Chapman & Hall/CRC
Machine Learning & Pattern Recognition Series

MACHINE LEARNING

An Algorithmic Perspective

SECOND EDITION

STEPHEN MARSLAND

CRC Press
Taylor & Francis Group
A CHAPMAN & HALL BOOK

WITH VITALSOURCE®
EBOOK

MACHINE LEARNING

An Algorithmic Perspective

SECOND EDITION

Chapman & Hall/CRC
Machine Learning & Pattern Recognition Series

SERIES EDITORS

Ralf Herbrich
Amazon Development Center
Berlin, Germany

Thore Graepel
Microsoft Research Ltd.
Cambridge, UK

AIMS AND SCOPE

This series reflects the latest advances and applications in machine learning and pattern recognition through the publication of a broad range of reference works, textbooks, and handbooks. The inclusion of concrete examples, applications, and methods is highly encouraged. The scope of the series includes, but is not limited to, titles in the areas of machine learning, pattern recognition, computational intelligence, robotics, computational/statistical learning theory, natural language processing, computer vision, game AI, game theory, neural networks, computational neuroscience, and other relevant topics, such as machine learning applied to bioinformatics or cognitive science, which might be proposed by potential contributors.

PUBLISHED TITLES

BAYESIAN PROGRAMMING
Pierre Bessière, Emmanuel Mazer, Juan-Manuel Ahuactzin, and Kamel Mekhnacha

UTILITY-BASED LEARNING FROM DATA
Craig Friedman and Sven Sandow

HANDBOOK OF NATURAL LANGUAGE PROCESSING, SECOND EDITION
Nitin Indurkhya and Fred J. Damerau

COST-SENSITIVE MACHINE LEARNING
Balaji Krishnapuram, Shipeng Yu, and Bharat Rao

COMPUTATIONAL TRUST MODELS AND MACHINE LEARNING
Xin Liu, Anwitaman Datta, and Ee-Peng Lim

MULTILINEAR SUBSPACE LEARNING: DIMENSIONALITY REDUCTION OF MULTIDIMENSIONAL DATA
Haiping Lu, Konstantinos N. Plataniotis, and Anastasios N. Venetsanopoulos

MACHINE LEARNING: An Algorithmic Perspective, Second Edition
Stephen Marsland

A FIRST COURSE IN MACHINE LEARNING
Simon Rogers and Mark Girolami

MULTI-LABEL DIMENSIONALITY REDUCTION
Liang Sun, Shuiwang Ji, and Jieping Ye

ENSEMBLE METHODS: FOUNDATIONS AND ALGORITHMS
Zhi-Hua Zhou

Chapman & Hall/CRC
Machine Learning & Pattern Recognition Series

MACHINE LEARNING

An Algorithmic Perspective

SECOND EDITION

STEPHEN MARSLAND

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business

A CHAPMAN & HALL BOOK

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

© 2015 by Taylor & Francis Group, LLC
CRC Press is an imprint of Taylor & Francis Group, an Informa business

No claim to original U.S. Government works
Version Date: 20140826

International Standard Book Number-13: 978-1-4665-8333-7 (eBook - PDF)

This book contains information obtained from authentic and highly regarded sources. Reasonable efforts have been made to publish reliable data and information, but the author and publisher cannot assume responsibility for the validity of all materials or the consequences of their use. The authors and publishers have attempted to trace the copyright holders of all material reproduced in this publication and apologize to copyright holders if permission to publish in this form has not been obtained. If any copyright material has not been acknowledged please write and let us know so we may rectify in any future reprint.

Except as permitted under U.S. Copyright Law, no part of this book may be reprinted, reproduced, transmitted, or utilized in any form by any electronic, mechanical, or other means, now known or hereafter invented, including photocopying, microfilming, and recording, or in any information storage or retrieval system, without written permission from the publishers.

For permission to photocopy or use material electronically from this work, please access www.copyright.com (http://www.copyright.com/) or contact the Copyright Clearance Center, Inc. (CCC), 222 Rosewood Drive, Danvers, MA 01923, 978-750-8400. CCC is a not-for-profit organization that provides licenses and registration for a variety of users. For organizations that have been granted a photocopy license by the CCC, a separate system of payment has been arranged.

Trademark Notice: Product or corporate names may be trademarks or registered trademarks, and are used only for identification and explanation without intent to infringe.

Visit the Taylor & Francis Web site at
http://www.taylorandfrancis.com

and the CRC Press Web site at
http://www.crcpress.com

Again, for Monika

Contents

Prologue to 2nd Edition

There have been some interesting developments in machine learning over the past four years, since the 1st edition of this book came out. One is the rise of Deep Belief Networks as an area of real research interest (and business interest, as large internet-based companies look to snap up every small company working in the area), while another is the continuing work on statistical interpretations of machine learning algorithms. This second one is very good for the field as an area of research, but it does mean that computer science students, whose statistical background can be rather lacking, find it hard to get started in an area that they are sure should be of interest to them. The hope is that this book, focussing on the *algorithms* of machine learning as it does, will help such students get a handle on the ideas, and that it will start them on a journey towards mastery of the relevant mathematics and statistics as well as the necessary programming and experimentation.

In addition, the libraries available for the Python language have continued to develop, so that there are now many more facilities available for the programmer. This has enabled me to provide a simple implementation of the Support Vector Machine that can be used for experiments, and to simplify the code in a few other places. All of the code that was used to create the examples in the book is available at `http://stephenmonika.net/` (in the 'Book' tab), and use and experimentation with any of this code, as part of any study on machine learning, is strongly encouraged.

Some of the changes to the book include:

- the addition of two new chapters on two of those new areas: Deep Belief Networks (Chapter 17) and Gaussian Processes (Chapter 18).
- a reordering of the chapters, and some of the material within the chapters, to make a more natural flow.
- the reworking of the Support Vector Machine material so that there is running code and the suggestions of experiments to be performed.
- the addition of Random Forests (as Section 13.3), the Perceptron convergence theorem (Section 3.4.1), a proper consideration of accuracy methods (Section 2.2.4), conjugate gradient optimisation for the MLP (Section 9.3.2), and more on the Kalman filter and particle filter in Chapter 16.
- improved code including better use of naming conventions in Python.
- various improvements in the clarity of explanation and detail throughout the book.

I would like to thank the people who have written to me about various parts of the book, and made suggestions about things that could be included or explained better. I would also like to thank the students at Massey University who have studied the material with me, either as part of their coursework, or as first steps in research, whether in the theory or the application of machine learning. Those that have contributed particularly to the content of the second edition include Nirosha Priyadarshani, James Curtis, Andy Gilman, Örjan

Ekeberg, and the Osnabrück Knowledge-Based Systems Research group, especially Joachim Hertzberg, Sven Albrecht, and Thomas Wieman.

Stephen Marsland
Ashhurst, New Zealand

Prologue to 1st Edition

One of the most interesting features of machine learning is that it lies on the boundary of several different academic disciplines, principally computer science, statistics, mathematics, and engineering. This has been a problem as well as an asset, since these groups have traditionally not talked to each other very much. To make it even worse, the areas where machine learning methods can be applied vary even more widely, from finance to biology and medicine to physics and chemistry and beyond. Over the past ten years this inherent multi-disciplinarity has been embraced and understood, with many benefits for researchers in the field. This makes writing a textbook on machine learning rather tricky, since it is potentially of interest to people from a variety of different academic backgrounds.

In universities, machine learning is usually studied as part of artificial intelligence, which puts it firmly into computer science and—given the focus on algorithms—it certainly fits there. However, understanding why these algorithms work requires a certain amount of statistical and mathematical sophistication that is often missing from computer science undergraduates. When I started to look for a textbook that was suitable for classes of undergraduate computer science and engineering students, I discovered that the level of mathematical knowledge required was (unfortunately) rather in excess of that of the majority of the students. It seemed that there was a rather crucial gap, and it resulted in me writing the first draft of the student notes that have become this book. The emphasis is on the algorithms that make up the machine learning methods, and on understanding how and why these algorithms work. It is intended to be a practical book, with lots of programming examples and is supported by a website that makes available all of the code that was used to make the figures and examples in the book. The website for the book is: `http://stephenmonika.net/MLbook.html`.

For this kind of practical approach, examples in a real programming language are preferred over some kind of pseudocode, since it enables the reader to run the programs and experiment with data without having to work out irrelevant implementation details that are specific to their chosen language. Any computer language can be used for writing machine learning code, and there are very good resources available in many different languages, but the code examples in this book are written in Python. I have chosen Python for several reasons, primarily that it is freely available, multi-platform, relatively nice to use and is becoming a default for scientific computing. If you already know how to write code in any other programming language, then you should not have many problems learning Python. If you don't know how to code at all, then it is an ideal first language as well. Chapter A provides a basic primer on using Python for numerical computing.

Machine learning is a rich area. There are lots of very good books on machine learning for those with the mathematical sophistication to follow them, and it is hoped that this book could provide an entry point to students looking to study the subject further as well as those studying it as part of a degree. In addition to books, there are many resources for machine learning available via the Internet, with more being created all the time. The Machine Learning Open Source Software website at `http://mloss.org/software/` provides links to a host of software in different languages.

There is a very useful resource for machine learning in the UCI Machine Learning Repos-

itory (`http://archive.ics.uci.edu/ml/`). This website holds lots of datasets that can be downloaded and used for experimenting with different machine learning algorithms and seeing how well they work. The repository is going to be the principal source of data for this book. By using these test datasets for experimenting with the algorithms, we do not have to worry about getting hold of suitable data and preprocessing it into a suitable form for learning. This is typically a large part of any real problem, but it gets in the way of learning about the algorithms.

I am very grateful to a lot of people who have read sections of the book and provided suggestions, spotted errors, and given encouragement when required. In particular for the first edition, thanks to Zbigniew Nowicki, Joseph Marsland, Bob Hodgson, Patrick Rynhart, Gary Allen, Linda Chua, Mark Bebbington, JP Lewis, Tom Duckett, and Monika Nowicki. Thanks especially to Jonathan Shapiro, who helped me discover machine learning and who may recognise some of his own examples.

Stephen Marsland
Ashhurst, New Zealand

Introduction

Suppose that you have a website selling software that you've written. You want to make the website more personalised to the user, so you start to collect data about visitors, such as their computer type/operating system, web browser, the country that they live in, and the time of day they visited the website. You can get this data for any visitor, and for people who actually buy something, you know what they bought, and how they paid for it (say PayPal or a credit card). So, for each person who buys something from your website, you have a list of data that looks like (computer type, web browser, country, time, software bought, how paid). For instance, the first three pieces of data you collect could be:

- Macintosh OS X, Safari, UK, morning, SuperGame1, credit card
- Windows XP, Internet Explorer, USA, afternoon, SuperGame1, PayPal
- Windows Vista, Firefox, NZ, evening, SuperGame2, PayPal

Based on this data, you would like to be able to populate a 'Things You Might Be Interested In' box within the webpage, so that it shows software that might be relevant to each visitor, based on the data that you can access while the webpage loads, i.e., computer and OS, country, and the time of day. Your hope is that as more people visit your website and you store more data, you will be able to identify trends, such as that Macintosh users from New Zealand (NZ) love your first game, while Firefox users, who are often more knowledgeable about computers, want your automatic download application and virus/internet worm detector, etc.

Once you have collected a large set of such data, you start to examine it and work out what you can do with it. The problem you have is one of prediction: given the data you have, predict what the next person will buy, and the reason that you think that it might work is that people who seem to be similar often act similarly. So how can you actually go about solving the problem? This is one of the fundamental problems that this book tries to solve. It is an example of what is called supervised learning, because we know what the right answers are for some examples (the software that was actually bought) so we can give the learner some examples where we know the right answer. We will talk about supervised learning more in Section 1.3.

1.1 IF DATA HAD MASS, THE EARTH WOULD BE A BLACK HOLE

Around the world, computers capture and store terabytes of data every day. Even leaving aside your collection of MP3s and holiday photographs, there are computers belonging to shops, banks, hospitals, scientific laboratories, and many more that are storing data incessantly. For example, banks are building up pictures of how people spend their money,

hospitals are recording what treatments patients are on for which ailments (and how they respond to them), and engine monitoring systems in cars are recording information about the engine in order to detect when it might fail. The challenge is to do something useful with this data: if the bank's computers can learn about spending patterns, can they detect credit card fraud quickly? If hospitals share data, then can treatments that don't work as well as expected be identified quickly? Can an intelligent car give you early warning of problems so that you don't end up stranded in the worst part of town? These are some of the questions that machine learning methods can be used to answer.

Science has also taken advantage of the ability of computers to store massive amounts of data. Biology has led the way, with the ability to measure gene expression in DNA microarrays producing immense datasets, along with protein transcription data and phylogenetic trees relating species to each other. However, other sciences have not been slow to follow. Astronomy now uses digital telescopes, so that each night the world's observatories are storing incredibly high-resolution images of the night sky; around a terabyte per night. Equally, medical science stores the outcomes of medical tests from measurements as diverse as magnetic resonance imaging (MRI) scans and simple blood tests. The explosion in stored data is well known; the challenge is to do something useful with that data. The Large Hadron Collider at CERN apparently produces about 25 petabytes of data per year.

The size and complexity of these datasets mean that humans are unable to extract useful information from them. Even the way that the data is stored works against us. Given a file full of numbers, our minds generally turn away from looking at them for long. Take some of the same data and plot it in a graph and we can do something. Compare the table and graph shown in Figure 1.1: the graph is rather easier to look at and deal with. Unfortunately, our three-dimensional world doesn't let us do much with data in higher dimensions, and even the simple webpage data that we collected above has four different features, so if we plotted it with one dimension for each feature we'd need four dimensions! There are two things that we can do with this: reduce the number of dimensions (until our simple brains can deal with the problem) or use computers, which don't know that high-dimensional problems are difficult, and don't get bored with looking at massive data files of numbers. The two pictures in Figure 1.2 demonstrate one problem with reducing the number of dimensions (more technically, projecting it into fewer dimensions), which is that it can hide useful information and make things look rather strange. This is one reason why machine learning is becoming so popular — the problems of our human limitations go away if we can make computers do the dirty work for us. There is one other thing that can help if the number of dimensions is not too much larger than three, which is to use glyphs that use other representations, such as size or colour of the datapoints to represent information about some other dimension, but this does not help if the dataset has 100 dimensions in it.

In fact, you have probably interacted with machine learning algorithms at some time. They are used in many of the software programs that we use, such as Microsoft's infamous paperclip in Office (maybe not the most positive example), spam filters, voice recognition software, and lots of computer games. They are also part of automatic number-plate recognition systems for petrol station security cameras and toll roads, are used in some anti-skid braking and vehicle stability systems, and they are even part of the set of algorithms that decide whether a bank will give you a loan.

The attention-grabbing title to this section would only be true if data was very heavy. It is very hard to work out how much data there actually is in all of the world's computers, but it was estimated in 2012 that was about 2.8 zettabytes (2.8×10^{21} bytes), up from about 160 exabytes (160×10^{18} bytes) of data that were created and stored in 2006, and projected to reach 40 zettabytes by 2020. However, to make a black hole the size of the earth would

x_1	x_2	Class
0.1	1	1
0.15	0.2	2
0.48	0.6	3
0.1	0.6	1
0.2	0.15	2
0.5	0.55	3
0.2	1	1
0.3	0.25	2
0.52	0.6	3
0.3	0.6	1
0.4	0.2	2
0.52	0.5	3

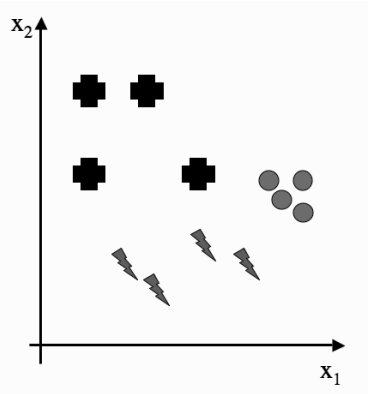

FIGURE 1.1 A set of datapoints as numerical values and as points plotted on a graph. It is easier for us to visualise data than to see it in a table, but if the data has more than three dimensions, we can't view it all at once.

