its nearest codebook vector according to some similarity metric, usually Euclidean distance, so that

Section 15.1.1

$$\mathbf{x}_n \rightarrow \arg\min_{\mathbf{c}_k \in \mathcal{C}} ||\mathbf{x}_n - \mathbf{c}_k||^2. \tag{12.38}$$

Since there are K codebook vectors, we can represent each $\mathbf{x}_n$ by a one-hot encoded K-dimensional vector, and since we can choose the value of K, we can control the trade-off between more accurate representation of the data, by using a larger value of K, or greater compression, by using a smaller value of K.

We can therefore take the original image pixels and map them into the lower-dimensional codebook space. An autoregressive transformer can then be trained to generate a sequence of codebook vectors, and this sequence can be mapped back into the original image space by replacing each codebook index k with the corresponding D-dimensional codebook vector $\mathbf{c}_k$.

Autoregressive transformers were first applied to images in ImageGPT (Chen, Radford, *et al.*, 2020). Here each pixel is treated as one of a discrete set of three-dimensional colour codebook vectors, each corresponding to a cluster in a K-means clustering of the colour space. A one-hot encoding therefore gives discrete tokens, analogous to language tokens, and allows the transformer to be trained in the same way as language models, with a next-token classification objective. This is a powerful objective for representation learning for subsequent fine-tuning, again in a similar way to language modelling.

Section 15.1

Section 6.3.3

Using the individual pixels as tokens directly, however, can lead to high computational cost since a forward pass is required per pixel, which means that both training and inference scale poorly with image resolution. Also, using individual pixels as inputs means that low-resolution images have to be used to give a reasonable context length when decoding the pixels later in the raster scan. As we saw with the ViT model, it is preferable to use patches of the image as tokens instead of pixels, as this can result in dramatically fewer tokens and therefore facilitates working with higher-resolution images. As before, we need to work with a discrete space of token values due to the potential multimodality of the conditional distributions. Again, this raises the challenge of dimensionality, which is now much more severe with patches than with individual pixels since the dimensionality is exponential with respect to the number of pixels in the patch. For example, even with just two possible pixel

Telegram: @uni_k

www.konkur.in

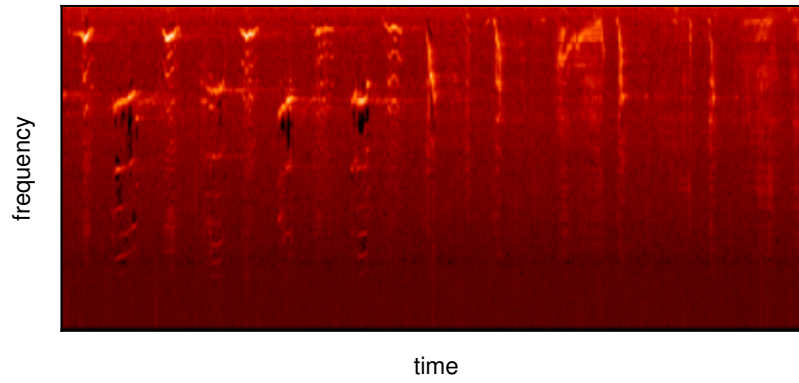


Figure 12.25 An example mel spectrogram of a humpback whale song. [Source data copyright ©2013–2023, librosa development team.]

tokens, representing black and white, and patches of size 16×16, we would have a dictionary of patch tokens of size $2^{256} \simeq 10^{77}$.

Once again we turn to vector quantization to address the challenge of dimensionality. The codebook vectors can be learned from a data set of image patches using simple clustering algorithms such as K-means or with more sophisticated methods such as fully convolutional networks (Oord, Vinyals, and Kavukcuoglu, 2017; Esser, Rombach, and Ommer, 2020) or even vision transformers (Yu *et al.*, 2021). One problem with learning to map each patch to a discrete set of codes and back again, is that vector quantization is a non-differentiable operation. Fortunately we can use a technique called straight-through gradient estimation (Bengio, Léonard, and Courville, 2013), which is a simple approximation that just copies the gradients through the non-differentiable function during backpropagation.

The use of autoregressive transformers to generate images can be extended to videos by treating a video as one long sequence of these vector-quantized tokens (Rakhimov *et al.*, 2020; Yan *et al.*, 2021; Hu *et al.*, 2023).

12.4.3 Audio data

We next look at the application of transformers to audio data. Sound is generally stored as a waveform obtained by measuring the amplitude of the air pressure at regular time intervals. Although this waveform could be used directly as input to a deep learning model, in practice it is more effective to pre-process it into a *mel spectrogram*. This is a matrix whose columns represent time steps and whose rows correspond to frequencies. The frequency bands follow a standard convention that was chosen through subjective assessment to give equal perceptual differences between successive frequencies (the word 'mel' comes from melody). An example of a mel spectrogram is shown in Figure 12.25.

One application for transformers in the audio domain is classification in which segments of audio are assigned to one of a number of predefined categories. For example, the *AudioSet* data set (Gemmeke *et al.*, 2017) is a widely used benchmark.

Telegram: @uni_k

www.konkur.in

It contains classes such as 'car', 'animal', and 'laughter'. Until the development of the transformer, the state-of-the-art approach for audio classification was based on mel spectrograms treated as images and used as the input to a convolutional neural network (CNN). However, although a CNN is good at understanding local relationships, one drawback is that it struggles with longer-range dependencies, which can be important in processing audio.

Chapter 10

Just as transformers replaced RNNs as the state-of-the-art in natural language processing, they have also come to replace CNNs for tasks such as audio classification. For example, a transformer encoder model of identical structure to that used for both language and vision, as shown in Figure 12.18, can be used to predict the class of audio inputs (Gong, Chung, and Glass, 2021). Here the mel spectrogram is viewed as an image which is then tokenized. This is done by splitting the image into patches in a similar way to vision transformers, possibly with some overlap so as not to lose any important neighbourhood relations. Each patch is then flattened, meaning it is converted to a one-dimensional array, in this case of length 256. A unique positional encoding is then added to each token, a specific ⟨`class`⟩ token is appended, and the tokens are then fed through the transformer encoder. The output token corresponding to the ⟨`class`⟩ input token from the last transformer layer can then be decoded using a linear layer followed by a softmax activation function, and the whole model can be trained end-to-end using a cross-entropy loss.

12.4.4 Text-to-speech

Classification is not the only task that deep learning, and more specifically the transformer architecture, has revolutionized in the audio domain. The success of transformers at synthesizing speech that imitates the voice of a given speaker is another demonstration of their versatility, and their application to this task is an informative case study in how to apply transformers in a new context.

Generating speech corresponding to a given passage of text is known as *text-to-speech synthesis*. A more traditional approach would be to collect recordings of speech from a given speaker and train a supervised regression model to predict the speech output, possibly in the form of a mel spectrogram, from corresponding transcribed text. During inference, the text for which we would like to synthesize speech is presented as input and the resulting mel spectrogram output can then be decoded back to an audio waveform since this is a fixed mapping.

This approach has a few major drawbacks, however. First, if we predict speech at a low level, for example using sub-word components known as *phonemes*, a larger context is needed to make the resulting sentences sound fluid. However, if we predict longer segments, then the space of possible inputs grows significantly, and an infeasible amount of training data might be required to achieve good generalization. Second, this approach does not transfer knowledge across speakers, and so a lot of data will be required for each new speaker. Finally, the problem is really a generative modelling task, as there are multiple correct speech outputs for a given speaker and text pair, so regression may not be suitable since it tends to average over target values.

Section 4.2

If instead we treat audio data in the same way as natural language and frame text-to-speech as a conditional language modelling task, then we should be able to

Telegram: @uni_k

www.konkur.in

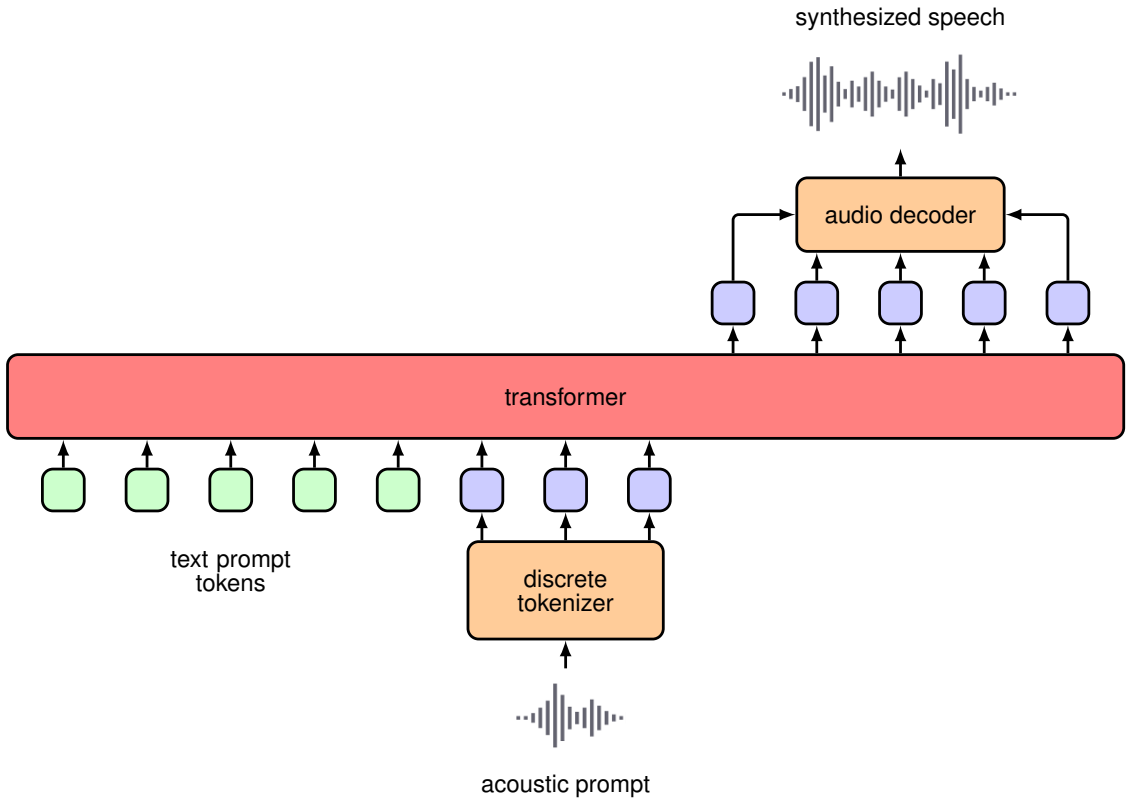


Figure 12.26 A diagram showing the high-level architecture of Vall-E. The input to the transformer model consists of standard text tokens, which prompt the model as to what words the synthesized speech should contain, together with acoustic prompt tokens that determine the speaker style and tone information. The sampled model output tokens are decoded back to speech with the learned decoder. For simplicity, the positional encodings and linear projections are not shown.

train the model in much the same way as with text-based large language models. There are two main implementation details that need to be addressed. The first is how to tokenize the training data and decode the predictions, and the second is how to condition the model on the speaker's voice.

One approach to text-to-speech synthesis that makes use of transformers and language modelling techniques is *Vall-E* (Wang *et al.*, 2023). New text can be mapped into speech in the voice of a new speaker using only a few seconds of sample speech from that person. Speech data is converted into a sequence of discrete tokens from a learned dictionary or *codebook* obtained using vector quantization, and we can think of these tokens as analogous to the one-hot encoded tokens from the natural language domain. The input consists of text tokens from a passage of text whereas the target outputs for training consist of the corresponding speech tokens. Additional speech tokens from a short segment of unrelated speech from the same speaker are

Section 12.4.2

Telegram: @uni_k

www.konkur.in

appended to the input text tokens, as illustrated in Figure 12.26. By including examples from many different speakers, the system can learn to read out a passage of text while imitating the voice represented by the additional speech input tokens. Once trained the system can be presented with new text, along with audio tokens from a brief segment of speech captured from a new speaker, and the resulting output tokens can be decoded, using the same codebook used during training, to create a speech waveform. This allows the system to synthesize speech corresponding to the input text in the voice of the new speaker.

12.4.5 Vision and language transformers

We have seen how to generate discrete tokens for text, audio, and images, and so it is a natural next step to ask if we can train a model with input tokens of one modality and output tokens of another, or whether we can have a combination of different modalities for either inputs or outputs or both. We will focus on the combination of text and vision data as this is the most widely studied example, but in principle the approaches discussed here could be applied to other combinations of input and output modalities.

The first requirement is that we have a large data set for training. The LAION-400M data set (Schuhmann *et al.*, 2021) has greatly accelerated research in text-to-image generation and image-to-text captioning in much the same way that ImageNet was critical in the development of deep image classification models. Text-to-image generation is actually much like the unconditional image generation we have looked at so far, except that we also allow the model to take as input the text information to condition the generation process. This is straightforward when using transformers as we can simply provide the text tokens as additional input when decoding each image token.

This approach can also be viewed as treating the text-to-image problem as a sequence-to-sequence language modelling problem, such as machine translation, except that the target tokens are discrete image tokens rather than language tokens. It therefore makes sense to choose a full encoder-decoder transformer model, as shown in Figure 12.20, in which $\mathbf{X}$ corresponds to the input text tokens and $\mathbf{Y}$ corresponds to the output image tokens. This is the approach taken in a model called *Parti* (Yu *et al.*, 2022) in which the transformer is scaled to 20 billion parameters while showing consistent performance improvements with increasing model size.

A lot of research has also been done on using pre-trained language models, and modifying or fine-tuning them so that they can also accept visual data as input (Alayrac *et al.*, 2022; Li *et al.*, 2022). These approaches largely use bespoke architectures, along with continuous-valued image tokens, and therefore are not natural fits for also generating visual data. Moreover, they cannot be used directly if we wish to include new modalities such as audio tokens. Although this is a step towards multimodality, we would ideally like to use both text and image tokens as both input and output. The simplest approach is to treat everything as a sequence of tokens as if this were natural language but with a dictionary that is the concatenation of a language token dictionary and the image token codebook. We can then treat any stream of audio and visual data as simply a sequence of tokens.

Telegram: @uni_k

www.konkur.in

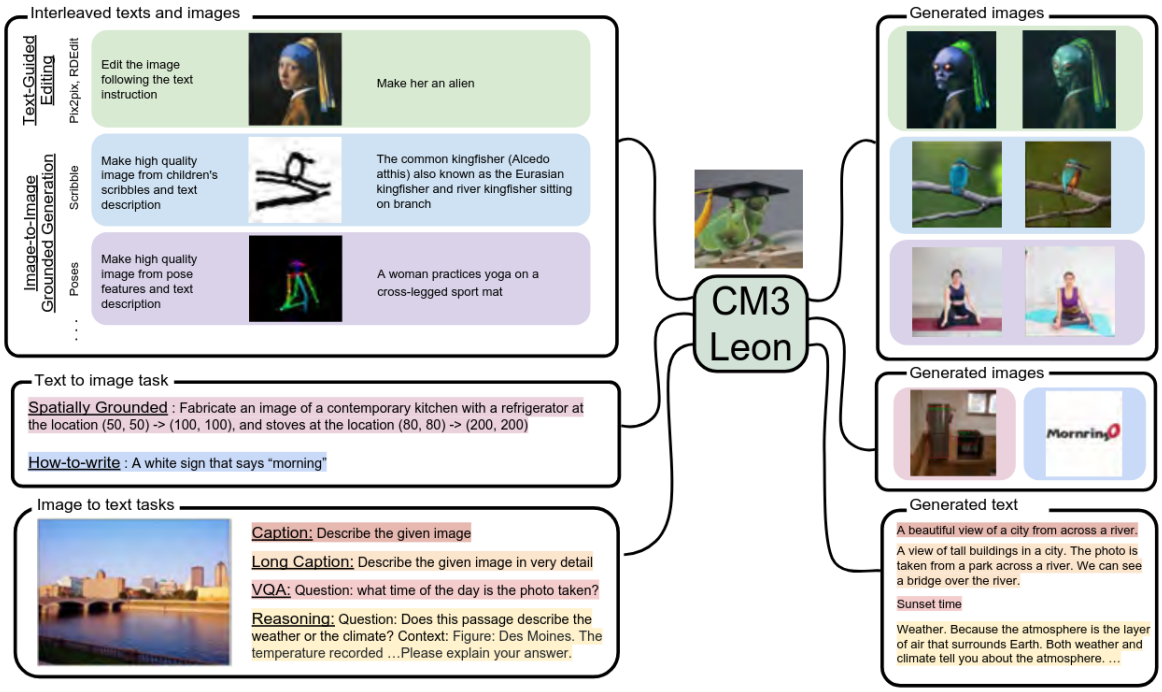


Figure 12.27 Examples of the CM3Leon model performing a variety of different tasks in the joint space of text and images. [From (Yu *et al.*, 2023) with permission.]

In CM3 (Aghajanyan *et al.*, 2022) and CM3Leon (Yu *et al.*, 2023), a variation of language modelling is used to train on HTML documents containing both image and text data taken from online sources. When this large quantity of training data was combined with a scalable architecture, the models became very powerful. Moreover, the multimodal nature of the training means that the models are very flexible. Such models are capable of completing many tasks that otherwise might require task-specific model architectures and training regimes, such as text-to-image generation, image-to-text captioning, image editing, text completion, and many more, including anything a regular language model is capable of. Examples of the CM3Leon model completing instances of a few different tasks are shown in Figure 12.27.

Exercises

12.1 ($\star\star$) Consider a set of coefficients a_{nm}, for $m = 1, \ldots, N$, with the properties that

$$a_{nm} \geqslant 0 \tag{12.39}$$

$$\sum_m a_{nm} = 1. \tag{12.40}$$

Telegram: @uni_k

www.konkur.in

Appendix C By using a Lagrange multiplier show that the coefficients must also satisfy

$$a_{nm} \leqslant 1 \quad \text{for} \quad n = 1, \ldots, N. \tag{12.41}$$

12.2 (⋆) Verify that the softmax function (12.5) satisfies the constraints (12.3) and (12.4) for any values of the vectors $\mathbf{x}_1, \ldots, \mathbf{x}_N$.

12.3 (⋆) Consider the input vectors $\mathbf{x}_n$ in the simple transformation defined by (12.2), in which the weighting coefficients a_{nm} are defined by (12.5). Show that if all the input vectors are orthogonal, so that $\mathbf{x}_n^{\mathrm{T}}\mathbf{x}_m = 0$ for $n \neq m$, then the output vectors will simply be equal to the input vectors so that $\mathbf{y}_n = \mathbf{x}_n$ for $n = 1, \ldots, N$.

12.4 (⋆) Consider two independent random vectors $\mathbf{a}$ and $\mathbf{b}$ each of dimension D and each being drawn from a Gaussian distribution with zero mean and unit variance $\mathcal{N}(\cdot|\mathbf{0}, \mathbf{I})$. Show that the expected value of $(\mathbf{a}^{\mathrm{T}}\mathbf{b})^2$ is given by D.

12.5 (⋆⋆⋆) Show that multi-head attention defined by (12.19) can be rewritten in the form

$$\mathbf{Y} = \sum_{h=1}^{H} \mathbf{H}_h \mathbf{X} \mathbf{W}^{(h)} \tag{12.42}$$

where $\mathbf{H}_h$ is given by (12.15) and we have defined

$$\mathbf{W}^{(h)} = \mathbf{W}_h^{(v)} \mathbf{W}_h^{(\mathrm{o})}. \tag{12.43}$$

Figure 12.7 Here we have partitioned the matrix $\mathbf{W}^{(\mathrm{o})}$ horizontally into sub-matrices denoted $\mathbf{W}_h^{(\mathrm{o})}$ each of dimension $D_{\mathrm{v}} \times D$, corresponding to the vertical segments of the concatenated attention matrix. Since D_{v} is typically smaller than D, for example $D_{\mathrm{v}} = D/H$ is a common choice, this combined matrix is rank deficient. Therefore, using a fully flexible matrix to replace $\mathbf{W}_h^{(\mathrm{v})} \mathbf{W}_h^{(\mathrm{o})}$ would not be equivalent to the original formulation given in the text.

12.6 (⋆⋆) Express the self-attention function (12.14) as a fully connected network in the form of a matrix that maps the full input sequence of concatenated word vectors into an output vector of the same dimension. Note that such a matrix would have $\mathcal{O}(N^2 D^2)$ parameters. Show that the self-attention network corresponds to a sparse version of this matrix with parameter sharing. Draw a sketch showing the structure of this matrix, indicating which blocks of parameters are shared and which blocks have all elements equal to zero.

12.7 (⋆) Show that if we omit the positional encoding of input vectors then the outputs of a multi-head attention layer defined by (12.19) are equivariant with respect to a reordering of the input sequence.

12.8 (⋆⋆⋆) Consider two D-dimensional unit vectors $\mathbf{a}$ and $\mathbf{b}$, satisfying $\|\mathbf{a}\| = 1$ and $\|\mathbf{b}\| = 1$, drawn from a random distribution. Assume that the distribution is symmetrical around the origin, i.e., it depends only on the distance from the origin and

Telegram: @uni_k

www.konkur.in

not the direction. Show that for large values of D the magnitude of the cosine of the angle between these vectors is close to zero and hence that these random vectors are nearly orthogonal in a high-dimensional space. To do this, consider an orthonormal basis set $\{\mathbf{u}_i\}$ where $\mathbf{u}_i^{\mathrm{T}}\mathbf{u}_j = \delta_{ij}$ and express $\mathbf{a}$ and $\mathbf{b}$ as expansions in this basis.

12.9 ($\star\star$) Consider a position encoding in which the input token vector $\mathbf{x}$ is concatenated with a position-encoding vector $\mathbf{e}$. Show that when this concatenated vector undergoes a general linear transformation by multiplication using a matrix, the result can be expressed as the sum of a linearly transformed input and a linearly transformed position vector.

12.10 ($\star\star$) Show that the positional encoding defined by (12.25) has the property that, for a fixed offset k, the encoding at position $n + k$ can be represented as a linear combination of the encoding at position n with coefficients that depend only on k and not on n. To do this make use of the following trigonometric identities:

$$\cos(A + B) = \cos A \cos B - \sin A \sin B \tag{12.44}$$

$$\sin(A + B) = \cos A \sin B + \sin A \cos B. \tag{12.45}$$

Show that if the encoding is based purely on sine functions, without cosine functions, then this property no longer holds.

12.11 ($\star$) Consider the bag-of-words model (12.28) in which each of the component distributions $p(\mathbf{x}_n)$ is given by a general probability table that is shared across all words. Show that the maximum likelihood solution, given a training set of vectors, is given by a table whose entries are the fractions of times each word occurs in the training set.

12.12 ($\star$) Consider the autoregressive language model given by (12.31) and suppose that the terms $p(\mathbf{x}_n|\mathbf{x}_1, \ldots, \mathbf{x}_{n-1})$ on the right-hand side are represented by general probability tables. Show that the number of entries in these tables grows exponentially with the value of n.

12.13 ($\star$) When using n-grams it is usual to train the n-gram and $(n-1)$-gram models at the same time and then compute the conditional probability using the product rule of probability in the form

$$p(\mathbf{x}_n|\mathbf{x}_{n-L+1}, \ldots, \mathbf{x}_{n-1}) = \frac{p_L(\mathbf{x}_{n-L+1}, \ldots, \mathbf{x}_n)}{p_{L-1}(\mathbf{x}_{n-L+1}, \ldots, \mathbf{x}_{n-1})}. \tag{12.46}$$

Explain why this is more convenient than storing the left-hand-side directly, and show that to obtain the correct probabilities the final token from each sequence must be omitted when evaluating $p_{L-1}(\cdots)$.

12.14 ($\star\star$) Write down pseudo-code for the inference process in a trained RNN with an architecture of the form depicted in Figure 12.13.

12.15 ($\star\star$) Consider a sequence of two tokens y_1 and y_2 each of which can take the states A or B. The table below shows the joint probability distribution $p(y_1, y_2)$:

Telegram: @uni_k

www.konkur.in

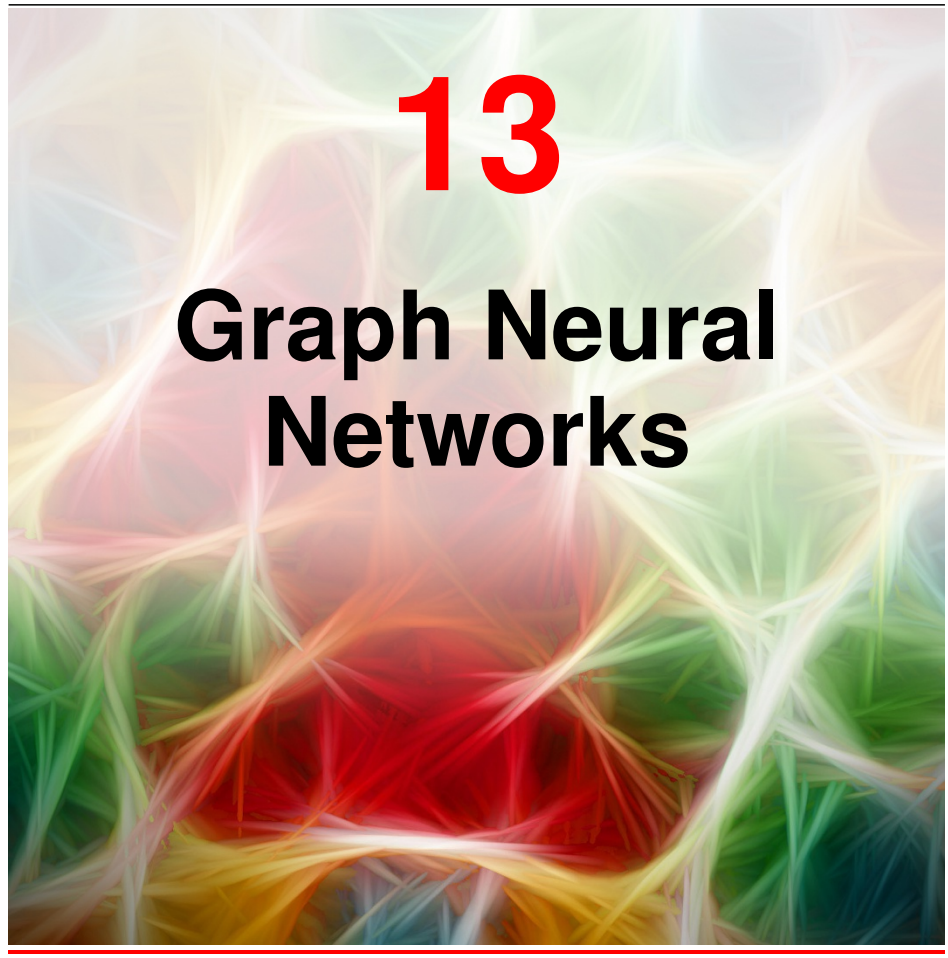
	$y_1 = A$	$y_1 = B$
$y_2 = A$	0.0	0.4
$y_2 = B$	0.1	0.25

We see that the most probable sequence is $y_1 = B$, $y_2 = B$ and that this has probability 0.4. Using the sum and product rules of probability, write down the values of the marginal distribution $p(y_1)$ and the conditional distribution $p(y_2|y_1)$. Show that if we first maximize $p(y_1)$ to give a value $y_1^\star$ and then subsequently maximize $p(y_2|y_1^\star)$ then we obtain a sequence that is different from the overall most probable sequence. Find the probability of the sequence.

12.16 ($\star$) The BERT-Large model (Devlin *et al.*, 2018) has a maximum input length of 512 tokens, each of dimensionality $D = 1{,}024$ and taken from a vocabulary of 30,000. It has 24 transformer layers each with 16 self-attention heads with $D_{\mathrm{q}} = D_{\mathrm{k}} = D_{\mathrm{v}} = 64$, and the MLP position-wise networks have two layers with 4,096 hidden nodes. Show that the total number of parameters in the BERT encoder transformer language model is approximately 340 million.

Telegram: @uni_k

www.konkur.in



13 Graph Neural Networks

In previous chapters we have encountered *structured data* in the form of sequences and images, corresponding to one-dimensional and two-dimensional arrays of variables respectively. More generally, there are many types of structured data that are best described by a *graph* as illustrated in Figure 13.1. In general a graph consists of a set of objects, known as *nodes*, connected by *edges*. Both the nodes and the edges can have data associated with them. For example, in a molecule the nodes and edges are associated with discrete variables corresponding to the types of atom (carbon, nitrogen, hydrogen, etc.) and the types of bonds (single bond, double bond, etc.). For a rail network, each railway line might be associated with a continuous variable given by the average journey time between two cities. Here we are assuming that the edges are symmetrical, for example that the journey time from London to Cambridge is the same as the journey time from Cambridge to London. Such edges are depicted by undirected links between the nodes. For the worldwide web the edges are directed

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024
C. M. Bishop, H. Bishop, *Deep Learning*, https://doi.org/10.1007/978-3-031-45468-4_13

Telegram: @uni_k

www.konkur.in

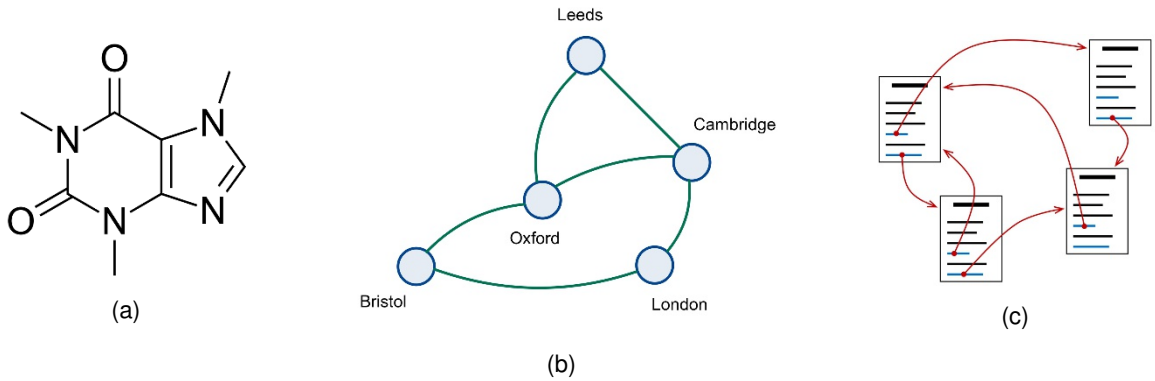


Figure 13.1 Three examples of graph-structured data: (a) the caffeine molecule consisting of atoms connected by chemical bonds, (b) a rail network consisting of cities connected by railway lines, and (c) the worldwide web consisting of pages connected by hyperlinks.

since if there is a hyperlink on page A that points to page B there is not necessarily a hyperlink on page B pointing back to page A.

Other examples of graph-structured data include a protein interaction network, in which the nodes are proteins and the edges express how strongly pairs of proteins interact, an electrical circuit where the nodes are components and the edges are conductors, or a social network where the nodes are people and the edges are 'friendships'. More complex graphical structures are also possible, for example the knowledge graph inside a company comprises multiple different kinds of nodes such as people, documents, and meetings, along with multiple kinds of edges capturing different properties such as a person being present at a meeting or a document referencing another document.

In this chapter we explore how to apply deep learning to graph-structured data. We have already encountered an example of structured data when we discussed images, in which the individual elements of an image data vector $\mathbf{x}$ correspond to pixels on a regular grid. An image is therefore a special instance of graph-structured data in which the nodes are the pixels and the edges describe which pixels are adjacent. *Chapter 10* Convolutional neural networks (CNNs) take this structure into account, incorporating prior knowledge of the relative positions of the pixels, together with the equivariance of properties such as segmentation and the invariance of properties such as classification. We will use CNNs for images as a source of inspiration to construct more general approaches to deep learning for graphical data known as *graph neural networks* (Zhou *et al.*, 2018; Wu *et al.*, 2019; Hamilton, 2020; Veličković, 2023). We will see that a key consideration when applying deep learning to graph-structured data is to ensure either equivariance or invariance with respect to a reordering of the nodes in the graph.

Telegram: @uni_k

www.konkur.in

13.1. Machine Learning on Graphs

There are many kinds of applications that we might wish to address using graph-structured data, and we can group these broadly according to whether the goal is to predict properties of nodes, of edges, or of the whole graph. An example of node prediction would be to classify documents according to their topic based on the hyperlinks and citations between the documents.

Regarding edges we might, for example, know some of the interactions in a protein network and would like to predict the presence of any additional ones. Such tasks are called *edge prediction* or *graph completion* tasks. There are also tasks where the edges are known in advance and the goal is to discover clusters or 'communities' within the graph.

Finally, we may wish to predict properties that relate to the graph as a whole. For example, we might wish to predict whether a particular molecule is soluble in water. Here instead of being given a single graph we will have a data set of different graphs, which we can view as being drawn from some common distribution, in other words we assume that the graphs themselves are *independent and identically distributed*. Such tasks can be considered as graph regression or graph classification tasks.

For the molecule solubility classification example, we might be given a labelled training set of molecules, along with a test set of new molecules whose solubility needs to be predicted. This is a standard example of an *inductive* task of the kind we have seen many times in previous chapters. However, some graph prediction examples are *transductive* in which we are given the structure of the entire graph along with labels for some of the nodes and the goal is to predict the labels of the remaining nodes. An example would be a large social network in which our goal is to classify each node as either a real person or an automated bot. Here a small number of nodes might be manually labelled, but it would be prohibitive to investigate every node individually in a large and ever-changing social network. During training, we therefore have access to the whole graph along with labels for a subset of the nodes, and we wish to predict the labels for the remaining nodes. This can be viewed as a form of semi-supervised learning.

As well as solving prediction tasks directly, we can also use deep learning on graphs to discover useful internal representations that can subsequently facilitate a range of downstream tasks. This is known as *graph representation learning*. For example we could seek to build a *foundation model* for molecules by training a deep learning system on a large corpus of molecular structures. The goal is that once trained, such a foundation model can be fine-tuned to specific tasks by using a small, labelled data set.

Graph neural networks define an *embedding vector* for each of the nodes, usually initialized with the observed node properties, which are then transformed through a series of learnable layers to create a learned representation. This is analogous to the way word embeddings, or tokens, are processed through a series of layers in the transformer to give a representation that better captures the meaning of the words in the context of the rest of the text. Graph neural networks can also use learned embeddings associated with the edges and with the graph as a whole.

Chapter 12

Telegram: @uni_k

www.konkur.in

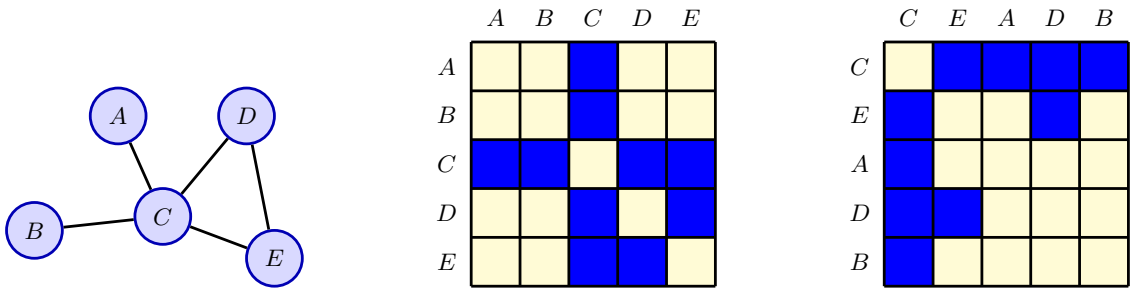


Figure 13.2 An example of an adjacency matrix showing (a) an example of a graph with five nodes, (b) the associated adjacency matrix for a particular choice of node order, and (c) the adjacency matrix corresponding to a different choice for the node order.

13.1.1 Graph properties

In this chapter we will focus on *simple graphs* where there is at most one edge between any pair of nodes, where the edges are undirected, and where there are no self-edges that connect a node to itself. This suffices to introduce the key concepts of graph neural networks, and it also encompasses a wide range of practical applications. These concepts can then be applied to more complex graphical structures.

We begin by introducing some notation associated with graphs and by defining some important properties. A graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ consists of a set of *nodes* or *vertices*, denoted by $\mathcal{V}$, along with a set of *edges* or *links*, denoted by $\mathcal{E}$. We index the nodes by $n = 1, \ldots N$, and we write the edge from node n to node m as (n, m). If two nodes are linked by an edge they are called *neighbours*, and the set of all neighbours of node n is denoted by $\mathcal{N}(n)$.

In addition to the graph structure, we usually also have observed data associated with the nodes. For each node n we can represent the corresponding node variables as a D-dimensional column vector $\mathbf{x}_n$ and we can group these into a data matrix $\mathbf{X}$ of dimensionality $N \times D$ in which row n is given by $\mathbf{x}_n^{\mathrm{T}}$. There may also be data variables associated with the edges in the graph, although to start with we will focus just on node variables.

Section 13.3.2

13.1.2 Adjacency matrix

A convenient way to specify the edges in a graph is to use an *adjacency matrix* denoted by $\mathbf{A}$. To define the adjacency matrix we first have to choose an ordering for the nodes. If there are N nodes in the graph, we can index them using $n = 1, \ldots, N$. The adjacency matrix has dimensions $N \times N$ and contains a 1 in every location n, m for which there is an edge going from node n to node m, with all other entries being 0. For graphs with undirected edges, the adjacency matrix will be symmetric since the presence of an edge from node n to node m implies that there is also an edge from node m to node n, and therefore $A_{mn} = A_{nm}$ for all n and m. An example of an adjacency matrix is shown in Figure 13.2.

Since the adjacency matrix defines the structure of a graph, we could consider

Telegram: @uni_k

www.konkur.in

using it directly as the input to a neural network. To do this we could 'flatten' the matrix, for example by concatenating the columns into one long column vector. However, a major problem with this approach is that the adjacency matrix depends on the arbitrary choice of node ordering, as seen in Figure 13.2. Suppose for instance that we want to predict the solubility of a molecule. This clearly should not depend on the ordering assigned to the nodes when writing down an adjacency matrix. Because the number of permutations increases factorially with the number of nodes, it is impractical to try to learn permutation invariance by using large data sets or by data augmentation. Instead, we should treat this invariance property as an inductive bias when constructing a network architecture.

13.1.3 Permutation equivariance

We can express node label permutation mathematically by introducing the concept of a *permutation matrix* $\mathbf{P}$, which has the same size as the adjacency matrix and which specifies a particular permutation of a node ordering. It contains a single 1 in each row and a single 1 in each column, with 0 in all the other elements, such that a 1 in position n, m indicates that node n will be relabelled as node m after the permutation. Consider, for example, the permutation from $(A, B, C, D, E) \rightarrow (C, E, A, D, B)$ corresponding to the two choices of node ordering in Figure 13.2.

Exercise 13.1

The corresponding permutation matrix takes the form

$$\mathbf{P} = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{pmatrix}. \tag{13.1}$$

We can define the permutation matrix more formally as follows. First we introduce the *standard unit vector* $\mathbf{u}_n$, for $n = 1, \ldots, N$. This is a column vector in which all elements are 0 except element n, which equals 1. In this notation the identity matrix is given by

$$\mathbf{I} = \begin{pmatrix} \mathbf{u}_1^{\mathrm{T}} \\ \mathbf{u}_2^{\mathrm{T}} \\ \cdots \\ \mathbf{u}_N^{\mathrm{T}} \end{pmatrix}. \tag{13.2}$$

We can now introduce a permutation function $\pi(\cdot)$ that maps n to $m = \pi(n)$. The associated permutation matrix is given by

$$\mathbf{P} = \begin{pmatrix} \mathbf{u}_{\pi(1)}^{\mathrm{T}} \\ \mathbf{u}_{\pi(2)}^{\mathrm{T}} \\ \cdots \\ \mathbf{u}_{\pi(N)}^{\mathrm{T}} \end{pmatrix}. \tag{13.3}$$

When we reorder the labelling on the nodes of a graph, the effect on the corresponding node data matrix $\mathbf{X}$ is to permute the rows according to $\pi(\cdot)$, which can be achieved by pre-multiplication by $\mathbf{P}$ to give

Exercise 13.4

Telegram: @uni_k

www.konkur.in

$$\widetilde{\mathbf{X}} = \mathbf{P}\mathbf{X}. \tag{13.4}$$

For the adjacency matrix, both the rows and the columns become permuted. Again the rows can be permuted using pre-multiplication by $\mathbf{P}$ whereas the columns are permuted using post-multiplication by $\mathbf{P}^{\mathrm{T}}$, giving a new adjacency matrix:

Exercise 13.5

$$\widetilde{\mathbf{A}} = \mathbf{P}\mathbf{A}\mathbf{P}^{\mathrm{T}}. \tag{13.5}$$

When applying deep learning to graph-structured data, we will need to represent the graph structure in numerical form so that it can be fed into a neural network, which requires that we assign an ordering to the nodes. However, the specific ordering we choose is arbitrary and so it will be important to ensure that any global property of the graph does not depend on this ordering. In other words, the network predictions must be *invariant* to node label reordering, so that

$$y(\widetilde{\mathbf{X}}, \widetilde{\mathbf{A}}) = y(\mathbf{X}, \mathbf{A}) \qquad \text{Invariance} \tag{13.6}$$

where $y(\cdot, \cdot)$ is the output of the network.

We may also want to make predictions that relate to individual nodes. In this case, if we reorder the node labelling then the corresponding predictions should show the same reordering so that a given prediction is always associated with the same node irrespective of the choice of order. In other words, node predictions should be *equivariant* with respect to node label reordering. This can be expressed as

$$\mathbf{y}(\widetilde{\mathbf{X}}, \widetilde{\mathbf{A}}) = \mathbf{P}\mathbf{y}(\mathbf{X}, \mathbf{A}) \qquad \text{Equivariance} \tag{13.7}$$

where $\mathbf{y}(\cdot, \cdot)$ is a vector of network outputs, with one element per node.

13.2. Neural Message-Passing

Ensuring invariance or equivariance under node label permutations is a key design consideration when we apply deep neural networks to graph-structured data. Another consideration is that we want to exploit the representational capabilities of deep neural networks and so we retain the concept of a 'layer' as a computational transformation that can be applied repeatedly. If each layer of the network is equivariant under node reordering then multiple layers applied in succession will also exhibit equivariance, while allowing each layer of the network to be informed by the graph structure.

For networks whose outputs represent node-level predictions, the whole network will be equivariant as required. If the network is being used to predict a graph-level property then a final layer can be included that is invariant to permutations of its inputs. We also want to ensure that each layer is a highly flexible nonlinear function and is differentiable with respect to its parameters so that it can be trained by stochastic gradient descent using gradients obtained by automatic differentiation.

Graphs come in various sizes. For example different molecules can have different numbers of atoms, so a fixed-length representation as used for standard neural

Telegram: @uni_k

www.konkur.in

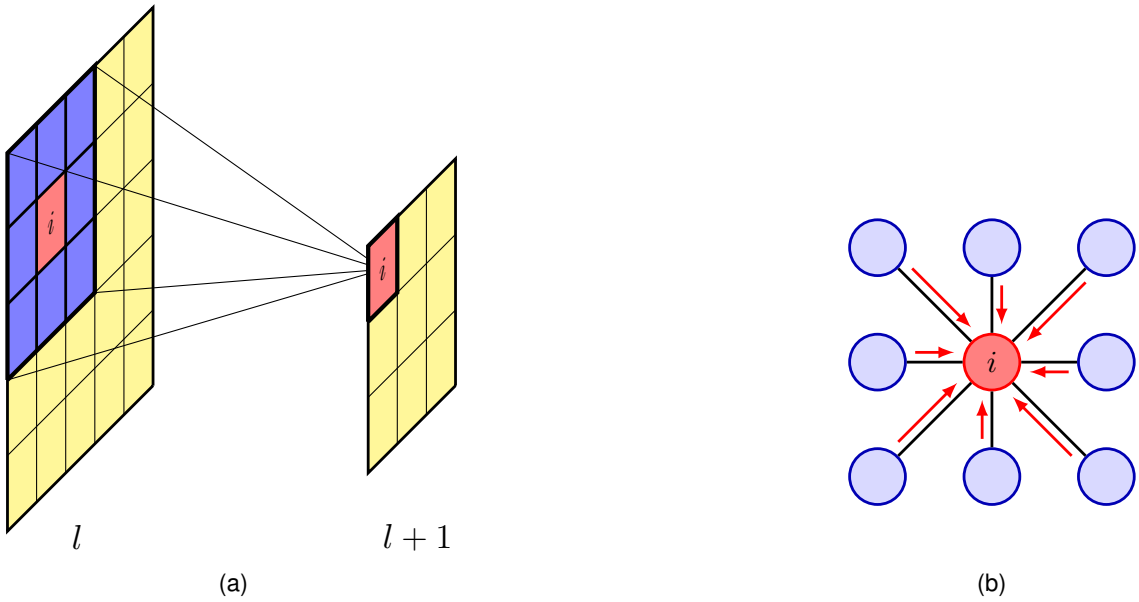


Figure 13.3 A convolutional filter for images can be represented as a graph-structured computation. (a) A filter computed by node i in layer $l + 1$ of a deep convolutional network is a function of the activation values in layer l over a local patch of pixels. (b) The same computation structure expressed as a graph showing 'messages' flowing into node i from its neighbours.

Chapter 12

networks is unsuitable. A further requirement is therefore that the network should be able to handle variable-length inputs, as we saw with transformer networks. Some graphs can be very large, for example a social network with many millions of participants, and so we also want to construct models that scale well. Not surprisingly, parameter sharing will play an important role, both to allow the invariance and equivariance properties to be built into the network architecture but also to facilitate scaling to large graphs.

13.2.1 Convolutional filters

Chapter 10

To develop a framework that meets all of these requirements, we can seek inspiration from image processing using convolutional neural networks. First note that an image can be viewed as a specific instance of graph-structured data, in which the nodes are the pixels and the edges represent pairs of pixels that are adjacent in the image, where adjacency includes nodes that are diagonally adjacent as well as those that are horizontally or vertically adjacent.

Section 10.2

In a convolutional network, we make successive transformations of the image domain such that a pixel at a particular layer computes a function of states of pixels in the previous layer through a local function called a *filter*. Consider a convolutional layer using 3×3 filters, as illustrated in Figure 13.3(a). The computation performed

Telegram: @uni_k

www.konkur.in

by a single filter at a single pixel in layer $l+1$ can be expressed as

$$z_i^{(l+1)} = f\left(\sum_j w_j z_j^{(l)} + b\right) \tag{13.8}$$

where $f(\cdot)$ is a differentiable nonlinear activation function such as ReLU, and the sum over j is taken over all nine pixels in a small patch in layer l. The same function is applied across multiple patches in the image, so that the weights w_j and bias b are shared across the patches (and therefore do not carry the index i).

As it stands, (13.8) is not equivariant under reordering of the nodes in layer l because the weight vector, with elements w_j, is not invariant under permutation of its elements. However, we can achieve equivariance with some simple modifications as follows. We first view the filter as a graph, as shown in Figure 13.3(b), and separate out the contribution from node i. The other eight 8 nodes are its neighbours $\mathcal{N}(i)$. We then assume that a single weight parameter w_{neigh} is shared across the neighbours so that

$$z_i^{(l+1)} = f\left(w_{\text{neigh}} \sum_{j\in\mathcal{N}(i)} z_j^{(l)} + w_{\text{self}} z_i^{(l)} + b\right) \tag{13.9}$$

where node i has its own weight parameter w_{self}.

We can interpret (13.9) as updating a local representation z_i at node i by gathering information from the neighbouring nodes by passing *messages* from the neighbouring nodes into node i. In this case the messages are simply the activations of the other nodes. These messages are then combined with information from node i, and the result is transformed using a nonlinear function. The information from the neighbouring nodes is *aggregated* through a simple summation in (13.9), and this is clearly invariant to any permutation of the labels associated with those nodes. Furthermore, the operation (13.9) is applied synchronously to every node in a graph, and so if the nodes are permuted then the resulting computations will be unchanged but their ordering will be likewise permuted, and hence, this calculation is equivariant under node reordering. Note that this depends on the parameters w_{neigh}, w_{self}, and b being shared across all nodes.

13.2.2 Graph convolutional networks

We now use the convolution example as a template to construct deep neural networks for graph-structured data. Our goal is to define a flexible, nonlinear transformation of the node embeddings that is differentiable with respect to a set of weight and bias parameters and which maps the variables in layer l into corresponding variables in layer $l+1$. For each node n in the graph and for each layer l in the network, we introduce a D-dimensional column vector $\mathbf{h}_n^{(l)}$ of node-embedding variables, where $n = 1, \ldots, N$ and $l = 1, \ldots, L$.

We see that the transformation given by (13.9) first gathers and combines information from neighbouring nodes and then updates the node as a function of the

Telegram: @uni_k

www.konkur.in

Algorithm 13.1: Simple message-passing neural network

Input: Undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$
Initial node embeddings $\{\mathbf{h}_n^{(0)} = \mathbf{x}_n\}$
Aggregate$(\cdot)$ function
Update$(\cdot, \cdot)$ function
Output: Final node embeddings $\{\mathbf{h}_n^{(L)}\}$

```
// Iterative message-passing
```
for $l \in \{0, \ldots, L-1\}$ **do**
  $\mathbf{z}_n^{(l)} \leftarrow \text{Aggregate}\left(\left\{\mathbf{h}_m^{(l)} : m \in \mathcal{N}(n)\right\}\right)$
  $\mathbf{h}_n^{(l+1)} \leftarrow \text{Update}\left(\mathbf{h}_n^{(l)}, \mathbf{z}_n^{(l)}\right)$
end for
return $\{\mathbf{h}_n^{(L)}\}$

current embedding of the node and the incoming messages. We can therefore view each layer of processing as having two successive stages. The first is the *aggregation* stage in which, for each node n, messages are passed to that node from its neighbours and combined to form a new vector $\mathbf{z}_n^{(l)}$ in a way that is permutation invariant. This is followed by an *update* step in which the aggregated information from neighbouring nodes is combined with local information from the node itself and used to calculate a revised embedding vector for that node.

Consider a specific node n in the graph. We first aggregate the node vectors from all the neighbours of node n:

$$\mathbf{z}_n^{(l)} = \text{Aggregate}\left(\left\{\mathbf{h}_m^{(l)} : m \in \mathcal{N}(n)\right\}\right). \tag{13.10}$$

The form of this aggregation function is very flexible if it is well defined for a variable number of neighbouring nodes and does not depend on the ordering of those nodes. It can potentially contain learnable parameters as long as it is a differentiable function with respect to those parameters to facilitate gradient descent training.

We then use another operation to update the embedding vector at node n:

$$\mathbf{h}_n^{(l+1)} = \text{Update}\left(\mathbf{h}_n^{(l)}, \mathbf{z}_n^{(l)}\right). \tag{13.11}$$

Again, this can be a differentiable function of a set of learnable parameters. Application of the Aggregate operation followed by the Update operation in parallel for every node in the graph represents one layer of the network. The node embeddings are typically initialized using observed node data so that $\mathbf{h}_n^{(0)} = \mathbf{x}_n$. Note that each layer generally has its own independent parameters, although the parameters can also be shared across layers. This framework is called a *message-passing neural network* (Gilmer *et al.*, 2017) and is summarized in Algorithm 13.1.

Telegram: @uni_k

www.konkur.in

13.2.3 Aggregation operators

There are many possible forms for the Aggregate function, but it must depend only on the set of inputs and not on their ordering. It must also be a differentiable function of any learnable parameters. The simplest such aggregation function, following from (13.9), is summation:

$$\text{Aggregate}\left(\{\mathbf{h}_m^{(l)} : m \in \mathcal{N}(n)\}\right) = \sum_{m \in \mathcal{N}(n)} \mathbf{h}_m^{(l)}. \tag{13.12}$$

A simple summation is clearly independent of the ordering of the neighbouring nodes and is also well defined no matter how many nodes are in the neighbourhood set. Note that this has no learnable parameters.

A summation gives a stronger influence over nodes that have many neighbours compared to those with few neighbours, and this can lead to numerical issues, particularly in applications such as social networks where the size of the neighbourhood set can vary by several orders of magnitude. A variation of this approach is to define the Aggregation operation to be the average of the neighbouring embedding vectors so that

$$\text{Aggregate}\left(\{\mathbf{h}_m^{(l)} : m \in \mathcal{N}(n)\}\right) = \frac{1}{|\mathcal{N}(n)|} \sum_{m \in \mathcal{N}(n)} \mathbf{h}_m^{(l)} \tag{13.13}$$

where $|\mathcal{N}(n)|$ denotes the number of nodes in the neighbourhood set $\mathcal{N}(n)$. However, this normalization also discards information about the network structure and is provably less powerful than a simple summation (Hamilton, 2020), and so the choice of whether to use it depends on the relative importance of node features compared to graph structure.

Another variation of this approach (Kipf and Welling, 2016) takes account of the number of neighbours for each of the neighbouring nodes:

$$\text{Aggregate}\left(\{\mathbf{h}_m^{(l)} : m \in \mathcal{N}(n)\}\right) = \sum_{m \in \mathcal{N}(n)} \frac{\mathbf{h}_m^{(l)}}{\sqrt{|\mathcal{N}(n)|\,|\mathcal{N}(m)|}}. \tag{13.14}$$

Yet another possibility is to take the element-wise maximum (or minimum) of the neighbouring embedding vectors, which also satisfies the desired properties of being well defined for a variable number of neighbours and of being independent of their order.

Since each node in a given layer of the network is updated by aggregating information from its neighbours in the previous layer, this defines a *receptive field* analogous to the receptive fields of filters used in CNNs. As information is processed through successive layers, the updates to a given node depend on a steadily increasing fraction of other nodes in earlier layers until the effective receptive field potentially spans the whole graph as illustrated in Figure 13.4. However, large, sparse graphs may require an excessive number of layers before each output is influenced by every input. Some architectures therefore introduce an additional 'super-node'

Chapter 10

Telegram: @uni_k

www.konkur.in

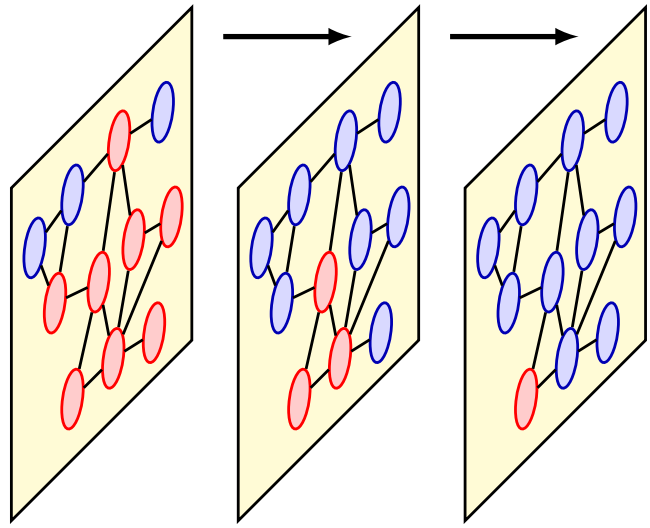


Figure 13.4 Schematic illustration of information flow through successive layers of a graph neural network. In the third layer a single node is highlighted in red. It receives information from its two neighbours in the previous layer and those in turn receive information from their neighbours in the first layer. As with convolutional neural networks for images, we see that the effective receptive field, corresponding to the number of nodes shown in red, grows with the number of processing layers.

that connects directly to every node in the original graph to ensure fast propagation of information.

The aggregation operators discussed so far have no learnable parameters. We can introduce such parameters if we first transform each of the embedding vectors from neighbouring nodes using a multilayer neural network, denoted by $\mathrm{MLP}_{\boldsymbol{\phi}}$, before combining their outputs, where MLP denotes 'multilayer perceptron' and $\boldsymbol{\phi}$ represents the parameters of the network. So long as the network has a structure and parameter values that are shared across nodes then this aggregation operator again be permutation invariant. We can also transform the combined vector with another neural network $\mathrm{MLP}_{\boldsymbol{\theta}}$, with parameters $\boldsymbol{\theta}$, to give an overall aggregation operator:

$$\text{Aggregate}\left(\{\mathbf{h}_m^{(l)} : m \in \mathcal{N}(n)\}\right) = \mathrm{MLP}_{\boldsymbol{\theta}}\left(\sum_{m\in\mathcal{N}(n)} \mathrm{MLP}_{\boldsymbol{\phi}}(\mathbf{h}_m^{(l)})\right) \tag{13.15}$$

in which $\mathrm{MLP}_{\boldsymbol{\phi}}$ and $\mathrm{MLP}_{\boldsymbol{\theta}}$ are shared across layer l. Due to the flexibility of MLPs, the transformation defined by (13.15) represents a universal approximator for any permutation-invariant function that maps a set of embeddings to a single embedding (Zaheer *et al.*, 2017). Note that the summation can be replaced by other invariant functions such as averages or an element-wise maximum or minimum.

A special case of graph neural networks arises if we consider a graph having no edges, which corresponds simply to an unstructured set of nodes. In this case if we use (13.15) for each vector $\mathbf{h}_n^{(l)}$ in the set, in which the summation is taken over all other vectors except $\mathbf{h}_n^{(l)}$, then we have a general framework for learning functions over unstructured sets of variables known as *deep sets*.

Telegram: @uni_k

www.konkur.in

13.2.4 Update operators

Having chosen a suitable Aggregate operator, we similarly need to decide on the form of the Update operator. By analogy with (13.9) for the CNN, a simple form for this operator would be

$$\text{Update}\left(\mathbf{h}_n^{(l)}, \mathbf{z}_n^{(l)}\right) = f\left(\mathbf{W}_{\text{self}}\mathbf{h}_n^{(l)} + \mathbf{W}_{\text{neigh}}\mathbf{z}_n^{(l)} + \mathbf{b}\right) \tag{13.16}$$

where $f(\cdot)$ is a nonlinear activation function such as ReLU applied element-wise to its vector argument, and where $\mathbf{W}_{\text{self}}$, $\mathbf{W}_{\text{neigh}}$, and $\mathbf{b}$ are the learnable weights and biases and $\mathbf{z}_n^{(l)}$ is defined by the Aggregate operator (13.10).

If we choose a simple summation (13.12) as the aggregation function and if we also share the same weight matrix between nodes and their neighbours so that $\mathbf{W}_{\text{self}} = \mathbf{W}_{\text{neigh}}$, we obtain a particularly simple form of Update operator given by

$$\mathbf{h}_n^{(l+1)} = \text{Update}\left(\mathbf{h}_n^{(l)}, \mathbf{z}_n^{(l)}\right) = f\left(\mathbf{W}_{\text{neigh}} \sum_{m \in \mathcal{N}(n), n} \mathbf{h}_m^{(l)} + \mathbf{b}\right). \tag{13.17}$$

The message-passing algorithm is typically initialized by setting $\mathbf{h}_n^{(0)} = \mathbf{x}_n$. Sometimes, however, we may want to have an internal representation vector for each node that has a higher, or lower, dimensionality than that of $\mathbf{x}_n$. Such a representation can be initialized by padding the node vectors $\mathbf{x}_n$ with additional zeros (to achieve a higher dimensionality) or simply by transforming the node vectors using a learnable linear transformation to a space of the desired number of dimensions. An alternative form of initialization, particularly when there are no data variables associated with the nodes, is to use a one-hot vector that labels the degree of each node (i.e., the number of neighbours).

Overall, we can represent a graph neural network as a sequence of layers that successively transform the node embeddings. If we group these embeddings into a matrix $\mathbf{H}$ whose nth row is the vector $\mathbf{h}_n^{\mathrm{T}}$, which is initialized to the data matrix $\mathbf{X}$, then we can write the successive transformations in the form

$$\begin{aligned}
\mathbf{H}^{(1)} &= \mathbf{F}\left(\mathbf{X}, \mathbf{A}, \mathbf{W}^{(1)}\right) \\
\mathbf{H}^{(2)} &= \mathbf{F}\left(\mathbf{H}^{(1)}, \mathbf{A}, \mathbf{W}^{(2)}\right) \\
\vdots \quad &= \quad \vdots \\
\mathbf{H}^{(L)} &= \mathbf{F}\left(\mathbf{H}^{(L-1)}, \mathbf{A}, \mathbf{W}^{(L)}\right)
\end{aligned} \tag{13.18}$$

where $\mathbf{A}$ is the adjacency matrix, and $\mathbf{W}^{(l)}$ represents the complete set of weight and biases in layer l of the network. Under a node reordering defined by a permutation matrix $\mathbf{P}$, the transformation of the node embeddings computed by layer l is equivariant:

$$\mathbf{P}\mathbf{H}^{(l)} = \mathbf{F}\left(\mathbf{P}\mathbf{H}^{(l-1)}, \mathbf{P}\mathbf{A}\mathbf{P}^{\mathrm{T}}, \mathbf{W}^{(l)}\right). \tag{13.19}$$

Exercise 13.7 As a consequence, the complete network computes an equivariant transformation.

Telegram: @uni_k

www.konkur.in

13.2.5 Node classification

A graph neural network can be viewed as a series of layers each of which transforms a set of node-embedding vectors $\{\mathbf{h}_n^{(l)}\}$ into a new set $\{\mathbf{h}_n^{(l+1)}\}$ of the same size and dimensionality. After the final convolutional layer of the network, we need to obtain predictions so that we can define a cost function for training and also for making predictions on new data using the trained network.

Consider first the task of classifying the nodes in a graph, which is one of the most common uses for graph neural networks. We can define an output layer, sometimes called a *readout layer*, which calculates a softmax function for each node corresponding to a classification over C classes, of the form

$$y_{ni} = \frac{\exp(\mathbf{w}_i^{\mathrm{T}}\mathbf{h}_n^{(L)})}{\sum_j \exp(\mathbf{w}_j^{\mathrm{T}}\mathbf{h}_n^{(L)})} \tag{13.20}$$

where $\{\mathbf{w}_i\}$ is a set of learnable weight vectors and $i = 1, \ldots, C$. We can then define a loss function as the sum of the cross-entropy loss across all nodes and all classes:

$$\mathcal{L} = -\sum_{n \in \mathcal{V}_{\text{train}}} \sum_{i=1}^{C} y_{ni}^{t_{ni}} \tag{13.21}$$

where $\{t_{ni}\}$ are target values with a one-hot encoding for each value of n. Because the weight vectors $\{\mathbf{w}_i\}$ are shared across the output nodes, the outputs y_{ni} are equivariant to permutation of the node ordering, and hence the loss function (13.21) is invariant. If the goal is to predict continuous values at the outputs then a simple linear transformation can be combined with a sum-of-squares error to define a suitable loss function.

The sum over n in (13.21) is taken over the subset of the nodes denoted by $\mathcal{V}_{\text{train}}$ and used for training. We can distinguish between three types of nodes as follows:

1. The nodes $\mathcal{V}_{\text{train}}$ are labelled and included in the message-passing operations of the graph neural network and are also used to compute the loss function used for training.

2. There is potentially also a *transductive* subset of nodes denoted by $\mathcal{V}_{\text{trans}}$, which are unlabelled and which do not contribute to the evaluation of the loss function used for training. However, they still participate in the message-passing operations during both training and inference, and their labels may be predicted as part of the inference process.

3. The remaining nodes, denoted $\mathcal{V}_{\text{induct}}$, are a set of *inductive* nodes that are not used to compute the loss function, and neither these nodes nor their associated edges participate in message-passing during the training phase. However, they do participate in message-passing during the inference phase and their labels are predicted as the outcome of inference.

Telegram: @uni_k

www.konkur.in

If there are no transductive nodes, and hence the test nodes (and their associated edges) are not available during the training phase, then the training is generally referred to as *inductive learning*, which can be considered to be a form of *supervised learning*. However, if there are transductive nodes then it is called *transductive learning*, which may be viewed as a form of *semi-supervised* learning.

13.2.6 Edge classification

In some applications we wish to make predictions about the edges of the graph rather than the nodes. A common form of edge classification task is edge completion in which the goal is to determine whether an edge should be present between two nodes. Given a set of node embeddings, the dot product between pairs of embeddings can be used to define a probability $p(n, m)$ for the presence of an edge between nodes n and m by using the logistic sigmoid function:

$$p(n, m) = \sigma\left(\mathbf{h}_n^{\mathrm{T}}\mathbf{h}_m\right). \tag{13.22}$$

An example application would be predicting whether two people in a social network have shared interests and therefore might wish to connect.

13.2.7 Graph classification

In some applications of graph neural networks, the goal is to predict the properties of new graphs given a training set of labelled graphs $\mathcal{G}_1, \ldots, \mathcal{G}_N$. This requires that we combine the final-layer embedding vectors in a way that does not depend on the arbitrary node ordering, thereby ensuring that the output predictions will be invariant to that ordering. The goal is somewhat like that of the Aggregate function except that all nodes in the graph are included, not just the neighbourhood sets of the individual nodes. The simplest approach is to take the sum of the node-embedding vectors:

$$\mathbf{y} = \mathbf{f}\left(\sum_{n\in\mathcal{V}} \mathbf{h}_n^{(L)}\right) \tag{13.23}$$

where the function $\mathbf{f}$ may contain learnable parameters such as a linear transformation or a neural network. Other invariant aggregation functions can be used such as averages or element-wise minimum or maximum.

A cross-entropy loss is typically used for classification problems, such as labelling a candidate drug molecule as toxic or safe, and a squared-error loss for regression problems, such as predicting the solubility of a candidate drug molecule. Graph-level predictions correspond to an inductive task since there must be separate sets of graphs for training and for inference.

13.3. General Graph Networks

There are many variations and extensions of the graph networks considered so far. Here we outline a few of the key concepts along with some practical considerations.

Telegram: @uni_k

www.konkur.in

13.3.1 Graph attention networks

Section 12.1

The attention mechanism is very powerful when used as the basis of a transformer architecture. It can be used in the context of graph neural networks to construct an aggregation function that combines messages from neighbouring nodes. The incoming messages are weighted by attention coefficients A_{nm} to give

$$\mathbf{z}_n^{(l)} = \text{Aggregate}\left(\{\mathbf{h}_m^{(l)} : m \in \mathcal{N}(n)\}\right) = \sum_{m \in \mathcal{N}(n)} A_{nm}\mathbf{h}_m^{(l)} \tag{13.24}$$

where the attention coefficients satisfy

$$A_{nm} \geqslant 0 \tag{13.25}$$

$$\sum_{m \in \mathcal{N}(n)} A_{nm} = 1. \tag{13.26}$$

This is known as a *graph attention network* (Veličković *et al.*, 2017) and can capture an inductive bias that says some neighbouring nodes will be more important than others in determining the best update in a way that depends on the data itself.

There are multiple ways to construct the attention coefficients, and these generally employ a softmax function. For example, we can use a bilinear form:

$$A_{nm} = \frac{\exp\left(\mathbf{h}_n^{\mathrm{T}}\mathbf{W}\mathbf{h}_m\right)}{\sum_{m' \in \mathcal{N}(n)} \exp\left(\mathbf{h}_n^{\mathrm{T}}\mathbf{W}\mathbf{h}_{m'}\right)} \tag{13.27}$$

where $\mathbf{W}$ is a $D \times D$ matrix of learnable parameters. A more general option is to use a neural network to combine the embedding vectors from the nodes at each end of the edge:

$$A_{nm} = \frac{\exp\left\{\text{MLP}\left(\mathbf{h}_n, \mathbf{h}_m\right)\right\}}{\sum_{m' \in \mathcal{N}(n)} \exp\left\{\text{MLP}\left(\mathbf{h}_n, \mathbf{h}_{m'}\right)\right\}} \tag{13.28}$$

where the MLP has a single continuous output variable whose value is invariant if the input vectors are exchanged. Provided the MLP is shared across all the nodes in the network, this aggregation function will be equivariant under node reordering.

Exercise 13.8

A graph attention network can be extended by introducing multiple attention heads in which H distinct sets of attention weights $A_{nm}^{(h)}$ are defined, for $h = 1, \ldots, H$, in which each head is evaluated using one of the mechanisms described above and with its own independent parameters. These are then combined in the aggregation step using concatenation and linear projection. Note that, for a fully-connected network, a multi-head graph attention network becomes a standard transformer encoder.

Section 12.1.6

Exercise 13.9

13.3.2 Edge embeddings

The graph neural networks discussed above use embedding vectors that are associated with the nodes. We have seen that some networks also have data associated with the edges. Even when there are no observable values associated with the edges,

Telegram: @uni_k

www.konkur.in

we can still maintain and update edge-based hidden variables and these can contribute to the internal representations learned by the graph neural network.

In addition to the node embeddings given by $\mathbf{h}_n^{(l)}$, we therefore introduce edge embeddings $\mathbf{e}_{nm}^{(l)}$. We can then define general message-passing equations in the form

$$\mathbf{e}_{nm}^{(l+1)} = \text{Update}_{\text{edge}}\left(\mathbf{e}_{nm}^{(l)}, \mathbf{h}_n^{(l)}, \mathbf{h}_m^{(l)}\right) \tag{13.29}$$

$$\mathbf{z}_n^{(l+1)} = \text{Aggregate}_{\text{node}}\left(\{\mathbf{e}_{nm}^{(l+1)} : m \in \mathcal{N}(n)\}\right) \tag{13.30}$$

$$\mathbf{h}_n^{(l+1)} = \text{Update}_{\text{node}}\left(\mathbf{h}_n^{(l)}, \mathbf{z}_n^{(l+1)}\right). \tag{13.31}$$

The learned edge embeddings $\mathbf{e}_{nm}^{(L)}$ from the final layer can be used directly to make predictions associated with the edges.

13.3.3 Graph embeddings

In addition to node and edge embeddings we can also maintain and update an embedding vector $\mathbf{g}^{(l)}$ that relates to the graph as a whole. Bringing all these aspects together allows us to define a more general set of message-passing functions, and a richer set of learned representations, for graph-structured applications. Specifically, we can define general message-passing equations (Battaglia *et al.*, 2018):

$$\mathbf{e}_{nm}^{(l+1)} = \text{Update}_{\text{edge}}\left(\mathbf{e}_{nm}^{(l)}, \mathbf{h}_n^{(l)}, \mathbf{h}_m^{(l)}, \mathbf{g}^{(l)}\right) \tag{13.32}$$

$$\mathbf{z}_n^{(l+1)} = \text{Aggregate}_{\text{node}}\left(\{\mathbf{e}_{nm}^{(l+1)} : m \in \mathcal{N}(n)\}\right) \tag{13.33}$$

$$\mathbf{h}_n^{(l+1)} = \text{Update}_{\text{node}}\left(\mathbf{h}_n^{(l)}, \mathbf{z}_n^{(l+1)}, \mathbf{g}^{(l)}\right) \tag{13.34}$$

$$\mathbf{g}^{(l+1)} = \text{Update}_{\text{graph}}\left(\mathbf{g}^{(l)}, \{\mathbf{h}_n^{(l+1)} : n \in \mathcal{V}\}, \{\mathbf{e}_{nm}^{(l+1)} : (n,m) \in \mathcal{E}\}\right). \tag{13.35}$$

These update equations start in (13.32) by updating the edge embedding vectors $\mathbf{e}_{nm}^{(l+1)}$ based on the previous states of those vectors, on the node embeddings for the nodes connected by each edge, and on a graph-level embedding vector $\mathbf{g}^{(l)}$. These updated edge embeddings are then aggregated across every edge connected to each node using (13.33) to give a set of aggregated vectors. These in turn then contribute to the update of the node-embedding vector $\{\mathbf{h}_n^{(l+1)}\}$ based on the current node-embedding vectors and on the graph-level embedding vector using (13.34). Finally, the graph-level embedding vector is updated using (13.35) based on information from all the nodes and all the edges in the graph along with the graph-level embedding from the previous layer. These message-passing updates are illustrated in Figure 13.5 and are summarized in Algorithm 13.2.

13.3.4 Over-smoothing

One significant problem that can arise with some graph neural networks is called *over-smoothing* in which the node-embedding vectors tend to become very similar to each other after a number of iterations of message-passing, which effectively limits the depth of the network. One way to help alleviate this issue is to introduce residual connections. For example, we can modify the update operator (13.34):

Section 9.5

$$\mathbf{h}_n^{(l+1)} = \text{Update}_{\text{node}}\left(\mathbf{h}_n^{(l)}, \mathbf{z}_n^{(l+1)}, \mathbf{g}^{(l)}\right) + \mathbf{h}_n^{(l)}. \tag{13.36}$$

Telegram: @uni_k

www.konkur.in

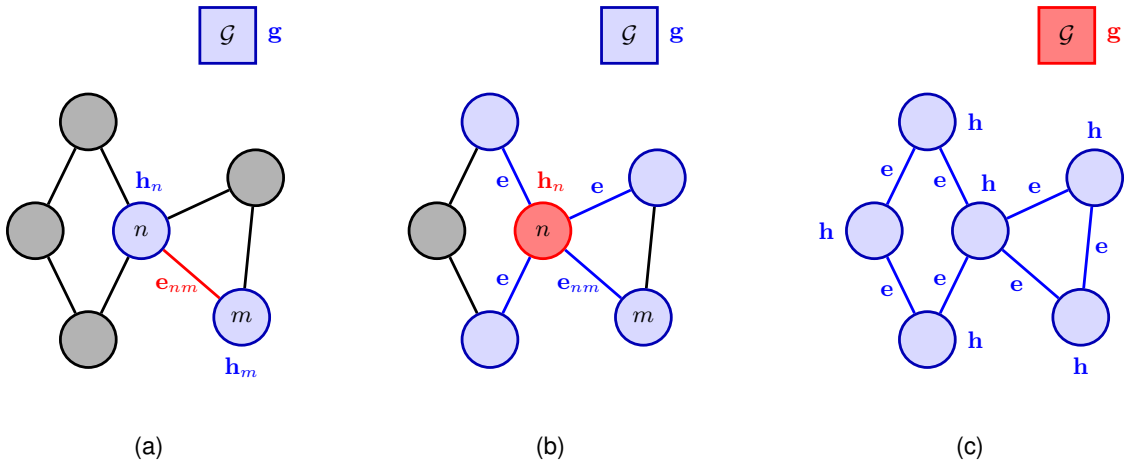


Figure 13.5 Illustration of the general graph message-passing updates defined by (13.32) to (13.35), showing (a) edge updates, (b) node updates, and (c) global graph updates. In each case the variable being updated is shown in red and the variables that contribute to that update are those shown in red and blue.

Another approach for mitigating the effects of over-smoothing is to allow the output layer to take information from all previous layers of the network and not just the final convolutional layer. This can be done for example by concatenating the representations from previous layers:

$$\mathbf{y}_n = \mathbf{f}\left(\mathbf{h}_n^{(1)} \oplus \mathbf{h}_n^{(2)} \oplus \cdots \oplus \mathbf{h}_n^{(L)}\right) \tag{13.37}$$

where $\mathbf{a} \oplus \mathbf{b}$ denotes the concatenation of vectors $\mathbf{a}$ and $\mathbf{b}$. A variant of this would be to combine the vectors using max pooling instead of concatenation. In this case each element of the output vector is given by the max of all the corresponding elements of the embedding vectors from the previous layers.

13.3.5 Regularization

Chapter 9

Standard techniques for regularization can be used with graph neural networks, including the addition of penalty terms, such as the sum-of-squares of the parameter values, to the loss function. In addition, some regularization methods have been developed specifically for graph neural networks.

Graph neural networks already employ weight sharing to achieve permutation equivariance and invariance, but typically they have independent parameters in each layer. However, weights and biases can also be shared across layers to reduce the number of independent parameters.

Dropout in the context of graph neural networks involves omitting random subsets of the graph nodes during training, with a fresh random subset chosen for each forward pass. This can likewise be applied to the edges in the graph in which randomly selected subsets of entries in the adjacency matrix are removed, or masked, during training.

Telegram: @uni_k

www.konkur.in

Algorithm 13.2: Graph neural network with node, edge, and graph embeddings

Input: Undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$
Initial node embeddings $\{\mathbf{h}_n^{(0)}\}$
Initial edge embeddings $\{\mathbf{e}_{nm}^{(0)}\}$
Initial graph embedding $\mathbf{g}^{(0)}$
Output: Final node embeddings $\{\mathbf{h}_n^{(L)}\}$
Final edge embeddings $\{\mathbf{e}_{nm}^{(L)}\}$
Final graph embedding $\mathbf{g}^{(L)}$

// Iterative message-passing
for $l \in \{0, \ldots, L-1\}$ **do**

$$\mathbf{e}_{nm}^{(l+1)} \leftarrow \text{Update}_{\text{edge}}\left(\mathbf{e}_{nm}^{(l)}, \mathbf{h}_n^{(l)}, \mathbf{h}_m^{(l)}, \mathbf{g}^{(l)}\right)$$
$$\mathbf{z}_n^{(l+1)} \leftarrow \text{Aggregate}_{\text{node}}\left(\left\{\mathbf{e}_{nm}^{(l+1)} : m \in \mathcal{N}(n)\right\}\right)$$
$$\mathbf{h}_n^{(l+1)} \leftarrow \text{Update}_{\text{node}}\left(\mathbf{h}_n^{(l)}, \mathbf{z}_n^{(l+1)}, \mathbf{g}^{(l)}\right)$$
$$\mathbf{g}^{(l+1)} \leftarrow \text{Update}_{\text{graph}}\left(\mathbf{g}^{(l)}, \{\mathbf{h}_n^{(l+1)}\}, \{\mathbf{e}_{nm}^{(l+1)}\}\right)$$

end for
return $\{\mathbf{h}_n^{(L)}\}, \{\mathbf{e}_{nm}^{(L)}\}, \mathbf{g}^{(L)}$

13.3.6 Geometric deep learning

We have seen how permutation symmetry is a key consideration when designing deep learning models for graph-structured data. It acts as a form of inductive bias, dramatically reducing the data requirements while improving predictive performance. In applications of graph neural networks associated with spatial properties, such as graphics meshes, fluid flow simulations, or molecular structures, there are additional equivariance and invariance properties that can be built into the network architecture.

Consider the task of predicting the properties of a molecule, for example when exploring the space of candidate drugs. The molecule can be represented as a list of atoms of given types (carbon, hydrogen, nitrogen, etc.) along with the spatial coordinates of each atom expressed as a three-dimensional column vector. We can introduce an associated embedding vector for each atom n at each layer l, denoted by $\mathbf{r}_n^{(l)}$, and these vectors can be initialized with the known atom coordinates. However, the values for the elements of these vectors depends on the arbitrary choice of coordinate system, whereas the properties of the molecule do not. For example, the solubility of the molecule is unchanged if it is rotated in space or translated to a new position relative to the origin of the coordinate system, or if the coordinate system itself is reflected to give the mirror image version of the molecule. The molecular

Telegram: @uni_k

www.konkur.in

properties should therefore be invariant under such transformations.

By making careful choices of the functional forms for the update and aggregation operations (Satorras, Hoogeboom, and Welling, 2021), the new embeddings $\mathbf{r}_n^{(l)}$ can be incorporated into the graph neural network update equations (13.29) to (13.31) to achieve the required symmetry properties:

$$\mathbf{e}_{nm}^{(l+1)} = \text{Update}_{\text{edge}}\left(\mathbf{e}_{nm}^{(l)}, \mathbf{h}_n^{(l)}, \mathbf{h}_m^{(l)}, \|\mathbf{r}_n^{(l)} - \mathbf{r}_m^{(l)}\|^2\right) \tag{13.38}$$

$$\mathbf{r}_n^{(l+1)} = \mathbf{r}_n^{(l)} + C \sum_{(n,m)\in\mathcal{E}} \left(\mathbf{r}_n^{(l)} - \mathbf{r}_m^{(l)}\right) \phi\left(\mathbf{e}_{nm}^{(l+1)}\right) \tag{13.39}$$

$$\mathbf{z}_n^{(l+1)} = \text{Aggregate}_{\text{node}}\left(\{\mathbf{e}_{nm}^{(l+1)} : m \in \mathcal{N}(n)\}\right) \tag{13.40}$$

$$\mathbf{h}_n^{(l+1)} = \text{Update}_{\text{node}}\left(\mathbf{h}_n^{(l)}, \mathbf{z}_n^{(l+1)}\right) \tag{13.41}$$

Note that the quantity $\|\mathbf{r}_n^{(l)} - \mathbf{r}_m^{(l)}\|^2$ represents the squared distance between the coordinates $\mathbf{r}_n^{(l)}$ and $\mathbf{r}_m^{(l)}$, and this does not depend on translations, rotations, or reflections. Also, the coordinates $\mathbf{r}_n^{(l)}$ are updated through a linear combination of the relative differences $\left(\mathbf{r}_n^{(l)} - \mathbf{r}_m^{(l)}\right)$. Here $\phi\left(\mathbf{e}_{nm}^{(l+1)}\right)$ is a general scalar function of the edge embeddings and is represented by a neural network, and the coefficient C is typically set equal to the reciprocal of the number of terms in the sum. It follows that under such transformations, the messages in (13.38), (13.40), and (13.41) are invariant and the coordinate embeddings given by (13.39) are equivariant.

Exercise 13.10

We have seen many examples of symmetries in structured data, from translations of objects within images and the permutation of node orderings on graphs, to rotations and translations of molecules in three-dimensional space. Capturing these symmetries in the structure of a deep neural network is a powerful form of inductive bias and forms the basis of a rich field of research known as *geometric deep learning* (Bronstein *et al.*, 2017; Bronstein *et al.*, 2021).

Exercises

13.1 ($\star$) Show that the permutation $(A, B, C, D, E) \to (C, E, A, D, B)$ corresponding to the two choices of node ordering in Figure 13.2 can be expressed in the form (13.5) with a permutation matrix given by (13.1).

13.2 ($\star\star$) Show that the number of edges connected to each node of a graph is given by the corresponding diagonal element of the matrix $\mathbf{A}^2$ where $\mathbf{A}$ is the adjacency matrix.

13.3 ($\star$) Draw the graph whose adjacency matrix is given by

$$\mathbf{A} = \begin{pmatrix} 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \end{pmatrix}. \tag{13.42}$$

Telegram: @uni_k

www.konkur.in

13.4 ($\star\star$) Show that the effect of pre-multiplying a data matrix $\mathbf{X}$ using a permutation matrix $\mathbf{P}$ defined by (13.3) is to create a new data matrix $\widetilde{\mathbf{X}}$ given by (13.4) whose rows are permuted according to the permutation function $\pi(\cdot)$.

13.5 ($\star\star$) Show that the transformed adjacency matrix $\widetilde{\mathbf{A}}$ defined by (13.5), where $\mathbf{P}$ is defined by (13.3), is such that both the rows and the columns are permuted according to the permutation function $\pi(\cdot)$ relative to the original adjacency matrix $\mathbf{A}$.

13.6 ($\star\star$) In this exercise we write the update equations (13.16) as graph-level equations using matrices. To keep the notation uncluttered, we omit the layer index l. First, gather the node-embedding vectors $\{\mathbf{h}_n\}$ into an $N \times D$ matrix $\mathbf{H}$ in which row n is given by $\mathbf{h}_n^{\mathrm{T}}$. Then show that the neighbourhood-aggregated vectors $\mathbf{z}_n$ given by

$$\mathbf{z}_n = \sum_{m \in \mathcal{N}(n)} \mathbf{h}_m \tag{13.43}$$

can be written in matrix form as $\mathbf{Z} = \mathbf{A}\mathbf{H}$ where $\mathbf{Z}$ is the $N \times D$ matrix in which row n is given by $\mathbf{z}_n^{\mathrm{T}}$, and $\mathbf{A}$ is the adjacency matrix. Finally, show that the argument to the nonlinear activation function in (13.16) can be written in matrix form as

$$\mathbf{A}\mathbf{H}\mathbf{W}_{\text{neigh}} + \mathbf{H}\mathbf{W}_{\text{self}} + \mathbf{1}_D\mathbf{b}^{\mathrm{T}} \tag{13.44}$$

where $\mathbf{1}_D$ is the D-dimensional column vector in which all elements are 1.

13.7 ($\star\star$) By making use of the equivariance property (13.19) for layer l of a deep graph convolutional network along with the permutation property (13.4) for the node variables, show that a complete deep graph convolutional network defined by (13.18) is also equivariant.

13.8 ($\star\star$) Explain why the aggregation function defined by (13.24), in which the attention weights are given by (13.28), is equivariant under a reordering of the nodes in the graph.

13.9 ($\star$) Show that a graph attention network in which the graph is fully connected, so that there is an edge between every pair of nodes, is equivalent to a standard transformer architecture.

13.10 ($\star\star$) When a coordinate system is translated, the location of an object defined by that coordinate system is transformed using

$$\widetilde{\mathbf{r}} = \mathbf{r} + \mathbf{c} \tag{13.45}$$

where $\mathbf{c}$ is a fixed vector describing the translation. Similarly, if the coordinate system is rotated and/or mirror reflected, the location vector of an object is transformed using

$$\widetilde{\mathbf{r}} = \mathbf{R}\mathbf{r} \tag{13.46}$$

where $\mathbf{R}$ is an *orthogonal matrix* whose inverse is given by its transpose so that

$$\mathbf{R}\mathbf{R}^{\mathrm{T}} = \mathbf{R}^{\mathrm{T}}\mathbf{R} = \mathbf{I}. \tag{13.47}$$

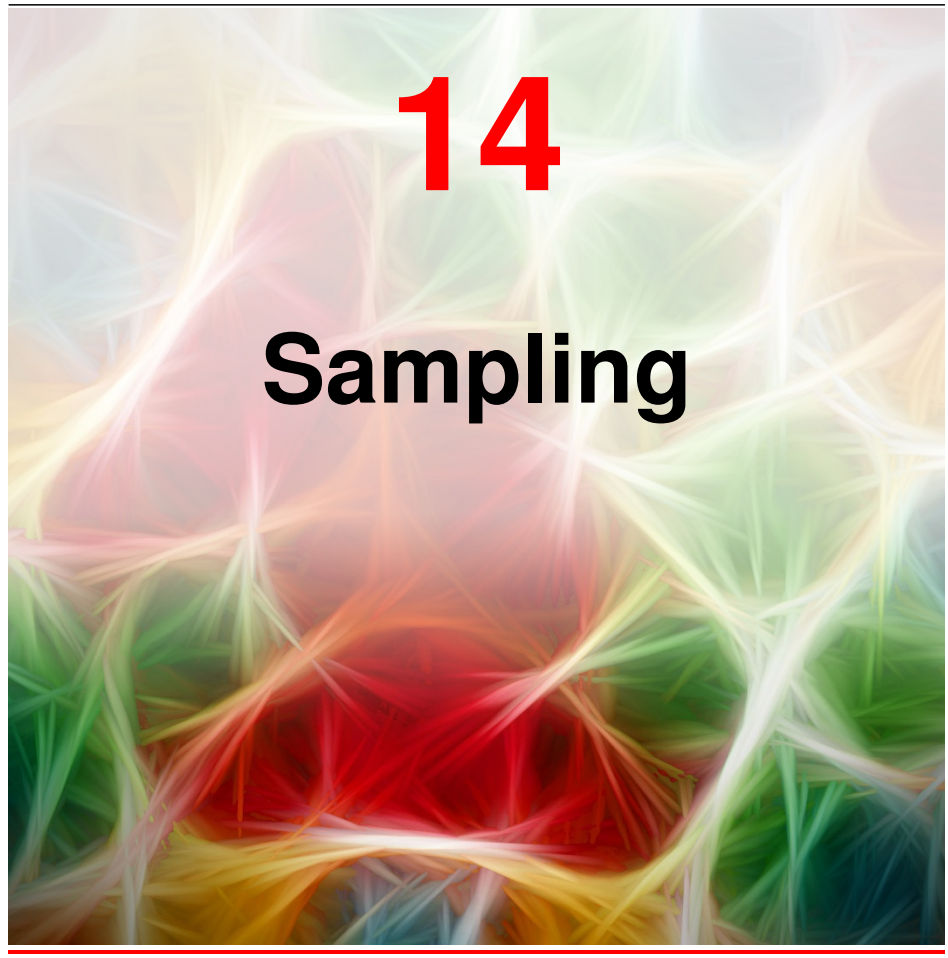
Telegram: @uni_k

www.konkur.in

Using these properties, show that under translations, rotations, and reflections, the messages in (13.38), (13.40), and (13.41) are invariant, and that the coordinate embeddings given by (13.39) are equivariant.

Telegram: @uni_k

www.konkur.in



14 Sampling

There are many situations in deep learning where we need to create synthetic examples of a variable $\mathbf{z}$ from a probability distribution $p(\mathbf{z})$. Here $\mathbf{z}$ might be a scalar and the distribution might be a univariate Gaussian, or $\mathbf{z}$ might be a high-resolution image and $p(\mathbf{z})$ might be a generative model defined by a deep neural network. The process of creating such examples is known as *sampling*, also known as *Monte Carlo sampling*. For many simple distributions there are numerical techniques that generate suitable samples directly, whereas for more complex distributions, including ones that are defined implicitly, we may need more sophisticated approaches. We adopt the convention of referring to each instantiated value as a sample, in contrast to the convention used in classical statistics whereby 'sample' refers to a set of values.

In this chapter we focus on aspects of sampling that are most relevant to deep learning. Further information on Monte Carlo methods more generally can be found in Gilks, Richardson, and Spiegelhalter (1996) and Robert and Casella (1999).

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024
C. M. Bishop, H. Bishop, *Deep Learning*, https://doi.org/10.1007/978-3-031-45468-4_14

Telegram: @uni_k

www.konkur.in

14.1. Basic Sampling Algorithms

In this section, we explore a variety of relatively simple strategies for generating random samples from a given distribution. Because the samples will be generated by a computer algorithm, they will in fact be *pseudo-random*, that is, they will be calculated using a deterministic algorithm but must nevertheless pass appropriate tests for randomness. Here we will assume that an algorithm has been provided that generates pseudo-random numbers distributed uniformly over $(0, 1)$, and indeed most software environments have such a facility built in.

14.1.1 Expectations

Although for some applications the samples themselves may be of direct interest, in other situations the goal is to evaluate *expectations* with respect to the distribution. Suppose we wish to find the expectation of a function $f(\mathbf{z})$ with respect to a probability distribution $p(\mathbf{z})$. Here, the components of $\mathbf{z}$ might comprise discrete or continuous variables or some combination of the two. For continuous variables the expectation is defined by

$$\mathbb{E}[f] = \int f(\mathbf{z})p(\mathbf{z})\,\mathrm{d}\mathbf{z} \tag{14.1}$$

where the integral is replaced by summation for discrete variables. This is illustrated schematically for a single continuous variable in Figure 14.1. We will suppose that such expectations are too complex to be evaluated exactly using analytical techniques.

The general idea behind sampling methods is to obtain a set of samples $\mathbf{z}^{(l)}$ (where $l = 1, \ldots, L$) drawn independently from the distribution $p(\mathbf{z})$. This allows the expectation (14.1) to be approximated by a finite sum:

$$\overline{f} = \frac{1}{L}\sum_{l=1}^{L} f(\mathbf{z}^{(l)}). \tag{14.2}$$

If the samples $\mathbf{z}^{(l)}$ are drawn from the distribution $p(\mathbf{z})$, then $\mathbb{E}[\overline{f}] = \mathbb{E}[f(\mathbf{z})]$ and so the estimator $\overline{f}$ has the correct mean. We can also write this in the form

Exercise 14.1

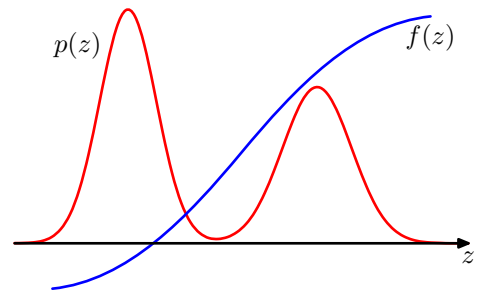


Figure 14.1 Schematic illustration of a function $f(z)$ whose expectation is to be evaluated with respect to a distribution $p(z)$.

Telegram: @uni_k

www.konkur.in

$$\mathbb{E}[f(\mathbf{z})] \simeq \frac{1}{L}\sum_{l=1}^{L} f(\mathbf{z}^{(l)}) \tag{14.3}$$

where the symbol $\simeq$ denotes that the right-hand side is an unbiased estimator of the left-hand side, that is the two sides are equal when averaged over the noise distribution.

Exercise 14.2

The variance of the estimator (14.2) is given by

$$\text{var}[\widehat{f}] = \frac{1}{L}\mathbb{E}\left[(f - \mathbb{E}[f])^2\right], \tag{14.4}$$

which is the variance of the function $f(\mathbf{z})$ under the distribution $p(\mathbf{z})$. Note that the linear decrease of this variance with increasing L does not depend on the dimensionality of $\mathbf{z}$, and that, in principle, high accuracy may be achievable with a relatively small number of samples $\{\mathbf{z}^{(l)}\}$. The problem, however, is that the samples $\{\mathbf{z}^{(l)}\}$ might not be independent, and so the effective sample size might be much smaller than the apparent sample size. Also, referring back to Figure 14.1, note that if $f(\mathbf{z})$ is small in regions where $p(\mathbf{z})$ is large and vice versa, then the expectation may be dominated by regions of small probability, implying that relatively large sample sizes will be required to achieve sufficient accuracy.

14.1.2 Standard distributions

We now consider how to generate random numbers from simple nonuniform distributions, assuming that we already have available a source of uniformly distributed random numbers. Suppose that z is uniformly distributed over the interval $(0, 1)$, and that we transform the values of z using some function $g(\cdot)$ so that $y = g(z)$. The distribution of y will be governed by

Section 2.4

$$p(y) = p(z)\left|\frac{\mathrm{d}z}{\mathrm{d}y}\right| \tag{14.5}$$

where, in this case, $p(z) = 1$. Our goal is to choose the function $g(z)$ such that the resulting values of y have some specific desired distribution $p(y)$. Integrating (14.5) we obtain

$$z = \int_{-\infty}^{y} p(\widehat{y}) \equiv h(y)\,\mathrm{d}\widehat{y} \tag{14.6}$$

Exercise 14.3

which is the indefinite integral of $p(y)$. Thus, $y = h^{-1}(z)$, and so we have to transform the uniformly distributed random numbers using a function that is the inverse of the indefinite integral of the desired distribution. This is illustrated in Figure 14.2.