FIGURE 1.2 Two views of the same two wind turbines (Te Apiti wind farm, Ashhurst, New Zealand) taken at an angle of about 30° to each other. The two-dimensional projections of three-dimensional objects hides information.

take a mass of about 40×10^{35} grams. So data would have to be so heavy that you couldn't possibly lift a data pen, let alone a computer before the section title were true! However, and more interestingly for machine learning, the same report that estimated the figure of 2.8 zettabytes ('*Big Data, Bigger Digital Shadows, and Biggest Growth in the Far East*' by John Gantz and David Reinsel and sponsored by EMC Corporation) also reported that while a quarter of this data could produce useful information, only around 3% of it was tagged, and less that 0.5% of it was actually used for analysis!

1.2 LEARNING

Before we delve too much further into the topic, let's step back and think about what learning actually is. The key concept that we will need to think about for our machines is learning from data, since data is what we have; terabytes of it, in some cases. However, it isn't too large a step to put that into human behavioural terms, and talk about learning from experience. Hopefully, we all agree that humans and other animals can display behaviours that we label as intelligent by learning from experience. Learning is what gives us flexibility in our life; the fact that we can adjust and adapt to new circumstances, and learn new tricks, no matter how old a dog we are! The important parts of animal learning for this book are remembering, adapting, and generalising: recognising that last time we were in this situation (saw this data) we tried out some particular action (gave this output) and it worked (was correct), so we'll try it again, or it didn't work, so we'll try something different. The last word, generalising, is about recognising similarity between different situations, so that things that applied in one place can be used in another. This is what makes learning useful, because we can use our knowledge in lots of different places.

Of course, there are plenty of other bits to intelligence, such as reasoning, and logical deduction, but we won't worry too much about those. We are interested in the most fundamental parts of intelligence—learning and adapting—and how we can model them in a computer. There has also been a lot of interest in making computers reason and deduce facts. This was the basis of most early Artificial Intelligence, and is sometimes known as symbolic processing because the computer manipulates symbols that reflect the environment. In contrast, machine learning methods are sometimes called subsymbolic because no symbols or symbolic manipulation are involved.

1.2.1 Machine Learning

Machine learning, then, is about making computers modify or adapt their actions (whether these actions are making predictions, or controlling a robot) so that these actions get more accurate, where accuracy is measured by how well the chosen actions reflect the correct ones. Imagine that you are playing Scrabble (or some other game) against a computer. You might beat it every time in the beginning, but after lots of games it starts beating you, until finally you never win. Either you are getting worse, or the computer is learning how to win at Scrabble. Having learnt to beat you, it can go on and use the same strategies against other players, so that it doesn't start from scratch with each new player; this is a form of generalisation.

It is only over the past decade or so that the inherent multi-disciplinarity of machine learning has been recognised. It merges ideas from neuroscience and biology, statistics, mathematics, and physics, to make computers learn. There is a fantastic existence proof that learning is possible, which is the bag of water and electricity (together with a few trace chemicals) sitting between your ears. In Section 3.1 we will have a brief peek inside and see

if there is anything we can borrow/steal in order to make machine learning algorithms. It turns out that there is, and neural networks have grown from exactly this, although even their own father wouldn't recognise them now, after the developments that have seen them reinterpreted as statistical learners. Another thing that has driven the change in direction of machine learning research is data mining, which looks at the extraction of useful information from massive datasets (by men with computers and pocket protectors rather than pickaxes and hard hats), and which requires efficient algorithms, putting more of the emphasis back onto computer science.

The computational complexity of the machine learning methods will also be of interest to us since what we are producing is algorithms. It is particularly important because we might want to use some of the methods on very large datasets, so algorithms that have high-degree polynomial complexity in the size of the dataset (or worse) will be a problem. The complexity is often broken into two parts: the complexity of training, and the complexity of applying the trained algorithm. Training does not happen very often, and is not usually time critical, so it can take longer. However, we often want a decision about a test point quickly, and there are potentially lots of test points when an algorithm is in use, so this needs to have low computational cost.

1.3 TYPES OF MACHINE LEARNING

In the example that started the chapter, your webpage, the aim was to predict what software a visitor to the website might buy based on information that you can collect. There are a couple of interesting things in there. The first is the data. It might be useful to know what software visitors have bought before, and how old they are. However, it is not possible to get that information from their web browser (even cookies can't tell you how old somebody is), so you can't use that information. Picking the variables that you want to use (which are called features in the jargon) is a very important part of finding good solutions to problems, and something that we will talk about in several places in the book. Equally, choosing how to process the data can be important. This can be seen in the example in the time of access. Your computer can store this down to the nearest millisecond, but that isn't very useful, since you would like to spot similar patterns between users. For this reason, in the example above I chose to quantise it down to one of the set `morning, afternoon, evening, night`; obviously I need to ensure that these times are correct for their time zones, too.

We are going to loosely define learning as meaning getting better at some task through practice. This leads to a couple of vital questions: how does the computer know whether it is getting better or not, and how does it know how to improve? There are several different possible answers to these questions, and they produce different types of machine learning. For now we will consider the question of knowing whether or not the machine is learning. We can tell the algorithm the correct answer for a problem so that it gets it right next time (which is what would happen in the webpage example, since we know what software the person bought). We hope that we only have to tell it a few right answers and then it can 'work out' how to get the correct answers for other problems (generalise). Alternatively, we can tell it whether or not the answer was correct, but not how to find the correct answer, so that it has to search for the right answer. A variant of this is that we give a score for the answer, according to how correct it is, rather than just a 'right or wrong' response. Finally, we might not have any correct answers; we just want the algorithm to find inputs that have something in common.

These different answers to the question provide a useful way to classify the different algorithms that we will be talking about:

Supervised learning A training set of examples with the correct responses (targets) is provided and, based on this training set, the algorithm generalises to respond correctly to all possible inputs. This is also called learning from exemplars.

Unsupervised learning Correct responses are not provided, but instead the algorithm tries to identify similarities between the inputs so that inputs that have something in common are categorised together. The statistical approach to unsupervised learning is known as density estimation.

Reinforcement learning This is somewhere between supervised and unsupervised learning. The algorithm gets told when the answer is wrong, but does not get told how to correct it. It has to explore and try out different possibilities until it works out how to get the answer right. Reinforcement learning is sometime called learning with a critic because of this monitor that scores the answer, but does not suggest improvements.

Evolutionary learning Biological evolution can be seen as a learning process: biological organisms adapt to improve their survival rates and chance of having offspring in their environment. We'll look at how we can model this in a computer, using an idea of fitness, which corresponds to a score for how good the current solution is.

The most common type of learning is supervised learning, and it is going to be the focus of the next few chapters. So, before we get started, we'll have a look at what it is, and the kinds of problems that can be solved using it.

1.4 SUPERVISED LEARNING

As has already been suggested, the webpage example is a typical problem for supervised learning. There is a set of data (the training data) that consists of a set of input data that has target data, which is the answer that the algorithm should produce, attached. This is usually written as a set of data $(\mathbf{x}_i, \mathbf{t}_i)$, where the inputs are $\mathbf{x}_i$, the targets are $\mathbf{t}_i$, and the i index suggests that we have lots of pieces of data, indexed by i running from 1 to some upper limit N. Note that the inputs and targets are written in boldface font to signify vectors, since each piece of data has values for several different features; the notation used in the book is described in more detail in Section 2.1. If we had examples of every possible piece of input data, then we could put them together into a big look-up table, and there would be no need for machine learning at all. The thing that makes machine learning better than that is generalisation: the algorithm should produce sensible outputs for inputs that weren't encountered during learning. This also has the result that the algorithm can deal with noise, which is small inaccuracies in the data that are inherent in measuring any real world process. It is hard to specify rigorously what generalisation means, but let's see if an example helps.

1.4.1 Regression

Suppose that I gave you the following datapoints and asked you to tell me the value of the output (which we will call y since it is not a target datapoint) when $x = 0.44$ (here, x, t, and y are not written in boldface font since they are scalars, as opposed to vectors).

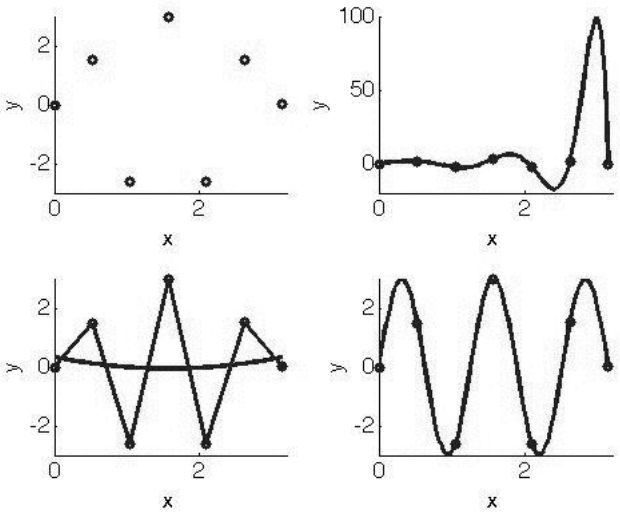

FIGURE 1.3 *Top left:* A few datapoints from a sample problem. *Bottom left:* Two possible ways to predict the values between the known datapoints: connecting the points with straight lines, or using a cubic approximation (which in this case misses all of the points). *Top and bottom right:* Two more complex approximators (see the text for details) that pass through the points, although the lower one is rather better than the top.

x	t
0	0
0.5236	1.5
1.0472	-2.5981
1.5708	3.0
2.0944	-2.5981
2.6180	1.5
3.1416	0

Since the value $x = 0.44$ isn't in the examples given, you need to find some way to predict what value it has. You assume that the values come from some sort of function, and try to find out what the function is. Then you'll be able to give the output value y for any given value of x. This is known as a regression problem in statistics: fit a mathematical function describing a curve, so that the curve passes as close as possible to all of the datapoints. It is generally a problem of function approximation or interpolation, working out the value between values that we know.

The problem is how to work out what function to choose. Have a look at Figure 1.3. The top-left plot shows a plot of the 7 values of x and y in the table, while the other plots show different attempts to fit a curve through the datapoints. The bottom-left plot shows two possible answers found by using straight lines to connect up the points, and also what happens if we try to use a cubic function (something that can be written as $ax^3 + bx^2 + cx + d = 0$). The top-right plot shows what happens when we try to match the function using a different polynomial, this time of the form $ax^{10} + bx^9 + \ldots + jx + k = 0$,

and finally the bottom-right plot shows the function $y = 3\sin(5x)$. Which of these functions would you choose?

The straight-line approximation probably isn't what we want, since it doesn't tell us much about the data. However, the cubic plot on the same set of axes is terrible: it doesn't get anywhere near the datapoints. What about the plot on the top-right? It looks like it goes through all of the datapoints exactly, but it is very wiggly (look at the value on the y-axis, which goes up to 100 instead of around three, as in the other figures). In fact, the data were made with the sine function plotted on the bottom-right, so that is the correct answer in this case, but the algorithm doesn't know that, and to it the two solutions on the right both look equally good. The only way we can tell which solution is better is to test how well they generalise. We pick a value that is between our datapoints, use our curves to predict its value, and see which is better. This will tell us that the bottom-right curve is better in the example.

So one thing that our machine learning algorithms can do is interpolate between datapoints. This might not seem to be intelligent behaviour, or even very difficult in two dimensions, but it is rather harder in higher dimensional spaces. The same thing is true of the other thing that our algorithms will do, which is classification—grouping examples into different classes—which is discussed next. However, the algorithms are learning by our definition if they adapt so that their performance improves, and it is surprising how often real problems that we want to solve can be reduced to classification or regression problems.

1.4.2 Classification

The classification problem consists of taking input vectors and deciding which of N classes they belong to, based on training from exemplars of each class. The most important point about the classification problem is that it is discrete — each example belongs to precisely one class, and the set of classes covers the whole possible output space. These two constraints are not necessarily realistic; sometimes examples might belong partially to two different classes. There are fuzzy classifiers that try to solve this problem, but we won't be talking about them in this book. In addition, there are many places where we might not be able to categorise every possible input. For example, consider a vending machine, where we use a neural network to learn to recognise all the different coins. We train the classifier to recognise all New Zealand coins, but what if a British coin is put into the machine? In that case, the classifier will identify it as the New Zealand coin that is closest to it in appearance, but this is not really what is wanted: rather, the classifier should identify that it is not one of the coins it was trained on. This is called novelty detection. For now we'll assume that we will not receive inputs that we cannot classify accurately.

Let's consider how to set up a coin classifier. When the coin is pushed into the slot, the machine takes a few measurements of it. These could include the diameter, the weight, and possibly the shape, and are the features that will generate our input vector. In this case, our input vector will have three elements, each of which will be a number showing the measurement of that feature (choosing a number to represent the shape would involve an encoding, for example that 1=circle, 2=hexagon, etc.). Of course, there are many other features that we could measure. If our vending machine included an atomic absorption spectroscope, then we could estimate the density of the material and its composition, or if it had a camera, we could take a photograph of the coin and feed that image into the classifier. The question of which features to choose is not always an easy one. We don't want to use too many inputs, because that will make the training of the classifier take longer (and also, as the number of input dimensions grows, the number of datapoints required increases

FIGURE 1.4 The New Zealand coins.

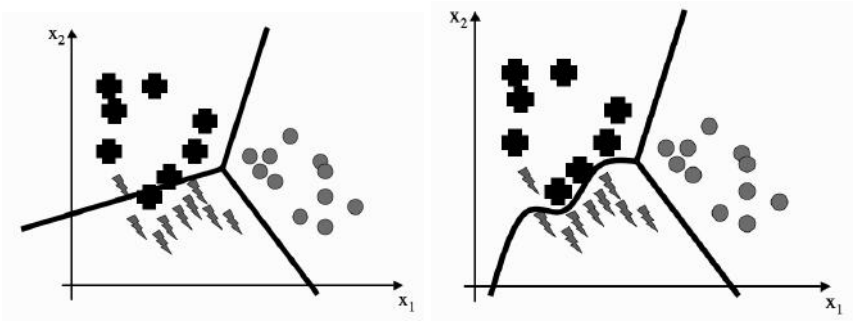

FIGURE 1.5 *Left:* A set of straight line decision boundaries for a classification problem. *Right:* An alternative set of decision boundaries that separate the plusses from the lightening strikes better, but requires a line that isn't straight.

faster; this is known as the curse of dimensionality and will be discussed in Section 2.1.2), but we need to make sure that we can reliably separate the classes based on those features. For example, if we tried to separate coins based only on colour, we wouldn't get very far, because the 20¢ and 50¢ coins are both silver and the $1 and $2 coins both bronze. However, if we use colour and diameter, we can do a pretty good job of the coin classification problem for NZ coins. There are some features that are entirely useless. For example, knowing that the coin is circular doesn't tell us anything about NZ coins, which are all circular (see Figure 1.4). In other countries, though, it could be very useful.

The methods of performing classification that we will see during this book are very different in the ways that they learn about the solution; in essence they aim to do the same thing: find decision boundaries that can be used to separate out the different classes. Given the features that are used as inputs to the classifier, we need to identify some values of those features that will enable us to decide which class the current input is in. Figure 1.5 shows a set of 2D inputs with three different classes shown, and two different decision boundaries; on the left they are straight lines, and are therefore simple, but don't categorise as well as the non-linear curve on the right.

Now that we have seen these two types of problem, let's take a look at the whole process of machine learning from the practitioner's viewpoint.

1.5 THE MACHINE LEARNING PROCESS

This section assumes that you have some problem that you are interested in using machine learning on, such as the coin classification that was described previously. It briefly examines the process by which machine learning algorithms can be selected, applied, and evaluated for the problem.

Data Collection and Preparation Throughout this book we will be in the fortunate position of having datasets readily available for downloading and using to test the algorithms. This is, of course, less commonly the case when the desire is to learn about some new problem, when either the data has to be collected from scratch, or at the very least, assembled and prepared. In fact, if the problem is completely new, so that appropriate data can be chosen, then this process should be merged with the next step of feature selection, so that only the required data is collected. This can typically be done by assembling a reasonably small dataset with all of the features that you believe might be useful, and experimenting with it before choosing the best features and collecting and analysing the full dataset.

Often the difficulty is that there is a large amount of data that *might* be relevant, but it is hard to collect, either because it requires many measurements to be taken, or because they are in a variety of places and formats, and merging it appropriately is difficult, as is ensuring that it is clean; that is, it does not have significant errors, missing data, etc.

For supervised learning, target data is also needed, which can require the involvement of experts in the relevant field and significant investments of time.

Finally, the quantity of data needs to be considered. Machine learning algorithms need significant amounts of data, preferably without too much noise, but with increased dataset size comes increased computational costs, and the sweet spot at which there is enough data without excessive computational overhead is generally impossible to predict.

Feature Selection An example of this part of the process was given in Section 1.4.2 when we looked at possible features that might be useful for coin recognition. It consists of identifying the features that are most useful for the problem under examination. This invariably requires prior knowledge of the problem and the data; our common sense was used in the coins example above to identify some potentially useful features and to exclude others.

As well as the identification of features that are useful for the learner, it is also necessary that the features can be collected without significant expense or time, and that they are robust to noise and other corruption of the data that may arise in the collection process.

Algorithm Choice Given the dataset, the choice of an appropriate algorithm (or algorithms) is what this book should be able to prepare you for, in that the knowledge of the underlying principles of each algorithm and examples of their use is precisely what is required for this.

Parameter and Model Selection For many of the algorithms there are parameters that have to be set manually, or that require experimentation to identify appropriate values. These requirements are discussed at the appropriate points of the book.

Training Given the dataset, algorithm, and parameters, training should be simply the use of computational resources in order to build a model of the data in order to predict the outputs on new data.

Evaluation Before a system can be deployed it needs to be tested and evaluated for accuracy on data that it was not trained on. This can often include a comparison with human experts in the field, and the selection of appropriate metrics for this comparison.

1.6 A NOTE ON PROGRAMMING

This book is aimed at helping you understand and use machine learning algorithms, and that means writing computer programs. The book contains algorithms in both pseudocode, and as fragments of Python programs based on NumPy (Appendix A provides an introduction to both Python and NumPy for the beginner), and the website provides complete working code for all of the algorithms.

Understanding how to use machine learning algorithms is fine in theory, but without testing the programs on data, and seeing what the parameters do, you won't get the complete picture. In general, writing the code for yourself is always the best way to check that you understand what the algorithm is doing, and finding the unexpected details.

Unfortunately, debugging machine learning code is even harder than general debugging – it is quite easy to make a program that compiles and runs, but just doesn't seem to actually learn. In that case, you need to start testing the program carefully. However, you can quickly get frustrated with the fact that, because so many of the algorithms are stochastic, the results are not repeatable anyway. This can be temporarily avoided by setting the random number seed, which has the effect of making the random number generator follow the same pattern each time, as can be seen in the following example of running code at the Python command line (marked as >>>), where the 10 numbers that appear after the seed is set are the same in both cases, and would carry on the same forever (there is more about the pseudo-random numbers that computers generate in Section 15.1.1):

```
>>> import numpy as np
>>> np.random.seed(4)
>>> np.random.rand(10)
array([ 0.96702984,  0.54723225,  0.97268436,  0.71481599,  0.69772882,
        0.2160895 ,  0.97627445,  0.00623026,  0.25298236,  0.43479153])
>>> np.random.rand(10)
array([ 0.77938292,  0.19768507,  0.86299324,  0.98340068,  0.16384224,
        0.59733394,  0.0089861 ,  0.38657128,  0.04416006,  0.95665297])
>>> np.random.seed(4)
>>> np.random.rand(10)
array([ 0.96702984,  0.54723225,  0.97268436,  0.71481599,  0.69772882,
        0.2160895 ,  0.97627445,  0.00623026,  0.25298236,  0.43479153])
```

This way, on each run the randomness will be avoided, and the parameters will all be the same.

Another thing that is useful is the use of 2D toy datasets, where you can plot things, since you can see whether or not something unexpected is going on. In addition, these

datasets can be made very simple, such as separable by a straight line (we'll see more of this in Chapter 3) so that you can see whether it deals with simple cases, at least.

Another way to 'cheat' temporarily is to include the target as one of the inputs, so that the algorithm really has no excuse for getting the wrong answer.

Finally, having a reference program that works and that you can compare is also useful, and I hope that the code on the book website will help people get out of unexpected traps and strange errors.

1.7 A ROADMAP TO THE BOOK

As far as possible, this book works from general to specific and simple to complex, while keeping related concepts in nearby chapters. Given the focus on algorithms and encouraging the use of experimentation rather than starting from the underlying statistical concepts, the book starts with some older, and reasonably simple algorithms, which are examples of supervised learning.

Chapter 2 follows up many of the concepts in this introductory chapter in order to highlight some of the overarching ideas of machine learning and thus the data requirements of it, as well as providing some material on basic probability and statistics that will not be required by all readers, but is included for completeness.

Chapters 3, 4, and 5 follow the main historical sweep of supervised learning using neural networks, as well as introducing concepts such as interpolation. They are followed by chapters on dimensionality reduction (Chapter 6) and the use of probabilistic methods like the EM algorithm and nearest neighbour methods (Chapter 7). The idea of optimal decision boundaries and kernel methods are introduced in Chapter 8, which focuses on the Support Vector Machine and related algorithms.

One of the underlying methods for many of the preceding algorithms, optimisation, is surveyed briefly in Chapter 9, which then returns to some of the material in Chapter 4 to consider the Multi-layer Perceptron purely from the point of view of optimisation. The chapter then continues by considering search as the discrete analogue of optimisation. This leads naturally into evolutionary learning including genetic algorithms (Chapter 10), reinforcement learning (Chapter 11), and tree-based learners (Chapter 12) which are search-based methods. Methods to combine the predictions of many learners, which are often trees, are described in Chapter 13.

The important topic of unsupervised learning is considered in Chapter 14, which focuses on the Self-Organising Feature Map; many unsupervised learning algorithms are also presented in Chapter 6.

The remaining four chapters primarily describe more modern, and statistically based, approaches to machine learning, although not all of the algorithms are completely new: following an introduction to Markov Chain Monte Carlo techniques in Chapter 15 the area of Graphical Models is surveyed, with comparatively old algorithms such as the Hidden Markov Model and Kalman Filter being included along with particle filters and Bayesian networks. The ideas behind Deep Belief Networks are given in Chapter 17, starting from the historical idea of symmetric networks with the Hopfield network. An introduction to Gaussian Processes is given in Chapter 18.

Finally, an introduction to Python and NumPy is given in Appendix A, which should be sufficient to enable readers to follow the code descriptions provided in the book and use the code supplied on the book website, assuming that they have some programming experience in any programming language.

I would suggest that Chapters 2 to 4 contain enough introductory material to be essential

for anybody looking for an introduction to machine learning ideas. For an introductory one semester course I would follow them with Chapters 6 to 8, and then use the second half of Chapter 9 to introduce Chapters 10 and 11, and then Chapter 14.

A more advanced course would certainly take in Chapters 13 and 15 to 18 along with the optimisation material in Chapter 9.

I have attempted to make the material reasonably self-contained, with the relevant mathematical ideas either included in the text at the appropriate point, or with a reference to where that material is covered. This means that the reader with some prior knowledge will certainly find some parts can be safely ignored or skimmed without loss.

FURTHER READING

For a different (more statistical and example-based) take on machine learning, look at:

- Chapter 1 of T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*, 2nd edition, Springer, Berlin, Germany, 2008.

Other texts that provide alternative views of similar material include:

- Chapter 1 of R.O. Duda, P.E. Hart, and D.G. Stork. *Pattern Classification*, 2nd edition, Wiley-Interscience, New York, USA, 2001.
- Chapter 1 of S. Haykin. *Neural Networks: A Comprehensive Foundation*, 2nd edition, Prentice-Hall, New Jersey, USA, 1999.

Preliminaries

This chapter has two purposes: to present some of the overarching important concepts of machine learning, and to see how some of the basic ideas of data processing and statistics arise in machine learning. One of the most useful ways to break down the effects of learning, which is to put it in terms of the statistical concepts of bias and variance, is given in Section 2.5, following on from a section where those concepts are introduced for the beginner.

2.1 SOME TERMINOLOGY

We start by considering some of the terminology that we will use throughout the book; we've already seen a bit of it in the Introduction. We will talk about inputs and input vectors for our learning algorithms. Likewise, we will talk about the outputs of the algorithm. The inputs are the data that is fed into the algorithm. In general, machine learning algorithms all work by taking a set of input values, producing an output (answer) for that input vector, and then moving on to the next input. The input vector will typically be several real numbers, which is why it is described as a vector: it is written down as a series of numbers, e.g., $(0.2, 0.45, 0.75, -0.3)$. The size of this vector, i.e., the number of elements in the vector, is called the dimensionality of the input. This is because if we were to plot the vector as a point, we would need one dimension of space for each of the different elements of the vector, so that the example above has 4 dimensions. We will talk about this more in Section 2.1.1.

We will often write equations in vector and matrix notation, with lowercase boldface letters being used for vectors and uppercase boldface letters for matrices. A vector $\mathbf{x}$ has elements $(x_1, x_2, \ldots, x_m)$. We will use the following notation in the book:

Inputs An input vector is the data given as one input to the algorithm. Written as $\mathbf{x}$, with elements x_i, where i runs from 1 to the number of input dimensions, m.

Weights w_{ij}, are the weighted connections between nodes i and j. For neural networks these weights are analogous to the synapses in the brain. They are arranged into a matrix $\mathbf{W}$.

Outputs The output vector is $\mathbf{y}$, with elements y_j, where j runs from 1 to the number of output dimensions, n. We can write $\mathbf{y}(\mathbf{x}, \mathbf{W})$ to remind ourselves that the output depends on the inputs to the algorithm and the current set of weights of the network.

Targets The target vector $\mathbf{t}$, with elements t_j, where j runs from 1 to the number of output dimensions, n, are the extra data that we need for supervised learning, since they provide the 'correct' answers that the algorithm is learning about.

Activation Function For neural networks, $g(\cdot)$ is a mathematical function that describes the firing of the neuron as a response to the weighted inputs, such as the threshold function described in Section 3.1.2.

Error E, a function that computes the inaccuracies of the network as a function of the outputs $\mathbf{y}$ and targets $\mathbf{t}$.

2.1.1 Weight Space

When working with data it is often useful to be able to plot it and look at it. If our data has only two or three input dimensions, then this is pretty easy: we use the x-axis for feature 1, the y-axis for feature 2, and the z-axis for feature 3. We then plot the positions of the input vectors on these axes. The same thing can be extended to as many dimensions as we like provided that we don't actually want to look at it in our 3D world. Even if we have 200 input dimensions (that is, 200 elements in each of our input vectors) then we can try to imagine it plotted by using 200 axes that are all mutually orthogonal (that is, at right angles to each other). One of the great things about computers is that they aren't constrained in the same way we are—ask a computer to hold a 200-dimensional array and it does it. Provided that you get the algorithm right (always the difficult bit!), then the computer doesn't know that 200 dimensions is harder than 2 for us humans.

We can look at projections of the data into our 3D world by plotting just three of the features against each other, but this is usually rather confusing: things can look very close together in your chosen three axes, but can be a very long way apart in the full set. You've experienced this in your 2D view of the 3D world; Figure 1.2 shows two different views of some wind turbines. The two turbines appear to be very close together from one angle, but are obviously separate from another.

As well as plotting datapoints, we can also plot anything else that we feel like. In particular, we can plot some of the parameters of a machine learning algorithm. This is particularly useful for neural networks (which we will start to see in the next chapter) since the parameters of a neural network are the values of a set of weights that connect the neurons to the inputs. There is a schematic of a neural network on the left of Figure 2.1, showing the inputs on the left, and the neurons on the right. If we treat the weights that get fed into one of the neurons as a set of coordinates in what is known as weight space, then we can plot them. We think about the weights that connect into a particular neuron, and plot the strengths of the weights by using one axis for each weight that comes into the neuron, and plotting the position of the neuron as the location, using the value of w_1 as the position on the 1st axis, the value of w_2 on the 2nd axis, etc. This is shown on the right of Figure 2.1.

Now that we have a space in which we can talk about how close together neurons and inputs are, since we can imagine positioning neurons and inputs in the same space by plotting the position of each neuron as the location where its weights say it should be. The two spaces will have the same dimension (providing that we don't use a bias node (see Section 3.3.2), otherwise the weight space will have one extra dimension) so we can plot the position of neurons in the input space. This gives us a different way of learning, since by changing the weights we are changing the location of the neurons in this weight space. We can measure distances between inputs and neurons by computing the Euclidean distance, which in two dimensions can be written as:

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}. \tag{2.1}$$

So we can use the idea of neurons and inputs being 'close together' in order to decide

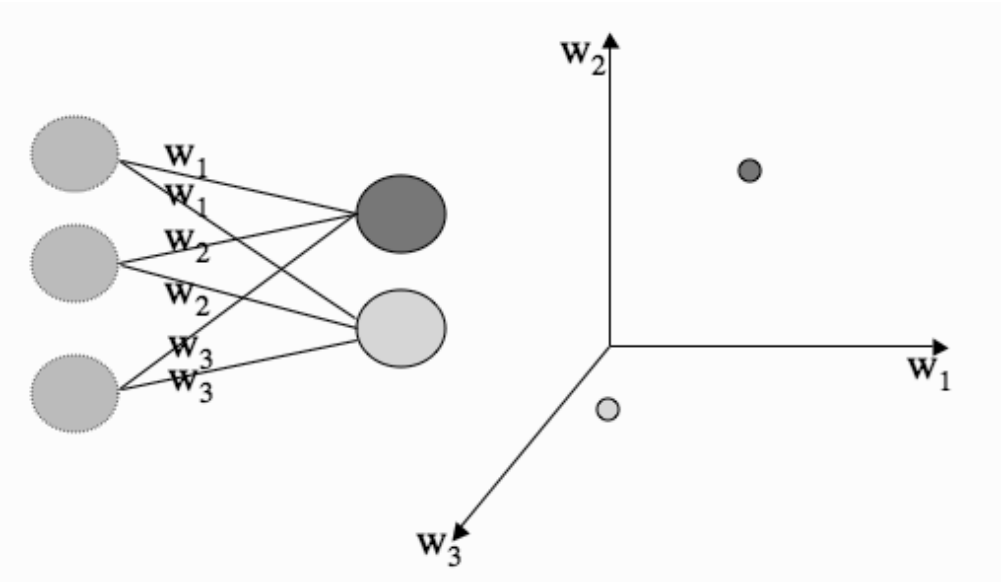

FIGURE 2.1 The position of two neurons in weight space. The labels on the network refer to the dimension in which that weight is plotted, not its value.

when a neuron should fire and when it shouldn't. If the neuron is close to the input in this sense then it should fire, and if it is not close then it shouldn't. This picture of weight space can be helpful for understanding another important concept in machine learning, which is what effect the number of input dimensions can have. The input vector is telling us everything we know about that example, and usually we don't know enough about the data to know what is useful and what is not (think back to the coin classification example in Section 1.4.2), so it might seem sensible to include all of the information that we can get, and let the algorithm sort out for itself what it needs. Unfortunately, we are about to see that doing this comes at a significant cost.

2.1.2 The Curse of Dimensionality

The curse of dimensionality is a very strong name, so you can probably guess that it is a bit of a problem. The essence of the curse is the realisation that as the number of dimensions increases, the volume of the unit hypersphere does not increase with it. The unit hypersphere is the region we get if we start at the origin (the centre of our coordinate system) and draw all the points that are distance 1 away from the origin. In 2 dimensions we get a circle of radius 1 around $(0, 0)$ (drawn in Figure 2.2), and in 3D we get a sphere around $(0, 0, 0)$ (Figure 2.3). In higher dimensions, the sphere becomes a hypersphere. The following table shows the size of the unit hypersphere for the first few dimensions, and the graph in Figure 2.4 shows the same thing, but also shows clearly that as the number of dimensions tends to infinity, so the volume of the hypersphere tends to zero.

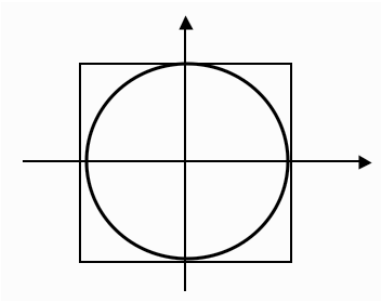

FIGURE 2.2 The unit circle in 2D with its bounding box.

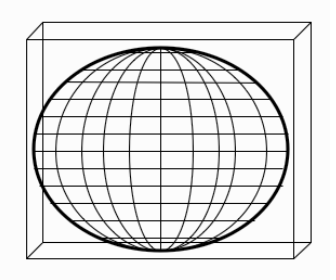

FIGURE 2.3 The unit sphere in 3D with its bounding cube. The sphere does not reach as far into the corners as the circle does, and this gets more noticeable as the number of dimensions increases.