Consider for example the *exponential distribution*

$$p(y) = \lambda \exp(-\lambda y) \tag{14.7}$$

where $0 \leqslant y < \infty$. In this case the lower limit of the integral in (14.6) is 0, and so $h(y) = 1 - \exp(-\lambda y)$. Thus, if we transform our uniformly distributed variable z using $y = -\lambda^{-1}\ln(1 - z)$, then y will have an exponential distribution.

Telegram: @uni_k

www.konkur.in

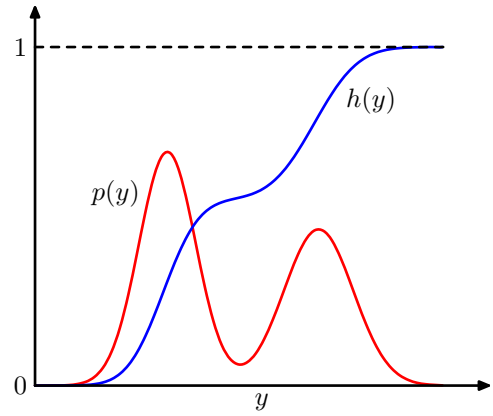


Figure 14.2 Geometrical interpretation of the transformation method for generating non-uniformly distributed random numbers. $h(y)$ is the indefinite integral of the desired distribution $p(y)$. If a uniformly distributed random variable z is transformed using $y = h^{-1}(z)$, then y will be distributed according to $p(y)$.

Another example of a distribution to which the transformation method can be applied is given by the Cauchy distribution

$$p(y) = \frac{1}{\pi}\frac{1}{1+y^2}. \tag{14.8}$$

Exercise 14.4 In this case, the inverse of the indefinite integral can be expressed in terms of the $\tan$ function.

Section 2.4 The generalization to multiple variables involves the Jacobian of the change of variables, so that

$$p(y_1,\ldots,y_M) = p(z_1,\ldots,z_M)\left|\frac{\partial(z_1,\ldots,z_M)}{\partial(y_1,\ldots,y_M)}\right|. \tag{14.9}$$

As a final example of the transformation technique, we consider the Box–Muller method for generating samples from a Gaussian distribution. First, suppose we generate pairs of uniformly distributed random numbers $z_1, z_2 \in (-1, 1)$, which we can do by transforming a variable distributed uniformly over $(0, 1)$ using $z \to 2z - 1$. Next we discard each pair unless it satisfies $z_1^2 + z_2^2 \leqslant 1$. This leads to a uniform distribution of points inside the unit circle with $p(z_1, z_2) = 1/\pi$, as illustrated in Figure 14.3. Then, for each pair z_1, z_2 we evaluate the quantities

$$y_1 = z_1\left(\frac{-2\ln r^2}{r^2}\right)^{1/2} \tag{14.10}$$

$$y_2 = z_2\left(\frac{-2\ln r^2}{r^2}\right)^{1/2} \tag{14.11}$$

Exercise 14.5 where $r^2 = z_1^2 + z_2^2$. Then the joint distribution of y_1 and y_2 is given by

$$\begin{aligned} p(y_1,y_2) &= p(z_1,z_2)\left|\frac{\partial(z_1,z_2)}{\partial(y_1,y_2)}\right| \\ &= \left[\frac{1}{\sqrt{2\pi}}\exp(-y_1^2/2)\right]\left[\frac{1}{\sqrt{2\pi}}\exp(-y_2^2/2)\right] \end{aligned} \tag{14.12}$$

Telegram: @uni_k

www.konkur.in

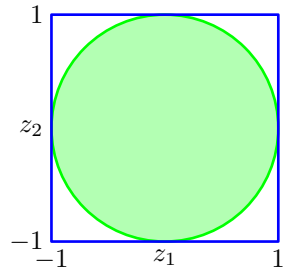


Figure 14.3 The Box–Muller method for generating Gaussian-distributed random numbers starts by generating samples from a uniform distribution inside the unit circle.

and so y_1 and y_2 are independent and each has a Gaussian distribution with zero mean and unit variance.

If y has a Gaussian distribution with zero mean and unit variance, then $\sigma y + \mu$ will have a Gaussian distribution with mean μ and variance σ^2. To generate vector-valued variables having a multivariate Gaussian distribution with mean $\boldsymbol{\mu}$ and covariance $\boldsymbol{\Sigma}$, we can make use of the *Cholesky decomposition*, which takes the form $\boldsymbol{\Sigma} = \mathbf{L}\mathbf{L}^{\mathrm{T}}$ (Deisenroth, Faisal, and Ong, 2020). Then, if $\mathbf{z}$ is a random vector whose components are independent and Gaussian distributed with zero mean and unit variance, then $\mathbf{y} = \boldsymbol{\mu} + \mathbf{L}\mathbf{z}$ will be Gaussian with mean $\boldsymbol{\mu}$ and covariance $\boldsymbol{\Sigma}$.

Exercise 14.6

Clearly, the transformation technique depends for its success on the ability to calculate and then invert the indefinite integral of the required distribution. Such operations are feasible only for a limited number of simple distributions, and so we must turn to alternative approaches in search of a more general strategy. Here we consider two techniques called *rejection sampling* and *importance sampling*. Although mainly limited to univariate distributions and thus not directly applicable to complex problems in many dimensions, they do form important components in more general strategies.

14.1.3 Rejection sampling

The rejection sampling framework allows us to sample from relatively complex distributions, subject to certain constraints. We begin by considering univariate distributions and subsequently discuss the extension to multiple dimensions.

Suppose we wish to sample from a distribution $p(\mathbf{z})$ that is not one of the simple, standard distributions considered so far and that sampling directly from $p(\mathbf{z})$ is difficult. Furthermore suppose, as is often the case, that we are easily able to evaluate $p(\mathbf{z})$ for any given value of $\mathbf{z}$, up to some normalizing constant Z, so that

$$p(z) = \frac{1}{Z_p}\widetilde{p}(z) \tag{14.13}$$

where $\widetilde{p}(z)$ can readily be evaluated, but Z_p is unknown.

To apply rejection sampling, we need some simpler distribution $q(z)$, sometimes called a *proposal distribution*, from which we can readily draw samples. We next introduce a constant k whose value is chosen such that $kq(z) \geqslant \widetilde{p}(z)$ for all values of z. The function $kq(z)$ is called the comparison function and is illustrated

Telegram: @uni_k

www.konkur.in

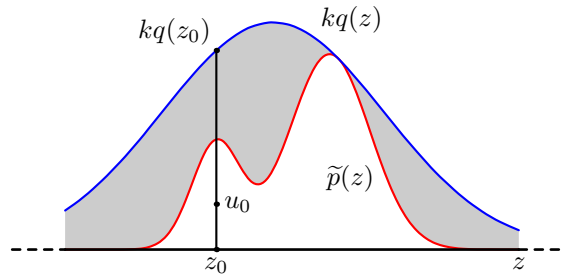


Figure 14.4 In the rejection sampling method, samples are drawn from a simple distribution $q(z)$ and rejected if they fall in the grey area between the unnormalized distribution $\widetilde{p}(z)$ and the scaled distribution $kq(z)$. The resulting samples are distributed according to $p(z)$, which is the normalized version of $\widetilde{p}(z)$.

for a univariate distribution in Figure 14.4. Each step of the rejection sampler involves generating two random numbers. First, we generate a number z_0 from the distribution $q(z)$. Next, we generate a number u_0 from the uniform distribution over $[0, kq(z_0)]$. This pair of random numbers has uniform distribution under the curve of the function $kq(z)$. Finally, if $u_0 > \widetilde{p}(z_0)$ then the sample is rejected, otherwise u_0 is retained. Thus, the pair is rejected if it lies in the grey shaded region in Figure 14.4. The remaining pairs then have uniform distribution under the curve of $\widetilde{p}(z)$, and hence the corresponding z values are distributed according to $p(z)$, as desired.

Exercise 14.7

The original values of z are generated from the distribution $q(z)$, and these samples are then accepted with probability $\widetilde{p}(z)/kq(z)$, and so the probability that a sample will be accepted is given by

$$\begin{aligned} p(\text{accept}) &= \int \{\widetilde{p}(z)/kq(z)\}\, q(z)\, \mathrm{d}z \\ &= \frac{1}{k} \int \widetilde{p}(z)\, \mathrm{d}z. \end{aligned} \tag{14.14}$$

Thus, the fraction of points that are rejected by this method depends on the ratio of the area under the unnormalized distribution $\widetilde{p}(z)$ to the area under the curve $kq(z)$. We therefore see that the constant k should be as small as possible subject to the limitation that $kq(z)$ must be nowhere less than $\widetilde{p}(z)$.

As an illustration of the use of rejection sampling, consider the task of sampling from the gamma distribution

$$\mathrm{Gam}(z|a, b) = \frac{b^a z^{a-1} \exp(-bz)}{\Gamma(a)}, \tag{14.15}$$

which, for $a > 1$, has a bell-shaped form, as shown in Figure 14.5. A suitable proposal distribution is therefore the Cauchy (14.8) because this too is bell-shaped and because we can use the transformation method, discussed earlier, to sample from it. We need to generalize the Cauchy slightly to ensure that it nowhere has a smaller value than the gamma distribution. This can be achieved by transforming a uniform random variable y using $z = b \tan y + c$, which gives random numbers distributed according to

Exercise 14.8

$$q(z) = \frac{k}{1 + (z - c)^2/b^2}. \tag{14.16}$$

Telegram: @uni_k

www.konkur.in

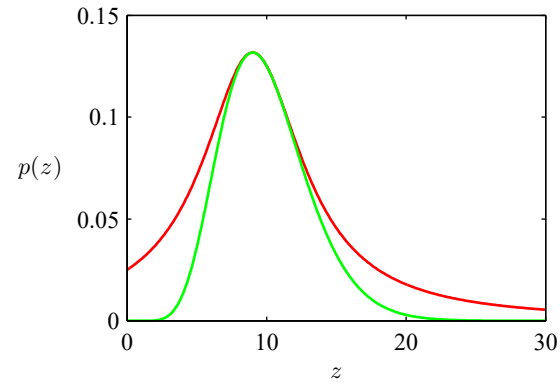


Figure 14.5 Plot showing the gamma distribution given by (14.15) as the green curve, with a scaled Cauchy proposal distribution shown by the red curve. Samples from the gamma distribution can be obtained by sampling from the Cauchy and then applying the rejection sampling criterion.

The minimum reject rate is obtained by setting $c = a-1$, and $b^2 = 2a-1$ and choosing the constant k to be as small as possible while still satisfying the requirement $kq(z) \geqslant \widetilde{p}(z)$. The resulting comparison function is also illustrated in Figure 14.5.

14.1.4 Adaptive rejection sampling

In many instances where we might wish to apply rejection sampling, it can be difficult to determine a suitable analytic form for the envelope distribution $q(z)$. An alternative approach is to construct the envelope function on the fly based on measured values of the distribution $p(z)$ (Gilks and Wild, 1992). Constructing an envelope function is particularly straightforward when $p(z)$ is log concave, in other words when $\ln p(z)$ has derivatives that are non-increasing functions of z. The construction of a suitable envelope function is illustrated graphically in Figure 14.6.

The function $\ln p(z)$ and its gradient are evaluated at some initial set of grid points, and the intersections of the resulting tangent lines are used to construct the envelope function. Next a sample value is drawn from the envelope distribution. This is straightforward because the log of the envelope distribution is a succession of linear functions, and hence the envelope distribution itself comprises a piecewise exponential distribution of the form

Exercise 14.10

$$q(z) = k_i \lambda_i \exp\{-\lambda_i(z - z_{i-1})\}, \qquad z_{i-1} < z \leqslant z_i. \tag{14.17}$$

Once a sample has been drawn, the usual rejection criterion can be applied. If the sample is accepted, then it will be a draw from the desired distribution. If, however, the sample is rejected, then it is incorporated into the set of grid points, a new tangent line is computed, and the envelope function is thereby refined. As the number of grid points increases, so the envelope function becomes a better approximation of the desired distribution $p(z)$ and the probability of rejection decreases.

There is a variant of the algorithm exists that avoids the evaluation of derivatives (Gilks, 1992). The adaptive rejection sampling framework can also be extended to distributions that are not log concave, simply by following each rejection sampling

Telegram: @uni_k

www.konkur.in

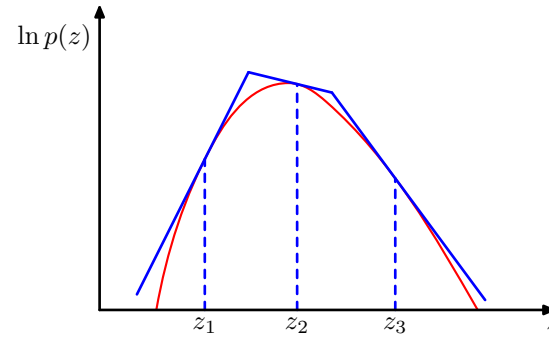


Figure 14.6 In rejection sampling, if a distribution is log concave then an envelope function can be constructed using the tangent lines computed at a set of grid points. If a sample point is rejected, it is added to the set of grid points and used to refine the envelope distribution.

step with a Metropolis–Hastings step (to be discussed in Section 14.2.3), giving rise to *adaptive rejection Metropolis* sampling (Gilks, Best, and Tan, 1995).

For rejection sampling to be of practical value, we require that the comparison function is close to the required distribution so that the rate of rejection is kept to a minimum. Now let us examine what happens when we try to use rejection sampling in spaces of high dimensionality. Consider, for illustration, a somewhat artificial problem in which we wish to sample from a zero-mean multivariate Gaussian distribution with covariance $\sigma_p^2\mathbf{I}$, where $\mathbf{I}$ is the unit matrix, by rejection sampling from a proposal distribution that is itself a zero-mean Gaussian distribution having covariance $\sigma_q^2\mathbf{I}$. Clearly, we must have $\sigma_q^2 \geqslant \sigma_p^2$ to ensure that there exists a k such that $kq(z) \geqslant p(z)$. In D-dimensions, the optimum value of k is given by $k = (\sigma_q/\sigma_p)^D$, as illustrated for $D = 1$ in Figure 14.7. The acceptance rate will be the ratio of volumes under $p(z)$ and $kq(z)$, which, because both distributions are normalized, is just $1/k$. Thus, the acceptance rate diminishes exponentially with dimensionality. Even if σ_q exceeds σ_p by just 1%, for $D = 1{,}000$ the acceptance ratio will be approximately $1/20{,}000$. In this illustrative example, the comparison function is close to the required distribution. For more practical examples, where the desired distribution may be multimodal and sharply peaked, it will be extremely difficult to find

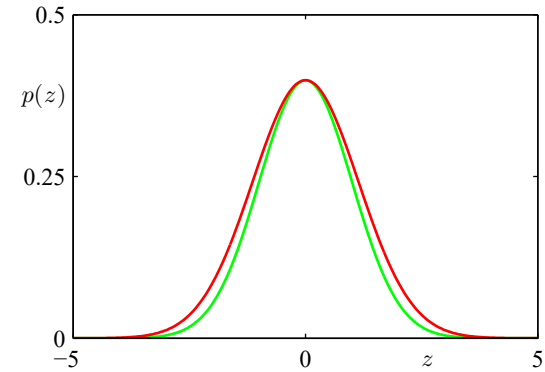


Figure 14.7 Illustrative example used to highlight a limitation of rejection sampling. Samples are drawn from a Gaussian distribution $p(z)$ shown by the green curve, by using rejection sampling from a proposal distribution $q(z)$ that is also Gaussian and whose scaled version $kq(z)$ is shown by the red curve.

Telegram: @uni_k

www.konkur.in

a good proposal distribution and comparison function. Furthermore, the exponential decrease of the acceptance rate with dimensionality is a generic feature of rejection sampling. Although rejection can be a useful technique in one or two dimensions, it is unsuited to problems of high dimensionality. It can, however, play a role as a subroutine in more sophisticated algorithms for sampling in high-dimensional spaces.

14.1.5 Importance sampling

One reason for wishing to sample from complicated probability distributions is to evaluate expectations of the form (14.1). The technique of *importance sampling* provides a framework for approximating expectations directly but does not itself provide a mechanism for drawing samples from a distribution $p(\mathbf{z})$.

The finite sum approximation to the expectation, given by (14.2), depends on being able to draw samples from the distribution $p(\mathbf{z})$. Suppose, however, that it is impractical to sample directly from $p(\mathbf{z})$ but that we can evaluate $p(\mathbf{z})$ easily for any given value of $\mathbf{z}$. One simplistic strategy for evaluating expectations would be to discretize $\mathbf{z}$-space into a uniform grid and to evaluate the integrand as a sum of the form

$$\mathbb{E}[f] \simeq \sum_{l=1}^{L} p(\mathbf{z}^{(l)}) f(\mathbf{z}^{(l)}). \tag{14.18}$$

An obvious problem with this approach is that the number of terms in the summation grows exponentially with the dimensionality of $\mathbf{z}$. Furthermore, as we have already noted, the kinds of probability distributions of interest will often have much of their mass confined to relatively small regions of $\mathbf{z}$-space and so uniform sampling will be very inefficient because in high-dimensional problems, only a very small proportion of the samples will make a significant contribution to the sum. We would really like to choose sample points from regions where $p(\mathbf{z})$ is large or ideally where the product $p(\mathbf{z})f(\mathbf{z})$ is large.

As with rejection sampling, importance sampling is based a proposal distribution $q(\mathbf{z})$ from which it is easy to draw samples, as illustrated in Figure 14.8. We can then express the expectation in the form of a finite sum over samples $\{\mathbf{z}^{(l)}\}$ drawn from $q(\mathbf{z})$:

$$\begin{aligned} \mathbb{E}[f] &= \int f(\mathbf{z}) p(\mathbf{z}) \, \mathrm{d}\mathbf{z} \\ &= \int f(\mathbf{z}) \frac{p(\mathbf{z})}{q(\mathbf{z})} q(\mathbf{z}) \, \mathrm{d}\mathbf{z} \\ &\simeq \frac{1}{L} \sum_{l=1}^{L} \frac{p(\mathbf{z}^{(l)})}{q(\mathbf{z}^{(l)})} f(\mathbf{z}^{(l)}). \end{aligned} \tag{14.19}$$

The quantities $r_l = p(\mathbf{z}^{(l)})/q(\mathbf{z}^{(l)})$ are known as *importance weights*, and they correct the bias introduced by sampling from the wrong distribution. Note that, unlike rejection sampling, all the samples generated are retained.

Often the distribution $p(\mathbf{z})$ can be evaluated only up to a normalization constant, so that $p(\mathbf{z}) = \widetilde{p}(\mathbf{z})/Z_p$ where $\widetilde{p}(\mathbf{z})$ can be evaluated easily, whereas Z_p is unknown.

Telegram: @uni_k

www.konkur.in

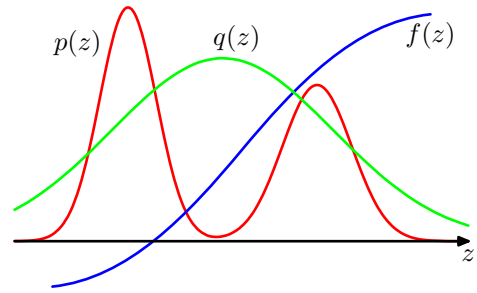


Figure 14.8 Importance sampling addresses the problem of evaluating the expectation of a function $f(z)$ with respect to a distribution $p(z)$ from which it is difficult to draw samples directly. Instead, samples $\{z^{(l)}\}$ are drawn from a simpler distribution $q(z)$, and the corresponding terms in the summation are weighted by the ratios $p(z^{(l)})/q(z^{(l)})$.

Similarly, we may wish to use an importance sampling distribution $q(\mathbf{z}) = \widetilde{q}(\mathbf{z})/Z_q$, which has the same property. We then have

$$\begin{aligned}\mathbb{E}[f] &= \int f(\mathbf{z})p(\mathbf{z})\,\mathrm{d}\mathbf{z} \\ &= \frac{Z_q}{Z_p}\int f(\mathbf{z})\frac{\widetilde{p}(\mathbf{z})}{\widetilde{q}(\mathbf{z})}q(\mathbf{z})\,\mathrm{d}\mathbf{z} \\ &\simeq \frac{Z_q}{Z_p}\frac{1}{L}\sum_{l=1}^{L}\widetilde{r}_l f(\mathbf{z}^{(l)}) \end{aligned} \tag{14.20}$$

where $\widetilde{r}_l = \widetilde{p}(\mathbf{z}^{(l)})/\widetilde{q}(\mathbf{z}^{(l)})$. We can use the same sample set to evaluate the ratio Z_p/Z_q with the result

$$\begin{aligned}\frac{Z_p}{Z_q} &= \frac{1}{Z_q}\int \widetilde{p}(\mathbf{z})\,\mathrm{d}\mathbf{z} = \int \frac{\widetilde{p}(\mathbf{z})}{\widetilde{q}(\mathbf{z})}q(\mathbf{z})\,\mathrm{d}\mathbf{z} \\ &\simeq \frac{1}{L}\sum_{l=1}^{L}\widetilde{r}_l \end{aligned} \tag{14.21}$$

and hence the expectation in (14.20) is given by a weighted sum:

$$\mathbb{E}[f] \simeq \sum_{l=1}^{L} w_l f(\mathbf{z}^{(l)}) \tag{14.22}$$

where we have defined

$$w_l = \frac{\widetilde{r}_l}{\sum_m \widetilde{r}_m} = \frac{\widetilde{p}(\mathbf{z}^{(l)})/q(\mathbf{z}^{(l)})}{\sum_m \widetilde{p}(\mathbf{z}^{(m)})/q(\mathbf{z}^{(m)})}. \tag{14.23}$$

Note that $\{w_l\}$ are non-negative numbers that sum to one.

As with rejection sampling, the success of importance sampling depends crucially on how well the sampling distribution $q(\mathbf{z})$ matches the desired distribution $p(\mathbf{z})$. If, as is often the case, $p(\mathbf{z})f(\mathbf{z})$ is strongly varying and has a significant proportion of its mass concentrated over relatively small regions of $\mathbf{z}$-space, then the

Telegram: @uni_k

www.konkur.in

set of importance weights $\{r_l\}$ may be dominated by a few weights having large values, with the remaining weights being relatively insignificant. Thus, the effective sample size can be much smaller than the apparent sample size L. The problem is even more severe if none of the samples falls in the regions where $p(\mathbf{z})f(\mathbf{z})$ is large. In that case, the apparent variances of r_l and $r_l f(\mathbf{z}^{(l)})$ may be small even though the estimate of the expectation may be severely wrong. Hence, a major drawback of importance sampling is its potential to produce results that are arbitrarily in error and with no diagnostic indication. This also highlights a key requirement for the sampling distribution $q(\mathbf{z})$, namely that it should not be small or zero in regions where $p(\mathbf{z})$ may be significant.

14.1.6 Sampling-importance-resampling

The rejection sampling method discussed in Section 14.1.3 depends in part for its success on the determination of a suitable value for the constant k. For many pairs of distributions $p(\mathbf{z})$ and $q(\mathbf{z})$, it will be impractical to determine a suitable value for k as any value that is sufficiently large to guarantee a bound on the desired distribution will lead to impractically small acceptance rates.

As with rejection sampling, the *sampling-importance-resampling* approach also makes use of a sampling distribution $q(\mathbf{z})$ but avoids having to determine the constant k. There are two stages to the scheme. In the first stage, L samples $\mathbf{z}^{(1)}, \ldots, \mathbf{z}^{(L)}$ are drawn from $q(\mathbf{z})$. Then in the second stage, weights $w_1, \ldots, w_L$ are constructed using (14.23). Finally, a second set of L samples is drawn from the discrete distribution $(\mathbf{z}^{(1)}, \ldots, \mathbf{z}^{(L)})$ with probabilities given by the weights $(w_1, \ldots, w_L)$.

The resulting L samples are only approximately distributed according to $p(\mathbf{z})$, but the distribution becomes correct in the limit $L \to \infty$. To see this, consider the univariate case, and note that the cumulative distribution of the resampled values is given by

$$\begin{aligned} p(z \leqslant a) &= \sum_{l: z^{(l)} \leqslant a} w_l \\ &= \frac{\sum_l I(z^{(l)} \leqslant a) \widetilde{p}(z^{(l)})/q(z^{(l)})}{\sum_l \widetilde{p}(z^{(l)})/q(z^{(l)})} \end{aligned} \tag{14.24}$$

where $I(.)$ is the indicator function (which equals 1 if its argument is true and 0 otherwise). Taking the limit $L \to \infty$ and assuming suitable regularity of the distributions, we can replace the sums by integrals weighted according to the original

Telegram: @uni_k

www.konkur.in

sampling distribution $q(z)$:

$$\begin{aligned} p(z \leqslant a) &= \frac{\int I(z \leqslant a)\,\{\widetilde{p}(z)/q(z)\}\,q(z)\,\mathrm{d}z}{\int \{\widetilde{p}(z)/q(z)\}\,q(z)\,\mathrm{d}z} \\ &= \frac{\int I(z \leqslant a)\widetilde{p}(z)\,\mathrm{d}z}{\int \widetilde{p}(z)\,\mathrm{d}z} \\ &= \int I(z \leqslant a)p(z)\,\mathrm{d}z, \end{aligned} \tag{14.25}$$

which is the cumulative distribution function of $p(z)$. Again, we see that normalization of $p(z)$ is not required.

For a finite value of L and a given initial sample set, the resampled values will only approximately be drawn from the desired distribution. As with rejection sampling, the approximation improves as the sampling distribution $q(\mathbf{z})$ gets closer to the desired distribution $p(\mathbf{z})$. When $q(\mathbf{z}) = p(\mathbf{z})$, the initial samples $(\mathbf{z}^{(1)}, \ldots, \mathbf{z}^{(L)})$ have the desired distribution and the weights $w_n = 1/L$, so that the resampled values also have the desired distribution.

If moments with respect to the distribution $p(\mathbf{z})$ are required, then they can be evaluated directly using the original samples together with the weights, because

$$\begin{aligned} \mathbb{E}[f(\mathbf{z})] &= \int f(\mathbf{z})p(\mathbf{z})\,\mathrm{d}\mathbf{z} \\ &= \frac{\int f(\mathbf{z})[\widetilde{p}(\mathbf{z})/q(\mathbf{z})]q(\mathbf{z})\,\mathrm{d}\mathbf{z}}{\int [\widetilde{p}(\mathbf{z})/q(\mathbf{z})]q(\mathbf{z})\,\mathrm{d}\mathbf{z}} \\ &\simeq \sum_{l=1}^{L} w_l f(\mathbf{z}_l). \end{aligned} \tag{14.26}$$

14.2. Markov Chain Monte Carlo

In the previous section, we discussed the rejection sampling and importance sampling strategies for evaluating expectations of functions, and we saw that they suffer from severe limitations particularly in spaces of high dimensionality. We therefore turn in this section to a very general and powerful framework called Markov chain Monte Carlo, which allows sampling from a large class of distributions and which scales well with the dimensionality of the sample space. Markov chain Monte Carlo methods have their origins in physics (Metropolis and Ulam, 1949), and it was only

Telegram: @uni_k

www.konkur.in

towards the end of the 1980s that they started to have a significant impact in the field of statistics.

As with rejection and importance sampling, we again sample from a proposal distribution. This time, however, we maintain a record of the current state $\mathbf{z}^{(\tau)}$, and the proposal distribution $q(\mathbf{z}|\mathbf{z}^{(\tau)})$ is conditioned on this current state, and so the sequence of samples $\mathbf{z}^{(1)}, \mathbf{z}^{(2)}, \ldots$ forms a Markov chain. Again, if we write $p(\mathbf{z}) = \widetilde{p}(\mathbf{z})/Z_p$, we will assume that $\widetilde{p}(\mathbf{z})$ can readily be evaluated for any given value of $\mathbf{z}$, although the value of Z_p may be unknown. The proposal distribution is chosen to be sufficiently simple that it is straightforward to draw samples from it directly. At each cycle of the algorithm, we generate a candidate sample $\mathbf{z}^\star$ from the proposal distribution and then accept the sample according to an appropriate criterion.

Section 14.2.2

14.2.1 The Metropolis algorithm

In the basic *Metropolis* algorithm (Metropolis *et al.*, 1953), we assume that the proposal distribution is symmetric, that is $q(\mathbf{z}_A|\mathbf{z}_B) = q(\mathbf{z}_B|\mathbf{z}_A)$ for all values of $\mathbf{z}_A$ and $\mathbf{z}_B$. The candidate sample is then accepted with probability

$$A(\mathbf{z}^\star, \mathbf{z}^{(\tau)}) = \min\left(1, \frac{\widetilde{p}(\mathbf{z}^\star)}{\widetilde{p}(\mathbf{z}^{(\tau)})}\right). \tag{14.27}$$

This can be achieved by choosing a random number u with uniform distribution over the unit interval $(0, 1)$ and then accepting the sample if $A(\mathbf{z}^\star, \mathbf{z}^{(\tau)}) > u$. Note that if the step from $\mathbf{z}^{(\tau)}$ to $\mathbf{z}^\star$ causes an increase in the value of $p(\mathbf{z})$, then the candidate point is certain to be kept.

If the candidate sample is accepted, then $\mathbf{z}^{(\tau+1)} = \mathbf{z}^\star$, otherwise the candidate point $\mathbf{z}^\star$ is discarded, $\mathbf{z}^{(\tau+1)}$ is set to $\mathbf{z}^{(\tau)}$, and another candidate sample is drawn from the distribution $q(\mathbf{z}|\mathbf{z}^{(\tau+1)})$. This is in contrast to rejection sampling, where rejected samples are simply discarded. In the Metropolis algorithm, when a candidate point is rejected, the previous sample is included in the final list of samples, leading to multiple copies of samples. Of course, in a practical implementation, only a single copy of each retained sample would be kept, along with an integer weighting factor recording how many times that state appears. As we will see, if $q(\mathbf{z}_A|\mathbf{z}_B)$ is positive for any values of $\mathbf{z}_A$ and $\mathbf{z}_B$ (this is a sufficient but not necessary condition), the distribution of $\mathbf{z}^{(\tau)}$ tends to $p(\mathbf{z})$ as $\tau \to \infty$. It should be emphasized, however, that the sequence $\mathbf{z}^{(1)}, \mathbf{z}^{(2)}, \ldots$ is not a set of independent samples from $p(\mathbf{z})$ because successive samples are highly correlated. If we wish to obtain independent samples, then we can discard most of the sequence and just retain every Mth sample. For M sufficiently large, the retained samples will for all practical purposes be independent. The Metropolis algorithm in summarized in Algorithm 14.1. Figure 14.9 shows a simple illustrative example of sampling from a two-dimensional Gaussian distribution using the Metropolis algorithm in which the proposal distribution is an isotropic Gaussian.

Further insight into the nature of Markov chain Monte Carlo algorithms can be gleaned by looking at the properties of a specific example, namely a simple random

Telegram: @uni_k

www.konkur.in

Algorithm 14.1: Metropolis sampling

Input: Unnormalized distribution $\widetilde{p}(\mathbf{z})$
Proposal distribution $q(\mathbf{z}|\widehat{\mathbf{z}})$
Initial state $\mathbf{z}^{(0)}$
Number of iterations T
Output: $\mathbf{z} \sim \widetilde{p}(\mathbf{z})$

$\mathbf{z}_{\text{prev}} \leftarrow \mathbf{z}^{(0)}$
// Iterative message-passing
for $\tau \in \{1, \ldots, T\}$ **do**
  $\mathbf{z}^\star \sim q(\mathbf{z}|\mathbf{z}_{\text{prev}})$ // Sample from proposal distribution
  $u \sim \mathcal{U}(0, 1)$ // Sample from uniform
  if $\widetilde{p}(\mathbf{z}^\star) \,/\, \widetilde{p}(\mathbf{z}_{\text{prev}}) > u$ **then**
    $\mathbf{z}_{\text{prev}} \leftarrow \mathbf{z}^\star$ // $\mathbf{z}^{(\tau)} = \mathbf{z}^\star$
  else
    $\mathbf{z}_{\text{prev}} \leftarrow \mathbf{z}_{\text{prev}}$ // $\mathbf{z}^{(\tau)} = \mathbf{z}^{(\tau-1)}$
  end if
end for
return $\mathbf{z}_{\text{prev}}$ // $\mathbf{z}^{(T)}$

walk. Consider a state space z consisting of the integers, with probabilities

$$p(z^{(\tau+1)} = z^{(\tau)}) = 0.5 \tag{14.28}$$
$$p(z^{(\tau+1)} = z^{(\tau)} + 1) = 0.25 \tag{14.29}$$
$$p(z^{(\tau+1)} = z^{(\tau)} - 1) = 0.25 \tag{14.30}$$

where $z^{(\tau)}$ denotes the state at step τ. If the initial state is $z^{(0)} = 0$, then by symmetry the expected state at time τ will also be zero $\mathbb{E}[z^{(\tau)}] = 0$, and similarly it is easily seen that $\mathbb{E}[(z^{(\tau)})^2] = \tau/2$. Thus, after τ steps, the random walk has travelled only a distance that on average is proportional to the square root of τ. This square root dependence is typical of random walk behaviour and shows that random walks are very inefficient in exploring the state space. As we will see, a central goal in designing Markov chain Monte Carlo methods is to avoid random walk behaviour.

Exercise 14.11

14.2.2 Markov chains

Before discussing Markov chain Monte Carlo methods in more detail, it is useful to study some general properties of Markov chains. In particular, we ask under what circumstances will a Markov chain converge to the desired distribution. A first-order

Telegram: @uni_k

www.konkur.in

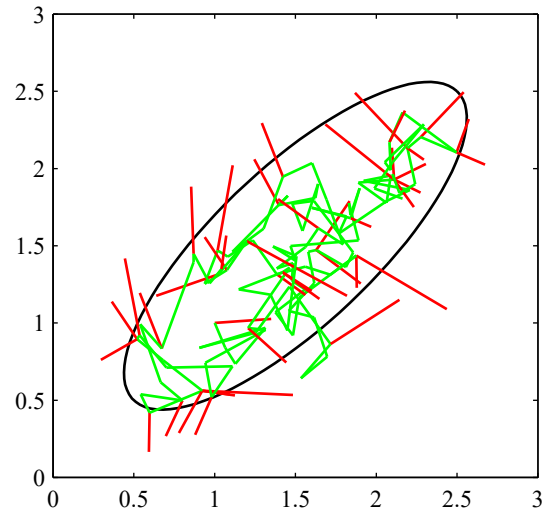


Figure 14.9 A simple illustration using the Metropolis algorithm to sample from a Gaussian distribution whose one standard-deviation contour is shown by the ellipse. The proposal distribution is an isotropic Gaussian distribution whose standard deviation is 0.2. Steps that are accepted are shown as green lines, and rejected steps are shown in red. A total of 150 candidate samples are generated, of which 43 are rejected.

Markov chain is defined to be a series of random variables $\mathbf{z}^{(1)}, \ldots, \mathbf{z}^{(M)}$ such that the following conditional independence property holds for $m \in \{1, \ldots, M-1\}$:

$$p(\mathbf{z}^{(m+1)}|\mathbf{z}^{(1)}, \ldots, \mathbf{z}^{(m)}) = p(\mathbf{z}^{(m+1)}|\mathbf{z}^{(m)}), \tag{14.31}$$

Figure 11.29

which can be represented as a directed graphical model in the form of a chain. We can then specify the Markov chain by giving the probability distribution for the initial variable $p(\mathbf{z}^{(0)})$ together with the conditional distributions for subsequent variables in the form of *transition probabilities* $T_m(\mathbf{z}^{(m)}, \mathbf{z}^{(m+1)}) \equiv p(\mathbf{z}^{(m+1)}|\mathbf{z}^{(m)})$. A Markov chain is called *homogeneous* if the transition probabilities are the same for all m.

The marginal probability for a particular variable can be expressed in terms of the marginal probability for the previous variable in the chain:

$$p(\mathbf{z}^{(m+1)}) = \int p(\mathbf{z}^{(m+1)}|\mathbf{z}^{(m)}) p(\mathbf{z}^{(m)}) \, \mathrm{d}\mathbf{z}^{(m)} \tag{14.32}$$

where the integral is replaced by a summation for discrete variables. A distribution is said to be invariant, or stationary, with respect to a Markov chain if each step in the chain leaves that distribution invariant. Thus, for a homogeneous Markov chain with transition probabilities $T(\mathbf{z}', \mathbf{z})$, the distribution $p^\star(\mathbf{z})$ is invariant if

$$p^\star(\mathbf{z}) = \int T(\mathbf{z}', \mathbf{z}) p^\star(\mathbf{z}') \, \mathrm{d}\mathbf{z}'. \tag{14.33}$$

Note that a given Markov chain may have more than one invariant distribution. For instance, if the transition probabilities are given by the identity transformation, then any distribution will be invariant.

Telegram: @uni_k

www.konkur.in

A sufficient (but not necessary) condition for ensuring that the required distribution $p(\mathbf{z})$ is invariant is to choose the transition probabilities to satisfy the property of *detailed balance*, defined by

$$p^\star(\mathbf{z})T(\mathbf{z},\mathbf{z}') = p^\star(\mathbf{z}')T(\mathbf{z}',\mathbf{z}) \tag{14.34}$$

for the particular distribution $p^\star(\mathbf{z})$. It is easily seen that a transition probability that satisfies detailed balance with respect to a particular distribution will leave that distribution invariant, because

$$\int p^\star(\mathbf{z}')T(\mathbf{z}',\mathbf{z})\,\mathrm{d}\mathbf{z}' = \int p^\star(\mathbf{z})T(\mathbf{z},\mathbf{z}')\,\mathrm{d}\mathbf{z}' \tag{14.35}$$

$$= p^\star(\mathbf{z})\int p(\mathbf{z}'|\mathbf{z})\,\mathrm{d}\mathbf{z}' \tag{14.36}$$

$$= p^\star(\mathbf{z}). \tag{14.37}$$

A Markov chain that respects detailed balance is said to be *reversible*.

Our goal is to use Markov chains to sample from a given distribution. We can achieve this if we set up a Markov chain such that the desired distribution is invariant. However, we must also require that for $m \to \infty$, the distribution $p(\mathbf{z}^{(m)})$ converges to the required invariant distribution $p^\star(\mathbf{z})$, irrespective of the choice of initial distribution $p(\mathbf{z}^{(0)})$. This property is called *ergodicity*, and the invariant distribution is then called the *equilibrium* distribution. Clearly, an ergodic Markov chain can have only one equilibrium distribution. It can be shown that a homogeneous Markov chain will be ergodic, subject only to weak restrictions on the invariant distribution and the transition probabilities (Neal, 1993).

In practice we often construct the transition probabilities from a set of 'base' transitions $B_1,\ldots,B_K$. This can be achieved through a mixture distribution of the form

$$T(\mathbf{z}',\mathbf{z}) = \sum_{k=1}^{K}\alpha_k B_k(\mathbf{z}',\mathbf{z}) \tag{14.38}$$

for some set of mixing coefficients $\alpha_1,\ldots,\alpha_K$ satisfying $\alpha_k \geqslant 0$ and $\sum_k \alpha_k = 1$. Alternatively, the base transitions may be combined through successive application, so that

$$T(\mathbf{z}',\mathbf{z}) = \sum_{\mathbf{z}_1}\ldots\sum_{\mathbf{z}_{n-1}} B_1(\mathbf{z}',\mathbf{z}_1)\ldots B_{K-1}(\mathbf{z}_{K-2},\mathbf{z}_{K-1})B_K(\mathbf{z}_{K-1},\mathbf{z}). \tag{14.39}$$

If a distribution is invariant with respect to each of the base transitions, then clearly it will also be invariant with respect to either of the $T(\mathbf{z}',\mathbf{z})$ given by (14.38) or (14.39). For the mixture (14.38), if each of the base transitions satisfies detailed balance, then the mixture transition T will also satisfy detailed balance. This does not hold for the transition probability constructed using (14.39), although by symmetrizing the order of application of the base transitions, namely $B_1, B_2, \ldots, B_K, B_K, \ldots, B_2, B_1$, detailed balance can be restored. A common example of the use of composite transition probabilities is where each base transition changes only a subset of the variables.

Telegram: @uni_k

www.konkur.in

14.2.3 The Metropolis–Hastings algorithm

Earlier we introduced the basic Metropolis algorithm without actually demonstrating that it samples from the required distribution. Before giving a proof, we first discuss a generalization, known as the *Metropolis–Hastings* algorithm (Hastings, 1970), which applies when the proposal distribution is no longer a symmetric function of its arguments. In particular at step τ of the algorithm, in which the current state is $\mathbf{z}^{(\tau)}$, we draw a sample $\mathbf{z}^\star$ from the distribution $q_k(\mathbf{z}|\mathbf{z}^{(\tau)})$ and then accept it with probability $A_k(\mathbf{z}^\star, \mathbf{z}^{(\tau)})$ where

$$A_k(\mathbf{z}^\star, \mathbf{z}^{(\tau)}) = \min\left(1, \frac{\widetilde{p}(\mathbf{z}^\star)q_k(\mathbf{z}^{(\tau)}|\mathbf{z}^\star)}{\widetilde{p}(\mathbf{z}^{(\tau)})q_k(\mathbf{z}^\star|\mathbf{z}^{(\tau)})}\right). \tag{14.40}$$

Here k labels the members of the set of possible transitions being considered. Again, evaluating the acceptance criterion does not require knowledge of the normalizing constant Z_p in the probability distribution $p(\mathbf{z}) = \widetilde{p}(\mathbf{z})/Z_p$. For a symmetric proposal distribution, the Metropolis–Hastings criterion (14.40) reduces to the standard Metropolis criterion given by (14.27). Metropolis–Hastings sampling is summarized in Algorithm 14.2.

We can show that $p(\mathbf{z})$ is an invariant distribution of the Markov chain defined by the Metropolis–Hastings algorithm by showing that detailed balance, defined by (14.34), is satisfied. Using (14.40) we have

$$\begin{aligned} p(\mathbf{z})q_k(\mathbf{z}'|\mathbf{z})A_k(\mathbf{z}', \mathbf{z}) &= \min\left(p(\mathbf{z})q_k(\mathbf{z}'|\mathbf{z}), p(\mathbf{z}')q_k(\mathbf{z}|\mathbf{z}')\right) \\ &= \min\left(p(\mathbf{z}')q_k(\mathbf{z}|\mathbf{z}'), p(\mathbf{z})q_k(\mathbf{z}'|\mathbf{z})\right) \\ &= p(\mathbf{z}')q_k(\mathbf{z}|\mathbf{z}')A_k(\mathbf{z}, \mathbf{z}') \end{aligned} \tag{14.41}$$

as required.

The specific choice of proposal distribution can have a marked effect on the performance of the algorithm. For continuous state spaces, a common choice is a Gaussian centred on the current state, leading to an important trade-off in determining the variance parameter of this distribution. If the variance is small, then the proportion of accepted transitions will be high, but progress through the state space takes the form of a slow random walk leading to long correlation times. However, if the variance parameter is large, then the rejection rate will be high because, in the kind of complex problems we are considering, many of the proposed steps will be to states for which the probability $p(\mathbf{z})$ is low. Consider a multivariate distribution $p(\mathbf{z})$ having strong correlations between the components of $\mathbf{z}$, as illustrated in Figure 14.10. The scale ρ of the proposal distribution should be as large as possible without incurring high rejection rates. This suggests that ρ should be of the same order as the smallest length scale $\sigma_{\min}$. The system then explores the distribution along the more extended direction by means of a random walk, and so the number of steps to arrive at a state that is more or less independent of the original state is of order $(\sigma_{\max}/\sigma_{\min})^2$. In fact in two dimensions, the increase in rejection rate as ρ increases is offset by the larger step sizes of those transitions that are accepted, and more generally for a multivariate Gaussian, the number of steps required to obtain independent samples scales like

Telegram: @uni_k

www.konkur.in

Algorithm 14.2: Metropolis-Hastings sampling

Input: Unnormalized distribution $\widetilde{p}(\mathbf{z})$
Proposal distributions $\{q_k(\mathbf{z}|\widehat{\mathbf{z}}) : k \in 1, \ldots, K\}$
Mapping from iteration index to distribution index $M(\cdot)$
Initial state $\mathbf{z}^{(0)}$
Number of iterations T

Output: $\mathbf{z} \sim \widetilde{p}(\mathbf{z})$

$\mathbf{z}_{\text{prev}} \leftarrow \mathbf{z}^{(0)}$

// Iterative message-passing

for $\tau \in \{1, \ldots, T\}$ **do**

  $k \leftarrow M(\tau)$ // get distribution index for this iteration

  $\mathbf{z}^\star \sim q_k(\mathbf{z}|\mathbf{z}_{\text{prev}})$ // sample from proposal distribution

  $u \sim \mathcal{U}(0, 1)$ // sample from uniform

  if $\widetilde{p}(\mathbf{z}^\star)q(\mathbf{z}_{\text{prev}}|\mathbf{z}^\star) \;/\; \widetilde{p}(\mathbf{z}_{\text{prev}})q(\mathbf{z}^\star|\mathbf{z}_{\text{prev}}) > u$ **then**

    $\mathbf{z}_{\text{prev}} \leftarrow \mathbf{z}^\star$ // $\mathbf{z}^{(\tau)} = \mathbf{z}^\star$

  else

    $\mathbf{z}_{\text{prev}} \leftarrow \mathbf{z}_{\text{prev}}$ // $\mathbf{z}^{(\tau)} = \mathbf{z}^{(\tau-1)}$

  end if

end for

return $\mathbf{z}_{\text{prev}}$ // $\mathbf{z}^{(T)}$

$(\sigma_{\max}/\sigma_2)^2$ where σ_2 is the second-smallest standard deviation (Neal, 1993). These details aside, if the length scales over which the distributions vary are very different in different directions, then the Metropolis Hastings algorithm can have very slow convergence.

14.2.4 Gibbs sampling

Gibbs sampling (Geman and Geman, 1984) is a simple and widely applicable Markov chain Monte Carlo algorithm and can be seen as a special case of the Metropolis–Hastings algorithm. Consider the distribution $p(\mathbf{z}) = p(z_1, \ldots, z_M)$ from which we wish to sample, and suppose that we have chosen some initial state for the Markov chain. Each step of the Gibbs sampling procedure involves replacing the value of one of the variables by a value drawn from the distribution of that variable conditioned on the values of the remaining variables. Thus, we replace z_i by a value drawn from the distribution $p(z_i|\mathbf{z}_{\setminus i})$, where z_i denotes the ith component of $\mathbf{z}$, and $\mathbf{z}_{\setminus i}$ denotes $\{z_1, \ldots, z_M\}$ but with z_i omitted. This procedure is repeated either by cycling through the variables in some particular order or by choosing the

Telegram: @uni_k

www.konkur.in

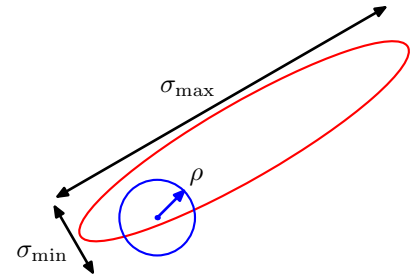


Figure 14.10 Schematic illustration of using an isotropic Gaussian proposal distribution (blue circle) to sample from a correlated multivariate Gaussian distribution (red ellipse) having very different standard deviations in different directions, using the Metropolis–Hastings algorithm. To keep the rejection rate low, the scale ρ of the proposal distribution should be of the order of the smallest standard deviation $\sigma_{\min}$, which leads to random walk behaviour in which the number of steps separating states that are approximately independent is of order $(\sigma_{\max}/\sigma_{\min})^2$ where $\sigma_{\max}$ is the largest standard deviation.

variable to be updated at each step at random from some distribution.

For example, suppose we have a distribution $p(z_1, z_2, z_3)$ over three variables, and at step τ of the algorithm we have selected values $z_1^{(\tau)}$, $z_2^{(\tau)}$, and $z_3^{(\tau)}$. We first replace $z_1^{(\tau)}$ by a new value $z_1^{(\tau+1)}$ obtained by sampling from the conditional distribution

$$p(z_1|z_2^{(\tau)}, z_3^{(\tau)}). \tag{14.42}$$

Next we replace $z_2^{(\tau)}$ by a value $z_2^{(\tau+1)}$ obtained by sampling from the conditional distribution

$$p(z_2|z_1^{(\tau+1)}, z_3^{(\tau)}) \tag{14.43}$$

so that the new value for z_1 is used straight away in subsequent sampling steps. Then we update z_3 with a sample $z_3^{(\tau+1)}$ drawn from

$$p(z_3|z_1^{(\tau+1)}, z_2^{(\tau+1)}) \tag{14.44}$$

and so on, cycling through the three variables in turn. Gibbs sampling is summarized in Algorithm 14.3.

To show that this procedure samples from the required distribution, we first note that the distribution $p(\mathbf{z})$ is an invariant of each of the Gibbs sampling steps individually and hence of the whole Markov chain. This follows since when we sample from $p(z_i|\mathbf{z}_{\setminus i})$, the marginal distribution $p(\mathbf{z}_{\setminus i})$ is clearly invariant because the value of $\mathbf{z}_{\setminus i}$ is unchanged. Also, each step by definition samples from the correct conditional distribution $p(z_i|\mathbf{z}_{\setminus i})$. Because these conditional and marginal distributions together specify the joint distribution, we see that the joint distribution is itself invariant.

The second requirement to be satisfied to ensure that the Gibbs sampling procedure samples from the correct distribution is that it is ergodic. A sufficient condition for ergodicity is that none of the conditional distributions are anywhere zero. If this is the case, then any point in z-space can be reached from any other point in a finite number of steps involving one update of each of the component variables. If this requirement is not satisfied, so that some of the conditional distributions have zeros, then ergodicity, if it applies, must be proven explicitly.

Telegram: @uni_k

www.konkur.in

Algorithm 14.3: Gibbs sampling

Input: Initial values $\{z_i : i \in 1, \ldots, M\}$
Conditional distributions $\{p(z_i|\{z_{j\neq i}\}) : i \in 1, \ldots, M\}$
Number of iterations T

Output: Final values $\{z_i : i \in 1, \ldots, M\}$

---

for $\tau \in \{1, \ldots, T\}$ **do**
  for $i \in \{1, \ldots, M\}$ **do**
    $z_i \sim p(z_i|\{z_{j\neq i}\})$
  end for
end for
return $\{z_i : i \in 1, \ldots, M\}$

The distribution of initial states must also be specified to complete the algorithm, although samples drawn after many iterations will effectively become independent of this distribution. Of course, successive samples from the Markov chain will be highly correlated, and so to obtain samples that are nearly independent it will be necessary to sub-sample the sequence.

We can obtain the Gibbs sampling procedure as a particular instance of the Metropolis–Hastings algorithm as follows. Consider a Metropolis–Hastings sampling step involving the variable z_k in which the remaining variables $\mathbf{z}_{\backslash k}$ remain fixed, and for which the transition probability from $\mathbf{z}$ to $\mathbf{z}^\star$ is given by $q_k(\mathbf{z}^\star|\mathbf{z}) = p(z_k^\star|\mathbf{z}_{\backslash k})$. Note that $\mathbf{z}_{\backslash k}^\star = \mathbf{z}_{\backslash k}$ because these components are unchanged by the sampling step. Also, $p(\mathbf{z}) = p(z_k|\mathbf{z}_{\backslash k})p(\mathbf{z}_{\backslash k})$. Thus, the factor that determines the acceptance probability in the Metropolis–Hastings (14.40) is given by

$$A(\mathbf{z}^\star, \mathbf{z}) = \frac{p(\mathbf{z}^\star)q_k(\mathbf{z}|\mathbf{z}^\star)}{p(\mathbf{z})q_k(\mathbf{z}^\star|\mathbf{z})} = \frac{p(z_k^\star|\mathbf{z}_{\backslash k}^\star)p(\mathbf{z}_{\backslash k}^\star)p(z_k|\mathbf{z}_{\backslash k}^\star)}{p(z_k|\mathbf{z}_{\backslash k})p(\mathbf{z}_{\backslash k})p(z_k^\star|\mathbf{z}_{\backslash k})} = 1 \tag{14.45}$$

where we have used $\mathbf{z}_{\backslash k}^\star = \mathbf{z}_{\backslash k}$. Thus, the Metropolis–Hastings steps are always accepted.

As with the Metropolis algorithm, we can gain some insight into the behaviour of Gibbs sampling by investigating its application to a Gaussian distribution. Consider a correlated Gaussian in two variables, as illustrated in Figure 14.11, having conditional distributions of width l and marginal distributions of width L. The typical step size is governed by the conditional distributions and will be of order l. Because the state evolves according to a random walk, the number of steps needed to obtain independent samples from the distribution will be of order $(L/l)^2$. Of course if the Gaussian distribution were uncorrelated, then the Gibbs sampling procedure would be optimally efficient. For this simple problem, we could rotate the coordinate system such that the new variables are uncorrelated. However, in practical applications

Telegram: @uni_k

www.konkur.in

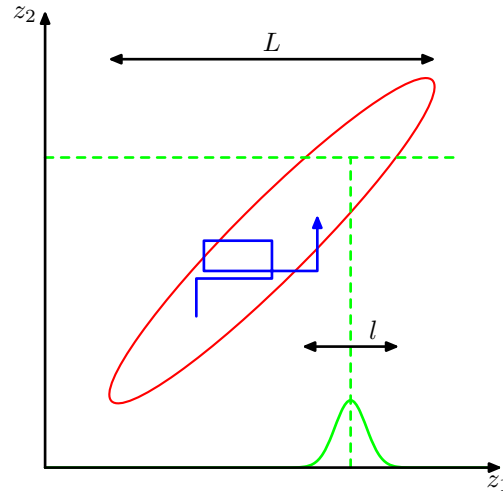


Figure 14.11 Illustration of Gibbs sampling by alternate updates of two variables whose distribution is a correlated Gaussian. The step size is governed by the standard deviation of the conditional distribution (green curve), and is $\mathcal{O}(l)$, leading to slow progress in the direction of elongation of the joint distribution (red ellipse). The number of steps needed to obtain an independent sample from the distribution is $\mathcal{O}((L/l)^2)$.

it will generally be infeasible to find such transformations.

One approach to reducing the random walk behaviour in Gibbs sampling is called *over-relaxation* (Adler, 1981). In its original form, it applies to problems for which the conditional distributions are Gaussian, which represents a more general class of distributions than the multivariate Gaussian because, for example, the non-Gaussian distribution $p(z, y) \propto \exp(-z^2y^2)$ has Gaussian conditional distributions. At each step of the Gibbs sampling algorithm, the conditional distribution for a particular component z_i has some mean μ_i and some variance σ_i^2. In the over-relaxation framework, the value of z_i is replaced with

$$z_i' = \mu_i + \alpha_i(z_i - \mu_i) + \sigma_i(1 - \alpha_i^2)^{1/2}\nu \tag{14.46}$$

where ν is a Gaussian random variable with zero mean and unit variance, and α is a parameter such that $-1 < \alpha < 1$. For $\alpha = 0$, the method is equivalent to standard Gibbs sampling, and for $\alpha < 0$ the step is biased to the opposite side of the mean. This step leaves the desired distribution invariant because if z_i has mean μ_i and variance σ_i^2, then so too does z_i'. The effect of over-relaxation is to encourage directed motion through state space when the variables are highly correlated. The framework of *ordered over-relaxation* (Neal, 1999) generalizes this approach to non-Gaussian distributions.

Exercise 14.14

The practical applicability of Gibbs sampling depends on the ease with which samples can be drawn from the conditional distributions $p(z_k|\mathbf{z}_{\setminus k})$. For probability distributions specified using directed graphical models, the conditional distributions for individual nodes depend only on the variables in the corresponding Markov blanket, as illustrated in Figure 14.12. For directed graphs, a wide choice of conditional distributions for the individual nodes conditioned on their parents will lead to conditional distributions for Gibbs sampling that are log concave. The adaptive rejection sampling methods discussed in Section 14.1.4 therefore provide a framework for Monte Carlo sampling from directed graphs with broad applicability.

Telegram: @uni_k

www.konkur.in

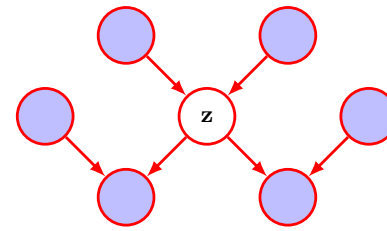


Figure 14.12 The Gibbs sampling method requires samples to be drawn from the conditional distribution of a variable $\mathbf{z}$ conditioned on the remaining variables. For directed graphical models, this conditional distribution is a function of only the states of the nodes in the Markov blanket, shaded in blue, which comprises the parents, the children, and the co-parents.

Because the basic Gibbs sampling technique considers one variable at a time, there are strong dependencies between successive samples. At the opposite extreme, if we could draw samples directly from the joint distribution (an operation that we are supposing is intractable), then successive samples would be independent. We can hope to improve on the simple Gibbs sampler by adopting an intermediate strategy in which we sample successively from groups of variables rather than individual variables. This is achieved in the *blocking Gibbs* sampling algorithm by choosing blocks of variables, not necessarily disjoint, and then sampling jointly from the variables in each block in turn, conditioned on the remaining variables (Jensen, Kong, and Kjaerulff, 1995).

14.2.5 Ancestral sampling

For many models, the joint distribution $p(\mathbf{z})$ is conveniently specified in terms of a graphical model. For a directed graph with no observed variables, it is straightforward to sample from the joint distribution using the following *ancestral sampling* approach. The joint distribution is specified by

$$p(\mathbf{z}) = \prod_{i=1}^{M} p(\mathbf{z}_i|\mathrm{pa}(i)) \tag{14.47}$$

where $\mathbf{z}_i$ are the set of variables associated with node i, and $\mathrm{pa}(i)$ denotes the set of variables associated with the parents of node i. To obtain a sample from the joint distribution, we make one pass through the set of variables in the order $\mathbf{z}_1, \ldots, \mathbf{z}_M$ sampling from the conditional distributions $p(\mathbf{z}_i|\mathrm{pa}(i))$. This is always possible because at each step, all the parent values will have been instantiated. After one pass through the graph, we will have obtained a sample from the joint distribution. This assumes that it is possible to sample from the individual conditional distributions at each node.

Now consider a directed graph in which some of the nodes, which comprise the *evidence set* $\mathcal{E}$, are instantiated with observed values. We can in principle extend the above procedure, at least for nodes representing discrete variables, to give the following *logic sampling* approach (Henrion, 1988), which can be seen as a special case of *importance sampling*. At each step, when a sampled value is obtained for a variable $\mathbf{z}_i$ whose value is observed, the sampled value is compared to the observed value, and if they agree then the sample value is retained and the algorithm proceeds to the next variable in turn. However, if the sampled value and the observed value disagree, then the whole sample so far is discarded and the algorithm starts again

Section 14.1.5

Telegram: @uni_k

www.konkur.in

with the first node in the graph. This algorithm samples correctly from the posterior distribution because it corresponds simply to drawing samples from the joint distribution of hidden variables and data variables and then discarding those samples that disagree with the observed data (with the slight saving of not continuing with the sampling from the joint distribution as soon as one contradictory value is observed). However, the overall probability of accepting a sample from the posterior decreases rapidly as the number of observed variables increases and as the number of states that those variables can take increases, and so this approach is rarely used in practice.

An improvement on this approach is called *likelihood weighted sampling* (Fung and Chang, 1990; Shachter and Peot, 1990). It is based on ancestral sampling combined with importance sampling. For each variable in turn, if that variable is in the evidence set, then it is just set to its instantiated value. If it is not in the evidence set, then it is sampled from the conditional distribution $p(\mathbf{z}_i|\mathrm{pa}(i))$ in which the conditioning variables are set to their currently sampled values. The weighting associated with the resulting sample $\mathbf{z}$ is then given by

Exercise 14.15

$$r(\mathbf{z}) = \prod_{\mathbf{z}_i \notin \mathbf{e}} \frac{p(\mathbf{z}_i|\mathrm{pa}(i))}{p(\mathbf{z}_i|\mathrm{pa}(i))} \prod_{\mathbf{z}_i \in \mathbf{e}} \frac{p(\mathbf{z}_i|\mathrm{pa}(i))}{1} = \prod_{\mathbf{z}_i \in \mathbf{e}} p(\mathbf{z}_i|\mathrm{pa}(i)). \tag{14.48}$$

This method can be further extended using *self-importance sampling* (Shachter and Peot, 1990) in which the importance sampling distribution is continually updated to reflect the current estimated posterior distribution.

14.3. Langevin Sampling

The Metropolis–Hastings algorithm draws samples from a probability distribution by creating a Markov chain of candidate samples using a proposal distribution and then accepting or rejecting them using the criterion (14.40). This can be relatively inefficient since the proposal distribution is often a simple, fixed distribution that can generate updates in any direction in the data space, leading to a random walk.

We have seen that when training neural networks, it is hugely advantageous to make use of the gradient of the log likelihood with respect to the learnable parameters of the model in order to maximize the likelihood function. By analogy, we can introduce Markov chain sampling algorithms that make use of the gradient of the probability density with respect to the data vector so as to take steps that preferentially move towards regions of higher probability. One such technique is called *Hamiltonian Monte Carlo*, also known as *hybrid Monte Carlo*. This again makes use of a Metropolis acceptance test (Duane *et al.*, 1987; Bishop, 2006). Here we will focus on a different approach that is widely used in deep learning, called *Langevin sampling*. Although it avoids the use of an acceptance test, the algorithm has to be designed carefully to ensure that the resulting samples are unbiased. An important application of Langevin sampling arises in the context of machine learning models defined in terms of energy functions.

Telegram: @uni_k

www.konkur.in

14.3.1 Energy-based models

Many generative models can be expressed as conditional probability distributions $p(\mathbf{x}|\mathbf{w})$ where $\mathbf{x}$ is the data vector and $\mathbf{w}$ represents a vector of learnable parameters. Such models can be trained by maximizing the corresponding likelihood function defined with respect to a training data set. However, to represent a valid probability distribution, the model must satisfy

$$\int p(\mathbf{x}|\mathbf{w})p(\mathbf{x})\,\mathrm{d}\mathbf{x} = 1. \tag{14.49}$$

Ensuring that this requirement is met can significantly limit the allowable forms for the model. If we put aside the normalization constraint then we can consider a much broader class of models called *energy-based models* (LeCun *et al.*, 2006). Suppose we have a function $E(\mathbf{x}, \mathbf{w})$, called the *energy function*, which is a real-valued function of its arguments but which has no other constraints. The exponential $\exp\{-E(\mathbf{x}, \mathbf{w})\}$ is a non-negative quantity and can therefore be viewed as an unnormalized probability distribution over $\mathbf{x}$. Here the introduction of the minus sign in the exponent is simply a convention, and it means that higher values of energy correspond to lower values of probability. We can then define a normalized distribution using

$$p(\mathbf{x}|\mathbf{w}) = \frac{1}{Z(\mathbf{w})} \exp\{-E(\mathbf{x}, \mathbf{w})\} \tag{14.50}$$

Exercise 14.16 where the normalizing constant $Z(\mathbf{w})$, known as the *partition function*, is defined by

$$Z(\mathbf{w}) = \int \exp\{-E(\mathbf{x}, \mathbf{w})\}\,\mathrm{d}\mathbf{x}. \tag{14.51}$$

The energy function is often modelled using a deep neural network with input vector $\mathbf{x}$ and a scalar output $E(\mathbf{x}, \mathbf{w})$, where $\mathbf{w}$ represents the weights and biases in the network.

Note that the partition function depends on $\mathbf{w}$, which creates problems for training. For example, the log likelihood function for a data set $\mathcal{D} = (\mathbf{x}_1, \ldots, \mathbf{x}_N)$ of *Exercise 14.17* i.i.d. data has the form

$$\ln p(\mathcal{D}|\mathbf{w}) = -\sum_{n=1}^{N} E(\mathbf{x}_n, \mathbf{w}) - N \ln Z(\mathbf{w}). \tag{14.52}$$

To compute the gradient of $\ln p(\mathcal{D}|\mathbf{w})$ with respect to $\mathbf{w}$, we need to know the form of $Z(\mathbf{w})$. However, for many choices of the energy function $E(\mathbf{x}, \mathbf{w})$, it will be impractical to evaluate the partition function in (14.51) because this involves integrating (or summing for discrete variables) over all the whole of $\mathbf{x}$-space. The term 'energy-based model' is generally used for models where this integral is intractable. Note, however, that probabilistic models can be seen as special cases of energy-based models, and therefore many of the models discussed in this book can be viewed as energy-based models. The big advantage of energy-based models, therefore, is their flexibility in that they bypass the requirement for normalization. A corresponding disadvantage, however, is that since the normalizing constant is unknown, they can be more difficult to train.

Telegram: @uni_k

www.konkur.in

14.3.2 Maximizing the likelihood

Various approximation methods have been developed to train energy-based models without having to evaluate the partition function (Song and Kingma, 2021). Here we look at techniques based on Markov chain Monte Carlo. An alternative approach, called score matching, will be discussed in the context of diffusion models.

Chapter 20

We have seen that for energy-based models, the likelihood function cannot be evaluated explicitly due to the unknown partition function $Z(\mathbf{w})$. However, we can make use of Monte Carlo sampling methods to approximate the gradient of the log likelihood with respect to the model parameters. Once an energy-based model has been trained, by whatever means, we also need a way to draw samples from the model, and again we can make use of Monte Carlo methods.

Using (14.50), the gradient, with respect to the model parameters, of the log likelihood function for an energy-based model can be written in the form

$$\nabla_{\mathbf{w}} \ln p(\mathbf{x}|\mathbf{w}) = -\nabla_{\mathbf{w}} E(\mathbf{x}, \mathbf{w}) - \nabla_{\mathbf{w}} \ln Z(\mathbf{w}). \tag{14.53}$$

This is the likelihood function for a single data point $\mathbf{x}$, but in practice we want to maximize the likelihood defined over a training set of data points drawn from some unknown distribution $p_{\mathcal{D}}(\mathbf{x})$. If we assume the data points are i.i.d., then we can consider the gradient of the expectation of the log likelihood with respect to $p_{\mathcal{D}}(\mathbf{x})$, which is then given by

$$\mathbb{E}_{\mathbf{x}\sim p_{\mathcal{D}}} \left[\nabla_{\mathbf{w}} \ln p(\mathbf{x}|\mathbf{w})\right] = -\mathbb{E}_{\mathbf{x}\sim p_{\mathcal{D}}} \left[\nabla_{\mathbf{w}} E(\mathbf{x}, \mathbf{w})\right] - \nabla_{\mathbf{w}} \ln Z(\mathbf{w}) \tag{14.54}$$

where we have made use of the fact that the final term $-\nabla_{\mathbf{w}} \ln Z(\mathbf{w})$ does not depend on $\mathbf{x}$ and can therefore be taken outside the expectation. The partition function $Z(\mathbf{w})$ is assumed to be unknown, but we can make use of (14.51) and rearrange to obtain

Exercise 14.18

$$-\nabla_{\mathbf{w}} \ln Z(\mathbf{w}) = \int \left\{\nabla_{\mathbf{w}} E(\mathbf{x}, \mathbf{w})\right\} p(\mathbf{x}|\mathbf{w}) \, \mathrm{d}\mathbf{x}. \tag{14.55}$$

The right-hand side of (14.55) corresponds to an expectation over the model distribution $p(\mathbf{x}|\mathbf{w})$ given by

$$\int \left\{\nabla_{\mathbf{w}} E(\mathbf{x}, \mathbf{w})\right\} p(\mathbf{x}|\mathbf{w}) \, \mathrm{d}\mathbf{x} = \mathbb{E}_{\mathbf{x}\sim\mathcal{M}} \left[\nabla_{\mathbf{w}} E(\mathbf{x}, \mathbf{w})\right]. \tag{14.56}$$

Exercise 14.18

Combining (14.54), (14.55), and (14.56) we obtain

$$\nabla_{\mathbf{w}} \mathbb{E}_{\mathbf{x}\sim p_{\mathcal{D}}} \left[\ln p(\mathbf{x}|\mathbf{w})\right] = -\mathbb{E}_{\mathbf{x}\sim p_{\mathcal{D}}} \left[\nabla_{\mathbf{w}} E(\mathbf{x}, \mathbf{w})\right] + \mathbb{E}_{\mathbf{x}\sim p_{\mathcal{M}}(\mathbf{x})} \left[\nabla_{\mathbf{w}} E(\mathbf{x}, \mathbf{w})\right]. \tag{14.57}$$

This result is illustrated in Figure 14.13, and has a nice interpretation, as follows. Our goal is to find values for the parameters $\mathbf{w}$ that maximize the likelihood function, and therefore consider a small change to $\mathbf{w}$ in the direction of the gradient $\nabla_{\mathbf{w}} \ln p(\mathbf{x}|\mathbf{w})$. From (14.57) we see that expected value of this gradient can be expressed as two terms, having opposite signs. The first term on the right-hand side of (14.57) acts

Telegram: @uni_k

www.konkur.in

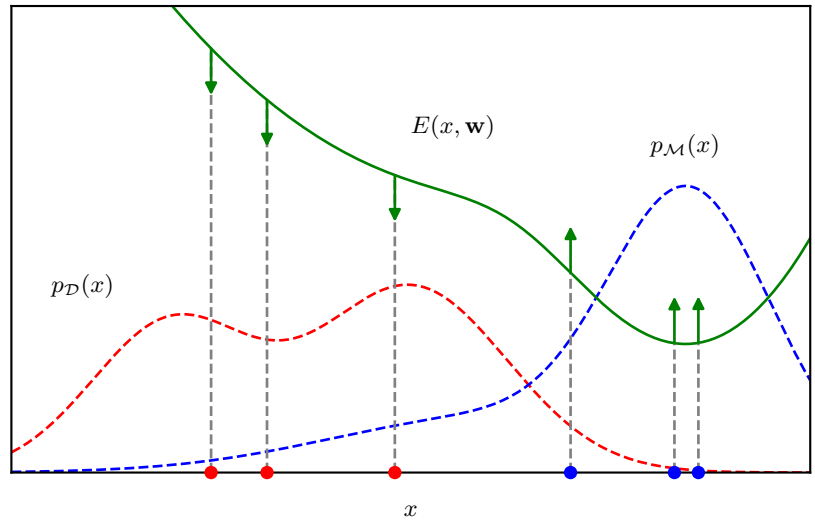


Figure 14.13 Illustration of the training of an energy-based model by maximizing the likelihood, showing the energy function $E(x, \mathbf{w})$ in green along with the associated model distribution $p_{\mathcal{M}}(x)$ and the true data distribution $p_{\mathcal{D}}(x)$. Increasing the expected log likelihood by using (14.57) pushes the energy function up at points corresponding to samples from the model (shown as blue dots) and pushes it down at points corresponding to samples from the data set (shown as red dots).

to decrease $E(\mathbf{x}, \mathbf{w})$, and therefore to increase the probability density defined by the model, for points $\mathbf{x}$ drawn from $p_{\mathcal{D}}(\mathbf{x})$. The second term on the right-hand side of (14.57) acts to increase the value of $E(\mathbf{x}, \mathbf{w})$, and therefore to decrease the probability density defined by the model, for data points drawn from the model itself. In regions where the model density exceeds the training data density, the net effect will be to increase the energy and therefore reduce the probability. Conversely, in regions where training data density exceeds the model density, the net effect will be to reduce the energy and therefore increase the probability density. Together these two terms move probability mass away from regions where there is a low density of training data and towards regions of high data density, as desired. The two terms will be equal in magnitude when the model distribution matches the data distribution, at which point the gradient on the left-hand-side of (14.57) will equal zero.

14.3.3 Langevin dynamics

When applying (14.57) as a practical training method, we need to approximate the two terms on the right-hand side. For any given value of $\mathbf{x}$, we can evaluate $\nabla_{\mathbf{w}} E(\mathbf{x}, \mathbf{w})$ using automatic differentiation. For the first term in (14.57), we can use the training data set to estimate the expectation over $\mathbf{x}$:

$$\mathbb{E}_{\mathbf{x} \sim p_{\mathcal{D}}} \left[\nabla_{\mathbf{w}} E(\mathbf{x}, \mathbf{w}) \right] \simeq \frac{1}{N} \sum_{n=1}^{N} \nabla_{\mathbf{w}} E(\mathbf{x}_n, \mathbf{w}). \tag{14.58}$$

Telegram: @uni_k

www.konkur.in

The second term is more challenging because we need to draw samples from the model distribution defined by an energy function whose corresponding partition function is intractable. This can be done using Markov chain Monte Carlo methods. One popular approach is called *stochastic gradient Langevin dynamics* or simply *Langevin sampling* (Parisi, 1981; Welling and Teh, 2011). This term depends on the distribution $p(\mathbf{x}|\mathbf{w})$ only through the *score function*, which is defined to be the gradient of the log likelihood with respect to the data vector $\mathbf{x}$, and is given by

$$\mathbf{s}(\mathbf{x}, \mathbf{w}) = \nabla_{\mathbf{x}} \ln p(\mathbf{x}|\mathbf{w}). \tag{14.59}$$

It is worth emphasising that this gradient is taken with respect to the data point $\mathbf{x}$ and is therefore not the usual gradient with respect to the learnable parameters $\mathbf{w}$. If we substitute (14.50) into (14.59) we obtain

$$\mathbf{s}(\mathbf{x}, \mathbf{w}) = -\nabla_{\mathbf{x}} E(\mathbf{x}, \mathbf{w}) \tag{14.60}$$

where we see that the partition function no longer appears at it is independent of $\mathbf{x}$.

We start by drawing an initial value $\mathbf{x}^{(0)}$ from a prior distribution, and then we iterate the following Markov chain steps:

$$\mathbf{x}^{(\tau+1)} = \mathbf{x}^{(\tau)} + \eta \nabla_{\mathbf{x}} \ln p(\mathbf{x}^{(\tau)}, \mathbf{w}) + \sqrt{2\eta}\boldsymbol{\epsilon}^{(\tau)}, \qquad \tau \in 1, \ldots, \mathcal{T} \tag{14.61}$$

where $\boldsymbol{\epsilon}^{(\tau)} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ are independent samples from a zero-mean, unit-covariance Gaussian distribution, and the parameter η controls the step size. Each iteration of the Langevin equation takes a step in the direction of the gradient of the log likelihood, and then adds Gaussian noise. It can be show that, in the limits of $\eta \to 0$ and $\mathcal{T} \to \infty$, the value of $\mathbf{z}^{(\mathcal{T})}$ is an independent sample from the distribution $p(\mathbf{x})$. Langevin sampling is summarized in Algorithm 14.4.

We can repeat the process to generate a set of samples $\{\mathbf{x}_1, \ldots, \mathbf{x}_M\}$ and then approximate the second term in (14.57) using

$$\mathbb{E}_{\mathbf{x} \sim p_{\mathcal{M}}(\mathbf{x})} \left[\nabla_{\mathbf{w}} E(\mathbf{x}, \mathbf{w})\right] \simeq \frac{1}{M} \sum_{m=1}^{M} \nabla_{\mathbf{w}} E(\mathbf{x}_m, \mathbf{w}). \tag{14.62}$$

Running long Markov chains to generate independent samples can be computationally expensive, and so we need to consider practical approximations. One approach is called *contrastive divergence* (Hinton, 2002). Here the samples used to evaluate (14.62) are obtained by running a Monte Carlo chain starting with one of the training data points $\mathbf{x}_n$. If the chain is run for a large number of steps, then the resulting value will be essentially an unbiased sample from the model distribution. Instead Hinton (2002) proposes running for only a few steps of Monte Carlo, perhaps even as few as one step, which is computationally much less costly. The resulting sample will be far from unbiased and will lie close to the data manifold. As a result, the effect of using gradient descent will be to shape the energy surface, and hence the probability density, only in the neighbourhood of the data manifold. This can prove effective for tasks such as discrimination but is expected to be less effective in learning a generative model.

Telegram: @uni_k

www.konkur.in

Algorithm 14.4: Langevin sampling

Input: Initial value $\mathbf{x}^{(0)}$
Probability density $p(\mathbf{x}, \mathbf{w})$
Learning rate parameter η
Number of iterations T

Output: Final value $\mathbf{x}^{(T)}$

$\mathbf{x} \leftarrow \mathbf{x}_0$
for $\tau \in \{1, \ldots, T\}$ **do**
  $\boldsymbol{\epsilon} \sim \mathcal{N}(\boldsymbol{\epsilon}|\mathbf{0}, \mathbf{I})$
  $\mathbf{x} \leftarrow \mathbf{x} + \eta \nabla_{\mathbf{x}} \ln p(\mathbf{x}, \mathbf{w}) + \sqrt{2\eta}\boldsymbol{\epsilon}$
end for
return $\mathbf{x}$ `// Final value` $\mathbf{x}^{(T)}$

Exercises

14.1 ($\star$) Show that $\overline{f}$ defined by (14.2) is an unbiased estimator, in other words that the expectation of the right-hand side is equal to $\mathbb{E}[f(\mathbf{z})]$.

14.2 ($\star$) Show that $\overline{f}$ defined by (14.2) has variance given by (14.4).

14.3 ($\star$) Suppose that z is a random variable with uniform distribution over $(0, 1)$ and that we transform z using $y = h^{-1}(z)$ where $h(y)$ is given by (14.6). Show that y has the distribution $p(y)$.

14.4 ($\star\star$) Given a random variable z that is uniformly distributed over $(0, 1)$, find a transformation $y = f(z)$ such that y has a Cauchy distribution given by (14.8).

14.5 ($\star\star$) Suppose that z_1 and z_2 are uniformly distributed over the unit circle, as shown in Figure 14.3, and that we make the change of variables given by (14.10) and (14.11). Show that (y_1, y_2) will be distributed according to (14.12).

14.6 ($\star\star$) Let $\mathbf{z}$ be a D-dimensional random variable having a Gaussian distribution with zero mean and unit covariance matrix, and suppose that the positive definite symmetric matrix $\boldsymbol{\Sigma}$ has the Cholesky decomposition $\boldsymbol{\Sigma} = \mathbf{L}\mathbf{L}^{\mathrm{T}}$, where $\mathbf{L}$ is a lower-triangular matrix (i.e., one with zeros above the leading diagonal). Show that the variable $\mathbf{y} = \boldsymbol{\mu} + \mathbf{L}\mathbf{z}$ has a Gaussian distribution with mean $\boldsymbol{\mu}$ and covariance $\boldsymbol{\Sigma}$. This provides a technique for generating samples from a general multivariate Gaussian using samples from a univariate Gaussian having zero mean and unit variance.

14.7 ($\star\star$) In this exercise, we show more carefully that rejection sampling does indeed draw samples from the desired distribution $p(\mathbf{z})$. Suppose the proposal distribution is $q(\mathbf{z})$. Show that the probability of a sample value $\mathbf{z}$ being accepted is given by

Telegram: @uni_k

www.konkur.in

$\widetilde{p}(\mathbf{z})/kq(\mathbf{z})$ where $\widetilde{p}$ is any unnormalized distribution that is proportional to $p(\mathbf{z})$, and the constant k is set to the smallest value that ensures $kq(\mathbf{z}) \geqslant \widetilde{p}(\mathbf{z})$ for all values of $\mathbf{z}$. Note that the probability of drawing a value $\mathbf{z}$ is given by the probability of drawing that value from $q(\mathbf{z})$ times the probability of accepting that value given that it has been drawn. Make use of this, along with the sum and product rules of probability, to write down the normalized form for the distribution over $\mathbf{z}$, and show that it equals $p(\mathbf{z})$.

14.8 ($\star$) Suppose that z has a uniform distribution over the interval $[0, 1]$. Show that the variable $y = b \tan z + c$ has a Cauchy distribution given by (14.16).

14.9 ($\star\star$) Determine expressions for the coefficients k_i in the envelope distribution (14.17) for adaptive rejection sampling using the requirements of continuity and normalization.

14.10 ($\star\star$) By making use of the technique discussed in Section 14.1.2 for sampling from a single exponential distribution, devise an algorithm for sampling from the piecewise exponential distribution defined by (14.17).

14.11 ($\star$) Show that the simple random walk over the integers defined by (14.28), (14.29), and (14.30) has the property that $\mathbb{E}[(z^{(\tau)})^2] = \mathbb{E}[(z^{(\tau-1)})^2] + 1/2$ and hence by induction that $\mathbb{E}[(z^{(\tau)})^2] = \tau/2$.

14.12 ($\star\star$) Show that the Gibbs sampling algorithm, discussed in Section 14.2.4, satisfies detailed balance as defined by (14.34).

14.13 ($\star$) Consider the distribution shown in Figure 14.14. Discuss whether the standard Gibbs sampling procedure for this distribution is ergodic and therefore whether it would sample correctly from this distribution

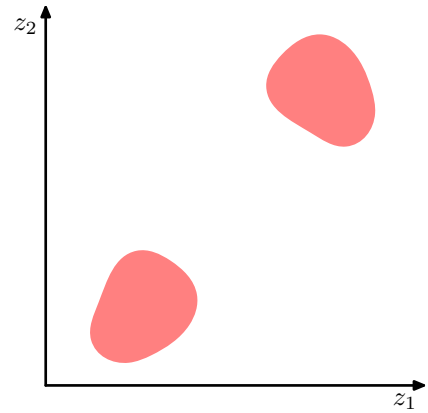


Figure 14.14 A probability distribution over two variables z_1 and z_2 that is uniform over the shaded regions and zero everywhere else.

14.14 ($\star$) Verify that the over-relaxation update (14.46), in which z_i has mean μ_i and variance σ_i and where ν has zero mean and unit variance gives a value z_i' with mean μ_i and variance σ_i^2.

Telegram: @uni_k

www.konkur.in

14.15 ($\star$) Show that in likelihood weighted sampling from a directed graph the importance sampling weights are given by (14.48).

14.16 ($\star$) Show that the distribution (14.50) is normalized with respect to $\mathbf{x}$ provided $Z(\mathbf{w})$ satisfies (14.51).

14.17 ($\star\star$) By making use of (14.50) show that the gradient of the log likelihood function for an energy-based model can be written in the form (14.52).

14.18 ($\star\star$) By making use of (14.54), (14.55), and (14.56), show that the gradient of the log likelihood function for an energy-based model can be written in the form (14.57).

Telegram: @uni_k

www.konkur.in



15 Discrete Latent Variables

Chapter 11

We have seen how complex distributions can be constructed by combining multiple simple distributions and how the resulting models can be described by directed graphs. In addition to the observed variables, which form part of the data set, such models often introduce additional hidden, or latent, variables. These might correspond to specific quantities involved in the data generation process, such as the unknown orientation of an object in three-dimensional space in the case of images, or they may be introduced simply as modelling constructs to allow much richer models to be created. If we define a joint distribution over observed and latent variables, the corresponding distribution of the observed variables alone is obtained by marginalization. This allows relatively complex marginal distributions over observed variables to be expressed in terms of more tractable joint distributions over the expanded space of observed and latent variables.

In this chapter, we will see that marginalizing over discrete latent variables gives

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024
C. M. Bishop, H. Bishop, *Deep Learning*, https://doi.org/10.1007/978-3-031-45468-4_15

Telegram: @uni_k

www.konkur.in

rise to mixture distributions. Our focus will be on mixtures of Gaussians that provide a good illustration of mixture distributions and that are also widely used in machine learning. One simple application for mixture models is to discover clusters in data, and we begin our discussion by considering a technique for clustering called the K-means algorithm, which corresponds to a particular non-probabilistic limit of Gaussian mixtures. Then we introduce the latent-variable view of mixture distributions in which the discrete latent variables can be interpreted as defining assignments of data points to specific components of the mixture.

A general technique for finding maximum likelihood estimators in latent-variable models is the expectation–maximization (EM) algorithm. We first use the Gaussian mixture distribution to motivate the EM algorithm in an informal way, and then we give a more careful treatment based on the latent-variable viewpoint. Finally we provide a general perspective by introducing the evidence lower bound (ELBO), which will play an important role in generative models such as variational autoencoders and diffusion models.

15.1. K-means Clustering

We begin by considering the problem of identifying groups, or clusters, of data points in a multi-dimensional space. Suppose we have a data set $\{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$ consisting of N observations of a D-dimensional Euclidean variable $\mathbf{x}$. Our goal is to partition the data set into some number K of clusters, where we will suppose for the moment that the value of K is given. Intuitively, we might think of a cluster as comprising a group of data points whose inter-point distances are small compared with the distances to points outside the cluster. We can formalize this notion by first introducing a set of D-dimensional vectors $\boldsymbol{\mu}_k$, where $k = 1, \ldots, K$, in which $\boldsymbol{\mu}_k$ is a 'prototype' associated with the kth cluster. As we will see shortly, we can think of the $\boldsymbol{\mu}_k$ as representing the centres of the clusters. Our goal is then to find a set of cluster vectors $\{\boldsymbol{\mu}_k\}$, along with an assignment of data points to clusters, such that the sum of the squares of the distances of each data point to its closest cluster vector $\boldsymbol{\mu}_k$ is a minimum.

It is convenient at this point to define some notation to describe the assignment of data points to clusters. For each data point $\mathbf{x}_n$, we introduce a corresponding set of binary indicator variables $r_{nk} \in \{0, 1\}$, where $k = 1, \ldots, K$. These indicators describe which of the K clusters the data point $\mathbf{x}_n$ is assigned to, so that if data point $\mathbf{x}_n$ is assigned to cluster k then $r_{nk} = 1$, and $r_{nj} = 0$ for $j \neq k$. This is an example of the 1-of-K coding scheme. We can then define an error function:

$$J = \sum_{n=1}^{N} \sum_{k=1}^{K} r_{nk} \|\mathbf{x}_n - \boldsymbol{\mu}_k\|^2, \tag{15.1}$$

which represents the sum of the squares of the distances of each data point to its assigned vector $\boldsymbol{\mu}_k$. Our goal is to find values for the $\{r_{nk}\}$ and the $\{\boldsymbol{\mu}_k\}$ so as to minimize J. We can do this through an iterative procedure in which each iteration

Telegram: @uni_k

www.konkur.in

involves two successive steps corresponding to successive optimizations with respect to the $\{r_{nk}\}$ and the $\{\boldsymbol{\mu}_k\}$. First we choose some initial values for the $\{\boldsymbol{\mu}_k\}$. Then in the first step, we minimize J with respect to the $\{r_{nk}\}$, keeping the $\{\boldsymbol{\mu}_k\}$ fixed. In the second step, we minimize J with respect to the $\{\boldsymbol{\mu}_k\}$, keeping $\{r_{nk}\}$ fixed. This two-step optimization is then repeated until convergence. We will see that these two stages of updating $\{r_{nk}\}$ and updating $\{\boldsymbol{\mu}_k\}$ correspond, respectively, to the E (expectation) and M (maximization) steps of the EM algorithm, and to emphasize this, we will use the terms E step and M step in the context of the K-means algorithm.

Section 15.3

Consider first the determination of the $\{r_{nk}\}$ with the $\{\boldsymbol{\mu}_k\}$ held fixed (the E step). Because J in (15.1) is a linear function of the $\{r_{nk}\}$, this optimization can be performed easily to give a closed-form solution. The terms involving different n are independent, and so we can optimize for each n separately by choosing r_{nk} to be 1 for whichever value of k gives the minimum value of $\|\mathbf{x}_n - \boldsymbol{\mu}_k\|^2$. In other words, we simply assign the nth data point to the closest cluster centre. More formally, this can be expressed as

$$r_{nk} = \begin{cases} 1, & \text{if } k = \arg\min_j \|\mathbf{x}_n - \boldsymbol{\mu}_j\|^2, \\ 0, & \text{otherwise.} \end{cases} \tag{15.2}$$

Now consider the optimization of the $\{\boldsymbol{\mu}_k\}$ with the $\{r_{nk}\}$ held fixed (the M step). The objective function J is a quadratic function of $\boldsymbol{\mu}_k$, and it can be minimized by setting its derivative with respect to $\boldsymbol{\mu}_k$ to zero giving

$$2\sum_{n=1}^{N} r_{nk}(\mathbf{x}_n - \boldsymbol{\mu}_k) = 0, \tag{15.3}$$

which we can easily solve for $\boldsymbol{\mu}_k$ to give

$$\boldsymbol{\mu}_k = \frac{\sum_n r_{nk}\mathbf{x}_n}{\sum_n r_{nk}}. \tag{15.4}$$

The denominator in this expression is equal to the number of points assigned to cluster k, and so this result has a simple interpretation, namely that $\boldsymbol{\mu}_k$ is equal to the mean of all the data points $\mathbf{x}_n$ assigned to cluster k. For this reason, the procedure is known as the *K-means* algorithm (Lloyd, 1982). It is summarized in Algorithm 15.1. Because the assignments $\{r_{nk}\}$ are discrete and each iteration will not lead to an increase in the error function, the K-means algorithm is guaranteed to converge in a finite number of steps.

Exercise 15.1

The two phases of reassigning data points to clusters and recomputing the cluster means are repeated in turn until there is no further change in the assignments (or until some maximum number of iterations is exceeded). However, this approach may converge to a local rather than a global minimum of J. The convergence properties of the K-means algorithm were studied by MacQueen (1967).

The K-means algorithm is illustrated in Figure 15.1 using data derived from eruptions of the Old Faithful geyser in Yellowstone National Park. The data set consists of 272 data points, each of which gives the duration of an eruption on the

Section 3.2.9

Telegram: @uni_k

www.konkur.in

Algorithm 15.1: K-means algorithm

Input: Initial prototype vectors $\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_K$
Data set $\mathbf{x}_1, \ldots, \mathbf{x}_N$
Output: Final prototype vectors $\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_K$

$\{r_{nk} \leftarrow 0\}$ `// Initially set all assignments to zero`
repeat
  $\{r_{nk}^{(\text{old})}\} \leftarrow \{r_{nk}\}$
  `// Update assignments`
  for $N \in \{1, \ldots, N\}$ **do**
    $k \leftarrow \arg\min_j \|\mathbf{x}_n - \boldsymbol{\mu}_j\|^2$
    $r_{nk} \leftarrow 1$
    $r_{nj} \leftarrow 0, \quad j \in \{1, \ldots, K\},\ j \neq k$
  end for
  `// Update prototype vectors`
  for $k \in \{1, \ldots, K\}$ **do**
    $\boldsymbol{\mu}_k \leftarrow \sum_n r_{nk}\mathbf{x}_n / \sum_n r_{nk}$
  end for
until $\{r_{nk}\} = \{r_{nk}^{(\text{old})}\}$ `// Assignments unchanged`
return $\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_K, \{r_{nk}\}$

horizontal axis and the time to the next eruption on the vertical axis. Here we have made a linear re-scaling of the data, known as *standardizing*, such that each of the variables has zero mean and unit standard deviation.

For this example, we have chosen $K = 2$ and so the assignment of each data point to the nearest cluster centre is equivalent to a classification of the data points according to which side they lie of the perpendicular bisector of the two cluster centres. A plot of the cost function J given by (15.1) for the Old Faithful example is shown in Figure 15.2. Note that we have deliberately chosen poor initial values for the cluster centres so that the algorithm takes several steps before convergence. In practice, a better initialization procedure would be to choose the cluster centres $\boldsymbol{\mu}_k$ to be equal to a random subset of K data points. Also note that the K-means algorithm is often used to initialize the parameters in a Gaussian mixture model before applying the EM algorithm.

Section 15.2.2

So far, we have considered a batch version of K-means in which the whole data set is used together to update the prototype vectors. We can also derive a sequential update in which, for each data point $\mathbf{x}_n$ in turn, we update the nearest prototype $\boldsymbol{\mu}_k$ using

Exercise 15.2

Telegram: @uni_k

www.konkur.in

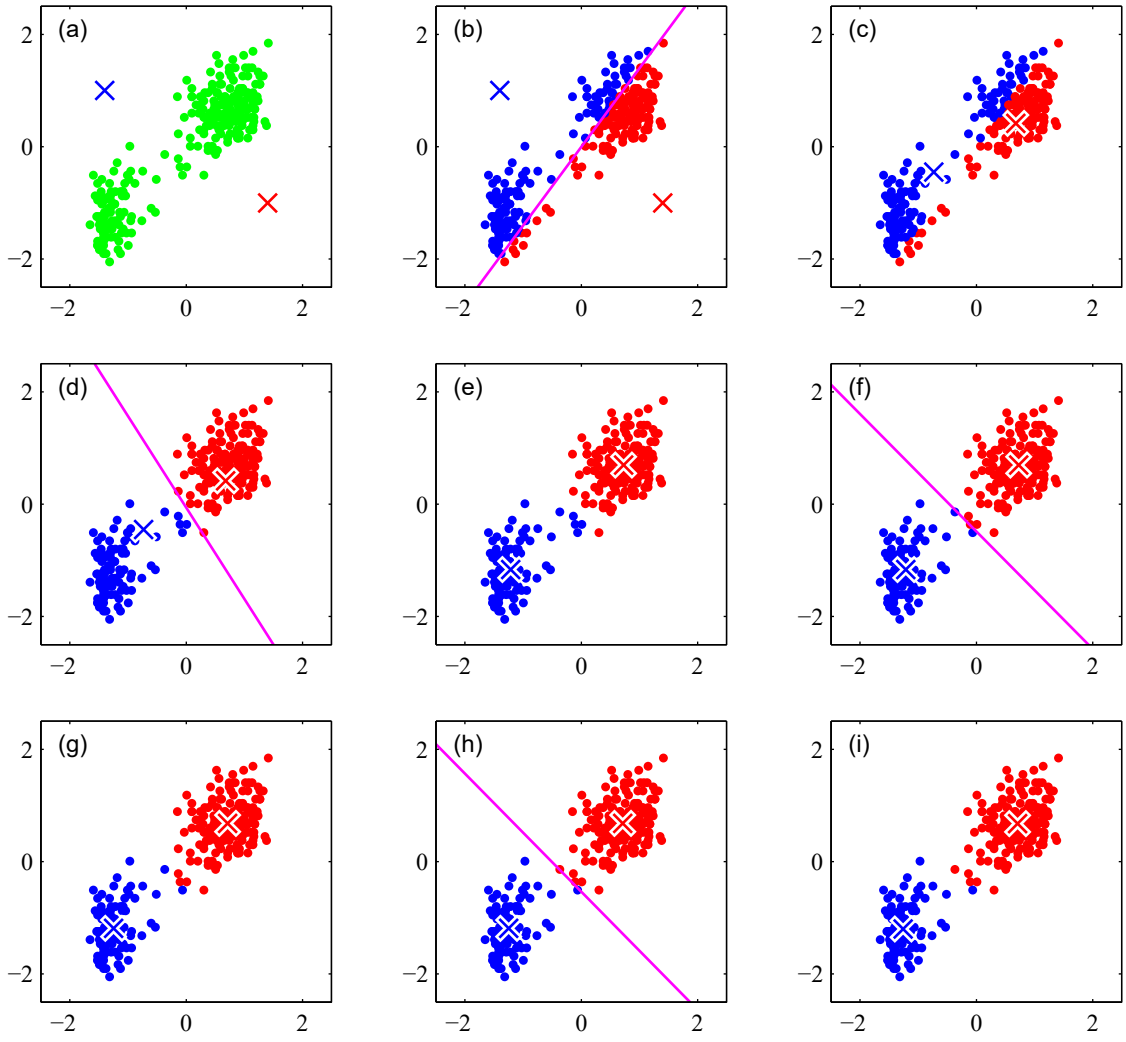


Figure 15.1 Illustration of the K-means algorithm using the re-scaled Old Faithful data set. (a) Green points denote the data set in a two-dimensional Euclidean space. The initial choices for centres $\boldsymbol{\mu}_1$ and $\boldsymbol{\mu}_2$ are shown by the red and blue crosses, respectively. (b) In the initial E step, each data point is assigned either to the red cluster or to the blue cluster, according to which cluster centre is nearer. This is equivalent to classifying the points according to which side of the perpendicular bisector of the two cluster centres, shown by the magenta line, they lie. (c) In the subsequent M step, each cluster centre is recomputed to be the mean of the points assigned to the corresponding cluster. (d)–(i) show successive E and M steps through to final convergence of the algorithm.

Telegram: @uni_k

www.konkur.in

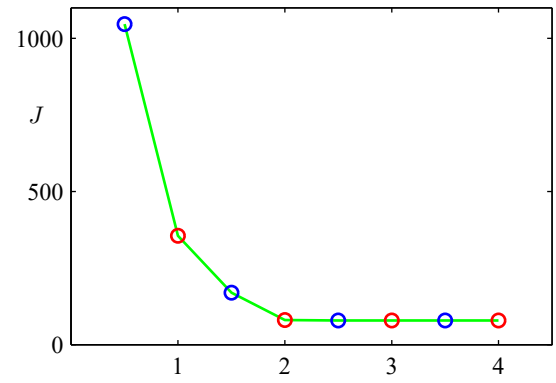


Figure 15.2 Plot of the cost function J given by (15.1) after each E step (blue points) and M step (red points) of the K-means algorithm for the example shown in Figure 15.1. The algorithm has converged after the third M step, and the final EM cycle produces no changes in either the assignments or the prototype vectors.

$$\boldsymbol{\mu}_k^{\text{new}} = \boldsymbol{\mu}_k^{\text{old}} + \frac{1}{N_k}(\mathbf{x}_n - \boldsymbol{\mu}_k^{\text{old}}) \tag{15.5}$$

where N_k is the number of data points that have so far been used to update $\boldsymbol{\mu}_k$. This allows each data point to be used once and then discarded before seeing the next data point.

One notable feature of the K-means algorithm is that at each iteration, every data point is assigned to one, and only one, of the clusters. Although some data points will be much closer to a particular centre $\boldsymbol{\mu}_k$ than to any other centre, there may be other data points that lie roughly midway between cluster centres. In the latter case, it is not clear that the hard assignment to the nearest cluster is the most appropriate. We will see that by adopting a probabilistic approach, we obtain 'soft' assignments of data points to clusters in a way that reflects the level of uncertainty over the most appropriate assignment. This probabilistic formulation has numerous benefits.

Section 15.2

15.1.1 Image segmentation

As an illustration of the application of the K-means algorithm, we consider the related problems of image segmentation and image compression. The goal of segmentation is to partition an image into regions such that each region has a reasonably homogeneous visual appearance or which corresponds to objects or parts of objects (Forsyth and Ponce, 2003). Each pixel in an image is a point in a three-dimensional space comprising the intensities of the red, blue, and green channels, and our segmentation algorithm simply treats each pixel in the image as a separate data point. Note that strictly this space is not Euclidean because the channel intensities are bounded by the interval $[0, 1]$. Nevertheless, we can apply the K-means algorithm without difficulty. We illustrate the result of running K-means to convergence, for any particular value of K, by redrawing the image in which we replace each pixel vector with the $\{R, G, B\}$ intensity triplet given by the centre $\boldsymbol{\mu}_k$ to which that pixel has been assigned. Results for various values of K are shown in Figure 15.3. We see that for a given value of K, the algorithm represents the image

Telegram: @uni_k

www.konkur.in



Figure 15.3 An example of the application of the K-means clustering algorithm to image segmentation showing an initial image together with their K-means segmentations obtained using various values of K. This also illustrates the use of vector quantization for data compression, in which smaller values of K give higher compression at the expense of poorer image quality.

using a palette of only K colours. It should be emphasized that this use of K-means is not a particularly sophisticated approach to image segmentation, not least because it takes no account of the spatial proximity of different pixels. Image segmentation is in general extremely difficult and remains the subject of active research and is introduced here simply to illustrate the behaviour of the K-means algorithm.

We can also use a clustering algorithm to perform data compression. It is important to distinguish between *lossless data compression*, in which the goal is to be able to reconstruct the original data exactly from the compressed representation, and *lossy data compression*, in which we accept some errors in the reconstruction in return for higher levels of compression than can be achieved in the lossless case. We can apply the K-means algorithm to the problem of lossy data compression as follows. For each of the N data points, we store only the identity k of the cluster to which it is assigned. We also store the values of the K cluster centres $\{\boldsymbol{\mu}_k\}$, which typically requires significantly less data, provided we choose $K \ll N$. Each data point is then approximated by its nearest centre $\boldsymbol{\mu}_k$. New data points can similarly be compressed by first finding the nearest $\boldsymbol{\mu}_k$ and then storing the label k instead of the original data vector. This framework is often called *vector quantization*, and the vectors $\{\boldsymbol{\mu}_k\}$ are called *codebook vectors*.

The image segmentation problem discussed above also provides an illustration of the use of clustering for data compression. Suppose the original image has N pixels comprising $\{R, G, B\}$ values, each of which is stored with 8 bits of precision. Directly transmitting the whole image would cost $24N$ bits. Now suppose we first run K-means on the image data, and then instead of transmitting the original pixel intensity vectors, we transmit the identity of the nearest vector $\boldsymbol{\mu}_k$. Because there are K such vectors, this requires $\log_2 K$ bits per pixel. We must also transmit the K code book vectors $\{\boldsymbol{\mu}_k\}$, which requires $24K$ bits, and so the total number of bits required to transmit the image is $24K + N \log_2 K$ (rounding up to the nearest

Telegram: @uni_k

www.konkur.in

integer). The original image shown in Figure 15.3 has $240 \times 180 = 43{,}200$ pixels and so requires $24 \times 43{,}200 = 1{,}036{,}800$ bits to transmit directly. By comparison, the compressed images require $43{,}248$ bits ($K = 2$), $86{,}472$ bits ($K = 3$), and $173{,}040$ bits ($K = 10$), respectively, to transmit. These represent compression ratios compared to the original image of 4.2%, 8.3%, and 16.7%, respectively. We see that there is a trade-off between the degree of compression and image quality. Note that our aim in this example is to illustrate the K-means algorithm. If we had been aiming to produce a good image compressor, then it would be more fruitful to consider small blocks of adjacent pixels, for instance 5×5, and thereby exploit the correlations that exist in natural images between nearby pixels.

15.2. Mixtures of Gaussians

Section 3.2.9

We have previously motivated the Gaussian mixture model as a simple linear superposition of Gaussian components, aimed at providing a richer class of density models than a single Gaussian. We now turn to a formulation of Gaussian mixtures in terms of discrete latent variables. This will provide us with a deeper insight into this important distribution and will also serve to motivate the expectation–maximization algorithm.

Recall from (3.111) that the Gaussian mixture distribution can be written as a linear superposition of Gaussians in the form

$$p(\mathbf{x}) = \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k). \tag{15.6}$$

Let us introduce a K-dimensional binary random variable $\mathbf{z}$ having a 1-of-K representation in which one of the elements is equal to 1 and all other elements are equal to 0. The values of z_k therefore satisfy $z_k \in \{0, 1\}$ and $\sum_k z_k = 1$, and we see that there are K possible states for the vector $\mathbf{z}$ according to which element is non-zero. We will define the joint distribution $p(\mathbf{x}, \mathbf{z})$ in terms of a marginal distribution $p(\mathbf{z})$ and a conditional distribution $p(\mathbf{x}|\mathbf{z})$. The marginal distribution over $\mathbf{z}$ is specified in terms of the mixing coefficients π_k, such that

$$p(z_k = 1) = \pi_k$$

where the parameters $\{\pi_k\}$ must satisfy

$$0 \leqslant \pi_k \leqslant 1 \tag{15.7}$$

together with

$$\sum_{k=1}^{K} \pi_k = 1 \tag{15.8}$$

Telegram: @uni_k

www.konkur.in



Figure 15.4 Graphical representation of a mixture model, in which the joint distribution is expressed in the form $p(\mathbf{x}, \mathbf{z}) = p(\mathbf{z})p(\mathbf{x}|\mathbf{z})$.

if they are to be valid probabilities. Because $\mathbf{z}$ uses a 1-of-K representation, we can also write this distribution in the form

$$p(\mathbf{z}) = \prod_{k=1}^{K} \pi_k^{z_k}. \tag{15.9}$$

Similarly, the conditional distribution of $\mathbf{x}$ given a particular value for $\mathbf{z}$ is a Gaussian:

$$p(\mathbf{x}|z_k = 1) = \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k),$$

which can also be written in the form

$$p(\mathbf{x}|\mathbf{z}) = \prod_{k=1}^{K} \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)^{z_k}. \tag{15.10}$$

The joint distribution is given by $p(\mathbf{z})p(\mathbf{x}|\mathbf{z})$ and is described by the graphical model in Figure 15.4. The marginal distribution of $\mathbf{x}$ is then obtained by summing the joint distribution over all possible states of $\mathbf{z}$ to give

Exercise 15.3

$$p(\mathbf{x}) = \sum_{\mathbf{z}} p(\mathbf{z})p(\mathbf{x}|\mathbf{z}) = \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \tag{15.11}$$

where we have made use of (15.9) and (15.10). Thus, the marginal distribution of $\mathbf{x}$ is a Gaussian mixture of the form (15.6). If we have several observations $\mathbf{x}_1, \ldots, \mathbf{x}_N$, then, because we have represented the marginal distribution in the form $p(\mathbf{x}) = \sum_{\mathbf{z}} p(\mathbf{x}, \mathbf{z})$, it follows that for every observed data point $\mathbf{x}_n$ there is a corresponding latent variable $\mathbf{z}_n$.

We have therefore found an equivalent formulation of the Gaussian mixture involving explicit latent variables. It might seem that we have not gained much by doing so. However, we are now able to work with the joint distribution $p(\mathbf{x}, \mathbf{z})$ instead of the marginal distribution $p(\mathbf{x})$, and this will lead to significant simplifications, most notably through the introduction of the EM algorithm.

Another quantity that will play an important role is the conditional probability of $\mathbf{z}$ given $\mathbf{x}$. We will use $\gamma(z_k)$ to denote $p(z_k = 1|\mathbf{x})$, whose value can be found

Telegram: @uni_k

www.konkur.in

using Bayes' theorem:

$$\gamma(z_k) \equiv p(z_k = 1|\mathbf{x}) = \frac{p(z_k = 1)p(\mathbf{x}|z_k = 1)}{\sum_{j=1}^{K} p(z_j = 1)p(\mathbf{x}|z_j = 1)} = \frac{\pi_k \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)}. \tag{15.12}$$

We will view π_k as the prior probability of $z_k = 1$, and the quantity $\gamma(z_k)$ as the corresponding posterior probability once we have observed $\mathbf{x}$. As we will see later, $\gamma(z_k)$ can also be viewed as the *responsibility* that component k takes for 'explaining' the observation $\mathbf{x}$.

Section 14.2.5

We can use ancestral sampling to generate random samples distributed according to the Gaussian mixture model. To do this, we first generate a value for $\mathbf{z}$, which we denote $\widehat{\mathbf{z}}$, from the marginal distribution $p(\mathbf{z})$ and then generate a value for $\mathbf{x}$ from the conditional distribution $p(\mathbf{x}|\widehat{\mathbf{z}})$. We can depict samples from the joint distribution $p(\mathbf{x}, \mathbf{z})$ by plotting points at the corresponding values of $\mathbf{x}$ and then colouring them according to the value of $\mathbf{z}$, in other words according to which Gaussian component was responsible for generating them, as shown in Figure 15.5(a). Similarly samples from the marginal distribution $p(\mathbf{x})$ are obtained by taking the samples from the joint distribution and ignoring the values of $\mathbf{z}$. These are illustrated in Figure 15.5(b) by plotting the $\mathbf{x}$ values without any coloured labels.

We can also use this synthetic data set to illustrate the 'responsibilities' by evaluating, for every data point, the posterior probability for each component in the mixture distribution from which this data set was generated. In particular, we can represent the value of the responsibilities $\gamma(z_{nk})$ associated with data point $\mathbf{x}_n$ by plotting the corresponding point using proportions of red, blue, and green ink given by $\gamma(z_{nk})$ for $k = 1, 2, 3$, respectively, as shown in Figure 15.5(c). So, for instance, a data point for which $\gamma(z_{n1}) = 1$ will be coloured red, whereas one for which $\gamma(z_{n2}) = \gamma(z_{n3}) = 0.5$ will be coloured with equal proportions of blue and green ink and so will appear cyan. This should be compared with Figure 15.5(a) in which the data points were labelled using the true identity of the component from which they were generated.

15.2.1 Likelihood function

Suppose we have a data set of observations $\{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$, and we wish to model this data using a mixture of Gaussians. We can represent this data set as an $N \times D$ matrix $\mathbf{X}$ in which the nth row is given by $\mathbf{x}_n^{\mathrm{T}}$. From (15.6) the log of the likelihood function is given by

$$\ln p(\mathbf{X}|\boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \sum_{n=1}^{N} \ln \left\{ \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \right\}. \tag{15.13}$$

Telegram: @uni_k

www.konkur.in

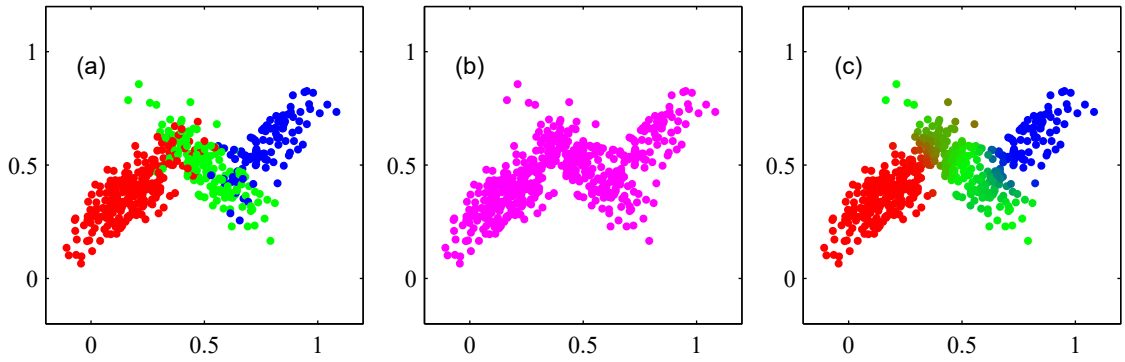


Figure 15.5 Example of 500 points drawn from the mixture of three Gaussians shown in Figure 3.8. (a) Samples from the joint distribution $p(\mathbf{z})p(\mathbf{x}|\mathbf{z})$ in which the three states of $\mathbf{z}$, corresponding to the three components of the mixture, are depicted in red, green, and blue, and (b) the corresponding samples from the marginal distribution $p(\mathbf{x})$, which is obtained by simply ignoring the values of $\mathbf{z}$ and just plotting the $\mathbf{x}$ values. The data set in (a) is said to be *complete*, whereas that in (b) is *incomplete*, as discussed further in Section 15.3. (c) The same samples in which the colours represent the value of the responsibilities $\gamma(z_{nk})$ associated with data point $\mathbf{x}_n$, obtained by plotting the corresponding point using proportions of red, blue, and green ink given by $\gamma(z_{nk})$ for $k = 1, 2, 3$, respectively.

Maximizing this log likelihood function (15.13) is a more complex problem than for a single Gaussian. The difficulty arises from the presence of the summation over k that appears inside the logarithm in (15.13), so that the logarithm function no longer acts directly on the Gaussian. If we set the derivatives of the log likelihood to zero, we will no longer obtain a closed-form solution, as we will see shortly.

Before discussing how to maximize this function, it is worth emphasizing that there is a significant problem associated with the maximum likelihood framework when applied to Gaussian mixture models, due to the presence of singularities. For simplicity, consider a Gaussian mixture whose components have covariance matrices given by $\boldsymbol{\Sigma}_k = \sigma_k^2\mathbf{I}$, where $\mathbf{I}$ is the unit matrix, although the conclusions will hold for general covariance matrices. Suppose that one of the components of the mixture model, let us say the jth component, has its mean $\boldsymbol{\mu}_j$ exactly equal to one of the data points so that $\boldsymbol{\mu}_j = \mathbf{x}_n$ for some value of n. This data point will then contribute a term in the likelihood function of the form

$$\mathcal{N}(\mathbf{x}_n|\mathbf{x}_n, \sigma_j^2\mathbf{I}) = \frac{1}{(2\pi)^{1/2}}\frac{1}{\sigma_j}. \tag{15.14}$$

If we consider the limit $\sigma_j \to 0$, then we see that this term goes to infinity and so the log likelihood function will also go to infinity. Thus, the maximization of the log likelihood function is not a well posed-problem because such singularities will always be present and will occur whenever one of the Gaussian components 'collapses' onto a specific data point. Recall that this problem did not arise with a single Gaussian distribution. To understand the difference, note that if a single Gaussian collapses onto a data point, it will contribute multiplicative factors to the

Telegram: @uni_k

www.konkur.in

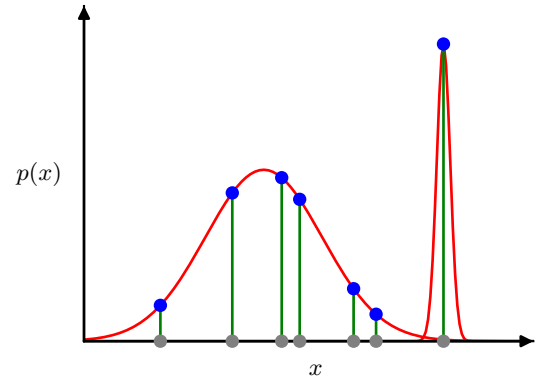


Figure 15.6 Illustration of how singularities in the likelihood function arise with mixtures of Gaussians. This should be compared with a single Gaussian shown in Figure 2.9 for which no singularities arise.

likelihood function arising from the other data points, and these factors will go to zero exponentially fast, giving an overall likelihood that goes to zero rather than infinity. However, once we have (at least) two components in the mixture, one of the components can have a finite variance and therefore assign finite probability to all the data points while the other component can shrink onto one specific data point and thereby contribute an ever increasing additive value to the log likelihood. This is illustrated in Figure 15.6. These singularities provide an example of the over-fitting that can occur in a maximum likelihood approach. When applying maximum likelihood to Gaussian mixture models, we must take steps to avoid finding such pathological solutions and instead seek local maxima of the likelihood function that are well behaved. We can try to avoid the singularities by using suitable heuristics, for instance by detecting when a Gaussian component is collapsing and resetting its mean to a randomly chosen value while also resetting its covariance to some large value and then continuing with the optimization. The singularities can also be avoided by adding a regularization term to the log likelihood corresponding to a prior distribution over the parameters.

Section 15.4.3

A further issue in finding maximum likelihood solutions arises because for any given maximum likelihood solution, a K-component mixture will have a total of $K!$ equivalent solutions corresponding to the $K!$ ways of assigning K sets of parameters to K components. In other words, for any given (non-degenerate) point in the space of parameter values, there will be a further $K! - 1$ additional points all of which give rise to exactly the same distribution. This problem is known as *identifiability* (Casella and Berger, 2002) and is an important issue when we wish to interpret the parameter values discovered by a model. Identifiability will also arise when we discuss models having continuous latent variables. However, when finding a good density model, it is irrelevant because any of the equivalent solutions is as good as any other.

Chapter 16

15.2.2 Maximum likelihood

An elegant and powerful method for finding maximum likelihood solutions for models with latent variables is called the *expectation–maximization* algorithm or *EM* algorithm (Dempster, Laird, and Rubin, 1977; McLachlan and Krishnan, 1997). In this chapter we will give three different derivations of the EM algorithm, each more

Telegram: @uni_k

www.konkur.in

general than the previous. We begin here with a relatively informal treatment in the context of a Gaussian mixture model. We emphasize, however, that EM has broad applicability, and the underlying concepts will be encountered in the context of several different models in this book.

We begin by writing down the conditions that must be satisfied at a maximum of the likelihood function. Setting the derivatives of $\ln p(\mathbf{X}|\boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Sigma})$ in (15.13) with respect to the means $\boldsymbol{\mu}_k$ of the Gaussian components to zero, we obtain

$$0 = \sum_{n=1}^{N} \underbrace{\frac{\pi_k \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\sum_j \pi_j \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)}}_{\gamma(z_{nk})} \boldsymbol{\Sigma}_k^{-1}(\mathbf{x}_n - \boldsymbol{\mu}_k) \tag{15.15}$$

where we have made use of the form (3.26) for the Gaussian distribution. Note that the posterior probabilities, or responsibilities, $\gamma(z_{nk})$ given by (15.12) appear naturally on the right-hand side. Multiplying by $\boldsymbol{\Sigma}_k$ (which we assume to be non-singular) and rearranging we obtain

$$\boldsymbol{\mu}_k = \frac{1}{N_k} \sum_{n=1}^{N} \gamma(z_{nk}) \mathbf{x}_n \tag{15.16}$$

where we have defined

$$N_k = \sum_{n=1}^{N} \gamma(z_{nk}). \tag{15.17}$$

We can interpret N_k as the effective number of points assigned to cluster k. Note carefully the form of this solution. We see that the mean $\boldsymbol{\mu}_k$ for the kth Gaussian component is obtained by taking a weighted mean of all the points in the data set, in which the weighting factor for data point $\mathbf{x}_n$ is given by the posterior probability $\gamma(z_{nk})$ that component k was responsible for generating $\mathbf{x}_n$.

If we set the derivative of $\ln p(\mathbf{X}|\boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Sigma})$ with respect to $\boldsymbol{\Sigma}_k$ to zero and follow a similar line of reasoning by making use of the result for the maximum likelihood solution for the covariance matrix of a single Gaussian, we obtain

Section 3.2.7

$$\boldsymbol{\Sigma}_k = \frac{1}{N_k} \sum_{n=1}^{N} \gamma(z_{nk})(\mathbf{x}_n - \boldsymbol{\mu}_k)(\mathbf{x}_n - \boldsymbol{\mu}_k)^{\mathrm{T}}, \tag{15.18}$$

which has the same form as the corresponding result for a single Gaussian fitted to the data set, but again with each data point weighted by the corresponding posterior probability and with the denominator given by the effective number of points associated with the corresponding component.

Finally, we maximize $\ln p(\mathbf{X}|\boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Sigma})$ with respect to the mixing coefficients π_k. Here we must take account of the constraint (15.8), which requires the mixing coefficients to sum to one. This can be achieved using a Lagrange multiplier λ and

Appendix C

Telegram: @uni_k

www.konkur.in

maximizing the following quantity:

$$\ln p(\mathbf{X}|\boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Sigma}) + \lambda \left(\sum_{k=1}^{K} \pi_k - 1 \right), \tag{15.19}$$

which gives

$$0 = \sum_{n=1}^{N} \frac{\mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\sum_j \pi_j \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)} + \lambda \tag{15.20}$$

where again we see the appearance of the responsibilities. If we now multiply both sides by π_k and sum over k making use of the constraint (15.8), we find $\lambda = -N$. Using this to eliminate λ and rearranging, we obtain

$$\pi_k = \frac{N_k}{N} \tag{15.21}$$

so that the mixing coefficient for the kth component is given by the average responsibility which that component takes for explaining the data points.

Note that the results (15.16), (15.18), and (15.21) do not constitute a closed-form solution for the parameters of the mixture model because the responsibilities $\gamma(z_{nk})$ depend on those parameters in a complex way through (15.12). However, these results do suggest a simple iterative scheme for finding a solution to the maximum likelihood problem, which as we will see turns out to be an instance of the EM algorithm for the particular case of the Gaussian mixture model. We first choose some initial values for the means, covariances, and mixing coefficients. Then we alternate between the following two updates, which we will call the E step and the M step for reasons that will become apparent shortly. In the *expectation* step, or E step, we use the current values for the parameters to evaluate the posterior probabilities, or responsibilities, given by (15.12). We then use these probabilities in the *maximization* step, or M step, to re-estimate the means, covariances, and mixing coefficients using the results (15.16), (15.18), and (15.21). Note that in so doing, we first evaluate the new means using (15.16) and then use these new values to find the covariances using (15.18), in keeping with the corresponding result for a single Gaussian distribution. We will show that each update to the parameters resulting from an E step followed by an M step is guaranteed to increase the log likelihood function. In practice, the algorithm is deemed to have converged when the change in the log likelihood function, or alternatively in the parameters, falls below some threshold.

Section 15.3

We illustrate the EM algorithm for a mixture of two Gaussians applied to the re-scaled Old Faithful data in Figure 15.7. Here a mixture of two Gaussians is used, with centres initialized using the same values as for the K-means algorithm in Figure 15.1 and with covariance matrices initialized to be proportional to the unit matrix. Plot (a) shows the data points in green, together with the initial configuration of the mixture model in which the one standard-deviation contours for the two Gaussian components are shown as blue and red circles. Plot (b) shows the result of the initial E step, in which each data point is depicted using a proportion of blue ink equal to the posterior probability of having been generated from the blue component and a

Telegram: @uni_k

www.konkur.in

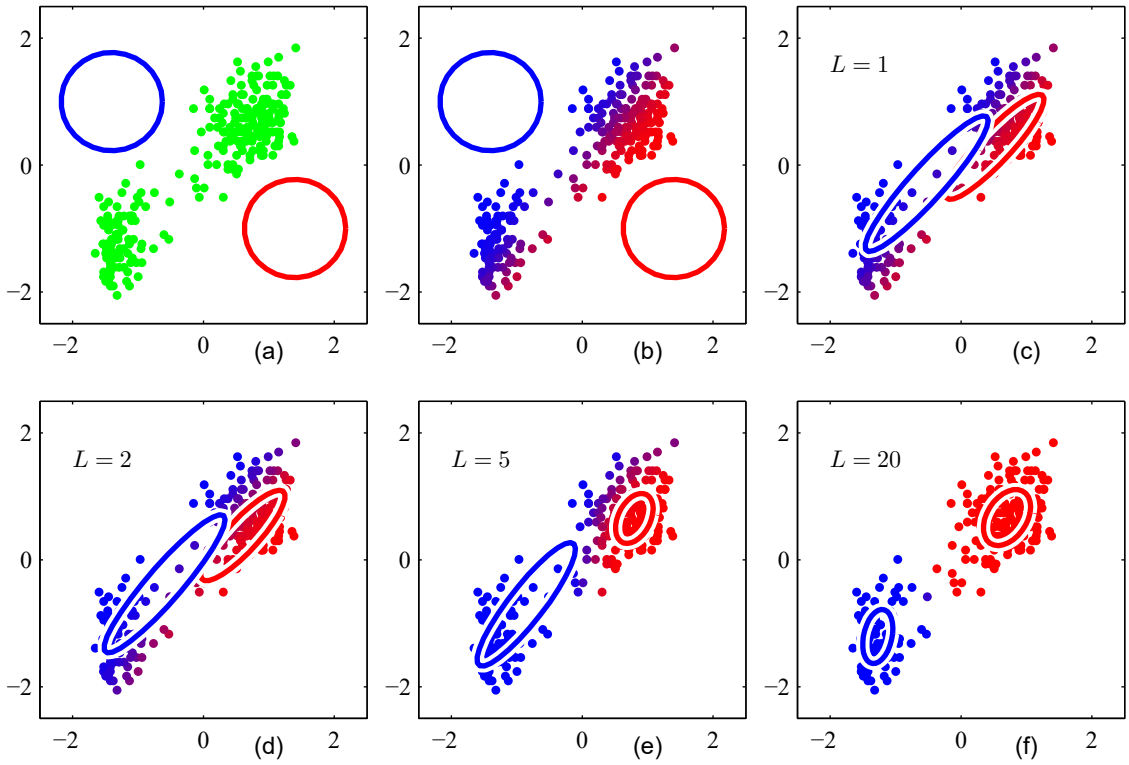


Figure 15.7 Application of the EM algorithm to the Old Faithful data set as used for the illustration of the K-means algorithm in Figure 15.1. See the text for details.

corresponding proportion of red ink given by the posterior probability of having been generated by the red component. Thus, points that have a roughly equal probability for belonging to either cluster appear purple. The situation after the first M step is shown in plot (c), in which the mean of the blue Gaussian has moved to the mean of the data set, weighted by the probabilities of each data point belonging to the blue cluster. In other words it has moved to the centre of mass of the blue ink. Similarly, the covariance of the blue Gaussian is set equal to the covariance of the blue ink. Analogous results hold for the red component. Plots (d), (e), and (f) show the results after 2, 5, and 20 complete cycles of EM, respectively. In plot (f) the algorithm is close to convergence.

Note that the EM algorithm takes many more iterations to reach (approximate) convergence compared with the K-means algorithm and that each cycle requires significantly more computation. It is therefore common to run the K-means algorithm to find a suitable initialization for a Gaussian mixture model that is subsequently adapted using EM. The covariance matrices can conveniently be initialized to the sample covariances of the clusters found by the K-means algorithm, and the mixing coefficients can be set to the fractions of data points assigned to the respective

Telegram: @uni_k

www.konkur.in

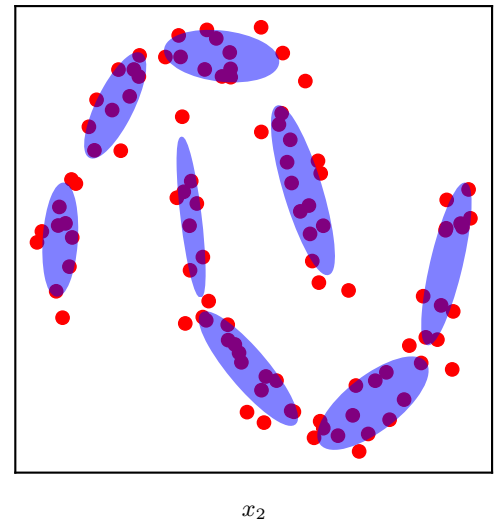


Figure 15.8 A Gaussian mixture model fitted to the 'two-moons' data set, showing that a large number of mixture components may be required to give an accurate representation of a complex data distribution. Here the ellipses represent the contours of constant density for the corresponding mixture components. As we move to spaces of larger dimensionality, the number of components required to model a distribution accurately can become unacceptably large.

clusters. Techniques such as parameter regularization must be employed to avoid singularities of the likelihood function in which a Gaussian component collapses onto a particular data point. It should be emphasized that there will generally be multiple local maxima of the log likelihood function and that EM is not guaranteed to find the largest of these maxima. Because the EM algorithm for Gaussian mixtures plays such an important role, we summarize it in Algorithm 15.2.

Mixture models are very flexible and can approximate complicated distributions to high accuracy given a sufficient number of components if the model parameters are chosen appropriately. In practice, however, the number of components can be extremely large, especially in spaces of high dimensionality. This problem is illustrated for the two-moons data set in Figure 15.8. Nevertheless, mixture models are useful in many applications. Also, an understanding of mixture models lays the foundations for models with continuous latent variables and for generative models based on deep neural networks, which have much better scaling to spaces of high dimensionality.

Chapter 16

15.3. Expectation–Maximization Algorithm

We turn now to a more general view of the EM algorithm in which we focus on the role of latent variables. As before we denote the set of all observed data points by $\mathbf{X}$, in which the nth row represents $\mathbf{x}_n^{\mathrm{T}}$. Similarly, the corresponding latent variables will be denoted by an $N \times K$ matrix $\mathbf{Z}$ with rows $\mathbf{z}_n^{\mathrm{T}}$. If we assume that the data points are drawn independently from the distribution, then we can express the Gaussian mixture model for this i.i.d. data set using the graphical representation shown in Figure 15.9. The set of all model parameters is denoted by $\boldsymbol{\theta}$, and so the log likelihood function

Telegram: @uni_k

www.konkur.in

Algorithm 15.2: EM algorithm for a Gaussian mixture model

Input: Initial model parameters $\{\boldsymbol{\mu}_k\}, \{\boldsymbol{\Sigma}_k\}, \{\pi_k\}$
Data set $\{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$
Output: Final model parameters $\{\boldsymbol{\mu}_k\}, \{\boldsymbol{\Sigma}_k\}, \{\pi_k\}$

repeat
  // E step
  for $n \in \{1, \ldots, N\}$ **do**
    for $k \in \{1, \ldots, K\}$ **do**
      $$\gamma(z_{nk}) \leftarrow \frac{\pi_k \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)}$$
    end for
  end for
  // M step
  for $k \in \{1, \ldots, K\}$ **do**
    $$N_k \leftarrow \sum_{n=1}^{N} \gamma(z_{nk})$$
    $$\boldsymbol{\mu}_k \leftarrow \frac{1}{N_k} \sum_{n=1}^{N} \gamma(z_{nk}) \mathbf{x}_n$$
    $$\boldsymbol{\Sigma}_k \leftarrow \frac{1}{N_k} \sum_{n=1}^{N} \gamma(z_{nk}) (\mathbf{x}_n - \boldsymbol{\mu}_k) (\mathbf{x}_n - \boldsymbol{\mu}_k)^{\mathrm{T}}$$
    $$\pi_k \leftarrow \frac{N_k}{N}$$
  end for
  // Log likelihood
  $$\mathcal{L} \leftarrow \sum_{n=1}^{N} \ln \left\{ \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \right\}$$
until convergence
return $\{\mathbf{x}_k\}, \{\boldsymbol{\Sigma}_k\}, \{\pi_k\}$

Telegram: @uni_k

www.konkur.in

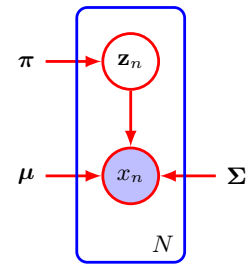


Figure 15.9 Graphical representation of a Gaussian mixture model for a set of N i.i.d. data points $\{\mathbf{x}_n\}$, with corresponding latent points $\{\mathbf{z}_n\}$, where $n = 1, \ldots, N$.

is given by

$$\ln p(\mathbf{X}|\boldsymbol{\theta}) = \ln \left\{ \sum_{\mathbf{Z}} p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\theta}) \right\}. \tag{15.22}$$

Chapter 16

Note that our discussion will apply equally well to continuous latent variables simply by replacing the sum over $\mathbf{Z}$ with an integral.

A key observation is that the summation over the latent variables appears inside the logarithm. Even if the joint distribution $p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\theta})$ belongs to the exponential family, the marginal distribution $p(\mathbf{X}|\boldsymbol{\theta})$ typically does not as a result of this summation. The presence of the sum prevents the logarithm from acting directly on the joint distribution, resulting in complicated expressions for the maximum likelihood solution.

Now suppose that, for each observation in $\mathbf{X}$, we were told the corresponding value of the latent variable $\mathbf{Z}$. We will call $\{\mathbf{X}, \mathbf{Z}\}$ the *complete* data set, and we will refer to the actual observed data $\mathbf{X}$ as *incomplete*, as illustrated in Figure 15.5. The likelihood function for the complete data set simply takes the form $\ln p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\theta})$, and we will suppose that maximization of this complete-data log likelihood function is straightforward.

In practice, however, we are not given the complete data set $\{\mathbf{X}, \mathbf{Z}\}$ but only the incomplete data $\mathbf{X}$. Our state of knowledge of the values of the latent variables in $\mathbf{Z}$ is given only by the posterior distribution $p(\mathbf{Z}|\mathbf{X}, \boldsymbol{\theta})$. Because we cannot use the complete-data log likelihood, we consider instead its expected value under the posterior distribution of the latent variables, which corresponds (as we will see) to the E step of the EM algorithm. In the subsequent M step, we maximize this expectation. If the current estimate for the parameters is denoted by $\boldsymbol{\theta}^{\text{old}}$, then a pair of successive E and M steps gives rise to a revised estimate $\boldsymbol{\theta}^{\text{new}}$. The algorithm is initialized by choosing some starting value for the parameters $\boldsymbol{\theta}_0$. Although this use of the expectation may seem somewhat arbitrary, we will see the motivation for this choice when we give a deeper treatment of EM in Section 15.4.

In the E step, we use the current parameter values $\boldsymbol{\theta}^{\text{old}}$ to find the posterior distribution of the latent variables given by $p(\mathbf{Z}|\mathbf{X}, \boldsymbol{\theta}^{\text{old}})$. We then use this posterior distribution to find the expectation of the complete-data log likelihood evaluated for some general parameter value $\boldsymbol{\theta}$. This expectation, denoted by $\mathcal{Q}(\boldsymbol{\theta}, \boldsymbol{\theta}^{\text{old}})$, is given by

$$\mathcal{Q}(\boldsymbol{\theta}, \boldsymbol{\theta}^{\text{old}}) = \sum_{\mathbf{Z}} p(\mathbf{Z}|\mathbf{X}, \boldsymbol{\theta}^{\text{old}}) \ln p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\theta}). \tag{15.23}$$

Telegram: @uni_k

www.konkur.in

Algorithm 15.3: General EM algorithm

Input: Joint distribution $p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\theta})$
Initial parameters $\boldsymbol{\theta}^{\text{old}}$
Data set $\mathbf{x}_1, \ldots, \mathbf{x}_N$
Output: Final parameters $\boldsymbol{\theta}$

---

repeat
  $\mathcal{Q}(\boldsymbol{\theta}, \boldsymbol{\theta}^{\text{old}}) \leftarrow \sum_{\mathbf{Z}} p(\mathbf{Z}|\mathbf{X}, \boldsymbol{\theta}^{\text{old}}) \ln p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\theta})$ `// E step`
  $\boldsymbol{\theta}^{\text{new}} \leftarrow \arg\max_{\boldsymbol{\theta}} \mathcal{Q}(\boldsymbol{\theta}, \boldsymbol{\theta}^{\text{old}})$ `// M step`
  $\mathcal{L} \leftarrow p(\mathbf{X}|\boldsymbol{\theta}^{\text{new}})$ `// Evaluate log likelihood`
  $\boldsymbol{\theta}^{\text{old}} \leftarrow \boldsymbol{\theta}^{\text{new}}$ `// Update the parameters`
until convergence
return $\boldsymbol{\theta}^{\text{new}}$

In the M step, we determine the revised parameter estimate $\boldsymbol{\theta}^{\text{new}}$ by maximizing this function:

$$\boldsymbol{\theta}^{\text{new}} = \arg\max_{\boldsymbol{\theta}} \mathcal{Q}(\boldsymbol{\theta}, \boldsymbol{\theta}^{\text{old}}). \tag{15.24}$$

Note that in the definition of $\mathcal{Q}(\boldsymbol{\theta}, \boldsymbol{\theta}^{\text{old}})$, the logarithm acts directly on the joint distribution $p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\theta})$, and so the corresponding M-step maximization will, according to our assumption, be tractable. The general EM algorithm is summarized in Algorithm 15.3. It has the property, as we will show later, that each cycle of EM will increase the incomplete-data log likelihood (unless it is already at a local maximum).

Section 15.4.1

The EM algorithm can also be used to find MAP (maximum posterior) solutions for models in which a prior $p(\boldsymbol{\theta})$ is defined over the parameters. In this case the E step remains the same as in the maximum likelihood case, whereas in the M step the quantity to be maximized is given by $\mathcal{Q}(\boldsymbol{\theta}, \boldsymbol{\theta}^{\text{old}}) + \ln p(\boldsymbol{\theta})$. Suitable choices for the prior will remove the singularities of the kind illustrated in Figure 15.6.

Exercise 15.5

Here we have considered the use of the EM algorithm to maximize a likelihood function when there are discrete latent variables. However, it can also be applied when the unobserved variables correspond to missing values in the data set. The distribution of the observed values is obtained by taking the joint distribution of all the variables and then marginalizing over the missing ones. EM can then be used to maximize the corresponding likelihood function. This will be a valid procedure if the data values are *missing at random*, meaning that the mechanism causing values to be missing does not depend on the unobserved values. In many situations this will not be the case, for instance if a sensor fails to return a value whenever the quantity it is measuring exceeds some threshold.

Telegram: @uni_k

www.konkur.in

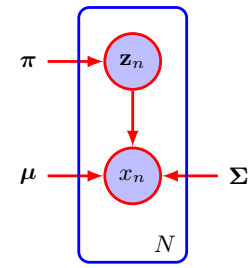


Figure 15.10 This shows the same graph as in Figure 15.9 except that we now suppose that the discrete variables $\mathbf{z}_n$ are observed, as well as the data variables $\mathbf{x}_n$.

15.3.1 Gaussian mixtures

We now consider the application of this latent-variable view of EM to the specific case of a Gaussian mixture model. Recall that our goal is to maximize the log likelihood function (15.13), which is computed using the observed data set $\mathbf{X}$, and we saw that this was more difficult than with a single Gaussian distribution due to the summation over k that occurs inside the logarithm. Suppose then that in addition to the observed data set $\mathbf{X}$, we were also given the values of the corresponding discrete variables $\mathbf{Z}$. Recall that Figure 15.5(a) shows a *complete* data set (i.e., one that includes labels showing which component generated each data point) whereas Figure 15.5(b) shows the corresponding *incomplete* data set. A graphical model for the complete data is shown in Figure 15.10.

Now consider the problem of maximizing the likelihood for the complete data set $\{\mathbf{X}, \mathbf{Z}\}$. From (15.9) and (15.10), this likelihood function takes the form

$$p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\mu}, \boldsymbol{\Sigma}, \boldsymbol{\pi}) = \prod_{n=1}^{N} \prod_{k=1}^{K} \pi_k^{z_{nk}} \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)^{z_{nk}} \tag{15.25}$$

where z_{nk} denotes the kth component of $\mathbf{z}_n$. Taking the logarithm, we obtain

$$\ln p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\mu}, \boldsymbol{\Sigma}, \boldsymbol{\pi}) = \sum_{n=1}^{N} \sum_{k=1}^{K} z_{nk} \left\{ \ln \pi_k + \ln \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \right\}. \tag{15.26}$$

Comparison with the log likelihood function (15.13) for the incomplete data shows that the summation over k and the logarithm have been interchanged. The logarithm now acts directly on the Gaussian distribution, which itself is a member of the exponential family. Not surprisingly, this leads to a much simpler solution to the maximum likelihood problem, as we now show. Consider first the maximization with respect to the means and covariances. Because $\mathbf{z}_n$ is a K-dimensional vector with all elements equal to 0 except for a single element having the value 1, the complete-data log likelihood function is simply a sum of K independent contributions, one for each mixture component. Thus, the maximization with respect to a mean or a covariance is exactly as for a single Gaussian, except that it involves only the subset of data points that are 'assigned' to that component. For the maximization with respect to the mixing coefficients, note that these are coupled for different values of k by virtue of the summation constraint (15.8). Again, this can be enforced using a Lagrange

Telegram: @uni_k

www.konkur.in

multiplier as before, which leads to the result

$$\pi_k = \frac{1}{N}\sum_{n=1}^{N} z_{nk} \tag{15.27}$$

so that the mixing coefficients are equal to the fractions of data points assigned to the corresponding components.

Thus, we see that the complete-data log likelihood function can be maximized trivially in closed form. In practice, however, we do not have values for the latent variables. Therefore, as discussed earlier, we consider the expectation, with respect to the posterior distribution of the latent variables, of the complete-data log likelihood. Using (15.9) and (15.10) together with Bayes' theorem, we see that this posterior distribution takes the form

$$p(\mathbf{Z}|\mathbf{X}, \boldsymbol{\mu}, \boldsymbol{\Sigma}, \boldsymbol{\pi}) \propto \prod_{n=1}^{N}\prod_{k=1}^{K} \left[\pi_k \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)\right]^{z_{nk}} . \tag{15.28}$$

Exercise 15.6
Section 11.2

We see that this factorizes over n so that under the posterior distribution, the $\{\mathbf{z}_n\}$ are independent. This is easily verified by inspecting the directed graph in Figure 15.9 and making use of the d-separation criterion. The expected value of the indicator variable z_{nk} under this posterior distribution is then given by

$$\begin{aligned}\mathbb{E}[z_{nk}] &= \frac{\sum_{\mathbf{z}_n} z_{nk} \prod_{k'} \left[\pi_{k'} \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_{k'}, \boldsymbol{\Sigma}_{k'})\right]^{z_{nk'}}}{\sum_{\mathbf{z}_n} \prod_{j} \left[\pi_j \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)\right]^{z_{nj}}} \\ &= \frac{\pi_k \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)} = \gamma(z_{nk}),\end{aligned} \tag{15.29}$$

which is just the responsibility of component k for data point $\mathbf{x}_n$. The expected value of the complete-data log likelihood function is therefore given by

$$\mathbb{E}_{\mathbf{Z}}[\ln p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\mu}, \boldsymbol{\Sigma}, \boldsymbol{\pi})] = \sum_{n=1}^{N}\sum_{k=1}^{K} \gamma(z_{nk}) \left\{\ln \pi_k + \ln \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)\right\} . \tag{15.30}$$

We can now proceed as follows. First we choose some initial values for the parameters $\boldsymbol{\mu}^{\text{old}}$, $\boldsymbol{\Sigma}^{\text{old}}$, and $\boldsymbol{\pi}^{\text{old}}$, and we use these to evaluate the responsibilities (the E step). We then keep the responsibilities fixed and maximize (15.30) with respect to $\boldsymbol{\mu}_k$, $\boldsymbol{\Sigma}_k$, and π_k (the M step). This leads to closed-form solutions for $\boldsymbol{\mu}^{\text{new}}$, $\boldsymbol{\Sigma}^{\text{new}}$, and $\boldsymbol{\pi}^{\text{new}}$ given by (15.16), (15.18), and (15.21) as before. This is precisely the EM algorithm for Gaussian mixtures as derived earlier. We will gain more insight into the role of the expected complete-data log likelihood function when discuss the convergence of the EM algorithm in Section 15.4.

Exercise 15.9

Telegram: @uni_k

www.konkur.in

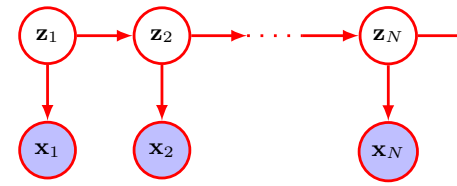


Figure 15.11 The probabilistic graphical model for sequential data corresponding to a hidden Markov model. The discrete latent variables are no longer independent but form a Markov chain.

Throughout this chapter we assume that the data observations are i.i.d. For ordered observations that form a sequence, the mixture model can be extended by connecting the latent variables in a Markov chain to give a *hidden Markov model* whose graphical structure is shown in Figure 15.11. The EM algorithm can be extended to this more complex model in which the E step involves a sequential calculation in which messages are passed along the chain of latent variables (Bishop, 2006).

15.3.2 Relation to K-means

Comparison of the K-means algorithm with the EM algorithm for Gaussian mixtures shows that there is a close similarity. Whereas the K-means algorithm performs a *hard* assignment of data points to clusters in which each data point is associated uniquely with one cluster, the EM algorithm makes a *soft* assignment based on the posterior probabilities. In fact, we can derive the K-means algorithm as a particular limit of EM for Gaussian mixtures as follows.

Consider a Gaussian mixture model in which the covariance matrices of the mixture components are given by $\epsilon\mathbf{I}$, where ϵ is a variance parameter that is shared by all the components, and $\mathbf{I}$ is the identity matrix, so that

$$p(\mathbf{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) = \frac{1}{(2\pi\epsilon)^{D/2}} \exp\left\{-\frac{1}{2\epsilon}\|\mathbf{x} - \boldsymbol{\mu}_k\|^2\right\}. \tag{15.31}$$

We now consider the EM algorithm for a mixture of K Gaussians of this form in which we treat ϵ as a fixed constant, instead of a parameter to be re-estimated. From (15.12) the posterior probabilities, or responsibilities, for a particular data point $\mathbf{x}_n$ are given by

$$\gamma(z_{nk}) = \frac{\pi_k \exp\{-\|\mathbf{x}_n - \boldsymbol{\mu}_k\|^2/2\epsilon\}}{\sum_j \pi_j \exp\{-\|\mathbf{x}_n - \boldsymbol{\mu}_j\|^2/2\epsilon\}}. \tag{15.32}$$

Consider the limit $\epsilon \to 0$. The denominator consists of a sum of terms indexed by j each of which goes to zero. The particular term for which $\|\mathbf{x}_n - \boldsymbol{\mu}_j\|^2$ is smallest, say $j = l$, will go to zero most slowly and will then dominate this sum. Therefore, the responsibilities $\gamma(z_{nk})$ for the data point $\mathbf{x}_n$ all go to zero except for term l, for which the responsibility $\gamma(z_{nl})$ will go to unity. Note that this holds independently of the values of the π_k so long as none of the π_k is zero. Thus, in this limit, we obtain a hard assignment of data points to clusters, just as in the K-means algorithm, so that $\gamma(z_{nk}) \to r_{nk}$ where r_{nk} is defined by (15.2). Each data point is thereby assigned to the cluster having the closest mean. The EM re-estimation equation for the $\boldsymbol{\mu}_k$, given by (15.16), then reduces to the K-means result (15.4). Note that the re-estimation formula for the mixing coefficients (15.21) simply resets the value of π_k to be equal

Telegram: @uni_k

www.konkur.in

to the fraction of data points assigned to cluster k, although these parameters no longer play an active role in the algorithm.

Finally, in the limit $\epsilon \to 0$, the expected complete-data log likelihood, given by (15.30), becomes

Exercise 15.12

$$\mathbb{E}_{\mathbf{Z}}[\ln p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\mu}, \boldsymbol{\Sigma}, \boldsymbol{\pi})] \to -\frac{1}{2}\sum_{n=1}^{N}\sum_{k=1}^{K} r_{nk}\|\mathbf{x}_n - \boldsymbol{\mu}_k\|^2 + \text{const.} \tag{15.33}$$

Thus, we see that in this limit, maximizing the expected complete-data log likelihood is equivalent to minimizing the error measure J for the K-means algorithm given by (15.1). Note that the K-means algorithm does not estimate the covariances of the clusters but only the cluster means.

15.3.3 Mixtures of Bernoulli distributions

So far in this chapter, we have focused on distributions over continuous variables described by mixtures of Gaussians. As a further example of mixture modelling and to illustrate the EM algorithm in a different context, we now discuss mixtures of discrete binary variables described by Bernoulli distributions. This model is also known as *latent class analysis* (Lazarsfeld and Henry, 1968; McLachlan and Peel, 2000).

Consider a set of D binary variables x_i, where $i = 1, \ldots, D$, each of which is governed by a Bernoulli distribution with parameter μ_i, so that

Section 3.1.1

$$p(\mathbf{x}|\boldsymbol{\mu}) = \prod_{i=1}^{D} \mu_i^{x_i}(1 - \mu_i)^{(1-x_i)} \tag{15.34}$$

where $\mathbf{x} = (x_1, \ldots, x_D)^{\mathrm{T}}$ and $\boldsymbol{\mu} = (\mu_1, \ldots, \mu_D)^{\mathrm{T}}$. We see that the individual variables x_i are independent, given $\boldsymbol{\mu}$. The mean and covariance of this distribution are easily seen to be

Exercise 15.13

$$\mathbb{E}[\mathbf{x}] = \boldsymbol{\mu} \tag{15.35}$$

$$\text{cov}[\mathbf{x}] = \text{diag}\{\mu_i(1 - \mu_i)\}. \tag{15.36}$$

Now let us consider a finite mixture of these distributions given by

$$p(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\pi}) = \sum_{k=1}^{K} \pi_k p(\mathbf{x}|\boldsymbol{\mu}_k) \tag{15.37}$$

where $\boldsymbol{\mu} = \{\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_K\}$, $\boldsymbol{\pi} = \{\pi_1, \ldots, \pi_K\}$, and

$$p(\mathbf{x}|\boldsymbol{\mu}_k) = \prod_{i=1}^{D} \mu_{ki}^{x_i}(1 - \mu_{ki})^{(1-x_i)}. \tag{15.38}$$

The mixing coefficients satisfy (15.7) and (15.8). The mean and covariance of this mixture distribution are given by

Exercise 15.14

Telegram: @uni_k

www.konkur.in

$$\mathbb{E}[\mathbf{x}] = \sum_{k=1}^{K} \pi_k \boldsymbol{\mu}_k \tag{15.39}$$

$$\mathrm{cov}[\mathbf{x}] = \sum_{k=1}^{K} \pi_k \left\{ \boldsymbol{\Sigma}_k + \boldsymbol{\mu}_k \boldsymbol{\mu}_k^{\mathrm{T}} \right\} - \mathbb{E}[\mathbf{x}]\mathbb{E}[\mathbf{x}]^{\mathrm{T}} \tag{15.40}$$

where $\boldsymbol{\Sigma}_k = \mathrm{diag}\{\mu_{ki}(1 - \mu_{ki})\}$. Because the covariance matrix $\mathrm{cov}[\mathbf{x}]$ is no longer diagonal, the mixture distribution can capture correlations between the variables, unlike a single Bernoulli distribution.

If we are given a data set $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$ then the log likelihood function for this model is given by

$$\ln p(\mathbf{X}|\boldsymbol{\mu}, \boldsymbol{\pi}) = \sum_{n=1}^{N} \ln \left\{ \sum_{k=1}^{K} \pi_k p(\mathbf{x}_n|\boldsymbol{\mu}_k) \right\}. \tag{15.41}$$

Again we see the appearance of the summation inside the logarithm, so that the maximum likelihood solution no longer has closed form.

We now derive the EM algorithm for maximizing the likelihood function for the mixture of Bernoulli distributions. To do this, we first introduce an explicit discrete latent variable $\mathbf{z}$ associated with each instance of $\mathbf{x}$. As with the Gaussian mixture, $\mathbf{z}$ has a 1-of-K coding so that $\mathbf{z} = (z_1, \ldots, z_K)^{\mathrm{T}}$ is a binary K-dimensional vector having a single component equal to 1, with all other components equal to 0. We can then write the conditional distribution of $\mathbf{x}$, given the latent variable, as

$$p(\mathbf{x}|\mathbf{z}, \boldsymbol{\mu}) = \prod_{k=1}^{K} p(\mathbf{x}|\boldsymbol{\mu}_k)^{z_k} \tag{15.42}$$

whereas the prior distribution for the latent variables is the same as for the mixture-of-Gaussians model, so that

$$p(\mathbf{z}|\boldsymbol{\pi}) = \prod_{k=1}^{K} \pi_k^{z_k}. \tag{15.43}$$

If we form the product of $p(\mathbf{x}|\mathbf{z}, \boldsymbol{\mu})$ and $p(\mathbf{z}|\boldsymbol{\pi})$ and then marginalize over $\mathbf{z}$, then we recover (15.37). *Exercise 15.16*

To derive the EM algorithm, we first write down the complete-data log likelihood function, which is given by

$$\ln p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\mu}, \boldsymbol{\pi}) = \sum_{n=1}^{N} \sum_{k=1}^{K} z_{nk} \left\{ \ln \pi_k + \sum_{i=1}^{D} \left[x_{ni} \ln \mu_{ki} + (1 - x_{ni}) \ln(1 - \mu_{ki}) \right] \right\} \tag{15.44}$$

Telegram: @uni_k

www.konkur.in

where $\mathbf{X} = \{\mathbf{x}_n\}$ and $\mathbf{Z} = \{\mathbf{z}_n\}$. Next we take the expectation of the complete-data log likelihood with respect to the posterior distribution of the latent variables to give

$$\mathbb{E}_{\mathbf{Z}}[\ln p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\mu}, \boldsymbol{\pi})] = \sum_{n=1}^{N}\sum_{k=1}^{K}\gamma(z_{nk})\left\{\ln \pi_k + \sum_{i=1}^{D}[x_{ni}\ln \mu_{ki} + (1 - x_{ni})\ln(1 - \mu_{ki})]\right\} \tag{15.45}$$

where $\gamma(z_{nk}) = \mathbb{E}[z_{nk}]$ is the posterior probability, or responsibility, of component k given data point $\mathbf{x}_n$. In the E step, these responsibilities are evaluated using Bayes' theorem, which takes the form

$$\begin{aligned}\gamma(z_{nk}) = \mathbb{E}[z_{nk}] &= \frac{\sum_{\mathbf{z}_n} z_{nk}\prod_{k'}[\pi_{k'}p(\mathbf{x}_n|\boldsymbol{\mu}_{k'})]^{z_{nk'}}}{\sum_{\mathbf{z}_n}\prod_{j}[\pi_j p(\mathbf{x}_n|\boldsymbol{\mu}_j)]^{z_{nj}}} \\ &= \frac{\pi_k p(\mathbf{x}_n|\boldsymbol{\mu}_k)}{\sum_{j=1}^{K}\pi_j p(\mathbf{x}_n|\boldsymbol{\mu}_j)}.\end{aligned} \tag{15.46}$$

If we consider the sum over n in (15.45), we see that the responsibilities enter only through two terms, which can be written as

$$N_k = \sum_{n=1}^{N}\gamma(z_{nk}) \tag{15.47}$$

$$\overline{\mathbf{x}}_k = \frac{1}{N_k}\sum_{n=1}^{N}\gamma(z_{nk})\mathbf{x}_n \tag{15.48}$$

where N_k is the effective number of data points associated with component k. In the M step, we maximize the expected complete-data log likelihood with respect to the parameters $\boldsymbol{\mu}_k$ and $\boldsymbol{\pi}$. If we set the derivative of (15.45) with respect to $\boldsymbol{\mu}_k$ equal to zero and rearrange the terms, we obtain

Exercise 15.17

$$\boldsymbol{\mu}_k = \overline{\mathbf{x}}_k. \tag{15.49}$$

We see that this sets the mean of component k equal to a weighted mean of the data, with weighting coefficients given by the responsibilities that component k takes for each of the data points. For the maximization with respect to π_k, we need to introduce a Lagrange multiplier to enforce the constraint $\sum_k \pi_k = 1$. Following analogous steps to those used for the mixture of Gaussians, we then obtain

Exercise 15.18

$$\pi_k = \frac{N_k}{N}, \tag{15.50}$$

Telegram: @uni_k

www.konkur.in

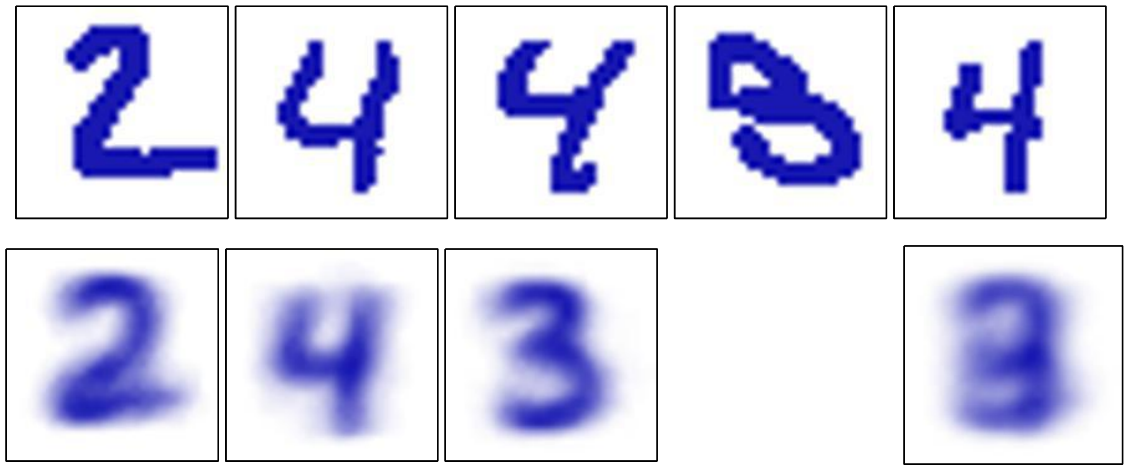


Figure 15.12 Illustration of the Bernoulli mixture model in which the top row shows examples from the digits data set after converting the pixel values from grey scale to binary using a threshold of 0.5. On the bottom row the first three images show the parameters μ_{ki} for each of the three components in the mixture model. As a comparison, we also fit the same data set using a single multivariate Bernoulli distribution, again using maximum likelihood. This amounts to simply averaging the counts in each pixel and is shown by the right-most image on the bottom row.

which represents the intuitively reasonable result that the mixing coefficient for component k is given by the effective fraction of points in the data set explained by that component.

Note that in contrast to the mixture of Gaussians, there are no singularities in which the likelihood function goes to infinity. This can be seen by noting that the likelihood function is bounded above because $0 \leqslant p(\mathbf{x}_n|\boldsymbol{\mu}_k) \leqslant 1$. There exist solutions for which the likelihood function is zero, but these will not be found by EM provided it is not initialized to a pathological starting point, because the EM algorithm always increases the value of the likelihood function, until a local maximum is found.

Exercise 15.19

Section 15.3

We illustrate the Bernoulli mixture model in Figure 15.12 by using it to model handwritten digits. Here the digit images have been turned into binary vectors by setting all elements whose values exceed 0.5 to 1 and setting the remaining elements to 0. We now fit a data set of $N = 600$ such digits, comprising the digits '2', '3', and '4', with a mixture of $K = 3$ Bernoulli distributions by running 10 iterations of the EM algorithm. The mixing coefficients were initialized to $\pi_k = 1/K$, and the parameters μ_{kj} were set to random values chosen uniformly in the range $(0.25, 0.75)$ and then normalized to satisfy the constraint that $\sum_j \mu_{kj} = 1$. We see that a mixture of three Bernoulli distributions is able to find the three clusters in the data set corresponding to the different digits. It is straightforward to extend the analysis of Bernoulli mixtures to the case of multinomial binary variables having $M > 2$ states by making use of the discrete distribution (3.14).

Exercise 15.20

Telegram: @uni_k

www.konkur.in

15.4. Evidence Lower Bound

We now present an even more general perspective on the EM algorithm by deriving a lower bound on the log likelihood function, which is known as the *evidence lower bound* or *ELBO*. It is sometimes called a *variational lower bound*. Here the term evidence refers to the (log) likelihood function, which is sometimes called the 'model evidence' in a Bayesian setting as it allows different models to be compared without the use of hold-out data (Bishop, 2006). As an illustration of this bound, we use it to re-derive the EM algorithm for Gaussian mixtures from a third perspective. The ELBO will play an important role in several of the deep generative models discussed in later chapters. It also provides an example of a *variational* framework in which we introduce a distribution $q(\mathbf{Z})$ over the latent variables and then optimize with respect to this distribution using the calculus of variations.

Appendix B

Consider a probabilistic model in which we collectively denote all the observed variables by $\mathbf{X}$ and all the hidden variables by $\mathbf{Z}$. The joint distribution $p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\theta})$ is governed by a set of parameters denoted by $\boldsymbol{\theta}$. Our goal is to maximize the likelihood function:

$$p(\mathbf{X}|\boldsymbol{\theta}) = \sum_{\mathbf{Z}} p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\theta}). \tag{15.51}$$

Here we are assuming that $\mathbf{Z}$ is discrete, although the discussion is identical if $\mathbf{Z}$ comprises continuous variables or a combination of discrete and continuous variables, with summation replaced by integration as appropriate.

We will suppose that direct optimization of $p(\mathbf{X}|\boldsymbol{\theta})$ is difficult, but that optimization of the complete-data likelihood function $p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\theta})$ is significantly easier. Next we introduce a distribution $q(\mathbf{Z})$ defined over the latent variables, and we observe that, for any choice of $q(\mathbf{Z})$, the following decomposition holds:

$$\ln p(\mathbf{X}|\boldsymbol{\theta}) = \mathcal{L}(q, \boldsymbol{\theta}) + \mathrm{KL}(q\|p) \tag{15.52}$$

where we have defined

$$\mathcal{L}(q, \boldsymbol{\theta}) = \sum_{\mathbf{Z}} q(\mathbf{Z}) \ln \left\{ \frac{p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\theta})}{q(\mathbf{Z})} \right\} \tag{15.53}$$

$$\mathrm{KL}(q\|p) = -\sum_{\mathbf{Z}} q(\mathbf{Z}) \ln \left\{ \frac{p(\mathbf{Z}|\mathbf{X}, \boldsymbol{\theta})}{q(\mathbf{Z})} \right\}. \tag{15.54}$$

Appendix B

Note that $\mathcal{L}(q, \boldsymbol{\theta})$ is a functional of the distribution $q(\mathbf{Z})$ and a function of the parameters $\boldsymbol{\theta}$. It is worth studying carefully the forms of the expressions (15.53) and (15.54), and in particular noting that they differ in sign and also that $\mathcal{L}(q, \boldsymbol{\theta})$ contains the joint distribution of $\mathbf{X}$ and $\mathbf{Z}$ whereas $\mathrm{KL}(q\|p)$ contains the conditional distribution of $\mathbf{Z}$ given $\mathbf{X}$. To verify the decomposition (15.52), we first make use of the product rule of probability to give

Exercise 15.21

$$\ln p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\theta}) = \ln p(\mathbf{Z}|\mathbf{X}, \boldsymbol{\theta}) + \ln p(\mathbf{X}|\boldsymbol{\theta}), \tag{15.55}$$

Telegram: @uni_k

www.konkur.in

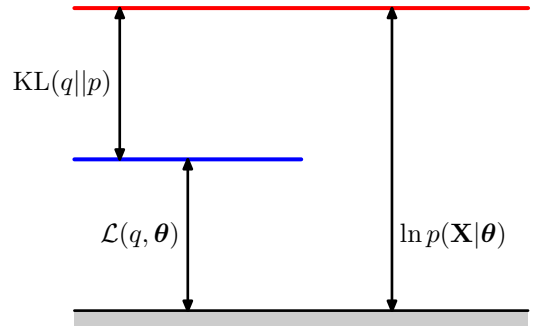


Figure 15.13 Illustration of the decomposition given by (15.52), which holds for any choice of distribution $q(\mathbf{Z})$. Because the Kullback–Leibler divergence satisfies $\mathrm{KL}(q\|p) \geqslant 0$, we see that the quantity $\mathcal{L}(q, \boldsymbol{\theta})$ is a lower bound on the log likelihood function $\ln p(\mathbf{X}|\boldsymbol{\theta})$.

which we then substitute into the expression for $\mathcal{L}(q, \boldsymbol{\theta})$. This gives rise to two terms, one of which cancels $\mathrm{KL}(q\|p)$ whereas the other gives the required log likelihood $\ln p(\mathbf{X}|\boldsymbol{\theta})$ after noting that $q(\mathbf{Z})$ is a normalized distribution that sums to 1.

From (15.54), we see that $\mathrm{KL}(q\|p)$ is the Kullback–Leibler divergence between $q(\mathbf{Z})$ and the posterior distribution $p(\mathbf{Z}|\mathbf{X}, \boldsymbol{\theta})$. Recall that the Kullback–Leibler divergence satisfies $\mathrm{KL}(q\|p) \geqslant 0$, with equality if, and only if, $q(\mathbf{Z}) = p(\mathbf{Z}|\mathbf{X}, \boldsymbol{\theta})$. It therefore follows from (15.52) that $\mathcal{L}(q, \boldsymbol{\theta}) \leqslant \ln p(\mathbf{X}|\boldsymbol{\theta})$, in other words that $\mathcal{L}(q, \boldsymbol{\theta})$ is a lower bound on $\ln p(\mathbf{X}|\boldsymbol{\theta})$. The decomposition (15.52) is illustrated in Figure 15.13.

Section 2.5.7

15.4.1 EM revisited

We can use the decomposition (15.52) to derive the EM algorithm and to demonstrate that it does indeed maximize the log likelihood. Suppose that the current value of the parameter vector is $\boldsymbol{\theta}^{\mathrm{old}}$. In the E step, the lower bound $\mathcal{L}(q, \boldsymbol{\theta}^{\mathrm{old}})$ is maximized with respect to $q(\mathbf{Z})$ while holding $\boldsymbol{\theta}^{\mathrm{old}}$ fixed. The solution to this maximization problem is easily seen by noting that the value of $\ln p(\mathbf{X}|\boldsymbol{\theta}^{\mathrm{old}})$ does not depend on $q(\mathbf{Z})$ and so the largest value of $\mathcal{L}(q, \boldsymbol{\theta}^{\mathrm{old}})$ will occur when the Kullback–Leibler divergence vanishes, in other words when $q(\mathbf{Z})$ is equal to the posterior distribution $p(\mathbf{Z}|\mathbf{X}, \boldsymbol{\theta}^{\mathrm{old}})$. In this case, the lower bound will equal the log likelihood, as illustrated in Figure 15.14.

In the subsequent M step, the distribution $q(\mathbf{Z})$ is held fixed and the lower bound

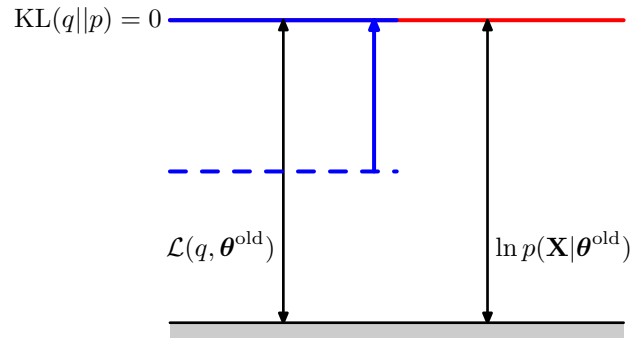


Figure 15.14 Illustration of the E step of the EM algorithm. The q distribution is set equal to the posterior distribution for the current parameter values $\boldsymbol{\theta}^{\mathrm{old}}$, causing the lower bound to move up to the same value as the log likelihood function, with the KL divergence vanishing.

Telegram: @uni_k

www.konkur.in

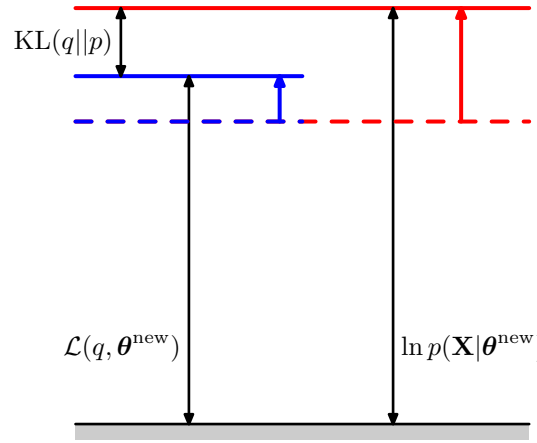


Figure 15.15 Illustration of the M step of the EM algorithm. The distribution $q(\mathbf{Z})$ is held fixed and the lower bound $\mathcal{L}(q, \boldsymbol{\theta})$ is maximized with respect to the parameter vector $\boldsymbol{\theta}$ to give a revised value $\boldsymbol{\theta}^{\text{new}}$. Because the Kullback–Leibler divergence is non-negative, this causes the log likelihood $\ln p(\mathbf{X}|\boldsymbol{\theta})$ to increase by at least as much as the lower bound does.

$\mathcal{L}(q, \boldsymbol{\theta})$ is maximized with respect to $\boldsymbol{\theta}$ to give some new value $\boldsymbol{\theta}^{\text{new}}$. This will cause the lower bound $\mathcal{L}$ to increase (unless it is already at a maximum), which will necessarily cause the corresponding log likelihood function to increase. Because the distribution q is determined using the old parameter values rather than the new values and is held fixed during the M step, it will not equal the new posterior distribution $p(\mathbf{Z}|\mathbf{X}, \boldsymbol{\theta}^{\text{new}})$, and hence there will be a non-zero Kullback–Leibler divergence. The increase in the log likelihood function is therefore greater than the increase in the lower bound, as shown in Figure 15.15. If we substitute $q(\mathbf{Z}) = p(\mathbf{Z}|\mathbf{X}, \boldsymbol{\theta}^{\text{old}})$ into (15.53), we see that, after the E step, the lower bound takes the form

$$\begin{aligned} \mathcal{L}(q, \boldsymbol{\theta}) &= \sum_{\mathbf{Z}} p(\mathbf{Z}|\mathbf{X}, \boldsymbol{\theta}^{\text{old}}) \ln p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\theta}) - \sum_{\mathbf{Z}} p(\mathbf{Z}|\mathbf{X}, \boldsymbol{\theta}^{\text{old}}) \ln p(\mathbf{Z}|\mathbf{X}, \boldsymbol{\theta}^{\text{old}}) \\ &= \mathcal{Q}(\boldsymbol{\theta}, \boldsymbol{\theta}^{\text{old}}) + \text{const} \end{aligned} \tag{15.56}$$

where the constant is simply the negative entropy of the q distribution and is therefore independent of $\boldsymbol{\theta}$. Here we recognize $\mathcal{Q}(\boldsymbol{\theta}, \boldsymbol{\theta}^{\text{old}})$ as the expected complete-data log-likelihood defined by (15.23), and it is therefore the quantity that is being maximized in the M step, as we saw earlier distribution for mixtures of Gaussians. Note that the variable $\boldsymbol{\theta}$ over which we are optimizing appears only inside the logarithm. If the joint distribution $p(\mathbf{Z}, \mathbf{X}|\boldsymbol{\theta})$ is a member of the exponential family or a product of such members, then we see that the logarithm will cancel the exponential and lead to an M step that will be typically much simpler than the maximization of the corresponding incomplete-data log likelihood function $p(\mathbf{X}|\boldsymbol{\theta})$.

Section 15.3

The operation of the EM algorithm can also be viewed in the space of parameters, as illustrated schematically in Figure 15.16. Here the red curve depicts the (incomplete-data) log likelihood function whose value we wish to maximize. We start with some initial parameter value $\boldsymbol{\theta}^{\text{old}}$, and in the first E step we evaluate the posterior distribution over latent variables, which gives rise to a lower bound $\mathcal{L}(q, \boldsymbol{\theta})$ whose value equals the log likelihood at $\boldsymbol{\theta}^{(\text{old})}$, as shown by the blue curve. Note that the bound makes a tangential contact with the log likelihood at $\boldsymbol{\theta}^{(\text{old})}$, so that both

Telegram: @uni_k

www.konkur.in

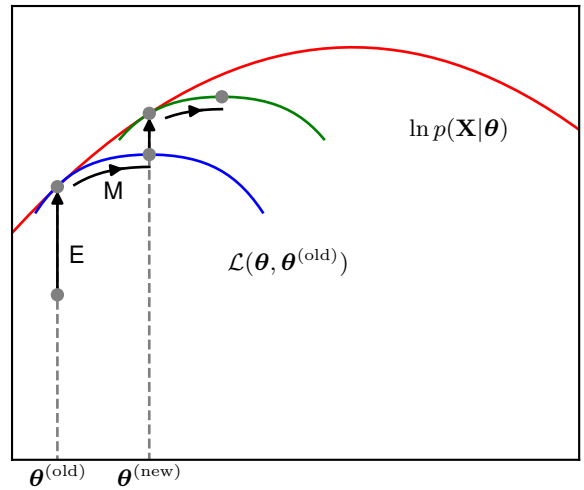


Figure 15.16 The EM algorithm involves alternately computing a lower bound on the log likelihood for the current parameter values and then maximizing this bound to obtain the new parameter values. See the text for a full discussion.

Exercise 15.22

curves have the same gradient. This bound is a convex function having a unique maximum (for mixture components from the exponential family). In the M step, the bound is maximized giving the value $\boldsymbol{\theta}^{(\mathrm{new})}$, which gives a larger value of the log likelihood than $\boldsymbol{\theta}^{(\mathrm{old})}$. The subsequent E step then constructs a bound that is tangential at $\boldsymbol{\theta}^{(\mathrm{new})}$ as shown by the green curve.

We have seen that both the E and the M steps of the EM algorithm increase the value of a well-defined bound on the log likelihood function and that the complete EM cycle will change the model parameters in such a way as to cause the log likelihood to increase (unless it is already at a maximum, in which case the parameters remain unchanged).

15.4.2 Independent and identically distributed data

For the particular case of an i.i.d. data set, $\mathbf{X}$ will comprise N data points $\{\mathbf{x}_n\}$ whereas $\mathbf{Z}$ will comprise N corresponding latent variables $\{\mathbf{z}_n\}$, where $n = 1, \ldots, N$. From the independence assumption, we have $p(\mathbf{X}, \mathbf{Z}) = \prod_n p(\mathbf{x}_n, \mathbf{z}_n)$, and by marginalizing over the $\{\mathbf{z}_n\}$ we have $p(\mathbf{X}) = \prod_n p(\mathbf{x}_n)$. Using the sum and product rules, we see that the posterior probability that is evaluated in the E step takes the form

$$p(\mathbf{Z}|\mathbf{X}, \boldsymbol{\theta}) = \frac{p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\theta})}{\sum_{\mathbf{Z}} p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\theta})} = \frac{\prod_{n=1}^{N} p(\mathbf{x}_n, \mathbf{z}_n|\boldsymbol{\theta})}{\sum_{\mathbf{Z}} \prod_{n=1}^{N} p(\mathbf{x}_n, \mathbf{z}_n|\boldsymbol{\theta})} = \prod_{n=1}^{N} p(\mathbf{z}_n|\mathbf{x}_n, \boldsymbol{\theta}) \quad (15.57)$$

and so the posterior distribution also factorizes with respect to n. For a Gaussian mixture model, this simply says that the responsibility that each of the mixture components takes for a particular data point $\mathbf{x}_n$ depends only on the value of $\mathbf{x}_n$ and

Telegram: @uni_k

www.konkur.in

on the parameters $\boldsymbol{\theta}$ of the mixture components, not on the values of the other data points.

15.4.3 Parameter priors

We can also use the EM algorithm to maximize the posterior distribution $p(\boldsymbol{\theta}|\mathbf{X})$ for models in which we have introduced a prior $p(\boldsymbol{\theta})$ over the parameters. To see this, note that as a function of $\boldsymbol{\theta}$, we have $p(\boldsymbol{\theta}|\mathbf{X}) = p(\boldsymbol{\theta}, \mathbf{X})/p(\mathbf{X})$ and so

$$\ln p(\boldsymbol{\theta}|\mathbf{X}) = \ln p(\boldsymbol{\theta}, \mathbf{X}) - \ln p(\mathbf{X}). \tag{15.58}$$

Making use of the decomposition (15.52), we have

$$\begin{aligned} \ln p(\boldsymbol{\theta}|\mathbf{X}) &= \mathcal{L}(q, \boldsymbol{\theta}) + \mathrm{KL}(q\|p) + \ln p(\boldsymbol{\theta}) - \ln p(\mathbf{X}) \\ &\geqslant \mathcal{L}(q, \boldsymbol{\theta}) + \ln p(\boldsymbol{\theta}) - \ln p(\mathbf{X}) \end{aligned} \tag{15.59}$$

where $\ln p(\mathbf{X})$ is a constant. We can again optimize the right-hand side alternately with respect to q and $\boldsymbol{\theta}$. The optimization with respect to q gives rise to the same E-step equations as for the standard EM algorithm, because q appears only in $\mathcal{L}(q, \boldsymbol{\theta})$. The M-step equations are modified through the introduction of the prior term $\ln p(\boldsymbol{\theta})$, which typically requires only a small modification to the standard maximum likelihood M-step equations. The additional term represents a form of regularization and has the effect of removing the singularities of the likelihood function for Gaussian mixture models.

Chapter 9

15.4.4 Generalized EM

The EM algorithm breaks down the potentially difficult problem of maximizing the likelihood function into two stages, the E step and the M step, each of which will often prove simpler to implement. Nevertheless, for complex models it may be the case that either the E step or the M step, or indeed both, remain intractable. This leads to two possible extensions of the EM algorithm, as follows.

The *generalized EM*, or *GEM*, algorithm addresses the problem of an intractable M step. Instead of aiming to maximize $\mathcal{L}(q, \boldsymbol{\theta})$ with respect to $\boldsymbol{\theta}$, it seeks instead to change the parameters in such a way as to increase its value. Again, because $\mathcal{L}(q, \boldsymbol{\theta})$ is a lower bound on the log likelihood function, each complete EM cycle of the GEM algorithm is guaranteed to increase the value of the log likelihood (unless the parameters already correspond to a local maximum). One way to exploit the GEM approach would be to use gradient-based iterative optimization algorithms during the M step. Another form of GEM algorithm, known as the *expectation conditional maximization* algorithm, involves making several constrained optimizations within each M step (Meng and Rubin, 1993). For instance, the parameters might be partitioned into groups and the M step broken down into multiple steps each of which involves optimizing one of the groups with the remainder held fixed.

We can similarly generalize the E step of the EM algorithm by performing a partial, rather than complete, optimization of $\mathcal{L}(q, \boldsymbol{\theta})$ with respect to $q(\mathbf{Z})$ (Neal and Hinton, 1999). As we have seen, for any given value of $\boldsymbol{\theta}$ there is a unique maximum of $\mathcal{L}(q, \boldsymbol{\theta})$ with respect to $q(\mathbf{Z})$ that corresponds to the posterior distribution $q_{\boldsymbol{\theta}}(\mathbf{Z}) =$

Telegram: @uni_k

www.konkur.in

$p(\mathbf{Z}|\mathbf{X}, \boldsymbol{\theta})$ and that for this choice of $q(\mathbf{Z})$, the bound $\mathcal{L}(q, \boldsymbol{\theta})$ is equal to the log likelihood function $\ln p(\mathbf{X}|\boldsymbol{\theta})$. It follows that any algorithm that converges to the global maximum of $\mathcal{L}(q, \boldsymbol{\theta})$ will find a value of $\boldsymbol{\theta}$ that is also a global maximum of the log likelihood $\ln p(\mathbf{X}|\boldsymbol{\theta})$. Provided $p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\theta})$ is a continuous function of $\boldsymbol{\theta}$ then, by continuity, any local maximum of $\mathcal{L}(q, \boldsymbol{\theta})$ will also be a local maximum of $\ln p(\mathbf{X}|\boldsymbol{\theta})$.

15.4.5 Sequential EM

Consider N independent data points $\mathbf{x}_1, \ldots, \mathbf{x}_N$ with corresponding latent variables $\mathbf{z}_1, \ldots, \mathbf{z}_N$. The joint distribution $p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\theta})$ factorizes over the data points, and this structure can be exploited in an incremental form of EM in which at each EM cycle, only one data point is processed at a time. In the E step, instead of recomputing the responsibilities for all the data points, we just re-evaluate the responsibilities for one data point. It might appear that the subsequent M step would require a computation involving the responsibilities for all the data points. However, if the mixture components are members of the exponential family, then the responsibilities enter only through simple sufficient statistics, and these can be updated efficiently. Consider, for instance, a Gaussian mixture, and suppose we perform an update for data point m in which the corresponding old and new values of the responsibilities are denoted by $\gamma^{\text{old}}(z_{mk})$ and $\gamma^{\text{new}}(z_{mk})$. In the M step, the required sufficient statistics can be updated incrementally. For instance, for the means, the sufficient statistics are defined by (15.16) and (15.17) from which we obtain

Exercise 15.23

$$\boldsymbol{\mu}_k^{\text{new}} = \boldsymbol{\mu}_k^{\text{old}} + \left(\frac{\gamma^{\text{new}}(z_{mk}) - \gamma^{\text{old}}(z_{mk})}{N_k^{\text{new}}} \right) \left(\mathbf{x}_m - \boldsymbol{\mu}_k^{\text{old}} \right) \tag{15.60}$$

together with

$$N_k^{\text{new}} = N_k^{\text{old}} + \gamma^{\text{new}}(z_{mk}) - \gamma^{\text{old}}(z_{mk}). \tag{15.61}$$

The corresponding results for the covariances and the mixing coefficients are analogous.

Thus, both the E step and the M step take a fixed time that is independent of the total number of data points. Because the parameters are revised after each data point, rather than waiting until after the whole data set is processed, this incremental version can converge faster than the batch version. Each E or M step in this incremental algorithm increases the value of $\mathcal{L}(q, \boldsymbol{\theta})$, and as we have shown above, if the algorithm converges to a local (or global) maximum of $\mathcal{L}(q, \boldsymbol{\theta})$, this will correspond to a local (or global) maximum of the log likelihood function $\ln p(\mathbf{X}|\boldsymbol{\theta})$.

Exercises

15.1 (⋆) Consider the K-means algorithm discussed in Section 15.1. Show that as a consequence of there being a finite number of possible assignments for the set of discrete indicator variables r_{nk} and that for each such assignment there is a unique optimum for the $\{\boldsymbol{\mu}_k\}$, the K-means algorithm must converge after a finite number of iterations.

Telegram: @uni_k

www.konkur.in

15.2 ($\star\star$) In this exercise we derive the sequential form for the K-means algorithm. At each step we consider a new data point $\mathbf{x}_n$, and only the prototype vector that is closest to $\mathbf{x}_n$ is updated. Starting from the expression (15.4) for the prototype vectors in the batch setting, separate out the contribution from the final data point $\mathbf{x}_n$. By rearranging the formula, show that this update takes the form (15.5). Note that, since no approximation is made in this derivation, the resulting prototype vectors will have the property that they each equal the mean of all the data vectors that were assigned to them.

15.3 ($\star$) Consider a Gaussian mixture model in which the marginal distribution $p(\mathbf{z})$ for the latent variable is given by (15.9) and the conditional distribution $p(\mathbf{x}|\mathbf{z})$ for the observed variable is given by (15.10). Show that the marginal distribution $p(\mathbf{x})$, obtained by summing $p(\mathbf{z})p(\mathbf{x}|\mathbf{z})$ over all possible values of $\mathbf{z}$, is a Gaussian mixture of the form (15.6).

15.4 ($\star$) Show that the number of equivalent parameter settings due to interchange symmetries in a mixture model with K components is $K!$.

15.5 ($\star\star$) Suppose we wish to use the EM algorithm to maximize the posterior distribution over parameters $p(\boldsymbol{\theta}|\mathbf{X})$ for a model containing latent variables, where $\mathbf{X}$ is the observed data set. Show that the E step remains the same as in the maximum likelihood case, whereas in the M step the quantity to be maximized is given by $\mathcal{Q}(\boldsymbol{\theta}, \boldsymbol{\theta}^{\text{old}}) + \ln p(\boldsymbol{\theta})$ where $\mathcal{Q}(\boldsymbol{\theta}, \boldsymbol{\theta}^{\text{old}})$ is defined by (15.23).

15.6 ($\star$) Consider the directed graph for a Gaussian mixture model shown in Figure 15.9. *Section 11.2* By making use of the d-separation criterion, show that the posterior distribution of the latent variables factorizes with respect to the different data points so that

$$p(\mathbf{Z}|\mathbf{X}, \boldsymbol{\mu}, \boldsymbol{\Sigma}, \boldsymbol{\pi}) = \prod_{n=1}^{N} p(\mathbf{z}_n|\mathbf{x}_n, \boldsymbol{\mu}, \boldsymbol{\Sigma}, \boldsymbol{\pi}). \tag{15.62}$$

15.7 ($\star\star$) Consider a special case of a Gaussian mixture model in which the covariance matrices $\boldsymbol{\Sigma}_k$ of the components are all constrained to have a common value $\boldsymbol{\Sigma}$. Derive the EM equations for maximizing the likelihood function under such a model.

15.8 ($\star\star$) Verify that maximization of the complete-data log likelihood (15.26) for a Gaussian mixture model leads to the result that the means and covariances of each component are fitted independently to the corresponding group of data points and that the mixing coefficients are given by the fractions of points in each group.

15.9 ($\star\star$) Show that if we maximize (15.30) with respect to $\boldsymbol{\mu}_k$ while keeping the responsibilities $\gamma(z_{nk})$ fixed, we obtain the closed-form solution given by (15.16).

15.10 ($\star\star$) Show that if we maximize (15.30) with respect to $\boldsymbol{\Sigma}_k$ and π_k while keeping the responsibilities $\gamma(z_{nk})$ fixed, we obtain the closed-form solutions given by (15.18) and (15.21).

Telegram: @uni_k

www.konkur.in

15.11 (⋆⋆) Consider a density model given by a mixture distribution:

$$p(\mathbf{x}) = \sum_{k=1}^{K} \pi_k p(\mathbf{x}|k) \tag{15.63}$$

and suppose that we partition the vector $\mathbf{x}$ into two parts so that $\mathbf{x} = (\mathbf{x}_a, \mathbf{x}_b)$. Show that the conditional density $p(\mathbf{x}_b|\mathbf{x}_a)$ is itself a mixture distribution, and find expressions for the mixing coefficients and for the component densities.

15.12 (⋆) In Section 15.3.2, we obtained a relationship between K means and EM for Gaussian mixtures by considering a mixture model in which all components have covariance $\epsilon\mathbf{I}$. Show that in the limit $\epsilon \to 0$, maximizing the expected complete-data log likelihood for this model, given by (15.30), is equivalent to minimizing the error measure J for the K-means algorithm given by (15.1).

15.13 (⋆⋆) Verify the results (15.35) and (15.36) for the mean and covariance of the Bernoulli distribution.

15.14 (⋆⋆) Consider a mixture distribution of the form

$$p(\mathbf{x}) = \sum_{k=1}^{K} \pi_k p(\mathbf{x}|k) \tag{15.64}$$

where the elements of $\mathbf{x}$ could be discrete or continuous or a combination of these. Denote the mean and covariance of $p(\mathbf{x}|k)$ by $\boldsymbol{\mu}_k$ and $\boldsymbol{\Sigma}_k$, respectively. By making use of the results of Exercise 15.13, show that the mean and covariance of the mixture distribution are given by (15.39) and (15.40).

15.15 (⋆⋆) Using the re-estimation equations for the EM algorithm, show that a mixture of Bernoulli distributions, with its parameters set to values corresponding to a maximum of the likelihood function, has the property that

$$\mathbb{E}[\mathbf{x}] = \frac{1}{N} \sum_{n=1}^{N} \mathbf{x}_n \equiv \overline{\mathbf{x}}. \tag{15.65}$$

Hence, show that if the parameters of this model are initialized such that all components have the same mean $\boldsymbol{\mu}_k = \widehat{\boldsymbol{\mu}}$ for $k = 1, \ldots, K$, then the EM algorithm will converge after one iteration, for any choice of the initial mixing coefficients, and that this solution has the property $\boldsymbol{\mu}_k = \overline{\mathbf{x}}$. Note that this represents a degenerate case of the mixture model in which all the components are identical, and in practice we try to avoid such solutions by using an appropriate initialization.

15.16 (⋆) Consider the joint distribution of latent and observed variables for the Bernoulli distribution obtained by forming the product of $p(\mathbf{x}|\mathbf{z}, \boldsymbol{\mu})$ given by (15.42) and $p(\mathbf{z}|\boldsymbol{\pi})$ given by (15.43). Show that if we marginalize this joint distribution with respect to $\mathbf{z}$, then we obtain (15.37).

Telegram: @uni_k

www.konkur.in

15.17 ($\star$) Show that if we maximize the expected complete-data log likelihood function (15.45) for a mixture of Bernoulli distributions with respect to $\boldsymbol{\mu}_k$, we obtain the M-step equation (15.49).

15.18 ($\star$) Show that if we maximize the expected complete-data log likelihood function (15.45) for a mixture of Bernoulli distributions with respect to the mixing coefficients π_k, and use a Lagrange multiplier to enforce the summation constraint, we obtain the M-step equation (15.50).

15.19 ($\star$) Show that as a consequence of the constraint $0 \leqslant p(\mathbf{x}_n|\boldsymbol{\mu}_k) \leqslant 1$ for the discrete variable $\mathbf{x}_n$, the incomplete-data log likelihood function for a mixture of Bernoulli distributions is bounded above and hence that there are no singularities for which the likelihood goes to infinity.

15.20 ($\star\star\star$) Consider a D-dimensional variable $\mathbf{x}$ each of whose components i is itself a multinomial variable of degree M so that $\mathbf{x}$ is a binary vector with components x_{ij} where $i = 1, \ldots, D$ and $j = 1, \ldots, M$, subject to the constraint that $\sum_j x_{ij} = 1$ for all i. Suppose that the distribution of these variables is described by a mixture of the discrete multinomial distributions so that

Section 3.1.3

$$p(\mathbf{x}) = \sum_{k=1}^{K} \pi_k p(\mathbf{x}|\boldsymbol{\mu}_k) \tag{15.66}$$

where

$$p(\mathbf{x}|\boldsymbol{\mu}_k) = \prod_{i=1}^{D} \prod_{j=1}^{M} \mu_{kij}^{x_{ij}}. \tag{15.67}$$

The parameters μ_{kij} represent the probabilities $p(x_{ij} = 1|\boldsymbol{\mu}_k)$ and must satisfy $0 \leqslant \mu_{kij} \leqslant 1$ together with the constraint $\sum_j \mu_{kij} = 1$ for all values of k and i. Given an observed data set $\{\mathbf{x}_n\}$, where $n = 1, \ldots, N$, derive the E-step and M-step equations of the EM algorithm for optimizing the mixing coefficients π_k and the component parameters μ_{kij} of this distribution by maximum likelihood.

15.21 ($\star$) Verify the relation (15.52) in which $\mathcal{L}(q, \boldsymbol{\theta})$ and $\mathrm{KL}(q\|p)$ are defined by (15.53) and (15.54), respectively.