Dimension	Volume
1	2.0000
2	3.1416
3	4.1888
4	4.9348
5	5.2636
6	5.1677
7	4.7248
8	4.0587
9	3.2985
10	2.5502

At first sight this seems completely counterintuitive. However, think about enclosing the hypersphere in a box of width 2 (between -1 and 1 along each axis), so that the box just touches the sides of the hypersphere. For the circle, almost all of the area inside the box is included in the circle, except for a little bit at each corner (see Figure 2.2) The same is true in 3D (Figure 2.3), but if we think about the 100-dimensional hypersphere (not necessarily something you want to imagine), and follow the diagonal line from the origin out to one of the corners of the box, then we intersect the boundary of the hypersphere when all the coordinates are 0.1. The remaining 90% of the line inside the box is outside the hypersphere, and so the volume of the hypersphere is obviously shrinking as the number of dimensions grows. The graph in Figure 2.4 shows that when the number of dimensions is above about 20, the volume is effectively zero. It was computed using the formula for the volume of the hypersphere of dimension n as $v_n = (2\pi/n)v_{n-2}$. So as soon as $n > 2\pi$, the volume starts to shrink.

The curse of dimensionality will apply to our machine learning algorithms because as the number of input dimensions gets larger, we will need more data to enable the algorithm to generalise sufficiently well. Our algorithms try to separate data into classes based on the features; therefore as the number of features increases, so will the number of datapoints we need. For this reason, we will often have to be careful about what information we give to the algorithm, meaning that we need to understand something about the data in advance.

Regardless of how many input dimensions there are, the point of machine learning is to

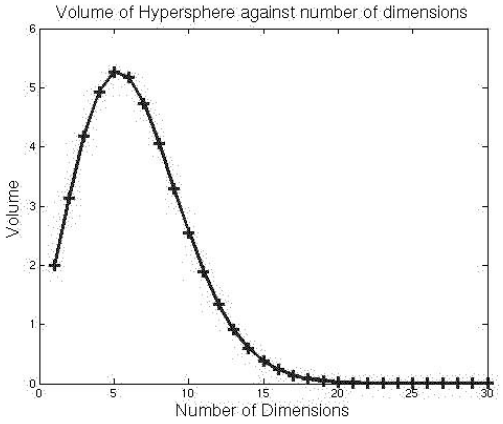

FIGURE 2.4 The volume of the unit hypersphere for different numbers of dimensions.

make predictions on data inputs. In the next section we consider how to evaluate how well an algorithm actually achieves this.

2.2 KNOWING WHAT YOU KNOW: TESTING MACHINE LEARNING ALGORITHMS

The purpose of learning is to get better at predicting the outputs, be they class labels or continuous regression values. The only real way to know how successfully the algorithm has learnt is to compare the predictions with known target labels, which is how the training is done for supervised learning. This suggests that one thing you can do is just to look at the error that the algorithm makes on the training set.

However, we want the algorithms to generalise to examples that were not seen in the training set, and we obviously can't test this by using the training set. So we need some different data, a test set, to test it on as well. We use this test set of (input, target) pairs by feeding them into the network and comparing the predicted output with the target, but we don't modify the weights or other parameters for them: we use them to decide how well the algorithm has learnt. The only problem with this is that it reduces the amount of data that we have available for training, but that is something that we will just have to live with.

2.2.1 Overfitting

Unfortunately, things are a little bit more complicated than that, since we might also want to know how well the algorithm is generalising as it learns: we need to make sure that we do enough training that the algorithm generalises well. In fact, there is at least as much danger in over-training as there is in under-training. The number of degrees of variability in most machine learning algorithms is huge — for a neural network there are lots of weights, and each of them can vary. This is undoubtedly more variation than there is in the function we are learning, so we need to be careful: if we train for too long, then we will overfit the data, which means that we have learnt about the noise and inaccuracies in the data as well as the actual function. Therefore, the model that we learn will be much too complicated, and won't be able to generalise.

Figure 2.5 shows this by plotting the predictions of some algorithm (as the curve) at

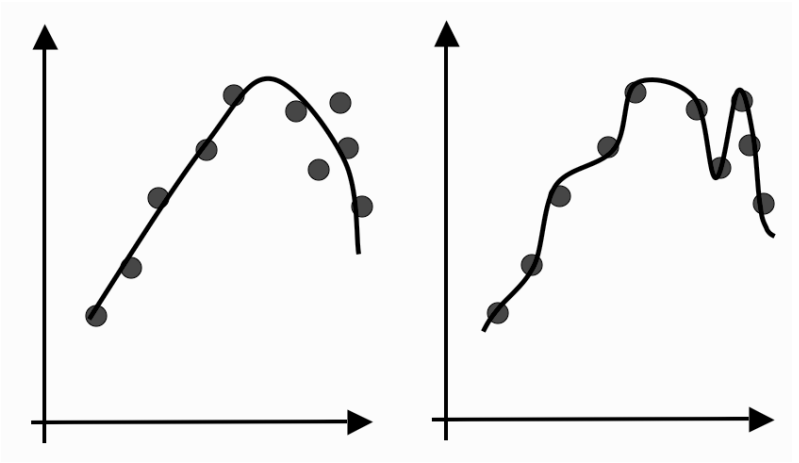

FIGURE 2.5 The effect of overfitting is that rather than finding the generating function (as shown on the left), the neural network matches the inputs perfectly, including the noise in them (on the right). This reduces the generalisation capabilities of the network.

two different points in the learning process. On the left of the figure the curve fits the overall trend of the data well (it has generalised to the underlying general function), but the training error would still not be that close to zero since it passes near, but not through, the training data. As the network continues to learn, it will eventually produce a much more complex model that has a lower training error (close to zero), meaning that it has memorised the training examples, including any noise component of them, so that is has overfitted the training data.

We want to stop the learning process before the algorithm overfits, which means that we need to know how well it is generalising at each timestep. We can't use the training data for this, because we wouldn't detect overfitting, but we can't use the testing data either, because we're saving that for the final tests. So we need a third set of data to use for this purpose, which is called the validation set because we're using it to validate the learning so far. This is known as cross-validation in statistics. It is part of model selection: choosing the right parameters for the model so that it generalises as well as possible.

2.2.2 Training, Testing, and Validation Sets

We now need three sets of data: the training set to actually train the algorithm, the validation set to keep track of how well it is doing as it learns, and the test set to produce the final results. This is becoming expensive in data, especially since for supervised learning it all has to have target values attached (and even for unsupervised learning, the validation and test sets need targets so that you have something to compare to), and it is not always easy to get accurate labels (which may well be why you want to learn about the data). The area of semi-supervised learning attempts to deal with this need for large amounts of labelled data; see the Further Reading section for some references.

Clearly, each algorithm is going to need some reasonable amount of data to learn from (precise needs vary, but the more data the algorithm sees, the more likely it is to have seen examples of each possible type of input, although more data also increases the computational time to learn). However, the same argument can be used to argue that the validation and

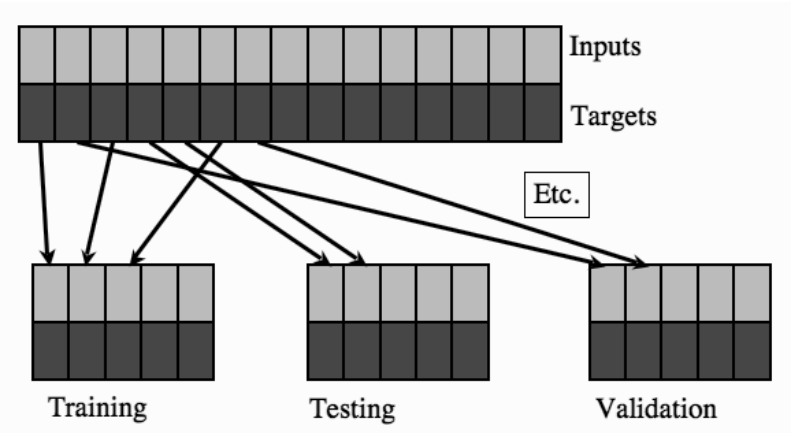

FIGURE 2.6 The dataset is split into different sets, some for training, some for validation, and some for testing.

test sets should also be reasonably large. Generally, the exact proportion of training to testing to validation data is up to you, but it is typical to do something like 50:25:25 if you have plenty of data, and 60:20:20 if you don't. How you do the splitting can also matter. Many datasets are presented with the first set of datapoints being in class 1, the next in class 2, and so on. If you pick the first few points to be the training set, the next the test set, etc., then the results are going to be pretty bad, since the training did not see all the classes. This can be dealt with by randomly reordering the data first, or by assigning each datapoint randomly to one of the sets, as is shown in Figure 2.6.

If you are really short of training data, so that if you have a separate validation set there is a worry that the algorithm won't be sufficiently trained; then it is possible to perform leave-some-out, multi-fold cross-validation. The idea is shown in Figure 2.7. The dataset is randomly partitioned into K subsets, and one subset is used as a validation set, while the algorithm is trained on all of the others. A different subset is then left out and a new model is trained on that subset, repeating the same process for all of the different subsets. Finally, the model that produced the lowest validation error is tested and used. We've traded off data for computation time, since we've had to train K different models instead of just one. In the most extreme case of this there is leave-one-out cross-validation, where the algorithm is validated on just one piece of data, training on all of the rest.

2.2.3 The Confusion Matrix

Regardless of how much data we use to test the trained algorithm, we still need to work out whether or not the result is good. We will look here at a method that is suitable for classification problems that is known as the confusion matrix. For regression problems things are more complicated because the results are continuous, and so the most common thing to use is the sum-of-squares error that we will use to drive the training in the following chapters. We will see these methods being used as we look at examples.

The confusion matrix is a nice simple idea: make a square matrix that contains all the possible classes in both the horizontal and vertical directions and list the classes along the top of a table as the predicted outputs, and then down the left-hand side as the targets. So for example, the element of the matrix at (i, j) tells us how many input patterns were put

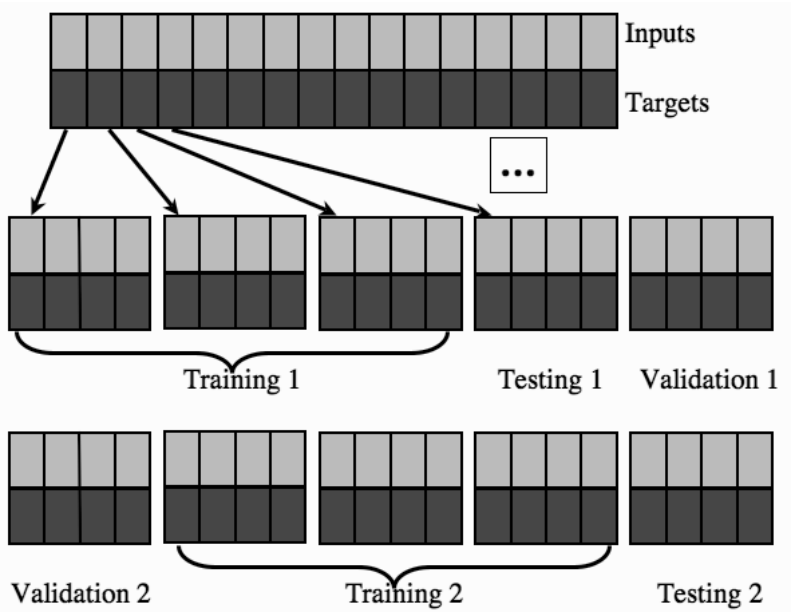

FIGURE 2.7 Leave-some-out, multi-fold cross-validation gets around the problem of data shortage by training many models. It works by splitting the data into sets, training a model on most sets and holding one out for validation (and another for testing). Different models are trained with different sets being held out.

into class i in the targets, but class j by the algorithm. Anything on the leading diagonal (the diagonal that starts at the top left of the matrix and runs down to the bottom right) is a correct answer. Suppose that we have three classes: C_1, C_2, and C_3. Now we count the number of times that the output was class C_1 when the target was C_1, then when the target was C_2, and so on until we've filled in the table:

	Outputs		
	C_1	C_2	C_3
C_1	5	1	0
C_2	1	4	1
C_3	2	0	4

This table tells us that, for the three classes, most examples were classified correctly, but two examples of class C_3 were misclassified as C_1, and so on. For a small number of classes this is a nice way to look at the outputs. If you just want one number, then it is possible to divide the sum of the elements on the leading diagonal by the sum of all of the elements in the matrix, which gives the fraction of correct responses. This is known as the accuracy, and we are about to see that it is not the last word in evaluating the results of a machine learning algorithm.

2.2.4 Accuracy Metrics

We can do more to analyse the results than just measuring the accuracy. If you consider the possible outputs of the classes, then they can be arranged in a simple chart like this

(where a true positive is an observation correctly put into class 1, while a false positive is an observation incorrectly put into class 1, while negative examples (both true and false) are those put into class 2):

True Positives	False Positives
False Negatives	True Negatives

The entries on the leading diagonal of this chart are correct and those off the diagonal are wrong, just as with the confusion matrix. Note, however, that this chart and the concepts of false positives, etc., are based on binary classification.

Accuracy is then defined as the sum of the number of true positives and true negatives divided by the total number of examples (where # means 'number of', and TP stands for True Positive, etc.):

$$\text{Accuracy} = \frac{\#TP + \#FP}{\#TP + \#FP + \#TN + \#FN}. \tag{2.2}$$

The problem with accuracy is that it doesn't tell us everything about the results, since it turns four numbers into just one. There are two complementary pairs of measurements that can help us to interpret the performance of a classifier, namely sensitivity and specificity, and precision and recall. Their definitions are shown next, followed by some explanation.

$$\text{Sensitivity} = \frac{\#TP}{\#TP + \#FN} \tag{2.3}$$

$$\text{Specificity} = \frac{\#TN}{\#TN + \#FP} \tag{2.4}$$

$$\text{Precision} = \frac{\#TP}{\#TP + \#FP} \tag{2.5}$$

$$\text{Recall} = \frac{\#TP}{\#TP + \#FN} \tag{2.6}$$

Sensitivity (also known as the true positive rate) is the ratio of the number of correct positive examples to the number classified as positive, while specificity is the same ratio for negative examples. Precision is the ratio of correct positive examples to the number of actual positive examples, while recall is the ratio of the number of correct positive examples out of those that were classified as positive, which is the same as sensitivity. If you look at the chart again you can see that sensitivity and specificity sum the columns for the denominator, while precision and recall sum the first column and the first row, and so miss out some information about how well the learner does on the negative examples.

Together, either of these pairs of measures gives more information than just the accuracy. If you consider precision and recall, then you can see that they are to some extent inversely related, in that if the number of false positives increases (meaning that the algorithm is using a broader definition of that class), then the number of false negatives often decreases, and vice versa. They can be combined to give a single measure, the F_1 measure, which can be written in terms of precision and recall as:

$$F_1 = 2\frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}} \tag{2.7}$$

FIGURE 2.8 An example of an ROC curve. The diagonal line represents exactly chance, so anything above the line is better than chance, and the further from the line, the better. Of the two curves shown, the one that is further away from the diagonal line would represent a more accurate method.

and in terms of the numbers of false positives, etc. (from which it can be seen that it computes the mean of the false examples) as:

$$F_1 = \frac{\#TP}{\#TP + (\#FN + \#FP)/2}. \tag{2.8}$$

2.2.5 The Receiver Operator Characteristic (ROC) Curve

Since we can use these measures to evaluate a particular classifier, we can also compare classifiers – either the same classifier with different learning parameters, or completely different classifiers. In this case, the Receiver Operator Characteristic curve (almost always known just as the ROC curve) is useful. This is a plot of the percentage of true positives on the y axis against false positives on the x axis; an example is shown in Figure 2.8. A single run of a classifier produces a single point on the ROC plot, and a perfect classifier would be a point at $(0, 1)$ (100% true positives, 0% false positives), while the anti-classifier that got everything wrong would be at (1,0); so the closer to the top-left-hand corner the result of a classifier is, the better the classifier has performed. Any classifier that sits on the diagonal line from (0,0) to (1,1) behaves exactly at the chance level (assuming that the positive and negative classes are equally common) and so presumably a lot of learning effort is wasted since a fair coin would do just as well.

In order to compare classifiers, or choices of parameters settings for the same classifier, you could just compute the point that is furthest from the 'chance' line along the diagonal. However, it is normal to compute the area under the curve (AUC) instead. If you only have one point for each classifier, the curve is the trapezoid that runs from (0,0) up to the point and then from there to (1,1). If there are more points (based on more runs of the classifier, such as trained and/or tested on different datasets), then they are just included in order along the diagonal line.

The key to getting a curve rather than a point on the ROC curve is to use cross-validation. If you use 10-fold cross-validation, then you have 10 classifiers, with 10 different

test sets, and you also have the 'ground truth' labels. The true labels can be used to produce a ranked list of the different cross-validation-trained results, which can be used to specify a curve through the 10 datapoints on the ROC curve that correspond to the results of this classifier. By producing an ROC curve for each classifier it is possible to compare their results.

2.2.6 Unbalanced Datasets

Note that for the accuracy we have implicitly assumed that there are the same number of positive and negative examples in the dataset (which is known as a balanced dataset). However, this is often not true (this can potentially cause problems for the learners as well, as we shall see later in the book). In the case where it is not, we can compute the balanced accuracy as the sum of sensitivity and specificity divided by 2. However, a more correct measure is Matthew's Correlation Coefficient, which is computed as:

$$MCC = \frac{\#TP \times \#TN - \#FP \times \#FN}{\sqrt{(\#TP + \#FP)(\#TP + \#FN)(\#TN + \#FP)(\#TN + \#FN)}} \tag{2.9}$$

If any of the brackets in the denominator are 0, then the whole of the denominator is set to 1. This provides a balanced accuracy computation.

As a final note on these methods of evaluation, if there are more than two classes and it is useful to distinguish the different types of error, then the calculations get a little more complicated, since instead of one set of false positives and one set of false negatives, you have some for each class. In this case, specificity and recall are not the same. However, it is possible to create a set of results, where you use one class as the positives and everything else as the negatives, and repeat this for each of the different classes.

2.2.7 Measurement Precision

There is a different way to evaluate the accuracy of a learning system, which unfortunately also uses the word precision, although with a different meaning. The concept here is to treat the machine learning algorithm as a measurement system. We feed in inputs and look at the outputs that we get. Even before comparing them to the target values, we can measure something about the algorithm: if we feed in a set of similar inputs, then we would expect to get similar outputs for them. This measure of the variability of the algorithm is also known as precision, and it tells us how repeatable the predictions that the algorithm makes are. It might be useful to think of precision as being something like the variance of a probability distribution: it tells you how much spread around the mean to expect.

The point is that just because an algorithm is precise it does not mean that it is accurate – it can be precisely wrong if it always gives the wrong prediction. One measure of how well the algorithm's predictions match reality is known as trueness, and it can be defined as the average distance between the correct output and the prediction. Trueness doesn't usually make much sense for classification problems unless there is some concept of certain classes being similar to each other. Figure 2.9 illustrates the idea of trueness and precision in the traditional way: as a darts game, with four examples with varying trueness and precision for the three darts thrown by a player.

This section has considered the endpoint of machine learning, looking at the outputs, and thinking about what we need to do with the input data in terms of having multiple datasets, etc. In the next section we return to the starting point and consider how we can start analysing a dataset by dealing with probabilities.

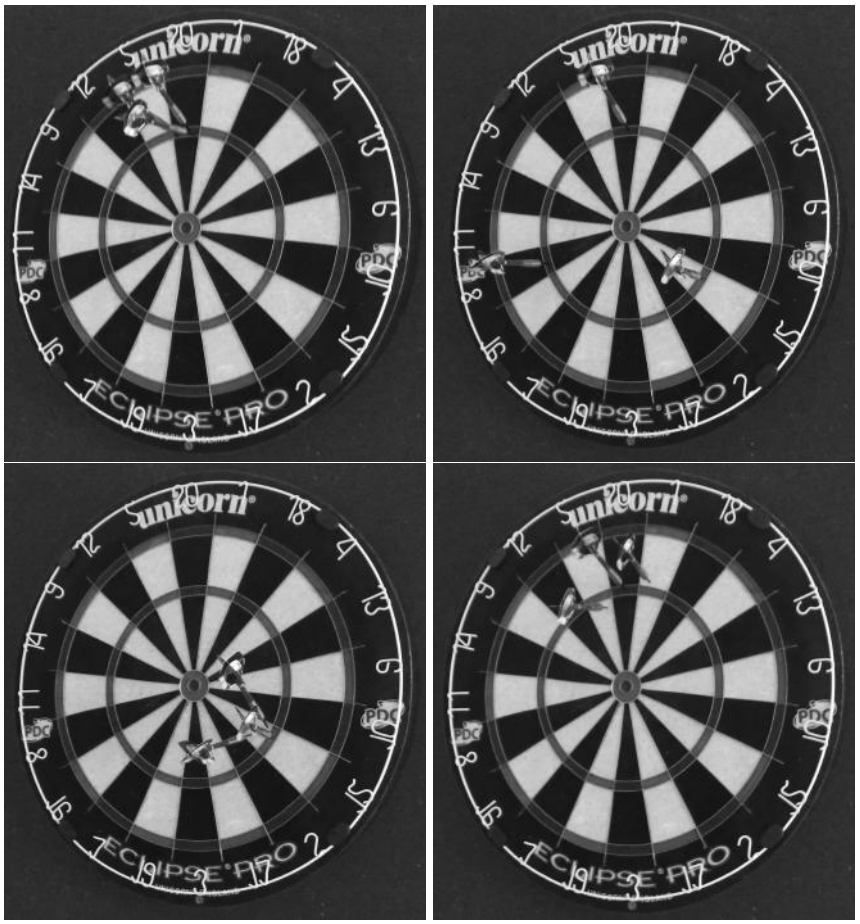

FIGURE 2.9 Assuming that the player was aiming for the highest-scoring triple 20 in darts (the segments each score the number they are labelled with, the narrow band on the outside of the circle scores double and the narrow band halfway in scores triple; the outer and inner 'bullseye' at the centre score 25 and 50, respectively), these four pictures show different outcomes. *Top left:* very accurate: high precision and trueness, *top right:* low precision, but good trueness, *bottom left:* high precision, but low trueness, and *bottom right:* reasonable trueness and precision, but the actual outputs are not very good. *(Thanks to Stefan Nowicki, whose dartboard was used for these pictures.)*

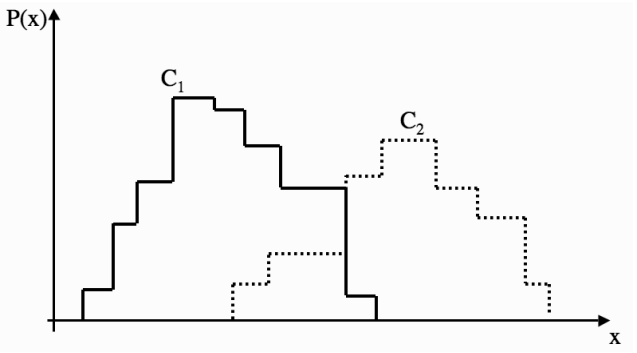

FIGURE 2.10 A histogram of feature values (x) against their probability for two classes.

2.3 TURNING DATA INTO PROBABILITIES

Take a look at the plot in Figure 2.10. It shows the measurements of some feature x for two classes, C_1 and C_2. Members of class C_2 tend to have larger values of feature x than members of class C_1, but there is some overlap between the two classes. The correct class is fairly easy to predict at the extremes of the range, but what to do in the middle is unclear. Suppose that we are trying to classify writing of the letters 'a' and 'b' based on their height (as shown in Figure 2.11). Most people write their 'a's smaller than their 'b's, but not everybody. However, in this example, we have a secret weapon. We know that in English text, the letter 'a' is much more common than the letter 'b' (we called this an unbalanced dataset earlier). If we see a letter that is either an 'a' or a 'b' in normal writing, then there is a 75% chance that it is an 'a.' We are using prior knowledge to estimate the probability that the letter is an 'a': in this example, $P(C_1) = 0.75, P(C_2) = 0.25$. If we weren't allowed to see the letter at all, and just had to classify it, then if we picked 'a' every time, we'd be right 75% of the time.

However, when we are asked to make a classification we are also given the value of x. It would be pretty silly to just use the value of $P(C_1)$ and ignore the value of x if it might help! In fact, we are given a training set of values of x and the class that each exemplar belongs to. This lets us calculate the value of $P(C_1)$ (we just count how many times out of the total the class was C_1 and divide by the total number of examples), and also another useful measurement: the conditional probability of C_1 given that x has value X: $P(C_1|X)$. The conditional probability tells us how likely it is that the class is C_1 given that the value of x is X. So in Figure 2.10 the value of $P(C_1|X)$ will be much larger for small values of X than for large values. Clearly, this is exactly what we want to calculate in order to perform classification. The question is how to get to this conditional probability, since we can't read it directly from the histogram.

The first thing that we need to do to get these values is to quantise the measurement x, which just means that we put it into one of a discrete set of values $\{X\}$, such as the bins in a histogram. This is exactly what is plotted in Figure 2.10. Now, if we have lots of examples of the two classes, and the histogram bins that their measurements fall into, we can compute $P(C_i, X_j)$, which is the joint probability, and tells us how often a measurement of C_i fell into histogram bin X_j. We do this by looking in histogram bin X_j, counting the number of examples of class C_i that are in it, and dividing by the total number of examples (of any class).

We can also define $P(X_j|C_i)$, which is a different conditional probability, and tells us

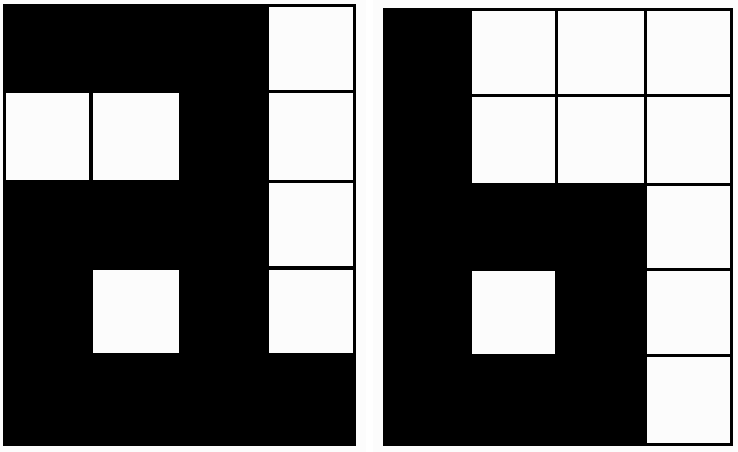

FIGURE 2.11 The letters 'a' and 'b' in pixel form.

how often (in the training set) there is a measurement of X_j given that the example is a member of class C_i. Again, we can just get this information from the histogram by counting the number of examples of class C_i in histogram bin X_j and dividing by the number of examples of that class there are (in any bin). Hopefully, this has just been revision for you from a statistics course at some stage; if not, and you don't follow it, get hold of any introductory probability book.

So we have now worked out two things from our training data: the joint probability $P(C_i, X_j)$ and the conditional probability $P(X_j|C_i)$. Since we actually want to compute $P(C_i|X_j)$ we need to know how to link these things together. As some of you may already know, the answer is Bayes' rule, which is what we are now going to derive. There is a link between the joint probability and the conditional probability. It is:

$$P(C_i, X_j) = P(X_j|C_i)P(C_i), \tag{2.10}$$

or equivalently:

$$P(C_i, X_j) = P(C_i|X_j)P(X_j). \tag{2.11}$$

Clearly, the right-hand side of these two equations must be equal to each other, since they are both equal to $P(C_i, X_j)$, and so with one division we can write:

$$P(C_i|X_j) = \frac{P(X_j|C_i)P(C_i)}{P(X_j)}. \tag{2.12}$$

This is Bayes' rule. If you don't already know it, learn it: it is the most important equation in machine learning. It relates the posterior probability $P(C_i|X_j)$ with the prior probability $P(C_i)$ and class-conditional probability $P(X_j|C_i)$. The denominator (the term on the bottom of the fraction) acts to normalise everything, so that all the probabilities sum to 1. It might not be clear how to compute this term. However, if we notice that any observation X_k has to belong to some class C_i, then we can marginalise over the classes to compute:

$$P(X_k) = \sum_i P(X_k|C_i)P(C_i). \tag{2.13}$$

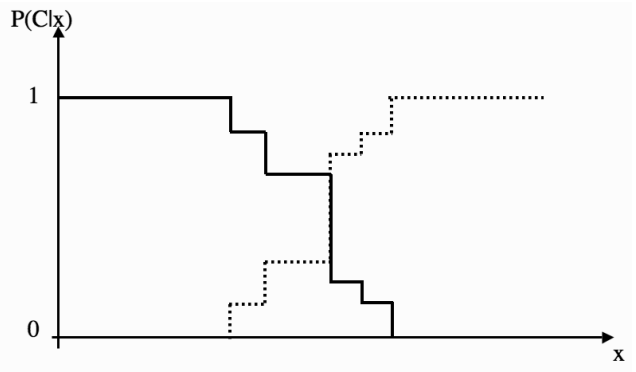

FIGURE 2.12 The posterior probabilities of the two classes C_1 and C_2 for feature x.

The reason why Bayes' rule is so important is that it lets us obtain the posterior probability—which is what we actually want—by calculating things that are much easier to compute. We can estimate the prior probabilities by looking at how often each class appears in our training set, and we can get the class-conditional probabilities from the histogram of the values of the feature for the training set. We can use the posterior probability (Figure 2.12) to assign each new observation to one of the classes by picking the class C_i where:

$$P(C_i|\mathbf{x}) > P(C_j|\mathbf{x}) \quad \forall \; i \neq j, \tag{2.14}$$

where $\mathbf{x}$ is a vector of feature values instead of just one feature. This is known as the maximum a posteriori or MAP hypothesis, and it gives us a way to choose which class to choose as the output one. The question is whether this is the right thing to do. There has been quite a lot of research in both the statistical and machine learning literatures into what is the right question to ask about our data to perform classification, but we are going to skate over it very lightly.

The MAP question is what is the most likely class given the training data? Suppose that there are three possible output classes, and for a particular input the posterior probabilities of the classes are $P(C_1|\mathbf{x}) = 0.35$, $P(C_2|\mathbf{x}) = 0.45$, $P(C_3|\mathbf{x}) = 0.2$. The MAP hypothesis therefore tells us that this input is in class C_2, because that is the class with the highest posterior probability. Now suppose that, based on the class that the data is in, we want to do something. If the class is C_1 or C_3 then we do action 1, and if the class is C_2 then we do action 2. As an example, suppose that the inputs are the results of a blood test, the three classes are different possible diseases, and the output is whether or not to treat with a particular antibiotic. The MAP method has told us that the output is C_2, and so we will not treat the disease. But what is the probability that it does not belong to class C_2, and so should have been treated with the antibiotic? It is $1 - P(C_2) = 0.55$. So the MAP prediction seems to be wrong: we should treat with antibiotic, because overall it is more likely. This method where we take into account the final outcomes of all of the classes is called the Bayes' Optimal Classification. It minimises the probability of misclassification, rather than maximising the posterior probability.

2.3.1 Minimising Risk

In the medical example we just saw it made sense to classify based on minimising the probability of misclassification. We can also consider the risk that is involved in the misclassification. The risk from misclassifying someone as unhealthy when they are healthy is usually smaller than the other way around, but not necessarily always: there are plenty of treatments that have nasty side effects, and you wouldn't want to suffer from those if you didn't have the disease. In cases like this we can create a loss matrix that specifies the risk involved in classifying an example of class C_i as class C_j. It looks like the confusion matrix we saw in Section 2.2, except that a loss matrix always contains zeros on the leading diagonal since there should never be a loss from getting the classification correct! Once we have the loss matrix, we just extend our classifier to minimise risk by multiplying each case by the relevant loss number.

2.3.2 The Naïve Bayes' Classifier

We're now going to return to performing classification, without worrying about the outcomes, so that we are back to calculating the MAP outcome, Equation (2.14). We can compute this exactly as described above, and it will work fine. However, suppose that the vector of feature values had many elements, so that there were lots of different features that were measured. How would this affect the classifier? We are trying to estimate $P(\mathbf{X}_j|C_i) = P(X_j^1, X_j^2, \ldots X_j^n|C_i)$ (where the superscripts index the elements of the vector) by looking at the histogram of all of our training data. As the dimensionality of $\mathbf{X}$ increases (as n gets larger), the amount of data in each bin of the histogram shrinks. This is the curse of dimensionality again (Section 2.1.2), and means that we need much more data as the dimensionality increases.