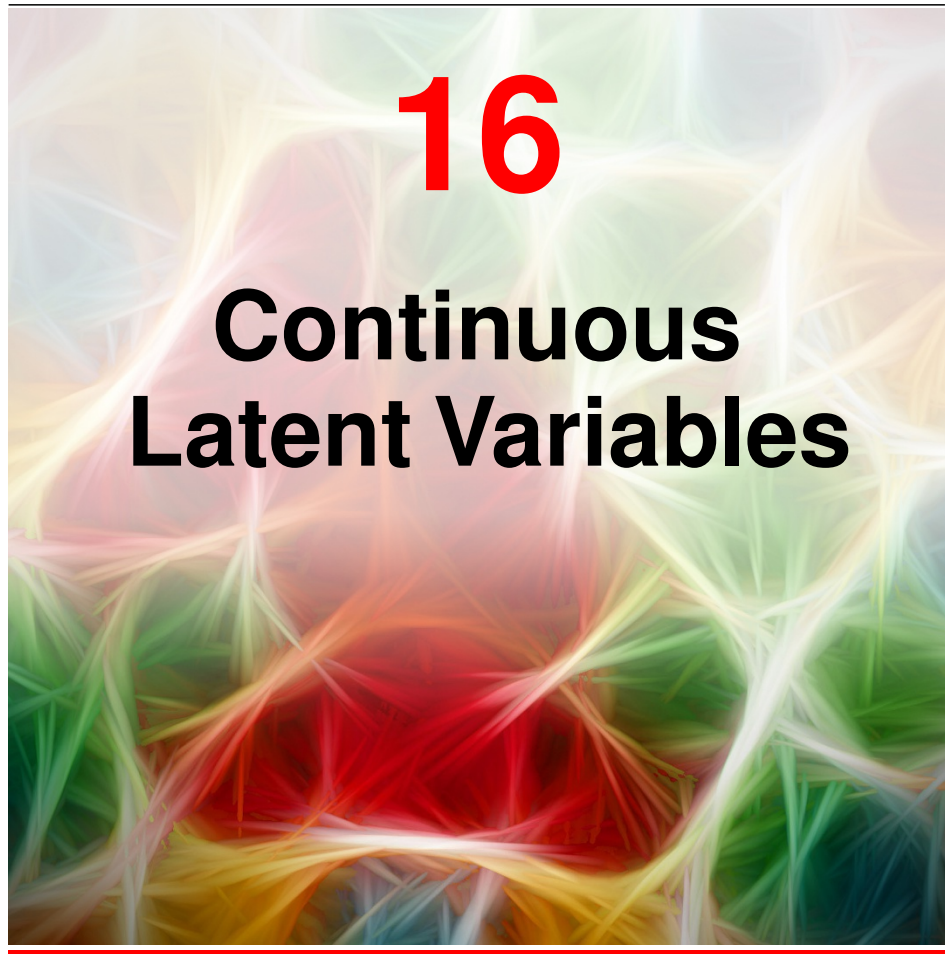
15.22 ($\star$) Show that the lower bound $\mathcal{L}(q, \boldsymbol{\theta})$ given by (15.53), with $q(\mathbf{Z}) = p(\mathbf{Z}|\mathbf{X}, \boldsymbol{\theta}^{(\mathrm{old})})$, has the same gradient with respect to $\boldsymbol{\theta}$ as the log likelihood function $\ln p(\mathbf{X}|\boldsymbol{\theta})$ at the point $\boldsymbol{\theta} = \boldsymbol{\theta}^{(\mathrm{old})}$.

15.23 ($\star\star$) Consider the incremental form of the EM algorithm for a mixture of Gaussians, in which the responsibilities are recomputed only for a specific data point $\mathbf{x}_m$. Starting from the M-step formulae (15.16) and (15.17), derive the results (15.60) and (15.61) for updating the component means.

15.24 ($\star\star$) Derive M-step formulae for updating the covariance matrices and mixing coefficients in a Gaussian mixture model when the responsibilities are updated incrementally, analogous to the result (15.60) for updating the means.

Telegram: @uni_k

www.konkur.in



16 Continuous Latent Variables

In the previous chapter we discussed probabilistic models having discrete latent variables, such as a mixture of Gaussians. We now explore models in which some, or all, of the latent variables are continuous. An important motivation for such models is that many data sets have the property that the data points lie close to a manifold of much lower dimensionality than that of the original data space. To see why this might arise, consider an artificial data set constructed by taking a handwritten digit from the MNIST data set (LeCun *et al.*, 1998), represented by a 64×64 pixel grey-level image, and embedding it in a larger image of size 100×100 by padding with pixels having the value zero (corresponding to white pixels) in which the location and orientation of the digit are varied at random, as illustrated in Figure 16.1. Each of the resulting images is represented by a point in the $100 \times 100 = 10{,}000$-dimensional data space. However, across a data set of such images, there are only three *degrees of freedom* of variability, corresponding to vertical and horizontal translations and

Section 6.1.3

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024
C. M. Bishop, H. Bishop, *Deep Learning*, https://doi.org/10.1007/978-3-031-45468-4_16

Telegram: @uni_k

www.konkur.in

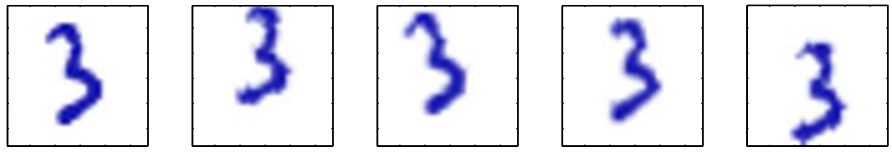


Figure 16.1 A synthetic data set obtained by taking an image of a handwritten digit and creating multiple copies in each of which the digit has undergone a random displacement and rotation within some larger image field. The resulting images each have $100 \times 100 = 10{,}000$ pixels.

rotations. The data points will therefore live on a subspace of the data space whose *intrinsic dimensionality* is three. Note that the manifold will be nonlinear because, for instance, if we translate the digit past a particular pixel, that pixel value will go from zero (white) to one (black) and back to zero again, which is clearly a nonlinear function of the digit position. In this example, the translation and rotation parameters are latent variables because we observe only the image vectors and are not told which values of the translation or rotation variables were used to create them.

For real data sets of handwritten digits, there will be further degrees of freedom arising from scaling and other variations due, for example, to the variability in an individual's writing as well as the differences in writing styles between individuals. Nevertheless, the number of such degrees of freedom will be small compared to the dimensionality of the data set.

In practice, the data points will not be confined precisely to a smooth low-dimensional manifold, and we can interpret the departures of data points from the manifold as 'noise'. This leads naturally to a generative view of such models in which we first select a point within the manifold according to some latent-variable distribution and then generate an observed data point by adding noise drawn from some conditional distribution of the data variables given the latent variables.

The simplest continuous latent-variable model assumes Gaussian distributions for both the latent and observed variables and makes use of a linear-Gaussian dependence of the observed variables on the state of the latent variables. This leads to a probabilistic formulation of the well-known technique of principal component analysis (PCA) as well as to a related model called factor analysis. In this chapter we will begin with a standard, non-probabilistic treatment of PCA, and then we show how PCA arises naturally as the maximum likelihood solution for a linear-Gaussian latent-variable model. This probabilistic reformulation brings many advantages, such as the use of EM for parameter estimation, principled extensions to mixtures of PCA models, and Bayesian formulations that allow the number of principal components to be determined automatically from the data (Bishop, 2006). This chapter also lays the foundations for nonlinear models having continuous latent variables including normalizing flows, variational autoencoders, and diffusion models.

Section 11.1.4

Section 16.1

Section 16.2

Telegram: @uni_k

www.konkur.in

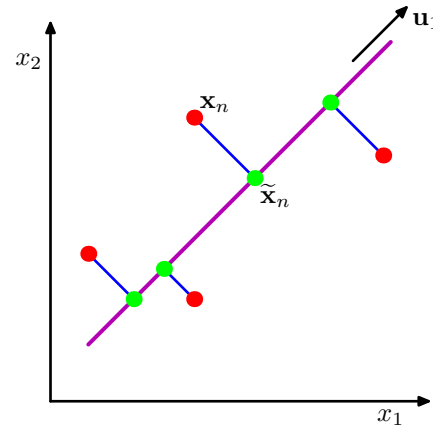


Figure 16.2 Principal component analysis seeks a space of lower dimensionality, known as the principal subspace and denoted by the magenta line, such that the orthogonal projection of the data points (red dots) onto this subspace maximizes the variance of the projected points (green dots). An alternative definition of PCA is based on minimizing the sum-of-squares of the projection errors, indicated by the blue lines.

16.1. Principal Component Analysis

Principal component analysis, or PCA, is widely used for applications such as dimensionality reduction, lossy data compression, feature extraction, and data visualization (Jolliffe, 2002). It is also known as the *Kosambi–Karhunen–Loève* transform.

Consider the orthogonal projection of a data set onto a lower-dimensional linear space, known as the *principal subspace*, as shown in Figure 16.2. PCA can be defined as the linear projection that maximizes the variance of the projected data (Hotelling, 1933). Equivalently, it can be defined as the linear projection that minimizes the average projection cost, defined as the mean squared distance between the data points and their projections (Pearson, 1901). We consider each of these definitions in turn.

16.1.1 Maximum variance formulation

Consider a data set of observations $\{\mathbf{x}_n\}$ where $n = 1, \ldots, N$, and $\mathbf{x}_n$ is a Euclidean variable with dimensionality D. Our goal is to project the data onto a space having dimensionality $M < D$ while maximizing the variance of the projected data. For the moment, we will assume that the value of M is given. Later in this chapter, we will consider techniques to determine an appropriate value of M from the data.

To begin with, consider the projection onto a one-dimensional space ($M = 1$). We can define the direction of this space using a D-dimensional vector $\mathbf{u}_1$, which for convenience (and without loss of generality) we will choose to be a unit vector so that $\mathbf{u}_1^{\mathrm{T}}\mathbf{u}_1 = 1$ (note that we are interested only in the direction defined by $\mathbf{u}_1$, not in the magnitude of $\mathbf{u}_1$ itself). Each data point $\mathbf{x}_n$ is then projected onto a scalar value $\mathbf{u}_1^{\mathrm{T}}\mathbf{x}_n$. The mean of the projected data is $\mathbf{u}_1^{\mathrm{T}}\overline{\mathbf{x}}$ where $\overline{\mathbf{x}}$ is the sample set mean given by

$$\overline{\mathbf{x}} = \frac{1}{N}\sum_{n=1}^{N}\mathbf{x}_n \tag{16.1}$$

Telegram: @uni_k

www.konkur.in

and the variance of the projected data is given by

$$\frac{1}{N}\sum_{n=1}^{N}\left\{\mathbf{u}_1^{\mathrm{T}}\mathbf{x}_n - \mathbf{u}_1^{\mathrm{T}}\overline{\mathbf{x}}\right\}^2 = \mathbf{u}_1^{\mathrm{T}}\mathbf{S}\mathbf{u}_1 \tag{16.2}$$

where $\mathbf{S}$ is the data covariance matrix defined by

$$\mathbf{S} = \frac{1}{N}\sum_{n=1}^{N}(\mathbf{x}_n - \overline{\mathbf{x}})(\mathbf{x}_n - \overline{\mathbf{x}})^{\mathrm{T}}. \tag{16.3}$$

We now maximize the projected variance $\mathbf{u}_1^{\mathrm{T}}\mathbf{S}\mathbf{u}_1$ with respect to $\mathbf{u}_1$. Clearly, this has to be a constrained maximization to prevent $\|\mathbf{u}_1\| \to \infty$. The appropriate constraint comes from the normalization condition $\mathbf{u}_1^{\mathrm{T}}\mathbf{u}_1 = 1$. To enforce this constraint, we introduce a Lagrange multiplier that we will denote by λ_1, and then make an unconstrained maximization of

Appendix C

$$\mathbf{u}_1^{\mathrm{T}}\mathbf{S}\mathbf{u}_1 + \lambda_1\left(1 - \mathbf{u}_1^{\mathrm{T}}\mathbf{u}_1\right). \tag{16.4}$$

By setting the derivative with respect to $\mathbf{u}_1$ equal to zero, we see that this quantity will have a stationary point when

$$\mathbf{S}\mathbf{u}_1 = \lambda_1\mathbf{u}_1, \tag{16.5}$$

which says that $\mathbf{u}_1$ must be an eigenvector of $\mathbf{S}$. If we left-multiply by $\mathbf{u}_1^{\mathrm{T}}$ and make use of $\mathbf{u}_1^{\mathrm{T}}\mathbf{u}_1 = 1$, we see that the variance is given by

$$\mathbf{u}_1^{\mathrm{T}}\mathbf{S}\mathbf{u}_1 = \lambda_1 \tag{16.6}$$

and so the variance will be a maximum when we set $\mathbf{u}_1$ equal to the eigenvector having the largest eigenvalue λ_1. This eigenvector is known as the first principal component.

We can define additional principal components in an incremental fashion by choosing each new direction to be that which maximizes the projected variance amongst all possible directions orthogonal to those already considered. If we consider the general case of an M-dimensional projection space, the optimal linear projection for which the variance of the projected data is maximized is now defined by the M eigenvectors $\mathbf{u}_1, \ldots, \mathbf{u}_M$ of the data covariance matrix $\mathbf{S}$ corresponding to the M largest eigenvalues $\lambda_1, \ldots, \lambda_M$. This is easily shown using proof by induction.

Exercise 16.1

To summarize, PCA involves evaluating the mean $\overline{\mathbf{x}}$ and the covariance matrix $\mathbf{S}$ of a data set and then finding the M eigenvectors of $\mathbf{S}$ corresponding to the M largest eigenvalues. Algorithms for finding eigenvectors and eigenvalues, as well as additional theorems related to eigenvector decomposition, can be found in Golub and Van Loan (1996). Note that the computational cost of computing the full eigenvector decomposition for a matrix of size $D \times D$ is $\mathcal{O}(D^3)$. If we plan to project our data onto the first M principal components, then we only need to find the first M eigenvalues and eigenvectors. This can be done with more efficient techniques, such as the *power method* (Golub and Van Loan, 1996), that scale like $\mathcal{O}(MD^2)$, or alternatively we can make use of the EM algorithm.

Section 16.3.2

Telegram: @uni_k

www.konkur.in

16.1.2 Minimum-error formulation

Appendix A

We now discuss an alternative formulation of PCA based on projection error minimization. To do this, we introduce a complete orthonormal set of D-dimensional basis vectors $\{\mathbf{u}_i\}$ where $i = 1, \ldots, D$ that satisfy

$$\mathbf{u}_i^{\mathrm{T}}\mathbf{u}_j = \delta_{ij}. \tag{16.7}$$

Because this basis is complete, each data point can be represented exactly by a linear combination of the basis vectors

$$\mathbf{x}_n = \sum_{i=1}^{D} \alpha_{ni}\mathbf{u}_i \tag{16.8}$$

where the coefficients α_{ni} will be different for different data points. This simply corresponds to a rotation of the coordinate system to a new system defined by the $\{\mathbf{u}_i\}$, and the original D components $\{x_{n1}, \ldots, x_{nD}\}$ are replaced by an equivalent set $\{\alpha_{n1}, \ldots, \alpha_{nD}\}$. Taking the inner product with $\mathbf{u}_j$, and making use of the orthonormality property, we obtain $\alpha_{nj} = \mathbf{x}_n^{\mathrm{T}}\mathbf{u}_j$, and so without loss of generality we can write

$$\mathbf{x}_n = \sum_{i=1}^{D} \left(\mathbf{x}_n^{\mathrm{T}}\mathbf{u}_i\right)\mathbf{u}_i. \tag{16.9}$$

Our goal, however, is to approximate this data point using a representation involving a restricted number $M < D$ of variables corresponding to a projection onto a lower-dimensional subspace. The M-dimensional linear subspace can be represented, without loss of generality, by the first M of the basis vectors, and so we approximate each data point $\mathbf{x}_n$ by

$$\widetilde{\mathbf{x}}_n = \sum_{i=1}^{M} z_{ni}\mathbf{u}_i + \sum_{i=M+1}^{D} b_i\mathbf{u}_i \tag{16.10}$$

where the $\{z_{ni}\}$ depend on the particular data point, whereas the $\{b_i\}$ are constants that are the same for all data points. We are free to choose the $\{\mathbf{u}_i\}$, the $\{z_{ni}\}$, and the $\{b_i\}$ so as to minimize the error introduced by the reduction in dimensionality. As our error measure, we will use the squared distance between the original data point $\mathbf{x}_n$ and its approximation $\widetilde{\mathbf{x}}_n$, averaged over the data set, so that our goal is to minimize

$$J = \frac{1}{N}\sum_{n=1}^{N} \|\mathbf{x}_n - \widetilde{\mathbf{x}}_n\|^2. \tag{16.11}$$

Consider first the minimization with respect to the quantities $\{z_{ni}\}$. Substituting for $\widetilde{\mathbf{x}}_n$, setting the derivative with respect to z_{nj} to zero, and making use of the orthonormality conditions, we obtain

$$z_{nj} = \mathbf{x}_n^{\mathrm{T}}\mathbf{u}_j \tag{16.12}$$

Telegram: @uni_k

www.konkur.in

where $j = 1, \ldots, M$. Similarly, setting the derivative of J with respect to b_i to zero and again making use of the orthonormality relations, gives

$$b_j = \overline{\mathbf{x}}^{\mathrm{T}} \mathbf{u}_j \tag{16.13}$$

where $j = M+1, \ldots, D$. If we substitute for z_{ni} and b_i and make use of the general expansion (16.9), we obtain

$$\mathbf{x}_n - \widetilde{\mathbf{x}}_n = \sum_{i=M+1}^{D} \left\{ (\mathbf{x}_n - \overline{\mathbf{x}})^{\mathrm{T}} \mathbf{u}_i \right\} \mathbf{u}_i \tag{16.14}$$

from which we see that the displacement vector from $\mathbf{x}_n$ to $\widetilde{\mathbf{x}}_n$ lies in the space orthogonal to the principal subspace, because it is a linear combination of $\{\mathbf{u}_i\}$ for $i = M+1, \ldots, D$, as illustrated in Figure 16.2. This is to be expected because the projected points $\widetilde{\mathbf{x}}_n$ must lie within the principal subspace, but we can move them freely within that subspace, and so the minimum error is given by the orthogonal projection.

We therefore obtain an expression for the error measure J as a function purely of the $\{\mathbf{u}_i\}$ in the form

$$J = \frac{1}{N} \sum_{n=1}^{N} \sum_{i=M+1}^{D} \left(\mathbf{x}_n^{\mathrm{T}} \mathbf{u}_i - \overline{\mathbf{x}}^{\mathrm{T}} \mathbf{u}_i \right)^2 = \sum_{i=M+1}^{D} \mathbf{u}_i^{\mathrm{T}} \mathbf{S} \mathbf{u}_i. \tag{16.15}$$

There remains the task of minimizing J with respect to the $\{\mathbf{u}_i\}$, which must be a constrained minimization otherwise we will obtain the vacuous result $\mathbf{u}_i = 0$. The constraints arise from the orthonormality conditions, and as we will see, the solution will be expressed in terms of the eigenvector expansion of the covariance matrix. Before considering a formal solution, let us try to obtain some intuition about the result by considering a two-dimensional data space $D = 2$ and a one-dimensional principal subspace $M = 1$. We have to choose a direction $\mathbf{u}_2$ so as to minimize $J = \mathbf{u}_2^{\mathrm{T}} \mathbf{S} \mathbf{u}_2$, subject to the normalization constraint $\mathbf{u}_2^{\mathrm{T}} \mathbf{u}_2 = 1$. Using a Lagrange multiplier λ_2 to enforce the constraint, we consider the minimization of

$$\widetilde{J} = \mathbf{u}_2^{\mathrm{T}} \mathbf{S} \mathbf{u}_2 + \lambda_2 \left(1 - \mathbf{u}_2^{\mathrm{T}} \mathbf{u}_2 \right). \tag{16.16}$$

Setting the derivative with respect to $\mathbf{u}_2$ to zero, we obtain $\mathbf{S}\mathbf{u}_2 = \lambda_2 \mathbf{u}_2$ so that $\mathbf{u}_2$ is an eigenvector of $\mathbf{S}$ with eigenvalue λ_2. Thus, any eigenvector will define a stationary point of the error measure. To find the value of J at the minimum, we back-substitute the solution for $\mathbf{u}_2$ into the error measure to give $J = \lambda_2$. We therefore obtain the minimum value of J by choosing $\mathbf{u}_2$ to be the eigenvector corresponding to the smaller of the two eigenvalues. Thus, we should choose the principal subspace to be aligned with the eigenvector having the *larger* eigenvalue. This result accords with our intuition that, to minimize the average squared projection distance, we should choose the principal component subspace so that it passes through the mean of the data points and is aligned with the directions of maximum variance. If

Telegram: @uni_k

www.konkur.in

the eigenvalues are equal, any choice of principal direction will give rise to the same value of J.

Exercise 16.2

The general solution to the minimization of J for arbitrary D and arbitrary $M < D$ is obtained by choosing the $\{\mathbf{u}_i\}$ to be eigenvectors of the covariance matrix given by

$$\mathbf{S}\mathbf{u}_i = \lambda_i \mathbf{u}_i \tag{16.17}$$

where $i = 1, \ldots, D$, and as usual the eigenvectors $\{\mathbf{u}_i\}$ are chosen to be orthonormal. The corresponding value of the error measure is then given by

$$J = \sum_{i=M+1}^{D} \lambda_i, \tag{16.18}$$

which is simply the sum of the eigenvalues of those eigenvectors that are orthogonal to the principal subspace. We therefore obtain the minimum value of J by selecting these eigenvectors to be those having the $D - M$ smallest eigenvalues, and hence the eigenvectors defining the principal subspace are those corresponding to the M largest eigenvalues.

Although we have considered $M < D$, the PCA analysis still holds if $M = D$, in which case there is no dimensionality reduction but simply a rotation of the coordinate axes to align with the principal components.

Finally, note that there is a related linear dimensionality reduction technique called *canonical correlation analysis* (Hotelling, 1936; Bach and Jordan, 2002). Whereas PCA works with a single random variable, canonical correlation analysis considers two (or more) variables and tries to find a corresponding pair of linear subspaces that have high cross-correlation, so that each component within one of the subspaces is correlated with a single component from the other subspace. Its solution can be expressed in terms of a generalized eigenvector problem.

16.1.3 Data compression

One application for PCA is data compression, and we can illustrate this by considering a data set of images of handwritten digits. Because each eigenvector of the covariance matrix is a vector in the original D-dimensional space, we can represent the eigenvectors as images of the same size as the data points. The mean image and the first four eigenvectors, along with their corresponding eigenvalues, are shown in Figure 16.3.

A plot of the complete spectrum of eigenvalues, sorted into decreasing order, is shown in Figure 16.4(a). The error measure J associated with choosing a particular value of M is given by the sum of the eigenvalues from $M+1$ up to D and is plotted for different values of M in Figure 16.4(b).

If we substitute (16.12) and (16.13) into (16.10), we can write the PCA approx-

Telegram: @uni_k

www.konkur.in

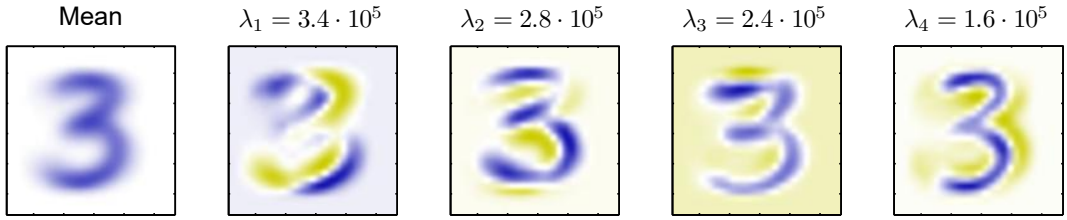


Figure 16.3 Illustration of PCA applied to a data set of 6,000 images of size 28×28, each comprising a hand-written image of the numeral '3' , showing the mean vector $\overline{\mathbf{x}}$ along with the first four PCA eigenvectors $\mathbf{u}_1, \ldots, \mathbf{u}_4$, together with their corresponding eigenvalues.

imation to a data vector $\mathbf{x}_n$ in the form

$$\widetilde{\mathbf{x}}_n = \sum_{i=1}^{M} (\mathbf{x}_n^{\mathrm{T}}\mathbf{u}_i)\mathbf{u}_i + \sum_{i=M+1}^{D} (\overline{\mathbf{x}}^{\mathrm{T}}\mathbf{u}_i)\mathbf{u}_i \tag{16.19}$$

$$= \overline{\mathbf{x}} + \sum_{i=1}^{M} \left(\mathbf{x}_n^{\mathrm{T}}\mathbf{u}_i - \overline{\mathbf{x}}^{\mathrm{T}}\mathbf{u}_i\right)\mathbf{u}_i \tag{16.20}$$

where we have made use of the relation

$$\overline{\mathbf{x}} = \sum_{i=1}^{D} \left(\overline{\mathbf{x}}^{\mathrm{T}}\mathbf{u}_i\right)\mathbf{u}_i, \tag{16.21}$$

which follows from the completeness of the $\{\mathbf{u}_i\}$. This represents a compression of the data set, because for each data point we have replaced the D-dimensional vector $\mathbf{x}_n$ with an M-dimensional vector having components $\left(\mathbf{x}_n^{\mathrm{T}}\mathbf{u}_i - \overline{\mathbf{x}}^{\mathrm{T}}\mathbf{u}_i\right)$. The smaller the value of M, the greater the degree of compression. Examples of PCA reconstructions of data points for the digits data set are shown in Figure 16.5.

16.1.4 Data whitening

Another application of PCA is to data pre-processing. In this case, the goal is not dimensionality reduction but rather the transformation of a data set to standardize certain of its properties. This can be important in allowing subsequent machine learning algorithms to be applied successfully to the data set. Typically, it is done when the original variables are measured in various different units or have significantly different variabilities. For instance in the Old Faithful data set, the time between eruptions is typically an order of magnitude greater than the duration of an eruption. When we applied the K-means algorithm to this data set, we first made a separate linear re-scaling of the individual variables such that each variable had zero

Section 15.1

Telegram: @uni_k

www.konkur.in

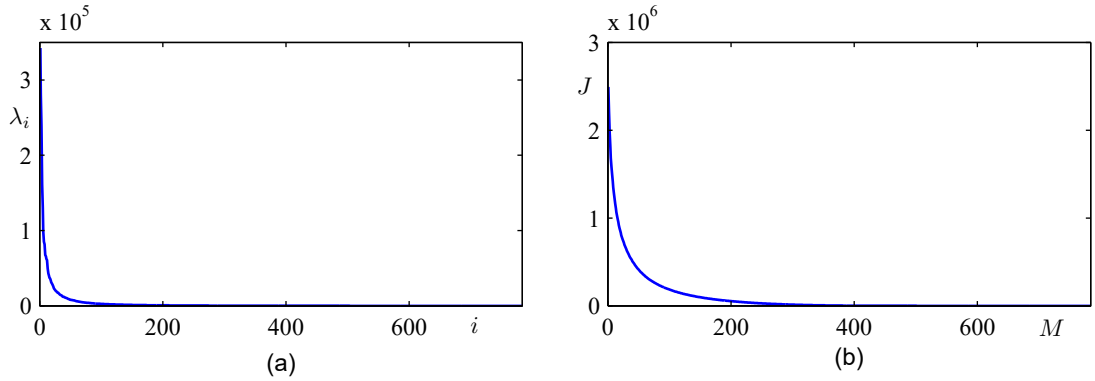


Figure 16.4 (a) Plot of the eigenvalue spectrum for the data set of handwritten digits used in Figure 16.3. (b) Plot of the sum of the discarded eigenvalues, which represents the sum-of-squares error J introduced by projecting the data onto a principal component subspace of dimensionality M.

mean and unit variance. This is known as *standardizing* the data, and the covariance matrix for the standardized data has components

$$\rho_{ij} = \frac{1}{N}\sum_{n=1}^{N}\frac{(x_{ni}-\overline{x}_i)}{\sigma_i}\frac{(x_{nj}-\overline{x}_j)}{\sigma_j} \tag{16.22}$$

where σ_i is the standard deviation of x_i. This is known as the *correlation* matrix of the original data and has the property that if two components x_i and x_j of the data are perfectly correlated, then $\rho_{ij} = 1$, and if they are uncorrelated, then $\rho_{ij} = 0$.

However, using PCA we can make a more substantial normalization of the data to give it zero mean and unit covariance, so that different variables become decorre-

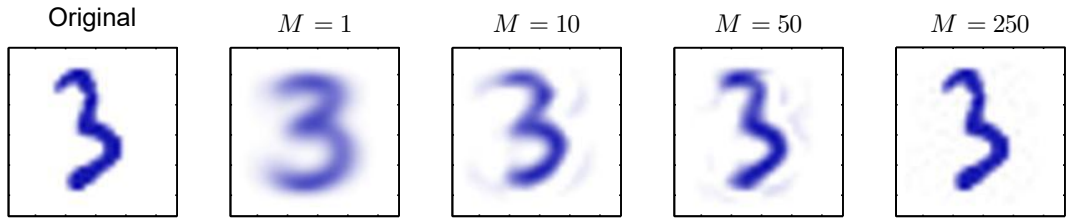


Figure 16.5 An example from the data set of handwritten digits together with its PCA reconstructions obtained by retaining M principal components for various values of M. As M increases, the reconstruction becomes more accurate and would become perfect when $M = D = 28 \times 28 = 784$.

Telegram: @uni_k

www.konkur.in

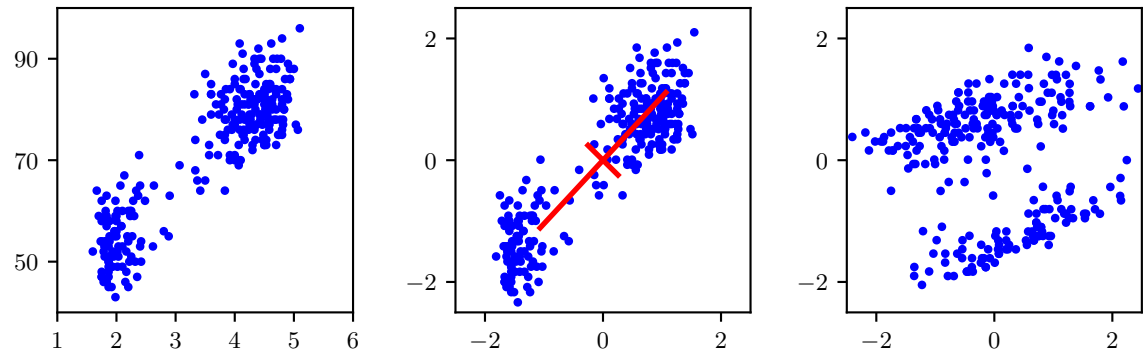


Figure 16.6 Illustration of the effects of linear pre-processing applied to the Old Faithful data set. The plot on the left shows the original data. The centre plot shows the result of standardizing the individual variables to zero mean and unit variance. Also shown are the principal axes of this normalized data set, plotted over the range $\pm\lambda_i^{1/2}$. The plot on the right shows the result of whitening the data to give it zero mean and unit covariance.

lated. To do this, we first write the eigenvector equation (16.17) in the form

$$\mathbf{SU} = \mathbf{UL} \tag{16.23}$$

where $\mathbf{L}$ is a $D \times D$ diagonal matrix with elements λ_i, and $\mathbf{U}$ is a $D \times D$ orthogonal matrix with columns given by $\mathbf{u}_i$. Then we define, for each data point $\mathbf{x}_n$, a transformed value given by

$$\mathbf{y}_n = \mathbf{L}^{-1/2}\mathbf{U}^{\mathrm{T}}(\mathbf{x}_n - \overline{\mathbf{x}}) \tag{16.24}$$

where $\overline{\mathbf{x}}$ is the sample mean defined by (16.1). Clearly, the set $\{\mathbf{y}_n\}$ has zero mean, and its covariance is given by the identity matrix because

$$\begin{aligned} \frac{1}{N}\sum_{n=1}^{N}\mathbf{y}_n\mathbf{y}_n^{\mathrm{T}} &= \frac{1}{N}\sum_{n=1}^{N}\mathbf{L}^{-1/2}\mathbf{U}^{\mathrm{T}}(\mathbf{x}_n - \overline{\mathbf{x}})(\mathbf{x}_n - \overline{\mathbf{x}})^{\mathrm{T}}\mathbf{U}\mathbf{L}^{-1/2} \\ &= \mathbf{L}^{-1/2}\mathbf{U}^{\mathrm{T}}\mathbf{S}\mathbf{U}\mathbf{L}^{-1/2} = \mathbf{L}^{-1/2}\mathbf{L}\mathbf{L}^{-1/2} = \mathbf{I}. \end{aligned} \tag{16.25}$$

This operation is known as *whitening* or *sphering* the data and is illustrated for the Old Faithful data set in Figure 16.6.

Section 15.1

16.1.5 High-dimensional data

In some applications of PCA, the number of data points is smaller than the dimensionality of the data space. For example, we might want to apply PCA to a data set of a few hundred images, each of which corresponds to a vector in a space of potentially several million dimensions (corresponding to three colour values for each of the pixels in the image). Note that in a D-dimensional space, a set of N points, where $N < D$, defines a linear subspace whose dimensionality is at most $N-1$, and so there is little point in applying PCA for values of M that are greater than $N-1$. Indeed, if we perform PCA we will find that at least $D-N+1$ of the eigenvalues are

Telegram: @uni_k

www.konkur.in

zero, corresponding to eigenvectors along whose directions the data set has zero variance. Furthermore, typical algorithms for finding the eigenvectors of a $D \times D$ matrix have a computational cost that scales like $\mathcal{O}(D^3)$, and so for applications such as the image example, a direct application of PCA will be computationally infeasible.

We can resolve this problem as follows. First, let us define $\mathbf{X}$ to be the $(N \times D)$-dimensional centred data matrix, whose nth row is given by $(\mathbf{x}_n - \overline{\mathbf{x}})^{\mathrm{T}}$. The covariance matrix (16.3) can then be written as $\mathbf{S} = N^{-1}\mathbf{X}^{\mathrm{T}}\mathbf{X}$, and the corresponding eigenvector equation becomes

$$\frac{1}{N}\mathbf{X}^{\mathrm{T}}\mathbf{X}\mathbf{u}_i = \lambda_i \mathbf{u}_i. \tag{16.26}$$

Now pre-multiply both sides by $\mathbf{X}$ to give

$$\frac{1}{N}\mathbf{X}\mathbf{X}^{\mathrm{T}}(\mathbf{X}\mathbf{u}_i) = \lambda_i(\mathbf{X}\mathbf{u}_i). \tag{16.27}$$

If we now define $\mathbf{v}_i = \mathbf{X}\mathbf{u}_i$, we obtain

$$\frac{1}{N}\mathbf{X}\mathbf{X}^{\mathrm{T}}\mathbf{v}_i = \lambda_i \mathbf{v}_i, \tag{16.28}$$

which is an eigenvector equation for the $N \times N$ matrix $N^{-1}\mathbf{X}\mathbf{X}^{\mathrm{T}}$. We see that this has the same $N-1$ eigenvalues as the original covariance matrix (which itself has an additional $D-N+1$ eigenvalues of value zero). Thus, we can solve the eigenvector problem in spaces of lower dimensionality with computational cost $\mathcal{O}(N^3)$ instead of $\mathcal{O}(D^3)$. To determine the eigenvectors, we multiply both sides of (16.28) by $\mathbf{X}^{\mathrm{T}}$ to give

$$\left(\frac{1}{N}\mathbf{X}^{\mathrm{T}}\mathbf{X}\right)(\mathbf{X}^{\mathrm{T}}\mathbf{v}_i) = \lambda_i(\mathbf{X}^{\mathrm{T}}\mathbf{v}_i) \tag{16.29}$$

from which we see that $(\mathbf{X}^{\mathrm{T}}\mathbf{v}_i)$ is an eigenvector of $\mathbf{S}$ with eigenvalue λ_i. Note, however, that these eigenvectors are not necessarily normalized. To determine the appropriate normalization, we re-scale $\mathbf{u}_i \propto \mathbf{X}^{\mathrm{T}}\mathbf{v}_i$ by a constant such that $\|\mathbf{u}_i\| = 1$, which, assuming $\mathbf{v}_i$ has been normalized to unit length, gives

Exercise 16.3

$$\mathbf{u}_i = \frac{1}{(N\lambda_i)^{1/2}}\mathbf{X}^{\mathrm{T}}\mathbf{v}_i. \tag{16.30}$$

In summary, to apply this approach we first evaluate $\mathbf{X}\mathbf{X}^{\mathrm{T}}$ and then find its eigenvectors and eigenvalues and then compute the eigenvectors in the original data space using (16.30).

Telegram: @uni_k

www.konkur.in

16.2. Probabilistic Latent Variables

We have seen in the previous section that PCA can be defined in terms of a linear projection of the data onto a subspace of lower dimensionality than the original data space. Each data point projects to a unique value of the quantities z_{nj} defined by (16.12), and we can view these quantities as deterministic latent variables. To introduce and motivate probabilistic continuous latent variables, we now show that PCA can also be expressed as the maximum likelihood solution of a probabilistic latent-variable model. This reformulation of PCA, known as *probabilistic PCA*, has several advantages compared with conventional PCA:

- A probabilistic PCA model represents a constrained form of a Gaussian distribution in which the number of free parameters can be restricted while still allowing the model to capture the dominant correlations in a data set.
- We can derive an EM algorithm for PCA that is computationally efficient in situations where only a few leading eigenvectors are required and that avoids having to evaluate the data covariance matrix as an intermediate step.

Section 16.3.2

- The combination of a probabilistic model and EM allows us to deal with missing values in the data set.
- Mixtures of probabilistic PCA models can be formulated in a principled way and trained using the EM algorithm.
- The existence of a likelihood function allows direct comparison with other probabilistic density models. By contrast, conventional PCA will assign a low reconstruction cost to data points that are close to the principal subspace even if they lie arbitrarily far from the training data.
- Probabilistic PCA can be used to model class-conditional densities and hence be applied to classification problems.
- A probabilistic PCA model can be run generatively to provide samples from the distribution.
- Probabilistic PCA forms the basis for a Bayesian treatment of PCA in which the dimensionality of the principal subspace can be found automatically from the data (Bishop, 2006).

This formulation of PCA as a probabilistic model was proposed independently by Tipping and Bishop 1997; 1999 and by Roweis (1998). As we will see later, it is closely related to *factor analysis* (Basilevsky, 1994).

16.2.1 Generative model

Section 11.1.4

Probabilistic PCA is a simple example of the linear-Gaussian framework in which all the marginal and conditional distributions are Gaussian. We can formulate probabilistic PCA by first introducing an explicit M-dimensional latent variable

Telegram: @uni_k

www.konkur.in

$\mathbf{z}$ corresponding to the principal-component subspace. Next we define a Gaussian prior distribution $p(\mathbf{z})$ over the latent variable, together with a Gaussian conditional distribution $p(\mathbf{x}|\mathbf{z})$ for the D-dimensional observed variable $\mathbf{x}$ conditioned on the value of the latent variable. Specifically, the prior distribution over $\mathbf{z}$ is given by a zero-mean unit-covariance Gaussian:

$$p(\mathbf{z}) = \mathcal{N}(\mathbf{z}|\mathbf{0}, \mathbf{I}). \tag{16.31}$$

Similarly, the conditional distribution of the observed variable $\mathbf{x}$, conditioned on the value of the latent variable $\mathbf{z}$, is again Gaussian:

$$p(\mathbf{x}|\mathbf{z}) = \mathcal{N}(\mathbf{x}|\mathbf{W}\mathbf{z} + \boldsymbol{\mu}, \sigma^2\mathbf{I}) \tag{16.32}$$

in which the mean of $\mathbf{x}$ is a general linear function of $\mathbf{z}$ governed by the $D \times M$ matrix $\mathbf{W}$ and the D-dimensional vector $\boldsymbol{\mu}$. Note that this factorizes with respect to the elements of $\mathbf{x}$. In other words this is an example of a naive Bayes model. As we will see shortly, the columns of $\mathbf{W}$ span a linear subspace within the data space that corresponds to the principal subspace. The other parameter in this model is the scalar σ^2 governing the variance of the conditional distribution. Note that there is no loss of generality in assuming a zero-mean unit-covariance Gaussian for the latent distribution $p(\mathbf{z})$ because a more general Gaussian distribution would give rise to an equivalent probabilistic model.

Section 11.2.3

Exercise 16.4

We can view the probabilistic PCA model from a generative viewpoint in which a sampled value of the observed variable is obtained by first choosing a value for the latent variable and then sampling the observed variable conditioned on this latent value. Specifically, the D-dimensional observed variable $\mathbf{x}$ is defined by a linear transformation of the M-dimensional latent variable $\mathbf{z}$ plus additive Gaussian noise, so that

$$\mathbf{x} = \mathbf{W}\mathbf{z} + \boldsymbol{\mu} + \boldsymbol{\epsilon} \tag{16.33}$$

where $\mathbf{z}$ is an M-dimensional Gaussian latent variable, and $\boldsymbol{\epsilon}$ is a D-dimensional zero-mean Gaussian-distributed noise variable with covariance $\sigma^2\mathbf{I}$. This generative process is illustrated in Figure 16.7. Note that this framework is based on a mapping from latent space to data space, in contrast to the more conventional view of PCA discussed above. The reverse mapping, from data space to the latent space, will be obtained shortly using Bayes' theorem.

16.2.2 Likelihood function

Suppose we wish to determine the values of the parameters $\mathbf{W}$, $\boldsymbol{\mu}$, and σ^2 using maximum likelihood. To write down the likelihood function, we need an expression for the marginal distribution $p(\mathbf{x})$ of the observed variable. This is expressed, from the sum and product rules of probability, in the form

$$p(\mathbf{x}) = \int p(\mathbf{x}|\mathbf{z})p(\mathbf{z})\,\mathrm{d}\mathbf{z}. \tag{16.34}$$

Because this corresponds to a linear-Gaussian model, this marginal distribution is again Gaussian, and is given by

Exercise 16.6

Telegram: @uni_k

www.konkur.in

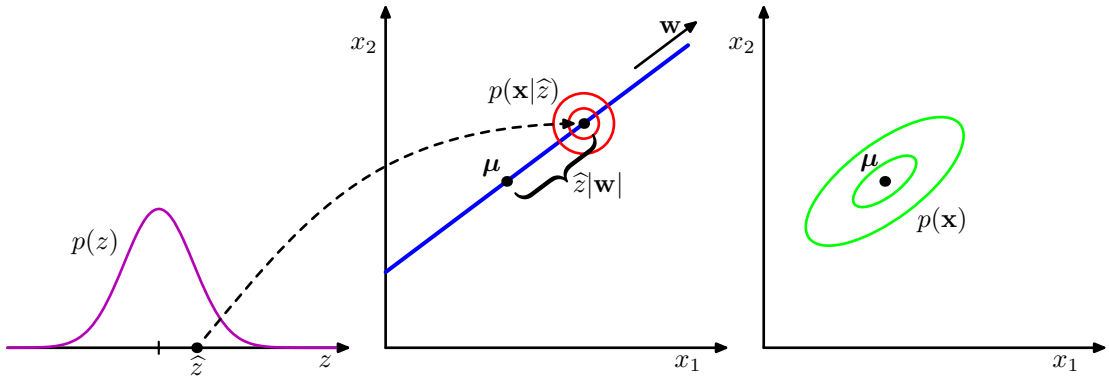


Figure 16.7 An illustration of the generative view of a probabilistic PCA model for a two-dimensional data space and a one-dimensional latent space. An observed data point $\mathbf{x}$ is generated by first drawing a value $\widehat{z}$ for the latent variable from its prior distribution $p(z)$ and then drawing a value for $\mathbf{x}$ from an isotropic Gaussian distribution (illustrated by the red circles) having mean $\mathbf{w}\widehat{z} + \boldsymbol{\mu}$ and covariance $\sigma^2\mathbf{I}$. The green ellipses show the density contours for the marginal distribution $p(\mathbf{x})$.

$$p(\mathbf{x}) = \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \mathbf{C}) \tag{16.35}$$

where the $D \times D$ covariance matrix $\mathbf{C}$ is defined by

$$\mathbf{C} = \mathbf{W}\mathbf{W}^{\mathrm{T}} + \sigma^2\mathbf{I}. \tag{16.36}$$

This result can also be derived more directly by noting that the predictive distribution will be Gaussian and then evaluating its mean and covariance using (16.33). This gives

$$\mathbb{E}[\mathbf{x}] = \mathbb{E}[\mathbf{W}\mathbf{z} + \boldsymbol{\mu} + \boldsymbol{\epsilon}] = \boldsymbol{\mu} \tag{16.37}$$

$$\begin{aligned} \mathrm{cov}[\mathbf{x}] &= \mathbb{E}\left[(\mathbf{W}\mathbf{z} + \boldsymbol{\epsilon})(\mathbf{W}\mathbf{z} + \boldsymbol{\epsilon})^{\mathrm{T}}\right] \\ &= \mathbb{E}\left[\mathbf{W}\mathbf{z}\mathbf{z}^{\mathrm{T}}\mathbf{W}^{\mathrm{T}}\right] + \mathbb{E}[\boldsymbol{\epsilon}\boldsymbol{\epsilon}^{\mathrm{T}}] \qquad (16.38) \\ &= \mathbf{W}\mathbf{W}^{\mathrm{T}} + \sigma^2\mathbf{I} \qquad (16.39) \end{aligned}$$

where we have used the fact that $\mathbf{z}$ and $\boldsymbol{\epsilon}$ are independent random variables and hence are uncorrelated.

Intuitively, we can think of the distribution $p(\mathbf{x})$ as being defined by taking an isotropic Gaussian 'spray can' and moving it across the principal subspace spraying Gaussian ink with density determined by σ^2 and weighted by the prior distribution. The accumulated ink density gives rise to a 'pancake' shaped distribution representing the marginal density $p(\mathbf{x})$.

The predictive distribution $p(\mathbf{x})$ is governed by the parameters $\boldsymbol{\mu}$, $\mathbf{W}$, and σ^2. However, there is redundancy in this parameterization corresponding to rotations of the latent space coordinates. To see this, consider a matrix $\widetilde{\mathbf{W}} = \mathbf{W}\mathbf{R}$ where $\mathbf{R}$ is an orthogonal matrix. Using the orthogonality property $\mathbf{R}\mathbf{R}^{\mathrm{T}} = \mathbf{I}$, we see that the quantity $\widetilde{\mathbf{W}}\widetilde{\mathbf{W}}^{\mathrm{T}}$ that appears in the covariance matrix $\mathbf{C}$ takes the form

$$\widetilde{\mathbf{W}}\widetilde{\mathbf{W}}^{\mathrm{T}} = \mathbf{W}\mathbf{R}\mathbf{R}^{\mathrm{T}}\mathbf{W}^{\mathrm{T}} = \mathbf{W}\mathbf{W}^{\mathrm{T}} \tag{16.40}$$

Telegram: @uni_k

www.konkur.in

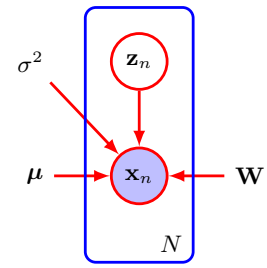


Figure 16.8 The probabilistic PCA model for a data set of N observations of $\mathbf{x}$ can be expressed as a directed graph in which each observation $\mathbf{x}_n$ is associated with a value $\mathbf{z}_n$ of the latent variable.

and hence is independent of $\mathbf{R}$. Thus, there is a whole family of matrices $\widetilde{\mathbf{W}}$ all of which give rise to the same predictive distribution. This invariance can be understood in terms of rotations within the latent space. We will return to a discussion of the number of independent parameters in this model later.

When we evaluate the predictive distribution, we require $\mathbf{C}^{-1}$, which involves the inversion of a $D \times D$ matrix. The computation required to do this can be reduced by making use of the matrix inversion identity (A.7) to give

$$\mathbf{C}^{-1} = \sigma^{-2}\mathbf{I} - \sigma^{-2}\mathbf{W}\mathbf{M}^{-1}\mathbf{W}^{\mathrm{T}} \tag{16.41}$$

where the $M \times M$ matrix $\mathbf{M}$ is defined by

$$\mathbf{M} = \mathbf{W}^{\mathrm{T}}\mathbf{W} + \sigma^2\mathbf{I}. \tag{16.42}$$

Because we invert $\mathbf{M}$ rather than inverting $\mathbf{C}$ directly, the cost of evaluating $\mathbf{C}^{-1}$ is reduced from $\mathcal{O}(D^3)$ to $\mathcal{O}(M^3)$.

As well as the predictive distribution $p(\mathbf{x})$, we will also require the posterior distribution $p(\mathbf{z}|\mathbf{x})$, which can again be written down directly using the result (3.100) for linear-Gaussian models to give

Exercise 16.8

$$p(\mathbf{z}|\mathbf{x}) = \mathcal{N}\left(\mathbf{z}|\mathbf{M}^{-1}\mathbf{W}^{\mathrm{T}}(\mathbf{x} - \boldsymbol{\mu}), \sigma^2\mathbf{M}^{-1}\right). \tag{16.43}$$

Note that the posterior mean depends on $\mathbf{x}$, whereas the posterior covariance is independent of $\mathbf{x}$.

16.2.3 Maximum likelihood

We next consider the determination of the model parameters using maximum likelihood. Given a data set $\mathbf{X} = \{\mathbf{x}_n\}$ of observed data points, the probabilistic PCA model can be expressed as a directed graph, as shown in Figure 16.8. The corresponding log likelihood function is given, from (16.35), by

$$\begin{aligned} \ln p(\mathbf{X}|\boldsymbol{\mu}, \mathbf{W}, \sigma^2) &= \sum_{n=1}^{N} \ln p(\mathbf{x}_n|\mathbf{W}, \boldsymbol{\mu}, \sigma^2) \\ &= -\frac{ND}{2}\ln(2\pi) - \frac{N}{2}\ln|\mathbf{C}| - \frac{1}{2}\sum_{n=1}^{N}(\mathbf{x}_n - \boldsymbol{\mu})^{\mathrm{T}}\mathbf{C}^{-1}(\mathbf{x}_n - \boldsymbol{\mu}). \end{aligned} \tag{16.44}$$

Telegram: @uni_k

www.konkur.in

Exercise 16.9

Setting the derivative of the log likelihood with respect to $\boldsymbol{\mu}$ equal to zero gives the expected result $\boldsymbol{\mu} = \overline{\mathbf{x}}$ where $\overline{\mathbf{x}}$ is the data mean defined by (16.1). Because the log likelihood is a quadratic function of $\boldsymbol{\mu}$, this solution represents the unique maximum, as can be confirmed by computing second derivatives. Back-substituting, we can then write the log likelihood function in the form

$$\ln p(\mathbf{X}|\mathbf{W}, \boldsymbol{\mu}, \sigma^2) = -\frac{N}{2}\left\{D\ln(2\pi) + \ln|\mathbf{C}| + \mathrm{Tr}\left(\mathbf{C}^{-1}\mathbf{S}\right)\right\} \tag{16.45}$$

where $\mathbf{S}$ is the data covariance matrix defined by (16.3).

Maximization with respect to $\mathbf{W}$ and σ^2 is more complex but nonetheless has an exact closed-form solution. It was shown by Tipping and Bishop (1999) that all the stationary points of the log likelihood function can be written as

$$\mathbf{W}_{\mathrm{ML}} = \mathbf{U}_M(\mathbf{L}_M - \sigma^2\mathbf{I})^{1/2}\mathbf{R} \tag{16.46}$$

where $\mathbf{U}_M$ is a $D \times M$ matrix whose columns are given by any subset (of size M) of the eigenvectors of the data covariance matrix $\mathbf{S}$. The $M \times M$ diagonal matrix $\mathbf{L}_M$ has elements given by the corresponding eigenvalues λ_i, and $\mathbf{R}$ is an arbitrary $M \times M$ orthogonal matrix.

Furthermore, Tipping and Bishop (1999) showed that the *maximum* of the likelihood function is obtained when the M eigenvectors are chosen to be those whose eigenvalues are the M largest (all other solutions being saddle points). A similar result was conjectured independently by Roweis (1998), although no proof was given. Again, we will assume that the eigenvectors have been arranged in order of decreasing values of the corresponding eigenvalues, so that the M principal eigenvectors are $\mathbf{u}_1, \ldots, \mathbf{u}_M$. In this case, the columns of $\mathbf{W}$ define the principal subspace of standard PCA. The corresponding maximum likelihood solution for σ^2 is then given by

$$\sigma^2_{\mathrm{ML}} = \frac{1}{D-M}\sum_{i=M+1}^{D}\lambda_i \tag{16.47}$$

so that σ^2_{ML} is the average variance associated with the discarded dimensions.

Because $\mathbf{R}$ is orthogonal, it can be interpreted as a rotation matrix in the M-dimensional latent space. If we substitute the solution for $\mathbf{W}$ into the expression for $\mathbf{C}$ and make use of the orthogonality property $\mathbf{R}\mathbf{R}^{\mathrm{T}} = \mathbf{I}$, we see that $\mathbf{C}$ is independent of $\mathbf{R}$. This simply says that the predictive density is unchanged by rotations in the latent space as discussed earlier. For the particular case $\mathbf{R} = \mathbf{I}$, we see that the columns of $\mathbf{W}$ are the principal component eigenvectors scaled by the variance parameters $\lambda_i - \sigma^2$. The interpretation of these scaling factors is clear once we recognize that for a convolution of independent Gaussian distributions (in this case the latent space distribution and the noise model) the variances are additive. Thus, the variance λ_i in the direction of an eigenvector $\mathbf{u}_i$ is composed of the sum of a contribution $\lambda_i - \sigma^2$ from the projection of the unit-variance latent space distribution into data space through the corresponding column of $\mathbf{W}$ plus an isotropic contribution of variance σ^2, which is added in all directions by the noise model.

Telegram: @uni_k

www.konkur.in

It is worth taking a moment to study the form of the covariance matrix given by (16.36). Consider the variance of the predictive distribution along some direction specified by the unit vector $\mathbf{v}$, where $\mathbf{v}^{\mathrm{T}}\mathbf{v} = 1$, which is given by $\mathbf{v}^{\mathrm{T}}\mathbf{C}\mathbf{v}$. First suppose that $\mathbf{v}$ is orthogonal to the principal subspace, in other words it is given by some linear combination of the discarded eigenvectors. Then $\mathbf{v}^{\mathrm{T}}\mathbf{U} = \mathbf{0}$ and hence $\mathbf{v}^{\mathrm{T}}\mathbf{C}\mathbf{v} = \sigma^2$. Thus, the model predicts a noise variance orthogonal to the principal subspace, which from (16.47) is just the average of the discarded eigenvalues. Now suppose that $\mathbf{v} = \mathbf{u}_i$ where $\mathbf{u}_i$ is one of the retained eigenvectors defining the principal subspace. Then $\mathbf{v}^{\mathrm{T}}\mathbf{C}\mathbf{v} = (\lambda_i - \sigma^2) + \sigma^2 = \lambda_i$. In other words, this model correctly captures the variance of the data along the principal axes and approximates the variance in all remaining directions with a single average value σ^2.

One way to construct the maximum likelihood density model would simply be to find the eigenvectors and eigenvalues of the data covariance matrix and then to evaluate $\mathbf{W}$ and σ^2 using the results given above. In this case, we would choose $\mathbf{R} = \mathbf{I}$ for convenience. However, if the maximum likelihood solution is found by numerical optimization of the likelihood function, for instance using an algorithm such as conjugate gradients (Fletcher, 1987; Nocedal and Wright, 1999) or through the EM algorithm, then the resulting value of $\mathbf{R}$ is essentially arbitrary. This implies that the columns of $\mathbf{W}$ need not be orthogonal. If an orthogonal basis is required, the matrix $\mathbf{W}$ can be post-processed appropriately (Golub and Van Loan, 1996). Alternatively, the EM algorithm can be modified in such a way as to yield orthonormal principal directions, sorted in descending order of the corresponding eigenvalues, directly (Ahn and Oh, 2003).

Section 16.3.2

The rotational invariance in latent space represents a form of statistical non-identifiability, analogous to that encountered for mixture models for discrete latent variables. Here there is a continuum of parameters, any value of which leads to the same predictive density, in contrast to the discrete non-identifiability associated with component relabelling in the mixture setting.

If we consider $M = D$, so that there is no reduction of dimensionality, then $\mathbf{U}_M = \mathbf{U}$ and $\mathbf{L}_M = \mathbf{L}$. Making use of the orthogonality properties $\mathbf{U}\mathbf{U}^{\mathrm{T}} = \mathbf{I}$ and $\mathbf{R}\mathbf{R}^{\mathrm{T}} = \mathbf{I}$, we see that the covariance $\mathbf{C}$ of the marginal distribution for $\mathbf{x}$ becomes

$$\mathbf{C} = \mathbf{U}(\mathbf{L} - \sigma^2\mathbf{I})^{1/2}\mathbf{R}\mathbf{R}^{\mathrm{T}}(\mathbf{L} - \sigma^2\mathbf{I})^{1/2}\mathbf{U}^{\mathrm{T}} + \sigma^2\mathbf{I} = \mathbf{U}\mathbf{L}\mathbf{U}^{\mathrm{T}} = \mathbf{S} \tag{16.48}$$

and so we obtain the standard maximum likelihood solution for an unconstrained Gaussian distribution in which the covariance matrix is given by the sample covariance.

Conventional PCA is generally formulated as a projection of points from the D-dimensional data space onto an M-dimensional linear subspace. Probabilistic PCA, however, is most naturally expressed as a mapping from the latent space into the data space via (16.33). For applications such as visualization and data compression, we can reverse this mapping using Bayes' theorem. Any point $\mathbf{x}$ in data space can then be summarized by its posterior mean and covariance in latent space. From (16.43) the mean is given by

$$\mathbb{E}[\mathbf{z}|\mathbf{x}] = \mathbf{M}^{-1}\mathbf{W}_{\mathrm{ML}}^{\mathrm{T}}(\mathbf{x} - \overline{\mathbf{x}}) \tag{16.49}$$

Telegram: @uni_k

www.konkur.in

where $\mathbf{M}$ is given by (16.42). This projects to a point in data space given by

$$\mathbf{W}\,\mathbb{E}[\mathbf{z}|\mathbf{x}] + \boldsymbol{\mu}. \tag{16.50}$$

Section 4.1.6 Note that this takes the same form as the equations for regularized linear regression and is a consequence of maximizing the likelihood function for a linear-Gaussian model. Similarly, from (16.43) the posterior covariance is given by $\sigma^2\mathbf{M}^{-1}$ and is independent of $\mathbf{x}$.

If we take the limit $\sigma^2 \to 0$, then the posterior mean reduces to

$$(\mathbf{W}_{\mathrm{ML}}^{\mathrm{T}}\mathbf{W}_{\mathrm{ML}})^{-1}\mathbf{W}_{\mathrm{ML}}^{\mathrm{T}}(\mathbf{x} - \overline{\mathbf{x}}), \tag{16.51}$$

which represents an orthogonal projection of the data point onto the latent space, *Exercise 16.11* and so we recover the standard PCA model. The posterior covariance in this limit is zero, however, and the density becomes singular. For $\sigma^2 > 0$, the latent projection *Exercise 16.12* is shifted towards the origin, relative to the orthogonal projection.

Finally, note that an important role for the probabilistic PCA model is in defining a multivariate Gaussian distribution in which the number of degrees of freedom, in other words the number of independent parameters, can be controlled while still allowing the model to capture the dominant correlations in the data. Recall that a general Gaussian distribution has $D(D+1)/2$ independent parameters in its co- *Section 3.2* variance matrix (plus another D parameters in its mean). Thus, the number of parameters scales quadratically with D and can become excessive in spaces of high dimensionality. If we restrict the covariance matrix to be diagonal, then it has only D independent parameters, and so the number of parameters now grows linearly with dimensionality. However, it now treats the variables as if they were independent and hence can no longer express any correlations between them. Probabilistic PCA provides an elegant compromise in which the M most significant correlations can be captured while still ensuring that the total number of parameters grows only linearly with D. We can see this by evaluating the number of degrees of freedom in the probabilistic PCA model as follows. The covariance matrix $\mathbf{C}$ depends on the parameters $\mathbf{W}$, which has size $D \times M$, and σ^2, giving a total parameter count of $DM + 1$. However, we have seen that there is some redundancy in this parameterization associated with rotations of the coordinate system in the latent space. The orthogonal matrix $\mathbf{R}$ that expresses these rotations has size $M \times M$. In the first column of this matrix, there are $M - 1$ independent parameters, because the column vector must be normalized to unit length. In the second column, there are $M - 2$ independent parameters, because the column must be normalized and also must be orthogonal to the previous column, and so on. Summing this arithmetic series, we see that $\mathbf{R}$ has a total of $M(M-1)/2$ independent parameters. Thus, the number of degrees of freedom in the covariance matrix $\mathbf{C}$ is given by

$$DM + 1 - M(M-1)/2. \tag{16.52}$$

The number of independent parameters in this model therefore only grows linearly with D, for fixed M. If we take $M = D - 1$, then we recover the standard result *Exercise 16.14* for a full covariance Gaussian. In this case, the variance along $D - 1$ linearly in-

Telegram: @uni_k

www.konkur.in

dependent directions is controlled by the columns of $\mathbf{W}$, and the variance along the remaining direction is given by σ^2. If $M = 0$, the model is equivalent to the isotropic covariance case.

16.2.4 Factor analysis

Factor analysis is a linear-Gaussian latent-variable model that is closely related to probabilistic PCA. Its definition differs from that of probabilistic PCA only in that the conditional distribution of the observed variable $\mathbf{x}$ given the latent variable $\mathbf{z}$ has a diagonal rather than an isotropic covariance so that

$$p(\mathbf{x}|\mathbf{z}) = \mathcal{N}(\mathbf{x}|\mathbf{W}\mathbf{z} + \boldsymbol{\mu}, \boldsymbol{\Psi}) \tag{16.53}$$

where $\boldsymbol{\Psi}$ is a $D \times D$ diagonal matrix. Note that the factor analysis model, in common with probabilistic PCA, assumes that the observed variables $x_1, \ldots, x_D$ are independent, given the latent variable $\mathbf{z}$. In essence, a factor analysis model explains the observed covariance structure of the data by representing the independent variance associated with each coordinate in the matrix $\boldsymbol{\Psi}$ and capturing the covariance between variables in the matrix $\mathbf{W}$. In the factor analysis literature, the columns of $\mathbf{W}$, which capture the correlations between observed variables, are called *factor loadings*, and the diagonal elements of $\boldsymbol{\Psi}$, which represent the independent noise variances for each of the variables, are called *uniquenesses*.

The origins of factor analysis are as old as those of PCA, and discussions of factor analysis can be found in the books by Everitt (1984), Bartholomew (1987), and Basilevsky (1994). Links between factor analysis and PCA were investigated by Lawley (1953) and Anderson (1963), who showed that at stationary points of the likelihood function, for a factor analysis model with $\boldsymbol{\Psi} = \sigma^2\mathbf{I}$, the columns of $\mathbf{W}$ are scaled eigenvectors of the sample covariance matrix and σ^2 is the average of the discarded eigenvalues. Later, Tipping and Bishop (1999) showed that the maximum of the log likelihood function occurs when the eigenvectors comprising $\mathbf{W}$ are chosen to be the principal eigenvectors.

Making use of (16.34), we see that the marginal distribution for the observed variable is given by $p(\mathbf{x}) = \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \mathbf{C})$ where now

$$\mathbf{C} = \mathbf{W}\mathbf{W}^{\mathrm{T}} + \boldsymbol{\Psi}. \tag{16.54}$$

Exercise 16.16 As with probabilistic PCA, this model is invariant to rotations in the latent space.

Historically, factor analysis has been the subject of controversy when attempts have been made to place an interpretation on the individual factors (the coordinates in $\mathbf{z}$-space), which has proven problematic due to the non-identifiability of factor analysis associated with rotations in this space. From our perspective, however, we shall view factor analysis as a form of latent-variable density model, in which the form of the latent space is of interest but not the particular choice of coordinates used to describe it. If we wish to remove the degeneracy associated with latent-space rotations, we must consider non-Gaussian latent-variable distributions, giving rise to independent component analysis models.

Section 16.2.5

Another difference between probabilistic PCA and factor analysis is their behaviour under transformations of the data set. For PCA and probabilistic PCA, if we

Exercise 16.17

Telegram: @uni_k

www.konkur.in

rotate the coordinate system in data space, then we obtain exactly the same fit to the data but with the $\mathbf{W}$ matrix transformed by the corresponding rotation matrix. However, for factor analysis, the analogous property is that if we make a component-wise re-scaling of the data vectors, then this is absorbed into a corresponding re-scaling of the elements of $\mathbf{\Psi}$.

16.2.5 Independent component analysis

One generalization of the linear-Gaussian latent-variable model is to consider models in which the observed variables are related linearly to the latent variables, but for which the latent distribution is non-Gaussian. An important class of such models, known as *independent component analysis*, or ICA, arises when we consider a distribution over the latent variables that factorizes, so that

$$p(\mathbf{z}) = \prod_{j=1}^{M} p(z_j). \tag{16.55}$$

To understand the role of such models, consider a situation in which two people are talking at the same time, and we record their voices using two microphones. If we ignore effects such as time delay and echoes, then the signals received by the microphones at any point in time will be given by linear combinations of the amplitudes of the two voices. The coefficients of this linear combination will be constant, and if we can infer their values from sample data, then we can invert the mixing process (assuming it is non-singular) and thereby obtain two clean signals each of which contains the voice of just one person. This is an example of a problem called *blind source separation* in which 'blind' refers to the fact that we are given only the mixed data, and neither the original sources nor the mixing coefficients are observed (Cardoso, 1998).

This type of problem is sometimes addressed using the following approach (MacKay, 2003) in which we ignore the temporal nature of the signals and treat the successive samples as i.i.d. We consider a generative model in which there are two latent variables corresponding to the unobserved speech signal amplitudes, and there are two observed variables given by the signal values at the microphones. The latent variables have a joint distribution that factorizes as above, and the observed variables are given by a linear combination of the latent variables. There is no need to include a noise distribution because the number of latent variables equals the number of observed variables, and therefore the marginal distribution of the observed variables will not in general be singular, so the observed variables are simply deterministic linear combinations of the latent variables. Given a data set of observations, the likelihood function for this model is a function of the coefficients in the linear combination. The log likelihood can be maximized using gradient-based optimization giving rise to a particular version of ICA.

The success of this approach requires that the latent variables have non-Gaussian distributions. To see this, recall that in probabilistic PCA (and in factor analysis) the latent-space distribution is given by a zero-mean isotropic Gaussian. The model therefore cannot distinguish between two different choices for the latent variables

Telegram: @uni_k

www.konkur.in

if these differ simply by a rotation in latent space. This can be verified directly by noting that the marginal density (16.35), and hence the likelihood function, is unchanged if we make the transformation $\mathbf{W} \to \mathbf{WR}$ where $\mathbf{R}$ is an orthogonal matrix satisfying $\mathbf{RR}^{\mathrm{T}} = \mathbf{I}$, because the matrix $\mathbf{C}$ given by (16.36) is itself invariant. Extending the model to allow more general Gaussian latent distributions does not change this conclusion because, as we have seen, such a model is equivalent to the zero-mean isotropic Gaussian latent-variable model.

Another way to see why a Gaussian latent-variable distribution in a linear model is insufficient to find independent components is to note that the principal components represent a rotation of the coordinate system in data space so as to diagonalize the covariance matrix. The data distribution in the new coordinates is then uncorrelated. Although zero correlation is a necessary condition for independence it is not, however, sufficient. In practice, a common choice for the latent-variable distribution is given by

Exercise 2.39

$$p(z_j) = \frac{1}{\pi \cosh(z_j)} = \frac{2}{\pi(e^{z_j} + e^{-z_j})}, \tag{16.56}$$

which has heavy tails compared to a Gaussian, reflecting the observation that many real-world distributions also exhibit this property.

The original ICA model (Bell and Sejnowski, 1995) was based on the optimization of an objective function defined by information maximization. One advantage of a probabilistic latent-variable formulation is that it helps to motivate and formulate generalizations of basic ICA. For instance, *independent factor analysis* (Attias, 1999) considers a model in which the number of latent and observed variables can differ, the observed variables are noisy, and the individual latent variables have flexible distributions modelled by mixtures of Gaussians. The log likelihood for this model is maximized using EM, and the reconstruction of the latent variables is approximated using a variational approach. Many other types of model have been considered, and there is now a huge literature on ICA and its applications (Jutten and Herault, 1991; Comon, Jutten, and Herault, 1991; Amari, Cichocki, and Yang, 1996; Pearlmutter and Parra, 1997; Hyvärinen and Oja, 1997; Hinton *et al.*, 2001; Miskin and MacKay, 2001; Hojen-Sorensen, Winther, and Hansen, 2002; Choudrey and Roberts, 2003; Chan, Lee, and Sejnowski, 2003; Stone, 2004).

16.2.6 Kalman filters

So far we have assumed that the data values are i.i.d. A common situation in which this assumption does not hold is when the data points form an ordered sequence. We have seen that a hidden Markov model can be viewed as an extension of the mixture models to allow for sequential correlations in the data. In a similar way, a continuous latent-variable model can be extended to handle sequential data by connecting the latent variables to form a Markov chain, as shown in the graphical model of Figure 16.9. This is known as a *linear dynamical system* or *Kalman filter* (Zarchan and Musoff, 2005). Note that this is the same graphical structure as a hidden Markov model. It is interesting to note that, historically, hidden Markov models and linear dynamical systems were developed independently. Once they are both expressed as graphical models, however, the deep relationship between them

Section 15.3.1

Section 15.3.1

Telegram: @uni_k

www.konkur.in

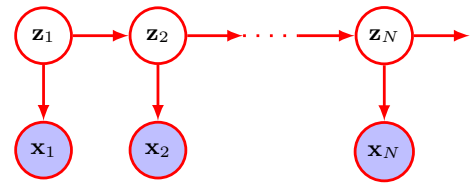


Figure 16.9 A probabilistic graphical model for sequential data, known as a linear dynamical system, or Kalman filter, in which the latent variables form a Markov chain.

immediately becomes apparent. Kalman filters are widely used in many real-time tracking applications, for example to track aircraft using radar reflections.

In the simplest such model, the distributions $p(\mathbf{x}_n|\mathbf{z}_n)$ in Figure 16.9 represent a linear-Gaussian latent-variable model for that particular observation, of the kind we have discussed previously for i.i.d. data. However, the latent variables $\{\mathbf{z}_n\}$ are no longer treated as independent but now form a Markov chain in which the distribution $p(\mathbf{z}_n|\mathbf{z}_{n-1})$ of each latent variable is conditioned on the state of the previous latent variable in the chain. Again these can be chosen to be linear-Gaussian in which the distribution of $\mathbf{z}_n$ is Gaussian with a mean given by a linear function of $\mathbf{z}_{n-1}$. Typically the parameters of all the distributions $p(\mathbf{x}_n|\mathbf{z}_n)$ are shared, and likewise the parameters of the distributions $p(\mathbf{z}_n|\mathbf{z}_{n-1})$ are shared, so that the total number of parameters in the model is fixed, independently of the length of the sequence. These parameters can be learned from data by maximum likelihood with efficient algorithms that involve propagating messages around the graph (Bishop, 2006). For the rest of this chapter, however, we will focus on i.i.d. data.

16.3. Evidence Lower Bound

Section 15.4

In our discussion of models with discrete latent variables, we derived the evidence lower bound (ELBO) on the marginal log likelihood and showed how this forms the basis for deriving the expectation–maximization (EM) algorithm including its generalizations such as variational inference. The same framework applies to continuous latent variables as well as to models that combine both discrete and continuous variables. Here we present a slightly different derivation of the ELBO, and we assume that the latent variables $\mathbf{z}$ are continuous.

Consider a model $p(\mathbf{x}, \mathbf{z}|\mathbf{w})$ with an observed variable $\mathbf{x}$, a latent variable $\mathbf{z}$, and a learnable parameter vector $\mathbf{w}$. If we introduce an arbitrary distribution $q(\mathbf{z})$ over the latent variable then we can write the log likelihood function $\ln p(\mathbf{x}|\mathbf{w})$ as a sum of two terms in the form

Exercise 16.18

$$\ln p(\mathbf{x}|\mathbf{w}) = \mathcal{L}(\mathbf{w}) + \mathrm{KL}\left(q(\mathbf{z})\|p(\mathbf{z}|\mathbf{x}, \mathbf{w})\right) \tag{16.57}$$

where we have defined

$$\mathcal{L}(q, \mathbf{w}) = \int q(\mathbf{z}) \ln \left\{ \frac{p(\mathbf{x}, \mathbf{z}|\mathbf{w})}{q(\mathbf{z})} \right\} \mathrm{d}\mathbf{z} \tag{16.58}$$

$$\mathrm{KL}\left(q(\mathbf{z})\|p(\mathbf{z}|\mathbf{x}, \mathbf{w})\right) = -\int q(\mathbf{z}) \ln \left\{ \frac{p(\mathbf{z}|\mathbf{x}, \mathbf{w})}{q(\mathbf{z})} \right\} \mathrm{d}\mathbf{z}. \tag{16.59}$$

Telegram: @uni_k

www.konkur.in

Since $\mathrm{KL}\,(q(\mathbf{z})\|p(\mathbf{z}|\mathbf{x},\mathbf{w}))$ is a Kullback–Leibler divergence, it satisfies the property $\mathrm{KL}\,(\cdot\|\cdot) \geqslant 0$ from which it follows that

Section 2.5.5

$$\ln p(\mathbf{x}|\mathbf{w}) \geqslant \mathcal{L}(\mathbf{w}) \tag{16.60}$$

and we therefore see that $\mathcal{L}(q,\mathbf{w})$ given by (16.58) forms a lower bound on the log likelihood, known as the *evidence lower bound* or ELBO. We see that $\mathcal{L}(q,\mathbf{w})$ takes the same form (15.53) as derived for the discrete case but with summations replaced by integrals.

We can maximize the log likelihood function using a two-stage iterative procedure called the *expectation maximization* algorithm, or EM algorithm, in which we alternately maximize $\mathcal{L}(q,\mathbf{w})$ with respect to $q(\mathbf{z})$ (the E step) and $\mathbf{w}$ (the M step). We first initialize the parameters $\mathbf{w}^{(\mathrm{old})}$. Then in the E step we keep $\mathbf{w}$ fixed and we maximize the lower bound with respect to $q(\mathbf{z})$. This is easily done by noting that the highest value for the bound is obtained by minimizing the Kullback–Leibler divergence in (16.59) and hence is achieved when $q(\mathbf{z}) = p(\mathbf{z}|\mathbf{x},\mathbf{w}^{(\mathrm{old})})$ for which the Kullback–Leibler divergence is zero. In the M step, we keep this choice of $q(\mathbf{z})$ fixed and maximize $\mathcal{L}(q,\mathbf{w})$ with respect to $\mathbf{w}$. Substituting for $q(\mathbf{z})$ in (16.58) we obtain

$$\begin{aligned}\mathcal{L}(q,\mathbf{w}) = &\int p(\mathbf{z}|\mathbf{x},\mathbf{w}^{(\mathrm{old})}) \ln p(\mathbf{x},\mathbf{z}|\mathbf{w})\,\mathrm{d}\mathbf{z} \\ &- \int p(\mathbf{z}|\mathbf{x},\mathbf{w}^{(\mathrm{old})}) \ln p(\mathbf{z}|\mathbf{x},\mathbf{w}^{(\mathrm{old})})\,\mathrm{d}\mathbf{z}.\end{aligned} \tag{16.61}$$

We now maximize this with respect to $\mathbf{w}$ in the M step while keeping $\mathbf{w}^{(\mathrm{old})}$ fixed. Note that the second term on the right-hand side of (16.61) is independent of $\mathbf{w}$ and so can be ignored during the M step. The first term on the right-hand side is the expectation of the *complete data log likelihood* where the expectation is taken with respect to the posterior distribution of $\mathbf{z}$ computed using $\mathbf{w}^{(\mathrm{old})}$.

Section 15.3

If we have a data set $\mathbf{x}_1,\ldots,\mathbf{x}_N$ of i.i.d. observations then the likelihood function takes the form

$$\ln p(\mathbf{X}|\mathbf{w}) = \sum_{n=1}^{N} \ln p(\mathbf{x}_n|\mathbf{w}) \tag{16.62}$$

where the data matrix $\mathbf{X}$ comprises $\mathbf{x}_1,\ldots,\mathbf{x}_N$, and the parameters $\mathbf{w}$ are shared across all data points. For each data point we introduce a corresponding latent variable $\mathbf{z}_n$ with its associated distribution $q(\mathbf{z}_n)$, and by following similar steps to those used to derive (16.58), we obtain the ELBO in the form

Exercise 16.19

$$\mathcal{L}(q,\mathbf{w}) = \sum_{n=1}^{N} \int q(\mathbf{z}_n) \ln \left\{ \frac{p(\mathbf{x}_n,\mathbf{z}_n|\mathbf{w})}{q(\mathbf{z}_n)} \right\} \mathrm{d}\mathbf{z}_n. \tag{16.63}$$

Section 19.2

When we discuss variational autoencoders, we will encounter a model for which an exact solution to the E step is not feasible so instead a partial maximization is performed by modelling $q(\mathbf{z})$ using a deep neural network and then using the ELBO to learn the parameters of the network.

Telegram: @uni_k

www.konkur.in

16.3.1 Expectation maximization

We can now use the EM algorithm, derived by iteratively maximizing the evidence lower bound, to learn the parameters of the probabilistic PCA model. This may seem rather pointless because we have already obtained an exact closed-form solution for the maximum likelihood parameter values. However, in spaces of high dimensionality, there may be computational advantages in using an iterative EM procedure rather than working directly with the sample covariance matrix. This EM procedure can also be extended to the factor analysis model, for which there is no closed-form solution. Finally, it allows missing data to be handled in a principled way.

Section 16.2.4

We can derive the EM algorithm for probabilistic PCA by following the general framework for EM. Thus, we write down the complete-data log likelihood and take its expectation with respect to the posterior distribution of the latent distribution evaluated using 'old' parameter values. Maximization of this expected complete-data log likelihood then yields the 'new' parameter values. Because the data points are assumed independent, the complete-data log likelihood function takes the form

Section 15.3

$$\ln p\left(\mathbf{X}, \mathbf{Z}|\boldsymbol{\mu}, \mathbf{W}, \sigma^2\right) = \sum_{n=1}^{N} \{\ln p(\mathbf{x}_n|\mathbf{z}_n) + \ln p(\mathbf{z}_n)\} \tag{16.64}$$

where the nth row of the matrix $\mathbf{Z}$ is given by $\mathbf{z}_n$. We already know that the exact maximum likelihood solution for $\boldsymbol{\mu}$ is given by the sample mean $\overline{\mathbf{x}}$ defined by (16.1), and it is convenient to substitute for $\boldsymbol{\mu}$ at this stage. Making use of the expressions (16.31) and (16.32) for the latent and conditional distributions, respectively, and taking the expectation with respect to the posterior distribution over the latent variables, we obtain

$$\begin{aligned}\mathbb{E}[\ln p\left(\mathbf{X}, \mathbf{Z}|\boldsymbol{\mu}, \mathbf{W}, \sigma^2\right)] = -\sum_{n=1}^{N} \Bigg\{ & \frac{D}{2}\ln(2\pi\sigma^2) + \frac{1}{2}\mathrm{Tr}\left(\mathbb{E}[\mathbf{z}_n\mathbf{z}_n^{\mathrm{T}}]\right) \\ & + \frac{1}{2\sigma^2}\|\mathbf{x}_n - \boldsymbol{\mu}\|^2 - \frac{1}{\sigma^2}\mathbb{E}[\mathbf{z}_n]^{\mathrm{T}}\mathbf{W}^{\mathrm{T}}(\mathbf{x}_n - \boldsymbol{\mu}) \\ & + \frac{1}{2\sigma^2}\mathrm{Tr}\left(\mathbb{E}[\mathbf{z}_n\mathbf{z}_n^{\mathrm{T}}]\mathbf{W}^{\mathrm{T}}\mathbf{W}\right) + \frac{M}{2}\ln(2\pi)\Bigg\}. \end{aligned} \tag{16.65}$$

Note that this depends on the posterior distribution only through the sufficient statistics of the Gaussian. Thus, in the E step, we use the old parameter values to evaluate

$$\mathbb{E}[\mathbf{z}_n] = \mathbf{M}^{-1}\mathbf{W}^{\mathrm{T}}(\mathbf{x}_n - \overline{\mathbf{x}}) \tag{16.66}$$

$$\mathbb{E}[\mathbf{z}_n\mathbf{z}_n^{\mathrm{T}}] = \sigma^2\mathbf{M}^{-1} + \mathbb{E}[\mathbf{z}_n]\mathbb{E}[\mathbf{z}_n]^{\mathrm{T}}, \tag{16.67}$$

which follow directly from the posterior distribution (16.43) together with the standard result $\mathbb{E}[\mathbf{z}_n\mathbf{z}_n^{\mathrm{T}}] = \mathrm{cov}[\mathbf{z}_n] + \mathbb{E}[\mathbf{z}_n]\mathbb{E}[\mathbf{z}_n]^{\mathrm{T}}$. Here $\mathbf{M}$ is defined by (16.42).

In the M step, we maximize with respect to $\mathbf{W}$ and σ^2, keeping the posterior statistics fixed. Maximization with respect to σ^2 is straightforward. For the maximization with respect to $\mathbf{W}$, we make use of (A.24) to obtain the M-step equations:

Exercise 16.21

Telegram: @uni_k

www.konkur.in

$$\mathbf{W}_{\text{new}} = \left[\sum_{n=1}^{N}(\mathbf{x}_n - \overline{\mathbf{x}})\mathbb{E}[\mathbf{z}_n]^{\mathrm{T}}\right]\left[\sum_{n=1}^{N}\mathbb{E}[\mathbf{z}_n\mathbf{z}_n^{\mathrm{T}}]\right]^{-1} \tag{16.68}$$

$$\sigma^2_{\text{new}} = \frac{1}{ND}\sum_{n=1}^{N}\left\{\|\mathbf{x}_n - \overline{\mathbf{x}}\|^2 - 2\mathbb{E}[\mathbf{z}_n]^{\mathrm{T}}\mathbf{W}_{\text{new}}^{\mathrm{T}}(\mathbf{x}_n - \overline{\mathbf{x}}) + \mathrm{Tr}\left(\mathbb{E}[\mathbf{z}_n\mathbf{z}_n^{\mathrm{T}}]\mathbf{W}_{\text{new}}^{\mathrm{T}}\mathbf{W}_{\text{new}}\right)\right\}. \tag{16.69}$$

The EM algorithm for probabilistic PCA proceeds by initializing the parameters and then alternately computing the sufficient statistics of the latent space posterior distribution using (16.66) and (16.67) in the E step and revising the parameter values using (16.68) and (16.69) in the M step.

One of the benefits of the EM algorithm for PCA is its computational efficiency for large-scale applications (Roweis, 1998). Unlike conventional PCA based on an eigenvector decomposition of the sample covariance matrix, the EM approach is iterative and so might appear to be less attractive. However, each cycle of the EM algorithm can be computationally much more efficient than conventional PCA in spaces of high dimensionality. To see this, note that the eigendecomposition of the covariance matrix requires $\mathcal{O}(D^3)$ computation. Often we are interested only in the first M eigenvectors and their corresponding eigenvalues, in which case we can use algorithms that are $\mathcal{O}(MD^2)$. However, evaluating the covariance matrix requires $\mathcal{O}(ND^2)$ computations, where N is the number of data points. Algorithms such as the snapshot method (Sirovich, 1987), which assume that the eigenvectors are linear combinations of the data vectors, avoid a direct evaluation of the covariance matrix but are $\mathcal{O}(N^3)$ and hence unsuited to large data sets. The EM algorithm described here also does not construct the covariance matrix explicitly. Instead, the most computationally demanding steps are those involving sums over the data set that are $\mathcal{O}(NDM)$. For large D, and $M \ll D$, this can be a significant saving compared to $\mathcal{O}(ND^2)$ and can offset the iterative nature of the EM algorithm.

Note that this EM algorithm can be implemented in an online form in which each D-dimensional data point is read in and processed and then discarded before the next data point is considered. To see this, note that the quantities evaluated in the E step (an M-dimensional vector and an $M \times M$ matrix) can be computed for each data point separately, and in the M step we need to accumulate sums over data points, which we can do incrementally. This approach can be advantageous if both N and D are large.

Because we now have a fully probabilistic model for PCA, we can deal with missing data, provided that it is *missing at random*, in other words that the process that determines which values are missing does not depend on the values of any observed or unobserved variables. Such data sets can be handled by marginalizing over the distribution of the unobserved variables, and the resulting likelihood function can be maximized using the EM algorithm.

Exercise 16.22

16.3.2 EM for PCA

Another elegant feature of the EM approach is that we can take the limit $\sigma^2 \to 0$, corresponding to standard PCA, and still obtain a valid EM-like algorithm (Roweis,

Telegram: @uni_k

www.konkur.in

1998). From (16.67), we see that the only quantity we need to compute in the E step is $\mathbb{E}[\mathbf{z}_n]$. Furthermore, the M step is simplified because $\mathbf{M} = \mathbf{W}^{\mathrm{T}}\mathbf{W}$. To emphasize the simplicity of the algorithm, let us define $\widetilde{\mathbf{X}}$ to be a matrix of size $N \times D$ whose nth row is given by the vector $\mathbf{x}_n - \overline{\mathbf{x}}$ and similarly define $\boldsymbol{\Omega}$ to be a matrix of size $M \times N$ whose nth column is given by the vector $\mathbb{E}[\mathbf{z}_n]$. The E step (16.66) of the EM algorithm for PCA then becomes

$$\boldsymbol{\Omega} = (\mathbf{W}_{\mathrm{old}}^{\mathrm{T}}\mathbf{W}_{\mathrm{old}})^{-1}\mathbf{W}_{\mathrm{old}}^{\mathrm{T}}\widetilde{\mathbf{X}}^{\mathrm{T}} \tag{16.70}$$

and the M step (16.68) takes the form

$$\mathbf{W}_{\mathrm{new}} = \widetilde{\mathbf{X}}^{\mathrm{T}}\boldsymbol{\Omega}^{\mathrm{T}}(\boldsymbol{\Omega}\boldsymbol{\Omega}^{\mathrm{T}})^{-1}. \tag{16.71}$$

Again these can be implemented in an online form. These equations have a simple interpretation as follows. From our earlier discussion, we see that the E step involves an orthogonal projection of the data points onto the current estimate for the principal subspace. Correspondingly, the M step represents a re-estimation of the principal subspace to minimize the reconstruction error in which the projections are fixed.

Exercise 16.23

We can give a simple physical analogy for this EM algorithm, which is easily visualized for $D = 2$ and $M = 1$. Consider a collection of data points in two dimensions, and let the one-dimensional principal subspace be represented by a solid rod. Now attach each data point to the rod via a spring obeying Hooke's law (force is proportional to the length of the spring and therefore stored energy is proportional to the square of the spring's length). In the E step, we keep the rod fixed and allow the attachment points to slide up and down the rod so as to minimize the energy. This causes each attachment point (independently) to position itself at the orthogonal projection of the corresponding data point onto the rod. In the M step, we keep the attachment points fixed and then release the rod and allow it to move to the minimum energy position. The E step and M step are then repeated until a suitable convergence criterion is satisfied, as is illustrated in Figure 16.10.

16.3.3 EM for factor analysis

Section 16.2.4

We can determine the parameters $\boldsymbol{\mu}$, $\mathbf{W}$, and $\boldsymbol{\Psi}$ in a factor analysis model by maximum likelihood. The solution for $\boldsymbol{\mu}$ is again given by the sample mean. However, unlike probabilistic PCA, there is no longer a closed-form maximum likelihood solution for $\mathbf{W}$, which must therefore be found iteratively. Because factor analysis is a latent-variable model, this can be done using an EM algorithm (Rubin and Thayer, 1982) that is analogous to the one used for probabilistic PCA. Specifically, the E-step equations are given by

Exercise 16.24

$$\mathbb{E}[\mathbf{z}_n] = \mathbf{G}\mathbf{W}^{\mathrm{T}}\boldsymbol{\Psi}^{-1}(\mathbf{x}_n - \overline{\mathbf{x}}) \tag{16.72}$$

$$\mathbb{E}[\mathbf{z}_n\mathbf{z}_n^{\mathrm{T}}] = \mathbf{G} + \mathbb{E}[\mathbf{z}_n]\mathbb{E}[\mathbf{z}_n]^{\mathrm{T}} \tag{16.73}$$

where we have defined

$$\mathbf{G} = (\mathbf{I} + \mathbf{W}^{\mathrm{T}}\boldsymbol{\Psi}^{-1}\mathbf{W})^{-1}. \tag{16.74}$$

Telegram: @uni_k

www.konkur.in

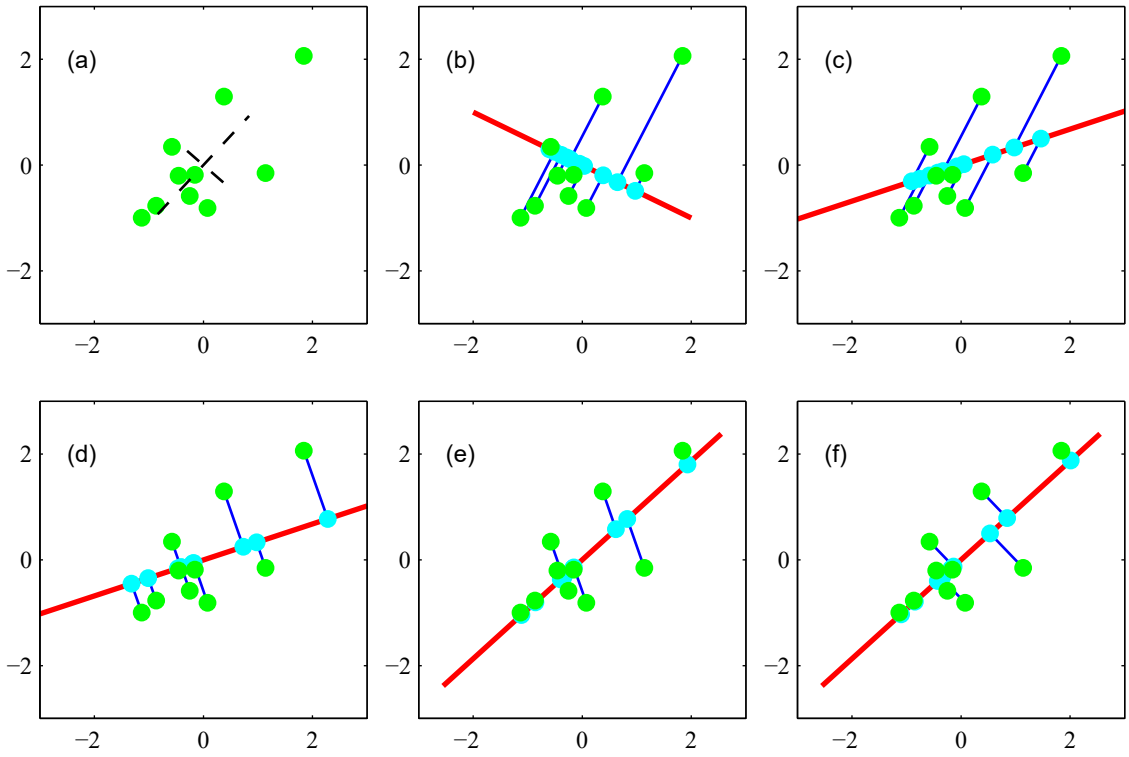


Figure 16.10 Synthetic data illustrating the EM algorithm for PCA defined by (16.70) and (16.71). (a) A set of data points shown in green, together with the true principal components (shown as eigenvectors scaled by the square roots of the eigenvalues). (b) Initial configuration of the principal subspace defined by $\mathbf{W}$, shown in red, together with the projections of the latent points $\mathbf{Z}$ into the data space, given by $\mathbf{Z}\mathbf{W}^{\mathrm{T}}$, shown in cyan. (c) After one M step, $\mathbf{W}$ has been updated with $\mathbf{Z}$ held fixed. (d) After the successive E step, the values of $\mathbf{Z}$ have been updated, giving orthogonal projections, with $\mathbf{W}$ held fixed. (e) After the second M step. (f) The converged solution.

Note that this is expressed in a form that involves inversion of matrices of size $M \times M$ rather than $D \times D$ (except for the $D \times D$ diagonal matrix $\boldsymbol{\Psi}$ whose inverse is trivial to compute in $\mathcal{O}(D)$ steps), which is convenient because often $M \ll D$. Similarly, the M-step equations take the form

Exercise 16.25

$$\mathbf{W}_{\text{new}} = \left[\sum_{n=1}^{N}(\mathbf{x}_n - \overline{\mathbf{x}})\mathbb{E}[\mathbf{z}_n]^{\mathrm{T}}\right]\left[\sum_{n=1}^{N}\mathbb{E}[\mathbf{z}_n\mathbf{z}_n^{\mathrm{T}}]\right]^{-1} \tag{16.75}$$

$$\boldsymbol{\Psi}_{\text{new}} = \operatorname{diag}\left\{\mathbf{S} - \mathbf{W}_{\text{new}}\frac{1}{N}\sum_{n=1}^{N}\mathbb{E}[\mathbf{z}_n](\mathbf{x}_n - \overline{\mathbf{x}})^{\mathrm{T}}\right\} \tag{16.76}$$

where the diag operator sets all the non-diagonal elements of a matrix to zero.

Telegram: @uni_k

www.konkur.in

16.4. Nonlinear Latent Variable Models

So far in this chapter we have focused on latent variable models based on linear transformations from the latent space to the data space. It is natural to ask whether we can use the flexibility of deep neural networks to represent more complex transformations, while exploiting the learning ability of deep networks to allow the resulting distribution to be fitted to a data set. Consider a simple distribution over a vector variable $\mathbf{z}$, for example a Gaussian of the form

$$p_{\mathbf{z}}(\mathbf{z}) = \mathcal{N}(\mathbf{z}|\mathbf{0}, \mathbf{I}). \tag{16.77}$$

Now suppose we transform $\mathbf{z}$ using a function $\mathbf{x} = \mathbf{g}(\mathbf{z}, \mathbf{w})$ given by a deep neural network, where $\mathbf{w}$ represents the weights and biases. The combination of the distribution over $\mathbf{z}$ together with the neural network defines a distribution over $\mathbf{x}$. Sampling from such a model is straightforward because we can generate samples from $p_{\mathbf{z}}(\mathbf{z})$ and then transform each of them using the neural network function to give corresponding samples of $\mathbf{x}$. This is an efficient process since it does not involve iteration.

To learn $\mathbf{g}(\mathbf{z}, \mathbf{w})$ from data, consider how to evaluate the likelihood function $p(\mathbf{x}|\mathbf{w})$. The distribution of $\mathbf{x}$ is given by the change of variables formula for densities:

Section 2.4

$$p_{\mathbf{x}}(\mathbf{x}) = p_{\mathbf{z}}(\mathbf{z}(\mathbf{x}))\,|\det \mathbf{J}(\mathbf{x})| \tag{16.78}$$

where $\mathbf{J}$ is the Jacobian matrix of partial derivatives whose elements are given by

$$J_{ij}(\mathbf{x}) = \frac{\partial z_i}{\partial x_j}. \tag{16.79}$$

To evaluate the distribution $p_{\mathbf{z}}(\mathbf{z}(\mathbf{x}))$ on the right-hand side of (16.78) for a given data vector $\mathbf{x}$ and to evaluate the Jacobian matrix in (16.79) for that same value of $\mathbf{x}$, we need the inverse $\mathbf{z} = \mathbf{g}^{-1}(\mathbf{x}, \mathbf{w})$ of the neural network function. For most neural networks this inverse will not be well defined. For example, the network may represent a many-to-one function in which multiple different input values map to the same output value, in which case the change of variable formula does not give a well-defined density. Moreover, if the dimensionality of the latent space is different from that of the data space then the transformation will not be invertible.

One approach is to restrict our attention to functions $\mathbf{g}(\mathbf{z}, \mathbf{w})$ that are invertible, which requires that $\mathbf{z}$ and $\mathbf{x}$ have the same dimensionality. We will explore this approach in more detail when we introduce the technique of *normalizing flows*.

Chapter 18

16.4.1 Nonlinear manifolds

Requiring that the latent and data spaces have the same number of dimensions is a significant limitation. Consider the situation in which $\mathbf{z}$ has dimensionality M and $\mathbf{x}$ has dimensionality D, where $M < D$. In this case the distribution over $\mathbf{x}$ is confined to a *manifold*, or subspace, of dimensionality M, as illustrated in Figure 16.11. Low-dimensional manifolds arise in many machine learning applications,

Telegram: @uni_k

www.konkur.in

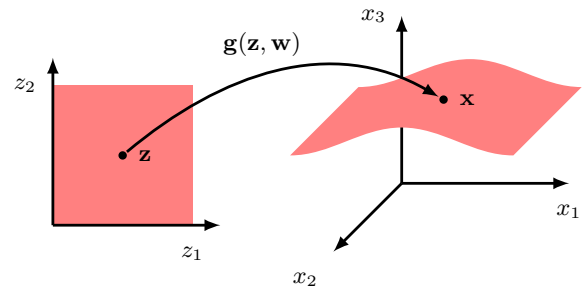


Figure 16.11 Illustration of a mapping from a two-dimensional latent space $\mathbf{z} = (z_1, z_2)$ to a three-dimensional data space $\mathbf{x} = (x_1, x_2, x_3)$ using a nonlinear function $\mathbf{x} = \mathbf{g}(\mathbf{z}, \mathbf{w})$ represented by a neural network with parameter vector $\mathbf{w}$.

Section 6.1.4

for example when modelling the distribution of natural images. Nonlinear latent-variable models can be very useful in modelling such data because they express the strong inductive bias that the data does not 'fill' the data space but is confined to a manifold, although the shape and dimensionality of this manifold are typically not known in advance.

However, one problem with this framework is that it assigns zero probability density to any data vector that does not lie *exactly* on the manifold, which is a problem for gradient-based learning since the likelihood function will be zero at each of the data points and constant for small changes in $\mathbf{w}$, for any realistic data set. To address this, we follow the approach used previously with regression and classification problems and define a conditional distribution across the entire data space, whose parameters are given by the output of the neural network. If, for example, $\mathbf{x}$ comprises a vector of continuous variables then we can choose the conditional distribution to be a Gaussian:

$$p(\mathbf{x}|\mathbf{z}, \mathbf{w}) = \mathcal{N}(\mathbf{x}|\mathbf{g}(\mathbf{z}, \mathbf{w}), \sigma^2\mathbf{I}) \tag{16.80}$$

in which the neural network $\mathbf{g}(\mathbf{z}, \mathbf{w})$ has linear output-unit activation functions, and $\mathbf{g} \in \mathbb{R}^D$. The generative model is specified by the latent distribution over $\mathbf{z}$ together with the conditional distribution over $\mathbf{x}$, and can be represented by the simple graphical model shown in Figure 16.12.

Note that it is straightforward, and computationally efficient, to draw independent samples from this distribution. We first draw a sample from the Gaussian distribution (16.77) using standard methods. Next, we use this value as input to the neural network, giving an output value $\mathbf{g}(\mathbf{z}, \mathbf{w})$. Finally, we draw a sample from a Gaussian distribution with mean $\mathbf{g}(\mathbf{z}, \mathbf{w})$ and covariance $\sigma^2\mathbf{I}$, as defined by (16.80). This three-step process can then be repeated to generate multiple independent samples.

Section 14.1.2

The combination of a latent-variable distribution $p(\mathbf{z})$ and a conditional distri-



Figure 16.12 Graphical model representing the distribution given by (16.77) and (16.80), which together define a joint distribution $p(\mathbf{x}, \mathbf{z}) = p(\mathbf{x}|\mathbf{z})p(\mathbf{z})$.

Telegram: @uni_k

www.konkur.in

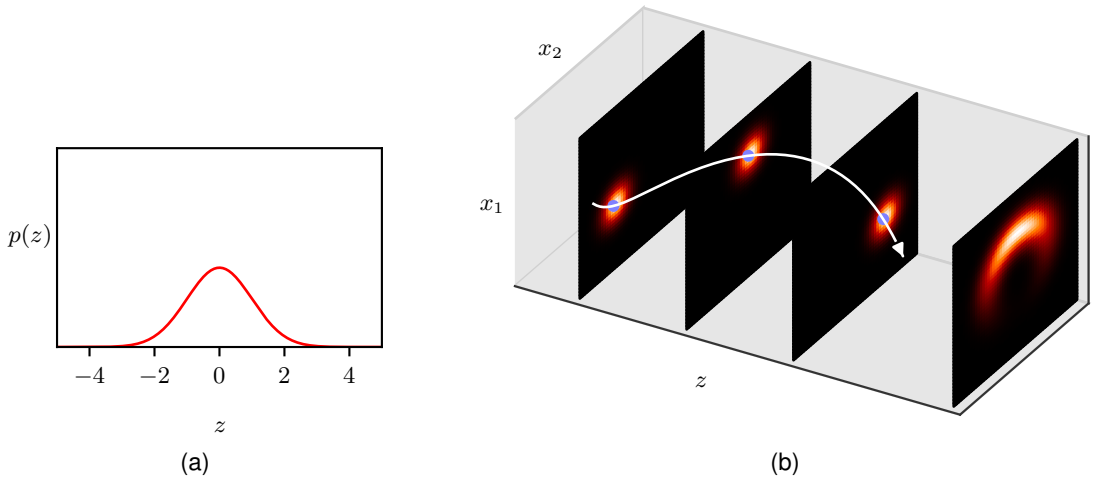


Figure 16.13 Illustration of a nonlinear latent-variable model for a one-dimensional latent space and a two-dimensional data space. (a) The prior distribution in latent space is given by a zero-mean unit-variance Gaussian distribution. (b) The three left-most plots show examples of the Gaussian conditional distribution $p(\mathbf{x}|z)$ for different values of z, whereas the right-most plot shows the marginal distribution $p(\mathbf{x})$. The nonlinear function $\mathbf{g}(z)$, which defines the mean of the conditional distribution, is given by $g_1(z) = \sin(z)$, $g_2(z) = \cos(z)$, and, therefore, traces out a circle in data space. The standard deviation of the conditional distribution is given by $\sigma = 0.3$. [Based on Prince (2020) with permission.]

bution $p(\mathbf{x}|\mathbf{z})$ defines a marginal distribution over the data space given by

$$p(\mathbf{x}) = \int p(\mathbf{z})p(\mathbf{x}|\mathbf{z})\,\mathrm{d}\mathbf{z}. \tag{16.81}$$

We illustrate this using a simple example involving a one-dimensional latent space and a two-dimensional data space in Figure 16.13.

16.4.2 Likelihood function

We have seen that it is easy to draw samples from this nonlinear latent-variable model. Now suppose we wish to fit the model to an observed data set by maximizing the likelihood function. The likelihood is obtained from the product and sum rules of probability by integrating over $\mathbf{z}$:

$$\begin{aligned} p(\mathbf{x}|\mathbf{w}) &= \int p(\mathbf{x}|\mathbf{z},\mathbf{w})p(\mathbf{z})\,\mathrm{d}\mathbf{z} \\ &= \int \mathcal{N}(\mathbf{x}|\mathbf{g}(\mathbf{z},\mathbf{w}),\sigma^2\mathbf{I})\mathcal{N}(\mathbf{z}|\mathbf{0},\mathbf{I})\,\mathrm{d}\mathbf{z}. \end{aligned} \tag{16.82}$$

Although both distributions inside the integral are Gaussian, the integral is analytically intractable due to the highly nonlinear function $\mathbf{g}(\mathbf{z},\mathbf{w})$ defined by the neural network.

Telegram: @uni_k

www.konkur.in



Figure 16.14 Three example images of handwritten digits, illustrating why sampling from the latent space to evaluate the likelihood function requires large numbers of samples. (a) shows the original image, (b) shows a corrupted image with part of the stroke removed, and (c) shows the original image shifted by half a pixel down and half a pixel to the right. Image (b) is closer to (a) in terms of likelihood, even though image (c) is much closer to (a) in appearance. [From Doersch (2016) with permission.]

One approach for evaluating the likelihood function would be to draw samples from the latent space distribution and use these to approximate (16.82) by

$$p(\mathbf{x}|\mathbf{w}) \simeq \frac{1}{K}\sum_{i=1}^{K} p(\mathbf{x}|\mathbf{z}_i, \mathbf{w}) \tag{16.83}$$

where $\mathbf{z}_i \sim p(\mathbf{z})$. This expresses the distribution over $\mathbf{z}$ as a mixture of Gaussians with fixed mixing coefficients given by $1/K$, and in the limit of an infinite number of samples, this gives the true likelihood function. However, the value of K needed for effective training will typically be far too high to be practical. To see why, consider the three images of handwritten digits shown in Figure 16.14, and suppose that image (a) represents the vector $\mathbf{x}$ for which we wish to evaluate the likelihood function. If a trained model generated image (b), we would consider this a poor model as this image is not a good representation of a digit '2', and so this should be assigned a much lower likelihood. Conversely, image (c), which was obtained by shifting the digit in (a) down and to the right by half a pixel, is a good example of a digit '2' and should therefore have a high likelihood. Since the distribution (16.80) is Gaussian, the likelihood function is proportional to the exponential of the negative squared distance between the output of the network and the data vector $\mathbf{x}$. However, the squared distance between (a) and (b) is 0.0387 whereas the squared distance between (a) and (c) is 0.2693. So if the variance parameter σ^2 is set to a sufficiently small value that image (b) has low likelihood, then image (c) will have an even lower likelihood. Even if the model is good at generating digits, we would have to consider extremely large numbers of samples for $\mathbf{z}$ before seeing a digit that is sufficiently close to (a). We therefore seek more sophisticated techniques for training nonlinear latent variable models that can be used in practical applications. Before outlining such methods, we first discuss briefly some considerations regarding discrete data spaces.

Telegram: @uni_k

www.konkur.in

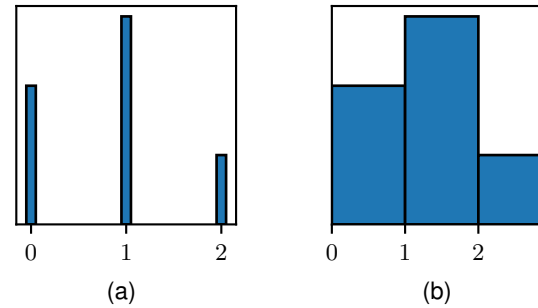


Figure 16.15 Schematic illustration of dequantization, showing (a) a discrete distribution over a single variable and (b) an associated dequantized continuous distribution.

16.4.3 Discrete data

If the observed data set comprises independent binary variables then we can use a conditional distribution of the form

$$p(\mathbf{x}|\mathbf{z},\mathbf{w}) = \prod_{i=1}^{D} g_i(\mathbf{z},\mathbf{w})^{x_i} \left(1 - g_i(\mathbf{z},\mathbf{w})\right)^{1-x_i} \tag{16.84}$$

where $g_i(\mathbf{z},\mathbf{w}) = \sigma(a_i(\mathbf{z},\mathbf{w}))$ represents the activation of output unit i, the activation function $\sigma(\cdot)$ is given by the logistic sigmoid, and $a_i(\mathbf{z},\mathbf{w})$ is the pre-activation for output unit i. Similarly, for one-hot encoded categorical variables, we can use a multinomial distribution:

$$p(\mathbf{x}|\mathbf{z},\mathbf{w}) = \prod_{i=1}^{D} g_i(\mathbf{z},\mathbf{w})^{x_i} \tag{16.85}$$

where

$$g_i(\mathbf{z},\mathbf{w}) = \frac{\exp(a_i(\mathbf{z},\mathbf{w}))}{\sum_j \exp(a_j(\mathbf{z},\mathbf{w}))} \tag{16.86}$$

is the softmax activation function. We can also consider combinations of discrete and continuous variables by forming the product of the associated conditional distributions.

In practice, continuous variables are represented with discrete values, for example in images, the red, green, and blue channel intensities might be expressed using 8-bit numbers representing the values $\{0, \ldots, 255\}$. This can cause problems when we employ highly flexible models based on deep neural networks, as the likelihood function can go to zero if the density collapses onto one or more of the discrete values. The problem can be resolved using a technique called *dequantization*, which involves adding noise to the variables, typically drawn from a uniform distribution over the region between successive discrete values, as shown in Figure 16.15. A training set is dequantized by replacing each observed value with a sample drawn randomly from the associated continuous distribution associated with that discrete value, and this makes it less likely that the model will discover a pathological solution.

Telegram: @uni_k

www.konkur.in

16.4.4 Four approaches to generative modelling

We have seen that nonlinear latent-variable models based on deep neural networks offer a highly flexible framework for building generative models. Due to the universality of the neural network transformation, such models are capable, in principle, of approximating essentially any desired distribution to high accuracy. Moreover, such models offer the potential, once trained, to generate samples from the distribution in using an efficient, non-iterative process. However, we have also identified some challenges associated with training such models that force us to develop more sophisticated techniques than those needed for linear models. Many such methods have been proposed, each having their own strengths and limitations. These can be broadly grouped into four approaches, as follows.

Chapter 17

With *generative adversarial networks*, or GANs, we relax the requirement for the network mapping to be invertible, thereby allowing the latent space to have a lower dimensionality than the data space. We also abandon the concept of a likelihood function and instead introduce a second neural network whose function is to provide a training signal for the generative network. Due to the absence of a well-defined likelihood function, the training procedure may be brittle, but once trained it is straightforward to generate samples from the model, and the results can be of high quality.

Chapter 19

The framework of *variational autoencoders*, or VAEs, also uses a second neural network whose role is to approximate the posterior distribution over the latent variables, thereby allowing an approximation to the likelihood function to be evaluated. Training is more robust than with GANs, and sampling from the trained model is straightforward, although it can be harder to obtain the highest quality results.

Chapter 18

In *normalizing flows*, we set the dimensionality of the latent space to be equal to that of the data space and then modify the generative neural network so that it becomes invertible. The requirement that the network is invertible restricts its functional form but it allows the likelihood function to be evaluated without approximation and it also allows for efficient sampling.

Chapter 20

Finally, *diffusion models* use a network that learns to transform a sample from the prior distribution into a sample from the data distribution through a sequence of denoising steps. This leads to state-of-the-art performance in many applications, although the cost of sampling can be high due to the multiple denoising passes through the network.

We explore these approaches in detail in the final four chapters of this book.

Exercises

16.1 ($\star\star$) In this exercise, we use proof by induction to show that the linear projection onto an M-dimensional subspace that maximizes the variance of the projected data is defined by the M eigenvectors of the data covariance matrix $\mathbf{S}$, given by (16.3), corresponding to the M largest eigenvalues. In Section 16.1, this result was proven for $M = 1$. Now suppose the result holds for some general value of M and show that it consequently holds for dimensionality $M + 1$. To do this, first set the derivative of the variance of the projected data with respect to a vector $\mathbf{u}_{M+1}$ defining the new

Telegram: @uni_k

www.konkur.in

direction in data space equal to zero. This should be done subject to the constraints that $\mathbf{u}_{M+1}$ are orthogonal to the existing vectors $\mathbf{u}_1, \ldots, \mathbf{u}_M$, and also that it is normalized to unit length. Use Lagrange multipliers to enforce these constraints. Then make use of the orthonormality properties of the vectors $\mathbf{u}_1, \ldots, \mathbf{u}_M$ to show that the new vector $\mathbf{u}_{M+1}$ is an eigenvector of $\mathbf{S}$. Finally, show that the variance is maximized if the eigenvector is chosen to be the one corresponding to eigenvalue λ_{M+1} where the eigenvalues have been ordered in decreasing value.

Appendix C

16.2 ($\star\star$) Show that the minimum value of the PCA error measure J given by (16.15) with respect to the $\mathbf{u}_i$, subject to the orthonormality constraints (16.7), is obtained when the $\mathbf{u}_i$ are eigenvectors of the data covariance matrix $\mathbf{S}$. To do this, introduce a matrix $\mathbf{H}$ of Lagrange multipliers, one for each constraint, so that the modified error measure, in matrix notation reads

$$\widetilde{J} = \mathrm{Tr}\left\{\widehat{\mathbf{U}}^{\mathrm{T}}\mathbf{S}\widehat{\mathbf{U}}\right\} + \mathrm{Tr}\left\{\mathbf{H}(\mathbf{I} - \widehat{\mathbf{U}}^{\mathrm{T}}\widehat{\mathbf{U}})\right\} \tag{16.87}$$

where $\widehat{\mathbf{U}}$ is a matrix of dimension $D \times (D - M)$ whose columns are given by $\mathbf{u}_i$. Now minimize $\widetilde{J}$ with respect to $\widehat{\mathbf{U}}$ and show that the solution satisfies $\mathbf{S}\widehat{\mathbf{U}} = \widehat{\mathbf{U}}\mathbf{H}$. Clearly, one possible solution is that the columns of $\widehat{\mathbf{U}}$ are eigenvectors of $\mathbf{S}$, in which case $\mathbf{H}$ is a diagonal matrix containing the corresponding eigenvalues. To obtain the general solution, show that $\mathbf{H}$ can be assumed to be a symmetric matrix, and by using its eigenvector expansion, show that the general solution to $\mathbf{S}\widehat{\mathbf{U}} = \widehat{\mathbf{U}}\mathbf{H}$ gives the same value for $\widetilde{J}$ as the specific solution in which the columns of $\widehat{\mathbf{U}}$ are the eigenvectors of $\mathbf{S}$. Because these solutions are all equivalent, it is convenient to choose the eigenvector solution.

16.3 ($\star$) Verify that the eigenvectors defined by (16.30) are normalized to unit length, assuming that the eigenvectors $\mathbf{v}_i$ have unit length.

16.4 ($\star$) Suppose we replace the zero-mean, unit-covariance latent space distribution (16.31) in the probabilistic PCA model by a general Gaussian distribution of the form $\mathcal{N}(\mathbf{z}|\mathbf{m}, \boldsymbol{\Sigma})$. By redefining the parameters of the model, show that this leads to an identical model for the marginal distribution $p(\mathbf{x})$ over the observed variables for any valid choice of $\mathbf{m}$ and $\boldsymbol{\Sigma}$.

16.5 ($\star\star$) Let $\mathbf{x}$ be a D-dimensional random variable having a Gaussian distribution given by $\mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma})$, and consider the M-dimensional random variable given by $\mathbf{y} = \mathbf{A}\mathbf{x} + \mathbf{b}$ where $\mathbf{A}$ is an $M \times D$ matrix. Show that $\mathbf{y}$ also has a Gaussian distribution, and find expressions for its mean and covariance. Discuss the form of this Gaussian distribution for $M < D$, for $M = D$, and for $M > D$.

16.6 ($\star\star$) By making use of the results (2.122) and (2.123) for the mean and covariance of a general distribution, derive the result (16.35) for the marginal distribution $p(\mathbf{x})$ in the probabilistic PCA model.

16.7 ($\star$) Draw a directed probabilistic graph for the probabilistic PCA model described in Section 16.2 in which the components of the observed variable $\mathbf{x}$ are shown explicitly

Telegram: @uni_k

www.konkur.in

as separate nodes. Hence, verify that the probabilistic PCA model has the same independence structure as the naive Bayes model discussed in Section 11.2.3.

16.8 ($\star\star$) By making use of the result (3.100), show that the posterior distribution $p(\mathbf{z}|\mathbf{x})$ for the probabilistic PCA model is given by (16.43).

16.9 ($\star$) Verify that maximizing the log likelihood (16.44) for the probabilistic PCA model with respect to the parameter $\boldsymbol{\mu}$ gives the result $\boldsymbol{\mu}_{\mathrm{ML}} = \overline{\mathbf{x}}$ where $\overline{\mathbf{x}}$ is the mean of the data vectors.

16.10 ($\star\star$) By evaluating the second derivatives of the log likelihood function (16.44) for the probabilistic PCA model with respect to the parameter $\boldsymbol{\mu}$, show that the stationary point $\boldsymbol{\mu}_{\mathrm{ML}} = \overline{\mathbf{x}}$ represents the unique maximum.

16.11 ($\star\star$) Show that in the limit $\sigma^2 \to 0$, the posterior mean for the probabilistic PCA model becomes an orthogonal projection onto the principal subspace, as in conventional PCA.

16.12 ($\star\star$) For $\sigma^2 > 0$ show that the posterior mean in the probabilistic PCA model is shifted towards the origin relative to the orthogonal projection.

16.13 ($\star\star$) Show that the optimal reconstruction of a data point under probabilistic PCA, according to the least-squares projection cost of conventional PCA, is given by

$$\widetilde{\mathbf{x}} = \mathbf{W}_{\mathrm{ML}}(\mathbf{W}_{\mathrm{ML}}^{\mathrm{T}}\mathbf{W}_{\mathrm{ML}})^{-1}\mathbf{M}\mathbb{E}[\mathbf{z}|\mathbf{x}]. \tag{16.88}$$

16.14 ($\star$) The number of independent parameters in the covariance matrix for a probabilistic PCA model with an M-dimensional latent space and a D-dimensional data space is given by (16.52). Verify that for $M = D - 1$, the number of independent parameters is the same as in a general covariance Gaussian, whereas for $M = 0$ it is the same as for a Gaussian with an isotropic covariance.

16.15 ($\star$) Derive an expression for the number of independent parameters in the factor analysis model described in Section 16.2.4.

16.16 ($\star\star$) Show that the factor analysis model described in Section 16.2.4 is invariant under rotations of the latent space coordinates.

16.17 ($\star\star$) Consider a linear-Gaussian latent-variable model having a latent space distribution $p(\mathbf{z}) = \mathcal{N}(\mathbf{x}|\mathbf{0}, \mathbf{I})$ and a conditional distribution for the observed variable $p(\mathbf{x}|\mathbf{z}) = \mathcal{N}(\mathbf{x}|\mathbf{W}\mathbf{z} + \boldsymbol{\mu}, \boldsymbol{\Phi})$ where $\boldsymbol{\Phi}$ is an arbitrary symmetric positive-definite noise covariance matrix. Now suppose that we make a non-singular linear transformation of the data variables $\mathbf{x} \to \mathbf{A}\mathbf{x}$, where $\mathbf{A}$ is a $D \times D$ matrix. If $\boldsymbol{\mu}_{\mathrm{ML}}$, $\mathbf{W}_{\mathrm{ML}}$, and $\boldsymbol{\Phi}_{\mathrm{ML}}$ represent the maximum likelihood solution corresponding to the original un-transformed data, show that $\mathbf{A}\boldsymbol{\mu}_{\mathrm{ML}}$, $\mathbf{A}\mathbf{W}_{\mathrm{ML}}$, and $\mathbf{A}\boldsymbol{\Phi}_{\mathrm{ML}}\mathbf{A}^{\mathrm{T}}$ represent the corresponding maximum likelihood solution for the transformed data set. Finally, show that the form of the model is preserved in two cases: (i) $\mathbf{A}$ is a diagonal matrix and $\boldsymbol{\Phi}$ is a diagonal matrix. This corresponds to factor analysis. The transformed $\boldsymbol{\Phi}$ remains diagonal, and hence factor analysis is *covariant* under component-wise

Telegram: @uni_k

www.konkur.in

re-scaling of the data variables; (ii) $\mathbf{A}$ is orthogonal and $\mathbf{\Phi}$ is proportional to the unit matrix so that $\mathbf{\Phi} = \sigma^2\mathbf{I}$. This corresponds to probabilistic PCA. The transformed $\mathbf{\Phi}$ matrix remains proportional to the unit matrix, and hence probabilistic PCA is covariant under a rotation of the axes of the data space, as is the case for conventional PCA.

16.18 ($\star$) Verify that the log likelihood function for a model with continuous latent variables can be written as the sum of two terms in the form (16.57) in which the terms are defined by (16.58) and (16.59). This can be done by using the product rule of probability in the form

$$p(\mathbf{x}, \mathbf{z}|\mathbf{w}) = p(\mathbf{z}|\mathbf{x}, \mathbf{w})p(\mathbf{x}|\mathbf{w}) \tag{16.89}$$

and then substituting for $p(\mathbf{x}, \mathbf{z}|\mathbf{w})$ in (16.58).

16.19 ($\star$) Show that, for a set of i.i.d. data, the evidence lower bound (ELBO) takes the form (16.63).

16.20 ($\star\star$) Draw a directed probabilistic graphical model representing a discrete mixture of probabilistic PCA models in which each PCA model has its own values of $\mathbf{W}$, $\boldsymbol{\mu}$, and σ^2. Now draw a modified graph in which these parameter values are shared between the components of the mixture.

16.21 ($\star\star$) Derive the M-step equations (16.68) and (16.69) for the probabilistic PCA model by maximizing the expected complete-data log likelihood function given by (16.65).

16.22 ($\star\star\star$) One benefit of a probabilistic formulation of principal component analysis is that it can be applied to a data set in which some of the values are missing, provided they are missing at random. Derive the EM algorithm for maximizing the likelihood function for the probabilistic PCA model in this situation. Note that the $\{\mathbf{z}_n\}$, as well as the missing data values that are components of the vectors $\{\mathbf{x}_n\}$, are now latent variables. Show that in the special case in which all the data values are observed, this reduces to the EM algorithm for probabilistic PCA derived in Section 16.3.2.

16.23 ($\star\star$) Let $\mathbf{W}$ be a $D \times M$ matrix whose columns define a linear subspace of dimensionality M embedded within a data space of dimensionality D, and let $\boldsymbol{\mu}$ be a D-dimensional vector. Given a data set $\{\mathbf{x}_n\}$ where $n = 1, \ldots, N$, we can approximate the data points using a linear mapping from a set of M-dimensional vectors $\{\mathbf{z}_n\}$, so that $\mathbf{x}_n$ is approximated by $\mathbf{W}\mathbf{z}_n + \boldsymbol{\mu}$. The associated sum-of-squares reconstruction cost is given by

$$J = \sum_{n=1}^{N} \|\mathbf{x}_n - \boldsymbol{\mu} - \mathbf{W}\mathbf{z}_n\|^2. \tag{16.90}$$

First show that minimizing J with respect to $\boldsymbol{\mu}$ leads to an analogous expression with $\mathbf{x}_n$ and $\mathbf{z}_n$ replaced by zero-mean variables $\mathbf{x}_n - \overline{\mathbf{x}}$ and $\mathbf{z}_n - \overline{\mathbf{z}}$, respectively, where $\overline{\mathbf{x}}$ and $\overline{\mathbf{z}}$ denote sample means. Then show that minimizing J with respect to $\mathbf{z}_n$, where

Telegram: @uni_k

www.konkur.in

$\mathbf{W}$ is kept fixed, gives rise to the PCA E step (16.70), and that minimizing J with respect to $\mathbf{W}$, where $\{\mathbf{z}_n\}$ is kept fixed, gives rise to the PCA M step (16.71).

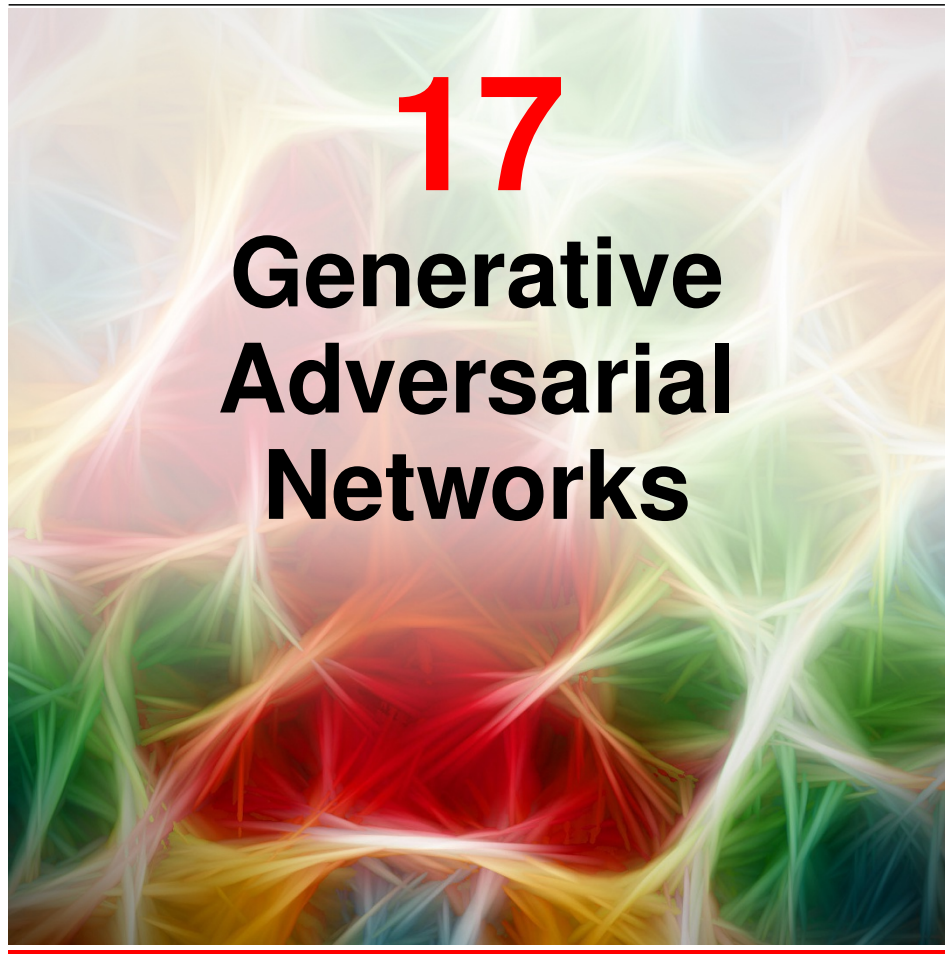
16.24 ($\star\star$) Derive the formulae (16.72) and (16.73) for the E step of the EM algorithm for factor analysis. Note that from the result of Exercise 16.26, the parameter $\boldsymbol{\mu}$ can be replaced by the sample mean $\overline{\mathbf{x}}$.

16.25 ($\star\star$) Write down an expression for the expected complete-data log likelihood function for the factor analysis model, and hence derive the corresponding M-step equations (16.75) and (16.76).

16.26 ($\star\star$) By considering second derivatives, show that the only stationary point of the log likelihood function for the factor analysis model discussed in Section 16.2.4 with respect to the parameter $\boldsymbol{\mu}$ is given by the sample mean defined by (16.1). Furthermore, show that this stationary point is a maximum.

Telegram: @uni_k

www.konkur.in



17 Generative Adversarial Networks

Generative models use machine learning algorithms to learn a distribution from a set of training data and then generate new examples from that distribution. For example, a generative model might be trained on images of animals and then used to generate new images of animals. We can think of such a generative model in terms of a distribution $p(\mathbf{x}|\mathbf{w})$ in which $\mathbf{x}$ is a vector in the data space, and $\mathbf{w}$ represent the learnable parameters of the model. In many cases we are interested in conditional generative models of the form $p(\mathbf{x}|\mathbf{c}, \mathbf{w})$ where $\mathbf{c}$ represents a vector of conditioning variables. In the case of our generative model for animal images, we may wish to specify that a generated image should be of a particular animal, such as a cat or a dog, specified by the value of $\mathbf{c}$.

For real-world applications such as image generation, the distributions are extremely complex, and consequently the introduction of deep learning has dramatically improved the performance of generative models. We have already encountered

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024
C. M. Bishop, H. Bishop, *Deep Learning*, https://doi.org/10.1007/978-3-031-45468-4_17

Telegram: @uni_k

www.konkur.in

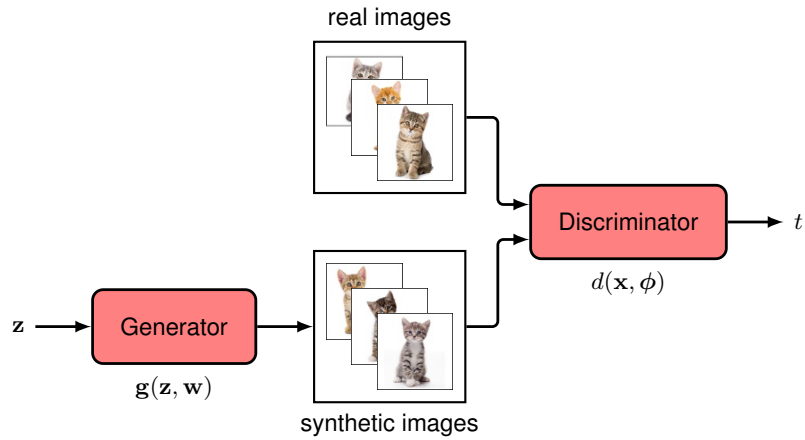


Figure 17.1 Schematic illustration of a GAN in which a discriminator neural network $d(\mathbf{x}, \boldsymbol{\phi})$ is trained to distinguish between real samples from the training set, in this case images of kittens, and synthetic samples produced by the generator network $\mathbf{g}(\mathbf{z}, \mathbf{w})$. The generator aims to maximize the error of the discriminator network by producing realistic images, whereas the discriminator network tries to minimize the same error by becoming better at distinguishing real from synthetic examples.

Chapter 12
Section 16.4.4

an important class of deep generative models when we discussed autoregressive large language models based on transformers. We have also outlined four important classes of generative model based on nonlinear latent variable models, and in this chapter we discuss the first of these, called generative adversarial networks. The other three approaches will be discussed in subsequent chapters.

17.1. Adversarial Training

Consider a generative model based on a nonlinear transformation from a latent space $\mathbf{z}$ to a data space $\mathbf{x}$. We introduce a latent distribution $p(\mathbf{z})$, which might take the form of a simple Gaussian

$$p(\mathbf{z}) = \mathcal{N}(\mathbf{z}|\mathbf{0}, \mathbf{I}), \tag{17.1}$$

along with with a nonlinear transformation $\mathbf{x} = \mathbf{g}(\mathbf{z}, \mathbf{w})$ defined by a deep neural network with learnable parameters $\mathbf{w}$ known as the *generator*. Together these implicitly define a distribution over $\mathbf{x}$, and our goal is to fit this distribution to a data set of training examples $\{\mathbf{x}_n\}$ where $n = 1, \ldots, N$. However, we cannot determine $\mathbf{w}$ by optimizing the likelihood function because this cannot, in general, be evaluated in closed form. The key idea of *generative adversarial networks*, or GANs, (Goodfellow *et al.*, 2014; Ruthotto and Haber, 2021) is to introduce a second *discriminator* network, which is trained jointly with the generator network and which provides a training signal to update the weights of the generator. This is illustrated in Figure 17.1.

Telegram: @uni_k

www.konkur.in

The goal of the discriminator network is to distinguish between real examples from the data set and synthetic, or 'fake', examples produced by the generator network, and it is trained by minimizing a conventional classification error function. Conversely, the goal of the generator network is to maximize this error by synthesizing examples from the same distribution as the training set. The generator and discriminator networks are therefore working against each other, hence the term 'adversarial'. This is an example of a *zero-sum game* in which any gain by one network represents a loss to the other. It allows the discriminator network to provide a training signal, which can be used to train the generator network, and this turns the unsupervised density modelling problem into a form of supervised learning.

17.1.1 Loss function

To make this precise, we define a binary target variable given by

$$t = 1, \quad \text{real data}, \tag{17.2}$$

$$t = 0, \quad \text{synthetic data}. \tag{17.3}$$

The discriminator network has a single output unit with a logistic-sigmoid activation function, whose output represents the probability that a data vector $\mathbf{x}$ is real:

$$P(t = 1) = d(\mathbf{x}, \boldsymbol{\phi}). \tag{17.4}$$

We train the discriminator network using the standard cross-entropy error function, which takes the form

$$E(\mathbf{w}, \boldsymbol{\phi}) = -\frac{1}{N} \sum_{n=1}^{N} \{t_n \ln d_n + (1 - t_n) \ln(1 - d_n)\} \tag{17.5}$$

where $d_n = d(\mathbf{x}_n, \boldsymbol{\phi})$ is the output of the discriminator network for input vector n, and we have normalized by the total number of data points. The training set comprises both real data examples denoted $\mathbf{x}_n$ and synthetic examples given by the output of the generator network $\mathbf{g}(\mathbf{z}_n, \mathbf{w})$ where $\mathbf{z}_n$ is a random sample from the latent space distribution $p(\mathbf{z})$. Since $t_n = 1$ for real examples and $t_n = 0$ for synthetic examples, we can write the error function (17.5) in the form

Section 1.2.4

$$E_{\text{GAN}}(\mathbf{w}, \boldsymbol{\phi}) = -\frac{1}{N_{\text{real}}} \sum_{n \in \text{real}} \ln d(\mathbf{x}_n, \boldsymbol{\phi}) - \frac{1}{N_{\text{synth}}} \sum_{n \in \text{synth}} \ln(1 - d(\mathbf{g}(\mathbf{z}_n, \mathbf{w}), \boldsymbol{\phi})) \tag{17.6}$$

where typically the number N_{real} of real data points is equal to the number N_{synth} of synthetic data points. This combination of generator and discriminator networks can be trained end-to-end using stochastic gradient descent with gradients evaluated using backpropagation. However, the unusual aspect is the adversarial training whereby the error is minimized with respect to $\boldsymbol{\phi}$ but *maximized* with respect to $\mathbf{w}$.

Chapter 7

Telegram: @uni_k

www.konkur.in

This maximization can be done using standard gradient-based methods with the sign of the gradient reversed, so that the parameter updates become

$$\Delta\boldsymbol{\phi} = -\lambda\nabla_{\boldsymbol{\phi}}E_n(\mathbf{w}, \boldsymbol{\phi}) \tag{17.7}$$

$$\Delta\mathbf{w} = \lambda\nabla_{\mathbf{w}}E_n(\mathbf{w}, \boldsymbol{\phi}) \tag{17.8}$$

where $E_n(\mathbf{w}, \boldsymbol{\phi})$ denotes the error defined for data point n or more generally for a mini-batch of data points. Note that the two terms in (17.7) and (17.8) have different signs since the discriminator is trained to decrease the error rate whereas the generator is trained to increase it. In practice, training alternates between updating the parameters of the generative network and updating those of the discriminative network, in each case taking just one gradient descent step using a mini-batch, after which a new set of synthetic samples is generated. If the generator succeeds in finding a perfect solution, then the discriminator network will be unable to tell the difference between the real and synthetic data and hence will always produce an output of 0.5. Once the GAN is trained, the discriminator network is discarded and the generator network can be used to synthesize new examples in the data space by sampling from the latent space and propagating those samples through the trained generator network. We can show that for generative and discriminative networks having unlimited flexibility, a fully optimized GAN will have a generative distribution that matches the data distribution exactly. Some impressive examples of synthetic face images generated by a GAN are shown in Figure 1.3.

Exercise 17.1

The GAN model discussed so far generates samples from the unconditional distribution $p(\mathbf{x})$. For example, it could generate synthetic images of dogs if it is trained on dog images. We can also create *conditional* GANs (Mirza and Osindero, 2014), which sample from a conditional distribution $p(\mathbf{x}|\mathbf{c})$ in which the conditioning vector $\mathbf{c}$ might, for example, represent different species of dog. To do this, both the generator and the discriminator network take $\mathbf{c}$ as an additional input, and labelled examples of images, comprising pairs $\{\mathbf{x}_n, \mathbf{c}_n\}$, are used for training. Once the GAN has been trained, images from a desired class can be generated by setting $\mathbf{c}$ to the corresponding class vector. Compared to training separate GANs for each class, this has the advantage that shared internal representations can be learned jointly across all classes, thereby making more efficient use of the data.

17.1.2 GAN training in practice

Although GANs can produce high quality results, they are not easy to train successfully due to the adversarial learning. Also, unlike standard error function minimization, there is no metric of progress because the objective can go up as well as down during training.

Exercise 17.2

One challenge that can arise is called *mode collapse*, in which the generator network weights adapt during training such that all latent-variable samples $\mathbf{z}$ are mapped to a subset of possible valid outputs. In extreme cases the output can correspond to just one, or a small number, of the output values $\mathbf{x}$. The discriminator then assigns the value 0.5 to these instances, and training ceases. For example, a GAN trained on handwritten digits might learn to generate only examples of the digit '3', and while

Telegram: @uni_k

www.konkur.in

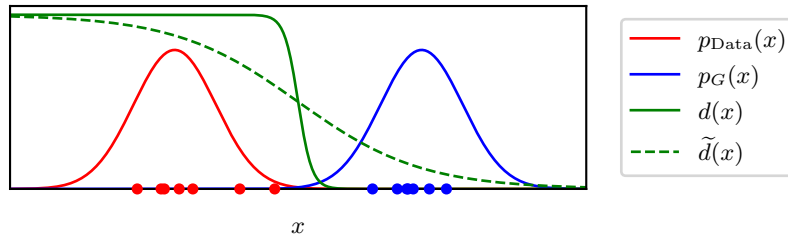


Figure 17.2 Conceptual illustration of why it can be difficult to train GANs, showing a simple one-dimensional data space x with the fixed, but unknown, data distribution $p_{\text{Data}}(x)$ and the initial generative distribution $p_{\text{G}}(x)$. The optimal discriminator function $d(x)$ has virtually zero gradient in the vicinity of either the training or synthetic data points, making learning very slow. A smoothed version $\widetilde{d}(x)$ of the discriminator function can lead to faster learning.

the discriminator is unable to distinguish these from genuine examples of the digit '3', it fails to recognize that the generator is not generating the full range of digits.

Insight into the difficulty of training GANs can be obtained by considering Figure 17.2, which shows a simple one§-dimensional data space x with samples $\{x_n\}$ drawn from the fixed, but unknown, data distribution $p_{\text{Data}}(x)$. Also shown is the initial generative distribution $p_{\text{G}}(x)$ together with samples drawn from this distribution. Because the data and generative distributions are so different, the optimal discriminator function $d(x)$ is easy to learn and has a very steep fall-off with virtually zero gradient in the vicinity of either the real or synthetic samples. Consider the second term in the GAN error function (17.6). Because $d(\mathbf{g}(\mathbf{z}, \mathbf{w}), \boldsymbol{\phi})$ is equal to zero across the region spanned by the generated samples, small changes in the parameters $\mathbf{w}$ of the generative network produce very little change in the output of the discriminator and so the gradients are small and learning proceeds slowly.

This can be addressed by using a smoothed version $\widetilde{d}(x)$ of the discriminator function, illustrated in Figure 17.2, thereby providing a stronger gradient to drive the training of the generator network. The *least-squares GAN* (Mao *et al.*, 2016) achieves smoothing by modifying the discriminator to produce a real-valued output rather than a probability in the range $(0, 1)$ and by replacing the cross-entropy error function with a sum-of-squares error function. Alternatively, the technique of *instance noise* (Sønderby *et al.*, 2016) adds Gaussian noise to both the real data and the synthetic samples, again leading to a smoother discriminator function.

Numerous other modifications to the GAN error function and training procedure have been proposed to improve training (Mescheder, Geiger, and Nowozin, 2018). One change that is often used is to replace the generative network term in the original error function

Telegram: @uni_k

www.konkur.in

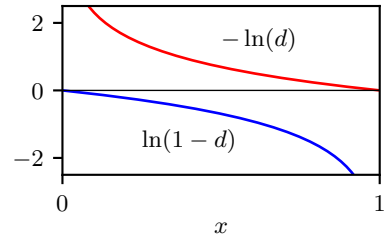


Figure 17.3 Plots of $-\ln(d)$ and $\ln(1-d)$ showing the very different behaviour of the gradients close to $d = 0$ and $d = 1$.

$$-\frac{1}{N_{\text{synth}}}\sum_{n\in\text{synth}}\ln(1-d(\mathbf{g}(\mathbf{z}_n,\mathbf{w}),\boldsymbol{\phi})) \tag{17.9}$$

with the modified form

$$\frac{1}{N_{\text{synth}}}\sum_{n\in\text{synth}}\ln d(\mathbf{g}(\mathbf{z}_n,\mathbf{w}),\boldsymbol{\phi}). \tag{17.10}$$

Although the first form minimizes the probability that the image is fake, the second version maximizes the probability that the image is real. The different properties of these two forms can be understood from Figure 17.3. When the generative distribution $p_{\text{G}}(x)$ is very different from the true data distribution $p_{\text{Data}}(x)$, the quantity $d(\mathbf{g}(\mathbf{z},\mathbf{w}))$ is close to zero, and hence the first form has a very small gradient, whereas the second form has a large gradient, leading to faster training.

A more direct way to ensure that the generator distribution $p_{\text{G}}(x)$ moves towards the data distribution $p_{\text{data}}(x)$ is to modify the error criterion to reflect how far apart the two distributions are in data space. This can be measured using the *Wasserstein distance*, also known as the *earth mover's distance*. Imagine the distribution $p_{\text{G}}(x)$ as a pile of earth that is transported in small increments to construct the distribution $p_{\text{data}}(x)$. The Wasserstein metric is the total amount of earth moved multiplied by the mean distance moved. Of the many ways of rearranging the pile of earth to build $p_{\text{data}}(x)$, the one that yields the smallest mean distance is the one used to define the metric. In practice, this cannot be implemented directly, and it is approximated by using a discriminator network that has real-valued outputs and then limiting the gradient $\nabla_x d(\mathbf{x},\boldsymbol{\phi})$ of the discriminator function with respect to x by using weight clipping, giving rise to the *Wasserstein GAN* (Arjovsky, Chintala, and Bottou, 2017). An improved approach is to introduce a penalty on the gradient, giving rise to the *gradient penalty Wasserstein GAN* (Gulrajani *et al.*, 2017) whose error function is given by

$$\begin{aligned}E_{\text{WGAN-GP}}(\mathbf{w},\boldsymbol{\phi}) = &-\frac{1}{N_{\text{real}}}\sum_{n\in\text{real}}\left[\ln d(\mathbf{x}_n,\boldsymbol{\phi})-\eta\left(\|\nabla_{\mathbf{x}_n}d(\mathbf{x}_n,\boldsymbol{\phi})\|^2-1\right)^2\right]\\ &+\frac{1}{N_{\text{synth}}}\sum_{n\in\text{synth}}\ln d(\mathbf{g}(\mathbf{z}_n,\mathbf{w},\boldsymbol{\phi}))\end{aligned} \tag{17.11}$$

where η controls the relative importance of the penalty term.

Telegram: @uni_k

www.konkur.in

17.2. Image GANs

The basic concept of the GAN has given rise to a huge research literature, with many algorithmic developments and numerous applications. One of the most widespread and successful application areas for GANs is the generation of images. Early GAN models used fully connected networks for the generator and discriminator. However, there are many benefits to using convolutional networks, especially for images of higher resolution. The discriminator network takes an image as input and provides a scalar probability as output, so a standard convolutional network is appropriate. The generator network needs to map a lower-dimensional latent space into a high-resolution image, and so a network based on transpose convolutions is used, as illustrated in Figure 17.4.

Chapter 10

Section 10.5.3

High quality images can be obtained by progressively growing both the generator network and the discriminator network starting from a low resolution and then successively adding new layers that model increasingly fine details as training progresses (Karras *et al.*, 2017). This speeds up the training and permits the synthesis of high-resolution images of size 1024×1024 starting from images of size 4×4. As an example of the scale and complexity of some GAN architectures, consider the GAN model for class-conditional image generation called *BigGAN*, whose architecture is shown in Figure 17.5.

17.2.1 CycleGAN

As an example of the broad variety of GANs we consider an architecture called a *CycleGAN* (Zhu *et al.*, 2017). This also illustrates how techniques in deep learning can be adapted to solve different kinds of problems beyond traditional tasks such as

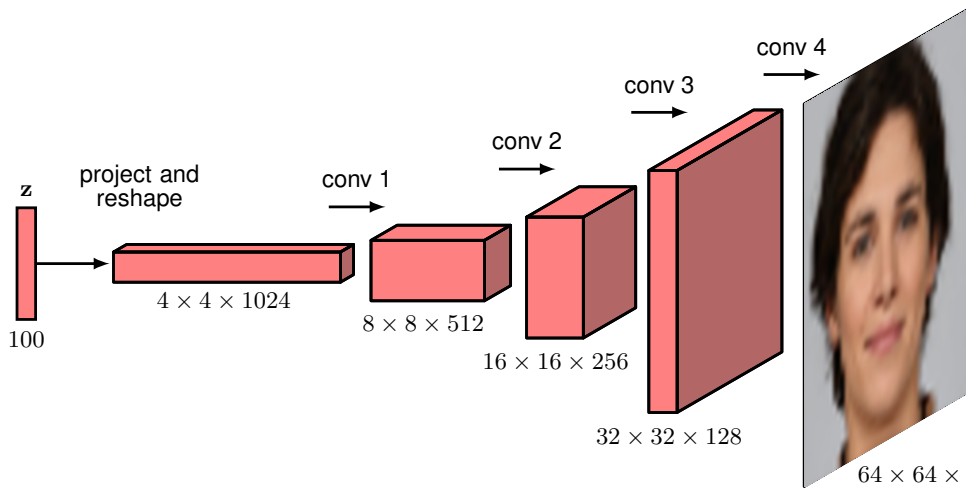


Figure 17.4 Example architecture of a deep convolutional GAN showing the use of transpose convolutions to expand the dimensionality in successive blocks of the network.

Telegram: @uni_k

www.konkur.in

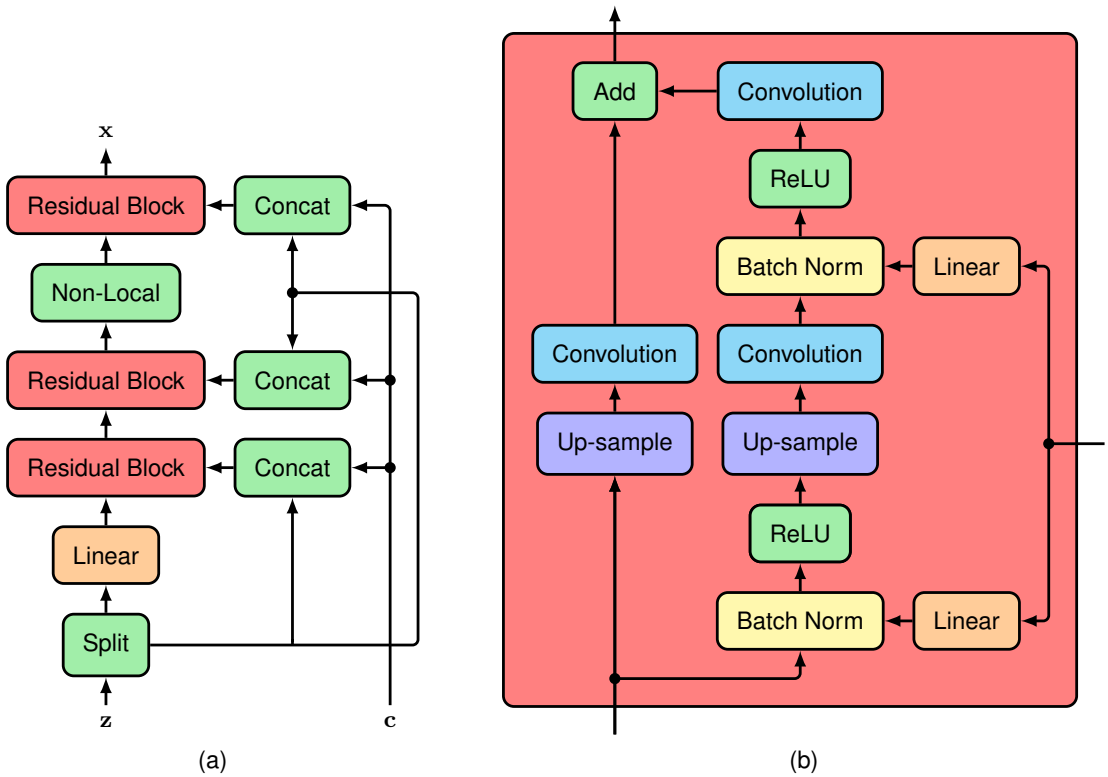


Figure 17.5 (a) Architecture of the generative network in the BigGAN model, which has over 70 million parameters. (b) Details of each of the residual blocks in the generative network. The discriminative network, which has 88 million parameters, has a somewhat analogous structure except that it uses average pooling layers to reduce the dimensionality, instead of using up-sampling to increase the dimensionality. [Based on Brock, Donahue, and Simonyan (2018).]

classification and density estimation. Consider the problem of turning a photograph into a Monet painting of the same scene, or vice versa. In Figure 17.6 we show examples of image pairs from a trained CycleGAN that has learned to perform such an image-to-image translation.

The aim is to learn two bijective (one-to-one) mappings, one that goes from the domain X of photographs to the domain Y of Monet paintings and one in the reverse direction. To achieve this, CycleGAN makes use of two conditional generators, $\mathbf{g}_X$ and $\mathbf{g}_Y$, and two discriminators, d_X and d_Y. The generator $\mathbf{g}_X(\mathbf{y}, \mathbf{w}_X)$ takes as input a sample painting $\mathbf{y} \in Y$ and generates a corresponding synthetic photograph, whereas the discriminator $d_X(\mathbf{x}, \boldsymbol{\phi}_X)$ distinguishes between synthetic and real photographs. Similarly, the generator $\mathbf{g}_Y(\mathbf{x}, \mathbf{w}_Y)$ takes a photograph $\mathbf{x} \in X$ as input and generates a synthetic painting $\mathbf{y}$, and the discriminator $d_Y(\mathbf{y}, \boldsymbol{\phi}_Y)$ distinguishes between synthetic paintings and real ones. The discriminator d_X is therefore trained on a combination of synthetic photographs generated by $\mathbf{g}_X$ and real photographs, whereas d_Y is trained on a combination of synthetic paintings generated by $\mathbf{g}_Y$ and

Telegram: @uni_k

www.konkur.in

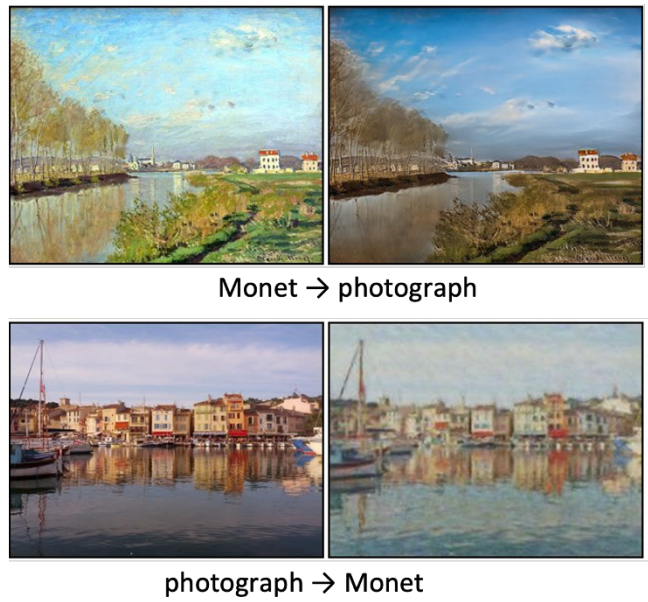


Figure 17.6 Examples of image translation using a CycleGAN showing the synthesis of a photographic-style image from a Monet painting (top row) and the synthesis of an image in the style of a Monet painting from a photograph (bottom row). [From Zhu *et al.* (2017) with permission.]

real paintings.

If we train this architecture using the standard GAN loss function, it would learn to generate realistic synthetic Monet paintings and realistic synthetic photographs, but there would be nothing to force a generated painting to look anything like the corresponding photograph, or vice versa. We therefore introduce an additional term in the loss function called the cycle consistency error, containing two terms, whose construction is illustrated in Figure 17.7.

The goal is to ensure that when a photograph is translated into a painting and then back into a photograph it should be close to the original photograph, thereby ensuring that the generated painting retains sufficient information about the photograph to allow the photograph to be reconstructed. Similarly, when a painting is translated into a photograph and then back into a painting it should be close to the

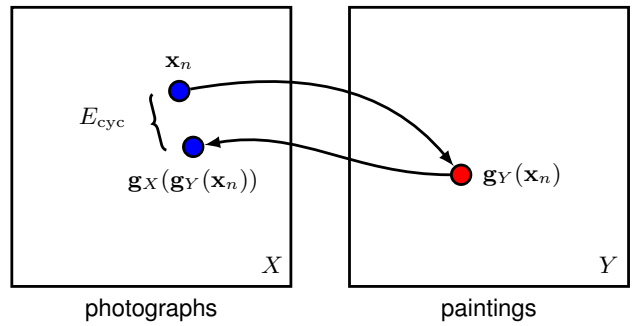


Figure 17.7 Diagram showing how the cycle consistency error is calculated for an example photograph $\mathbf{x}_n$. The photograph is first mapped into the painting domain using the generator $\mathbf{g}_Y$, and the resulting vector is then mapped back into the photograph domain using the generator $\mathbf{g}_X$. The discrepancy between the resulting photograph and the original $\mathbf{x}_n$ defines a contribution to the cycle consistency error. An analogous process is used to calculate the contribution to the cycle consistency error from a painting $\mathbf{y}_n$ by mapping it to a photograph using $\mathbf{g}_X$ and then back to a painting using $\mathbf{g}_Y$.

Telegram: @uni_k

www.konkur.in

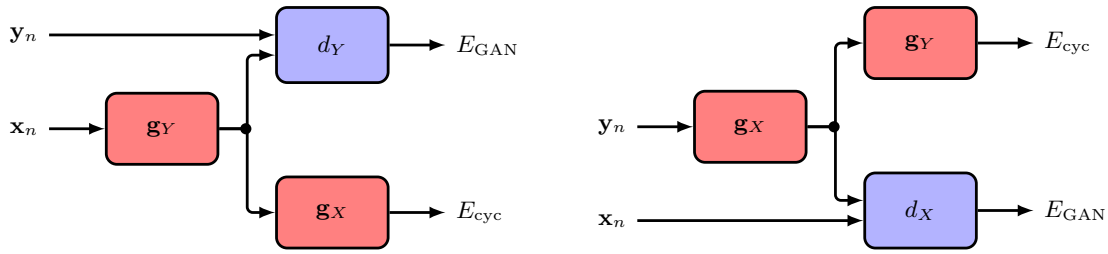


Figure 17.8 Flow of information through a CycleGAN. The total error for the data points $\mathbf{x}_n$ and $\mathbf{y}_n$ is the sum of the four component errors.

original painting. Applying this to all the photographs and paintings in the training set then gives a cycle consistency error of the form

$$E_{\text{cyc}}(\mathbf{w}_X, \mathbf{w}_Y) = \frac{1}{N_X} \sum_{n \in X} \|\mathbf{g}_X(\mathbf{g}_Y(\mathbf{x}_n)) - \mathbf{x}_n\|_1 + \frac{1}{N_Y} \sum_{n \in Y} \|\mathbf{g}_Y(\mathbf{g}_X(\mathbf{y}_n)) - \mathbf{y}_n\|_1 \tag{17.12}$$

where $\|\cdot\|_1$ denotes the L1 norm. The cycle consistency error is added to the usual GAN loss functions defined by (17.6) to give a total error function:

$$E_{\text{GAN}}(\mathbf{w}_X, \boldsymbol{\phi}_X) + E_{\text{GAN}}(\mathbf{w}_Y, \boldsymbol{\phi}_Y) + \eta E_{\text{cyc}}(\mathbf{w}_X, \mathbf{w}_Y) \tag{17.13}$$

where the coefficient η determines the relative importance of the GAN errors and the cycle consistency error. Information flow through the CycleGAN when calculating the error function for one image and one painting is shown in Figure 17.8.

Section 6.3.3

We have seen that GANs can perform well as generative models, but they can also be used for *representation learning* in which rich statistical structure in a data set is revealed through unsupervised learning. When the deep convolutional GAN shown in Figure 17.4 is trained on a data set of bedroom images (Radford, Metz, and Chintala, 2015) and random samples from the latent space are propagated through the trained network, the generated images also look like bedrooms, as expected. In addition, however, the latent space has become organized in ways that are semantically meaningful. For example, if we follow a smooth trajectory through the latent space and generate the corresponding series of images, we obtain smooth transitions from one image to the next, as seen in Figure 17.9.

Moreover, it is possible to identify directions in latent space that correspond to semantically meaningful transformations. For example, for faces, one direction might correspond to changes in the orientation of the face, whereas other directions might correspond to changes in lighting or the degree to which the face is smiling or not. These are called *disentangled representations* and allow new images to be synthesized having specified properties. Figure 17.10 is an example from a GAN trained on face images, showing that semantic attributes such as gender or the presence of glasses correspond to particular directions in latent space.

Telegram: @uni_k

www.konkur.in

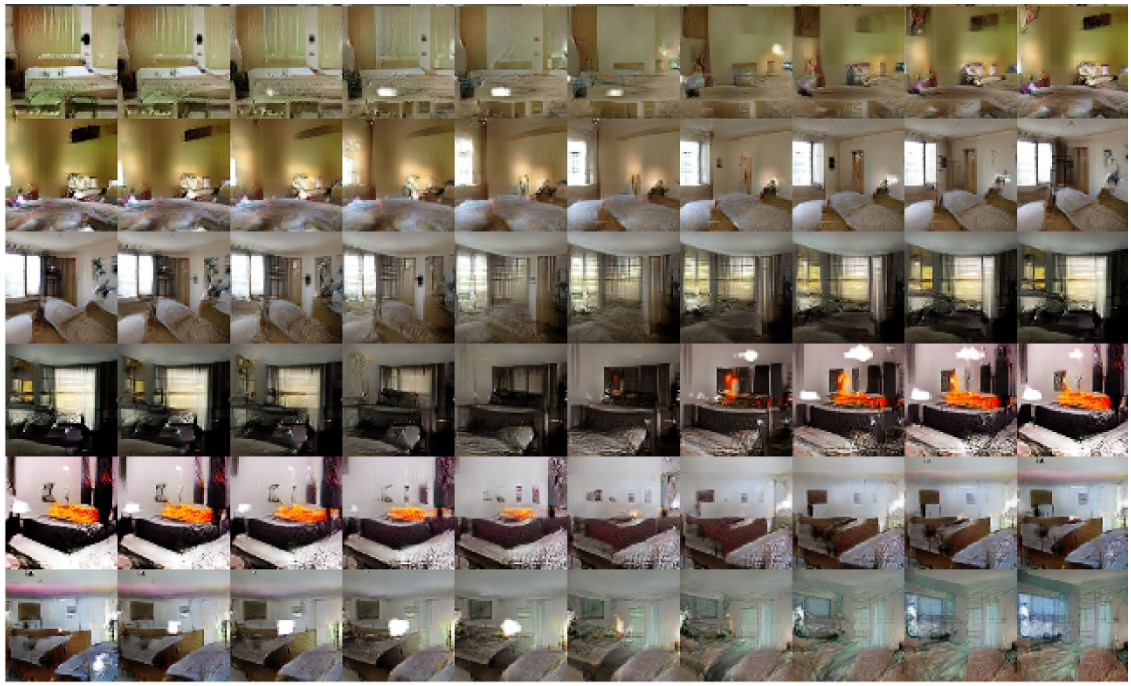


Figure 17.9 Samples generated by a deep convolutional GAN trained on images of bedrooms. Each row is generated by taking a smooth walk through latent space between randomly generated locations. We see smooth transitions, with each image plausibly looking like a bedroom. In the bottom row, for example, we see a TV on the wall gradually morph into a window. [From Radford, Metz, and Chintala (2015) with permission.]

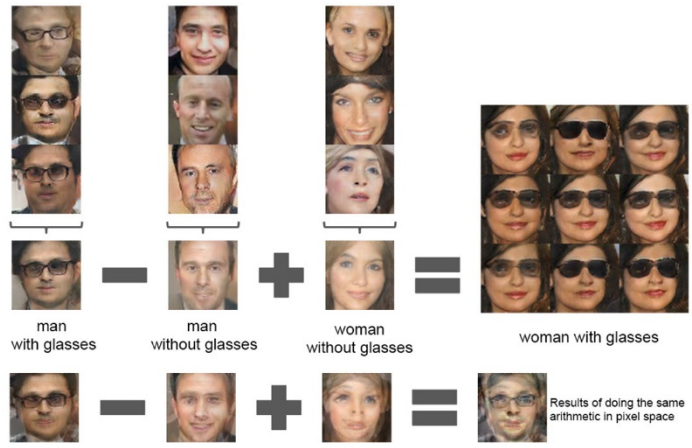


Figure 17.10 An example of vector arithmetic in the latent space of a trained GAN. In each of the three columns, the latent space vectors that generated these images are averaged and then vector arithmetic is applied to the resulting mean vectors to create a new vector corresponding to the central image in the 3×3 array on the right. Adding noise to this vector generates another eight sample images. The four images on the bottom row show that the same arithmetic applied directly in data space simply results in a blurred image due to misalignment. [From Radford, Metz, and Chintala (2015) with permission.]

Telegram: @uni_k

www.konkur.in

Exercises

17.1 ($\star\star\star$) We would like the GAN error function (17.6) to have the property that, given sufficiently flexible neural networks, the stationary point is obtained when the generator distribution matches the true data distribution. In this exercise we prove this result for network models with infinite flexibility by optimizing over the full space of probability distributions $p_{\mathrm{G}}(\mathbf{x})$ and over the full space of functions $d(\mathbf{x})$ corresponding to the generative and discriminative networks, respectively. Specifically, we assume that the discriminative model is optimized in an inner loop, giving rise to an effective outer loop error function for the generative model. First, show that, in the limit of an infinite number of data samples, the GAN error function (17.6) can be rewritten in the form

$$E(p_{\mathrm{G}}, d) = -\int p_{\mathrm{data}}(\mathbf{x}) \ln d(\mathbf{x})\, \mathrm{d}\mathbf{x} - \int p_{\mathrm{G}}(\mathbf{x}) \ln(1 - d(\mathbf{x}))\, \mathrm{d}\mathbf{x} \tag{17.14}$$

where $p_{\mathrm{data}}(\mathbf{x})$ is the fixed distribution of real data points. Now consider a variational optimization over all functions $d(\mathbf{x})$. Show that, for a fixed generative network, the solution for the discriminator $d(\mathbf{x})$ that minimizes E is given by

Appendix B

$$d^{\star}(\mathbf{x}) = \frac{p_{\mathrm{data}}(\mathbf{x})}{p_{\mathrm{data}}(\mathbf{x}) + p_{\mathrm{G}}(\mathbf{x})}. \tag{17.15}$$

Hence, show that the error function E can be written as a function of the generator network $p_{\mathrm{G}}(\mathbf{x})$ in the form

$$\begin{aligned} C(p_{\mathrm{G}}) = &-\int p_{\mathrm{data}}(\mathbf{x}) \ln \left\{ \frac{p_{\mathrm{data}}(\mathbf{x})}{p_{\mathrm{data}}(\mathbf{x}) + p_{\mathrm{G}}(\mathbf{x})} \right\} \mathrm{d}\mathbf{x} \\ &- \int p_{\mathrm{G}}(\mathbf{x}) \ln \left\{ \frac{p_{\mathrm{G}}(\mathbf{x})}{p_{\mathrm{data}}(\mathbf{x}) + p_{\mathrm{G}}(\mathbf{x})} \right\} \mathrm{d}\mathbf{x}. \end{aligned} \tag{17.16}$$

Now show that this can be rewritten in the form

$$C(p_{\mathrm{G}}) = -\ln(4) + \mathrm{KL}\left(p_{\mathrm{data}} \,\middle\|\, \frac{p_{\mathrm{data}} + p_{\mathrm{G}}}{2}\right) + \mathrm{KL}\left(p_{\mathrm{G}} \,\middle\|\, \frac{p_{\mathrm{data}} + p_{\mathrm{G}}}{2}\right) \tag{17.17}$$

where the Kullback–Leibler divergence $\mathrm{KL}(p\|q)$ is defined by (2.100). Finally, using the property that $\mathrm{KL}(p\|q) \geqslant 0$ with equality if, and only if, $p(\mathbf{x}) = q(\mathbf{x})$ for all $\mathbf{x}$, show that the minimum of $C(p_{\mathrm{G}})$ occurs when $p_{\mathrm{G}}(\mathbf{x}) = p_{\mathrm{data}}(\mathbf{x})$. Note that the sum of the two Kullback–Leibler divergence terms in (17.17) is known as the *Jensen–Shannon divergence* between p_{data} and p_{G}. Like the Kullback–Leibler divergence, this is a non-negative quantity that vanishes if, and only if, the two distributions are equal, but unlike the KL divergence, it is symmetric with respect to the two distributions.

Section 2.5.5

17.2 ($\star\star\star$) In this exercise we explore the problems that can arise from the adversarial nature of GAN training. Consider a cost function $E(a, b) = ab$ defined over two parameters a and b, analogous to the parameters of a generative and discriminative

Telegram: @uni_k

www.konkur.in

network, respectively. Show that the point $a = 0, b = 0$ is a stationary point of the cost function. By considering the second derivatives along the lines $b = a$ and $b = -a$ show that the point $a = 0, b = 0$ is a saddle point. Now suppose that we optimize this error function by taking infinitesimal steps, so that the variables become functions of continuous time $a(t)$, $b(t)$ defined by a continuous-time gradient descent, in which the parameter $a(t)$ of the generative network is updated so as to increase $E(a, b)$, whereas the parameter $b(t)$ is updated so as to decrease $E(a, b)$. Show that the evolution of the parameters is governed by the equations

$$\frac{\mathrm{d}a}{\mathrm{d}t} = \eta \frac{\partial E}{\partial a}, \qquad \frac{\mathrm{d}b}{\mathrm{d}t} = -\eta \frac{\partial E}{\partial b}. \tag{17.18}$$

Hence, show that $a(t)$ satisfies the second-order differential equation

$$\frac{\mathrm{d}^2 a}{\mathrm{d}t^2} = -\eta^2 a(t). \tag{17.19}$$

Verify that the following expression is a solution of (17.19):

$$a(t) = C\cos(\eta t) + D\sin(\eta t) \tag{17.20}$$

where C and D are arbitrary constants. If the system is initialized at $t = 0$ with the values $a = 1$, $b = 0$, find the values of C and D and hence show that the resulting values of $a(t)$ and $b(t)$ trace out a circle of unit radius in a, b space centred on the origin, and that they therefore never converge to the saddle point.

17.3 ($\star$) Consider a GAN in which the training set consists of equal numbers of cat and dog images and in which the generator network has learned to produce high quality images of dogs. Show that, when presented with a dog image, the optimal output for the discriminator network (trained to generate the probability that the image is real) is $1/3$.

Telegram: @uni_k

www.konkur.in



18 Normalizing Flows

Chapter 17

We have seen how generative adversarial networks (GANs) extend the framework of linear latent-variable models by using deep neural networks to represent highly flexible and learnable nonlinear transformations from the latent space to the data space. However, the likelihood function is generally either intractable, because the network function cannot be inverted, or may not even be defined if the latent space has a lower dimensionality than the data space. In GANs, a second, discriminative network was therefore introduced to facilitate adversarial training.

Section 16.4.4

Here we discuss the second of our four approaches to training nonlinear latent variable models that involves restricting the form of the neural network model such that the likelihood function can be evaluated without approximation while still ensuring that sampling from the trained model is straightforward. Suppose we define a distribution $p_{\mathbf{z}}(\mathbf{z})$, sometimes also called a *base distribution*, over a latent variable $\mathbf{z}$ along with a nonlinear function $\mathbf{x} = \mathbf{f}(\mathbf{z}, \mathbf{w})$, given by a deep neural network, that

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024
C. M. Bishop, H. Bishop, *Deep Learning*, https://doi.org/10.1007/978-3-031-45468-4_18

Telegram: @uni_k

www.konkur.in

transforms the latent space into the data space. Assuming $p_{\mathbf{z}}(\mathbf{z})$ is a simple distribution such as a Gaussian, sampling from such a model is easy as each latent sample $\mathbf{z}^\star \sim p_{\mathbf{z}}(\mathbf{z})$ is simply passed through the neural network to generate a corresponding data sample $\mathbf{x}^\star = \mathbf{f}(\mathbf{z}^\star, \mathbf{w})$.

To calculate the likelihood function for this model, we need the data-space distribution, which depends on the *inverse* of the neural network function. We write this as $\mathbf{z} = \mathbf{g}(\mathbf{x}, \mathbf{w})$, and it satisfies $\mathbf{z} = \mathbf{g}(\mathbf{f}(\mathbf{z}, \mathbf{w}), \mathbf{w})$. This requires that, for every value of $\mathbf{w}$, the functions $\mathbf{f}(\mathbf{z}, \mathbf{w})$ and $\mathbf{g}(\mathbf{x}, \mathbf{w})$ are invertible, also called *bijective*, so that each value of $\mathbf{x}$ corresponds to a unique value of $\mathbf{z}$ and vice versa. We can then use the change of variables formula to calculate the data density:

Section 2.4

$$p_{\mathbf{x}}(\mathbf{x}|\mathbf{w}) = p_{\mathbf{z}}(\mathbf{g}(\mathbf{x}, \mathbf{w}))\,|\det \mathbf{J}(\mathbf{x})| \tag{18.1}$$

where $\mathbf{J}(\mathbf{x})$ is the Jacobian matrix of partial derivatives whose elements are given by

$$J_{ij}(\mathbf{x}) = \frac{\partial g_i(\mathbf{x}, \mathbf{w})}{\partial x_j} \tag{18.2}$$

and $|\cdot|$ denotes the modulus or absolute value. We will continue to refer to $\mathbf{z}$ as a 'latent' variable even though the deterministic mapping means that any given data value $\mathbf{x}$ corresponds to a unique value of $\mathbf{z}$ whose value is therefore no longer uncertain.

The mapping function $\mathbf{f}(\mathbf{z}, \mathbf{w})$ will be defined in terms of a special form of neural network, whose structure we will discuss shortly. One consequence of requiring an invertible mapping is that the dimensionality of the latent space must be the same as that of the data space, which can lead to large models for high-dimensional data such as images. Also, in general, the cost of evaluating the determinant of a $D \times D$ matrix is $\mathcal{O}(D^3)$, so we will seek to impose some further restrictions on the model in order that evaluation of the Jacobian matrix determinant is more efficient.

If we consider a training set $\mathcal{D} = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$ of independent data points, the log likelihood function is given from (18.1) by

$$\ln p(\mathcal{D}|\mathbf{w}) = \sum_{n=1}^{N} \ln p_{\mathbf{x}}(\mathbf{x}_n|\mathbf{w}) \tag{18.3}$$

$$= \sum_{n=1}^{N} \Big\{ \ln p_{\mathbf{z}}(\mathbf{g}(\mathbf{x}_n, \mathbf{w})) + \ln |\det \mathbf{J}(\mathbf{x}_n)| \Big\} \tag{18.4}$$

and our goal is to use the likelihood function to train the neural network. To be able to model a wide range of distributions, we want the transformation function $\mathbf{x} = \mathbf{f}(\mathbf{z}, \mathbf{w})$ to be highly flexible, and so we use a deep neural network architecture. We can ensure that the overall function is invertible if we make each layer of the network invertible. To see this, consider three successive transformations, each corresponding to one layer, of the form:

Exercise 18.2

$$\mathbf{x} = \mathbf{f}^A(\mathbf{f}^B(\mathbf{f}^C(\mathbf{z}))). \tag{18.5}$$

Then the inverse function is given by

$$\mathbf{z} = \mathbf{g}^C(\mathbf{g}^B(\mathbf{g}^A(\mathbf{x}))) \tag{18.6}$$

Telegram: @uni_k

www.konkur.in

where $\mathbf{g}^A, \mathbf{g}^B$, and $\mathbf{g}^C$ are the inverse functions of $\mathbf{f}^A, \mathbf{f}^B$, and $\mathbf{f}^C$, respectively. Moreover, the determinant of the Jacobian for such a layered structure is also easy to evaluate in terms of the Jacobian determinants for each of the individual layers by making use of the chain rule of calculus:

$$J_{ij} = \frac{\partial z_i}{\partial x_j} = \sum_k \sum_l \frac{\partial g_i^C}{\partial g_k^B} \frac{\partial g_k^B}{\partial g_l^A} \frac{\partial g_l^A}{\partial x_j}. \tag{18.7}$$

Appendix A

We recognize the right-hand side as the product of three matrices, and the determinant of a product is the product of the determinants. Therefore, the log determinant of the overall Jacobian will be the sum of the log determinants corresponding to each layer.

This approach to modelling a flexible distribution is called a *normalizing flow* because the transformation of a probability distribution through a sequence of mappings is somewhat analogous to the flow of a fluid. Also, the effect of the inverse mapping is to transform the complex data distribution into a normalized form, typically a Gaussian or normal distribution. Normalizing flows have been reviewed by Kobyzev, Prince, and Brubaker (2019) and Papamakarios *et al.* (2019). Here we discuss the core concepts from the two main classes of normalizing flows used in practice: *coupling flows* and *autoregressive flows*. We also look at the use of neural differential equations to define invertible mappings, leading to *continuous flows*.

18.1. Coupling Flows

Our goal is to design a single invertible function layer, so that we can compose many of them together to define a highly flexible class of invertible functions. Consider first a linear transformation of the form

$$\mathbf{x} = a\mathbf{z} + \mathbf{b}. \tag{18.8}$$

This is easy to invert, giving

$$\mathbf{z} = \frac{1}{a}(\mathbf{x} - \mathbf{b}). \tag{18.9}$$

Exercise 3.6

However, linear transformations are closed under composition, meaning that a sequence of linear transformations is equivalent to a single overall linear transformation. Moreover, a linear transformation of a Gaussian distribution is again Gaussian. So even if we have many such 'layers' of linear transformation, we will only ever have a Gaussian distribution. The question is whether we can retain the invertability of a linear transformation while allowing additional flexibility so that the resulting distribution can be non-Gaussian.

One solution to this problem is given by a form of normalizing flow model called *real NVP* (Dinh, Krueger, and Bengio, 2014; Dinh, Sohl-Dickstein, and Bengio, 2016), which is short for 'real-valued non-volume-preserving'. The idea is to partition the latent-variable vector $\mathbf{z}$ into two parts $\mathbf{z} = (\mathbf{z}_A, \mathbf{z}_B)$, so that if $\mathbf{z}$ has dimension D and $\mathbf{z}_A$ has dimension d, then $\mathbf{z}_B$ has dimension $D - d$. We similarly

Telegram: @uni_k

www.konkur.in

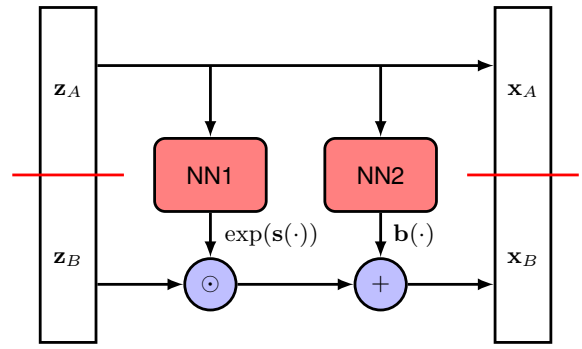


Figure 18.1 A single layer of the real NVP normalizing flow model. Here the network NN1 computes the function $\exp(\mathbf{s}(\mathbf{z}_A, \mathbf{w}))$ and the network NN2 computes the function $\mathbf{b}(\mathbf{z}_A, \mathbf{w})$. The output vector is then defined by (18.10) and (18.11).

partition the output vector $\mathbf{x} = (\mathbf{x}_A, \mathbf{x}_B)$ where $\mathbf{x}_A$ has dimension d and $\mathbf{x}_B$ has dimension $D - d$. For the first part of the output vector, we simply copy the input:

$$\mathbf{x}_A = \mathbf{z}_A. \tag{18.10}$$

The second part of the vector undergoes a linear transformation, but now the coefficients in the linear transformation are given by nonlinear functions of $\mathbf{z}_A$:

$$\mathbf{x}_B = \exp(\mathbf{s}(\mathbf{z}_A, \mathbf{w})) \odot \mathbf{z}_B + \mathbf{b}(\mathbf{z}_A, \mathbf{w}) \tag{18.11}$$

where $\mathbf{s}(\mathbf{z}_A, \mathbf{w})$ and $\mathbf{b}(\mathbf{z}_A, \mathbf{w})$ are the real-valued outputs of neural networks, and the exponential ensures that the multiplicative term is non-negative. Here $\odot$ denotes the *Hadamard product* involving an element-wise multiplication of the two vectors. Similarly, the exponential in (18.11) is taken element-wise. Note that we have shown the same vector $\mathbf{w}$ in both network functions. In practice, these may be implemented as separate networks with their own parameters, or as one network with two sets of outputs.

Due to the use of neural network functions, the value of $\mathbf{x}_B$ can be a very flexible function of $\mathbf{x}_A$. Nevertheless, the overall transformation is easily invertible: given a value for $\mathbf{x} = (\mathbf{x}_A, \mathbf{x}_B)$ we first compute

$$\mathbf{z}_A = \mathbf{x}_A, \tag{18.12}$$

then we evaluate $\mathbf{s}(\mathbf{z}_A, \mathbf{w})$ and $\mathbf{b}(\mathbf{z}_A, \mathbf{w})$, and finally we compute $\mathbf{z}_B$ using

$$\mathbf{z}_B = \exp(-\mathbf{s}(\mathbf{z}_A, \mathbf{w})) \odot (\mathbf{x}_B - \mathbf{b}(\mathbf{z}_A, \mathbf{w})). \tag{18.13}$$

The overall transformation is illustrated in Figure 18.1. Note that there is no requirement for the individual neural network functions $\mathbf{s}(\mathbf{z}_A, \mathbf{w})$ and $\mathbf{b}(\mathbf{z}_A, \mathbf{w})$ to be invertible.

Now consider the evaluation of the Jacobian defined by (18.2) and its determinant. We can divide the Jacobian matrix into blocks, corresponding to the partitioning of $\mathbf{z}$ and $\mathbf{x}$, giving

$$\mathbf{J} = \begin{bmatrix} \mathbf{I}_d & \mathbf{0} \\ \dfrac{\partial \mathbf{z}_B}{\partial \mathbf{x}_A} & \text{diag}(\exp(-\mathbf{s})) \end{bmatrix}. \tag{18.14}$$

Telegram: @uni_k

www.konkur.in

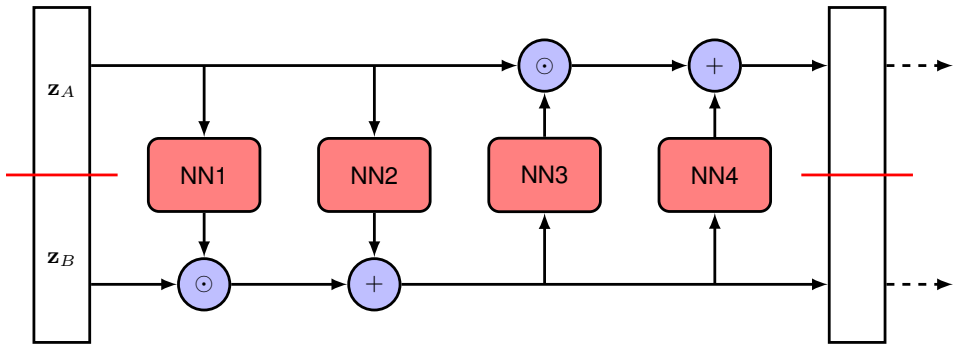


Figure 18.2 By composing two layers of the form shown in Figure 18.1, we obtain a more flexible, but still invertible, nonlinear layer. Each sub-layer is invertible and has an easily evaluated Jacobian, and hence the overall double layer has the same properties.

The top left block corresponds to the derivatives of $\mathbf{z}_A$ with respect to $\mathbf{x}_A$ and hence from (18.12) is given by the $d \times d$ identity matrix. The top right block corresponds to the derivatives of $\mathbf{z}_A$ with respect to $\mathbf{x}_B$ and these terms vanish, again from (18.12). The bottom left block corresponds to the derivatives of $\mathbf{z}_B$ with respect to $\mathbf{x}_A$. From (18.13), these are complicated expressions involving the neural network functions. Finally, the bottom right block corresponds to the derivatives of $\mathbf{z}_B$ with respect to $\mathbf{x}_B$, which from (18.13) are given by a diagonal matrix whose diagonal elements are given by the exponentials of the negative elements of $\mathbf{s}(\mathbf{z}_A, \mathbf{w})$. We therefore see that the Jacobian matrix (18.14) is a lower triangular matrix, meaning that all elements above the leading diagonal are zero. For such a matrix, the determinant is just the product of the elements along the leading diagonal, and therefore it does not depend on the complicated expressions in the lower left block. Consequently, the determinant of the Jacobian is simply given by the product of the elements of $\exp(-\mathbf{s}(\mathbf{z}_A, \mathbf{w}))$.

Appendix A

A clear limitation of this approach is that the value of $\mathbf{z}_A$ is unchanged by the transformation. This is easily resolved by adding another layer in which the roles of $\mathbf{z}_A$ and $\mathbf{z}_B$ are reversed, as illustrated in Figure 18.2. This double-layer structure can then be repeated multiple times to facilitate a very flexible class of generative models.

The overall training procedure involves creating mini-batches of data points, in which the contribution of each data point to the log likelihood function is obtained from (18.4). For a latent distribution of the form $\mathcal{N}(\mathbf{z}|\mathbf{0}, \mathbf{I})$, the log density is simply $-\|\mathbf{z}\|^2/2$ up to an additive constant. The inverse transformation $\mathbf{z} = \mathbf{g}(\mathbf{x})$ is calculated using a sequence of inverse transformations of the form (18.13). Similarly, the log of the Jacobian determinant is given by a sum of log determinants for each layer where each term is itself a sum of terms of the form $-s_i(\mathbf{x}, \mathbf{w})$. Gradients of the log likelihood can be evaluated using automatic differentiation, and the network parameters updated by stochastic gradient descent.

The real NVP model belongs to a broad class of normalizing flows called *coupling flows*, in which the linear transformation (18.11) is replaced by a more general

Telegram: @uni_k

www.konkur.in

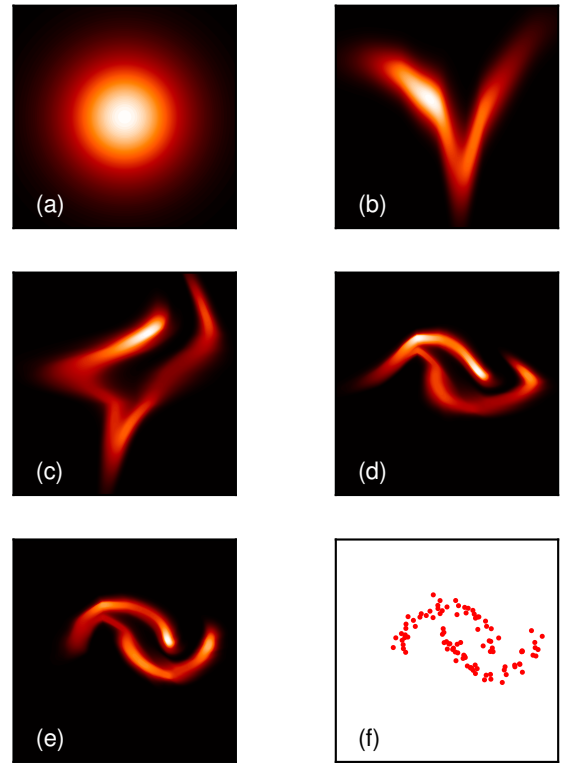


Figure 18.3 Illustration of the real NVP normalizing flow model applied to the two-moons data set showing (a) the Gaussian base distribution, (b) the distribution after a transformation of the vertical axis only, (c) the distribution after a subsequent transformation of the horizontal axis, (d) the distribution after a second transformation of the vertical axis, (e) the distribution after a second transformation of the horizontal axis, and (f) the data set on which the model was trained.

form:

$$\mathbf{x}_B = \mathbf{h}(\mathbf{z}_B, \mathbf{g}(\mathbf{z}_A, \mathbf{w})) \tag{18.15}$$

where $\mathbf{h}(\mathbf{z}_B, \mathbf{g})$ is a function of $\mathbf{z}_B$ that is efficiently invertible for any given value of $\mathbf{g}$ and is called the *coupling function*. The function $\mathbf{g}(\mathbf{z}_A, \mathbf{w})$ is called a *conditioner* and is typically represented by a neural network.

We can illustrate the real NVP normalizing flow using a simple data set, sometimes known as 'two moons', as shown in Figure 18.3. Here a two-dimensional Gaussian distribution is transformed into a more complex distribution by using two successive layers each of which consists of alternate transformations on each of the two dimensions.

18.2. Autoregressive Flows

A related formulation of normalizing flows can be motivated by noting that the joint distribution over a set of variables can always be written as the product of conditional distributions, one for each variable. We first choose an ordering of the variables in

Section 11.1

Telegram: @uni_k

www.konkur.in

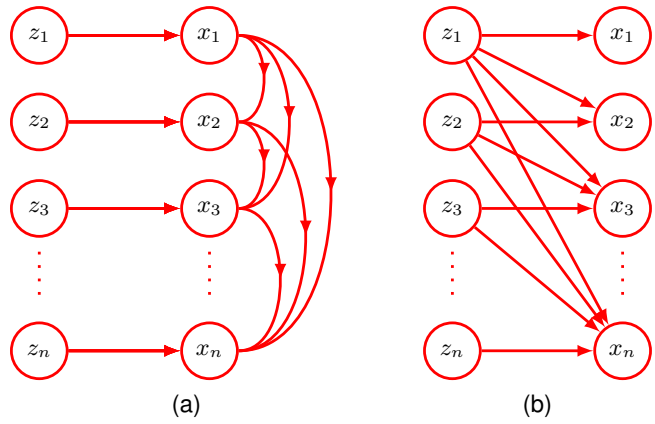


Figure 18.4 Illustration of two alternative structures for autoregressive normalizing flows. The *masked autoregressive flow* shown in (a) allows efficient evaluation of the likelihood function, whereas the alternative *inverse autoregressive flow* shown in (b) allows for efficient sampling.

the vector $\mathbf{x}$, from which we can write, without loss of generality,

$$p(x_1, \ldots, x_D) = \prod_{i=1}^{D} p(x_i | \mathbf{x}_{1:i-1}) \tag{18.16}$$

where $\mathbf{x}_{1:i-1}$ denotes $x_1, \ldots, x_{i-1}$. This factorization can be used to construct a class of normalizing flow called a *masked autoregressive flow*, or MAF (Papamakarios, Pavlakou, and Murray, 2017), given by

$$x_i = h(z_i, \mathbf{g}_i(\mathbf{x}_{1:i-1}, \mathbf{w}_i)) \tag{18.17}$$

which is illustrated in Figure 18.4(a). Here $h(z_i, \cdot)$ is the *coupling function*, which is chosen to be easily invertible with respect to z_i, and $\mathbf{g}_i$ is the *conditioner*, which is typically represented by a deep neural network. The term *masked* refers to the use of a single neural network to implement a set of equations of the form (18.17) along with a binary mask (Germain *et al.*, 2015) that force a subset of the network weights to be zero to implement the autoregressive constraint (18.16).

In this case the reverse calculations needed to evaluate the likelihood function are given by

$$z_i = h^{-1}(x_i, \mathbf{g}_i(\mathbf{x}_{1:i-1}, \mathbf{w}_i)) \tag{18.18}$$

and hence can be performed efficiently on modern hardware since the individual functions in (18.18) needed to evaluate $z_1, \ldots, z_D$ can be evaluated in parallel. The Jacobian matrix corresponding to the set of transformations (18.18) has elements $\partial z_i / \partial x_j$, which form an upper-triangular matrix whose determinant is given by the product of the diagonal elements and can therefore also be evaluated efficiently. However, sampling from this model must be done by evaluating (18.17), which is intrinsically sequential and therefore slow because the values of $x_1, \ldots, x_{i-1}$ must be evaluated before x_i can be computed.