There is one simplifying assumption that we can make. We can assume that the elements of the feature vector are conditionally independent of each other, given the classification. So given the class C_i, the values of the different features do not affect each other. This is the naïveté in the name of the classifier, since it often doesn't make much sense—it tells us that the features are independent of each other. If we were to try to classify coins it would say that the weight and the diameter of the coin are independent of each other, which clearly isn't true. However, it does mean that the probability of getting the string of feature values $P(X_j^1 = a_1, X_j^2 = a_2, \ldots, X_j^n = a_n|C_i)$ is just equal to the product of multiplying together all of the individual probabilities:

$$P(X_j^1 = a_1|C_i) \times P(X_j^2 = a_2|C_i) \times \ldots \times P(X_j^n = a_n|C_i) = \prod_k P(X_j^k = a_k|C_i), \quad (2.15)$$

which is much easier to compute, and reduces the severity of the curse of dimensionality. So the classifier rule for the naïve Bayes' classifier is to select the class C_i for which the following computation is the maximum:

$$P(C_i) \prod_k P(X_j^k = a_k|C_i). \quad (2.16)$$

This is clearly a great simplification over evaluating the full probability, so it might come as a surprise that the naïve Bayes' classifier has been shown to have comparable results to other classification methods in certain domains. Where the simplification is true, so that the features are conditionally independent of each other, the naïve Bayes' classifier produces exactly the MAP classification.

In Chapter 12 on learning with trees, particularly Section 12.4, there is an example concerned with what to do in the evening based on whether you have an assignment deadline and what is happening. The data, shown below, consists of a set of prior examples from the last few days.

Deadline?	Is there a party?	Lazy?	Activity
Urgent	Yes	Yes	Party
Urgent	No	Yes	Study
Near	Yes	Yes	Party
None	Yes	No	Party
None	No	Yes	Pub
None	Yes	No	Party
Near	No	No	Study
Near	No	Yes	TV
Near	Yes	Yes	Party
Urgent	No	No	Study

In Chapter 12 we will see the results of a decision tree learning about this data, but here we will use the naïve Bayes' classifier. We feed in the current values for the feature variables (deadline, whether there is a party, etc.) and ask the classifier to compute the probabilities of each of the four possible things that you might do in the evening based on the data in the training set. Then we pick the most likely class. Note that the probabilities will be very small. This is one of the problems with the Bayes' classifier: since we are multiplying lots of probabilities, which are all less than one, the numbers get very small.

Suppose that you have deadlines looming, but none of them are particularly urgent, that there is no party on, and that you are currently lazy. Then the classifier needs to evaluate:

- P(Party) × P(Near | Party) × P(No Party | Party) × P(Lazy | Party)
- P(Study) × P(Near | Study) × P(No Party | Study) × P(Lazy | Study)
- P(Pub) × P(Near | Pub) × P(No Party | Pub) × P(Lazy | Pub)
- P(TV) × P(Near | TV) × P(No Party | TV) × P(Lazy | TV)

Using the data above these evaluate to:

$$\begin{aligned} P(\text{Party}|\text{near (not urgent) deadline, no party, lazy}) &= \frac{5}{10} \times \frac{2}{5} \times \frac{0}{5} \times \frac{3}{5} \\ &= 0 \qquad (2.17) \\ P(\text{Study}|\text{near (not urgent) deadline, no party, lazy}) &= \frac{3}{10} \times \frac{1}{3} \times \frac{3}{3} \times \frac{1}{3} \\ &= \frac{1}{30} \qquad (2.18) \\ P(\text{Pub}|\text{near (not urgent) deadline, no party, lazy}) &= \frac{1}{10} \times \frac{0}{1} \times \frac{1}{1} \times \frac{1}{1} \\ &= 0 \qquad (2.19) \\ P(\text{TV}|\text{near (not urgent) deadline, no party, lazy}) &= \frac{1}{10} \times \frac{1}{1} \times \frac{1}{1} \times \frac{1}{1} \\ &= \frac{1}{10} \qquad (2.20) \end{aligned}$$

So based on this you will be watching TV tonight.

2.4 SOME BASIC STATISTICS

This section will provide a quick summary of a few important statistical concepts. You may well already know about them, but just in case we'll go over them, highlighting the points that are important for machine learning. Any basic statistics book will give considerably more detailed information.

2.4.1 Averages

We'll start as basic as can be, with the two numbers that can be used to characterise a dataset: the mean and the variance. The mean is easy, it is the most commonly used average of a set of data, and is the value that is found by adding up all the points in the dataset and dividing by the number of points. There are two other averages that are used: the median and the mode. The median is the middle value, so the most common way to find it is to sort the dataset according to size and then find the point that is in the middle (of course, if there is an even number of datapoints then there is no exact middle, so people typically take the value halfway between the two points that are closest to the middle). There is a faster algorithm for computing the median based on a randomised algorithm that is described in most textbooks on algorithms. The mode is the most common value, so it just requires counting how many times each element appears and picking the most frequent one. We will also need to develop the idea of variance within a dataset, and of probability distributions.

2.4.2 Variance and Covariance

If we are given a set of random numbers, then we already know how to compute the mean of the set, together with the median. However, there are other useful statistics that can be computed, one of which is the expectation. The name expectation shows the gambling roots of most probability theory, since it describes the amount of money you can expect to win. It consists of multiplying together the payoff for each possibility with the probability of that thing happening, and then adding them all together. So if you are approached in the street by somebody selling raffle tickets for $1 and they tell you that there is a prize of $100,000 and they are selling 200,000 tickets, then you can work out the expected value of your ticket as:

$$E = -1 \times \frac{199,999}{200,000} + 99,999 \times \frac{1}{200,000} = -0.5, \tag{2.21}$$

where the -1 is the price of your ticket, which does not win 199,999 times out of 200,000 and the 99,999 is the prize minus the cost of your ticket. Note that the expected value is not a real value: you will never actually get 50 cents back, no matter what happens. If we just compute the expected value of a set of numbers, then we end up with the mean value.

The variance of the set of numbers is a measure of how spread out the values are. It is computed as the sum of the squared distances between each element in the set and the expected value of the set (the mean, $\boldsymbol{\mu}$):

$$\text{var}(\{\mathbf{x}_i\}) = \sigma^2(\{\mathbf{x}_i\}) = E((\{\mathbf{x}_i\} - \boldsymbol{\mu})^2) = \sum_{i=1}^{N} (\mathbf{x}_i - \boldsymbol{\mu})^2. \tag{2.22}$$

The square root of the variance, σ, is known as the standard deviation. The variance looks at the variation in one variable compared to its mean. We can generalise this to look at how two variables vary together, which is known as the covariance. It is a measure of how dependent the two variables are (in the statistical sense). It is computed by:

$$\text{cov}(\{\mathbf{x}_i\}, \{\mathbf{y}_i\}) = E(\{\mathbf{x}_i\} - \boldsymbol{\mu})E(\{\mathbf{y}_i\} - \boldsymbol{\nu}), \tag{2.23}$$

where $\boldsymbol{\nu}$ is the mean of set $\{\mathbf{y}_i\}$. If two variables are independent, then the covariance is 0 (the variables are then known as uncorrelated), while if they both increase and decrease at the same time, then the covariance is positive, and if one goes up while the other goes down, then the covariance is negative.

The covariance can be used to look at the correlation between all pairs of variables within a set of data. We need to compute the covariance of each pair, and these are then put together into what is imaginatively known as the covariance matrix. It can be written as:

$$\boldsymbol{\Sigma} = \begin{pmatrix} E[(\mathbf{x}_1 - \boldsymbol{\mu}_1)(\mathbf{x}_1 - \boldsymbol{\mu}_1)] & E[(\mathbf{x}_1 - \boldsymbol{\mu}_1)(\mathbf{x}_2 - \boldsymbol{\mu}_2)] & \dots & E[(\mathbf{x}_1 - \boldsymbol{\mu}_1)(\mathbf{x}_n - \mu_n)] \\ E[(\mathbf{x}_2 - \boldsymbol{\mu}_2)(\mathbf{x}_1 - \boldsymbol{\mu}_1)] & E[(\mathbf{x}_2 - \boldsymbol{\mu}_2)(\mathbf{x}_2 - \boldsymbol{\mu}_2)] & \dots & E[(\mathbf{x}_2 - \boldsymbol{\mu}_2)(\mathbf{x}_n - \boldsymbol{\mu}_n)] \\ \dots & \dots & \dots & \dots \\ E[(\mathbf{x}_n - \boldsymbol{\mu}_n)(\mathbf{x}_1 - \boldsymbol{\mu}_1)] & E[(\mathbf{x}_n - \boldsymbol{\mu}_n)(\mathbf{x}_2 - \boldsymbol{\mu}_2)] & \dots & E[(\mathbf{x}_n - \boldsymbol{\mu}_n)(\mathbf{x}_n - \boldsymbol{\mu}_n)] \end{pmatrix} \tag{2.24}$$

where $\mathbf{x}_i$ is a column vector describing the elements of the ith variable, and $\boldsymbol{\mu}_i$ is their mean. Note that the matrix is square, that the elements on the leading diagonal of the matrix are equal to the variances, and that it is symmetric since $\text{cov}(\mathbf{x}_i, \mathbf{x}_j) = \text{cov}(\mathbf{x}_j, \mathbf{x}_i)$. Equation (2.24) can also be written in matrix form as $\boldsymbol{\Sigma} = E[(\mathbf{X} - E[\mathbf{X}])(\mathbf{X} - E[\mathbf{X}])^T]$, recalling that the mean of a variable $\mathbf{X}$ is $E(\mathbf{X})$.

We will see in Chapter 6 that the covariance matrix has other uses, but for now we will think about what it tells us about a dataset. In essence, it says how the data varies along each data dimension. This is useful if we want to think about distances again. Suppose I gave you the two datasets shown in Figure 2.13 and the test point (labelled by the large 'X' in the figures) and asked you if the 'X' was part of the data. For the figure on the left you would probably say yes, while for the figure on the right you would say no, even though the two points are the same distance from the centre of the data. The reason for this is that as well as looking at the mean, you've also looked at where the test point lies in relation to the spread of the actual datapoints. If the data is tightly controlled then the test point has to be close to the mean, while if the data is very spread out, then the distance of the test point from the mean does not matter as much. We can use this to construct a distance measure that takes this into account. It is called the Mahalanobis distance after the person who described it in 1936, and is written as:

$$D_M(\mathbf{x}) = \sqrt{(\mathbf{x} - \boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1} (\mathbf{x} - \boldsymbol{\mu})}, \tag{2.25}$$

where $\mathbf{x}$ is the data arranged as a column vector, $\boldsymbol{\mu}$ is column vector representing the mean, and $\boldsymbol{\Sigma}^{-1}$ is the inverse of the covariance matrix. If we set the covariance matrix to the identity matrix, then the Mahalanobis distance reduces to the Euclidean distance.

Computing the Mahalanobis distance requires some fairly heavy computational machinery in computing the covariance matrix and then its inverse. Fortunately these are very easy to do in NumPy. There is a function that estimates the covariance matrix of a dataset (`np.cov(x)` for data matrix `x`) and the inverse is called `np.linalg.inv(x)`. The inverse does not have to exist in all cases, of course.

We are now going to consider a probability distribution, which describes the probabilities of something occurring over the range of possible feature values. There are lots of probability distributions that are common enough to have names, but there is one that is much better known than any other, because it occurs so often; therefore, that is the only one we will worry about here.

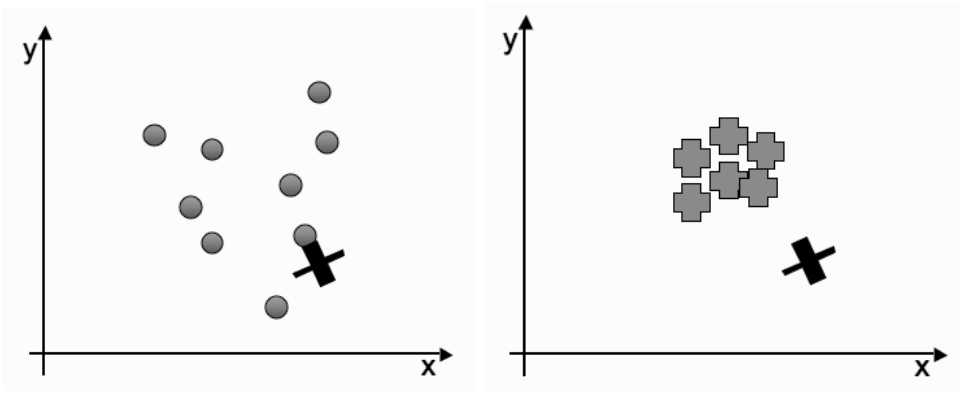

FIGURE 2.13 Two different datasets and a test point.

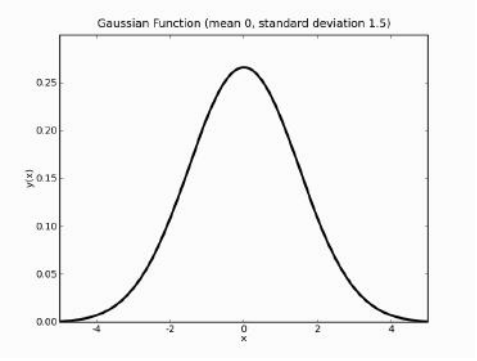

FIGURE 2.14 Plot of the one-dimensional Gaussian curve.

2.4.3 The Gaussian

The probability distribution that is most well known (indeed, the only one that many people know, or even need to know) is the Gaussian or normal distribution. In one dimension it has the familiar 'bell-shaped' curve shown in Figure 2.14, and its equation in one dimension is:

$$p(x) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(\frac{-(x-\mu)^2}{2\sigma^2}\right), \tag{2.26}$$

where μ is the mean and σ the standard deviation. The Gaussian distribution turns up in many problems because of the Central Limit Theorem, which says that lots of small random numbers will add up to something Gaussian. In higher dimensions it looks like:

$$p(\mathbf{x}) = \frac{1}{(2\pi)^{d/2}|\mathbf{\Sigma}|^{1/2}} \exp\left(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^T\mathbf{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})\right), \tag{2.27}$$

where $\mathbf{\Sigma}$ is the $n \times n$ covariance matrix (with $|\mathbf{\Sigma}|$ being its determinant and $\mathbf{\Sigma}^{-1}$ being its inverse). Figure 2.15 shows the appearance in two dimensions of three different cases: when the covariance matrix is the identity; when there are only numbers on the leading diagonal of the matrix; and the general case. The first case is known as a spherical covariance matrix, and has only 1 parameter. The second and third cases define ellipses in two dimensions, either aligned with the axes (with n parameters) or more generally, with n^2 parameters.

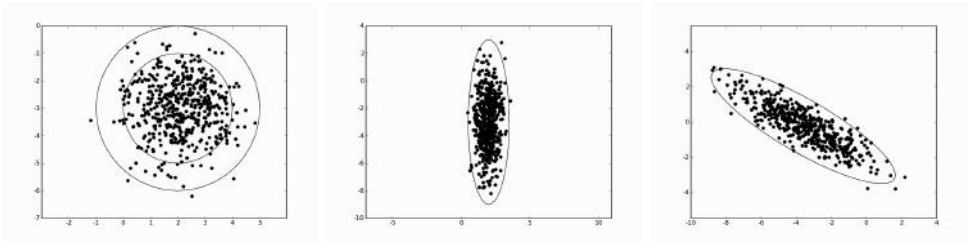

FIGURE 2.15 The two-dimensional Gaussian when (*left*) the covariance matrix is the identity, (*centre*) the covariance matrix has elements on the leading diagonal only, and (*right*) the general case.

2.5 THE BIAS-VARIANCE TRADEOFF

To round off this chapter, we use the statistical ideas of the previous section to look again at the idea of how to evaluate the amount of learning that can be performed, from a theoretical perspective.

Whenever we train any type of machine learning algorithm we are making some choices about a model to use, and fitting the parameters of that model. The more degrees of freedom the algorithm has, the more complicated the model that can be fitted. We have already seen that more complicated models have inherent dangers such as overfitting, and requiring more training data, and we have seen the need for validation data to ensure that the model does not overfit. There is another way to understand this idea that more complex models do not necessarily result in better results. Some people call it the bias-variance dilemma rather than a tradeoff, but this seems to be over-dramatising things a little.