Exercise 18.4

To avoid this inefficient sampling, we can instead define an *inverse autoregressive flows*, or IAF (Kingma *et al.*, 2016), given by

$$x_i = h(z_i, \widetilde{\mathbf{g}}_i(\mathbf{z}_{1:i-1}, \mathbf{w}_i)) \tag{18.19}$$

Telegram: @uni_k

www.konkur.in

as illustrated in Figure 18.4(b). Sampling is now efficient since, for a given choice of $\mathbf{z}$, the evaluation of the elements $x_1, \ldots, x_D$ using (18.19) can be performed in parallel. However, the inverse function, which is needed to evaluate the likelihood, requires a series of calculations of the form

$$z_i = h^{-1}(x_i, \widetilde{\mathbf{g}}_i(\mathbf{z}_{1:i-1}, \mathbf{w}_i)), \tag{18.20}$$

which are intrinsically sequential and therefore slow. Whether a masked autoregressive flow or an inverse autoregressive flow is preferred will depend on the specific application.

We see that coupling flows and autoregressive flows are closely related. Although autoregressive flows introduce considerable flexibility, this comes with a computational cost that grows linearly in the dimensionality D of the data space due to the need for sequential ancestral sampling. Coupling flows can be viewed as a special case of autoregressive flows in which some of this generality is sacrificed for efficiency by dividing the variables into two groups instead of D groups.

18.3. Continuous Flows

The final approach to normalizing flows that we consider in this chapter will make use of deep neural networks defined in terms of an ordinary differential equation, or ODE. This can be thought of as a deep network with an infinite number of layers. We first introduce the concept of a neural ODE then we see how this can be applied to the formulation of a normalizing flow model.

18.3.1 Neural differential equations

We have seen that neural networks are especially useful when they comprise many layers of processing, and so we can ask what happens if we explore the limit of an infinitely large number of layers. Consider a residual network where each layer of processing generates an output given by the input vector with the addition of some parameterized nonlinear function of that input vector:

$$\mathbf{z}^{(t+1)} = \mathbf{z}^{(t)} + \mathbf{f}(\mathbf{z}^{(t)}, \mathbf{w}) \tag{18.21}$$

where $t = 1, \ldots, T$ labels the layers in the network. Note that we have used the same function at each layer, with a shared parameter vector $\mathbf{w}$, because this allows us to consider an arbitrarily large number of such layers while keeping the number of parameters bounded. Imagine that we increase the number of layers while ensuring that the changes introduced at each layer become correspondingly smaller. In the limit, the hidden-unit activation vector becomes a function $\mathbf{z}(t)$ of a continuous variable t, and we can express the evolution of this vector through the network as a differential equation:

Exercise 18.5

$$\frac{\mathrm{d}\mathbf{z}(\mathbf{t})}{\mathrm{d}t} = \mathbf{f}(\mathbf{z}(t), \mathbf{w}) \tag{18.22}$$

where t is often referred to as 'time'. The formulation in (18.22) is known as a *neural ordinary differential equation* or *neural ODE* (Chen *et al.*, 2018). Here 'ordinary'

Telegram: @uni_k

www.konkur.in

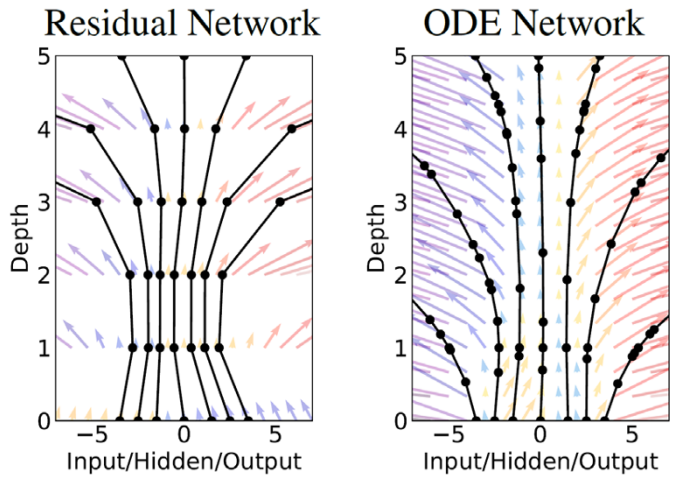


Figure 18.5 Comparison of a conventional layered network with a neural differential equation. The diagram on the left corresponds to a residual network with five layers and shows trajectories for several starting values of a single scalar input. The diagram on the right shows the result of numerical integration of a continuous neural ODE, again for several starting values of the scalar input, in which we see that the function is not evaluated at uniformly-spaced time intervals, but instead the evaluation points are chosen adaptively by the numerical solver and depend on the choice of input value. [From Chen *et al.* (2018) with permission.]

means that there is a single variable t. If we denote the input to the network by the vector $\mathbf{z}(0)$, then the output $\mathbf{z}(T)$ is obtained by integration of the differential equation

$$\mathbf{z}(T) = \int_0^T \mathbf{f}(\mathbf{z}(t), \mathbf{w})\, \mathrm{d}t. \tag{18.23}$$

This integral can be evaluated using standard numerical integration packages. The simplest method for solving differential equations is *Euler's forward integration* method, which corresponds to the expression (18.21). In practice, more powerful numerical integration algorithms can adapt their function evaluation to achieve. In particular, they can adaptively choose values of t that typically are not uniformly spaced. The number of such evaluations replaces the concept of depth in a conventional layered network. A comparison of a standard layered neural network and a neural differential equation are shown in Figure 18.5.

18.3.2 Neural ODE backpropagation

We now need to address the challenge of how to train a neural ODE, that is how to determine the value of $\mathbf{w}$ by optimizing a loss function. Let us assume that we are given a data set comprising values of the input vector $\mathbf{z}(0)$ along with an associated output target vector and a loss function $L(\cdot)$ that depends on the output vector $\mathbf{z}(T)$. One approach would be to use automatic differentiation to differentiate through all of the operations performed by the ODE solver during the forward pass. Although

Section 8.2

Telegram: @uni_k

www.konkur.in

this is straightforward to do, it is costly from a memory perspective and is not optimal in terms of controlling numerical error. Instead, Chen *et al.* (2018) treat the ODE solver as a black box and use a technique called the *adjoint sensitivity method*, which can be viewed as the continuous analogue of explicit backpropagation. Recall that backpropagation involves, for each data point, three successive phases: first a forward propagation to evaluate the activation vectors at each layer of the network, second the evaluation of the derivatives of the loss with respect to the activations at each layer starting at the output and propagating backwards through the network by exploiting the chain rule of calculus, and third the evaluation of the derivatives with respect to network parameters by forming products of activations from the forward pass and gradients from the backward pass. We will see that there are analogous steps when computing the gradients for a neural ODE.

Chapter 8

To apply backpropagation to neural ODEs, we define a quantity called the *adjoint* given by

$$\mathbf{a}(t) = \frac{\mathrm{d}L}{\mathrm{d}\mathbf{z}(t)}. \tag{18.24}$$

We see that $\mathbf{a}(T)$ corresponds to the usual derivative of the loss with respect to the output vector. The adjoint satisfies its own differential equation given by

Exercise 18.6

$$\frac{\mathrm{d}\mathbf{a}(t)}{\mathrm{d}t} = -\mathbf{a}(t)^{\mathrm{T}} \nabla_{\mathbf{z}} f(\mathbf{z}(t), \mathbf{w}), \tag{18.25}$$

which is a continuous version of the chain rule of calculus. This can be solved by integrating backwards starting from $\mathbf{a}(T)$, which again can be done using a black-box ODE solver. In principle, this requires that we have stored the trajectory $\mathbf{z}(t)$ computed during the forward phase, which could be problematic as the inverse solver might wish to evaluate $\mathbf{z}(t)$ at different values of t compared to the forward solver. Instead we simply allow the backwards solver to recompute any required values of $\mathbf{z}(t)$ by integrating (18.22) alongside (18.25) starting with the output value $\mathbf{z}(T)$.

The third step in the backpropagation method is to evaluate derivatives of the loss with respect to network parameters by forming appropriate products of activations and gradients. When a parameter value is shared across multiple connections in a network, the total derivative is formed from the sum of derivatives for each of the connections. For our neural ODE, in which the same parameter vector $\mathbf{w}$ is shared throughout the network, this summation becomes an integration over t, which takes the form

Exercise 9.7

Exercise 18.7

$$\nabla_{\mathbf{w}} L = -\int_0^T \mathbf{a}(t)^{\mathrm{T}} \nabla_{\mathbf{w}} f(\mathbf{z}(t), \mathbf{w})\, \mathrm{d}t. \tag{18.26}$$

The derivatives $\nabla_{\mathbf{z}} f$ in (18.25) and $\nabla_{\mathbf{w}} f$ in (18.26) can be evaluated efficiently using automatic differentiation. Note that the above results can equally be applied to a more general neural network function $\mathbf{f}(\mathbf{z}(t), t, \mathbf{w})$ that has an explicit dependence on t in addition to the implicit dependence through $\mathbf{z}(t)$.

Section 8.2

One benefit of neural ODEs trained using the adjoint method, compared to conventional layered networks, is that there is no need to store the intermediate results of the forward propagation, and hence the memory cost is constant. Furthermore,

Telegram: @uni_k

www.konkur.in

neural ODEs can naturally handle continuous-time data in which observations occur at arbitrary times. If the error function L depends on values of $\mathbf{z}(t)$ other than the output value, then multiple runs of the reverse-model solver are required, with one run for each consecutive pair of outputs, so that the single solution is broken down into multiple consecutive solutions in order to access the intermediate states (Chen *et al.*, 2018). Note that a high level of accuracy in the solver can be used during training, with a lower accuracy, and hence fewer function evaluations, during inference in applications for which compute resources are limited.

18.3.3 Neural ODE flows

We can make use of a neural ordinary differential equation to define an alternative approach to the construction of tractable normalizing flow models. A neural ODE defines a highly flexible transformation from an input vector $\mathbf{z}(0)$ to an output vector $\mathbf{z}(T)$ in terms of a differential equation of the form

$$\frac{\mathrm{d}\mathbf{z}(t)}{\mathrm{d}t} = \mathbf{f}(\mathbf{z}(t), \mathbf{w}). \tag{18.27}$$

If we define a base distribution over the input vector $p(\mathbf{z}(0))$ then the neural ODE propagates this forward through time to give a distribution $p(\mathbf{z}(t))$ for each value of t, leading to a distribution over the output vector $p(\mathbf{z}(T))$. Chen *et al.* (2018) showed that for neural ODEs, the transformation of the density can be evaluated by integrating a differential equation given by

Exercise 18.8

$$\frac{\mathrm{d}\ln p(\mathbf{z}(t))}{\mathrm{d}t} = -\mathrm{Tr}\left(\frac{\partial \mathbf{f}}{\partial \mathbf{z}(t)}\right) \tag{18.28}$$

where $\partial\mathbf{f}/\partial\mathbf{z}$ represents the Jacobian matrix with elements $\partial f_i/\partial z_j$. This integration can be performed using standard ODE solvers. Likewise, samples from this density can be obtained by sampling from the base density $p(\mathbf{z}(0))$, which is chosen to be a simple distribution such as a Gaussian, and propagating the values to the output by integrating (18.27) again using the ODE solver. The resulting framework is known as a *continuous normalizing flow* and is illustrated in Figure 18.6. Continuous normalizing flows can be trained using the *adjoint sensitivity method* used for neural ODEs, which can be viewed as the continuous time equivalent of backpropagation.

Exercise 18.9
Section 18.3.1

Since (18.28) involves the trace of the Jacobian rather than the determinant, which arises in discrete normalizing flows, it might appear to be more computationally efficient. In general, evaluating the determinant of a $D \times D$ matrix requires $\mathcal{O}(D^3)$ operations, whereas evaluating the trace requires $\mathcal{O}(D)$ operations. However, if the determinant is lower diagonal, as in many forms of normalizing flow, then the determinant is the product of the diagonal terms and therefore also involves $\mathcal{O}(D)$ operations. Since evaluating the individual elements of the Jacobian matrix requires a separate forward propagation, which itself requires $\mathcal{O}(D)$ operations, evaluating the trace or the determinant (for a lower triangular matrix) takes $\mathcal{O}(D^2)$ operations overall. However, the cost of evaluating the trace can be reduced to $\mathcal{O}(D)$ by using *Hutchinson's trace estimator* (Grathwohl *et al.*, 2018), which for a matrix

Telegram: @uni_k

www.konkur.in

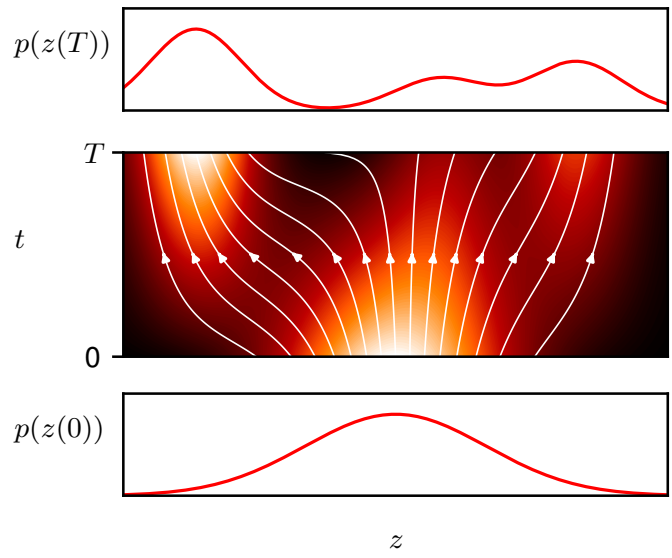


Figure 18.6 Illustration of a continuous normalizing flow showing a simple Gaussian distribution at $t = 0$ that is continuously transformed into a multimodal distribution at $t = T$. The flow lines show how points along the z-axis evolve as a function of t. Where the flow lines spread apart the density is reduced, and where they move together the density is increased.

$\mathbf{A}$ takes the form

$$\text{Tr}(\mathbf{A}) = \mathbb{E}_{\boldsymbol{\epsilon}}\left[\boldsymbol{\epsilon}^{\text{T}}\mathbf{A}\boldsymbol{\epsilon}\right] \tag{18.29}$$

where $\boldsymbol{\epsilon}$ is a random vector whose distribution has zero mean and unit covariance, for example, a Gaussian $\mathcal{N}(\mathbf{0}, \mathbf{I})$. For a specific $\boldsymbol{\epsilon}$, the matrix-vector product $\mathbf{A}\boldsymbol{\epsilon}$ can be evaluated efficiently in a single pass using reverse-mode automatic differentiation. We can then approximate the trace using a finite number of samples in the form

$$\text{Tr}(\mathbf{A}) \simeq \frac{1}{M}\sum_{m=1}^{M}\boldsymbol{\epsilon}_m^{\text{T}}\mathbf{A}\boldsymbol{\epsilon}_m. \tag{18.30}$$

In practice we can set $M = 1$ and just use a single sample, which is refreshed for each new data point. Although this is a noisy estimate, this might not be too significant since it forms part of a noisy stochastic gradient descent procedure. Importantly it is unbiased, meaning that the expectation of the estimator is equal to the true value.

Exercise 18.11

Significant improvements in training efficiency for continuous normalizing flows can be achieved using a technique called *flow matching* (Lipman *et al.*, 2022). This brings normalizing flows closer to diffusion models and avoids the need for back-propagation through the integrator while significantly reducing memory requirements and enabling faster inference and more stable training.

Chapter 20

Telegram: @uni_k

www.konkur.in

Exercises

18.1 ($\star\star$) Consider a transformation $\mathbf{x} = \mathbf{f}(\mathbf{z})$ along with its inverse $\mathbf{z} = \mathbf{g}(\mathbf{x})$. By differentiating $\mathbf{x} = \mathbf{f}(\mathbf{g}(\mathbf{x}))$, show that

$$\mathbf{JK} = \mathbf{I} \tag{18.31}$$

where $\mathbf{I}$ is the identity matrix, and $\mathbf{J}$ and $\mathbf{K}$ are matrices with elements

$$J_{ij} = \frac{\partial g_i}{\partial x_j}, \qquad K_{ij} = \frac{\partial f_i}{\partial z_j}. \tag{18.32}$$

Using the result that the determinant of a product of matrices is the product of their determinants, show that

$$\det(\mathbf{J}) = \frac{1}{\det(\mathbf{K})}. \tag{18.33}$$

Hence, show that the formula (18.1) for the transformation of a density under a change of variables can be rewritten as

$$p_{\mathbf{x}}(\mathbf{x}) = p_{\mathbf{z}}(\mathbf{g}(\mathbf{x}))\,|\det \mathbf{K}|^{-1} \tag{18.34}$$

where $\mathbf{K}$ is evaluated at $\mathbf{z} = \mathbf{g}(\mathbf{x})$.

18.2 ($\star$) Consider a sequence of invertible transformations of the form

$$\mathbf{x} = \mathbf{f}_1(\mathbf{f}_2(\cdots \mathbf{f}_{M-1}(\mathbf{f}_M(\mathbf{z}))\cdots)). \tag{18.35}$$

Show that the inverse function is given by

$$\mathbf{z} = \mathbf{f}_M^{-1}(\mathbf{f}_{M-1}^{-1}(\cdots \mathbf{f}_2^{-1}(\mathbf{f}_1^{-1}(\mathbf{x}))\cdots)). \tag{18.36}$$

18.3 ($\star$) Consider a linear change of variables of the form

$$\mathbf{x} = \mathbf{z} + \mathbf{b}. \tag{18.37}$$

Show that the Jacobian of this transformation is the identity matrix. Interpret this result by comparing the volume of a small region of $\mathbf{z}$-space with the volume of the corresponding region of $\mathbf{x}$-space.

18.4 ($\star\star$) Show that the Jacobian of the autoregressive normalizing flow transformation given by (18.18) is a lower triangular matrix. The determinant of such a matrix is given by the product of the terms on the leading diagonal and is therefore easily evaluated.

18.5 ($\star$) Consider the forward propagation equation for a residual network given by (18.21) in which we consider a small increment ϵ in the 'time' variable t:

$$\mathbf{z}^{(t+\epsilon)} = \mathbf{z}^{(t)} + \epsilon \mathbf{f}(\mathbf{z}^{(t)}, \mathbf{w}). \tag{18.38}$$

Here the additive contribution from the neural network is scaled by ϵ. Note that (18.21) corresponds to the case $\epsilon = 1$. By taking the limit $\epsilon \to 0$, derive the forward propagation differential equation given by (18.22).

Telegram: @uni_k

www.konkur.in

18.6 (⋆⋆) In this exercise and the next we provide an informal derivation of the backpropagation and gradient evaluation equations for a neural ODE. A more formal derivation of these results can be found in Chen *et al.* (2018). Write down the backpropagation equation corresponding to the forward equation (18.38). By taking the limit $\epsilon \to 0$, derive the backward propagation equation (18.25), where $\mathbf{a}(t)$ is defined by (18.24).

18.7 (⋆⋆) By making use of the result (8.10), write down an expression for the gradient of a loss function $L(\mathbf{z}(T))$ for a multilayered residual network defined by (18.38) in which all layers share the same parameter vector $\mathbf{w}$. By taking the limit $\epsilon \to 0$, derive the equation (18.26) for the derivative of the loss function.

18.8 (⋆⋆⋆) In this exercise we give an informal derivation of (18.28) for one-dimensional distributions. Consider a distribution $q(z)$ at time t that is transformed to a new distribution $p(x)$ at time $t + \delta t$ as a result of a transformation from z to x. Also consider nearby values z and $z + \Delta z$ along with corresponding values x and $x + \Delta x$ as shown in Figure 18.7. First, write down an equation that expresses that the probability mass in the interval Δz is the same as that in the interval Δx. Second, write down an equation that shows how the probability density changes in going from t to $t + \delta t$, expressed in terms of the derivative $\mathrm{d}q(t)/\,\mathrm{d}t$. Third, write down an equation for Δx in terms of Δz by introducing the function $f(z) = \mathrm{d}z/\,\mathrm{d}t$. Finally, by combining these three equations and taking the limit $\delta t \to 0$, show that

$$\frac{\mathrm{d}}{\mathrm{d}t} \ln q(z) = -f'(z), \tag{18.39}$$

which is the one-dimensional version of (18.28).

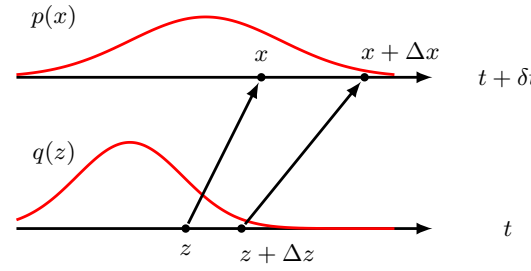


Figure 18.7 Schematic illustration of the transformation of probability densities used to derive the equation for continuous normalizing flows in one dimension.

18.9 (⋆⋆) The flow lines in Figure 18.6 were plotted by taking a set of equally spaced values and using the inverse of the cumulative distribution function at each value of t to plot the corresponding points in z-space. Show that this is equivalent to using the differential equation (18.27) to compute the flow lines where f is defined by (18.28).

18.10 (⋆⋆) Using the differential equation (18.27) write down an expression for the base density of a continuous normalizing flow in terms of the output density, expressed as an integral over t. Hence, by making use of the fact that changing the sign of a definite integral is equivalent to swapping the limits on that integral, show that the computational cost of inverting a continuous normalizing flow is the same as that needed to evaluate the forward flow.

Telegram: @uni_k

www.konkur.in

18.11 ($\star$) Show that the expectation of the right-hand side in the Hutchinson trace estimator (18.30) is equal to $\mathrm{Tr}(\mathbf{A})$ for any value of M. This shows that the estimator is unbiased.

Telegram: @uni_k

www.konkur.in



19 Autoencoders

A central goal of deep learning is to discover representations of data that are useful for one or more subsequent applications. One well-established approach to learning internal representations is called the *auto-associative neural network* or *autoencoder*. This consists of a neural network having the same number of output units as inputs and which is trained to generate an output $\mathbf{y}$ that is close to the input $\mathbf{x}$. Once trained, an internal layer within the neural network gives a representation $\mathbf{z}(\mathbf{x})$ for each new input. Such a network can be viewed as having two parts. The first is an *encoder*, which maps the input $\mathbf{x}$ into a hidden representation $\mathbf{z}(\mathbf{x})$, and the second is a *decoder*, which maps the hidden representation onto the output $\mathbf{y}(\mathbf{z})$.

If an autoencoder is to find non-trivial solutions, it is necessary to introduce some form of constraint, otherwise the network can simply copy the input values to the outputs. This constraint might be achieved, for example, by restricting the dimensionality of $\mathbf{z}$ relative to that of $\mathbf{x}$ or by requiring $\mathbf{z}$ to have a sparse represen-

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024
C. M. Bishop, H. Bishop, *Deep Learning*, https://doi.org/10.1007/978-3-031-45468-4_19

Telegram: @uni_k

www.konkur.in

tation. Alternatively, the network can be forced to discover non-trivial solutions by modifying the training process such that the network has to learn to undo corruptions to the input vectors such as additive noise or missing values. These kinds of constraint encourage the network to discover interesting structure within the data to achieve good training performance.

In this chapter, we start with deterministic autoencoders and then later generalize to stochastic models that learn an encoder distribution $p(\mathbf{z}|\mathbf{x})$ together with a decoder distribution $p(\mathbf{y}|\mathbf{z})$. These probabilistic models are known as *variational autoencoders* and represent the third of our four approaches to learning nonlinear latent variable models.

Section 16.4.4

19.1. Deterministic Autoencoders

We encountered a simple form of autoencoder when we studied principal component analysis (PCA). This is a model that makes a linear transformation of an input vector onto a lower dimensional manifold, and the resulting projection can be approximately reconstructed back in the original data space, again through a linear transformation. We can make use of the nonlinearity of neural networks to define a form of nonlinear PCA in which the latent manifold is no longer a linear subspace of the data space. This is achieved by using a network having the same number of outputs as inputs and by optimizing the weights so as to minimize some measure of the reconstruction error between inputs and outputs with respect to a set of training data.

Section 16.1

Simple autoencoders are rarely used directly in modern deep learning, as they do not provide semantically meaningful representations in the latent space and they are not able directly to generate new examples from the data distribution. However, they provide an important conceptual foundation for some of the more powerful deep generative models such as variational autoencoders.

Section 19.2

19.1.1 Linear autoencoders

Consider first a multilayer perceptron of the form shown in Figure 19.1, having D inputs, D output units, and M hidden units, with $M < D$. The targets used to train the network are simply the input vectors themselves, so that the network attempts to map each input vector onto itself. Such a network is said to form an *auto-associative* mapping. Since the number of hidden units is smaller than the number of inputs, a perfect reconstruction of all input vectors is not in general possible. We therefore determine the network parameters $\mathbf{w}$ by minimizing an error function that captures the degree of mismatch between the input vectors and their reconstructions. In particular, we choose a sum-of-squares error of the form

$$E(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N} \|\mathbf{y}(\mathbf{x}_n, \mathbf{w}) - \mathbf{x}_n\|^2. \tag{19.1}$$

If the hidden units have linear activation functions, then it can be shown that the error function has a unique global minimum and that at this minimum the network

Telegram: @uni_k

www.konkur.in

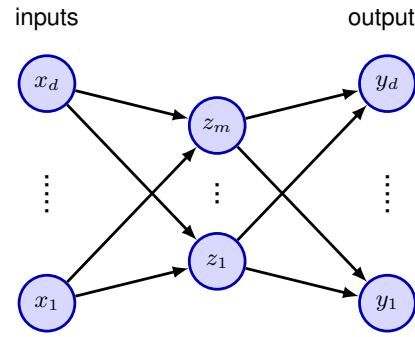


Figure 19.1 An autoencoder neural network having two layers of weights. Such a network is trained to map input vectors onto themselves by minimizing a sum-of-squares error. Even with nonlinear units in the hidden layer, such a network is equivalent to linear principal component analysis. Links representing bias parameters have been omitted for clarity.

performs a projection onto the M-dimensional subspace that is spanned by the first M principal components of the data (Bourlard and Kamp, 1988; Baldi and Hornik, 1989). Thus, the vectors of weights that lead into the hidden units in Figure 19.1 form a basis set that spans the principal subspace. Note, however, that these vectors need not be orthogonal or normalized. This result is unsurprising, since both PCA and neural networks rely on linear dimensionality reduction and minimize the same sum-of-squares error function.

It might be thought that the limitations of a linear manifold could be overcome by using nonlinear activation functions for the hidden units in the network in Figure 19.1. However, even with nonlinear hidden units, the minimum error solution is again given by the projection onto the principal component subspace (Bourlard and Kamp, 1988). There is therefore no advantage in using two-layer neural networks to perform dimensionality reduction. Standard techniques for PCA, based on singular-value decomposition (SVD), are guaranteed to give the correct solution in finite time, and they also generate an ordered set of eigenvalues with corresponding orthonormal eigenvectors.

19.1.2 Deep autoencoders

The situation is different, however, if additional nonlinear layers are included in the network. Consider the four-layer auto-associative network shown in Figure 19.2. Again, the output units are linear, and the M units in the second layer can also be linear. However, the first and third layers have sigmoidal nonlinear activation functions. The network is again trained by minimizing the error function (19.1). We can view this network as two successive functional mappings $\mathbf{F}_1$ and $\mathbf{F}_2$, as indicated in Figure 19.2. The first mapping $\mathbf{F}_1$ projects the original D-dimensional data onto an M-dimensional subspace $\mathcal{S}$ defined by the activations of the units in the second layer. Because of the first layer of nonlinear units, this mapping is very general and is not restricted to being linear. Similarly, the second half of the network defines an arbitrary functional mapping from the M-dimensional hidden space back into the original D-dimensional input space. This has a simple geometrical interpretation, as indicated for $D = 3$ and $M = 2$ in Figure 19.3.

Such a network effectively performs a nonlinear form of PCA. It has the advantage of not being limited to linear transformations, although it contains standard

Telegram: @uni_k

www.konkur.in

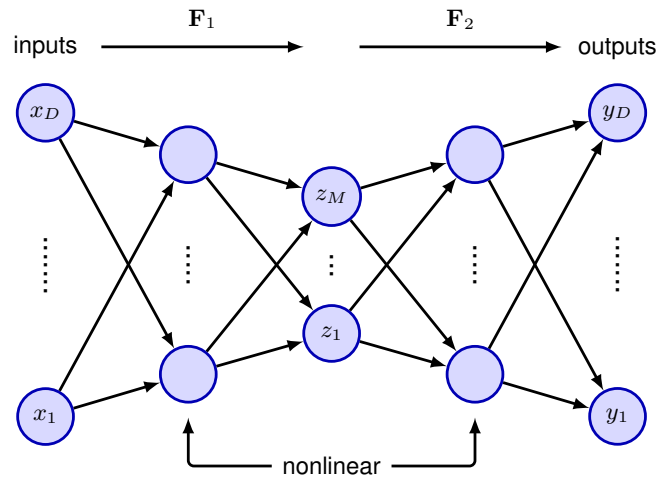


Figure 19.2 Adding extra hidden layers of nonlinear units produces an auto-associative network, which can perform a nonlinear dimensionality reduction.

PCA as a special case. However, training the network now involves a nonlinear optimization, since the error function (19.1) is no longer a quadratic function of the network parameters. Computationally intensive nonlinear optimization techniques must be used, and there is the risk of finding a sub-optimal local minimum of the error function. Also, the dimensionality of the subspace must be specified before training the network.

19.1.3 Sparse autoencoders

Instead of limiting the number of nodes in one of the hidden layers in the network, an alternative way to constrain the internal representation is to use a regularizer to encourage a sparse representation, leading to a lower effective dimensionality.

Section 9.2.2

A simple choice is the L_1 regularizer since this encourages sparseness, giving a

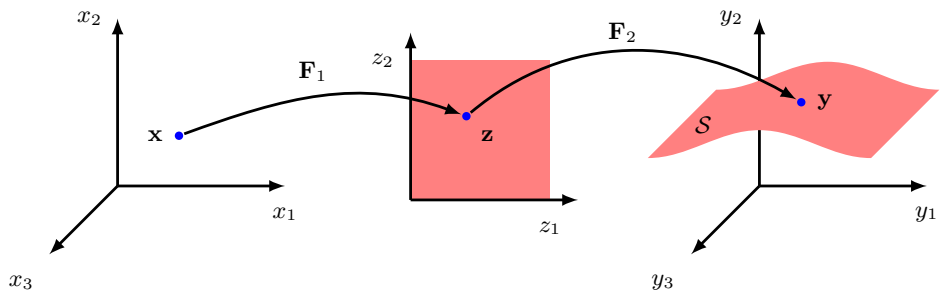


Figure 19.3 Geometrical interpretation of the mappings performed by the network in Figure 19.2 for a model with $D = 3$ inputs and $M = 2$ units in the second layer. The function $\mathbf{F}_2$ from the latent space defines the way in which the manifold $\mathcal{S}$ is embedded within the higher-dimensional data space. Since $\mathbf{F}_2$ can be nonlinear, the embedding of $\mathcal{S}$ can be non-planar, as indicated in the figure. The function $\mathbf{F}_1$ then defines a projection from the original D-dimensional data space into the M-dimensional latent space.

Telegram: @uni_k

www.konkur.in

regularized error function of the form

$$\widetilde{E}(\mathbf{w}) = E(\mathbf{w}) + \lambda \sum_{k=1}^{K} |z_k| \tag{19.2}$$

where $E(\mathbf{w})$ is the unregularized error, and the sum over k is taken over the activation values of all the units in one of the hidden layers. Note that regularization is usually applied to the parameters of a network, whereas here it is being used on the unit activations. The derivatives required for gradient descent training can be evaluated using automatic differentiation, as usual.

19.1.4 Denoising autoencoders

We have seen the importance of constraining the dimensionality of the latent space layer in a simple autoencoder to avoid the model simply learning the identity mapping. An alternative approach, that also forces the model to discover interesting internal structure in the data, is to use a *denoising autoencoder* (Vincent *et al.*, 2008). The idea is to take each input vector $\mathbf{x}_n$ and to corrupt it with noise to give a modified vector $\widetilde{\mathbf{x}}_n$ which is then input to an autoencoder to give an output $\mathbf{y}(\widetilde{\mathbf{x}}_n, \mathbf{w})$. The network is trained to reconstruct the original noise-free input vector by minimizing an error function such as the sum-of squares given by

$$E(\mathbf{w}) = \sum_{n=1}^{N} \|\mathbf{y}(\widetilde{\mathbf{x}}_n, \mathbf{w}) - \mathbf{x}_n\|^2. \tag{19.3}$$

One form of noise involves setting a randomly chosen subset of the input variables to zero. The fraction ν of such inputs represents the noise level, and lies in the range $0 \leqslant \nu \leqslant 1$. An alternative approach is to add independent zero-mean Gaussian noise to every input variable, where the scale of the noise is set by the variance of the Gaussian. By learning to denoise the input data, the network is forced to learn aspects of the structure of that data. For example, if the data comprises images, then learning that nearby pixel values are strongly correlated allows noise-corrupted pixels to be corrected.

More formally, the training of denoising autoencoders is related to score matching (Vincent, 2011) where the score is defined by $\mathbf{s}(\mathbf{x}) = \nabla_{\mathbf{x}} \ln p(\mathbf{x})$. Some intuition for this relationship is given in Figure 19.4. The autoencoder learns to reverse the distortion vector $\widetilde{\mathbf{x}}_n - \mathbf{x}_n$ and therefore learns a vector for each point in data space that points towards the manifold and therefore towards the region of high data density. The score vector $\nabla \ln p(\mathbf{x})$ is similarly a vector pointing towards the region of high data density. We will explore the relationship between score matching and denoising in more depth when we discuss diffusion models which also learn to remove noise from noise-corrupted inputs.

Section 20.3

19.1.5 Masked autoencoders

We have seen that transformer models such as BERT can learn rich internal representations of natural languages through self-supervision by masking random

Telegram: @uni_k

www.konkur.in

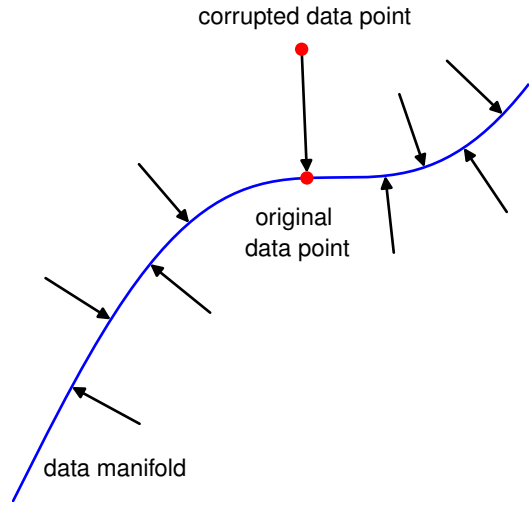


Figure 19.4 In a denoising autoencoder, data points, which are assumed to live on a lower-dimensional manifold in data space, are corrupted with additive noise. The autoencoder learns to map corrupted data points back to their original values and therefore learns a vector for each point in data space that points towards the manifold.

Section 12.2

subsets of the inputs, and it is natural to ask if a similar approach can be applied to natural images. In a *masked autoencoder* (He *et al.*, 2021), a deep network is used to reconstruct an image given a corrupted version of that image as input, similar to denoising autoencoders. However in this case, the form of corruption is masking, or dropping out, part of the input image. This technique is generally used in combination with a vision transformer architecture, as in this case, masking part of the input can be easily implemented by passing only a subset of randomly selected input patch tokens to the encoder. The overall algorithm is summarized in Figure 19.5.

Section 12.4.1

Compared to language, images have much more redundancy along with strong local correlations. Omitting a single word from a sentence can greatly increase ambiguity whereas removing a random patch from an image typically has little impact on the semantics of the image. Unsurprisingly, the best internal representations are learned when a relatively high proportion of the input image is masked, typically 75% compared with the 15% masking for BERT. In BERT the masked inputs are replaced by a fixed mask token, whereas in the masked autoencoder the masked patches are simply omitted. By omitting a large fraction of the input patches, we can save significant computation, particularly as the computation required for a training instance of a transformer scales poorly with input sequence length, thus making the masked autoencoder a good choice for pre-training large transformer encoders.

As the decoder layer is also a transformer, it needs to work in the dimensionality of the original image. Since the output of a transformer has the same dimensionality as the input, we need to restore the image dimensionality between the output of the encoder and the input of the decoder. This is achieved by reinstating the masked patches, represented by a fixed mask token vector, with each patch token augmented by positional encoding information. Due to the much higher dimensionality of the decoder representation, the decoder transformer has far fewer learnable parameters than the encoder. The output of the decoder is followed by a learnable linear layer

Telegram: @uni_k

www.konkur.in

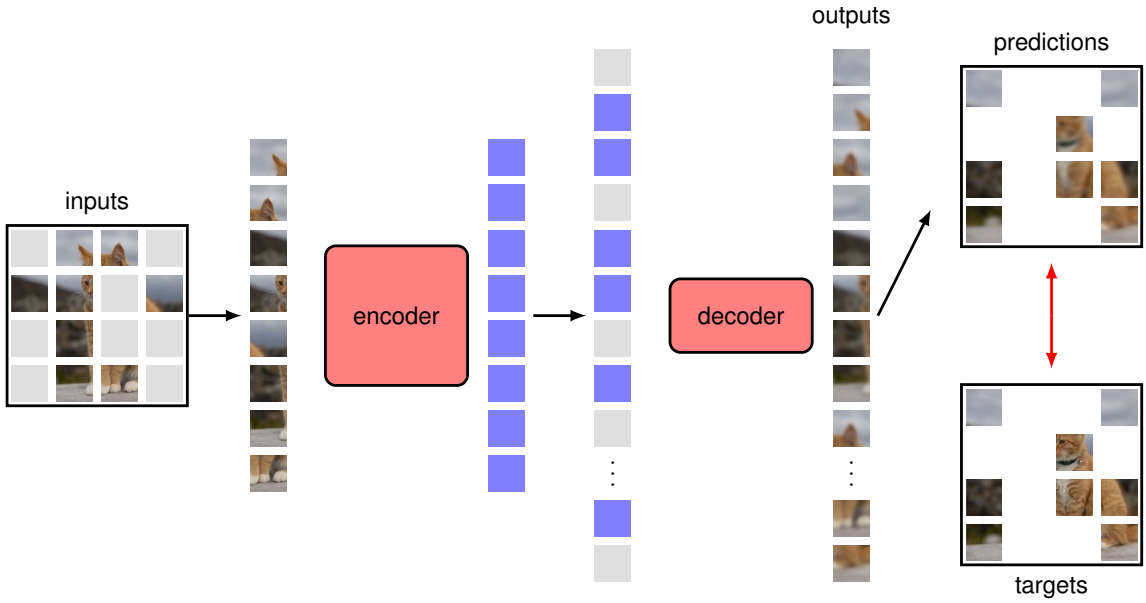


Figure 19.5 Architecture of a masked autoencoder during the training phase. Note that the target is the complement of the input as the loss is only applied on masked patches. After training, the decoder is discarded and the encoder is used to map images to an internal representation for use in downstream tasks.

that maps the output representation into the space of pixel values, and the training error function is simply the mean squared error averaged over the missing patches for each image. Examples of images reconstructed by a trained masked autoencoder are shown in Figure 19.6 and demonstrate the ability of a trained autoencoder to generate semantically plausible reconstructions. However, the ultimate goal is to learn useful internal representations for subsequent downstream tasks, for which the decoder is discarded and the encoder is applied to the full image with no masking and with a fresh set of output layers that are fine-tuned for the required application. Note also that although this algorithm was initially designed for image data, it can in theory be applied to any modality.

19.2. Variational Autoencoders

We have already seen that the likelihood function for a latent-variable model given by

$$p(\mathbf{x}|\mathbf{w}) = \int p(\mathbf{x}|\mathbf{z}, \mathbf{w})p(\mathbf{z})\,\mathrm{d}\mathbf{z}, \tag{19.4}$$

in which $p(\mathbf{x}|\mathbf{z}, \mathbf{w})$ is defined by a deep neural network, is intractable because the integral over $\mathbf{z}$ cannot be evaluated analytically. The *variational autoencoder*, or

Telegram: @uni_k

www.konkur.in



Figure 19.6 Four examples of images reconstructed using a trained masked autoencoder, in which 80% of the input patches are masked. In each case the masked image is on the left, the reconstructed image is in the centre, and the original image is on the right. [From He *et al.* (2021) with permission.]

VAE (Kingma and Welling, 2013; Rezende, Mohamed, and Wierstra, 2014; Doersch, 2016; Kingma and Welling, 2019) instead works with an approximation to this likelihood when training the model. There are three key ideas in the VAE: (i) use of the evidence lower bound (ELBO) to approximate the likelihood function, leading to a close relationship to the EM algorithm, (ii) *amortized inference* in which a second model, the encoder network, is used to approximate the posterior distributions over latent variables in the E step, rather than evaluating the posterior distribution for each data point exactly, and (iii) making the training of the encoder model tractable using the *reparameterization trick*.

Section 15.3

Consider a generative model with a conditional distribution $p(\mathbf{x}|\mathbf{z}, \mathbf{w})$ over the D-dimensional data variable $\mathbf{x}$ governed by the output of a deep neural network $\mathbf{g}(\mathbf{z}, \mathbf{w})$. For example, $\mathbf{g}(\mathbf{z}, \mathbf{w})$ might represent the mean of a Gaussian conditional distribution. Also, consider a distribution over the M-dimensional latent variable $\mathbf{z}$ that is given by a zero-mean unit-variance Gaussian:

$$p(\mathbf{z}) = \mathcal{N}(\mathbf{z}|\mathbf{0}, \mathbf{I}). \tag{19.5}$$

To derive the VAE approximation, first recall that, for an arbitrary probability distribution $q(\mathbf{z})$ over a space described by the latent variable $\mathbf{z}$, the following relationship holds:

Section 15.4

$$\ln p(\mathbf{x}|\mathbf{w}) = \mathcal{L}(\mathbf{w}) + \mathrm{KL}\left(q(\mathbf{z})\|p(\mathbf{z}|\mathbf{x}, \mathbf{w})\right) \tag{19.6}$$

where $\mathcal{L}$ is the *evidence lower bound*, or ELBO, also known as the *variational lower bound*, given by

$$\mathcal{L}(\mathbf{w}) = \int q(\mathbf{z}) \ln \left\{ \frac{p(\mathbf{x}|\mathbf{z}, \mathbf{w})p(\mathbf{z})}{q(\mathbf{z})} \right\} \mathrm{d}\mathbf{z} \tag{19.7}$$

Telegram: @uni_k

www.konkur.in

and the Kullback–Leibler divergence $\mathrm{KL}\,(\cdot\|\cdot)$ is defined by

$$\mathrm{KL}\,(q(\mathbf{z})\|p(\mathbf{z}|\mathbf{x},\mathbf{w})) = -\int q(\mathbf{z})\ln\left\{\frac{p(\mathbf{z}|\mathbf{x},\mathbf{w})}{q(\mathbf{z})}\right\}\mathrm{d}\mathbf{z}. \tag{19.8}$$

Because the Kullback–Leibler divergence satisfies $\mathrm{KL}\,(q\|p) \geqslant 0$, it follows that

$$\ln p(\mathbf{x}|\mathbf{w}) \geqslant \mathcal{L} \tag{19.9}$$

and so $\mathcal{L}$ is a lower bound on $\ln p(\mathbf{x}|\mathbf{w})$. Although the log likelihood $\ln p(\mathbf{x}|\mathbf{w})$ is intractable, we will see how the lower bound can be evaluated using a Monte Carlo estimate. Hence it provides an approximation to the true log likelihood.

Now consider a set of training data points $\mathcal{D} = \{\mathbf{x}_1,\ldots,\mathbf{x}_N\}$, which are assumed to be drawn independently from the model distribution $p(\mathbf{x})$. The log likelihood function for this data set is given by

$$\ln p(\mathcal{D}|\mathbf{w}) = \sum_{n=1}^{N}\mathcal{L}_n + \sum_{n=1}^{N}\mathrm{KL}\,(q_n(\mathbf{z}_n)\|p(\mathbf{z}_n|\mathbf{x}_n,\mathbf{w})) \tag{19.10}$$

where

$$\mathcal{L}_n = \int q_n(\mathbf{z}_n)\ln\left\{\frac{p(\mathbf{x}_n|\mathbf{z}_n,\mathbf{w})p(\mathbf{z}_n)}{q_n(\mathbf{z}_n)}\right\}\mathrm{d}\mathbf{z}_n. \tag{19.11}$$

Note that this introduces a separate latent variable $\mathbf{z}_n$ corresponding to each data vector $\mathbf{x}_n$, as we saw with mixture models and with the probabilistic PCA model. Consequently, each latent variable has its own independent distribution $q_n(\mathbf{z}_n)$, each of which can be optimized separately.

Section 15.2 *Section 16.2*

Since (19.10) holds for any choice of the distributions $q_n(\mathbf{z})$, we can choose the distributions that maximize the bound $\mathcal{L}_n$, or equivalently the distributions that minimize the Kullback–Leibler divergences $\mathrm{KL}\,(q_n(\mathbf{z}_n)\|p(\mathbf{z}_n|\mathbf{x}_n,\mathbf{w}))$. For the simple Gaussian mixture and probabilistic PCA models considered previously, we were able to evaluate these posterior distributions exactly in the E step of the EM algorithm, which corresponds to setting each $q_n(\mathbf{z}_n)$ equal to the corresponding posterior distribution $p(\mathbf{z}_n|\mathbf{x}_n,\mathbf{w})$. This gives zero Kullback–Leibler divergence, and hence the lower bound is equal to the true log likelihood. The interpretation of the posterior distribution is illustrated in Figure 19.7 using the simple example introduced earlier in the context of generative adversarial networks.

Section 16.4.1

The exact posterior distribution of $\mathbf{z}_n$ is given from Bayes' theorem by

$$p(\mathbf{z}_n|\mathbf{x}_n,\mathbf{w}) = \frac{p(\mathbf{x}_n|\mathbf{z}_n,\mathbf{w})p(\mathbf{z}_n)}{p(\mathbf{x}_n|\mathbf{w})}. \tag{19.12}$$

The numerator is straightforward to evaluate for our deep generative model. However, we see that the denominator is given by the likelihood function, which as we have already noted, is intractable. We therefore need to find an approximation to the posterior distribution. In principle, we could consider a separate parameterized model for each of the distributions $q_n(\mathbf{z}_n)$ and optimize each model numerically,

Telegram: @uni_k

www.konkur.in

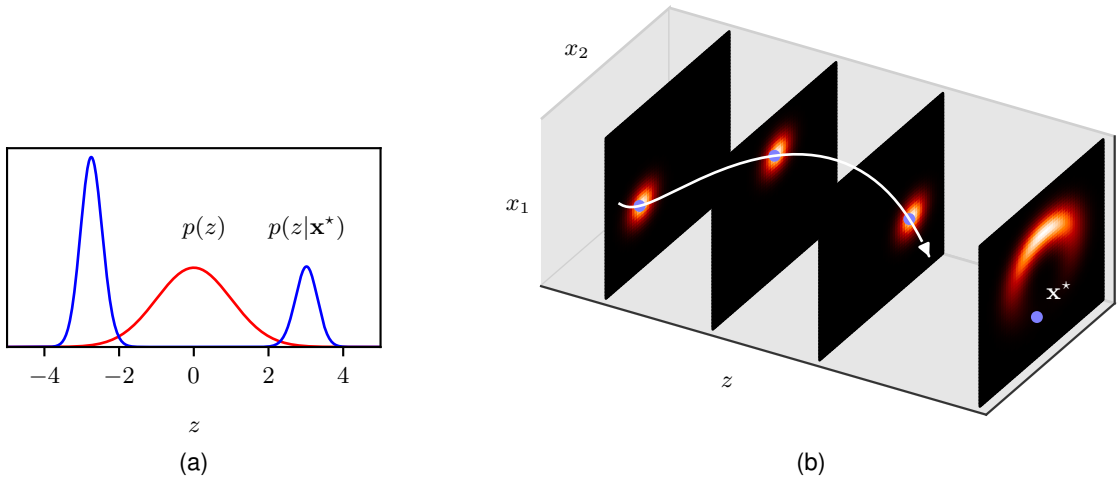


Figure 19.7 Evaluation of the posterior distribution for the same model as shown in Figure 16.13. The marginal distribution $p(\mathbf{x})$, shown in the right-most plot in (b), has a banana shape, and the specific data point $\mathbf{x}^\star$ is closer to the horns of the shape than to the middle. Consequently the posterior distribution $p(z|\mathbf{x}^\star)$, shown in (a), is bimodal, even though the prior distribution $p(\mathbf{z})$ is unimodal. [Based on Prince (2020) with permission.]

but this would be computationally very expensive, especially for large data sets, and moreover we would have to re-evaluate the distributions after every update of $\mathbf{w}$. Instead, we turn now to a different, more efficient approximation framework based on the introduction of a second neural network.

19.2.1 Amortized inference

In the variational autoencoder, instead of trying to evaluate a separate posterior distribution $p(\mathbf{z}_n|\mathbf{x}_n, \mathbf{w})$ for each of the data points $\mathbf{x}_n$ individually, we train a single neural network, called the *encoder network*, to approximate all these distributions. This technique is called *amortized inference* and requires an encoder that produces a single distribution $q(\mathbf{z}|\mathbf{x}, \boldsymbol{\phi})$ that is conditioned on $\mathbf{x}$, where $\boldsymbol{\phi}$ represents the parameters of the network. The objective function, given by the evidence lower bound, now has a dependence on $\boldsymbol{\phi}$ as well as on $\mathbf{w}$, and we use gradient-based optimization methods to maximize the bound jointly with respect to both sets of parameters.

A VAE therefore comprises two neural networks that have independent parameters but which are trained jointly: an encoder network that takes a data vector and maps it to a latent space, and the original network that takes a latent space vector and maps it back to the data space and which we can therefore interpret as a *decoder network*. This like the simple neural network autoencoder model, except that we now define a probability distribution over the latent space. We will see that the encoder calculates an approximate probabilistic inverse of the decoder according to Bayes' theorem.

Section 19.1

A typical choice for the encoder is a Gaussian distribution with a diagonal covariance matrix whose mean and variance parameters, μ_j and σ_j^2, are given by the

Telegram: @uni_k

www.konkur.in

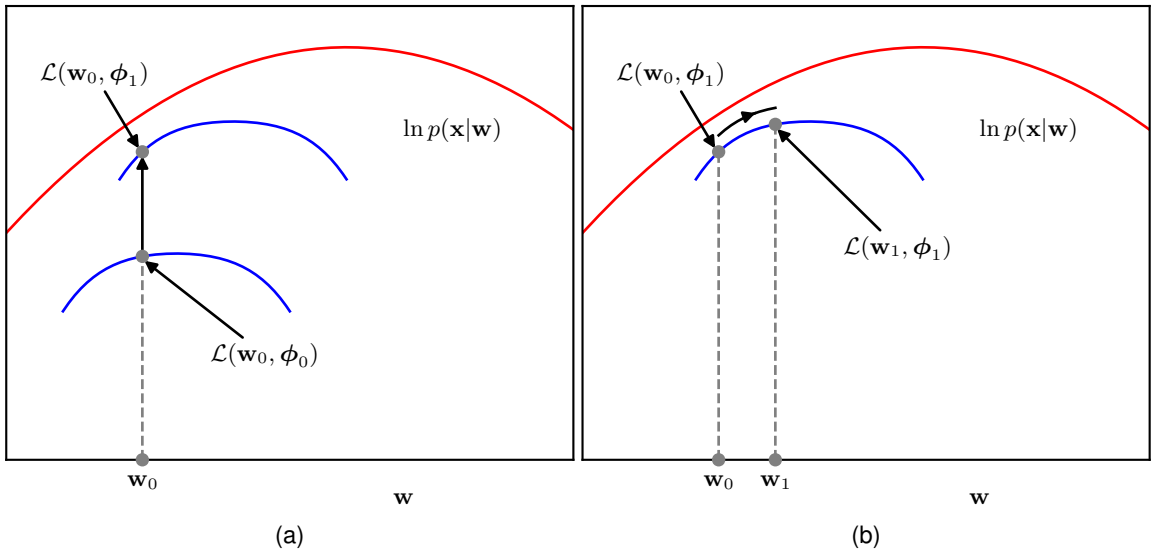


Figure 19.8 Illustration of the optimization of the ELBO (evidence lower bound). (a) For a given value $\mathbf{w}_0$ of the decoder network parameters $\mathbf{w}$, we can increase the bound by optimizing the parameters $\boldsymbol{\phi}$ of the encoder network. (b) For a given value of $\boldsymbol{\phi}$, we can increase the value of the ELBO function by optimizing $\mathbf{w}$. Note that the ELBO function, shown by the blue curves, always lies somewhat below the log likelihood function, shown in red, because the encoder network is generally not able to match the true posterior distribution exactly.

outputs of a neural network that takes $\mathbf{x}$ as input:

$$q(\mathbf{z}|\mathbf{x}, \boldsymbol{\phi}) = \prod_{j=1}^{M} \mathcal{N}\left(z_j|\mu_j(\mathbf{x}, \boldsymbol{\phi}), \sigma_j^2(\mathbf{x}, \boldsymbol{\phi})\right). \tag{19.13}$$

Note that the means $\mu_j(\mathbf{x}, \boldsymbol{\phi})$ lie in the range $(-\infty, \infty)$, and so the corresponding output-unit activation functions can be linear, whereas the variances $\sigma_j^2(\mathbf{x}, \boldsymbol{\phi})$ must be non-negative and so the associated output units typically use $\exp(\cdot)$ as their activation function.

The goal is to use gradient-based optimization to maximize the bound with respect to both sets of parameters $\boldsymbol{\phi}$ and $\mathbf{w}$, typically by using stochastic gradient descent based on mini-batches. Although we optimize the parameters jointly, conceptually we could imagine alternating between optimizing $\boldsymbol{\phi}$ and optimizing $\mathbf{w}$, in the spirit of the EM algorithm, as illustrated in Figure 19.8.

A key difference compared to EM is that, for a given value of $\mathbf{w}$, optimizing with respect to the parameters $\boldsymbol{\phi}$ of the encoder does not in general reduce the Kullback-Leibler divergence to zero, because the encoder network is not a perfect predictor of the posterior latent distribution and so there is a residual gap between the lower bound and the true log likelihood. Although the encoder is very flexible, since it is based on a deep neural network, it is not expected to model the true posterior distribution exactly because (i) the true conditional posterior distribution will not be

Telegram: @uni_k

www.konkur.in

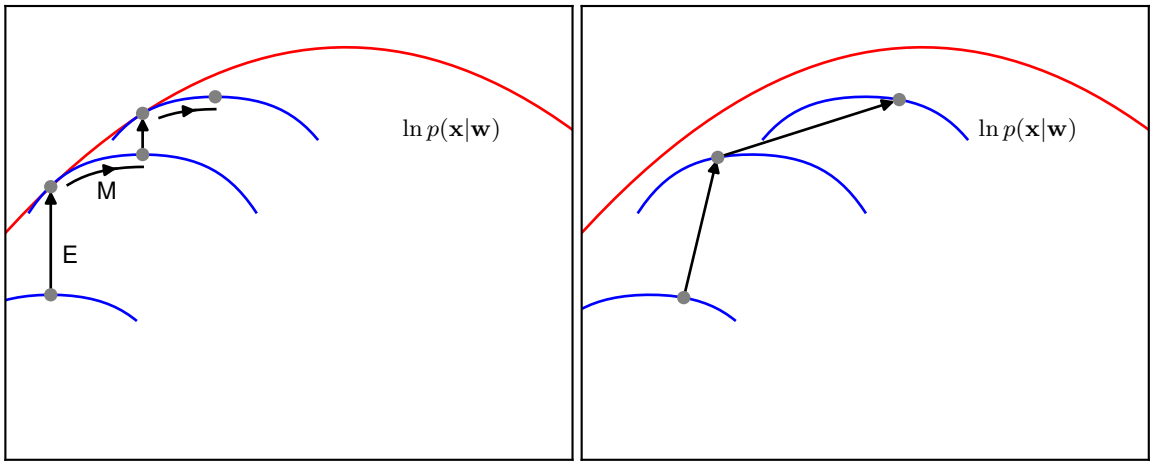


Figure 19.9 Comparison of the EM algorithm with ELBO optimization in a VAE. (a) In the EM algorithm we alternate between updating the variational posterior distribution in the E step, and the model parameters in the M step. When the E step is exact, the gap between the lower bound and the log likelihood is reduced to zero after each E step. (b) In the VAE we perform joint optimization of the encoder network parameters $\boldsymbol{\phi}$ (analogous to the E step) and the decoder network parameters $\mathbf{w}$ (analogous to the M step).

a factorized Gaussian, (ii) even a large neural network has limited flexibility, and (iii) the training process is only an approximate optimization. The relation between the EM algorithm and ELBO optimization is summarized in Figure 19.9.

19.2.2 The reparameterization trick

Unfortunately, as it stands, the lower bound (19.11) is still intractable to compute because it involves integrals over the latent variables $\{\mathbf{z}_n\}$ in which the integrand has a complicated dependence on the latent variables because of the decoder network. For data point $\mathbf{x}_n$ we can write the contribution to the lower bound in the form

$$\begin{aligned}\mathcal{L}_n(\mathbf{w}, \boldsymbol{\phi}) &= \int q(\mathbf{z}_n|\mathbf{x}_n, \boldsymbol{\phi}) \ln \left\{ \frac{p(\mathbf{x}_n|\mathbf{z}_n, \mathbf{w})p(\mathbf{z}_n)}{q(\mathbf{z}_n|\mathbf{x}_n, \boldsymbol{\phi})} \right\} \mathrm{d}\mathbf{z}_n \\ &= \int q(\mathbf{z}_n|\mathbf{x}_n, \boldsymbol{\phi}) \ln p(\mathbf{x}_n|\mathbf{z}_n, \mathbf{w}) \, \mathrm{d}\mathbf{z}_n - \mathrm{KL}(q(\mathbf{z}_n|\mathbf{x}_n, \boldsymbol{\phi}) \| p(\mathbf{z}_n)). \end{aligned} \tag{19.14}$$

The second term on the right-hand side is a Kullback–Leibler divergence between two Gaussian distributions and can be evaluated analytically:

Exercise 2.27

$$\mathrm{KL}\left(q(\mathbf{z}_n|\mathbf{x}_n, \boldsymbol{\phi}) \| p(\mathbf{z}_n)\right) = \frac{1}{2} \sum_{j=1}^{M} \left\{1 + \ln \sigma_j^2(\mathbf{x}_n) - \mu_j^2(\mathbf{x}_n) - \sigma_j^2(\mathbf{x}_n)\right\}. \tag{19.15}$$

Telegram: @uni_k

www.konkur.in

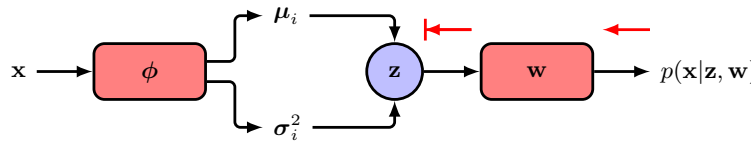


Figure 19.10 When the ELBO is estimated by fixing the latent variable $\mathbf{z}$ to a sampled value this blocks backpropagation of the error signal to the encoder network.

For the first term in (19.14), we could try to approximate the integral over $\mathbf{z}_n$ with a simple Monte Carlo estimator:

$$\int q(\mathbf{z}_n|\mathbf{x}_n, \boldsymbol{\phi}) \ln p(\mathbf{x}_n|\mathbf{z}_n, \mathbf{w}) \, \mathrm{d}\mathbf{z}_n \simeq \frac{1}{L} \sum_{l=1}^{L} \ln p(\mathbf{x}_n|\mathbf{z}_n^{(l)}, \mathbf{w}) \tag{19.16}$$

where $\{\mathbf{z}_n^{(l)}\}$ are samples drawn from the encoder distribution $q(\mathbf{z}_n|\mathbf{x}_n, \boldsymbol{\phi})$. This is easily differentiated with respect to $\mathbf{w}$, but the gradient with respect to $\boldsymbol{\phi}$ is problematic because changes to $\boldsymbol{\phi}$ will change the distribution $q(\mathbf{z}_n|\mathbf{x}_n, \boldsymbol{\phi})$ from which the samples are drawn and yet these samples are fixed values so that we do not have a way to obtain the derivatives of these samples with respect to $\boldsymbol{\phi}$. Conceptually, we can think of the process of fixing $\mathbf{z}_n$ to a specific sample value as blocking the back-propagation of the error signal to the encoder network, as illustrated in Figure 19.10.

We can resolve this by making use of the *reparameterization trick* in which we reformulate the Monte Carlo sampling procedure such that derivatives with respect to $\boldsymbol{\phi}$ can be calculated explicitly. First, note that if ϵ is a Gaussian random variable with zero mean and unit variance, then the quantity

$$z = \sigma\epsilon + \mu \tag{19.17}$$

Exercise 19.2 will have a Gaussian distribution, with mean μ and variance σ^2. We now apply this to the samples in (19.16) in which μ and σ are defined by the outputs $\mu_j(\mathbf{x}_n, \boldsymbol{\phi})$ and $\sigma_j^2(\mathbf{x}_n, \boldsymbol{\phi})$ of the encoder network, which represent the means and variances in distribution (19.13). Instead of drawing samples of $\mathbf{z}_n$ directly, we draw samples for ϵ and use (19.17) to evaluate corresponding samples for $\mathbf{z}_n$:

$$z_{nj}^{(l)} = \mu_j(\mathbf{x}_n, \boldsymbol{\phi})\epsilon_{nj}^{(l)} + \sigma_j^2(\mathbf{x}_n, \boldsymbol{\phi}) \tag{19.18}$$

where $l = 1, \ldots, L$ indexes the samples. This makes the dependence on $\boldsymbol{\phi}$ explicit and allows gradients with respect to $\boldsymbol{\phi}$ to be evaluated, as illustrated in Figure 19.11. The reparameterization trick can be extended to other distributions but is limited to continuous variables. There are techniques to evaluate gradients directly without the reparameterization trick (Williams, 1992), but these estimators have high variance, and so reparameterization can also be viewed as a variance reduction technique.

The full error function for the VAE, using our specific modelling assumptions, therefore becomes

$$\mathcal{L} = \sum_n \left\{ \frac{1}{2} \sum_{j=1}^{M} \{1 + \ln \sigma_{nj}^2 - \mu_{nj}^2 - \sigma_{nj}^2\} + \frac{1}{L} \sum_{l=1}^{L} \ln p(\mathbf{x}_n|\mathbf{z}_n^{(l)}, \mathbf{w}) \right\} \tag{19.19}$$

Telegram: @uni_k

www.konkur.in

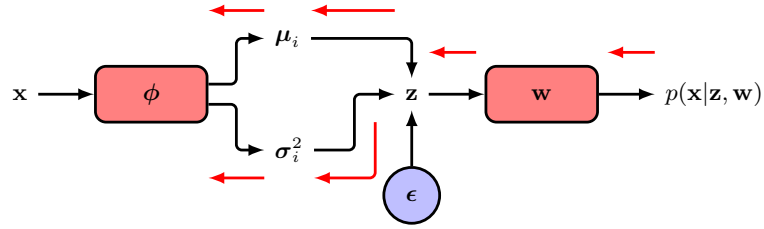


Figure 19.11 The reparameterization trick replaces a direct sample of $\mathbf{z}$ by one that is calculated from a sample of an independent random variable ϵ, thereby allowing the error signal to be back-propagated to the encoder network. The resulting model can be trained using gradient-based optimization to learn the parameters of both the encoder and decoder networks.

where $\mathbf{z}_n^{(l)}$ has components $z_{nj}^{(l)} = \sigma_{nj}\epsilon^{(l)} + \mu_{nj}$, in which $\mu_{nj} = \mu_j(\mathbf{x}_n, \boldsymbol{\phi})$ and $\sigma_{nj} = \sigma_j(\mathbf{x}_n, \boldsymbol{\phi})$, and the summation over n in (19.19) is over the data points in a mini-batch. The number of samples L, for each data point $\mathbf{x}_n$, is typically set to 1, so that only a single sample is used. Although this gives a noisy estimate of the bound, it forms part of the stochastic gradient optimization step, which is already noisy, and overall leads to more efficient optimization.

We can summarize VAE training as follows. For each data point in a mini-batch, forward propagate through the encoder network to evaluate the means and variances of the approximate latent distribution, sample from this distribution using the reparameterization trick, and then propagate these samples through the decoder network to evaluate the ELBO (19.19). The gradients with respect to $\mathbf{w}$ and $\boldsymbol{\phi}$ are then evaluated using automatic differentiation. VAE training is summarized in Algorithm 19.1, where, for clarity, we have omitted that this would generally be done using mini-batches. Once the model is trained, the encoder network is discarded and new data points are generated by sampling from the prior $p(\mathbf{z})$ and forward propagating through the decoder network to obtain samples in the data space.

After training we might want to assess how well the model represents a new test point $\widehat{\mathbf{x}}$. Since the log likelihood is intractable, we can use the lower bound $\mathcal{L}$ as an approximation. To estimate this we can sample from $q(\mathbf{z}|\widehat{\mathbf{x}}, \boldsymbol{\phi})$ as this gives more accurate estimates than sampling from $p(\mathbf{z})$.

Section 10.5.3

There are many variants of VAEs. When applied to image data, the encoder is typically based on convolutions and the decoder based on transpose convolutions. In a *conditional VAE* both the encoder and decoder take a conditioning variable $\mathbf{c}$ as an additional input. For example, we might want to generate images of objects, in which $\mathbf{c}$ represents the object class. The latent-space prior distribution $p(\mathbf{z})$ can again be a simple Gaussian, or it can be extended to a conditional distribution $p(\mathbf{z}|\mathbf{c})$ given by another neural network. Training and testing proceed as before.

Note that the first term in the ELBO (19.14) encourages the encoder distribution $q(\mathbf{z}|\mathbf{x}, \boldsymbol{\phi})$ to be close to the prior $p(\mathbf{z})$, and so the decoder model is encouraged to produce realistic outputs when the trained model is run generatively by sampling from $p(\mathbf{z})$. When training VAEs, a problem can arise in which the variational distribution $q(\mathbf{z}|\mathbf{x}, \boldsymbol{\phi})$ converges to the prior distribution $p(\mathbf{z})$ and therefore becomes uninformative because it no longer depends on $\mathbf{x}$. In effect the latent code is ig-

Telegram: @uni_k

www.konkur.in

Algorithm 19.1: Variational autoencoder training

Input: Training data set $\mathcal{D} = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$
Encoder network $\{\mu_j(\mathbf{x}_n, \boldsymbol{\phi}), \sigma_j^2(\mathbf{x}_n, \boldsymbol{\phi})\}, \quad j \in \{1, \ldots, M\}$
Decoder network $\mathbf{g}(\mathbf{z}, \mathbf{w})$
Initial weight vectors $\mathbf{w}, \boldsymbol{\phi}$
Learning rate η

Output: Final weight vectors $\mathbf{w}, \boldsymbol{\phi}$

---

repeat
  $\mathcal{L} \leftarrow 0$
  for $j \in \{1, \ldots, M\}$ **do**
    $\epsilon_{nj} \sim \mathcal{N}(0, 1)$
    $z_{nj} \leftarrow \mu_j(\mathbf{x}_n, \boldsymbol{\phi})\epsilon_{nj} + \sigma_j^2(\mathbf{x}_n, \boldsymbol{\phi})$
    $\mathcal{L} \leftarrow \mathcal{L} + \frac{1}{2}\left\{1 + \ln \sigma_{nj}^2 - \mu_{nj}^2 - \sigma_{nj}^2\right\}$
  end for
  $\mathcal{L} \leftarrow \mathcal{L} + \ln p(\mathbf{x}_n|\mathbf{z}_n, \mathbf{w})$
  $\mathbf{w} \leftarrow \mathbf{w} + \eta\nabla_{\mathbf{w}}\mathcal{L}$ `// Update decoder weights`
  $\boldsymbol{\phi} \leftarrow \boldsymbol{\phi} + \eta\nabla_{\boldsymbol{\phi}}\mathcal{L}$ `// Update encoder weights`
until converged
return $\mathbf{w}, \boldsymbol{\phi}$

nored. This is known as *posterior collapse*. A symptom of this is that if we take an input and encode it and then decode it, we get a poor reconstruction that looks blurry. In this case the Kullback–Leibler divergence $\mathrm{KL}(q(\mathbf{z}|\mathbf{x}, \boldsymbol{\phi})\|p(\mathbf{z}))$ is close to zero.

A different problem occurs when the latent code is not compressed, which is characterized by highly accurate reconstructions, but such that outputs generated by sampling $p(\mathbf{z})$ and passing the samples through the decoder network have poor quality and do not resemble the training data. In this case the Kullback–Leibler divergence is relatively large, and because the trained system has a variational distribution that is very different from the prior, samples from the prior do not generate realistic outputs.

Both problems can be addressed by introducing a coefficient β in front of the first term in (19.14) to control the regularization effectiveness of the Kullback–Leibler divergence, where typically $\beta > 1$ (Higgins *et al.*, 2017). If the reconstructions look poor then β can be increased, whereas if the samples look poor then β can be decreased. The value of β can also be set to follow an annealing schedule in which it starts with a small value and is gradually increased during training.

Finally, note that we have considered a decoder network $\mathbf{g}(\mathbf{z}, \mathbf{w})$ that represents

Telegram: @uni_k

www.konkur.in

the mean of a Gaussian output distribution. We can extend the VAE to include outputs representing the variance of the Gaussian or, more generally, the parameters that characterize other more complex distributions.

Section 6.5

Exercises

19.1 (⋆⋆) Show that, for any distribution $q(\mathbf{z}|\boldsymbol{\phi})$ and any function $G(\mathbf{z})$, the following relation holds:

$$\nabla_{\boldsymbol{\phi}} \int q(\mathbf{z}|\boldsymbol{\phi}) G(\mathbf{z}) \, \mathrm{d}\mathbf{z} = \int q(\mathbf{z}|\boldsymbol{\phi}) G(\mathbf{z}) \nabla_{\boldsymbol{\phi}} \ln q(\mathbf{z}|\boldsymbol{\phi}) \, \mathrm{d}\mathbf{z}. \tag{19.20}$$

Hence, show that the left-hand side of (19.20) can be approximated by the following Monte Carlo estimator:

$$\nabla_{\boldsymbol{\phi}} \int q(\mathbf{z}|\boldsymbol{\phi}) G(\mathbf{z}) \, \mathrm{d}\mathbf{z} \simeq \sum_i G(\mathbf{z}^{(i)}) \nabla_{\boldsymbol{\phi}} \ln q(\mathbf{z}^{(i)}|\boldsymbol{\phi}) \tag{19.21}$$

where the samples $\{\mathbf{z}^{(i)}\}$ are drawn independently from the distribution $q(\mathbf{z}|\boldsymbol{\phi})$. Verify that this estimator is unbiased, i.e., that the average value of the right-hand side of (19.21), averaged over the distribution of the samples, is equal to the left-hand side. In principle, by setting $G(\mathbf{z}) = p(\mathbf{x}|\mathbf{z}, \mathbf{w})$, this result would allow the gradient of the second term on the right-hand side of (19.14) with respect to $\boldsymbol{\phi}$ to be evaluated without making use of the reparameterization trick. Also, because this method is unbiased, it will give the exact answer in the limit of an infinite number of samples. However, the reparameterization trick is more efficient, meaning that fewer samples are needed to get good accuracy, because it directly computes the change of $p(\mathbf{x}|\mathbf{z}, \mathbf{w})$ due to the change in $\mathbf{z}$ that results from a change in $\boldsymbol{\phi}$.

19.2 (⋆) Verify that if ϵ has a zero-mean unit-variance Gaussian distribution, then the variable z in (19.17) will have a Gaussian distribution with mean μ and variance σ^2.

19.3 (⋆⋆) In this exercise we extend the diagonal covariance VAE encoder network (19.13) to one with a general covariance matrix. Consider a K-dimensional random vector drawn from a simple Gaussian:

$$\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{z}|\mathbf{0}, \mathbf{I}), \tag{19.22}$$

which is then linearly transformed using the relation

$$\mathbf{z} = \boldsymbol{\mu} + \mathbf{L}\boldsymbol{\epsilon} \tag{19.23}$$

where $\mathbf{L}$ is a lower-triangular matrix (i.e., a $K \times K$ matrix with all elements above the leading diagonal being zero). Show that $\mathbf{z}$ has a distribution $\mathcal{N}(\mathbf{z}|\boldsymbol{\mu}, \boldsymbol{\Sigma})$, and write down an expression for $\boldsymbol{\Sigma}$ in terms of $\mathbf{L}$. Explain why the diagonal elements of $\mathbf{L}$ must be non-negative. Describe how $\boldsymbol{\mu}$ and $\mathbf{L}$ can be expressed as the outputs of a neural network, and discuss suitable choices for output-unit activation functions.

Telegram: @uni_k

www.konkur.in

19.4 ($\star\star$) Evaluate the Kullback–Leibler divergence term in (19.14). Hence, show how the gradients of this term with respect to $\mathbf{w}$ and $\boldsymbol{\phi}$ can be evaluated for training the encoder and decoder networks.

19.5 ($\star$) We have seen that the ELBO given by (19.11) can be written in the form (19.14). Show that it can also be written as

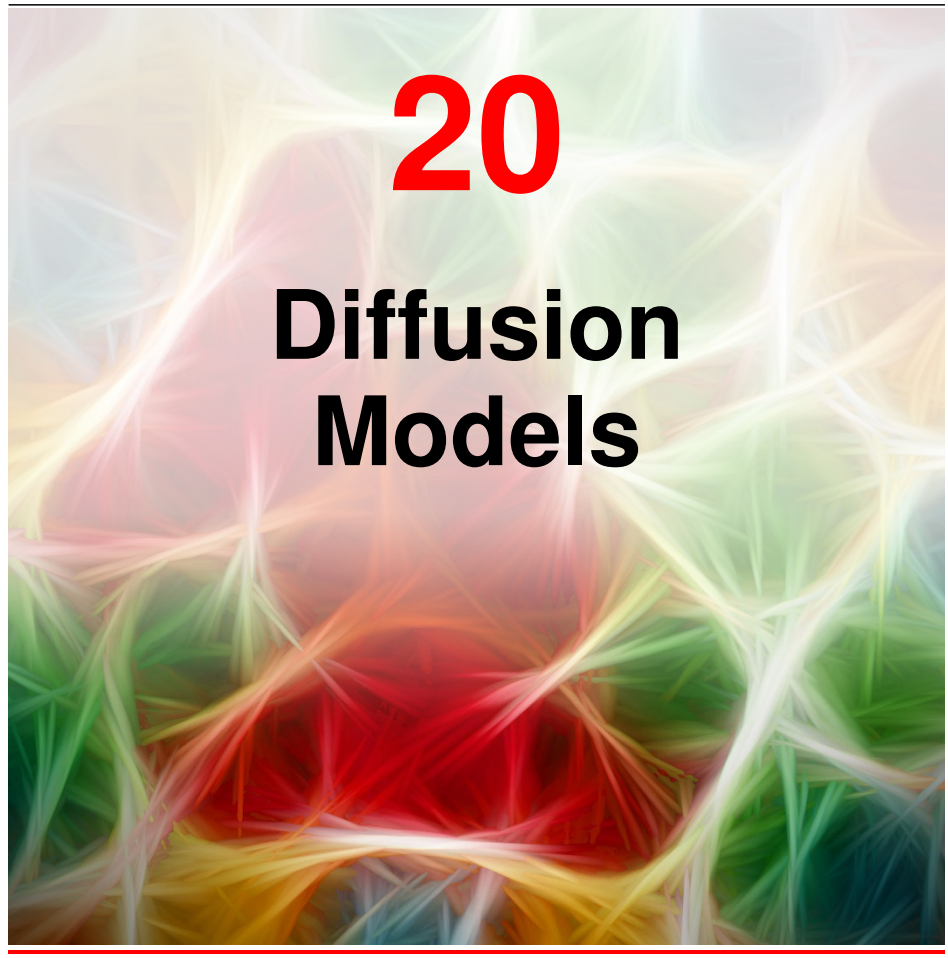
$$\mathcal{L}_n(\mathbf{w}, \boldsymbol{\phi}) = \int q(\mathbf{z}_n|\mathbf{x}_n, \boldsymbol{\phi}) \ln \{p(\mathbf{x}_n|\mathbf{z}_n, \mathbf{w})p(\mathbf{z}_n)\} \, \mathrm{d}\mathbf{z}_n - \int q(\mathbf{z}_n|\mathbf{x}_n, \boldsymbol{\phi}) \ln q(\mathbf{z}_n|\mathbf{x}_n, \boldsymbol{\phi}) \, \mathrm{d}\mathbf{z}_n. \tag{19.24}$$

19.6 ($\star$) Show that the ELBO given by (19.11) can be written in the form

$$\mathcal{L}_n(\mathbf{w}, \boldsymbol{\phi}) = \int q(\mathbf{z}_n|\mathbf{x}_n, \boldsymbol{\phi}) \ln p(\mathbf{z}_n) \, \mathrm{d}\mathbf{z}_n + \int q(\mathbf{z}_n|\mathbf{x}_n, \boldsymbol{\phi}) \ln \left\{ \frac{p(\mathbf{x}_n|\mathbf{z}_n, \mathbf{w})}{q(\mathbf{z}_n|\mathbf{x}_n, \boldsymbol{\phi})} \right\} \mathrm{d}\mathbf{z}_n. \tag{19.25}$$

Telegram: @uni_k

www.konkur.in



20 Diffusion Models

We have seen that a powerful way to construct rich generative models is to introduce a distribution $p(\mathbf{z})$ over a latent variable $\mathbf{z}$, and then to transform $\mathbf{z}$ into the data space $\mathbf{x}$ using a deep neural network. It is sufficient to use a simple, fixed distribution for $p(\mathbf{z})$, such as a Gaussian $\mathcal{N}(\mathbf{z}|\mathbf{0}, \mathbf{I})$, since the generality of the neural network transforms this into a highly flexible family of distributions over $\mathbf{x}$. In previous chapters we have explored several models which fit within this framework but which take different approaches to defining and training the deep neural network, based on generative adversarial networks, variational autoencoders, and normalizing flows.

Section 16.4.4

In this chapter we discuss a fourth class of models within this general framework, known as *diffusion models*, also called *denoising diffusion probabilistic models*, or DDPMs (Sohl-Dickstein *et al.*, 2015; Ho, Jain, and Abbeel, 2020), which have emerged as the state of the art for many applications. For illustration we will focus on models of image data although the framework has much broader applicabil-

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024
C. M. Bishop, H. Bishop, *Deep Learning*, https://doi.org/10.1007/978-3-031-45468-4_20

Telegram: @uni_k

www.konkur.in

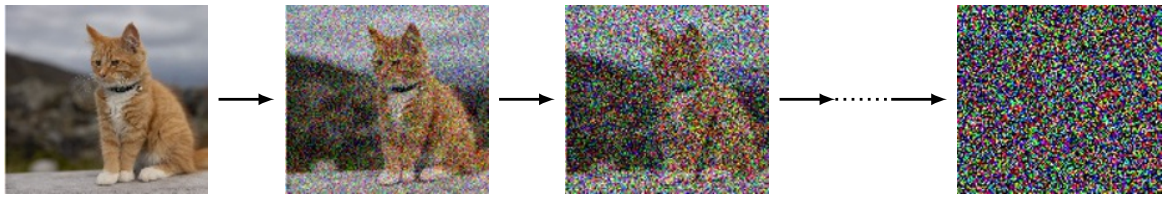


Figure 20.1 Illustration of the encoding process in a diffusion model showing an image $\mathbf{x}$ that is gradually corrupted with multiple stages of additive Gaussian noise giving a sequence of increasingly noisy images. After a large number T of steps the result is indistinguishable from a sample drawn from a Gaussian distribution. A deep neural network is then trained to reverse this process.

ity. The central idea is to take each training image and to corrupt it using a multi-step noise process to transform it into a sample from a Gaussian distribution. This is illustrated in Figure 20.1. A deep neural network is then trained to invert this process, and once trained the network can then generate new images starting with samples from a Gaussian as input.

Section 19.2

Diffusion models can be viewed as a form of hierarchical variational autoencoder in which the encoder distribution is fixed, and defined by the noise process, and only the generative distribution is learned (Luo, 2022). They are easy to train, they scale well on parallel hardware, and they avoid the challenges and instabilities of adversarial training while producing results that have quality comparable to, or better than, generative adversarial networks. However, generating new samples can be computationally expensive due to the need for multiple forward passes through the decoder network (Dhariwal and Nichol, 2021).

20.1. Forward Encoder

Suppose we take an image from the training set, which we will denote by $\mathbf{x}$, and blend it with Gaussian noise independently for each pixel to give a noise-corrupted image $\mathbf{z}_1$ defined by

$$\mathbf{z}_1 = \sqrt{1-\beta_1}\mathbf{x} + \sqrt{\beta_1}\boldsymbol{\epsilon}_1 \tag{20.1}$$

Exercise 20.1

where $\boldsymbol{\epsilon}_1 \sim \mathcal{N}(\boldsymbol{\epsilon}_1|\mathbf{0}, \mathbf{I})$ and $\beta_1 < 1$ is the variance of the noise distribution. The choice of coefficients $\sqrt{1-\beta_1}$ and $\sqrt{\beta_1}$ in (20.1) and (20.3) ensures that the mean of the distribution of $\mathbf{z}_t$ is closer to zero than the mean of $\mathbf{z}_{t-1}$ and that the variance of $\mathbf{z}_t$ is closer to the unit matrix than the variance of $\mathbf{z}_{t-1}$. We can write the transformation (20.1) in the form

Exercise 20.2

$$q(\mathbf{z}_1|\mathbf{x}) = \mathcal{N}(\mathbf{z}_1|\sqrt{1-\beta_1}\mathbf{x}, \beta_1\mathbf{I}). \tag{20.2}$$

We then repeat the process with additional independent Gaussian noise steps to give a sequence of increasingly noisy images $\mathbf{z}_2, \ldots, \mathbf{z}_T$. Note that in the literature on diffusion models, these latent variables are sometimes denoted $\mathbf{x}_1, \ldots, \mathbf{x}_T$ and the observed variable is denoted $\mathbf{x}_0$. We use the notation of $\mathbf{z}$ for latent variables and $\mathbf{x}$

Telegram: @uni_k

www.konkur.in

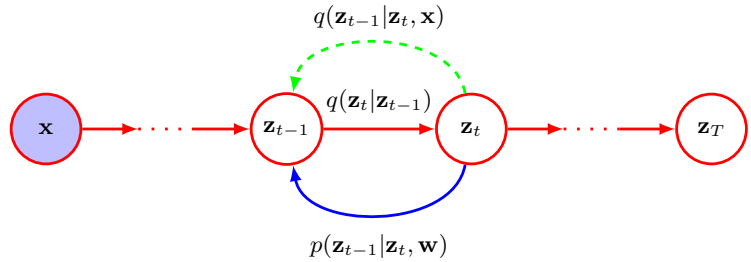


Figure 20.2 A diffusion process represented as a probabilistic graphical model. The original image $\mathbf{x}$ is shown by the shaded node, since it is an observed variable, whereas the noise-corrupted images $\mathbf{z}_1, \ldots, \mathbf{z}_T$ are considered to be latent variables. The noise process is defined by the forward distribution $q(\mathbf{z}_t|\mathbf{z}_{t-1})$ and can be viewed as an encoder. Our goal is to learn a model $p(\mathbf{z}_{t-1}|\mathbf{z}_t, \mathbf{w})$ that tries to reverse this noise process and which can be viewed as a decoder. As we will see later, the conditional distribution $q(\mathbf{z}_{t-1}|\mathbf{z}_t, \mathbf{x})$ plays an important role in defining the training procedure.

for the observed variable for consistency with the rest of the book. Each successive image is given by

$$\mathbf{z}_t = \sqrt{1-\beta_t}\mathbf{z}_{t-1} + \sqrt{\beta_t}\boldsymbol{\epsilon}_t \tag{20.3}$$

where $\boldsymbol{\epsilon}_t \sim \mathcal{N}(\boldsymbol{\epsilon}_t|\mathbf{0}, \mathbf{I})$. Again, we can write (20.3) in the form

$$q(\mathbf{z}_t|\mathbf{z}_{t-1}) = \mathcal{N}(\mathbf{z}_t|\sqrt{1-\beta_t}\mathbf{z}_{t-1}, \beta_t\mathbf{I}). \tag{20.4}$$

Section 11.3 The sequence of conditional distributions (20.4) forms a Markov chain and can be expressed as a probabilistic graphical model as shown in Figure 20.2. The values of the variance parameters $\beta_t \in (0, 1)$ are set by hand and are typically chosen such that the variance values increase through the chain according to a prescribed schedule such that $\beta_1 < \beta_2 < \ldots < \beta_T$.

20.1.1 Diffusion kernel

The joint distribution of the latent variables, conditioned on the observed data vector $\mathbf{x}$, is given by

$$q(\mathbf{z}_1, \ldots, \mathbf{z}_t|\mathbf{x}) = q(\mathbf{z}_1|\mathbf{x}) \prod_{\tau=2}^{t} q(\mathbf{z}_\tau|\mathbf{z}_{\tau-1}). \tag{20.5}$$

Exercise 20.3 If we now marginalize over the intermediate variables $\mathbf{z}_1, \ldots, \mathbf{z}_{t-1}$, we obtain the *diffusion kernel*:

$$q(\mathbf{z}_t|\mathbf{x}) = \mathcal{N}(\mathbf{z}_t|\sqrt{\alpha_t}\mathbf{x}, (1-\alpha_t)\mathbf{I}) \tag{20.6}$$

where we have defined

$$\alpha_t = \prod_{\tau=1}^{t} (1-\beta_\tau). \tag{20.7}$$

Telegram: @uni_k

www.konkur.in

We see that each intermediate distribution has a simple closed-form Gaussian expression from which we can directly sample, which will prove useful when training DDPMs as it allows efficient stochastic gradient descent using randomly chosen intermediate terms in the Markov chain without having to run the whole chain. We can also write (20.6) in the form

$$\mathbf{z}_t = \sqrt{\alpha_t}\mathbf{x} + \sqrt{1-\alpha_t}\boldsymbol{\epsilon}_t \tag{20.8}$$

where again $\boldsymbol{\epsilon}_t \sim \mathcal{N}(\boldsymbol{\epsilon}_t|\mathbf{0}, \mathbf{I})$. Note that that $\boldsymbol{\epsilon}$ now represents the total noise added to the original image instead of the incremental noise added at this step of the Markov chain.

After many steps the image becomes indistinguishable from Gaussian noise, and in the limit $T \to \infty$ we have

Exercise 20.4

$$q(\mathbf{z}_T|\mathbf{x}) = \mathcal{N}(\mathbf{z}_T|\mathbf{0}, \mathbf{I}) \tag{20.9}$$

and therefore all information about the original image is lost. The choice of coefficients $\sqrt{1-\beta_t}$ and $\sqrt{\beta_t}$ in (20.3) ensures that once the Markov chain converges to a distribution with zero mean and unit covariance, further updates will leave this unchanged.

Exercise 20.5

Since the right-hand side of (20.9) is independent of $\mathbf{x}$, it follows that the marginal distribution of $\mathbf{z}_T$ is given by

$$q(\mathbf{z}_T) = \mathcal{N}(\mathbf{z}_T|\mathbf{0}, \mathbf{I}). \tag{20.10}$$

It is common to refer to the Markov chain (20.4) as the *forward process*, and it is analogous to the encoder in a VAE, except that here it is fixed rather than learned. Note, however, that the usual terminology in the literature is the opposite of that typically used in the literature regarding normalizing flows, where the mapping from latent space to data space is considered the forward process.

20.1.2 Conditional distribution

Our goal is to learn to undo the noise process, and so it is natural to consider the reverse of the conditional distribution $q(\mathbf{z}_t|\mathbf{z}_{t-1})$, which we can express using Bayes' theorem in the form

$$q(\mathbf{z}_{t-1}|\mathbf{z}_t) = \frac{q(\mathbf{z}_t|\mathbf{z}_{t-1})q(\mathbf{z}_{t-1})}{q(\mathbf{z}_t)}. \tag{20.11}$$

We can write the marginal distribution $q(\mathbf{z}_{t-1})$ in the form

$$q(\mathbf{z}_{t-1}) = \int q(\mathbf{z}_{t-1}|\mathbf{x})p(\mathbf{x})\,\mathrm{d}\mathbf{x} \tag{20.12}$$

where $q(\mathbf{z}_{t-1}|\mathbf{x})$ is given by the conditional Gaussian (20.6). This distribution is intractable, however, because we must integrate over the unknown data density $p(\mathbf{x})$. If we approximate the integration using samples from the training data set, we obtain a complicated distribution expressed as a mixture of Gaussians.

Telegram: @uni_k

www.konkur.in

Instead, we consider the conditional version of the reverse distribution, conditioned on the data vector $\mathbf{x}$, defined by $q(\mathbf{z}_{t-1}|\mathbf{z}_t, \mathbf{x})$, which as we will see shortly turns out to be a simple Gaussian distribution. Intuitively this is reasonable since, given a noisy image, it is difficult to guess which lower-noise image gave rise to it, whereas if we also know the starting image then the problem becomes much easier. We can calculate this conditional distribution using Bayes' theorem:

$$q(\mathbf{z}_{t-1}|\mathbf{z}_t, \mathbf{x}) = \frac{q(\mathbf{z}_t|\mathbf{z}_{t-1}, \mathbf{x})q(\mathbf{z}_{t-1}|\mathbf{x})}{q(\mathbf{z}_t|\mathbf{x})}. \tag{20.13}$$

We now make use of the Markov property of the forward process to write

$$q(\mathbf{z}_t|\mathbf{z}_{t-1}, \mathbf{x}) = q(\mathbf{z}_t|\mathbf{z}_{t-1}) \tag{20.14}$$

where the right-hand side is given by (20.4). As a function of $\mathbf{z}_{t-1}$, this takes the form of an exponential of a quadratic form. The term $q(\mathbf{z}_{t-1}|\mathbf{x})$ in the numerator of (20.13) is the diffusion kernel given by (20.6), which again involves the exponential of a quadratic form with respect to $\mathbf{z}_{t-1}$. We can ignore the denominator in (20.13) since as a function of $\mathbf{z}_{t-1}$ it is constant. Thus, we see that the right-hand side of (20.13) takes the form of a Gaussian distribution, and we can identify its mean and covariance using the technique of 'completing the square' to give

Exercise 20.6

$$q(\mathbf{z}_{t-1}|\mathbf{z}_t, \mathbf{x}) = \mathcal{N}\left(\mathbf{z}_{t-1}|\mathbf{m}_t(\mathbf{x}, \mathbf{z}_t), \sigma_t^2\mathbf{I}\right) \tag{20.15}$$

where

$$\mathbf{m}_t(\mathbf{x}, \mathbf{z}_t) = \frac{(1-\alpha_{t-1})\sqrt{1-\beta_t}\mathbf{z}_t + \sqrt{\alpha_{t-1}}\beta_t\mathbf{x}}{1-\alpha_t} \tag{20.16}$$

$$\sigma_t^2 = \frac{\beta_t(1-\alpha_{t-1})}{1-\alpha_t} \tag{20.17}$$

and we have made use of (20.7).

20.2. Reverse Decoder

We have seen that the forward encoder model is defined by a sequence of Gaussian conditional distributions $q(\mathbf{z}_t|\mathbf{z}_{t-1})$ but that inverting this directly leads to a distribution $q(\mathbf{z}_{t-1}|\mathbf{z}_t)$ that is intractable, as it would require integrating over all possible values of the starting vector $\mathbf{x}$ whose distribution is the unknown data distribution $p(\mathbf{x})$ that we wish to model. Instead, we will learn an approximation to the reverse distribution by using a distribution $p(\mathbf{z}_{t-1}|\mathbf{z}_t, \mathbf{w})$ governed by a deep neural network, where $\mathbf{w}$ represents the network weights and biases. This reverse step is analogous to the decoder in a variational autoencoder and is illustrated in Figure 20.2. Once the network is trained, we can sample from the simple Gaussian distribution over $\mathbf{z}_T$ and transform it into a sample from the data distribution $p(\mathbf{x})$ through a sequence of reverse sampling steps by repeated application of the trained network.

Chapter 19

Telegram: @uni_k

www.konkur.in

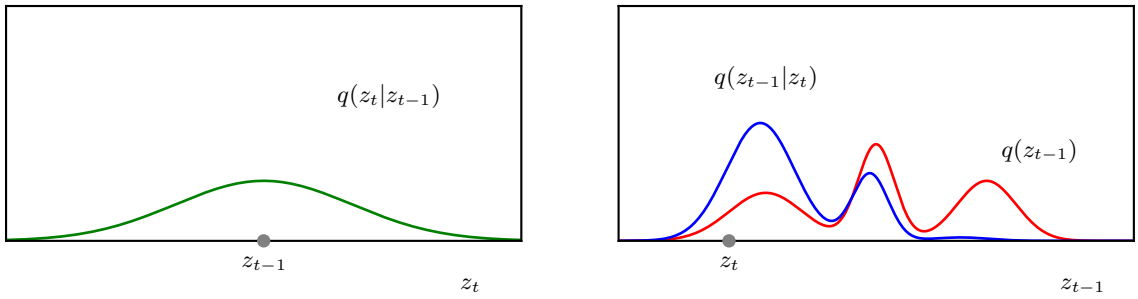


Figure 20.3 Illustration of the evaluation of the reverse distribution $q(z_{t-1}|z_t)$ using Bayes' theorem (20.13) for scalar variables. The red curve on the right-hand plot shows the marginal distribution $q(z_{t-1})$ illustrated using a mixture of three Gaussians, whereas the left-hand plot shows the Gaussian forward noise process $q(z_t|z_{t-1})$ as a distribution over z_t centred on z_{t-1}. By multiplying these together and normalizing, we obtain the distribution $q(z_{t-1}|z_t)$ shown for a particular choice of z_t by the blue curve. Because the distribution on the left is relatively broad, corresponding to a large variance β_t, the distribution $q(z_{t-1}|z_t)$ has a complex multimodal structure.

Intuitively, if we keep the variances small so that $\beta_t \ll 1$ then the change in the latent vector between steps will be relatively small, and hence it should be easier to learn to invert the transformation. More specifically, if $\beta_t \ll 1$ then the distribution $q(\mathbf{z}_{t-1}|\mathbf{z}_t)$ will be approximately a Gaussian distribution over $\mathbf{z}_{t-1}$. This can be seen from (20.11) since the right-hand side depends on $\mathbf{z}_{t-1}$ through $q(\mathbf{z}_t|\mathbf{z}_{t-1})$ and $q(\mathbf{z}_{t-1})$. If $q(\mathbf{z}_t|\mathbf{z}_{t-1})$ is a sufficiently narrow Gaussian then $q(\mathbf{z}_{t-1})$ will vary only a small amount over the region in which $q(\mathbf{z}_t|\mathbf{z}_{t-1})$ has significant mass, and hence $q(\mathbf{z}_{t-1}|\mathbf{z}_t)$ will also be approximately Gaussian. This intuition can be confirmed using a simple example as shown in Figures 20.3 and 20.4. However, since the variances at each step are small, we must use a large number of steps to ensure that the distribution over the final latent variable $\mathbf{z}_T$ obtained from the forward noising process will still be close to a Gaussian, and this increases the cost of generating new samples. In practice, T may be several thousand.

We can see more formally that $q(\mathbf{z}_{t-1}|\mathbf{z}_t)$ will be approximately Gaussian by

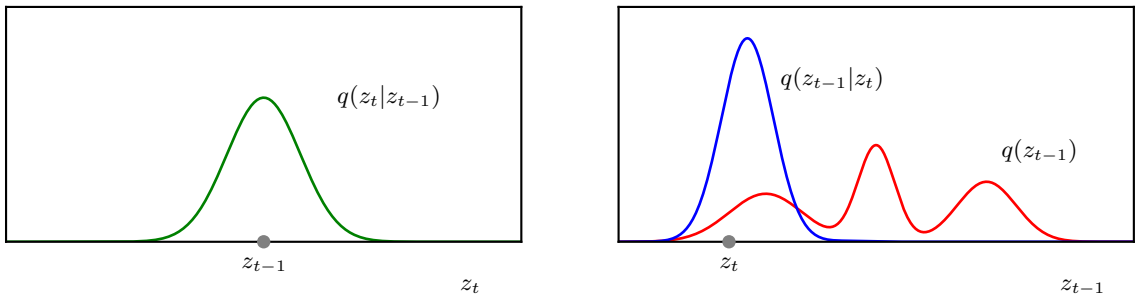


Figure 20.4 As in Figure 20.3 but in which the Gaussian distribution $q(z_t|z_{t-1})$ in the left-hand plot has a much smaller variance β_t. We see that the corresponding distribution $q(z_{t-1}|z_t)$ shown in blue on the right-hand plot is close to being Gaussian, with a similar variance to $q(z_t|z_{t-1})$.

Telegram: @uni_k

www.konkur.in

making a Taylor series expansion of $\ln q(\mathbf{z}_{t-1}|\mathbf{z}_t)$ around the point $\mathbf{z}_t$ as a function of $\mathbf{z}_{t-1}$. This also shows that for small variance, the reverse distribution $q(\mathbf{z}_t|\mathbf{z}_{t-1})$ will have a covariance that is close to the covariance $\beta_t\mathbf{I}$ of the forward noise process $q(\mathbf{z}_{t-1}|\mathbf{z}_t)$. We therefore model the reverse process using a Gaussian distribution of the form

Exercise 20.7

$$p(\mathbf{z}_{t-1}|\mathbf{z}_t, \mathbf{w}) = \mathcal{N}(\mathbf{z}_{t-1}|\boldsymbol{\mu}(\mathbf{z}_t, \mathbf{w}, t), \beta_t\mathbf{I}) \tag{20.18}$$

where $\boldsymbol{\mu}(\mathbf{z}_t, \mathbf{w}, t)$ is a deep neural network governed by a set of parameters $\mathbf{w}$. Note that the network takes the step index t explicitly as an input so that it can account for the variation of the variance β_t across different steps of the chain. This allows us to use a single network to invert all the steps in the Markov chain, instead of having to learn a separate network for each step. It is also possible to learn the covariances of the denoising process by incorporating further outputs in the network to account for the curvature in the distribution $q(\mathbf{z}_{t-1})$ in the neighbourhood of $\mathbf{z}_t$ (Nichol and Dhariwal, 2021). There considerable flexibility in the choice of architecture for the neural network used to model $\boldsymbol{\mu}(\mathbf{z}_t, \mathbf{w}, t)$ provided the output has the same dimensionality as the input. Given this restriction, a U-net architecture is a common choice for image processing applications.

Section 10.5.4

The overall reverse denoising process then takes the form of a Markov chain given by

$$p(\mathbf{x}, \mathbf{z}_1, \ldots, \mathbf{z}_T|\mathbf{w}) = p(\mathbf{z}_T)\left\{\prod_{t=2}^{T} p(\mathbf{z}_{t-1}|\mathbf{z}_t, \mathbf{w})\right\} p(\mathbf{x}|\mathbf{z}_1, \mathbf{w}). \tag{20.19}$$

Here $p(\mathbf{z}_T)$ is assumed to be the same as the distribution of $q(\mathbf{z}_T)$ and hence is given by $\mathcal{N}(\mathbf{z}_T|\mathbf{0}, \mathbf{I})$. Once the model has been trained, sampling is straightforward because we first sample from the simple Gaussian $p(\mathbf{z}_T)$ and then we sample sequentially from each of the conditional distributions $p(\mathbf{z}_{t-1}|\mathbf{z}_t, \mathbf{w})$ in turn, finally sampling from $p(\mathbf{x}|\mathbf{z}_1, \mathbf{w})$ to obtain a sample $\mathbf{x}$ in the data space.

20.2.1 Training the decoder

We next have to decide on an objective function for training the neural network. The obvious choice is the likelihood function, which for data point $\mathbf{x}$ is given by

$$p(\mathbf{x}|\mathbf{w}) = \int \cdots \int p(\mathbf{x}, \mathbf{z}_1, \ldots, \mathbf{z}_T|\mathbf{w})\, \mathrm{d}\mathbf{z}_1 \ldots \mathrm{d}\mathbf{z}_T \tag{20.20}$$

in which $p(\mathbf{x}, \mathbf{z}_1, \ldots, \mathbf{z}_T|\mathbf{w})$ is defined by (20.19). This is an instance of the general latent-variable model (16.81) in which the latent variables comprise $\mathbf{z} = (\mathbf{z}_1, \ldots, \mathbf{z}_T)$ and the observed variable is $\mathbf{x}$. Note that the latent variables all have the same dimensionality as the data space, as was the case for normalizing flows but not for variational autoencoders or generative adversarial networks. We see from (20.20) that the likelihood involves integrating over all possible trajectories by which noise samples could give rise to the observed data point. The integrals in (20.20) are intractable as they involve integrating over the highly complex neural network functions.

Telegram: @uni_k

www.konkur.in

20.2.2 Evidence lower bound

Since the exact likelihood is intractable, we can adopt a similar approach to that used with variational autoencoders and maximize a lower bound on the log likelihood called the *evidence lower bound* (ELBO), which we re-derive here in the context of diffusion models. For any choice of distribution $q(\mathbf{z})$, the following relation always holds:

Section 16.3

$$\ln p(\mathbf{x}|\mathbf{w}) = \mathcal{L}(\mathbf{w}) + \mathrm{KL}\left(q(\mathbf{z})\|p(\mathbf{z}|\mathbf{x},\mathbf{w})\right) \tag{20.21}$$

where $\mathcal{L}$ is the evidence lower bound, also known as the *variational lower bound*, given by

$$\mathcal{L}(\mathbf{w}) = \int q(\mathbf{z}) \ln \left\{ \frac{p(\mathbf{x},\mathbf{z}|\mathbf{w})}{q(\mathbf{z})} \right\} \mathrm{d}\mathbf{z} \tag{20.22}$$

and the Kullback–Leibler divergence $\mathrm{KL}\left(f\|g\right)$ between two probability densities $f(\mathbf{z})$ and $g(\mathbf{z})$ is defined by

Section 2.5.7

$$\mathrm{KL}\left(f(\mathbf{z})\|g(\mathbf{z})\right) = -\int f(\mathbf{z}) \ln \left\{ \frac{g(\mathbf{z})}{f(\mathbf{z})} \right\} \mathrm{d}\mathbf{z}. \tag{20.23}$$

To verify the relation (20.21) first note that, from the product rule of probability, we have

$$p(\mathbf{x},\mathbf{z}|\mathbf{w}) = p(\mathbf{z}|\mathbf{x},\mathbf{w})p(\mathbf{x}|\mathbf{w}). \tag{20.24}$$

Exercise 20.8
Section 2.5.7

Substituting (20.24) into (20.22) and making use of (20.23) gives (20.21). The Kullback–Leibler divergence has the property $\mathrm{KL}\left(\cdot\|\cdot\right) \geqslant 0$ from which it follows that

$$\ln p(\mathbf{x}|\mathbf{w}) \geqslant \mathcal{L}(\mathbf{w}). \tag{20.25}$$

Since the log likelihood function is intractable, we train the neural network by maximizing the lower bound $\mathcal{L}(\mathbf{w})$.

To do this, we first derive an explicit form for the lower bound of the diffusion model. In defining the lower bound we are free to choose any form we like for $q(\mathbf{z})$ as long as it is a valid probability distribution, i.e., that it is non-negative and integrates to 1. With many applications of the ELBO, such as the variational autoencoder, we chose a form for $q(\mathbf{z})$ that has adjustable parameters, often in the form of a deep neural network, and then we maximize the ELBO with respect to those parameters as well as with respect to the parameters of the distribution $p(\mathbf{x},\mathbf{z}|\mathbf{w})$. Optimizing the distribution $q(\mathbf{z})$ encourages the bound to be tight, which brings the optimization of the parameters in $p(\mathbf{x},\mathbf{z}|\mathbf{w})$ closer to that of maximum likelihood. With diffusion models, however, we chose $q(\mathbf{z})$ to be given by the *fixed* distribution $q(\mathbf{z}_1,\ldots,\mathbf{z}_T|\mathbf{x})$ defined by the Markov chain (20.5), and so the only adjustable parameters are those in the model $p(\mathbf{x},\mathbf{z}_1,\ldots,\mathbf{z}_T|\mathbf{w})$ for the reverse Markov chain. Note that we are using the flexibility in the choice of $q(\mathbf{z})$ to select a form that depends on $\mathbf{x}$.

We therefore substitute for $q(\mathbf{z}_1,\ldots,\mathbf{z}_T|\mathbf{x})$ in (20.21) using (20.5), and likewise we substitute for $p(\mathbf{x},\mathbf{z}_1,\ldots,\mathbf{z}_T|\mathbf{w})$ using (20.19), which allows us to write the

Telegram: @uni_k

www.konkur.in

ELBO in the form

$$\mathcal{L}(\mathbf{w}) = \mathbb{E}_q\left[\ln \frac{p(\mathbf{z}_T)\left\{\prod_{t=2}^{T} p(\mathbf{z}_{t-1}|\mathbf{z}_t,\mathbf{w})\right\} p(\mathbf{x}|\mathbf{z}_1,\mathbf{w})}{q(\mathbf{z}_1|\mathbf{x})\prod_{t=2}^{T} q(\mathbf{z}_t|\mathbf{z}_{t-1},\mathbf{x})}\right]$$

$$= \mathbb{E}_q\left[\ln p(\mathbf{z}_T) + \sum_{t=2}^{T} \ln \frac{p(\mathbf{z}_{t-1}|\mathbf{z}_t,\mathbf{w})}{q(\mathbf{z}_t|\mathbf{z}_{t-1},\mathbf{x})} - \ln q(\mathbf{z}_1|\mathbf{x}) + \ln p(\mathbf{x}|\mathbf{z}_1,\mathbf{w})\right] \tag{20.26}$$

where we have defined

$$\mathbb{E}_q\left[\,\cdot\,\right] \equiv \int \cdots \int q(\mathbf{z}_1|\mathbf{x}) \prod_{t=2}^{T} q(\mathbf{z}_t|\mathbf{z}_{t-1})\left[\,\cdot\,\right] \mathrm{d}\mathbf{z}_1 \ldots \mathrm{d}\mathbf{z}_T. \tag{20.27}$$

The first term $\ln p(\mathbf{z}_T)$ on the right-hand side of (20.26) is just the fixed distribution $\mathcal{N}(\mathbf{z}_T|\mathbf{0},\mathbf{I})$. This has no trainable parameters and can therefore be omitted from the ELBO since it represents a fixed additive constant. Similarly, the third term $-\ln q(\mathbf{z}_1|\mathbf{x})$ is independent of $\mathbf{w}$ and so again can be omitted.

The fourth term on the right-hand side of (20.26) corresponds to the reconstruction term from the variational autoencoder. It can be evaluated by approximating the expectation $\mathbb{E}_q\left[\,\cdot\,\right]$ by a Monte Carlo estimate obtained by drawing samples from the distribution over $\mathbf{z}_1$ defined by (20.2) so that

$$\mathbb{E}_q\left[\ln p(\mathbf{x}|\mathbf{z}_1,\mathbf{w})\right] \simeq \sum_{l=1}^{L} \ln p(\mathbf{x}|\mathbf{z}_1^{(l)},\mathbf{w}) \tag{20.28}$$

where $\mathbf{z}_1^{(l)} \sim \mathcal{N}(\mathbf{z}_1|\sqrt{1-\beta_1}\mathbf{x},\beta_1\mathbf{I})$. Unlike with VAEs we do not need to back-propagate an error signal through the sampled value because the q-distribution is fixed and so there is no need here for the reparameterization trick.

Section 19.2.2

This leaves the second term on the right-hand side of (20.26), which comprises a sum of terms each of which is dependent on a pair of adjacent latent-variable values $\mathbf{z}_{t-1}$ and $\mathbf{z}_t$. We saw earlier when we derived the diffusion kernel (20.6) that we can sample from $q(\mathbf{z}_{t-1}|\mathbf{x})$ directly as a Gaussian distribution and we could then obtain a corresponding sample of $\mathbf{z}_t$ using (20.4), which is also a Gaussian. Although this would be a correct procedure in the limit of an infinite number of samples, the use of pairs of sampled values creates very noisy estimates with high variance, so that an unnecessarily large numbers of samples is required. Instead, we rewrite the ELBO in a form that can be estimated by sampling just one value per term.

20.2.3 Rewriting the ELBO

Following our discussion of the ELBO for the variational autoencoder, our goal here is to write the ELBO in terms of Kullback–Leibler divergences, which we can then subsequently express in closed form. The neural network is a model of the distribution in the reverse direction $p(\mathbf{z}_{t-1}|\mathbf{z}_t,\mathbf{w})$ whereas the q-distribution is expressed in the forward direction $q(\mathbf{z}_t|\mathbf{z}_{t-1},\mathbf{x})$, and so we use Bayes' theorem to

Telegram: @uni_k

www.konkur.in

reverse the conditional distribution by writing

$$q(\mathbf{z}_t|\mathbf{z}_{t-1},\mathbf{x}) = \frac{q(\mathbf{z}_{t-1}|\mathbf{z}_t,\mathbf{x})q(\mathbf{z}_t|\mathbf{x})}{q(\mathbf{z}_{t-1}|\mathbf{x})}. \tag{20.29}$$

This allows us to write the second term in (20.26) in the form

$$\ln \frac{p(\mathbf{z}_{t-1}|\mathbf{z}_t,\mathbf{w})}{q(\mathbf{z}_t|\mathbf{z}_{t-1},\mathbf{x})} = \ln \frac{p(\mathbf{z}_{t-1}|\mathbf{z}_t,\mathbf{w})}{q(\mathbf{z}_{t-1}|\mathbf{z}_t,\mathbf{x})} + \ln \frac{q(\mathbf{z}_{t-1}|\mathbf{x})}{q(\mathbf{z}_t|\mathbf{x})}. \tag{20.30}$$

The second term on the right-hand side of (20.30) is independent of $\mathbf{w}$ and so can be omitted. Substituting (20.30) into (20.26), we then obtain

$$\mathcal{L}(\mathbf{w}) = \mathbb{E}_q\left[\sum_{t=2}^{T} \ln \frac{p(\mathbf{z}_{t-1}|\mathbf{z}_t,\mathbf{w})}{q(\mathbf{z}_{t-1}|\mathbf{z}_t,\mathbf{x})} + \ln p(\mathbf{x}|\mathbf{z}_1,\mathbf{w})\right]. \tag{20.31}$$

Exercise 20.9

Finally, we can rewrite (20.31) in the form

$$\mathcal{L}(\mathbf{w}) = \underbrace{\int q(\mathbf{z}_1|\mathbf{x}) \ln p(\mathbf{x}|\mathbf{z}_1,\mathbf{w})\,\mathrm{d}\mathbf{z}_1}_{\text{reconstruction term}} - \underbrace{\sum_{t=2}^{T} \int \mathrm{KL}(q(\mathbf{z}_{t-1}|\mathbf{z}_t,\mathbf{x})\|p(\mathbf{z}_{t-1}|\mathbf{z}_t,\mathbf{w}))q(\mathbf{z}_t|\mathbf{x})\,\mathrm{d}\mathbf{z}_t}_{\text{consistency terms}} \tag{20.32}$$

where we have simplified the expectation over $q(\mathbf{z}_1,\ldots,\mathbf{z}_T|\mathbf{x})$ in the first term since $\mathbf{z}_1$ is the only latent variable appearing in the integrand. Therefore in the expectation defined by (20.27), all the conditional distributions integrate to unity leaving only the integral over $\mathbf{z}_1$. Likewise, in the second term, each integral involves only two adjacent latent variables $\mathbf{z}_{t-1}$ and $\mathbf{z}_t$, and all remaining variables can be integrated out.

The bound (20.32) is now very similar to the ELBO for the variational autoencoder given by (19.14), except that there are now multiple encoder and decoder stages. The reconstruction term rewards high probability for the observed data sample and can be trained in the same way as the corresponding term in the VAE by using the sampling approximation (20.28). The consistency terms in (20.32) are defined between pairs of Gaussian distributions and therefore can be expressed in closed form, as follows. The distribution $q(\mathbf{z}_{t-1}|\mathbf{z}_t,\mathbf{x})$ is given by (20.15) whereas the distribution $p(\mathbf{z}_{t-1}|\mathbf{z}_t,\mathbf{w})$ is given by (20.18) and so the Kullback–Leibler divergence becomes

Chapter 19

Exercise 20.11

$$\begin{aligned}&\mathrm{KL}(q(\mathbf{z}_{t-1}|\mathbf{z}_t,\mathbf{x})\|p(\mathbf{z}_{t-1}|\mathbf{z}_t,\mathbf{w}))\\ &\quad = \frac{1}{2\beta_t}\left\|\mathbf{m}_t(\mathbf{x},\mathbf{z}_t) - \boldsymbol{\mu}(\mathbf{z}_t,\mathbf{w},t)\right\|^2 + \text{const}\end{aligned} \tag{20.33}$$

Telegram: @uni_k

www.konkur.in

where $\mathbf{m}_t(\mathbf{x}, \mathbf{z}_t)$ is defined by (20.16) and where any additive terms that are independent of the network parameters $\mathbf{w}$ have been absorbed into the constant term, which plays no role in training. Each of the consistency terms in (20.32) has one remaining integral over $\mathbf{z}_t$, weighted by $q(\mathbf{z}_t|\mathbf{x})$. This can be approximated by drawing a sample from $q(\mathbf{z}_t|\mathbf{x})$, which can be done efficiently using the diffusion kernel (20.6).

We see that the KL divergence (20.33) takes the form of a simple squared-loss function. Since we adjust the network parameters to maximize the lower bound in (20.32), we will be minimizing this squared error because there is a minus sign in front of the Kullback–Leibler divergence terms in the ELBO.

20.2.4 Predicting the noise

One modification that leads to higher quality results is to change the role of the neural network so that instead of predicting the denoised image at each step of the Markov chain it predicts the total noise component that was added to the original image to create the noisy image at that step (Ho, Jain, and Abbeel, 2020). To do this we first take (20.8) and rearrange to give

$$\mathbf{x} = \frac{1}{\sqrt{\alpha_t}}\mathbf{z}_t - \frac{\sqrt{1-\alpha_t}}{\sqrt{\alpha_t}}\boldsymbol{\epsilon}_t. \tag{20.34}$$

If we now substitute this into (20.16) we can rewrite the mean $\mathbf{m}_t(\mathbf{x}, \mathbf{z}_t)$ of the reverse conditional distribution $q(\mathbf{z}_{t-1}|\mathbf{z}_t, \mathbf{x})$ in terms of the original data vector $\mathbf{x}$ and the noise $\boldsymbol{\epsilon}$ to give

Exercise 20.12

$$\mathbf{m}_t(\mathbf{x}, \mathbf{z}_t) = \frac{1}{\sqrt{1-\beta_t}}\left\{\mathbf{z}_t - \frac{\beta_t}{\sqrt{1-\alpha_t}}\boldsymbol{\epsilon}_t\right\}. \tag{20.35}$$

Similarly, instead of a neural network $\boldsymbol{\mu}(\mathbf{z}_t, \mathbf{w}, t)$ that predicts the denoised image, we introduce a neural network $\mathbf{g}(\mathbf{z}_t, \mathbf{w}, t)$ that aims to predict the total noise that was added to $\mathbf{x}$ to generate $\mathbf{z}_t$. Following the same steps that led to (20.35) shows that these two network functions are related by

$$\boldsymbol{\mu}(\mathbf{z}_t, \mathbf{w}, t) = \frac{1}{\sqrt{1-\beta_t}}\left\{\mathbf{z}_t - \frac{\beta_t}{\sqrt{1-\alpha_t}}\mathbf{g}(\mathbf{z}_t, \mathbf{w}, t)\right\}. \tag{20.36}$$

We can now substitute (20.35) and (20.36) into (20.33) to give

$$\begin{aligned}
&\mathrm{KL}(q(\mathbf{z}_{t-1}|\mathbf{z}_t, \mathbf{x})\|p(\mathbf{z}_{t-1}|\mathbf{z}_t, \mathbf{w})) \\
&= \frac{\beta_t}{2(1-\alpha_t)(1-\beta_t)}\left\|\mathbf{g}(\mathbf{z}_t, \mathbf{w}, t) - \boldsymbol{\epsilon}_t\right\|^2 + \text{const} \\
&= \frac{\beta_t}{2(1-\alpha_t)(1-\beta_t)}\left\|\mathbf{g}(\sqrt{\alpha_t}\mathbf{x} + \sqrt{1-\alpha_t}\boldsymbol{\epsilon}_t, \mathbf{w}, t) - \boldsymbol{\epsilon}_t\right\|^2 + \text{const}
\end{aligned} \tag{20.37}$$

where in the final line we have substituted for $\mathbf{z}_t$ using (20.8).

Telegram: @uni_k

www.konkur.in

The reconstruction term in the ELBO (20.32) can be approximated using (20.28) with a sampled value of $\mathbf{z}_1$. Using the form (20.18) for $p(\mathbf{x}|\mathbf{z}, \mathbf{w})$ we have

$$\ln p(\mathbf{x}|\mathbf{z}_1, \mathbf{w}) = -\frac{1}{2\beta_1}\|\mathbf{x} - \boldsymbol{\mu}(\mathbf{z}_1, \mathbf{w}, 1)\|^2 + \text{const.} \tag{20.38}$$

If we substitute for $\boldsymbol{\mu}(\mathbf{z}_1, \mathbf{w}, 1)$ using (20.36) and we substitute for $\mathbf{x}$ using (20.1) and then make use of $\alpha_1 = (1 - \beta_1)$, which follows from (20.7), we obtain

Exercise 20.13

$$\ln p(\mathbf{x}|\mathbf{z}_1, \mathbf{w}) = -\frac{1}{2(1-\beta_t)}\|\mathbf{g}(\mathbf{z}_1, \mathbf{w}, 1) - \boldsymbol{\epsilon}_1\|^2 + \text{const.} \tag{20.39}$$

This is precisely the same form as (20.37) for the special case $t = 1$, and so the reconstruction and consistency terms can be combined.

Ho, Jain, and Abbeel (2020) found empirically that performance is further improved simply by omitting the factor $\beta_t/2(1 - \alpha_t)(1 - \beta_t)$ in front of (20.37), so that all steps in the Markov chain have equal weighting. Substituting this simplified version of (20.37) into (20.33) gives a training objective function in the form

$$\mathcal{L}(\mathbf{w}) = -\sum_{t=1}^{T} \left\|\mathbf{g}(\sqrt{\alpha_t}\mathbf{x} + \sqrt{1-\alpha_t}\boldsymbol{\epsilon}_t, \mathbf{w}, t) - \boldsymbol{\epsilon}_t\right\|^2. \tag{20.40}$$

The squared error on the right-hand side of (20.40) has a very simple interpretation: for a given step t in the Markov chain and for a given training data point $\mathbf{x}$, we sample a noise vector $\boldsymbol{\epsilon}_t$ and use this to create the corresponding noisy latent vector $\mathbf{z}_t$ for that step. The loss function is then the squared difference between the predicted noise and the actual noise. Note that the network $\mathbf{g}(\cdot, \cdot, \cdot)$ is predicting the total noise added to the original data vector $\mathbf{x}$, not just the incremental noise added in step t.

When we use stochastic gradient descent, we evaluate the gradient vector of the loss function with respect to the network parameters for a randomly selected data point $\mathbf{x}$ from the training set. Also, for each such data point we randomly select a step t along the Markov chain, rather than evaluate the error for every term in the summation over t in (20.40). These gradients are accumulated over mini-batches of data samples and then used to update the weights.

Also note that this loss function automatically builds in a form of data augmentation, because every time a particular training sample $\mathbf{x}$ is used it is combined with a fresh sample $\boldsymbol{\epsilon}_t$ of noise. All the above relates to a single data point $\mathbf{x}$ from the training set. The corresponding computation of the gradient is shown in Algorithm 20.1.

20.2.5 Generating new samples

Once the network has been trained we can generate new samples in the data space by first sampling from the Gaussian distribution $p(\mathbf{z}_T)$ and then denoising successively through each step of the Markov chain. Given a denoised sample $\mathbf{z}_t$ at step t, we generate a sample $\mathbf{z}_{t-1}$ in three steps. First we evaluate the output of the neural network given by $\mathbf{g}(\mathbf{z}_t, \mathbf{w}, t)$. From this we evaluate $\boldsymbol{\mu}(\mathbf{z}_t, \mathbf{w}, t)$ using (20.36).

Telegram: @uni_k

www.konkur.in

Algorithm 20.1: Training a denoising diffusion probabilistic model

Input: Training data $\mathcal{D} = \{\mathbf{x}_n\}$
Noise schedule $\{\beta_1, \ldots, \beta_T\}$
Output: Network parameters $\mathbf{w}$

---

for $t \in \{1, \ldots, T\}$ **do**
  $\alpha_t \leftarrow \prod_{\tau=1}^{t}(1-\beta_\tau)$ `// Calculate alphas from betas`
end for
repeat
  $\mathbf{x} \sim \mathcal{D}$ `// Sample a data point`
  $t \sim \{1, \ldots, T\}$ `// Sample a point along the Markov chain`
  $\boldsymbol{\epsilon} \sim \mathcal{N}(\boldsymbol{\epsilon}|\mathbf{0}, \mathbf{I})$ `// Sample a noise vector`
  $\mathbf{z}_t \leftarrow \sqrt{\alpha_t}\mathbf{x} + \sqrt{1-\alpha_t}\boldsymbol{\epsilon}$ `// Evaluate noisy latent variable`
  $\mathcal{L}(\mathbf{w}) \leftarrow \|\mathbf{g}(\mathbf{z}_t, \mathbf{w}, t) - \boldsymbol{\epsilon}\|^2$ `// Compute loss term`
  Take optimizer step
until converged
return $\mathbf{w}$

Finally we generate a sample $\mathbf{z}_{t-1}$ from $p(\mathbf{z}_{t-1}|\mathbf{z}_t, \mathbf{w}) = \mathcal{N}(\mathbf{z}_{t-1}|\boldsymbol{\mu}(\mathbf{z}_t, \mathbf{w}, t), \beta_t\mathbf{I})$ by adding noise scaled by the variance so that

$$\mathbf{z}_{t-1} = \boldsymbol{\mu}(\mathbf{z}_t, \mathbf{w}, t) + \sqrt{\beta_t}\boldsymbol{\epsilon} \tag{20.41}$$

where $\boldsymbol{\epsilon} \sim \mathcal{N}(\boldsymbol{\epsilon}|\mathbf{0}, \mathbf{I})$. Note that the network $\mathbf{g}(\cdot, \cdot, \cdot)$ predicts the total noise added to the original data vector $\mathbf{x}$ to obtain $\mathbf{z}_t$, but in the sampling step, we subtract off only a fraction $\beta_t/\sqrt{1-\alpha_t}$ of this noise from $\mathbf{z}_{t-1}$ and then add additional noise with variance β_t to generate $\mathbf{z}_{t-1}$. At the final step when we calculate a synthetic data sample $\mathbf{x}$, we do not add additional noise since we are aiming to generate a noise-free output. The sampling procedure is summarized in Algorithm 20.2.

The main drawback of diffusion models for generating data is that they require multiple sequential inference passes through the trained network, which can be computationally expensive. One way to speed up the sampling process is first to convert the denoising process to a differential equation over continuous time and then to use alternative efficient discretization methods to solve the equation efficiently.

Section 20.3.4

We have assumed in this chapter that the data and latent variables are continuous and that we can therefore use Gaussian noise models. Diffusion models can also be defined for discrete spaces (Austin *et al.*, 2021), for example, to generate new candidate drug molecules in which part of the generation process involves choosing atom types from a subset of chemical elements.

We have seen that diffusion models can be computationally intensive because

Telegram: @uni_k

www.konkur.in

Algorithm 20.2: Sampling from a denoising diffusion probabilistic model

Input: Trained denoising network $\mathbf{g}(\mathbf{z}, \mathbf{w}, t)$
Noise schedule $\{\beta_1, \ldots, \beta_T\}$
Output: Sample vector $\mathbf{x}$ in data space

$\mathbf{z}_T \sim \mathcal{N}(\mathbf{z}|\mathbf{0}, \mathbf{I})$ // Sample from final latent space
for $t \in T, \ldots, 2$ **do**
  $\alpha_t \leftarrow \prod_{\tau=1}^{t}(1-\beta_\tau)$ // Calculate alpha
  // Evaluate network output
  $\boldsymbol{\mu}(\mathbf{z}_t, \mathbf{w}, t) \leftarrow \frac{1}{\sqrt{1-\beta_t}}\left\{\mathbf{z}_t - \frac{\beta_t}{\sqrt{1-\alpha_t}}\mathbf{g}(\mathbf{z}_t, \mathbf{w}, t)\right\}$
  $\boldsymbol{\epsilon} \sim \mathcal{N}(\boldsymbol{\epsilon}|\mathbf{0}, \mathbf{I})$ // Sample a noise vector
  $\mathbf{z}_{t-1} \leftarrow \boldsymbol{\mu}(\mathbf{z}_t, \mathbf{w}, t) + \sqrt{\beta_t}\boldsymbol{\epsilon}$ // Add scaled noise
end for
$\mathbf{x} = \frac{1}{\sqrt{1-\beta_1}}\left\{\mathbf{z}_1 - \frac{\beta_1}{\sqrt{1-\alpha_1}}\mathbf{g}(\mathbf{z}_1, \mathbf{w}, t)\right\}$ // Final denoising step
return $\mathbf{x}$

they sequentially reverse a noise process that can have hundreds or thousands of steps. Song, Meng, and Ermon (2020) introduced a related technique called *denoising diffusion implicit models* that relax the Markovian assumption on the noise process while retaining the same objective function for training. This thereby allows one or two orders of magnitude speed-up during sampling without degrading the quality of the generated samples.

20.3. Score Matching

The denoising diffusion models discussed so far in this chapter are closely related to another class of deep generative models that were developed relatively independently and which are based on *score matching* (Hyvärinen, 2005; Song and Ermon, 2019). These make use of the *score function* or *Stein score*, which is defined as the gradient of the log likelihood with respect to the data vector $\mathbf{x}$ and is given by

$$\mathbf{s}(\mathbf{x}) = \nabla_{\mathbf{x}} \ln p(\mathbf{x}). \tag{20.42}$$

Here it is important to emphasize that the gradient is with respect to the data vector, not with respect to any parameter vector. Note that $\mathbf{s}(\mathbf{x})$ is a vector-valued function of the same dimensionality as $\mathbf{x}$ and that each element $s_i(\mathbf{x}) = \partial \ln p(\mathbf{x})/\partial x_i$ is associated with a corresponding element x_i of $\mathbf{x}$. For example, if $\mathbf{x}$ is an image then $\mathbf{s}(\mathbf{x})$ can also be represented as an image of the same dimensions with corresponding

Telegram: @uni_k

www.konkur.in

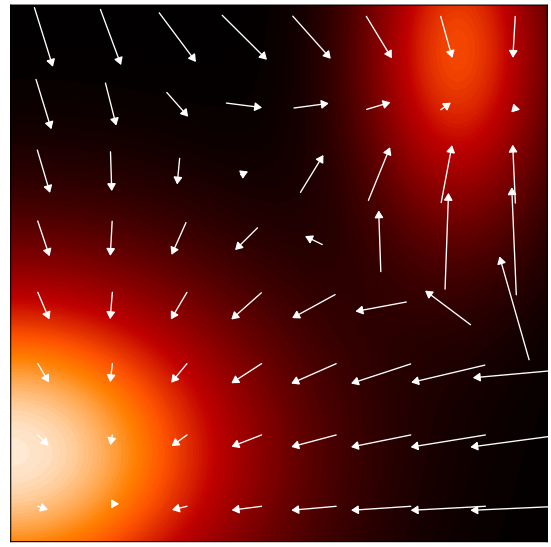


Figure 20.5 Illustration of the score function, showing a distribution in two dimensions comprising a mixture of Gaussians represented as a heat map and the corresponding score function defined by (20.42) plotted as vectors on a regular grid of $\mathbf{x}$-values.

pixels. Figure 20.5 shows an example of a probability density in two dimensions, along with the corresponding score function.

To see why the score function is useful, consider two functions $q(\mathbf{x})$ and $p(\mathbf{x})$ that have the property that their scores are equal, so that $\nabla_{\mathbf{x}} \ln q(\mathbf{x}) = \nabla_{\mathbf{x}} \ln p(\mathbf{x})$ for all values of $\mathbf{x}$. If we integrate both sides of the equation with respect to $\mathbf{x}$ and take exponentials, we obtain $q(\mathbf{x}) = Kp(\mathbf{x})$ where K is a constant independent of $\mathbf{x}$. So if we are able to learn a model $\mathbf{s}(\mathbf{x}, \mathbf{w})$ of the score function then we have modelled the original data density, up to a multiplicative constant.

20.3.1 Score loss function

To train such a model we need to define a loss function that aims to match the model score function $\mathbf{s}(\mathbf{x}, \mathbf{w})$ to the score function $\nabla_{\mathbf{x}} \ln p(\mathbf{x})$ of the distribution $p(\mathbf{x})$ that generated the data. An example of such a loss function is the expected squared error between the model score and the true score, given by

$$J(\mathbf{w}) = \frac{1}{2} \int \|\mathbf{s}(\mathbf{x}, \mathbf{w}) - \nabla_{\mathbf{x}} \ln p(\mathbf{x})\|^2 \, p(\mathbf{x}) \, \mathrm{d}\mathbf{x}. \tag{20.43}$$

Section 14.3.1

As we saw in the discussion of energy-based models, the score function does not require the associated probability density to be normalized, because the normalization constant is removed by the gradient operator, and so there is considerable flexibility in the choice of model. There are broadly two ways to represent the score function $\mathbf{s}(\mathbf{x}, \mathbf{w})$ using a deep neural network. Each element s_i of $\mathbf{s}$ corresponds to one of the elements x_i of $\mathbf{x}$, so the first approach is to have a network with the same number of outputs as inputs. However, the score function is defined to be the gradient of a scalar function (the log probability density), which is a more restricted class of functions. So an alternative approach is to have a network with a single output $\phi(\mathbf{x})$

Exercise 20.14

Telegram: @uni_k

www.konkur.in

and then to compute $\nabla_{\mathbf{x}}\phi(\mathbf{x})$ using automatic differentiation. This second approach, however, requires two backpropagation steps and is therefore computationally more expensive. For this reason, most applications simply adopt the first approach.

Exercise 20.15

20.3.2 Modified score loss

One problem with the loss function (20.43) is that we cannot minimize it directly because we do not know the true data score $\nabla_{\mathbf{x}} \ln p(\mathbf{x})$. All we have is the finite data set $\mathcal{D} = (\mathbf{x}_1, \ldots, \mathbf{x}_N)$ from which we can construct an empirical distribution:

$$p_{\mathcal{D}}(\mathbf{x}) = \frac{1}{N}\sum_{n=1}^{N} \delta(\mathbf{x} - \mathbf{x}_n). \tag{20.44}$$

Here $\delta(\mathbf{x})$ is the Dirac delta function, which can be thought of informally as an infinitely tall 'spike' at $\mathbf{x} = \mathbf{0}$ with the properties

$$\delta(\mathbf{x}) = 0, \quad \mathbf{x} \neq \mathbf{0} \tag{20.45}$$

$$\int \delta(\mathbf{x})\, \mathrm{d}\mathbf{x} = 1. \tag{20.46}$$

Since (20.44) is not a differentiable function of $\mathbf{x}$, we cannot compute its score function. We can address this by introducing a noise model to 'smear out' the data points and give a smooth, differentiable representation of the density. This is known as a *Parzen estimator* or *kernel density estimator* and is defined by

Section 3.5.2

$$q_\sigma(\mathbf{z}) = \int q(\mathbf{z}|\mathbf{x}, \sigma) p(\mathbf{x})\, \mathrm{d}\mathbf{x} \tag{20.47}$$

where $q(\mathbf{z}|\mathbf{x}, \sigma)$ is the *noise kernel*. A common choice of kernel is the Gaussian

$$q(\mathbf{z}|\mathbf{x}, \sigma) = \mathcal{N}(\mathbf{z}|\mathbf{x}, \sigma^2\mathbf{I}). \tag{20.48}$$

Instead of minimizing the loss function (20.43), we then use the corresponding loss with respect to the smoothed Parzen density in the form

$$J(\mathbf{w}) = \frac{1}{2}\int \left\|\mathbf{s}(\mathbf{z}, \mathbf{w}) - \nabla_{\mathbf{z}} \ln q_\sigma(\mathbf{z})\right\|^2 q_\sigma(\mathbf{z})\, \mathrm{d}\mathbf{z}. \tag{20.49}$$

A key result is that by substituting (20.47) into (20.49) we can rewrite this loss function in an equivalent form given by (Vincent, 2011)

Exercise 20.17

$$J(\mathbf{w}) = \frac{1}{2}\iint \left\|\mathbf{s}(\mathbf{z}, \mathbf{w}) - \nabla_{\mathbf{z}} \ln q(\mathbf{z}|\mathbf{x}, \sigma)\right\|^2 q(\mathbf{z}|\mathbf{x}, \sigma) p(\mathbf{x})\, \mathrm{d}\mathbf{z}\, \mathrm{d}\mathbf{x} + \text{const}. \tag{20.50}$$

If we substitute for $p(\mathbf{x})$ using the empirical density (20.44), we obtain

$$J(\mathbf{w}) = \frac{1}{2N}\sum_{n=1}^{N}\int \left\|\mathbf{s}(\mathbf{z}, \mathbf{w}) - \nabla_{\mathbf{z}} \ln q(\mathbf{z}|\mathbf{x}_n, \sigma)\right\|^2 q(\mathbf{z}|\mathbf{x}_n, \sigma)\, \mathrm{d}\mathbf{z} + \text{const}. \tag{20.51}$$

Telegram: @uni_k

www.konkur.in

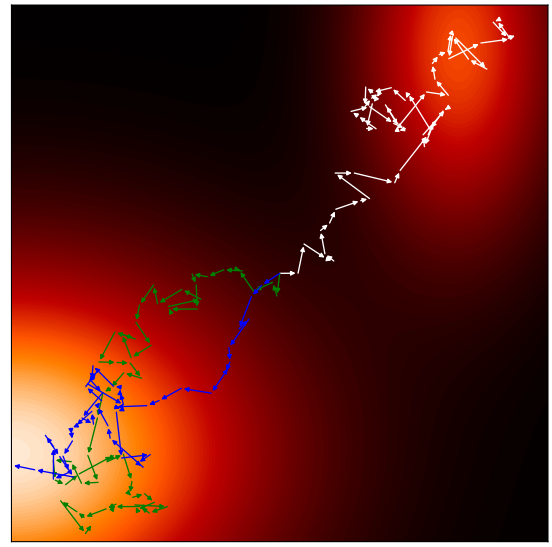


Figure 20.6 Examples of sampling trajectories obtained using Langevin dynamics defined by (14.61) for the distribution shown in Figure 20.5, showing three trajectories all starting at the centre of the plot.

For the Gaussian Parzen kernel (20.48), the score function becomes

$$\nabla_{\mathbf{z}} \ln q(\mathbf{z}|\mathbf{x}, \sigma) = -\frac{1}{\sigma}\boldsymbol{\epsilon} \tag{20.52}$$

where $\boldsymbol{\epsilon} = \mathbf{z} - \mathbf{x}$ is drawn from $\mathcal{N}(\mathbf{z}|\mathbf{0}, \mathbf{I})$. If we consider the specific noise model (20.6) then we obtain

$$\nabla_{\mathbf{z}} \ln q(\mathbf{z}|\mathbf{x}, \sigma) = -\frac{1}{\sqrt{1-\alpha_t}}\boldsymbol{\epsilon}. \tag{20.53}$$

We therefore see that the score loss (20.50) measures the difference between the neural network prediction and the noise $\boldsymbol{\epsilon}$. Therefore, this loss function has the same minimum as the form (20.37) used in the denoising diffusion model, with the score function $\mathbf{s}(\mathbf{z}, \mathbf{w})$ playing the same role as the noise prediction network $\mathbf{g}(\mathbf{z}, \mathbf{w})$ up to a constant scaling $-1/\sqrt{1-\alpha_t}$ (Song and Ermon, 2019). Minimizing (20.50) is known as *denoising score matching*, and we see the close connection to denoising diffusion models. There remains the question of how to choose the noise variance σ^2, and we will return to this shortly.

Having trained a score-based model we then need to draw new samples. Langevin dynamics is well-suited to score-based models because it is based on the score function and therefore does not require a normalized probability distribution, and is illustrated in Figure 20.6.

Section 14.3

20.3.3 Noise variance

We have seen how to learn the score function from a set of training data and how to generate new samples from the learned distribution using Langevin sampling. However, we can identify three potential problems with this approach (Song and

Telegram: @uni_k

www.konkur.in

Ermon, 2019; Luo, 2022). First, if the data distribution lies on a manifold of lower dimensionality than the data space, the probability density will be zero at points off the manifold and here the score function is undefined since $\ln p(\mathbf{x})$ is undefined. Second, in regions of low data density, the estimate of the score function may be inaccurate since the loss function (20.43) is weighted by the density. An inaccurate score function can lead to poor trajectories when using Langevin sampling. Third, even with an accurate model of the score function, the Langevin procedure may not sample correctly if the data distribution comprises a mixture of disjoint distributions.

Chapter 16

Exercise 20.18

All three problems can be addressed by choosing a sufficiently large value for the noise variance σ^2 used in the kernel function (20.48), because this smears out the data distribution. However, too large a variance will introduce a significant distortion of the original distribution and this itself introduces inaccuracies in the modelling of the score function. This trade-off can be addressed by considering a sequence of variance values $\sigma_1^2 < \sigma_2^2 < \ldots < \sigma_T^2$ (Song and Ermon, 2019), in which σ_1^2 is sufficiently small that the data distribution is accurately represented whereas σ_T^2 is sufficiently large that the aforementioned problems are avoided. The score network is then modified to take the variance as an additional input $\mathbf{s}(\mathbf{x}, \mathbf{w}, \sigma^2)$ and is trained by using a loss function that is a weighted sum of the loss functions of the form (20.51) in which each term represents the error between the associated network and the corresponding perturbed data set. For a data vector $\mathbf{x}_n$, the loss function then takes the form

$$\frac{1}{2}\sum_{i=1}^{L}\lambda(i)\int \left\|\mathbf{s}(\mathbf{z}, \mathbf{w}, \sigma_i^2) - \nabla_{\mathbf{z}} \ln q(\mathbf{z}|\mathbf{x}_n, \sigma_i)\right\|^2 q(\mathbf{z}|\mathbf{x}_n, \sigma_i)\,\mathrm{d}\mathbf{z} \tag{20.54}$$

where $\lambda(i)$ are weighting coefficients. We see that this training procedure precisely mirrors that used to train hierarchical denoising networks.

Section 20.2.1

Once trained, samples can be generated by running a few steps of Langevin sampling from each of the models for $i = L, L-1, \ldots, 2, 1$ in turn. This technique is called *annealed Langevin dynamics*, and is analogous to Algorithm 20.2 used to sample from denoising diffusion models.

20.3.4 Stochastic differential equations

We have seen that it is helpful to use a large number of steps, often several thousand, when constructing the noise process for a diffusion model. It is therefore natural to ask what happens if we consider the limit of an infinite number of steps, much as we did for infinitely deep neural networks when we introduced neural differential equations. In taking such a limit, we need to ensure that the noise variance β_t at each step becomes smaller in keeping with the step size. This leads to a formulation of diffusion models for continuous time as *stochastic differential equations* or SDEs (Song *et al.*, 2020). Both denoising diffusion probabilistic models and score matching models can then be viewed as a discretization of a continuous-time SDE.

Section 18.3.1

We can write a general SDE as an infinitesimal update to the vector $\mathbf{z}$ in the form

$$\mathrm{d}\mathbf{z} = \underbrace{\mathbf{f}(\mathbf{z}, t)\,\mathrm{d}t}_{\text{drift}} + \underbrace{g(t)\,\mathrm{d}\mathbf{v}}_{\text{diffusion}} \tag{20.55}$$

Telegram: @uni_k

www.konkur.in

where the drift term is deterministic, as in an ODE, but the diffusion term is stochastic, for example given by infinitesimal Gaussian steps. Here the parameter t is often called 'time' by analogy with physical systems. The forward noise process (20.3) for a diffusion model can be written as an SDE of the form (20.55) by taking the continuous-time limit.

Exercise 20.19

For the SDE (20.55), there is a corresponding reverse SDE (Song *et al.*, 2020) given by

$$\mathrm{d}\mathbf{z} = \left\{\mathbf{f}(\mathbf{z}, t) - g^2(t)\nabla_{\mathbf{z}} \ln p(\mathbf{z})\right\} \mathrm{d}t + g(t)\,\mathrm{d}\mathbf{v} \tag{20.56}$$

where we recognize $\nabla_{\mathbf{z}} \ln p(\mathbf{z})$ as the score function. The SDE given by (20.55) is to be solved in reverse from $t = T$ to $t = 0$.

To solve an SDE numerically, we need to discretize the time variable. The simplest approach is to use fixed, equally spaced time steps, which is known as the Euler–Maruyama solver. For the reverse SDE, we then recover a form of the Langevin equation. However, more sophisticated solvers can be employed that use more flexible forms of discretization (Kloeden and Platen, 2013).

Section 14.3

For all diffusion processes governed by an SDE, there exists a corresponding deterministic process described by an ODE whose trajectories have the same marginal probability densities $p(\mathbf{z}|t)$ as the SDE (Song *et al.*, 2020). For an SDE of the form (20.56), the corresponding ODE is given by

$$\frac{\mathrm{d}\mathbf{z}}{\mathrm{d}t} = \mathbf{f}(\mathbf{z}, t) - \frac{1}{2}g^2(t)\nabla_{\mathbf{z}} \ln p(\mathbf{z}). \tag{20.57}$$

The ODE formulation allows the use of efficient adaptive-step solvers to reduce the number of function evaluations dramatically. Moreover, it allows probabilistic diffusion models to be related to normalizing flow models, from which the change-of-variables formula (18.1) can be used to provide an exact evaluation of the log likelihood.

Chapter 18

20.4. Guided Diffusion

So far, we have considered diffusion models as a way to represent the unconditional density $p(\mathbf{x})$ learned from a set of training examples $\mathbf{x}_1, \ldots, \mathbf{x}_N$ drawn independently from $p(\mathbf{x})$. Once the model has been trained, we can generate new samples from this distribution. We have already seen an example of unconditional sampling from a deep generative model for face images in Figure 1.3, in that case from a GAN model.

In many applications, however, we want to sample from a conditional distribution $p(\mathbf{x}|\mathbf{c})$ where the conditioning variable $\mathbf{c}$ could, for example, be a class label or a textual description of the desired content for an image. This also forms the basis for applications such as image super-resolution, image inpainting, video generation, and many others. The simplest approach to achieving this would be to treat $\mathbf{c}$ as an additional input into the denoising neural network $\mathbf{g}(\mathbf{z}, \mathbf{w}, t, \mathbf{c})$ and then to train the network using matched pairs $\{\mathbf{x}_n, \mathbf{c}_n\}$. The main limitation of this approach is that

Telegram: @uni_k

www.konkur.in

the network can give insufficient weight to, or even ignore, the conditioning variables, so we need a way to control how much weight is given to the conditioning information and to trade this off against sample diversity. This additional pressure to match the conditioning information is called *guidance*. There are two main approaches to guidance depending on whether or not a separate classifier model is used.

20.4.1 Classifier guidance

Suppose that a trained classifier $p(\mathbf{c}|\mathbf{x})$ is available, and consider a diffusion model from the perspective of the score function. Using Bayes' theorem we can write the score function for the conditional diffusion model in the form

$$\begin{aligned}\nabla_{\mathbf{x}} \ln p(\mathbf{x}|\mathbf{c}) &= \nabla_{\mathbf{x}} \ln \left\{ \frac{p(\mathbf{c}|\mathbf{x})p(\mathbf{x})}{p(\mathbf{c})} \right\} \\ &= \nabla_{\mathbf{x}} \ln p(\mathbf{x}) + \nabla_{\mathbf{x}} \ln p(\mathbf{c}|\mathbf{x}) \end{aligned} \tag{20.58}$$

where we have used $\nabla_{\mathbf{x}} \ln p(\mathbf{c}) = 0$ since $p(\mathbf{c})$ is independent of $\mathbf{x}$. The first term on the right-hand side of (20.58) is the usual unconditional score function, whereas the second term pushes the denoising process towards the direction that maximizes the probability of the given label $\mathbf{c}$ under the classifier model (Dhariwal and Nichol, 2021). The influence of the classifier can be controlled by introducing a hyperparameter λ, called the *guidance scale*, which controls the weight given to the classifier gradient. The score function used for sampling then becomes

$$\text{score}(\mathbf{x}, \mathbf{c}, \lambda) = \nabla_{\mathbf{x}} \ln p(\mathbf{x}) + \lambda \nabla_{\mathbf{x}} \ln p(\mathbf{c}|\mathbf{x}). \tag{20.59}$$

If $\lambda = 0$ we recover the original unconditional diffusion model, whereas if $\lambda = 1$ we obtain the score corresponding to the conditional distribution $p(\mathbf{x}|\mathbf{c})$. For $\lambda > 1$ the model is strongly encouraged to respect the conditioning label, and values of $\lambda \gg 1$ may be used, for example $\lambda = 10$. However, this comes at the expense of diversity in the samples as the model prefers 'easy' examples that the classifier is able to classify correctly.

One problem with the classifier-based approach to guidance is that a separate classifier must be trained. Furthermore, this classifier needs to be able to classify examples with varying degrees of noise, whereas standard classifiers are trained on clean examples. We therefore turn to an alternative approach that avoids the use of a separate classifier.

20.4.2 Classifier-free guidance

If we use (20.58) to replace $\nabla_{\mathbf{x}} \ln p(\mathbf{c}|\mathbf{x})$ in (20.59), we can write the score function in the form

Exercise 20.20

$$\text{score}(\mathbf{x}, \mathbf{c}, \lambda) = \lambda \nabla_{\mathbf{x}} \ln p(\mathbf{x}|\mathbf{c}) + (1 - \lambda) \nabla_{\mathbf{x}} \ln p(\mathbf{x}), \tag{20.60}$$

which for $0 < \lambda < 1$ represents a convex combination of the conditional log density $\ln p(\mathbf{x}|\mathbf{c})$ and the unconditional log density $\ln p(\mathbf{x})$. For $\lambda > 1$ the contribution from

Telegram: @uni_k

www.konkur.in

the unconditional score becomes negative, meaning the model actively reduces the probability of generating samples that ignore the conditioning information in favour of samples that do.

Furthermore, we can avoid training separate networks to model $p(\mathbf{x}|\mathbf{c})$ and $p(\mathbf{x})$ by training a single conditional model in which the conditioning variable $\mathbf{c}$ is set to a null value, for example $\mathbf{c} = \mathbf{0}$, with some probability during training, typically around 10–20%. Then $p(\mathbf{x})$ is represented by $p(\mathbf{x}|\mathbf{c} = \mathbf{0})$. This is somewhat analogous to dropout in which the conditioning inputs are collectively set to zero for a random subset of training vectors.

Section 9.6.1

Once trained, the score function (20.60) is then used to encourage a strong weighting of the conditional information. In practice, classifier-free guidance gives much higher quality results than classifier guidance (Nichol *et al.*, 2021; Saharia *et al.*, 2022). The reason is that a classifier $p(\mathbf{c}|\mathbf{x})$ can ignore most of the input vector $\mathbf{x}$ as long as it makes a good prediction of $\mathbf{c}$ whereas classifier-free guidance is based on the conditional density $p(\mathbf{x}|\mathbf{c})$, which must assign a high probability to all aspects of $\mathbf{x}$.

Text-guided diffusion models can leverage techniques from large language models to allow the conditioning input to be a general text sequence, known as a *prompt*, and not simply a selection from a predefined set of class labels. This allows the text input to influence the denoising process in two ways, first by concatenating the internal representation from a transformer-based language model with the input to the denoising network and second by allowing cross-attention layers within the denoising network to attend to the text token sequence. Classifier-free guidance, conditioned on a text prompt, is illustrated in Figure 20.7.

Chapter 12

Another application for conditional diffusion models is image super-resolution in which a low-resolution image is transformed into a corresponding high-resolution image. This is intrinsically an inverse problem, and multiple high-resolution images will be consistent with a given low-resolution image. Super-resolution can be achieved by denoising a high-resolution sample from a Gaussian using the low-resolution image as a conditioning variable (Saharia, Ho, *et al.*, 2021). Examples of this method are shown in Figure 20.8. Such models can be cascaded to achieve very high resolution (Ho *et al.*, 2021), for example going from 64×64 to 256×256, and then from 256×256 to 1024×1024. Each stage is typically represented by a U-net architecture, with each U-net conditioned on the final denoised output of the previous one.

Section 10.5.4

This type of cascade can also be used with image-generation diffusion models, in which the image denoising is performed at a lower resolution and the result is subsequently up-sampled using a separate network (which may also take a text prompt as input) to give a final high-resolution output (Nichol *et al.*, 2021; Saharia *et al.*, 2022). This can significantly reduce the computational cost compared to working directly in a high-dimensional space since the denoising process may involve hundreds of passes through the denoising network. Note that these approaches still work within the image space directly but at lower resolution.

A different approach to addressing the high computational cost of applying diffusion models directly in the space of high-resolution images is called *latent diffu-*

Telegram: @uni_k

www.konkur.in



Figure 20.7 Illustration of classifier-free guidance of diffusion models, generated from a model called GLIDE using the conditioning text *A stained glass window of a panda eating bamboo*. Examples on the left were generated with $\lambda = 0$ (no guidance, just the plain conditional model) whereas examples on the right were generated with $\lambda = 3$. [From Nichol *et al.* (2021) with permission.]

Section 19.1

sion models (Rombach *et al.*, 2021). Here an autoencoder is first trained on noise-free images to obtain a lower-dimensional representation of the images and is then fixed. A U-net architecture is then trained to perform the denoising within the lower-dimensional space, which itself is not directly interpretable as an image. Finally, the denoised representation is mapped into the high-resolution image space using the output half of the fixed autoencoder network. This approach makes more efficient use of the low-dimensional space, which can then focus on image semantics, leaving the decoder to create a corresponding sharp, high-resolution image from the denoised low-dimensional representation.

There are many other applications of conditional image generation including inpainting, un-cropping, restoration, image morphing, style transfer, colourization, de-blurring, and video generation (Yang, Srivastava, and Mandt, 2022). An example of inpainting is shown in Figure 20.9.

Telegram: @uni_k

www.konkur.in

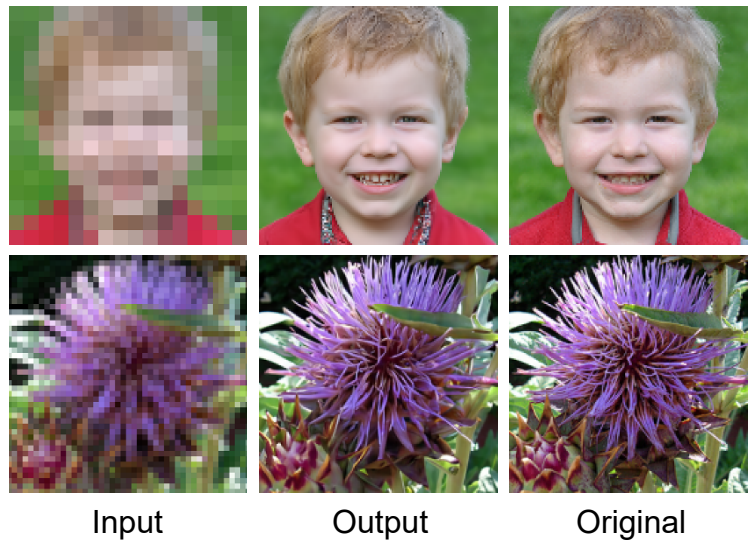


Figure 20.8 Two examples of low-resolution images along with associated samples of corresponding high-resolution images generated by a diffusion model. The top row shows a 16×16 input image and the corresponding 128×128 output image along with the original image from which the input image was generated. The bottom row shows a 64×64 input image with a 256×256 output image, again with the original image for comparison. [From Saharia, Ho, *et al.* (2021) with permission.]



Figure 20.9 Example of inpainting showing the original image on the left, an image with sections removed in the middle, and the image with inpainting on the right. [From Saharia, Chan, Chang, *et al.* (2021) with permission.]

Exercises

20.1 ($\star$) Using (20.3) write down expressions for the mean and covariance of $\mathbf{z}_t$ in terms of the mean and covariance of $\mathbf{z}_{t-1}$. Hence, show that for $0 < \beta_t < 1$ the mean of the distribution of $\mathbf{z}_t$ is closer to zero than the mean of $\mathbf{z}_{t-1}$ and that the covariance of $\mathbf{z}_t$ is closer to the unit matrix $\mathbf{I}$ than the covariance of $\mathbf{z}_{t-1}$.

20.2 ($\star$) Show that the transformation (20.1) can be written in the equivalent form (20.2).

20.3 ($\star\star\star$) In this exercise we use proof by induction to show that the marginal distribution of $\mathbf{x}_t$ for the forward process of the diffusion model, as defined by (20.4), is given by (20.6) where α_t is defined by (20.7). First verify that (20.6) holds when $t = 1$. Now assume that (20.6) is true for some particular value of t and derive the corresponding result for the value $t + 1$. To do this, it is easiest to write the forward process using the representation (20.3) and to make use of the result (3.212), which shows that the sum of two independent Gaussian random variables is itself a Gaussian in which the means and covariances are additive.

20.4 ($\star$) By using the result (20.6), where α_t is defined by (20.7), show that in the limit $T \to \infty$ we obtain (20.9).

Telegram: @uni_k

www.konkur.in

20.5 (⋆⋆) Consider two independent random variables $\mathbf{a}$ and $\mathbf{b}$ along with a fixed scalar λ. Show that

$$\text{cov}[\mathbf{a}+\mathbf{b}] = \text{cov}[\mathbf{a}] + \text{cov}[\mathbf{b}] \tag{20.61}$$

$$\text{cov}[\lambda\mathbf{a}] = \lambda^2\text{cov}[\mathbf{a}]. \tag{20.62}$$

Use these results to show that if the distribution of $\mathbf{z}_{t-1}$ has zero mean and unit covariance, then the distribution of $\mathbf{z}_t$, defined by (20.3), will also have zero mean and unit covariance, irrespective of the value of β_t.

20.6 (⋆⋆⋆) In this exercise we will use the technique of completing the square to derive the result (20.15) starting from Bayes' theorem (20.13). First note that the two terms in the numerator on the right-hand side of (20.13), given by (20.4) and (20.6), both take the form of exponentials of quadratic functions of $\mathbf{z}_{t-1}$. The required distribution is therefore a Gaussian, and so we need only to find its mean and covariance. To do this, consider only the terms in the exponentials that depend on $\mathbf{z}_{t-1}$ and note that the product of two exponentials is the exponential of the sum of the two exponents. Gather together all the terms that are quadratic in $\mathbf{z}_{t-1}$ as well as those that are linear in $\mathbf{z}_{t-1}$ and then rearrange them in the form $(\mathbf{z}_{t-1}-\mathbf{m}_t)^{\mathrm{T}}\mathbf{S}_t^{-1}(\mathbf{z}_{t-1}-\mathbf{m}_t)$. Then, by inspection, find expressions for $\mathbf{m}_t(\mathbf{x},\mathbf{z}_t)$ and $\mathbf{S}_t$. Note that additive terms that are independent of $\mathbf{z}_{t-1}$ can be ignored.

20.7 (⋆⋆⋆) In this exercise we show that the reverse of the conditional distribution $q(\mathbf{z}_t|\mathbf{z}_{t-1})$ for the forward noise process in a diffusion model can be approximated by a Gaussian when the noise variance is small. Consider the inverse conditional distribution $q(\mathbf{z}_{t-1}|\mathbf{z}_t)$ given by Bayes' theorem in the form (20.11) where the forward distribution $q(\mathbf{z}_t|\mathbf{z}_{t-1})$ is given by (20.4). By taking the logarithm of both sides of (20.11) and then making a Taylor expansion of $q(\mathbf{z}_{t-1})$ centred on the value $\mathbf{z}_t$, show that, for small values of the noise variance β_t, the distribution $q(\mathbf{z}_{t-1}|\mathbf{z}_t)$ is approximately a Gaussian with mean $\mathbf{z}_t$ and covariance $\beta_t\mathbf{I}$. Find expressions for the lowest-order corrections to the mean and to the covariance as expansions in powers of β_t.

20.8 (⋆⋆) By substituting the product rule of probability in the form (20.24) into the definition (20.22) of the ELBO for the diffusion model and making use of the definition (20.23) of the Kullback–Leibler divergence, verify that the log likelihood function can be written as the sum of a lower bound and a Kullback–Leibler divergence in the form (20.21).

20.9 (⋆⋆) Verify that the ELBO for the diffusion model given by (20.31) can be written in the form (20.32) where the Kullback–Leibler divergence is defined by (20.23).

20.10 (⋆⋆) When we derived the ELBO for the diffusion model given by (20.32), we omitted the first and third terms in (20.26) because they are independent of $\mathbf{w}$. Similarly we omitted the second term in the right-hand side of (20.30) because this is also independent of $\mathbf{w}$. Show that if all of these omitted terms are retained they lead to an additional term in the ELBO $\mathcal{L}(\mathbf{x})$ given by

$$\text{KL}\left(q(\mathbf{z}_T|\mathbf{x})\|p(\mathbf{z}_T)\right). \tag{20.63}$$

Telegram: @uni_k

www.konkur.in

Note that the noise process is constructed in such a way that the distribution $q(\mathbf{z}_T|\mathbf{x})$ is equal to the Gaussian $\mathcal{N}(\mathbf{x}|\mathbf{0}, \mathbf{I})$. Similarly, the distribution $p(\mathbf{z}_T)$ is defined to be equal to $\mathcal{N}(\mathbf{x}|\mathbf{0}, \mathbf{I})$, and hence the two distributions in (20.63) are equal and so the Kullback–Leibler divergence vanishes.

20.11 ($\star\star$) By making use of (20.15) for the distribution $q(\mathbf{z}_{t-1}|\mathbf{z}_t, \mathbf{x})$ and (20.18) for the distribution $p(\mathbf{z}_{t-1}|\mathbf{z}_t, \mathbf{w})$, show that the Kullback–Leibler divergence appearing in the consistency terms in (20.32) is given by (20.33).

20.12 ($\star\star$) By substituting (20.34) into (20.16) rewrite the mean $\mathbf{m}_t(\mathbf{x}, \mathbf{z}_t)$ in terms of the original data vector $\mathbf{x}$ and the noise $\boldsymbol{\epsilon}$ in the form (20.35), where α_t is defined by (20.7).

20.13 ($\star\star$) Show that the reconstruction term (20.38) in the ELBO for diffusion models can be written in the form (20.39). To do this, substitute for $\boldsymbol{\mu}(\mathbf{z}_1, \mathbf{w}, 1)$ using (20.36) and substitute for $\mathbf{x}$ using (20.1), and then make use of $\alpha_1 = (1 - \beta_1)$, which follows from (20.7).

20.14 ($\star$) The score function is defined by $\mathbf{s}(\mathbf{x}) = \nabla_{\mathbf{x}} p(\mathbf{x}|\mathbf{w})$ and is therefore a vector of the same dimensionality as the input vector $\mathbf{x}$. Consider a matrix whose elements are given by

$$M_{ij} = \frac{\partial s_i}{\partial x_j} - \frac{\partial s_j}{\partial x_i}. \tag{20.64}$$

Show that if the score function is defined by taking the gradient $\mathbf{s} = \nabla_{\mathbf{x}} \phi(\mathbf{x})$ of the output of a neural network with a single output variable $\phi(\mathbf{x})$, then all the matrix elements $M_{ij} = 0$ for all pairs i, j. Note that if the score function $\mathbf{s}(\mathbf{x}) = \nabla_{\mathbf{x}} p(\mathbf{x}|\mathbf{w})$ is instead represented directly by a deep neural network with the same number of outputs as inputs, then only the diagonal matrix elements $M_{ii} = 0$, and so the output of the network does not in general correspond to the gradient of any scalar function.

20.15 ($\star\star$) Consider a deep neural network representation $\mathbf{s}(\mathbf{x}, \mathbf{w})$ for the score function defined by (20.42), where $\mathbf{x}$ and $\mathbf{s}$ have dimensionality D. Compare the computational complexity of evaluating the score for a network with D outputs that represents the score function directly with one that computes a single scalar function $\phi(\mathbf{x}, \mathbf{w})$ in which the score function is computed indirectly through automatic differentiation. Show that the latter approach is typically more computationally expensive.

20.16 ($\star\star\star$) We cannot minimize the score function (20.43) directly because we do not know the functional form of the true data density $p(\mathbf{x})$, and therefore we cannot write down an expression for the score function $\nabla_{\mathbf{x}} \ln p(\mathbf{x})$. However, by using integration by parts (Hyvärinen, 2005), we can rewrite (20.43) in the form

$$J(\mathbf{w}) = \int \left\{ \nabla \cdot \mathbf{s}(\mathbf{x}, \mathbf{w}) + \frac{1}{2} \|\mathbf{s}(\mathbf{x}, \mathbf{w})\|^2 \right\} p(\mathbf{x}) \, \mathrm{d}\mathbf{x} + \text{const} \tag{20.65}$$

Telegram: @uni_k

www.konkur.in

where the constant term is independent of the network parameters $\mathbf{w}$, and the divergence $\nabla \cdot \mathbf{s}(\mathbf{x}, \mathbf{w})$ is defined by

$$\nabla \cdot \mathbf{s} = \sum_{i=1}^{D} \frac{\partial s_i}{\partial x_i} = \sum_{i=1}^{D} \frac{\partial^2 \ln p(\mathbf{x})}{\partial x_i^2} \tag{20.66}$$

in which D is the dimensionality of $\mathbf{x}$. Derive the result (20.65) by first expanding the square in (20.43) and noting that the term involving $\|\mathbf{s}(\mathbf{x}, \mathbf{w})\|^2$ already appears in (20.43) whereas the term involving $\|\mathbf{s}_\mathcal{D}\|^2$ can be absorbed into the additive constant, where we have defined $\mathbf{s}_\mathcal{D} = \nabla \ln p_\mathcal{D}(\mathbf{x})$. Now consider the formula

$$\frac{\mathrm{d}}{\mathrm{d}x}\{p(x)g(x)\} = \frac{\mathrm{d}p(x)}{\mathrm{d}x}g(x) + p(x)\frac{\mathrm{d}g(x)}{\mathrm{d}x} \tag{20.67}$$

for the derivative of the product of two functions. Integrate both sides of this formula with respect to x and rearrange to obtain the integration-by-parts formula:

$$\int_{-\infty}^{\infty} \frac{\mathrm{d}p(x)}{\mathrm{d}x}g(x)\,\mathrm{d}x = -\int_{-\infty}^{\infty} \frac{\mathrm{d}g(x)}{\mathrm{d}x}p(x)\,\mathrm{d}x \tag{20.68}$$

where we have assumed that $p(\infty) = p(-\infty) = 0$. Apply this result together with the definition $\mathbf{s}_\mathcal{D} = \nabla \ln p(\mathbf{x})$ to the term involving $\mathbf{s}(\mathbf{x}, \mathbf{w})^{\mathrm{T}}\mathbf{s}_\mathcal{D}$ to complete the proof. Note that the evaluation of the second derivatives in (20.66) requires a separate backward propagation pass for each derivative and hence has an overall computational cost that grows quadratically with the dimensionality D of the data space (Martens, Sutskever, and Swersky, 2012). This precludes the direct application of this loss function to spaces of high dimensionality, and so techniques such as *sliced score matching* (Song *et al.*, 2019) have been developed to help address this inefficiency.

20.17 ($\star\star$) In this exercise we show that the score function loss (20.50) is equivalent, up to an additive constant, to the form (20.49). To do this, first expand the square in (20.49) and by using (20.47) show that the term in $\mathbf{s}^{\mathrm{T}}\mathbf{s}$ from (20.49) is the same as the corresponding term obtained by expanding the square in (20.50). Next note that the term in $\|\nabla_\mathbf{z} \ln q\|^2$ in (20.49) is independent of $\mathbf{w}$ and likewise that the corresponding term in (20.50) is also independent of $\mathbf{w}$, and so these can be viewed as additive constants in the loss function and play no role in training. Finally, consider the cross-term in (20.49). By substituting for $q(\mathbf{z})$ using (20.47), show that this is equal to the corresponding cross-term from (20.50). Hence, show that the two loss functions are equal up to an additive constant.

20.18 ($\star$) Consider a probability distribution that consists of a mixture of two disjoint distributions (i.e., distributions with the property that when one of them is non-zero the other must be zero) of the form

$$p(\mathbf{x}) = \lambda p_A(\mathbf{x}) + (1 - \lambda)p_B(\mathbf{x}). \tag{20.69}$$

Telegram: @uni_k

www.konkur.in

Show that when the score function, defined by (20.42), is evaluated for any given point $\mathbf{x}$, the mixing coefficient λ does not appear. From this it follows that Langevin dynamics defined by (14.61) will not sample from the two component distributions with the correct proportions. This problem is resolved by adding noise from a broad distribution, as discussed in the text.

20.19 ($\star\star$) For discrete steps, the forward noise process in a diffusion model is defined by (20.3). Here we take the continuous-time limit and convert this to an SDE. We first introduce a continuously-changing variance function $\beta(t)$ such that $\beta_t = \beta(t)\Delta t$. By making a Taylor expansion of the square root in the first term on the right-hand-side of (20.3), show that the infinitesimal update can be written in the form

$$\mathrm{d}\mathbf{z} = -\frac{1}{2}\beta(t)\mathbf{z}\,\mathrm{d}t + \sqrt{\beta(t)}\,\mathrm{d}\mathbf{v}. \tag{20.70}$$

We see that this is a special case of the general SDE (20.55).

20.20 ($\star$) By using (20.58) to replace $\nabla_{\mathbf{x}} \ln p(\mathbf{c}|\mathbf{x})$, show that the score function in (20.59) can be written in the form (20.60).

Telegram: @uni_k

www.konkur.in

Appendix A. Linear Algebra

In this appendix, we gather together some useful properties and identities involving matrices and determinants. This is not intended to be an introductory tutorial, and it is assumed that the reader is already familiar with basic linear algebra. For some results, we indicate how to prove them, whereas in more complex cases we leave the interested reader to refer to standard textbooks on the subject. In all cases, we assume that inverses exist and that matrix dimensions are such that the formulae are correctly defined. A comprehensive discussion of linear algebra can be found in Golub and Van Loan (1996), and an extensive collection of matrix properties is given by Lütkepohl (1996). Matrix derivatives are discussed in Magnus and Neudecker (1999).

A.1. Matrix Identities

A matrix $\mathbf{A}$ has elements A_{ij} where i indexes the rows and j indexes the columns. We use $\mathbf{I}_N$ to denote the $N \times N$ identity matrix (also called the unit matrix), and if there is no ambiguity over dimensionality, we simply use $\mathbf{I}$. The transpose matrix $\mathbf{A}^{\mathrm{T}}$ has elements $(\mathbf{A}^{\mathrm{T}})_{ij} = A_{ji}$. From the definition of a transpose, we have

$$(\mathbf{AB})^{\mathrm{T}} = \mathbf{B}^{\mathrm{T}}\mathbf{A}^{\mathrm{T}}, \tag{A.1}$$

which can be verified by writing out the indices. The inverse of $\mathbf{A}$, denoted $\mathbf{A}^{-1}$, satisfies

$$\mathbf{A}\mathbf{A}^{-1} = \mathbf{A}^{-1}\mathbf{A} = \mathbf{I}. \tag{A.2}$$

Because $\mathbf{ABB}^{-1}\mathbf{A}^{-1} = \mathbf{I}$, we have

$$(\mathbf{AB})^{-1} = \mathbf{B}^{-1}\mathbf{A}^{-1}. \tag{A.3}$$

Also we have

$$\left(\mathbf{A}^{\mathrm{T}}\right)^{-1} = \left(\mathbf{A}^{-1}\right)^{\mathrm{T}}, \tag{A.4}$$

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024
C. M. Bishop, H. Bishop, *Deep Learning*, https://doi.org/10.1007/978-3-031-45468-4

Telegram: @uni_k

www.konkur.in

which is easily proven by taking the transpose of (A.2) and applying (A.1).

A useful identity involving matrix inverses is the following:

$$(\mathbf{P}^{-1} + \mathbf{B}^{\mathrm{T}}\mathbf{R}^{-1}\mathbf{B})^{-1}\mathbf{B}^{\mathrm{T}}\mathbf{R}^{-1} = \mathbf{P}\mathbf{B}^{\mathrm{T}}(\mathbf{B}\mathbf{P}\mathbf{B}^{\mathrm{T}} + \mathbf{R})^{-1}, \tag{A.5}$$

which is easily verified by right-multiplying both sides by $(\mathbf{B}\mathbf{P}\mathbf{B}^{\mathrm{T}} + \mathbf{R})$. Suppose that $\mathbf{P}$ has dimensionality $N \times N$ and that $\mathbf{R}$ has dimensionality $M \times M$, so that $\mathbf{B}$ is $M \times N$. Then if $M \ll N$, it will be much cheaper to evaluate the right-hand side of (A.5) than the left-hand side. A special case that sometimes arises is

$$(\mathbf{I} + \mathbf{A}\mathbf{B})^{-1}\mathbf{A} = \mathbf{A}(\mathbf{I} + \mathbf{B}\mathbf{A})^{-1}. \tag{A.6}$$

Another useful identity involving inverses is the following:

$$(\mathbf{A} + \mathbf{B}\mathbf{D}^{-1}\mathbf{C})^{-1} = \mathbf{A}^{-1} - \mathbf{A}^{-1}\mathbf{B}(\mathbf{D} + \mathbf{C}\mathbf{A}^{-1}\mathbf{B})^{-1}\mathbf{C}\mathbf{A}^{-1}, \tag{A.7}$$

which is known as the *Woodbury identity*. It can be verified by multiplying both sides by $(\mathbf{A} + \mathbf{B}\mathbf{D}^{-1}\mathbf{C})$. This is useful, for instance, when $\mathbf{A}$ is large and diagonal and hence easy to invert, and when $\mathbf{B}$ has many rows but few columns (and conversely for $\mathbf{C}$), so that the right-hand side is much cheaper to evaluate than the left-hand side.

A set of vectors $\{\mathbf{a}_1, \ldots, \mathbf{a}_N\}$ is said to be *linearly independent* if the relation $\sum_n \alpha_n \mathbf{a}_n = 0$ holds only if all $\alpha_n = 0$. This implies that none of the vectors can be expressed as a linear combination of the remainder. The rank of a matrix is the maximum number of linearly independent rows (or equivalently the maximum number of linearly independent columns).

A.2. Traces and Determinants

Square matrices have traces and determinants. The trace $\mathrm{Tr}(\mathbf{A})$ of a matrix $\mathbf{A}$ is defined as the sum of the elements on the leading diagonal. By writing out the indices, we see that

$$\mathrm{Tr}(\mathbf{A}\mathbf{B}) = \mathrm{Tr}(\mathbf{B}\mathbf{A}). \tag{A.8}$$

By applying this formula multiple times to the product of three matrices, we see that

$$\mathrm{Tr}(\mathbf{A}\mathbf{B}\mathbf{C}) = \mathrm{Tr}(\mathbf{C}\mathbf{A}\mathbf{B}) = \mathrm{Tr}(\mathbf{B}\mathbf{C}\mathbf{A}), \tag{A.9}$$

which is known as the *cyclic* property of the trace operator. It clearly extends to the product of any number of matrices. The determinant $|\mathbf{A}|$ of an $N \times N$ matrix $\mathbf{A}$ is defined by

$$|\mathbf{A}| = \sum (\pm 1) A_{1i_1} A_{2i_2} \cdots A_{Ni_N} \tag{A.10}$$

in which the sum is taken over all products consisting of precisely one element from each row and one element from each column, with a coefficient $+1$ or -1 according to whether the permutation $i_1 i_2 \ldots i_N$ is even or odd, respectively. Note that $|\mathbf{I}| = 1$,

Telegram: @uni_k

www.konkur.in

and that the determinant of a diagonal matrix is given by the product of the elements on the leading diagonal. Thus, for a 2×2 matrix, the determinant takes the form

$$|\mathbf{A}| = \begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} = a_{11}a_{22} - a_{12}a_{21}. \tag{A.11}$$

The determinant of a product of two matrices is given by

$$|\mathbf{AB}| = |\mathbf{A}||\mathbf{B}| \tag{A.12}$$

as can be shown from (A.10). Also, the determinant of an inverse matrix is given by

$$\left|\mathbf{A}^{-1}\right| = \frac{1}{|\mathbf{A}|}, \tag{A.13}$$

which can be shown by taking the determinant of (A.2) and applying (A.12).

If $\mathbf{A}$ and $\mathbf{B}$ are matrices of size $N \times M$, then

$$\left|\mathbf{I}_N + \mathbf{A}\mathbf{B}^{\mathrm{T}}\right| = \left|\mathbf{I}_M + \mathbf{A}^{\mathrm{T}}\mathbf{B}\right|. \tag{A.14}$$

A useful special case is

$$\left|\mathbf{I}_N + \mathbf{a}\mathbf{b}^{\mathrm{T}}\right| = 1 + \mathbf{a}^{\mathrm{T}}\mathbf{b} \tag{A.15}$$

where $\mathbf{a}$ and $\mathbf{b}$ are N-dimensional column vectors.

A.3. Matrix Derivatives

Sometimes we need to consider derivatives of vectors and matrices with respect to scalars. The derivative of a vector $\mathbf{a}$ with respect to a scalar x is a vector whose components are given by

$$\left(\frac{\partial \mathbf{a}}{\partial x}\right)_i = \frac{\partial a_i}{\partial x} \tag{A.16}$$

with an analogous definition for the derivative of a matrix. Derivatives with respect to vectors and matrices can also be defined, for instance

$$\left(\frac{\partial x}{\partial \mathbf{a}}\right)_i = \frac{\partial x}{\partial a_i} \tag{A.17}$$

and similarly

$$\left(\frac{\partial \mathbf{a}}{\partial \mathbf{b}}\right)_{ij} = \frac{\partial a_i}{\partial b_j}. \tag{A.18}$$

The following is easily proven by writing out the components:

$$\frac{\partial}{\partial \mathbf{x}}\left(\mathbf{x}^{\mathrm{T}}\mathbf{a}\right) = \frac{\partial}{\partial \mathbf{x}}\left(\mathbf{a}^{\mathrm{T}}\mathbf{x}\right) = \mathbf{a}. \tag{A.19}$$

Telegram: @uni_k

www.konkur.in

Similarly

$$\frac{\partial}{\partial x}(\mathbf{AB}) = \frac{\partial \mathbf{A}}{\partial x}\mathbf{B} + \mathbf{A}\frac{\partial \mathbf{B}}{\partial x}. \tag{A.20}$$

The derivative of the inverse of a matrix can be expressed as

$$\frac{\partial}{\partial x}\left(\mathbf{A}^{-1}\right) = -\mathbf{A}^{-1}\frac{\partial \mathbf{A}}{\partial x}\mathbf{A}^{-1} \tag{A.21}$$

as can be shown by differentiating the equation $\mathbf{A}^{-1}\mathbf{A} = \mathbf{I}$ using (A.20) and then right-multiplying by $\mathbf{A}^{-1}$. Also

$$\frac{\partial}{\partial x}\ln|\mathbf{A}| = \mathrm{Tr}\left(\mathbf{A}^{-1}\frac{\partial \mathbf{A}}{\partial x}\right), \tag{A.22}$$

which we shall prove later. If we choose x to be one of the elements of $\mathbf{A}$, we have

$$\frac{\partial}{\partial A_{ij}}\mathrm{Tr}\,(\mathbf{AB}) = B_{ji} \tag{A.23}$$

as can be seen by writing out the matrices using index notation. We can write this result more compactly in the form

$$\frac{\partial}{\partial \mathbf{A}}\mathrm{Tr}\,(\mathbf{AB}) = \mathbf{B}^{\mathrm{T}}. \tag{A.24}$$

With this notation, we have the following properties:

$$\frac{\partial}{\partial \mathbf{A}}\mathrm{Tr}\left(\mathbf{A}^{\mathrm{T}}\mathbf{B}\right) = \mathbf{B}, \tag{A.25}$$

$$\frac{\partial}{\partial \mathbf{A}}\mathrm{Tr}(\mathbf{A}) = \mathbf{I}, \tag{A.26}$$

$$\frac{\partial}{\partial \mathbf{A}}\mathrm{Tr}(\mathbf{ABA}^{\mathrm{T}}) = \mathbf{A}(\mathbf{B} + \mathbf{B}^{\mathrm{T}}), \tag{A.27}$$

which can again be proven by writing out the matrix indices. We also have

$$\frac{\partial}{\partial \mathbf{A}}\ln|\mathbf{A}| = \left(\mathbf{A}^{-1}\right)^{\mathrm{T}}, \tag{A.28}$$

which follows from (A.22) and (A.24).

A.4. Eigenvectors

For a square matrix $\mathbf{A}$ of size $M \times M$, the eigenvector equation is defined by

$$\mathbf{A}\mathbf{u}_i = \lambda_i \mathbf{u}_i \tag{A.29}$$

Telegram: @uni_k

www.konkur.in

for $i = 1, \ldots, M$, where $\mathbf{u}_i$ is an *eigenvector* and λ_i is the corresponding *eigenvalue*. This can be viewed as a set of M simultaneous homogeneous linear equations, and the condition for a solution is that

$$|\mathbf{A} - \lambda_i \mathbf{I}| = 0, \tag{A.30}$$

which is known as the *characteristic equation*. Because this is a polynomial of order M in λ_i, it must have M solutions (though these need not all be distinct). The rank of $\mathbf{A}$ is equal to the number of non-zero eigenvalues.

Of particular interest are symmetric matrices, which arise as covariance matrices, kernel matrices, and Hessians. Symmetric matrices have the property that $A_{ij} = A_{ji}$, or equivalently $\mathbf{A}^{\mathrm{T}} = \mathbf{A}$. The inverse of a symmetric matrix is also symmetric, as can be seen by taking the transpose of $\mathbf{A}^{-1}\mathbf{A} = \mathbf{I}$ and using $\mathbf{A}\mathbf{A}^{-1} = \mathbf{I}$ together with the symmetry of $\mathbf{I}$.

In general, the eigenvalues of a matrix are complex numbers, but for symmetric matrices, the eigenvalues λ_i are real. This can be seen by first left-multiplying (A.29) by $(\mathbf{u}_i^\star)^{\mathrm{T}}$, where $\star$ denotes the complex conjugate, to give

$$(\mathbf{u}_i^\star)^{\mathrm{T}} \mathbf{A}\mathbf{u}_i = \lambda_i (\mathbf{u}_i^\star)^{\mathrm{T}} \mathbf{u}_i. \tag{A.31}$$

Next we take the complex conjugate of (A.29) and left-multiply by $\mathbf{u}_i^{\mathrm{T}}$ to give

$$\mathbf{u}_i^{\mathrm{T}} \mathbf{A}\mathbf{u}_i^\star = \lambda_i^\star \mathbf{u}_i^{\mathrm{T}} \mathbf{u}_i^\star \tag{A.32}$$

where we have used $\mathbf{A}^\star = \mathbf{A}$ because we are considering only real matrices $\mathbf{A}$. Taking the transpose of the second of these equations and using $\mathbf{A}^{\mathrm{T}} = \mathbf{A}$, we see that the left-hand sides of the two equations are equal and hence that $\lambda_i^\star = \lambda_i$, and so λ_i must be real.

The eigenvectors $\mathbf{u}_i$ of a real symmetric matrix can be chosen to be orthonormal (i.e., orthogonal and of unit length) so that

$$\mathbf{u}_i^{\mathrm{T}} \mathbf{u}_j = I_{ij} \tag{A.33}$$

where I_{ij} are the elements of the identity matrix $\mathbf{I}$. To show this, we first left-multiply (A.29) by $\mathbf{u}_j^{\mathrm{T}}$ to give

$$\mathbf{u}_j^{\mathrm{T}} \mathbf{A}\mathbf{u}_i = \lambda_i \mathbf{u}_j^{\mathrm{T}} \mathbf{u}_i \tag{A.34}$$

and hence, by exchanging the indices, we have

$$\mathbf{u}_i^{\mathrm{T}} \mathbf{A}\mathbf{u}_j = \lambda_j \mathbf{u}_i^{\mathrm{T}} \mathbf{u}_j. \tag{A.35}$$

We now take the transpose of the second equation and make use of the symmetry property $\mathbf{A}^{\mathrm{T}} = \mathbf{A}$, and then subtract the two equations to give

$$(\lambda_i - \lambda_j)\, \mathbf{u}_i^{\mathrm{T}} \mathbf{u}_j = 0. \tag{A.36}$$

Hence, for $\lambda_i \neq \lambda_j$, we have $\mathbf{u}_i^{\mathrm{T}} \mathbf{u}_j = 0$ so that $\mathbf{u}_i$ and $\mathbf{u}_j$ are orthogonal. If the two eigenvalues are equal, then any linear combination $\alpha\mathbf{u}_i + \beta\mathbf{u}_j$ is also an eigenvector

Telegram: @uni_k

www.konkur.in

with the same eigenvalue, so we can select one linear combination arbitrarily, and then choose the second to be orthogonal to the first (it can be shown that the degenerate eigenvectors are never linearly dependent). Hence, the eigenvectors can be chosen to be orthogonal, and by normalizing can be set to unit length. Because there are M eigenvalues, the corresponding M orthogonal eigenvectors form a complete set and so any M-dimensional vector can be expressed as a linear combination of the eigenvectors.

We can take the eigenvectors $\mathbf{u}_i$ to be the columns of an $M \times M$ matrix $\mathbf{U}$, which from orthonormality satisfies

$$\mathbf{U}^{\mathrm{T}}\mathbf{U} = \mathbf{I}. \tag{A.37}$$

Such a matrix is said to be *orthogonal*. Interestingly, the rows of this matrix are also orthogonal, so that $\mathbf{U}\mathbf{U}^{\mathrm{T}} = \mathbf{I}$. To show this, note that (A.37) implies $\mathbf{U}^{\mathrm{T}}\mathbf{U}\mathbf{U}^{-1} = \mathbf{U}^{-1} = \mathbf{U}^{\mathrm{T}}$ and so $\mathbf{U}\mathbf{U}^{-1} = \mathbf{U}\mathbf{U}^{\mathrm{T}} = \mathbf{I}$. Using (A.12), it also follows that $|\mathbf{U}| = 1$.

The eigenvector equation (A.29) can be expressed in terms of $\mathbf{U}$ in the form

$$\mathbf{A}\mathbf{U} = \mathbf{U}\boldsymbol{\Lambda} \tag{A.38}$$

where $\boldsymbol{\Lambda}$ is an $M \times M$ diagonal matrix whose diagonal elements are given by the eigenvalues λ_i.

If we consider a column vector $\mathbf{x}$ that is transformed by an orthogonal matrix $\mathbf{U}$ to give a new vector

$$\widetilde{\mathbf{x}} = \mathbf{U}\mathbf{x} \tag{A.39}$$

then the length of the vector is preserved because

$$\widetilde{\mathbf{x}}^{\mathrm{T}}\widetilde{\mathbf{x}} = \mathbf{x}^{\mathrm{T}}\mathbf{U}^{\mathrm{T}}\mathbf{U}\mathbf{x} = \mathbf{x}^{\mathrm{T}}\mathbf{x} \tag{A.40}$$

and similarly the angle between any two such vectors is preserved because

$$\widetilde{\mathbf{x}}^{\mathrm{T}}\widetilde{\mathbf{y}} = \mathbf{x}^{\mathrm{T}}\mathbf{U}^{\mathrm{T}}\mathbf{U}\mathbf{y} = \mathbf{x}^{\mathrm{T}}\mathbf{y}. \tag{A.41}$$

Thus, multiplication by $\mathbf{U}$ can be interpreted as a rigid rotation of the coordinate system.

From (A.38), it follows that

$$\mathbf{U}^{\mathrm{T}}\mathbf{A}\mathbf{U} = \boldsymbol{\Lambda} \tag{A.42}$$

and because $\boldsymbol{\Lambda}$ is a diagonal matrix, we say that the matrix $\mathbf{A}$ is *diagonalized* by the matrix $\mathbf{U}$. If we left-multiply by $\mathbf{U}$ and right-multiply by $\mathbf{U}^{\mathrm{T}}$, we obtain

$$\mathbf{A} = \mathbf{U}\boldsymbol{\Lambda}\mathbf{U}^{\mathrm{T}}. \tag{A.43}$$

Taking the inverse of this equation and using (A.3) together with $\mathbf{U}^{-1} = \mathbf{U}^{\mathrm{T}}$, we have

$$\mathbf{A}^{-1} = \mathbf{U}\boldsymbol{\Lambda}^{-1}\mathbf{U}^{\mathrm{T}}. \tag{A.44}$$

Telegram: @uni_k

www.konkur.in

These last two equations can also be written in the form

$$\mathbf{A} = \sum_{i=1}^{M} \lambda_i \mathbf{u}_i \mathbf{u}_i^{\mathrm{T}} \tag{A.45}$$

$$\mathbf{A}^{-1} = \sum_{i=1}^{M} \frac{1}{\lambda_i} \mathbf{u}_i \mathbf{u}_i^{\mathrm{T}}. \tag{A.46}$$

If we take the determinant of (A.43) and use (A.12), we obtain

$$|\mathbf{A}| = \prod_{i=1}^{M} \lambda_i. \tag{A.47}$$

Similarly, taking the trace of (A.43), and using the cyclic property (A.8) of the trace operator together with $\mathbf{U}^{\mathrm{T}}\mathbf{U} = \mathbf{I}$, we have

$$\mathrm{Tr}(\mathbf{A}) = \sum_{i=1}^{M} \lambda_i. \tag{A.48}$$

We leave it as an exercise for the reader to verify (A.22) by making use of the results (A.33), (A.45), (A.46), and (A.47).

A matrix $\mathbf{A}$ is said to be *positive definite*, denoted by $\mathbf{A} \succ 0$, if $\mathbf{w}^{\mathrm{T}}\mathbf{A}\mathbf{w} > 0$ for all non-zero values of the vector $\mathbf{w}$. Equivalently, a positive definite matrix has $\lambda_i > 0$ for all of its eigenvalues (as can be seen by setting $\mathbf{w}$ to each of the eigenvectors in turn and noting that an arbitrary vector can be expanded as a linear combination of the eigenvectors). Note that having all positive elements does not necessarily mean that a matrix is that positive definite. For example, the matrix

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \tag{A.49}$$

has eigenvalues $\lambda_1 \simeq 5.37$ and $\lambda_2 \simeq -0.37$. A matrix is said to be *positive semidefinite* if $\mathbf{w}^{\mathrm{T}}\mathbf{A}\mathbf{w} \geqslant 0$ holds for all values of $\mathbf{w}$, which is denoted $\mathbf{A} \succeq 0$ and is equivalent to $\lambda_i \geqslant 0$.

The condition number of a matrix is given by

$$\mathrm{CN} = \left(\frac{\lambda_{\max}}{\lambda_{\min}} \right)^{1/2} \tag{A.50}$$

where $\lambda_{\max}$ is the largest eigenvalue and $\lambda_{\min}$ is the smallest eigenvalue.