In fact, it is a very simple idea. A model can be bad for two different reasons. Either it is not accurate and doesn't match the data well, or it is not very precise and there is a lot of variation in the results. The first of these is known as the bias, while the second is the statistical variance. More complex classifiers will tend to improve the bias, but the cost of this is higher variance, while making the model more specific by reducing the variance will increase the bias. Just like the Heisenberg Uncertainty Principle in quantum physics, there is a fundamental law at work behind the scenes that says that we can't have everything at once. As an example, consider the difference between a straight line fit to some data and a high degree polynomial, which can go precisely though the datapoints. The straight line has no variance at all, but high bias since it is a bad fit to the data in general. The spline can fit the training data to arbitrary accuracy, but the variance will increase. Note that the variance probably increases by rather less than the bias decreases, since we expect that the spline will give a better fit. Some models are definitely better than others, but choosing the complexity of the model is important for getting good results.

The most common way to compute the error between the targets and the predicted outputs is to sum up the squares of the difference between the two (the reason for squaring them is that if we don't, and just add up the differences, and if we had one example where the target was bigger than the prediction, and one where it was smaller by the same amount, then they would sum to zero). When looking at this sum-of-squares error function we can split it up into separate pieces that represent the bias and the variance. Suppose that the function that we are trying to approximate is $y = f(\mathbf{x}) + \epsilon$, where ϵ is the noise, which is assumed to be Gaussian with 0 mean and variance σ^2. We use our machine learning algorithm to fit

a hypothesis $h(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + b$ (where $\mathbf{w}$ is the weight vector from Section 2.1) to the data in order to minimise the sum-of-squares error $\sum_i (y_i - h(\mathbf{x}_i))^2$.

In order to decide whether or not our method is successful we need to consider it on independent data, so we consider a new input $\mathbf{x}^*$ and compute the expected value of the sum-of squares error, which we will assume is a random variable. Remember that $E[x] = \bar{x}$, the mean value. We are now going to do some algebraic manipulation, mostly based on the fact that (where Z is just some random variable):

$$\begin{aligned} E[(Z - \bar{Z})^2] &= E[Z^2 - 2Z\bar{Z} + \bar{Z}^2] \\ &= E[Z^2] - 2E[Z]\bar{Z} + \bar{Z}^2 \\ &= E[Z^2] - 2\bar{Z}\bar{Z} + \bar{Z}^2 \\ &= E[Z^2] - \bar{Z}^2. \end{aligned} \tag{2.28}$$

Using this, we can compute the expectation of the sum-of-squares error of a new datapoint:

$$\begin{aligned} E[(y^* - h(\mathbf{x}^*))^2] &= E[y^{*2} - 2y^* h(\mathbf{x}^*) + h(\mathbf{x}^*)^2] \\ &= E[y^{*2}] - 2E[y^* h(\mathbf{x}^*)] + E[h(\mathbf{x}^*)^2] \\ &= E[(y^{*2} - f(\mathbf{x}^*))^2] + f(\mathbf{x}^*)^2 + E[(h(\mathbf{x}^* - \bar{h}(\mathbf{x}^*))^2] \\ &+ \bar{h}(\mathbf{x}^*)^2 - 2f(\mathbf{x}^*)\bar{h}(\mathbf{x}^*) \\ &= E[(y^{*2} - f(\mathbf{x}^*))^2] + E[(h(\mathbf{x}^*) - \bar{h}(\mathbf{x}^*))^2] + (f(\mathbf{x}^*) + \bar{h}(\mathbf{x}^*))^2 \\ &= \text{noise}^2 + \text{variance} + \text{bias}^2. \end{aligned} \tag{2.29}$$

The first of the three terms on the right of the equation is beyond our control. It is the irreducible error and is the variance of the test data. The second term is variance, and the third is the square of the bias. The variance tells us how much $\mathbf{x}^*$ changes depending on the particular training set that was used, while the bias tells us about the average error of $h(\mathbf{x}^*)$. It is possible to exchange bias and variance, so that you can have a model with low bias (meaning that on average the outputs are current), but high variance (meaning that the answers wibble around all over the place) or vice versa, but you can't make them both zero – for each model there is a tradeoff between them. However, for any particular model and dataset there is some reasonable set of parameters that will give the best results for the bias and variance together, and part of the challenge of model fitting is to find this point.

This tradeoff is a useful way to see what machine learning is doing in general, but it is time now to go and see what we can actually do with some real machine learning algorithms, starting with neural networks.

FURTHER READING

Any standard statistics textbook gives more detail about the basic probability and statistics introduced here, but for an alternative take from the point of view of machine learning, see:

- Sections 1.2 and 1.4 of C.M. Bishop. *Pattern Recognition and Machine Learning*. Springer, Berlin, Germany, 2006.

For more on the bias-variance tradeoff, see:

- Sections 7.2 and 7.3 of T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*, 2nd edition, Springer, Berlin, Germany, 2008.

There are two books on semi-supervised learning that can be used to get an overview of the area:

- O. Chapelle, B. Schölkopf, and A. Zien. *Semi-supervised learning*. MIT Press, Cambridge, MA, USA, 2006.
- X. Zhu and A.B. Goldberg. *Introduction to Semi-Supervised Learning*. Synthesis Lectures on Artificial Intelligence and Machine Learning, 2009.

PRACTICE QUESTIONS

Problem 2.1 Use Bayes' rule to solve the following problem: At a party you meet a person who claims to have been to the same school as you. You vaguely recognise them, but can't remember properly, so decide to work out how likely it is, given that:

- 1 in 2 of the people you vaguely recognise went to school with you
- 1 in 10 of the people at the party went to school with you
- 1 in 5 people at the party you vaguely recognise

Problem 2.2 Consider how using the risk calculation in Section 2.3.1 would change the naïve Bayes classifier.

Neurons, Neural Networks, and Linear Discriminants

We've spent enough time with the concepts of machine learning, now it is time to actually see it in practice. To start the process, we will return to our demonstration that learning is possible, which is the squishy thing that your skull protects.

3.1 THE BRAIN AND THE NEURON

In animals, learning occurs within the brain. If we can understand how the brain works, then there might be things in there for us to copy and use for our machine learning systems. While the brain is an impressively powerful and complicated system, the basic building blocks that it is made up of are fairly simple and easy to understand. We'll look at them shortly, but it's worth noting that in computational terms the brain does exactly what we want. It deals with noisy and even inconsistent data, and produces answers that are usually correct from very high dimensional data (such as images) very quickly. All amazing for something that weighs about 1.5 kg and is losing parts of itself all the time (neurons die as you age at impressive/depressing rates), but its performance does not degrade appreciably (in the jargon, this means it is robust).

So how does it actually work? We aren't actually that sure on most levels, but in this book we are only going to worry about the most basic level, which is the processing units of the brain. These are nerve cells called neurons. There are lots of them (100 billion = 10^{11} is the figure that is often given) and they come in lots of different types, depending upon their particular task. However, their general operation is similar in all cases: transmitter chemicals within the fluid of the brain raise or lower the electrical potential inside the body of the neuron. If this membrane potential reaches some threshold, the neuron spikes or fires, and a pulse of fixed strength and duration is sent down the axon. The axons divide (arborise) into connections to many other neurons, connecting to each of these neurons in a synapse. Each neuron is typically connected to thousands of other neurons, so that it is estimated that there are about 100 trillion (= 10^{14}) synapses within the brain. After firing, the neuron must wait for some time to recover its energy (the refractory period) before it can fire again.

Each neuron can be viewed as a separate processor, performing a very simple computation: deciding whether or not to fire. This makes the brain a massively parallel computer made up of 10^{11} processing elements. If that is all there is to the brain, then we should be able to model it inside a computer and end up with animal or human intelligence inside a computer. This is the view of strong AI. We aren't aiming at anything that grand in this

book, but we do want to make programs that learn. So how does learning occur in the brain? The principal concept is plasticity: modifying the strength of synaptic connections between neurons, and creating new connections. We don't know all of the mechanisms by which the strength of these synapses gets adapted, but one method that does seem to be used was first postulated by Donald Hebb in 1949, and that is what is discussed now.

3.1.1 Hebb's Rule

Hebb's rule says that the changes in the strength of synaptic connections are proportional to the correlation in the firing of the two connecting neurons. So if two neurons consistently fire simultaneously, then any connection between them will change in strength, becoming stronger. However, if the two neurons never fire simultaneously, the connection between them will die away. The idea is that if two neurons both respond to something, then they should be connected. Let's see a trivial example: suppose that you have a neuron somewhere that recognises your grandmother (this will probably get input from lots of visual processing neurons, but don't worry about that). Now if your grandmother always gives you a chocolate bar when she comes to visit, then some neurons, which are happy because you like the taste of chocolate, will also be stimulated. Since these neurons fire at the same time, they will be connected together, and the connection will get stronger over time. So eventually, the sight of your grandmother, even in a photo, will be enough to make you think of chocolate. Sound familiar? Pavlov used this idea, called classical conditioning, to train his dogs so that when food was shown to the dogs and the bell was rung at the same time, the neurons for salivating over the food and hearing the bell fired simultaneously, and so became strongly connected. Over time, the strength of the synapse between the neurons that responded to hearing the bell and those that caused the salivation reflex was enough that just hearing the bell caused the salivation neurons to fire in sympathy.

There are other names for this idea that synaptic connections between neurons and assemblies of neurons can be formed when they fire together and can become stronger. It is also known as long-term potentiation and neural plasticity, and it does appear to have correlates in real brains.

3.1.2 McCulloch and Pitts Neurons

Studying neurons isn't actually that easy. You need to be able to extract the neuron from the brain, and then keep it alive so that you can see how it reacts in controlled circumstances. Doing this takes a lot of care. One of the problems is that neurons are generally quite small (they must be if you've got 10^{11} of them in your head!) so getting electrodes into the synapses is difficult. It has been done, though, using neurons from the giant squid, which has some neurons that are large enough to see. Hodgkin and Huxley did this in 1952, measuring and writing down differential equations that compute the membrane potential based on various chemical concentrations, something that earned them a Nobel prize. We aren't going to worry about that, instead, we're going to look at a mathematical model of a neuron that was introduced in 1943. The purpose of a mathematical model is that it extracts only the bare essentials required to accurately represent the entity being studied, removing all of the extraneous details. McCulloch and Pitts produced a perfect example of this when they modelled a neuron as:

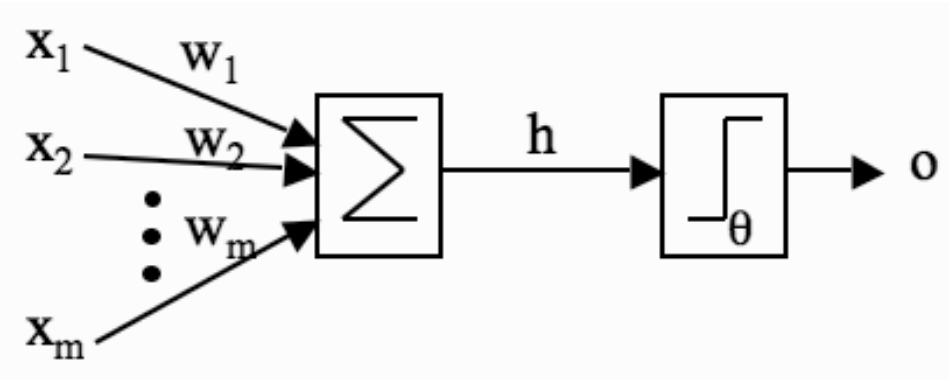

FIGURE 3.1 A picture of McCulloch and Pitts' mathematical model of a neuron. The inputs x_i are multiplied by the weights w_i, and the neurons sum their values. If this sum is greater than the threshold θ then the neuron fires; otherwise it does not.

(1) a set of weighted inputs w_i that correspond to the synapses

(2) an adder that sums the input signals (equivalent to the membrane of the cell that collects electrical charge)

(3) an activation function (initially a threshold function) that decides whether the neuron fires ('spikes') for the current inputs

A picture of their model is given in Figure 3.1, and we'll use the picture to write down a mathematical description. On the left of the picture are a set of input nodes (labelled $x_1, x_2, \ldots x_m$). These are given some values, and as an example we'll assume that there are three inputs, with $x_1 = 1, x_2 = 0, x_3 = 0.5$. In real neurons those inputs come from the outputs of other neurons. So the 0 means that a neuron didn't fire, the 1 means it did, and the 0.5 has no biological meaning, but never mind. (Actually, this isn't quite fair, but it's a long story and not very relevant.) Each of these other neuronal firings flowed along a synapse to arrive at our neuron, and those synapses have strengths, called weights. The strength of the synapse affects the strength of the signal, so we multiply the input by the weight of the synapse (so we get $x_1 \times w_1$ and $x_2 \times w_2$, etc.). Now when all of these signals arrive into our neuron, it adds them up to see if there is enough strength to make it fire. We'll write that as

$$h = \sum_{i=1}^{m} w_i x_i, \tag{3.1}$$

which just means sum (add up) all the inputs multiplied by their synaptic weights. I've assumed that there are m of them, where $m = 3$ in the example. If the synaptic weights are $w_1 = 1, w_2 = -0.5, w_3 = -1$, then the inputs to our model neuron are $h = 1 \times 1 + 0 \times -0.5 + 0.5 \times -1 = 1 + 0 + -0.5 = 0.5$. Now the neuron needs to decide if it is going to fire. For a real neuron, this is a question of whether the membrane potential is above some threshold. We'll pick a threshold value (labelled θ), say $\theta = 0$ as an example. Now, does our neuron fire? Well, $h = 0.5$ in the example, and $0.5 > 0$, so the neuron does fire, and produces output 1. If the neuron did not fire, it would produce output 0.

The McCulloch and Pitts neuron is a binary threshold device. It sums up the inputs (multiplied by the synaptic strengths or weights) and either fires (produces output 1) or does not fire (produces output 0) depending on whether the input is above some threshold. We can write the second half of the work of the neuron, the decision about whether or not to fire (which is known as an activation function), as:

$$o = g(h) = \begin{cases} 1 & \text{if } h > \theta \\ 0 & \text{if } h \leq \theta. \end{cases} \tag{3.2}$$

This is a very simple model, but we are going to use these neurons, or very simple variations on them using slightly different activation functions (that is, we'll replace the threshold function with something else) for most of our study of neural networks. In fact, these neurons might look simple, but as we shall see, a network of such neurons can perform any computation that a normal computer can, provided that the weights w_i are chosen correctly. So one of the main things we are going to talk about for the next few chapters is methods of setting these weights.

3.1.3 Limitations of the McCulloch and Pitts Neuronal Model

One question that is worth considering is how realistic is this model of a neuron? The answer is: not very. Real neurons are much more complicated. The inputs to a real neuron are not necessarily summed linearly: there may be non-linear summations. However, the most noticeable difference is that real neurons do not output a single output response, but a spike train, that is, a sequence of pulses, and it is this spike train that encodes information. This means that neurons don't actually respond as threshold devices, but produce a graded output in a continuous way. They do still have the transition between firing and not firing, though, but the threshold at which they fire changes over time. Because neurons are biochemical devices, the amount of neurotransmitter (which affects how much charge they required to spike, amongst other things) can vary according to the current state of the organism. Furthermore, the neurons are not updated sequentially according to a computer clock, but update themselves randomly (asynchronously), whereas in many of our models we will update the neurons according to the clock. There are neural network models that are asynchronous, but for our purposes we will stick to algorithms that are updated by the clock.

Note that the weights w_i can be positive or negative. This corresponds to excitatory and inhibitory connections that make neurons more likely to fire and less likely to fire, respectively.

Both of these types of synapses do exist within the brain, but with the McCulloch and Pitts neurons, the weights can change from positive to negative or vice versa, which has not been seen biologically—synaptic connections are either excitatory or inhibitory, and never change from one to the other. Additionally, real neurons can have synapses that link back to themselves in a feedback loop, but we do not usually allow that possibility when we make networks of neurons. Again, there are exceptions, but we won't get into them.

It is possible to improve the model to include many of these features, but the picture is complicated enough already, and McCulloch and Pitts neurons already provide a great deal of interesting behaviour that resembles the action of the brain, such as the fact that networks of McCulloch and Pitts neurons can memorise pictures and learn to represent functions and classify data, as we shall see in the next couple of chapters. In the last chapter we saw a simple model of a neuron that simulated what seems to be the most important function of a neuron—deciding whether or not to fire—and ignored the nasty biological things like

chemical concentrations, refractory periods, etc. Having this model is only useful if we can use it to understand what is happening when we learn, or use the model in order to solve some kind of problem. We are going to try to do both in this chapter, although the learning that we try to understand will be machine learning rather than animal learning.