Telegram: @uni_k

www.konkur.in

Appendix B. Calculus of Variations

We can think of a function $y(x)$ as being an operator that, for any input value x, returns an output value y. In the same way, we can define a *functional* $F[y]$ to be an operator that takes a function $y(x)$ and returns an output value F. An example of a functional is the length of a curve drawn in a two-dimensional plane in which the path of the curve is defined in terms of a function. In the context of machine learning, a widely used functional is the entropy $\mathrm{H}[x]$ for a continuous variable x because, for any choice of probability density function $p(x)$, it returns a scalar value representing the entropy of x under that density. Thus, the entropy of $p(x)$ could equally well have been written as $\mathrm{H}[p]$.

A common problem in conventional calculus is to find a value of x that maximizes (or minimizes) a function $y(x)$. Similarly, in the calculus of variations we seek a function $y(x)$ that maximizes (or minimizes) a functional $F[y]$. That is, of all possible functions $y(x)$, we wish to find the particular function for which the functional $F[y]$ is a maximum (or minimum). The calculus of variations can be used, for instance, to show that the shortest path between two points is a straight line or that the maximum entropy distribution is a Gaussian.

If we were not familiar with the rules of ordinary calculus, we could evaluate a conventional derivative $\mathrm{d}y/\,\mathrm{d}x$ by making a small change ϵ to the variable x and then expanding in powers of ϵ, so that

$$y(x+\epsilon) = y(x) + \frac{\mathrm{d}y}{\mathrm{d}x}\epsilon + \mathcal{O}(\epsilon^2) \tag{B.1}$$

and finally taking the limit $\epsilon \to 0$. Similarly, for a function of several variables $y(x_1, \ldots, x_D)$, the corresponding partial derivatives are defined by

$$y(x_1+\epsilon_1, \ldots, x_D+\epsilon_D) = y(x_1, \ldots, x_D) + \sum_{i=1}^{D} \frac{\partial y}{\partial x_i}\epsilon_i + \mathcal{O}(\epsilon^2). \tag{B.2}$$

The analogous definition of a functional derivative arises when we consider how much a functional $F[y]$ changes when we make a small change $\epsilon\eta(x)$ to the function

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024

C. M. Bishop, H. Bishop, *Deep Learning*, https://doi.org/10.1007/978-3-031-45468-4

Telegram: @uni_k

www.konkur.in

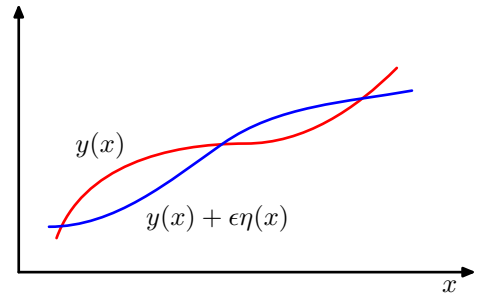


Figure B.1 A functional derivative can be defined by considering how the value of a functional $F[y]$ changes when the function $y(x)$ is changed to $y(x) + \epsilon\eta(x)$ where $\eta(x)$ is an arbitrary function of x.

$y(x)$, where $\eta(x)$ is an arbitrary function of x, as illustrated in Figure B.1. We denote the functional derivative of $F[y]$ with respect to $y(x)$ by $\delta F/\delta y(x)$ and define it by the following relation:

$$F[y(x) + \epsilon\eta(x)] = F[y(x)] + \epsilon \int \frac{\delta F}{\delta y(x)} \eta(x)\, \mathrm{d}x + \mathcal{O}(\epsilon^2). \tag{B.3}$$

This can be seen as a natural extension of (B.2) in which $F[y]$ now depends on a continuous set of variables, namely the values of y at all points x. Requiring that the functional be stationary with respect to small variations in the function $y(x)$ gives

$$\int \frac{\delta F}{\delta y(x)} \eta(x)\, \mathrm{d}x = 0. \tag{B.4}$$

Because this must hold for an arbitrary choice of $\eta(x)$, it follows that the functional derivative must vanish. To see this, imagine choosing a perturbation $\eta(x)$ that is zero everywhere except in the neighbourhood of a point $\widehat{x}$, in which case the functional derivative must be zero at $x = \widehat{x}$. However, because this must be true for every choice of $\widehat{x}$, the functional derivative must vanish for all values of x.

Consider a functional that is defined by an integral over a function $G(y, y', x)$, which depends on both $y(x)$ and its derivative $y'(x)$ and has a direct dependence on x:

$$F[y] = \int G\left(y(x), y'(x), x\right)\, \mathrm{d}x \tag{B.5}$$

where the value of $y(x)$ is assumed to be fixed at the boundary of the region of integration (which might be at infinity). If we now consider variations in the function $y(x)$, we obtain

$$F[y(x) + \epsilon\eta(x)] = F[y(x)] + \epsilon \int \left\{ \frac{\partial G}{\partial y}\eta(x) + \frac{\partial G}{\partial y'}\eta'(x) \right\} \mathrm{d}x + \mathcal{O}(\epsilon^2). \tag{B.6}$$

We now have to cast this in the form (B.3). To do so, we integrate the second term by parts and note that $\eta(x)$ must vanish at the boundary of the integral (because $y(x)$ is fixed at the boundary). This gives

$$F[y(x) + \epsilon\eta(x)] = F[y(x)] + \epsilon \int \left\{ \frac{\partial G}{\partial y} - \frac{\mathrm{d}}{\mathrm{d}x}\left(\frac{\partial G}{\partial y'}\right) \right\} \eta(x)\, \mathrm{d}x + \mathcal{O}(\epsilon^2) \tag{B.7}$$

Telegram: @uni_k

www.konkur.in

from which we can read off the functional derivative by comparison with (B.3). Requiring that the functional derivative vanishes then gives

$$\frac{\partial G}{\partial y} - \frac{\mathrm{d}}{\mathrm{d}x}\left(\frac{\partial G}{\partial y'}\right) = 0, \tag{B.8}$$

which are known as the *Euler–Lagrange* equations. For example, if

$$G = y(x)^2 + \left(y'(x)\right)^2 \tag{B.9}$$

then the Euler–Lagrange equations take the form

$$y(x) - \frac{\mathrm{d}^2 y}{\mathrm{d}x^2} = 0. \tag{B.10}$$

This second-order differential equation can be solved for $y(x)$ by making use of the boundary conditions on $y(x)$.

Often, we consider functionals defined by integrals whose integrands take the form $G(y, x)$ and that do not depend on the derivatives of $y(x)$. In this case, stationarity simply requires that $\partial G/\partial y(x) = 0$ for all values of x.

If we are optimizing a functional with respect to a probability distribution, then we need to maintain the normalization constraint on the probabilities. This is often most conveniently done using a Lagrange multiplier, which then allows an unconstrained optimization to be performed.

Appendix C

The extension of the above results to a multi-dimensional variable $\mathbf{x}$ is straightforward. For a more comprehensive discussion of the calculus of variations, see Sagan (1969).

Telegram: @uni_k

www.konkur.in

Appendix C. Lagrange Multipliers

Lagrange multipliers, also sometimes called *undetermined multipliers*, are used to find the stationary points of a function of several variables subject to one or more constraints.

Consider the problem of finding the maximum of a function $f(x_1, x_2)$ subject to a constraint relating x_1 and x_2, which we write in the form

$$g(x_1, x_2) = 0. \tag{C.1}$$

One approach would be to solve the constraint equation (C.1) and thus express x_2 as a function of x_1 in the form $x_2 = h(x_1)$. This can then be substituted into $f(x_1, x_2)$ to give a function of x_1 alone of the form $f(x_1, h(x_1))$. The maximum with respect to x_1 could then be found by differentiation in the usual way, to give the stationary value $x_1^\star$, with the corresponding value of x_2 given by $x_2^\star = h(x_1^\star)$.

One problem with this approach is that it may be difficult to find an analytic solution of the constraint equation that allows x_2 to be expressed as an explicit function of x_1. Also, this approach treats x_1 and x_2 differently and so spoils the natural symmetry between these variables.

A more elegant, and often simpler, approach introduces a parameter λ called a Lagrange multiplier. We shall motivate this technique from a geometrical perspective. Consider a D-dimensional variable $\mathbf{x}$ with components $x_1, \ldots, x_D$. The constraint equation $g(\mathbf{x}) = 0$ then represents a $(D-1)$-dimensional surface in $\mathbf{x}$-space as indicated in Figure C.1.

First note that at any point on the constraint surface, the gradient $\nabla g(\mathbf{x})$ of the constraint function is orthogonal to the surface. To see this, consider a point $\mathbf{x}$ that lies on the constraint surface along with a nearby point $\mathbf{x} + \boldsymbol{\epsilon}$ that also lies on the surface. If we make a Taylor expansion around $\mathbf{x}$, we have

$$g(\mathbf{x} + \boldsymbol{\epsilon}) \simeq g(\mathbf{x}) + \boldsymbol{\epsilon}^{\mathrm{T}} \nabla g(\mathbf{x}). \tag{C.2}$$

Because both $\mathbf{x}$ and $\mathbf{x} + \boldsymbol{\epsilon}$ lie on the constraint surface, we have $g(\mathbf{x}) = g(\mathbf{x} + \boldsymbol{\epsilon})$ and hence $\boldsymbol{\epsilon}^{\mathrm{T}} \nabla g(\mathbf{x}) \simeq 0$. In the limit $\|\boldsymbol{\epsilon}\| \to 0$, we have $\boldsymbol{\epsilon}^{\mathrm{T}} \nabla g(\mathbf{x}) = 0$, and because $\boldsymbol{\epsilon}$ is

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024
C. M. Bishop, H. Bishop, *Deep Learning*, https://doi.org/10.1007/978-3-031-45468-4

Telegram: @uni_k

www.konkur.in

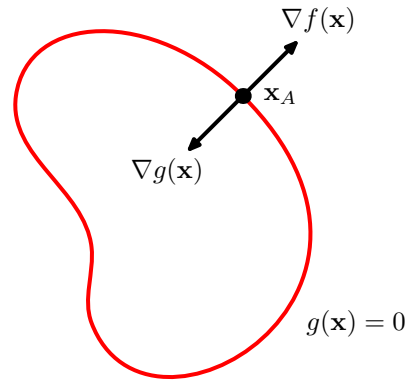


Figure C.1 A geometrical picture of the technique of Lagrange multipliers in which we seek to maximize a function $f(\mathbf{x})$, subject to the constraint $g(\mathbf{x}) = 0$. If $\mathbf{x}$ is D dimensional, the constraint $g(\mathbf{x}) = 0$ corresponds to a subspace of dimensionality $D - 1$, as indicated by the red curve. The problem can be solved by optimizing the Lagrangian function $L(\mathbf{x}, \lambda) = f(\mathbf{x}) + \lambda g(\mathbf{x})$.

then parallel to the constraint surface $g(\mathbf{x}) = 0$, we see that the vector ∇g is normal to the surface.

Next we seek a point $\mathbf{x}^\star$ on the constraint surface such that $f(\mathbf{x})$ is maximized. Such a point must have the property that the vector $\nabla f(\mathbf{x})$ is also orthogonal to the constraint surface, as illustrated in Figure C.1, because otherwise we could increase the value of $f(\mathbf{x})$ by moving a short distance along the constraint surface. Thus, ∇f and ∇g are parallel (or anti-parallel) vectors, and so there must exist a parameter λ such that

$$\nabla f + \lambda \nabla g = 0 \tag{C.3}$$

where $\lambda \neq 0$ is known as a *Lagrange multiplier*. Note that λ can have either sign.

At this point, it is convenient to introduce the *Lagrangian* function defined by

$$L(\mathbf{x}, \lambda) \equiv f(\mathbf{x}) + \lambda g(\mathbf{x}). \tag{C.4}$$

The constrained stationarity condition (C.3) is obtained by setting $\nabla_{\mathbf{x}} L = 0$. Furthermore, the condition $\partial L/\partial \lambda = 0$ leads to the constraint equation $g(\mathbf{x}) = 0$.

Thus, to find the maximum of a function $f(\mathbf{x})$ subject to the constraint $g(\mathbf{x}) = 0$, we define the Lagrangian function given by (C.4) and we then find the stationary point of $L(\mathbf{x}, \lambda)$ with respect to both $\mathbf{x}$ and λ. For a D-dimensional vector $\mathbf{x}$, this gives $D + 1$ equations that determine both the stationary point $\mathbf{x}^\star$ and the value of λ. If we are interested only in $\mathbf{x}^\star$, then we can eliminate λ from the stationarity equations without needing to find its value (hence, the term 'undetermined multiplier').

As a simple example, suppose we wish to find the stationary point of the function $f(x_1, x_2) = 1 - x_1^2 - x_2^2$ subject to the constraint $g(x_1, x_2) = x_1 + x_2 - 1 = 0$, as illustrated in Figure C.2. The corresponding Lagrangian function is given by

$$L(\mathbf{x}, \lambda) = 1 - x_1^2 - x_2^2 + \lambda(x_1 + x_2 - 1). \tag{C.5}$$

The conditions for this Lagrangian to be stationary with respect to x_1, x_2, and λ give the following coupled equations:

$$-2x_1 + \lambda = 0 \tag{C.6}$$

$$-2x_2 + \lambda = 0 \tag{C.7}$$

$$x_1 + x_2 - 1 = 0. \tag{C.8}$$

Telegram: @uni_k

www.konkur.in

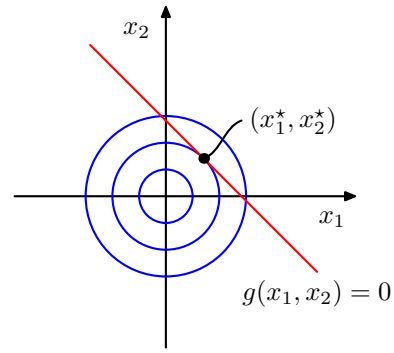


Figure C.2 A simple example of the use of Lagrange multipliers in which the aim is to maximize $f(x_1, x_2) = 1 - x_1^2 - x_2^2$ subject to the constraint $g(x_1, x_2) = 0$ where $g(x_1, x_2) = x_1 + x_2 - 1$. The circles show contours of the function $f(x_1, x_2)$, and the diagonal line shows the constraint surface $g(x_1, x_2) = 0$.

Solving these equations then gives the stationary point as $(x_1^\star, x_2^\star) = (1/2, 1/2)$, and the corresponding value for the Lagrange multiplier is $\lambda = 1$.

So far, we have considered the problem of maximizing a function subject to an *equality constraint* of the form $g(\mathbf{x}) = 0$. We now consider the problem of maximizing $f(\mathbf{x})$ subject to an *inequality constraint* of the form $g(\mathbf{x}) \geqslant 0$, as illustrated in Figure C.3.

There are now two kinds of solution possible, according to whether the constrained stationary point lies in the region where $g(\mathbf{x}) > 0$, in which case the constraint is *inactive*, or whether it lies on the boundary $g(\mathbf{x}) = 0$, in which case the constraint is said to be *active*. In the former case, the function $g(\mathbf{x})$ plays no role and so the stationary condition is simply $\nabla f(\mathbf{x}) = 0$. This again corresponds to a stationary point of the Lagrange function (C.4) but this time with $\lambda = 0$. The latter case, where the solution lies on the boundary, is analogous to the equality constraint discussed previously and corresponds to a stationary point of the Lagrange function (C.4) with $\lambda \neq 0$. Now, however, the sign of the Lagrange multiplier is crucial, because the function $f(\mathbf{x})$ is at a maximum only if its gradient is oriented away from the region $g(\mathbf{x}) > 0$, as illustrated in Figure C.3. We therefore have $\nabla f(\mathbf{x}) = -\lambda \nabla g(\mathbf{x})$ for some value of $\lambda > 0$.

For either of these two cases, the product $\lambda g(\mathbf{x}) = 0$. Thus, the solution to

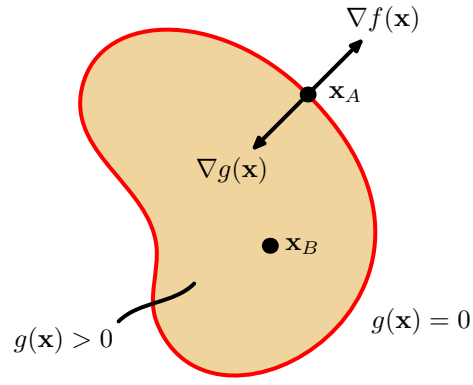


Figure C.3 Illustration of the problem of maximizing $f(\mathbf{x})$ subject to the inequality constraint $g(\mathbf{x}) \geqslant 0$.

Telegram: @uni_k

www.konkur.in

the problem of maximizing $f(\mathbf{x})$ subject to $g(\mathbf{x}) \geqslant 0$ is obtained by optimizing the Lagrange function (C.4) with respect to $\mathbf{x}$ and λ subject to the conditions

$$g(\mathbf{x}) \geqslant 0 \tag{C.9}$$

$$\lambda \geqslant 0 \tag{C.10}$$

$$\lambda g(\mathbf{x}) = 0. \tag{C.11}$$

These are known as the *Karush–Kuhn–Tucker* (KKT) conditions (Karush, 1939; Kuhn and Tucker, 1951).

Note that if we wish to minimize (rather than maximize) the function $f(\mathbf{x})$ subject to an inequality constraint $g(\mathbf{x}) \geqslant 0$, then we minimize the Lagrangian function $L(\mathbf{x}, \lambda) = f(\mathbf{x}) - \lambda g(\mathbf{x})$ with respect to $\mathbf{x}$, again subject to $\lambda \geqslant 0$.

Finally, it is straightforward to extend the technique of Lagrange multipliers to cases with multiple equality and inequality constraints. Suppose we wish to maximize $f(\mathbf{x})$ subject to $g_j(\mathbf{x}) = 0$ for $j = 1, \ldots, J$, and $h_k(\mathbf{x}) \geqslant 0$ for $k = 1, \ldots, K$. We then introduce Lagrange multipliers $\{\lambda_j\}$ and $\{\mu_k\}$, and then optimize the Lagrangian function given by

$$L(\mathbf{x}, \{\lambda_j\}, \{\mu_k\}) = f(\mathbf{x}) + \sum_{j=1}^{J} \lambda_j g_j(\mathbf{x}) + \sum_{k=1}^{K} \mu_k h_k(\mathbf{x}) \tag{C.12}$$

subject to $\mu_k \geqslant 0$ and $\mu_k h_k(\mathbf{x}) = 0$ for $k = 1, \ldots, K$. Extensions to constrained functional derivatives are similarly straightforward. For a more detailed discussion of the technique of Lagrange multipliers, see Nocedal and Wright (1999).

Appendix B

Telegram: @uni_k

www.konkur.in

Bibliography

Abramowitz, M., and I. A. Stegun. 1965. *Handbook of Mathematical Functions.* Dover.

Adler, S. L. 1981. "Over-relaxation method for the Monte Carlo evaluation of the partition function for multiquadratic actions." *Physical Review D* 23:2901–2904.

Aghajanyan, Armen, Bernie Huang, Candace Ross, Vladimir Karpukhin, Hu Xu, Naman Goyal, Dmytro Okhonko, *et al.* 2022. *CM3: A Causal Masked Multimodal Model of the Internet.* Technical report. arXiv:2201.07520.

Aghajanyan, Armen, Luke Zettlemoyer, and Sonal Gupta. 2020. *Intrinsic Dimensionality Explains the Effectiveness of Language Model Fine-Tuning.* Technical report. arXiv:2012.13255.

Ahn, J. H., and J. H. Oh. 2003. "A constrained EM algorithm for principal component analysis." *Neural Computation* 15 (1): 57–65.

Alayrac, Jean-Baptiste, Jeff Donahue, Pauline Luc, Antoine Miech, Iain Barr, Yana Hasson, Karel Lenc, *et al.* 2022. *Flamingo: a Visual Language Model for Few-Shot Learning.* Technical report. arXiv:2204.14198.

Amari, S., A. Cichocki, and H. H. Yang. 1996. "A new learning algorithm for blind signal separation." In *Advances in Neural Information Processing Systems,* edited by D. S. Touretzky, M. C. Mozer, and M. E. Hasselmo, 8:757–763. MIT Press.

Anderson, J. A., and E. Rosenfeld. 1988. *Neurocomputing: Foundations of Research.* MIT Press.

Anderson, T. W. 1963. "Asymptotic Theory for Principal Component Analysis." *Annals of Mathematical Statistics* 34:122–148.

Arjovsky, M., S. Chintala, and L. Bottou. 2017. *Wasserstein GAN.* Technical report. arXiv:1701.07875.

Attias, H. 1999. "Independent factor analysis." *Neural Computation* 11 (4): 803–851.

Austin, Jacob, Daniel D. Johnson, Jonathan Ho, Daniel Tarlow, and Rianne van den Berg. 2021. "Structured Denoising Diffusion Models in Discrete State-Spaces." In *Advances in Neural Information Processing Systems,* 34:17981–17993.

Ba, Jimmy Lei, Jamie Ryan Kiros, and Geoffrey E Hinton. 2016. *Layer Normalization.* Technical report. arXiv:1607.06450.

Bach, F. R., and M. I. Jordan. 2002. "Kernel Independent Component Analysis." *Journal of Machine Learning Research* 3:1–48.

Badrinarayanan, Vijay, Alex Kendall, and Roberto Cipolla. 2015. *SegNet: A Deep Convolu-*

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024
C. M. Bishop, H. Bishop, *Deep Learning*, https://doi.org/10.1007/978-3-031-45468-4

Telegram: @uni_k

www.konkur.in

tional Encoder-Decoder Architecture for Image Segmentation. Technical report. arXiv:1511.00561.

Bahdanau, Dzmitry, Kyunghyun Cho, and Yoshua Bengio. 2014. *Neural Machine Translation by Jointly Learning to Align and Translate.* Technical report. arXiv:1409.0473.

Baldi, P., and K. Hornik. 1989. "Neural networks and principal component analysis: learning from examples without local minima." *Neural Networks* 2 (1): 53–58.

Balduzzi, David, Marcus Frean, Lennox Leary, JP Lewis, Kurt Wan-Duo Ma, and Brian McWilliams. 2017. *The Shattered Gradients Problem: If resnets are the answer, then what is the question?* Technical report. arXiv:1702.08591.

Bartholomew, D J. 1987. *Latent Variable Models and Factor Analysis.* Charles Griffin.

Basilevsky, Alexander. 1994. *Statistical Factor Analysis and Related Methods: Theory and Applications.* Wiley.

Bather, J. 2000. *Decision Theory: An Introduction to Dynamic Programming and Sequential Decisions.* Wiley.

Battaglia, Peter W., Jessica B. Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, *et al.* 2018. *Relational inductive biases, deep learning, and graph networks.* Technical report. arXiv:1806.01261.

Baydin, A. G., B. A. Pearlmutter, A. A. Radul, and J. M. Siskind. 2018. "Automatic differentiation in machine learning: a survey." *Journal of Machine Learning Research* 18:1–43.

Becker, S., and Y. LeCun. 1989. "Improving the convergence of back-propagation learning with second order methods." In *Proceedings of the 1988 Connectionist Models Summer School,* edited by D. Touretzky, G. E. Hinton, and T. J. Sejnowski, 29–37. Morgan Kaufmann.

Belkin, Mikhail, Daniel Hsu, Siyuan Ma, and Soumik Mandal. 2019. "Reconciling modern machine-learning practice and the classical bias-variance trade-off." *Proceedings of the National Academy of Sciences* 116 (32): 15849–15854.

Bell, A. J., and T. J. Sejnowski. 1995. "An information maximization approach to blind separation and blind deconvolution." *Neural Computation* 7 (6): 1129–1159.

Bellman, R. 1961. *Adaptive Control Processes: A Guided Tour.* Princeton University Press.

Bengio, Yoshua, Aaron Courville, and Pascal Vincent. 2012. *Representation Learning: A Review and New Perspectives.* Technical report. arXiv:1206.5538.

Bengio, Yoshua, Nicholas Léonard, and Aaron Courville. 2013. *Estimating or Propagating Gradients Through Stochastic Neurons for Conditional Computation.* Technical report. arXiv:1308.3432.

Berger, J. O. 1985. *Statistical Decision Theory and Bayesian Analysis.* Second. Springer.

Bernardo, J. M., and A. F. M. Smith. 1994. *Bayesian Theory.* Wiley.

Bishop, C. M. 1995a. "Regularization and Complexity Control in Feed-forward Networks." In *Proceedings International Conference on Artificial Neural Networks ICANN'95,* edited by F. Fougelman-Soulie and P. Gallinari, 1:141–148. EC2 et Cie.

Bishop, Christopher M. 1992. "Exact Calculation of the Hessian Matrix for the Multilayer Perceptron." *Neural Computation* 4 (4): 494–501.

Bishop, Christopher M. 1994. "Novelty Detection and Neural Network Validation." *IEE Proceedings: Vision, Image and Signal Processing* 141 (4): 217–222.

Bishop, Christopher M. 1995b. *Neural Networks for Pattern Recognition.* Oxford University Press.

Bishop, Christopher M. 1995c. "Training with noise is equivalent to Tikhonov regularization." *Neural Computation* 7 (1): 108–116.

Bishop, Christopher M. 2006. *Pattern Recognition and Machine Learning.* Springer.

Telegram: @uni_k

www.konkur.in

Bommasani, Rishi, Drew A. Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S. Bernstein, *et al.* 2021. *On the Opportunities and Risks of Foundation Models.* Technical report. arXiv:2108.07258.

Bottou, L. 2010. "Large-scale machine learning with stochastic gradient descent." In *Proceedings COMPSTAT 2010,* 177–186. Springer.

Bourlard, H., and Y. Kamp. 1988. "Auto-association by multilayer perceptrons and singular value decomposition." *Biological Cybernetics* 59:291–294.

Breiman, L. 1996. "Bagging predictors." *Machine Learning* 26:123–140.

Brinker, T. J., A. Hekler, A. H. Enk, C. Berking, S Haferkamp, A. Hauschild, M. Weichenthal, *et al.* 2019. "Deep neural networks are superior to dermatologists in melanoma image classification." *European Journal of Cancer* 119:11–17.

Brock, Andrew, Jeff Donahue, and Karen Simonyan. 2018. "Large-Scale GAN Training for High Fidelity Natural Image Synthesis." In *Proceedings of the International Conference Learning Representations (ICLR).* ArXiv:1809.11096.

Bronstein, Michael M., Joan Bruna, Taco Cohen, and Petar Velickovic. 2021. *Geometric Deep Learning: Grids, Groups, Graphs, Geodesics, and Gauges.* Technical report. arXiv:2104.13478.

Bronstein, Michael M., Joan Bruna, Yann LeCun, Arthur Szlam, and Pierre Vandergheynst. 2017. "Geometric Deep Learning: Going Beyond Eulcidean Data." In *IEEE Signal Processing Magazine,* vol. 34. 4. IEEE, July.

Broomhead, D. S., and D. Lowe. 1988. "Multivariable functional interpolation and adaptive networks." *Complex Systems* 2:321–355.

Brown, Tom B., Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, *et al.* 2020. *Language Models are Few-Shot Learners.* Technical report. arXiv:2005.14165.

Bubeck, Sébastien, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, *et al.* 2023. *Sparks of Artificial General Intelligence: Early experiments with GPT-4.* Technical report. arXiv:2303.12712.

Cardoso, J-F. 1998. "Blind signal separation: statistical principles." *Proceedings of the IEEE* 9 (10): 2009–2025.

Caruana, R. 1997. "Multitask learning." *Machine Learning* 28:41–75.

Casella, G., and R. L. Berger. 2002. *Statistical Inference.* Second. Duxbury.

Chan, K., T. Lee, and T. J. Sejnowski. 2003. "Variational Bayesian learning of ICA with missing data." *Neural Computation* 15 (8): 1991–2011.

Chen, A. M., H. Lu, and R. Hecht-Nielsen. 1993. "On the geometry of feedforward neural network error surfaces." *Neural Computation* 5 (6): 910–927.

Chen, Mark, Alec Radford, Rewon Child, Jeffrey Wu, Heewoo Jun, David Luan, and Ilya Sutskever. 2020. "Generative Pretraining From Pixels." *Proceedings of Machine Learning Research* 119:1691–1703.

Chen, R. T. Q., Rubanova Y, J. Bettencourt, and D. Duvenaud. 2018. *Neural Ordinary Differential Equations.* Technical report. arXiv:1806.07366.

Chen, Ting, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. 2020. *A Simple Framework for Contrastive Learning of Visual Representations.* Technical report. arXiv:2002.05709.

Cho, Kyunghyun, Bart van Merrienboer, Çaglar Gülçehre, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. *Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translations.* Technical report. arXiv:1406.1078.

Choudrey, R. A., and S. J. Roberts. 2003. "Variational mixture of Bayesian independent component analyzers." *Neural Computation* 15 (1): 213–252.

Telegram: @uni_k

www.konkur.in

Christiano, Paul, Jan Leike, Tom B. Brown, Miljan Martic, Shane Legg, and Dario Amodei. 2017. *Deep reinforcement learning from human preferences.* Technical report. arXiv:1706.03741.

Collobert, R. 2004. "Large Scale Machine Learning." PhD diss., Université Paris VI.

Comon, P., C. Jutten, and J. Herault. 1991. "Blind source separation, 2: problems statement." *Signal Processing* 24 (1): 11–20.

Cover, T., and P. Hart. 1967. "Nearest neighbor pattern classification." *IEEE Transactions on Information Theory* IT-11:21–27.

Cover, T. M., and J. A. Thomas. 1991. *Elements of Information Theory.* Wiley.

Cox, R. T. 1946. "Probability, frequency and reasonable expectation." *American Journal of Physics* 14 (1): 1–13.

Cybenko, G. 1989. "Approximation by superpositions of a sigmoidal function." *Mathematics of Control, Signals and Systems* 2:304–314.

Dawid, A. P. 1979. "Conditional Independence in Statistical Theory (with discussion)." *Journal of the Royal Statistical Society, Series B* 4:1–31.

Dawid, A. P. 1980. "Conditional Independence for Statistical Operations." *Annals of Statistics* 8:598–617.

Deisenroth, M. P., A. A. Faisal, and C. S. Ong. 2020. *Mathematics for Machine Learning.* Cambridge University Press.

Dempster, A. P., N. M. Laird, and D. B. Rubin. 1977. "Maximum likelihood from incomplete data via the EM algorithm." *Journal of the Royal Statistical Society, B* 39 (1): 1–38.

Deng, Jia, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. "ImageNet: A large-scale hierarchical image database." In *IEEE Conference on Computer Vision and Pattern Recognition.*

Devlin, Jacob, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding.* Technical report. arXiv:1810.04805.

Dhariwal, Prafulla, and Alex Nichol. 2021. *Diffusion Models Beat GANs on Image Synthesis.* Technical report. arXiv:2105.05233.

Dinh, Laurent, David Krueger, and Yoshua Bengio. 2014. *NICE: Non-linear Independent Components Estimation.* Technical report. arXiv:1410.8516.

Dinh, Laurent, Jascha Sohl-Dickstein, and Samy Bengio. 2016. *Density estimation using Real NVP.* Technical report. arXiv:1605.08803.

Dodge, Samuel, and Lina Karam. 2017. *A Study and Comparison of Human and Deep Learning Recognition Performance Under Visual Distortions.* Technical report. arXiv:1705.02498.

Doersch, C. 2016. *Tutorial on Variational Autoencoders.* Technical report. arXiv:1606.05908.

Dosovitskiy, Alexey, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, *et al.* 2020. *An Image is Worth* 16×16 *Words: Transformers for Image Recognition at Scale.* Technical report. arXiv:2010.11929.

Duane, S., A. D. Kennedy, B. J. Pendleton, and D. Roweth. 1987. "Hybrid Monte Carlo." *Physics Letters B* 195 (2): 216–222.

Duchi, J., E. Hazan, and Y. Singer. 2011. "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization." *Journal of Machine Learning Research* 12:2121–2159.

Duda, R. O., and P. E. Hart. 1973. *Pattern Classification and Scene Analysis.* Wiley.

Dufter, Philipp, Martin Schmitt, and Hinrich Schütze. 2021. *Position Information in Transformers: An Overview.* Technical report. arXiv:2102.11090.

Dumoulin, Vincent, and Francesco Visin. 2016. *A guide to convolution arithmetic for deep learning.* Technical report. arXiv:1603.07285.

Elliott, R. J., L. Aggoun, and J. B. Moore. 1995. *Hidden Markov Models: Estimation and Control.* Springer.

Telegram: @uni_k

www.konkur.in

Esser, Patrick, Robin Rombach, and Björn Ommer. 2020. *Taming Transformers for High-Resolution Image Synthesis.* Technical report. arXiv:2012.09841.

Esteva, A., B. Kuprel, R. A. Novoa, J. Ko, S. M. Swetter, H. M. Blau, and S. Thrun. 2017. "Dermatologist-level classification of skin cancer with deep neural networks." *Nature* 542:115–118.

Everitt, B. S. 1984. *An Introduction to Latent Variable Models.* Chapman / Hall.

Eykholt, Kevin, Ivan Evtimov, Earlence Fernandes, Bo Li, Amir Rahmati, Chaowei Xiao, Atul Prakash, Tadayoshi Kohno, and Dawn Song. 2018. "Robust Physical-World Attacks on Deep Learning Visual Classification." In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR).*

Fawcett, T. 2006. "An introduction to ROC analysis." *Pattern Recognition Letters* 27:861–874.

Feller, W. 1966. *An Introduction to Probability Theory and its Applications.* Second. Vol. 2. Wiley.

Fletcher, R. 1987. *Practical Methods of Optimization.* Second. Wiley.

Forsyth, D. A., and J. Ponce. 2003. *Computer Vision: A Modern Approach.* Prentice Hall.

Freund, Y., and R. E. Schapire. 1996. "Experiments with a new boosting algorithm." In *Thirteenth International Conference on Machine Learning,* edited by L. Saitta, 148–156. Morgan Kaufmann.

Fukushima, K. 1980. "Neocognitron: A Self-organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position." *Biological Cybernetics* 36:193–202.

Funahashi, K. 1989. "On the approximate realization of continuous mappings by neural networks." *Neural Networks* 2 (3): 183–192.

Fung, R., and K. C. Chang. 1990. "Weighting and Integrating Evidence for Stochastic Simulation in Bayesian Networks." In *Uncertainty in Artificial Intelligence,* edited by P. P. Bonissone, M. Henrion, L. N. Kanal, and J. F. Lemmer, 5:208–219. Elsevier.

Gatys, Leon A., Alexander S. Ecker, and Matthias Bethge. 2015. *A Neural Algorithm of Artistic Style.* Technical report. arXiv:1508.06576.

Geman, S., and D. Geman. 1984. "Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images." *IEEE PAMI* 6 (1): 721–741.

Gemmeke, Jort F., Daniel P. W. Ellis, Dylan Freedman, Aren Jansen, Wade Lawrence, R. Channing Moore, Manoj Plakal, and Marvin Ritter. 2017. "Audio Set: An ontology and human-labeled dataset for audio events." In *Proc. IEEE ICASSP 2017.* New Orleans, LA.

Germain, Mathieu, Karol Gregor, Iain Murray, and Hugo Larochelle. 2015. *MADE: Masked Autoencoder for Distribution Estimation.* Technical report. arXiv:1502.03509.

Gilks, W. R. 1992. "Derivative-free adaptive rejection sampling for Gibbs sampling." In *Bayesian Statistics,* edited by J. Bernardo, J. Berger, A. P. Dawid, and A. F. M. Smith, vol. 4. Oxford University Press.

Gilks, W. R., N. G. Best, and K. K. C. Tan. 1995. "Adaptive rejection Metropolis sampling." *Applied Statistics* 44:455–472.

Gilks, W. R., S. Richardson, and D. J. Spiegelhalter. 1996. *Markov Chain Monte Carlo in Practice.* Chapman / Hall.

Gilks, W. R., and P. Wild. 1992. "Adaptive rejection sampling for Gibbs sampling." *Applied Statistics* 41:337–348.

Gilmer, Justin, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. 2017. *Neural Message Passing for Quantum Chemistry.* Technical report. arXiv:1704.01212.

Girshick, Ross B. 2015. *Fast R-CNN.* Technical report. arXiv:1504.08083.

Golub, G. H., and C. F. Van Loan. 1996. *Matrix Computations.* Third. John Hopkins University Press.

Telegram: @uni_k

www.konkur.in

Gong, Yuan, Yu-An Chung, and James R. Glass. 2021. *AST: Audio Spectrogram Transformer.* Technical report. arXiv:2104.01778.

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press.

Goodfellow, Ian J., Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. *Generative Adversarial Networks.* Technical report. arXiv:1406.2661.

Goodfellow, Ian J., Jonathon Shlens, and Christian Szegedy. 2014. *Explaining and Harnessing Adversarial Examples.* Technical report. arXiv:1412.6572.

Grathwohl, Will, Ricky T. Q. Chen, Jesse Bettencourt, Ilya Sutskever, and David Duvenaud. 2018. *FFJORD: Free-form Continuous Dynamics for Scalable Reversible Generative Models.* Technical report. arXiv:1810.01367.

Griewank, A., and A Walther. 2008. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation.* Second. SIAM.

Grosse, R. 2018. *Automatic Differentiation.* CSC321 Lecture 10. University of Toronto.

Gulrajani, I., F. Ahmed, M. Arjovsky, V. Dumoulin, and A. Courville. 2017. *Improved training of Wasserstein GANs.* Technical report. arXiv:1704.00028.

Gutmann, Michael, and Aapo Hyvärinen. 2010. "Noise-contrastive estimation: A new estimation principle for unnormalized statistical models." *Journal of Machine Learning Research* 9:297–304.

Hamilton, W. L. 2020. *Graph Representation Learning*. Morgan / Claypool.

Hartley, R., and A. Zisserman. 2004. *Multiple View Geometry in Computer Vision.* Second. Cambridge University Press.

Hassibi, B., and D. G. Stork. 1993. "Second order derivatives for network pruning: optimal brain surgeon." In *Proceedings International Conference on Neural Information Processing Systems (NeurIPS),* edited by S. J. Hanson, J. D. Cowan, and C. L. Giles, 5:164–171. Morgan Kaufmann.

Hastie, T., R. Tibshirani, and J. Friedman. 2009. *The Elements of Statistical Learning*. Second. Springer.

Hastings, W. K. 1970. "Monte Carlo sampling methods using Markov chains and their applications." *Biometrika* 57:97–109.

He, Kaiming, Xinlei Chen, Saining Xie, Yanghao Li, Piotr Dollár, and Ross B. Girshick. 2021. *Masked Autoencoders Are Scalable Vision Learners.* Technical report. arXiv:2111.06377.

He, Kaiming, Haoqi Fan, Yuxin Wu, Saining Xie, and Ross Girshick. 2019. *Momentum Contrast for Unsupervised Visual Representation Learning.* Technical report. arXiv:1911.05722.

He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015a. *Deep Residual Learning for Image Recognition.* Technical report. arXiv:1512.03385.

He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015b. *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification.* Technical report. arXiv:1502.01852.

Henrion, M. 1988. "Propagation of Uncertainty by Logic Sampling in Bayes' Networks." In *Uncertainty in Artificial Intelligence,* edited by J. F. Lemmer and L. N. Kanal, 2:149–164. North Holland.

Higgins, I., L. Matthey, A. Pal, C. Burgess, X. Glorot, M. Botvinik, S. Mohamed, and A. Lerchner. 2017. "β-VAE: learning basic visual concepts with a constrained variational framework." In *Proceedings of the International Conference Learning Representations (ICLR).*

Hinton, G. E. 2012. *Neural Networks for Machine Learning. Lecture 6.5. Coursera Lectures.*

Hinton, G. E., M. Welling, Y. W. Teh, and S Osindero. 2001. "A new view of ICA." In *Proceedings of the International Conference on Independent Component Analysis and Blind Signal Separation,* vol. 3.

Telegram: @uni_k

www.konkur.in

Hinton, Geoffrey, Oriol Vinyals, and Jeff Dean. 2015. *Distilling the Knowledge in a Neural Network.* Technical report. arXiv:1503.02531.

Hinton, Geoffrey E. 2002. "Training products of experts by minimizing contrastive divergence." *Neural Computation* 14:1771–1800.

Ho, Jonathan, Ajay Jain, and Pieter Abbeel. 2020. *Denoising Diffusion Probabilistic Models.* Technical report. arXiv:2006.11239.

Ho, Jonathan, Chitwan Saharia, William Chan, David J. Fleet, Mohammad Norouzi, and Tim Salimans. 2021. *Cascaded Diffusion Models for High Fidelity Image Generation.* Technical report. arXiv:2106.15282.

Hochreiter, S., and J. Schmidhuber. 1997. "Long short-term Memory." *Neural Computation* 9 (8): 1735–1780.

Hojen-Sorensen, P. A., O. Winther, and L. K. Hansen. 2002. "Mean field approaches to independent component analysis." *Neural Computation* 14 (4): 889–918.

Holtzman, Ari, Jan Buys, Maxwell Forbes, and Yejin Choi. 2019. *The Curious Case of Neural Text Degeneration.* Technical report. arXiv:1904.09751.

Hornik, K., M. Stinchcombe, and H. White. 1989. "Multilayer feedforward networks are universal approximators." *Neural Networks* 2 (5): 359–366.

Hospedales, Timothy, Antreas Antoniou, Paul Micaelli, and Amos Storkey. 2021. "Meta-learning in neural networks: A survey." *IEEE Transactions on Pattern Analysis and Machine Intelligence* 44 (9): 5149–5169.

Hotelling, H. 1933. "Analysis of a complex of statistical variables into principal components." *Journal of Educational Psychology* 24:417–441.

Hotelling, H. 1936. "Relations between two sets of variables." *Biometrika* 28:321–377.

Hu, Anthony, Lloyd Russell, Hudson Yeo, Zak Murez, George Fedoseev, Alex Kendall, Jamie Shotton, and Gianluca Corrado. 2023. *GAIA-1: A Generative World Model for Autonomous Driving.* Technical report. arXiv:2309.17080.

Hu, Edward J., Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. *LoRA: Low-Rank Adaptation of Large Language Models.* Technical report. arXiv:2106.09685.

Hubel, D. H., and T. N. Wiesel. 1959. "Receptive fields of single neurons in the cat's striate cortex." *Journal of Physiology* 148:574–591.

Hyvärinen, A. 2005. "Estimation of Non-Normalized Statistical Models by Score Matching." *Journal of Machine Learning Research* 6:695–709.

Hyvärinen, A., and E. Oja. 1997. "A fast fixed-point algorithm for independent component analysis." *Neural Computation* 9 (7): 1483–1492.

Hyvärinen, Aapo, Jarmo Hurri, and Patrick O. Hoyer. 2009. *Natural Image Statistics: A Probabilistic Approach to Early Computational Vision.* Springer.

Ioffe, S., and C. Szegedy. 2015. "Batch normalization." In *Proceedings of the International Conference on Machine Learning (ICML),* 448–456.

Jacobs, R. A., M. I. Jordan, S. J. Nowlan, and G. E. Hinton. 1991. "Adaptive mixtures of local experts." *Neural Computation* 3 (1): 79–87.

Jebara, T. 2004. *Machine Learning: Discriminative and Generative.* Kluwer.

Jensen, C., A. Kong, and U. Kjaerulff. 1995. "Blocking Gibbs sampling in very large probabilistic expert systems." *International Journal of Human Computer Studies. Special Issue on Real-World Applications of Uncertain Reasoning.* 42:647–666.

Jolliffe, I. T. 2002. *Principal Component Analysis.* Second. Springer.

Jumper, John, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, and Olaf Ronneberger. 2021. "Highly accurate protein structure prediction with AlphaFold." *Nature* 596:583–589.

Telegram: @uni_k

www.konkur.in

Jutten, C., and J. Herault. 1991. "Blind separation of sources, 1: An adaptive algorithm based on neuromimetic architecture." *Signal Processing* 24 (1): 1–10.

Kaplan, Jared, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. *Scaling Laws for Neural Language Models.* Technical report. arXiv:2001.08361.

Karras, Tero, Timo Aila, Samuli Laine, and Jaakko Lehtinen. 2017. *Progressive Growing of GANs for Improved Quality, Stability, and Variation.* Technical report. arXiv:1710.10196.

Karush, W. 1939. "Minima of functions of several variables with inequalities as side constraints." Master's thesis, Department of Mathematics, University of Chicago.

Khosla, Prannay, Piotr Teterwak, Chen Wang, Aaron Sarna, Yonglong Tian, Phillip Isola, Aaron Maschinot, Ce Liu, and Dilip Krishnan. 2020. *Supervised Contrastive Learning.* Technical report. arXiv:2004.11362.

Kingma, D., and J. Ba. 2014. *Adam: A method for stochastic optimization.* Technical report. arXiv:1412.6980.

Kingma, D. P., and M. Welling. 2013. "Auto-encoding variational Bayes." In *Proceedings of the International Conference on Machine Learning (ICML).* ArXiv:1312.6114.

Kingma, Diederik P., and Max Welling. 2019. *An Introduction to Variational Autoencoders.* Technical report. arXiv:1906.02691.

Kingma, Durk P, Tim Salimans, Rafal Jozefowicz, Xi Chen, Ilya Sutskever, and Max Welling. 2016. "Improved variational inference with inverse autoregressive flow." *Advances in Neural Information Processing Systems* 29.

Kipf, Thomas N., and Max Welling. 2016. *Semi-Supervised Classification with Graph Convolutional Networks.* Technical report. arXiv:1609.02907.

Kloeden, Peter E, and Eckhard Platen. 2013. *Numerical solution of stochastic differential equations.* Vol. 23. Stochastic Modelling and Applied Probability. Springer.

Kobyzev, I., S. J. D. Prince, and M. A. Brubaker. 2019. "Normalizing flows: an introduction and review of current methods." *IEEE Transactions on Pattern Analysis and Machine Intelligence* 43 (11): 3964–3979.

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. 2012. "Imagenet classification with deep convolutional neural networks." In *Advances in Neural Information Processing Systems,* vol. 25.

Kuhn, H. W., and A. W. Tucker. 1951. "Nonlinear programming." In *Proceedings of the 2nd Berkeley Symposium on Mathematical Statistics and Probabilities,* 481–492. University of California Press.

Kullback, S., and R. A. Leibler. 1951. "On information and sufficiency." *Annals of Mathematical Statistics* 22 (1): 79–86.

Kurková, V., and P. C. Kainen. 1994. "Functionally Equivalent Feed-forward Neural Networks." *Neural Computation* 6 (3): 543–558.

Lasserre, J., Christopher M. Bishop, and T. Minka. 2006. "Principled hybrids of generative and discriminative models." In *Proceedings 2006 IEEE Conference on Computer Vision and Pattern Recognition, New York.*

Lauritzen, S. L. 1996. *Graphical Models.* Oxford University Press.

Lawley, D. N. 1953. "A Modified Method of Estimation in Factor Analysis and Some Large Sample Results." In *Uppsala Symposium on Psychological Factor Analysis,* 35–42. Number 3 in Nordisk Psykologi Monograph Series. Uppsala: Almqvist / Wiksell.

Lazarsfeld, P. F., and N. W. Henry. 1968. *Latent Structure Analysis.* Houghton Mifflin.

LeCun, Y., B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. 1989. "Backpropagation Applied to Handwritten ZIP Code Recognition." *Neural Computation* 1 (4): 541–551.

Telegram: @uni_k

www.konkur.in

LeCun, Y., L. Bottou, Y. Bengio, and P. Haffner. 1998. "Gradient-Based Learning Applied to Document Recognition." *Proceedings of the IEEE* 86:2278–2324.

LeCun, Y., J. S. Denker, and S. A. Solla. 1990. "Optimal Brain Damage." In *Proceedings International Conference on Neural Information Processing Systems (NeurIPS),* edited by D. S. Touretzky, 2:598–605. Morgan Kaufmann.

LeCun, Yann, Yoshua Bengio, and Geoffrey Hinton. 2015. "Deep Learning." *Nature* 512:436–444.

LeCun, Yann, Sumit Chopra, Raia Hadsell, Marc'Aurelio Ranzato, and Fu-Jie Huang. 2006. "A Tutorial on Energy-Based Learning." In *Predicting Structured Data,* edited by G. Bakir, T. Hofman, B. Schölkopf, A. Smola, and B. Taskar. MIT Press.

Leen, T. K. 1995. "From data distributions to regularization in invariant learning." *Neural Computation* 7:974–981.

Leshno, M., V. Y. Lin, A. Pinkus, and S. Schocken. 1993. "Multilayer feedforward networks with a polynomial activation function can approximate any function." *Neural Networks* 6:861–867.

Li, Hao, Zheng Xu, Gavin Taylor, Christoph Studer, and Tom Goldstein. 2017. *Visualizing the Loss Landscape of Neural Nets.* Technical report. arXiv:1712.09913.

Li, Junnan, Dongxu Li, Caiming Xiong, and Steven Hoi. 2022. *BLIP: Bootstrapping Language-Image Pre-training for Unified Vision-Language Understanding and Generation.* Technical report. arXiv:2201.12086.

Lin, Min, Qiang Chen, and Shuicheng Yan. 2013. *Network in Network.* Technical report. arXiv:1312.4400.

Lin, Tianyang, Yuxin Wang, Xiangyang Liu, and Xipeng Qiu. 2021. *A Survey of Transformers.* Technical report. arXiv:2106.04554.

Lipman, Yaron, Ricky T. Q. Chen, Heli Ben-Hamu, Maximilian Nickel, and Matt Le. 2022. *Flow Matching for Generative Modeling.* Technical report arXiv:2210.02747. https://arxiv.org/.

Liu, Pengfei, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2021. *Pre-train, Prompt, and Predict: A Systematic Survey of Prompting Methods in Natural Language Processing.* Technical report. arXiv:2107.13586.

Lloyd, S. P. 1982. "Least squares quantization in PCM." *IEEE Transactions on Information Theory* 28 (2): 129–137.

Long, Jonathan, Evan Shelhamer, and Trevor Darrell. 2014. *Fully Convolutional Networks for Semantic Segmentation.* Technical report. arXiv:1411.4038.

Luo, Calvin. 2022. *Understanding Diffusion Models: A Unified Perspective.* Technical report. arXiv:2208.11970.

Lütkepohl, H. 1996. *Handbook of Matrices.* Wiley.

MacKay, D. J. C. 1992. "A Practical Bayesian Framework for Back-propagation Networks." *Neural Computation* 4 (3): 448–472.

MacKay, D. J. C. 2003. *Information Theory, Inference and Learning Algorithms.* Cambridge University Press.

MacQueen, J. 1967. "Some methods for classification and analysis of multivariate observations." In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability,* edited by L. M. LeCam and J. Neyman, I:281–297. University of California Press.

Magnus, J. R., and H. Neudecker. 1999. *Matrix Differential Calculus with Applications in Statistics and Econometrics.* Wiley.

Mallat, S. 1999. *A Wavelet Tour of Signal Processing.* Second. Academic Press.

Mao, X., Q. Li, H. Xie, R. Lau, Z. Wang, and S. Smolley. 2016. *Least Squares Generative Adversarial Networks.* Technical report. arXiv:1611.04076.

Mardia, K. V., and P. E. Jupp. 2000. *Directional Statistics.* Wiley.

Martens, James, Ilya Sutskever, and Kevin Swersky. 2012. "Estimating the Hessian by Back-propagating Curvature." In *Proceedings of the*

Telegram: @uni_k

www.konkur.in

International Conference on Machine Learning (ICML). ArXiv:1206.6464.

McCullagh, P., and J. A. Nelder. 1989. *Generalized Linear Models.* Second. Chapman / Hall.

McCulloch, W. S., and W. Pitts. 1943. "A Logical Calculus of the Ideas Immanent in Nervous Activity." Reprinted in Anderson and Rosenfeld (1988), *Bulletin of Mathematical Biophysics* 5:115–133.

McLachlan, G. J., and T. Krishnan. 1997. *The EM Algorithm and its Extensions.* Wiley.

McLachlan, G. J., and D. Peel. 2000. *Finite Mixture Models.* Wiley.

Meng, X. L., and D. B. Rubin. 1993. "Maximum likelihood estimation via the ECM algorithm: a general framework." *Biometrika* 80:267–278.

Mescheder, L., A. Geiger, and S. Nowozin. 2018. *Which Training Methods for GANs do actually Converge?* Technical report. arXiv:1801.04406.

Metropolis, N., A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller. 1953. "Equation of State Calculations by Fast Computing Machines." *Journal of Chemical Physics* 21 (6): 1087–1092.

Metropolis, N., and S. Ulam. 1949. "The Monte Carlo method." *Journal of the American Statistical Association* 44 (247): 335–341.

Mikolov, Tomas, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. *Efficient Estimation of Word Representations in Vector Space.* Technical report. arXiv:1301.3781.

Minsky, M. L., and S. A. Papert. 1969. *Perceptrons.* Expanded edition 1990. MIT Press.

Mirza, M., and S. Osindero. 2014. *Conditional Generative Adversarial Nets.* Technical report. arXiv:1411.1784.

Miskin, J. W., and D. J. C. MacKay. 2001. "Ensemble learning for blind source separation." In *Independent Component Analysis: Principles and Practice,* edited by S. J. Roberts and R. M. Everson. Cambridge University Press.

Møller, M. 1993. "Efficient Training of Feed-Forward Neural Networks." PhD diss., Aarhus University, Denmark.

Montúfar, G. F., R. Pascanu, K. Cho, and Y. Bengio. 2014. "On the number of linear regions of deep neural networks." In *Proceedings of the International Conference on Neural Information Processing Systems (NeurIPS).* ArXiv:1402.1869.

Mordvintsev, Alexander, Christopher Olah, and Mike Tyka. 2015. *Inceptionism: Going Deeper into Neural Networks.* Google AI blog.

Murphy, Kevin P. 2022. *Probabilistic Machine Learning: An introduction.* MIT Press. probml.ai.

Murphy, Kevin P. 2023. *Probabilistic Machine Learning: Advanced Topics.* MIT Press. http://probml.github.io/book2.

Nakkiran, Preetum, Gal Kaplun, Yamini Bansal, Tristan Yang, Boaz Barak, and Ilya Sutskever. 2019. *Deep Double Descent: Where Bigger Models and More Data Hurt.* Technical report. arXiv:1912.02292.

Neal, R. M. 1993. *Probabilistic inference using Markov chain Monte Carlo methods.* Technical report CRG-TR-93-1. Department of Computer Science, University of Toronto, Canada.

Neal, R. M. 1999. "Suppressing random walks in Markov chain Monte Carlo using ordered overrelaxation." In *Learning in Graphical Models,* edited by Michael I. Jordan, 205–228. MIT Press.

Neal, R. M., and G. E. Hinton. 1999. "A new view of the EM algorithm that justifies incremental and other variants." In *Learning in Graphical Models,* edited by M. I. Jordan, 355–368. MIT Press.

Nelder, J. A., and R. W. M. Wedderburn. 1972. "Generalized linear models." *Journal of the Royal Statistical Society, A* 135:370–384.

Nesterov, Y. 2004. *Introductory Lectures on Convex Optimization: A Basic Course.* Kluwer.

Telegram: @uni_k

www.konkur.in

Nichol, Alex, and Prafulla Dhariwal. 2021. *Improved Denoising Diffusion Probabilistic Models.* Technical report. arXiv:2102.09672.

Nichol, Alex, Prafulla Dhariwal, Aditya Ramesh, Pranav Shyam, Pamela Mishkin, Bob McGrew, Ilya Sutskever, and Mark Chen. 2021. *GLIDE: Towards Photorealistic Image Generation and Editing with Text-Guided Diffusion Models.* Technical report. arXiv:2112.10741.

Nocedal, J., and S. J. Wright. 1999. *Numerical Optimization.* Springer.

Noh, Hyeonwoo, Seunghoon Hong, and Bohyung Han. 2015. *Learning Deconvolution Network for Semantic Segmentation.* Technical report. arXiv:1505.04366.

Nowlan, S. J., and G. E. Hinton. 1992. "Simplifying neural networks by soft weight sharing." *Neural Computation* 4 (4): 473–493.

Ogden, R. T. 1997. *Essential Wavelets for Statistical Applications and Data Analysis.* Birkhäuser.

Oord, Aaron van den, Nal Kalchbrenner, and Koray Kavukcuoglu. 2016. *Pixel Recurrent Neural Networks.* Technical report. arXiv:1601.06759.

Oord, Aaron van den, Nal Kalchbrenner, Oriol Vinyals, Lasse Espeholt, Alex Graves, and Koray Kavukcuoglu. 2016. *Conditional Image Generation with PixelCNN Decoders.* Technical report. arXiv:1606.05328.

Oord, Aaron van den, Yazhe Li, and Oriol Vinyals. 2018. *Representation Learning with Contrastive Predictive Coding.* Technical report. arXiv:1807.03748.

Oord, Aaron van den, Oriol Vinyals, and Koray Kavukcuoglu. 2017. *Neural Discrete Representation Learning.* Technical report. arXiv:1711.00937.

OpenAI. 2023. *GPT-4 Technical Report.* Technical report. arXiv:2303.08774.

Opper, M., and O. Winther. 2000. "Gaussian processes and SVM: mean field theory and leave-one-out." In *Advances in Large Margin Classifiers,* edited by A. J. Smola, P. L. Bartlett, B. Schölkopf, and D. Shuurmans, 311–326. MIT Press.

Papamakarios, G., T. Pavlakou, and Iain Murray. 2017. "Masked Autoregressive Flow for Density Estimation." In *Proceedings of the International Conference on Neural Information Processing Systems (NeurIPS),* vol. 30.

Papamakarios, George, Eric Nalisnick, Danilo Jimenez Rezende, Shakir Mohamed, and Balaji Lakshminarayanan. 2019. *Normalizing Flows for Probabilistic Modeling and Inference.* Technical report. arXiv:1912.02762.

Parisi, Giorgio. 1981. "Correlation functions and computer simulations." *Nuclear Physics B* 180:378–384.

Pearl, J. 1988. *Probabilistic Reasoning in Intelligent Systems.* Morgan Kaufmann.

Pearlmutter, B. A. 1994. "Fast exact multiplication by the Hessian." *Neural Computation* 6 (1): 147–160.

Pearlmutter, B. A., and L. C. Parra. 1997. "Maximum likelihood source separation: a context-sensitive generalization of ICA." In *Advances in Neural Information Processing Systems,* edited by M. C. Mozer, M. I. Jordan, and T. Petsche, 9:613–619. MIT Press.

Pearson, Karl. 1901. "On lines and planes of closest fit to systems of points in space." *The London, Edinburgh and Dublin Philosophical Magazine and Journal of Science, Sixth Series* 2:559–572.

Phuong, Mary, and Marcus Hutter. 2022. *Formal Algorithms for Transformers.* Technical report. arXiv:2207.09238.

Prince, Simon J.D. 2020. *Variational autoencoders.* Https://www.borealisai.com/research-blogs/tutorial-5-variational-auto-encoders.

Prince, Simon J.D. 2023. *Understanding Deep Learning.* MIT Press. http://udlbook.com.

Radford, A., L. Metz, and S. Chintala. 2015. *Unsupervised representation learning with deep convolutional generative adversarial networks.* Technical report. arXiv:1511.06434.

Telegram: @uni_k

www.konkur.in

Radford, Alec, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, *et al.* 2021. *Learning Transferable Visual Models From Natural Language Supervision.* Technical report. arXiv:2103.00020.

Radford, Alec, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. *Language Models are Unsupervised Multitask Learners.* Technical report. OpenAI.

Rakhimov, Ruslan, Denis Volkhonskiy, Alexey Artemov, Denis Zorin, and Evgeny Burnaev. 2020. *Latent Video Transformer.* Technical report. arXiv:2006.10704.

Ramachandran, P., B. Zoph, and Q. V. Le. 2017. *Searching for Activation Functions.* Technical report. arXiv:1710.05941v2.

Rao, C. R., and S. K. Mitra. 1971. *Generalized Inverse of Matrices and Its Applications.* Wiley.

Redmon, Joseph, Santosh Kumar Divvala, Ross B. Girshick, and Ali Farhadi. 2015. *You Only Look Once: Unified, Real-Time Object Detection.* Technical report. arxiv:1506.02640.

Ren, Shaoqing, Kaiming He, Ross B. Girshick, and Jian Sun. 2015. *Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks.* Technical report. arxiv:1506.01497.

Rezende, Danilo J, Shakir Mohamed, and Daan Wierstra. 2014. "Stochastic backpropagation and approximate inference in deep generative models." In *Proceedings of the 31st International Conference on Machine Learning (ICML-14),* 1278–1286.

Ricotti, L. P., S. Ragazzini, and G. Martinelli. 1988. "Learning of word stress in a sub-optimal second order backpropagation neural network." In *Proceedings of the IEEE International Conference on Neural Networks,* 1:355–361. IEEE.

Robert, C. P., and G. Casella. 1999. *Monte Carlo Statistical Methods.* Springer.

Rombach, Robin, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. 2021. *High-Resolution Image Synthesis with Latent Diffusion Models.* Technical report. arXiv:2112.10752.

Ronneberger, Olaf, Philipp Fischer, and Thomas Brox. 2015. "U-Net: Convolutional Networks for Biomedical Image Segmentation." In *Medical Image Computing and Computer-Assisted Intervention – MICCAI,* edited by N. Navab, J. Hornegger, W. Wells, and A. Frangi. Springer.

Rosenblatt, F. 1962. *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms.* Spartan.

Roweis, S. 1998. "EM algorithms for PCA and SPCA." In *Advances in Neural Information Processing Systems,* edited by M. I. Jordan, M. J. Kearns, and S. A. Solla, 10:626–632. MIT Press.

Roweis, S., and Z. Ghahramani. 1999. "A unifying review of linear Gaussian models." *Neural Computation* 11 (2): 305–345.

Rubin, D. B., and D. T. Thayer. 1982. "EM algorithms for ML factor analysis." *Psychometrika* 47 (1): 69–76.

Rumelhart, D. E., G. E. Hinton, and R. J. Williams. 1986. "Learning internal representations by error propagation." In *Parallel Distributed Processing: Explorations in the Microstructure of Cognition,* edited by D. E. Rumelhart, J. L. McClelland, and the PDP Research Group, vol. 1: Foundations, 318–362. Reprinted in Anderson and Rosenfeld (1988). MIT Press.

Ruthotto, L., and E. Haber. 2021. *An introduction to deep generative modeling.* Technical report. arXiv:2103.05180.

Sagan, H. 1969. *Introduction to the Calculus of Variations.* Dover.

Saharia, Chitwan, William Chan, Huiwen Chang, Chris A. Lee, Jonathan Ho, Tim Salimans, David J. Fleet, and Mohammad Norouzi. 2021. *Palette: Image-to-Image Diffusion Models.* Technical report. arXiv:2111.05826.

Saharia, Chitwan, William Chan, Saurabh Saxena, Lala Li, Jay Whang, Emily Denton, Seyed Kamyar Seyed Ghasemipour, *et al.* 2022. *Photorealistic Text-to-Image Diffusion Models*

Telegram: @uni_k

www.konkur.in

with Deep Language Understanding. Technical report. arXiv:2205.11487.

Saharia, Chitwan, Jonathan Ho, William Chan, Tim Salimans, David J. Fleet, and Mohammad Norouzi. 2021. *Image Super-Resolution via Iterative Refinement.* Technical report. arXiv:2104.07636.

Santurkar, S., D. Tsipras, A. Ilyas, and A. Madry. 2018. *How does batch normalization help optimization?* Technical report. arXiv:1805.11604.

Satorras, Victor Garcia, Emiel Hoogeboom, and Max Welling. 2021. *E(n) Equivariant Graph Neural Networks.* Technical report. arXiv:2102.09844.

Schölkopf, B., and A. J. Smola. 2002. *Learning with Kernels.* MIT Press.

Schuhmann, Christoph, Richard Vencu, Romain Beaumont, Robert Kaczmarczyk, Clayton Mullis, Aarush Katta, Theo Coombes, Jenia Jitsev, and Aran Komatsuzaki. 2021. *LAION-400M: Open Dataset of CLIP-Filtered 400 Million Image-Text Pairs.* Technical report. arXiv:2111.02114.

Schuster, Mike, and Kaisuke Nakajima. 2012. "Japanese and Korean voice search." In *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP),* 5149–5152.

Selvaraju, Ramprasaath R., Abhishek Das, Ramakrishna Vedantam, Michael Cogswell, Devi Parikh, and Dhruv Batra. 2016. *Grad-CAM: Visual Explanations from Deep Networks via Gradient-based Localization.* Technical report. arXiv:1610.02391.

Sennrich, Rico, Barry Haddow, and Alexandra Birch. 2015. *Neural Machine Translation of Rare Words with Subword Units.* Technical report. arXiv:1508.07909.

Sermanet, Pierre, David Eigen, Xiang Zhang, Michael Mathieu, Rob Fergus, and Yann LeCun. 2013. *OverFeat: Integrated Recognition, Localization and Detection using Convolutional Networks.* Technical report. arXiv:1312.6229.

Shachter, R. D., and M. Peot. 1990. "Simulation Approaches to General Probabilistic Inference on Belief Networks." In *Uncertainty in Artificial Intelligence,* edited by P. P. Bonissone, M. Henrion, L. N. Kanal, and J. F. Lemmer, vol. 5. Elsevier.

Shannon, C. E. 1948. "A mathematical theory of communication." *The Bell System Technical Journal* 27 (3): 379–423 and 623–656.

Shen, Sheng, Zhen Dong, Jiayu Ye, Linjian Ma, Zhewei Yao, Amir Gholami, Michael W. Mahoney, and Kurt Keutzer. 2019. *Q-BERT: Hessian Based Ultra Low Precision Quantization of BERT.* Technical report. arXiv:1909.05840.

Simard, P., B. Victorri, Y. LeCun, and J. Denker. 1992. "Tangent prop – a formalism for specifying selected invariances in an adaptive network." In *Advances in Neural Information Processing Systems,* edited by J. E. Moody, S. J. Hanson, and R. P. Lippmann, 4:895–903. Morgan Kaufmann.

Simard, P. Y., D. Steinkraus, and J. Platt. 2003. "Best practice for convolutional neural networks applied to visual document analysis." In *Proceedings International Conference on Document Analysis and Recognition (ICDAR),* 958–962. IEEE Computer Society.

Simonyan, Karen, Andrea Vedaldi, and Andrew Zisserman. 2013. "Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps." In *Computer Vision and Pattern Recognition.* ArXiv:1312.6034.

Simonyan, Karen, and Andrew Zisserman. 2014. *Very Deep Convolutional Networks for Large-Scale Image Recognition.* Technical report. arXiv:1409.1556.

Sirovich, L. 1987. "Turbulence and the Dynamics of Coherent Structures." *Quarterly Applied Mathematics* 45 (3): 561–590.

Sohl-Dickstein, Jascha, Eric A. Weiss, Niru Maheswaranathan, and Surya Ganguli. 2015. *Deep Unsupervised Learning using Nonequi-*

Telegram: @uni_k

www.konkur.in

librium Thermodynamics. Technical report. arXiv:1503.03585.

Sønderby, C., J. Caballero, L. Theis, W. Shi, and F. Huszár. 2016. *Amortised MAP inference for image super-resolution.* Technical report. arXiv:1610.04490.

Song, Jiaming, Chenlin Meng, and Stefano Ermon. 2020. *Denoising Diffusion Implicit Models.* Technical report. arXiv:2010.02502.

Song, Yang, and Stefano Ermon. 2019. "Generative Modeling by Estimating Gradients of the Data Distribution." In *Advances in Neural Information Processing Systems,* 11895–11907. ArXiv:1907.05600.

Song, Yang, Sahaj Garg, Jiaxin Shi, and Stefano Ermon. 2019. "Sliced score matching: A scalable approach to density and score estimation." In *Uncertainty in Artificial Intelligence,* 204. ArXiv:1905.07088.

Song, Yang, and Diederik P. Kingma. 2021. *How to Train Your Energy-Based Models.* Technical report. arXiv:2101.03288.

Song, Yang, Jascha Sohl-Dickstein, Diederik P. Kingma, Abhishek Kumar, Stefano Ermon, and Ben Poole. 2020. *Score-Based Generative Modeling through Stochastic Differential Equations.* Technical report. arXiv:2011.13456.

Srivastava, N., G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. 2014. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting." *Journal of Machine Learning Research* 15:1929–1958.

Stone, J. V. 2004. *Independent Component Analysis: A Tutorial Introduction.* MIT Press.

Sutskever, I., J. Martens, G. Dahl, and G. E. Hinton. 2013. "On the importance of initialization and momentum in deep learning." In *Proceedings of the International Conference on Machine Learning (ICML).*

Sutton, R. 2019. *The Bitter Lesson.* URL: incompleteideas.net/IncIdeas/BitterLesson.html.

Szegedy, Christian, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. 2013. *Intriguing properties of neural networks.* Technical report. arXiv:1312.6199.

Szeliski, R. 2022. *Computer Vision: Algorithms and Applications.* Second. Springer.

Tarassenko, L. 1995. "Novelty detection for the identification of masses in mamograms." In *Proceedings of the Fourth IEE International Conference on Artificial Neural Networks,* 4:442–447. IEE.

Tay, Yi, Mostafa Dehghani, Dara Bahri, and Donald Metzler. 2020. *Efficient Transformers: A Survey.* Technical report. arXiv:2009.06732.

Tibshirani, R. 1996. "Regression shrinkage and selection via the lasso." *Journal of the Royal Statistical Society, B* 58:267–288.

Tipping, M. E., and Christopher M. Bishop. 1997. *Probabilistic Principal Component Analysis.* Technical report NCRG/97/010. Neural Computing Research Group, Aston University.

Tipping, M. E., and Christopher M. Bishop. 1999. "Probabilistic Principal Component Analysis." *Journal of the Royal Statistical Society, Series B* 21 (3): 611–622.

Vapnik, V. N. 1995. *The nature of statistical learning theory.* Springer.

Vaswani, Ashish, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. *Attention Is All You Need.* Technical report. arXiv:1706.03762.

Veličković, Petar. 2023. *Everything is Connected: Graph Neural Networks.* Technical report. arXiv:2301.08210.

Veličković, Petar, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2017. *Graph Attention Networks.* Technical report. arXiv:1710.10903.

Vidakovic, B. 1999. *Statistical Modelling by Wavelets.* Wiley.

Telegram: @uni_k

www.konkur.in

Vig, Jesse, Ali Madani, Lav R. Varshney, Caiming Xiong, Richard Socher, and Nazneen Fatema Rajani. 2020. *BERTology Meets Biology: Interpreting Attention in Protein Language Models.* Technical report. arXiv:2006.15222.

Vincent, P. 2011. "A connection between score matching and denoising autoencoders." *Neural Computation* 23:1661–1674.

Vincent, Pascal, Hugo Larochelle, Yoshua Bengio, and Pierre-Antoine Manzagol. 2008. "Extracting and Composing Robust Features with Denoising Autoencoders." In *Proceedings of the International Conference on Machine Learning (ICML).*

Walker, A. M. 1969. "On the asymptotic behaviour of posterior distributions." *Journal of the Royal Statistical Society, B* 31 (1): 80–88.

Wang, Chengyi, Sanyuan Chen, Yu Wu, Ziqiang Zhang, Long Zhou, Shujie Liu, Zhuo Chen, *et al.* 2023. *Neural Codec Language Models are Zero-Shot Text to Speech Synthesizers.* Technical report. arXiv:2301.02111.

Weisstein, E. W. 1999. *CRC Concise Encyclopedia of Mathematics.* Chapman / Hall, / CRC.

Welling, Max, and Yee Whye Teh. 2011. "Bayesian Learning via Stochastic Gradient Langevin Dynamics." In *Proceedings of the International Conference on Machine Learning (ICML).*

Williams, P. M. 1996. "Using neural networks to model conditional multivariate densities." *Neural Computation* 8 (4): 843–854.

Williams, R J. 1992. "Simple statistical gradient-following algorithms for connectionist reinforcement learning." *Machine Learning* 8:229–256.

Winn, J., C. M. Bishop, T. Diethe, J. Guiver, and Y. Zaykov. 2023. *Model-Based Machine Learning*. Www.mbmlbook.com. Chapman / Hall.

Wolpert, D. H. 1996. "The lack of a-priori distinctions between learning algorithms." *Neural Computation* 8:1341–1390.

Wu, Zhirong, Yuanjun Xiong, Stella Yu, and Dahua Lin. 2018. *Unsupervised Feature Learning via Non-Parametric Instance-level Discrimination.* Technical report. arXiv:1805.01978.

Wu, Zonghan, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. 2019. *A Comprehensive Survey on Graph Neural Networks.* Technical report. arXiv:1901.00596.

Yan, Wilson, Yunzhi Zhang, Pieter Abbeel, and Aravind Srinivas. 2021. *VideoGPT: Video Generation using VQ-VAE and Transformers.* Technical report. arXiv:2104.10157.

Yang, Ruihan, Prakhar Srivastava, and Stephan Mandt. 2022. *Diffusion Probabilistic Modeling for Video Generation.* Technical report. arXiv:2203.09481.

Yilmaz, Fatih Furkan, and Reinhard Heckel. 2022. *Regularization-wise double descent: Why it occurs and how to eliminate it.* Technical report. arXiv:2206.01378.

Yosinski, Jason, Jeff Clune, Anh Mai Nguyen, Thomas J. Fuchs, and Hod Lipson. 2015. *Understanding Neural Networks Through Deep Visualization.* Technical report. arXiv:1506.06579.

Yu, Jiahui, Xin Li, Jing Yu Koh, Han Zhang, Ruoming Pang, James Qin, Alexander Ku, Yuanzhong Xu, Jason Baldridge, and Yonghui Wu. 2021. *Vector-quantized Image Modeling with Improved VQGAN.* Technical report. arXiv:2110.04627.

Yu, Jiahui, Yuanzhong Xu, Jing Yu Koh, Thang Luong, Gunjan Baid, Zirui Wang, Vijay Vasudevan, *et al.* 2022. *Scaling Autoregressive Models for Content-Rich Text-to-Image Generation.* Technical report. arXiv:2206.10789.

Yu, Lili, Bowen Shi, Ramakanth Pasunuru, Benjamin Muller, Olga Golovneva, Tianlu Wang, Arun Babu, *et al.* 2023. *Scaling Autoregressive Multi-Modal Models: Pretraining and Instruction Tuning.* Technical report. arXiv:2309.02591.

Telegram: @uni_k

www.konkur.in

Zaheer, Manzil, Satwik Kottur, Siamak Ravanbakhsh, Barnabas Poczos, Ruslan Salakhutdinov, and Alexander Smola. 2017. *Deep Sets.* Technical report. arXiv:1703.06114.

Zarchan, P., and H. Musoff. 2005. *Fundamentals of Kalman Filtering: A Practical Approach.* Second. AIAA.

Zeiler, Matthew D., and Rob Fergus. 2013. *Visualizing and Understanding Convolutional Networks.* Technical report. arXiv:1311.2901.

Zhang, Chiyuan, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. 2016. *Understanding deep learning requires rethinking generalization.* Technical report. arXiv:1611.03530.

Zhao, Wayne Xin, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, *et al.* 2023. *A Survey of Large Language Models.* Technical report. arXiv:2303.18223.

Zhou, Jie, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. 2018. *Graph Neural Networks: A Review of Methods and Applications.* Technical report. arXiv:1812.08434.

Zhou, Y., and R. Chellappa. 1988. "Computation of optic flow using a neural network." In *International Conference on Neural Networks,* 71–78. IEEE.

Zhu, J-Y, T. Park, P. Isola, and A. Efros. 2017. *Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks.* Technical report. arXiv:1703.10593.

Telegram: @uni_k

www.konkur.in

Index

Page numbers in **bold** indicate the primary source of information for the corresponding topic.

1×1 convolution, 296
1-of-K coding, 68, **135**, 460

acceptance criterion, **441**, 445, 448
activation, 17
activation function, 17, 158, 180, **182**
active constraint, 623
AdaGrad, 223
Adam optimization, 224
adaptive rejection sampling, 435
adjacency matrix, 410
adjoint sensitivity method, 556
adversarial attack, 306
aggregation, 415
aleatoric uncertainty, 23
AlexNet, 300
alpha family, 61
amortized inference, 572
ancestral sampling, 450
anchor, 191
annealed Langevin dynamics, 598
AR model, *see* autoregressive model
area under the ROC curve, 149
artificial intelligence, 1
attention, 358
attention head, 366
audio data, 399
auto-associative neural network, *see* autoencoder
autoencoder, 188, **563**
automatic differentiation, 22, 233, **244**
autoregressive flow, 552
autoregressive model, 5, **350**, 379
average pooling, 297

backpropagation, 19, **233**
backpropagation through time, 381
bag of words, 378
bagging, 278
base distribution, 547
basis function, 112, 158, 172, **172**
batch gradient descent, 214
batch learning, 117
batch normalization, 227
Bayes net, 326
Bayes' theorem, 28
Bayesian network, 326
Bayesian probability, 54
beam search, 386
Bernoulli distribution, **66**, 94
Bernoulli mixture model, 481
BERT, 388
bi-gram model, 379
bias, **39**, 125
bias parameter, **112**, 132, 180
bias–variance trade-off, 123
BigGAN, 539

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024
C. M. Bishop, H. Bishop, *Deep Learning*, https://doi.org/10.1007/978-3-031-45468-4

Telegram: @uni_k

www.konkur.in

bijective function, 548
binomial distribution, 67
bits, 46
blind source separation, 514
blocked path, 339, **343**
boosting, 279
bootstrap, 278
bottleneck, 382
bounding box, 309
Box–Muller method, 432
byte pair encoding, 377

canonical correlation analysis, 501
canonical link function, 164
Cauchy distribution, 432
causal attention, 384
causality, 347
central differences, 239
central limit theorem, 71
ChatGPT, 394
child node, 249, **327**
Cholesky decomposition, 433
circular normal distribution, 89
classical probability, 54
classification, 3
CLIP, 192
co-parents, 348
codebook vector, 398, **465**
collider node, 341
combining models, 146
committee, 277
complete data set, 476
completing the square, 77
computer vision, 288
concave function, 52
concentration parameter, 92
condition number, 220
conditional entropy, 53
conditional expectation, 35
conditional independence, 146, **337**
conditional mixture model, 199
conditional probability, 27
conditional VAE, 576
conditioner, 552
confusion matrix, 147
continuous bag of words, 375
continuous normalizing flow, 557
contrastive divergence, 455
contrastive learning, 191
convex function, 51
convolution, **290**, 322
convolutional network, 287
correlation matrix, 503
cost function, 140
coupling flow, 549
coupling function, 552
covariance, 35
Cox's axioms, 54
cross attention, 390
cross-correlation, **292**, 322
cross-entropy error function, **160**, 162, 196
cross-validation, 14
cumulative distribution function, 32
curse of dimensionality, 172
curve fitting, 6
CycleGAN, 539

d-separation, 338, **343**, 479
DAG, *see* directed acyclic graph
data augmentation, 192, **257**
data compression, 465
DDIM, 594
DDPM, 581
decision, 120
decision boundary, 131, **139**
decision region, 131, **139**
decision surface, *see* decision boundary
decision theory, 120, **138**
decoder, 563
deep double descent, 268
deep learning, 20
deep neural networks, 20
deep sets, 417
DeepDream, 308
degrees of freedom, 495
denoising, 581
denoising autoencoder, 567
denoising diffusion implicit model, 594
denoising diffusion probabilistic model, 581
denoising score matching, 597

Telegram: @uni_k

www.konkur.in

density estimation, 37, **65**
dequantization, 526
descendant node, 341
design matrix, 116
development set, 14
diagonal covariance matrix, 75
differential entropy, 50
diffusion kernel, 583
diffusion model, 581
Dirac delta function, 34
directed acyclic graph, 329
directed cycle, 329
directed factorization, 349
directed graph, 326
directed graphical model, 326
discriminant function, **132**, 143
discriminative model, **144**, 157, 346
disentangled representations, 542
distributed representation, 187
dot-product attention, 363
double descent, 268
dropout, 279

E step, **472**, 476
early stopping, 266
earth mover's distance, 538
ECM, *see* expectation conditional maximization
edge, **326**, 410
edge detection, 292
ELBO, *see* evidence lower bound
EM, *see* expectation maximization
embedding space, 188
embedding vector, 409
encoder, 563
energy function, 452
energy-based models, 452
ensemble methods, 277
entropy, 46
epistemic uncertainty, 23
epoch, 215
equality constraint, 623
equivariance, **259**, 292, 296, 371, 412
erf function, 164
error backpropagation, *see* backpropagation
error function, **8**, 55, 194, 210

Euler–Lagrange equations, 619
evaluation trace, 247
evidence lower bound, **485**, 516, 570, 588
expectation, 34
expectation conditional maximization, 489
expectation maximization, 470, **474**, 517, 519
expectation step, *see* E step
expectations, 430
explaining away, 343
exploding gradient, **227**, 382
exponential distribution, **34**, 431
exponential family, **94**, 156, 329
expression swell, 245

factor analysis, 513
factor graph, 327
factor loading, 513
false negative, 25
false positive, 25
fast gradient sign method, 306
fast R-CNN, 314
feature extraction, **20**, 113
feature map, 291
features, 179
feed-forward network, 172, **193**
feed-forward networks, 19
few-shot learning, **191**, 394
filter, 291
fine-tuning, 3, 22, **189**, 392
flow matching, 558
forward kinematics, 199
forward problem, 198
forward propagation, 235
foundation model, **22**, 358, 392, 409
frequentist probability, 54
fuel system, 341
fully connected graphical model, 328
fully convolutional network, 318
functional, 617

Gabor filters, 302
gamma distribution, 434
GAN, *see* generative adversarial network
gated recurrent unit, 382
Gaussian, 36, **70**
Gaussian mixture, 86, 200, 271, **466**

Telegram: @uni_k

www.konkur.in

GEM, *see* generalized EM algorithm
generalization, 6
generalized EM algorithm, 489
generalized linear model, **158**, 165
generative adversarial network, 533
generative AI, 4
generative model, 4, **144**, 346, 533
generative pre-trained transformer, 6, **383**
geometric deep learning, 424
Gibbs sampling, 446
global minimum, 211
GNN, *see* graph neural network
GPT, *see* generative pre-trained transformer
GPU, *see* graphics processing unit
gradient descent, 209
graph attention network, 421
graph convolutional network, 414
graph neural network, 407
graph representation learning, 409
graphical model, 326
graphical model factorization, 329
graphics processing unit, **20**, 358
group theory, 256
guidance, 600

Hadamard product, 550
Hamiltonian Monte Carlo, 451
handwritten digit, 501
He initialization, 216
head-to-head path, 341
head-to-tail path, 340
Heaviside step function, 161
Hessian matrix, **211**, 242
Hessian outer product approximation, 243
heteroscedastic, 200
hidden Markov model, 380, **480**
hidden unit, **19**, 180
hidden variable, *see* latent variable
hierarchical representation, 187
histogram density estimation, 98
history of machine learning, 16
hold-out set, 14
homogeneous Markov chain, 443
Hooke's law, 520
Hutchinson's trace estimator, 557
hybrid Monte Carlo, 451
hyperparameter, 14

IAF, *see* inverse autoregressive flow
ICA, *see* independent component analysis
identifiability, 470
IID, *see* independent and identically distributed
image segmentation, 315
ImageNet data set, 299
importance sampling, **437**, 450
importance weight, 437
improper distribution, 33
improper prior, 263
inactive constraint, 623
incomplete data set, 476
independent and identically distributed, **37**, 344
independent component analysis, 514
independent factor analysis, 515
independent variables, 31
inductive bias, 19, **254**
inductive learning, **409**, 420
inequality constraint, 623
inference, 120, 138, **143**, 336
InfoNCE, 191
information theory, 46
instance discrimination, 192
internal covariate shift, 229
internal representation, 308
intersection-over-union, 310
intrinsic dimensionality, 496
invariance, 256, **256**, 297, 412
inverse autoregressive flow, 553
inverse kinematics, 199
inverse problem, 123, **198**, 254, 346
Iris data, 173
IRLS, *see* iterative reweighted least squares
isotropic covariance matrix, 75
iterative reweighted least squares, 160

Jacobian matrix, 44, **240**
Jensen's inequality, 52
Jensen–Shannon divergence, 544

K nearest neighbours, 103
K-means clustering algorithm, **460**, 480
Kalman filter, 353, **515**

Telegram: @uni_k

www.konkur.in

Karush–Kuhn–Tucker conditions, 624
kernel density estimator, **100**, 596
kernel function, 101
kernel image, 291
KKT, *see* Karush-Kuhn-Tucker conditions
KL divergence, *see* Kullback–Leibler divergence
Kosambi–Karhunen–Loève transform, 497
Kullback–Leibler divergence, **51**, 486

Lagrange multiplier, 621
Lagrangian, 622
Langevin dynamics, 454
Langevin sampling, 455
language model, 382
Laplace distribution, 34
large language model, 5, 382, **390**
lasso, 264
latent class analysis, 481
latent diffusion model, 601
latent variable, 76, 335, **459**, 495
layer normalization, **229**, 369
LDM, *see* latent diffusion model
LDS, *see* linear dynamical system
leaky ReLU, 185
learning curve, 223, **266**
learning rate parameter, 214
learning to learn, 190
least-mean-squares algorithm, 118
least-squares GAN, 537
leave-one-out, 15
LeNet convolutional network, 299
Levenberg–Marquardt approximation, 244
likelihood function, **38**, 468
likelihood weighted sampling, 451
linear discriminant, 132
linear dynamical system, 515
linear independence, 610
linear regression, 6, **112**
linear-Gaussian model, 79, 332, **332**
linearly separable, 132
link, *see* edge
link function, **158**, 165
LLM, *see* large language model
LMS, *see* least-mean-squares algorithm
local minimum, 211
log odds, 151
logic sampling, 450
logistic regression, 159
logistic sigmoid, **95**, 113, 151, 159
logit function, 151
long short-term memory, 382
LoRA, *see* low-rank adaptation
loss function, **120**, 140
loss matrix, 142
lossless data compression, 465
lossy data compression, 465
low-rank adaptation, 392
LSGAN, *see* least-squares GAN
LSTM, *see* long short-term memory

M step, **472**, 477
macrostate, 48
MAE, *see* masked autoencoder
MAF, *see* masked autoregressive flow
Mahalanobis distance, 71
manifold, **177**, 522
MAP, *see* maximum a posteriori
marginal probability, 27
Markov blanket, **347**, 449
Markov boundary, *see* Markov blanket
Markov chain, **351**, 442
Markov chain Monte Carlo, 440
Markov model, 351
Markov random field, 327
masked attention, 384
masked autoencoder, 567
masked autoregressive flow, 553
max-pooling, 297
max-unpooling, 317
maximization step, *see* M step
maximum a posteriori, **56**, 477
maximum likelihood, **38**, 84, 115, 153
MCMC, *see* Markov chain Monte Carlo
MDN, *see* mixture density network
mean, 36
mean value theorem, 49
measure theory, 33
mel spectrogram, 399
message-passing, 414
message-passing neural network, 415

Telegram: @uni_k

www.konkur.in

meta-learning, 190
Metropolis algorithm, 441
Metropolis–Hastings algorithm, 445
microstate, 48
mini-batches, 216
minimum risk, 145
Minkowski loss, 122
missing at random, **477**, 519
missing data, 519
mixing coefficient, 87
mixture component, 87
mixture density network, 198
mixture distribution, 459
mixture model, 459
mixture of Gaussians, 86, 200, 271, **466**
MLP, *see* multilayer perceptron
MNIST data, 495
mode collapse, 536
model averaging, 277
model comparison, 9
model selection, 14
moment, 37
momentum, 220
Monte Carlo dropout, 280
Monte Carlo sampling, 429
Moore-Penrose pseudo-inverse, *see* pseudo-inverse
MRF, *see* Markov random field
multi-class logistic regression, 161
multi-head attention, 366
multilayer perceptron, 18, **172**
multimodal transformer, 394
multimodality, 199
multinomial distribution, **70**, 95
multiplicity, 48
multitask learning, 190
mutual information, 54

n-gram model, 379
naive Bayes model, 147, **344**, 378
nats, 47
natural language processing, 374
natural parameter, 94
nearest-neighbours, 103
neocognitron, 302
Nesterov momentum, 221
neural ordinary differential equation, 554
neuroscience, 302
NLP, *see* natural language processing
no free lunch theorem, 255
node, **326**, 410
noise, 23
noiseless coding theorem, 47
noisy-OR, 354
non-identifiability, 513
non-max suppression, 314
nonparametric methods, 66, **98**
normal distribution, *see* Gaussian
normal equations, 116
normalized exponential, *see* softmax function
novelty detection, 144

object detection, 308
observed variable, 335
Old Faithful data, 86
on-hot encoding, *see* 1-of-K encoding
one-shot learning, 191
one-versus-one classifier, 134
one-versus-the-rest classifier, 134
online gradient descent, 215
online learning, 117
ordered over-relaxation, 449
outer product approximation, 244
outlier, **137**, 144, 164
over-fitting, **10**, 123, 470
over-relaxation, 449
over-smoothing, 422

padding, 294
parameter sharing, **270**, 331
parameter shrinkage, 118
parameter tying, *see* parameter sharing
parent node, 247, **327**
partition function, 452
Parzen estimator, *see* kernel density estimator
Parzen window, 101
PCA, *see* principal component analysis
perceptron, 17
periodic variables, 89
permutation matrix, 411
PixelCNN, 397
PixelRNN, 397

Telegram: @uni_k

www.konkur.in

plate, 334
polynomial curve fitting, 6
pooling, 296
positional encoding, 371
positive definite covariance, 72
positive definite matrix, 615
posterior collapse, 577
posterior probability, 31
power method, 498
pre-activation, 17
pre-processing, 20
pre-training, **189**, 392
precision matrix, 77
precision parameter, 36
predictive distribution, **42**, 120
prefix prompt, 394
principal component analysis, **497**, 506, 565
principal subspace, 497
prior, 263
prior knowledge, 19, **255**
prior probability, **31**, 145
probabilistic graphical model, *see* graphical model
probabilistic PCA, 506
probability, 25
probability density, 32
probability theory, 23
probit function, 164
probit regression, 163
product rule of probability, 26, **28**, 326
prompt, **394**, 601
prompt engineering, 394
proposal distribution, **433**, 437, 441
pseudo-inverse, **116**, 136
pseudo-random numbers, 430

quadratic discriminant, 153

radial basis functions, 179
random variable, 26
raster scan, 397
readout layer, 419
real NVP normalizing flow, 549
receiver operating characteristic, *see* ROC curve
receptive field, **290**, 416
recurrent neural network, 380
regression, 3
regression function, 121
regularization, 12, **253**
regularized least squares, 118
reject option, **142**, 145
rejection sampling, 433
relative entropy, 51
reparameterization trick, 574
representation learning, 22, **188**
residual block, 275
residual connection, 22, **274**
residual network, 275
resnet, *see* residual network
responsibility, 88, **468**
RLHF, 394
RMS error, *see* root-mean-square error
RMSProp, 223
RNN, *see* recurrent neural network
robot arm, 198
robustness, 137
ROC curve, 148
root-mean-square error, 10

saliency map, 305
same convolution, 294
sample mean, 39
sample variance, 39
sampling, 429
sampling-importance-resampling, 439
scale invariance, 256
scaled self-attention, 366
scaling hypothesis, 358
Schur complement, 79
score function, 455, 594
score matching, 594
self-attention, 362
self-supervised learning, **5**, 375
semi-supervised learning, 420
sequential estimation, 85
sequential gradient descent, 118
sequential learning, 117
SGD, *see* stochastic gradient descent
shared parameters, *see* parameter sharing
shared weights, 292
shattered gradients, 274
shrinkage, 13

Telegram: @uni_k

www.konkur.in

sigmoid, *see* logistic sigmoid
singular value decomposition, 117
SIR, *see* sampling-importance-resampling
skip-grams, 375
skip-layer connections, 274
sliding window, 311
smoothing parameter, 100
soft ReLU, 185
soft weight sharing, 271
softmax function, 96, **152**, 197, 201, 363
softplus activation function, 185
sparse autoencoders, 566
sparse connections, 292
sparsity, 264
sphering, 504
standard deviation, 36
standardizing, 462, **503**
state-space model, 352
statistical bias, *see* bias
statistical independence, *see* independent variables
steepest descent, 214
Stein score, *see* score function
Stirling's approximation, 48
stochastic, 8
stochastic differential equation, 598
stochastic gradient descent, 19, **214**, 215
stochastic variable, 26
strided convolution, 294
strides, 311
structured data, 287, **407**
style transfer, 320
sufficient statistics, 67, 69, 84, **97**
sum rule of probability, 26, **28**, 326
sum-of-squares error, **8**, 41, 136
supervised learning, **3**, 420
support vector machine, 179
SVD, *see* singular value decomposition
SVM, *see* support vector machine
swish activation function, 205
symmetry, 256
symmetry breaking, 216

tail-to-tail path, 339
tangent propagation, 258
temperature, 387
tensor, **194**, 295
test set, 10, **14**
text-to-speech, 400
tied parameters, *see* parameter sharing
token, 360
tokenization, 377
training set, 3
transductive, 409, **419**
transductive learning, 420
transfer learning, 3, **189**, 218, 388
transformers, 357
transition probability, 443
translation invariance, 256
transpose convolution, 318
tri-gram model, 379
TTS, *see* text-to-speech

U-net, 319
undetermined multiplier, *see* Lagrange multiplier
undirected graphical model, 327
uniquenesses, 513
universal approximation theorems, 182
unobserved variable, *see* latent variable
unsupervised learning, **4**, 188
utility function, 140

VAE, *see* variational autoencoder
valid convolution, 294
validation set, 14
vanishing gradient, **227**, 382
variance, **35**, 36, 125
variational autoencoder, 569
variational inference, 485
variational lower bound, *see* evidence lower bound
vector quantization, 398, **465**
vertex, *see* node
vision transformer, 395
von Mises distribution, 89
voxel, 289

Wasserstein distance, 538
Wasserstein GAN, 538
wavelets, 114
weakly supervised, 192
weight decay, 13, **260**
weight parameter, **17**, 180

Telegram: @uni_k

www.konkur.in

weight sharing, *see* parameter sharing
weight vector, 132
weight-space symmetry, 185
WGAN, *see* Wasserstein GAN
whitening, 502
Woodbury identity, 610
word embedding, 375
word2vec, 375
wrapped distribution, 94

Yellowstone National Park, 86

Telegram: @uni_k