3.2 NEURAL NETWORKS

One thing that is probably fairly obvious is that one neuron isn't that interesting. It doesn't do very much, except fire or not fire when we give it inputs. In fact, it doesn't even learn. If we feed in the same set of inputs over and over again, the output of the neuron never varies—it either fires or does not. So to make the neuron a little more interesting we need to work out how to make it learn, and then we need to put sets of neurons together into neural networks so that they can do something useful.

The question we need to think about first is how our neurons can learn. We are going to look at supervised learning for the next few chapters, which means that the algorithms will learn by example: the dataset that we learn from has the correct output values associated with each datapoint. At first sight this might seem pointless, since if you already know the correct answer, why bother learning at all? The key is in the concept of generalisation that we saw in Section 1.2. Assuming that there is some pattern in the data, then by showing the neural network a few examples we hope that it will find the pattern and predict the other examples correctly. This is sometimes known as pattern recognition.

Before we worry too much about this, let's think about what learning is. In the Introduction it was suggested that you learn if you get better at doing something. So if you can't program in the first semester and you can in the second, you have learnt to program. Something has changed (adapted), presumably in your brain, so that you can do a task that you were not able to do previously. Have a look again at the McCulloch and Pitts neuron (e.g., in Figure 3.1) and try to work out what can change in that model. The only things that make up the neuron are the inputs, the weights, and the threshold (and there is only one threshold for each neuron, but lots of inputs). The inputs can't change, since they are external, so we can only change the weights and the threshold, which is interesting since it tells us that most of the learning is in the weights, which aren't part of the neuron at all; they are the model of the synapse! Getting excited about neurons turns out to be missing something important, which is that the learning happens *between* the neurons, in the way that they are connected together.

So in order to make a neuron learn, the question that we need to ask is:

How should we change the weights and thresholds of the neurons so that the network gets the right answer more often?

Now that we know the right question to ask we'll have a look at our very first neural network, the space-age sounding Perceptron, and see how we can use it to solve the problem (it really was space-age, too: created in 1958). Once we've worked out the algorithm and how it works, we'll look at what it can and cannot do, and then see how statistics can give us insights into learning as well.

3.3 THE PERCEPTRON

The Perceptron is nothing more than a collection of McCulloch and Pitts neurons together with a set of inputs and some weights to fasten the inputs to the neurons. The network is shown in Figure 3.2. On the left of the figure, shaded in light grey, are the input nodes. These are not neurons, they are just a nice schematic way of showing how values are fed

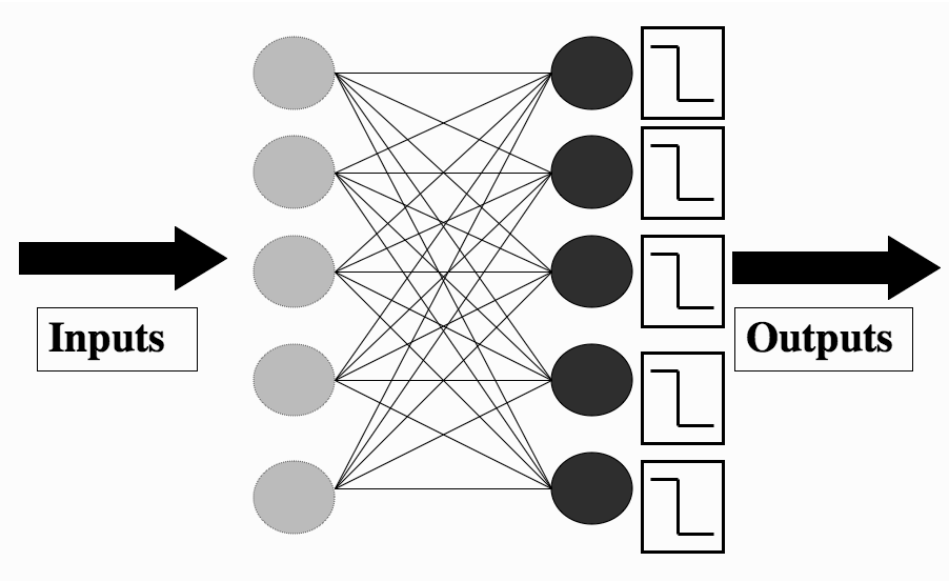

FIGURE 3.2 The Perceptron network, consisting of a set of input nodes (left) connected to McCulloch and Pitts neurons using weighted connections.

into the network, and how many of these input values there are (which is the dimension (number of elements) in the input vector). They are almost always drawn as circles, just like neurons, which is rather confusing, so I've shaded them a different colour. The neurons are shown on the right, and you can see both the additive part (shown as a circle) and the thresholder. In practice nobody bothers to draw the thresholder separately, you just need to remember that it is part of the neuron.

Notice that the neurons in the Perceptron are completely independent of each other: it doesn't matter to any neuron what the others are doing, it works out whether or not to fire by multiplying together its own weights and the input, adding them together, and comparing the result to its own threshold, regardless of what the other neurons are doing. Even the weights that go into each neuron are separate for each one, so the only thing they share is the inputs, since every neuron sees all of the inputs to the network.

In Figure 3.2 the number of inputs is the same as the number of neurons, but this does not have to be the case — in general there will be m inputs and n neurons. The number of inputs is determined for us by the data, and so is the number of outputs, since we are doing supervised learning, so we want the Perceptron to learn to reproduce a particular target, that is, a pattern of firing and non-firing neurons for the given input.

When we looked at the McCulloch and Pitts neuron, the weights were labelled as w_i, with the i index running over the number of inputs. Here, we also need to work out which neuron the weight feeds into, so we label them as w_{ij}, where the j index runs over the number of neurons. So w_{32} is the weight that connects input node 3 to neuron 2. When we make an implementation of the neural network, we can use a two-dimensional array to hold these weights.

Now, working out whether or not a neuron should fire is easy: we set the values of the input nodes to match the elements of an input vector and then use Equations (3.1) and (3.2) for each neuron. We can do this for all of the neurons, and the result is a pattern

of firing and non-firing neurons, which looks like a vector of 0s and 1s, so if there are 5 neurons, as in Figure 3.2, then a typical output pattern could be $(0, 1, 0, 0, 1)$, which means that the second and fifth neurons fired and the others did not. We compare that pattern to the target, which is our known correct answer for this input, to identify which neurons got the answer right, and which did not.

For a neuron that is correct, we are happy, but any neuron that fired when it shouldn't have done, or failed to fire when it should, needs to have its weights changed. The trouble is that we don't know what the weights should be—that's the point of the neural network, after all, so we want to change the weights so that the neuron gets it right next time. We are going to talk about this in a lot more detail in Chapter 4, but for now we're going to do something fairly simple to see that it is possible to find a solution.

Suppose that we present an input vector to the network and one of the neurons gets the wrong answer (its output does not match the target). There are m weights that are connected to that neuron, one for each of the input nodes. If we label the neuron that is wrong as k, then the weights that we are interested in are w_{ik}, where i runs from 1 to m. So we know which weights to change, but we still need to work out how to change the values of those weights. The first thing we need to know is whether each weight is too big or too small. This seems obvious at first: some of the weights will be too big if the neuron fired when it shouldn't have, and too small if it didn't fire when it should. So we compute $y_k - t_k$ (the difference between the output y_k, which is what the neuron did, and the target for that neuron, t_k, which is what the neuron should have done. This is a possible error function). If it is negative then the neuron should have fired and didn't, so we make the weights bigger, and vice versa if it is positive, which we can do by subtracting the error value. Hold on, though. That element of the input could be negative, which would switch the values over; so if we wanted the neuron to fire we'd need to make the value of the weight negative as well. To get around this we'll multiply those two things together to see how we should change the weight: $\Delta w_{ik} = -(y_k - t_k) \times x_i$, and the new value of the weight is the old value plus this value.

Note that we haven't said anything about changing the threshold value of the neuron. To see how important this is, suppose that a particular input is 0. In that case, even if a neuron is wrong, changing the relevant weight doesn't do anything (since anything times 0 is 0): we need to change the threshold. We will deal with this in an elegant way in Section 3.3.2. However, before we get to that, the learning rule needs to be finished—we need to decide how much to change the weight by. This is done by multiplying the value above by a parameter called the learning rate, usually labelled as η. The value of the learning rate decides how fast the network learns. It's quite important, so it gets a little subsection of its own (next), but first let's write down the final rule for updating a weight w_{ij}:

$$w_{ij} \leftarrow w_{ij} - \eta(y_j - t_j) \cdot x_i. \tag{3.3}$$

The other thing that we need to realise now is that the network needs to be shown every training example several times. The first time the network might get some of the answers correct and some wrong; the next time it will hopefully improve, and eventually its performance will stop improving. Working out how long to train the network for is not easy (we will see more methods in Section 4.3.3), but for now we will predefine the maximum number of iterations, T. Of course, if the network got all of the inputs correct, then this would also be a good time to stop.

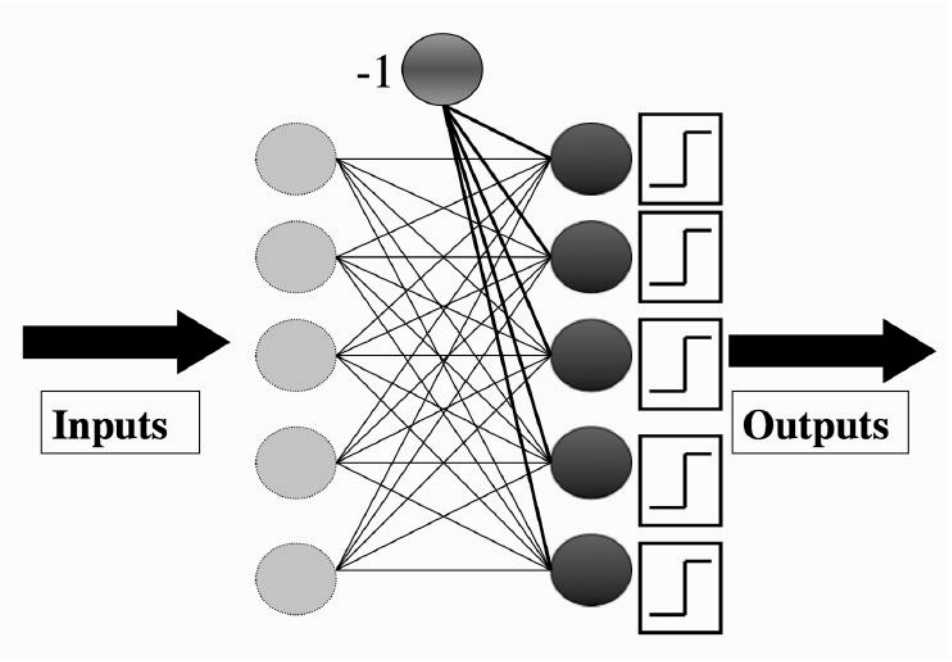

FIGURE 3.3 The Perceptron network again, showing the bias input.

3.3.1 The Learning Rate η

Equation (3.3) above tells us how to change the weights, with the parameter η controlling how much to change the weights by. We could miss it out, which would be the same as setting it to 1. If we do that, then the weights change a lot whenever there is a wrong answer, which tends to make the network unstable, so that it never settles down. The cost of having a small learning rate is that the weights need to see the inputs more often before they change significantly, so that the network takes longer to learn. However, it will be more stable and resistant to noise (errors) and inaccuracies in the data. We therefore use a moderate learning rate, typically $0.1 < \eta < 0.4$, depending upon how much error we expect in the inputs. It doesn't matter for the Perceptron algorithm, but for many of the algorithms that we will see in the book, the learning rate is a crucial parameter.

3.3.2 The Bias Input

When we discussed the McCulloch and Pitts neuron, we gave each neuron a firing threshold θ that determined what value it needed before it should fire. This threshold should be adjustable, so that we can change the value that the neuron fires at. Suppose that all of the inputs to a neuron are zero. Now it doesn't matter what the weights are (since zero times anything equals zero), the only way that we can control whether the neuron fires or not is through the threshold. If it wasn't adjustable and we wanted one neuron to fire when all the inputs to the network were zero, and another not to fire, then we would have a problem. No matter what values of the weights were set, the two neurons would do the same thing since they had the same threshold and the inputs were all zero.

The trouble is that changing the threshold requires an extra parameter that we need to write code for, and it isn't clear how we can do that in terms of the weight update that we

worked out earlier. Fortunately, there is a neat way around this problem. Suppose that we fix the value of the threshold for the neuron at zero. Now, we add an extra input weight to the neuron, with the value of the input to that weight always being fixed (usually the value of -± is chosen; in this book I'm going to use -1 to make it stand out, but any non-zero value will do). We include that weight in our update algorithm (like all the other weights), so we don't need to think of anything new. And the value of the weight will change to make the neuron fire—or not fire, whichever is correct—when an input of all zeros is given, since the input on that weight is always -1, even when all the other inputs are zero. This input is called a bias node, and its weights are usually given a 0 subscript, so that the weight connecting it to the jth neuron is w_{0j}.

3.3.3 The Perceptron Learning Algorithm

We are now ready to write our first learning algorithm. It might be useful to keep Figure 3.3 in mind as you read the algorithm, and we'll work through an example of using it afterwards. The algorithm is separated into two parts: a training phase, and a recall phase. The recall phase is used after training, and it is the one that should be fast to use, since it will be used far more often than the training phase. You can see that the training phase uses the recall equation, since it has to work out the activations of the neurons before the error can be calculated and the weights trained.

The Perceptron Algorithm

- **Initialisation**
  - set all of the weights w_{ij} to small (positive and negative) random numbers
- **Training**
  - for T iterations or until all the outputs are correct:
    - for each input vector:
      - compute the activation of each neuron j using activation function g:

$$y_j = g\left(\sum_{i=0}^{m} w_{ij}x_i\right) = \begin{cases} 1 & \text{if } \sum_{i=0}^{m} w_{ij}x_i > 0 \\ 0 & \text{if } \sum_{i=0}^{m} w_{ij}x_i \leq 0 \end{cases} \tag{3.4}$$

      - update each of the weights individually using:

$$w_{ij} \leftarrow w_{ij} - \eta(y_j - t_j) \cdot x_i \tag{3.5}$$

- **Recall**
  - compute the activation of each neuron j using:

$$y_j = g\left(\sum_{i=0}^{m} w_{ij}x_i\right) = \begin{cases} 1 & \text{if } w_{ij}x_i > 0 \\ 0 & \text{if } w_{ij}x_i \leq 0 \end{cases} \tag{3.6}$$

Note that the code on the website for the Perceptron has a different form, as will be discussed in Section 3.3.5.

Computing the computational complexity of this algorithm is very easy. The recall phase

In_1	In_2	t
0	0	0
0	1	1
1	0	1
1	1	1

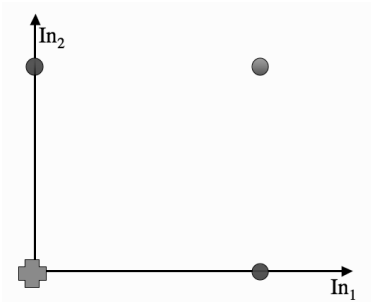

FIGURE 3.4 Data for the OR logic function and a plot of the four datapoints.

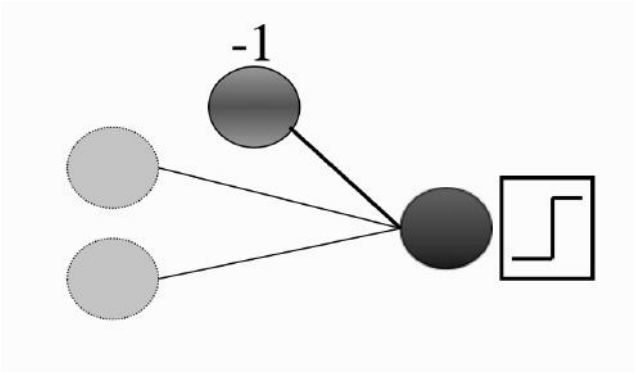

FIGURE 3.5 The Perceptron network for the example in Section 3.3.4.

loops over the neurons, and within that loops over the inputs, so its complexity is $\mathcal{O}(mn)$. The training part does this same thing, but does it for T iterations, so costs $\mathcal{O}(Tmn)$.

It might be the first time that you have seen an algorithm written out like this, and it could be hard to see how it can be turned into code. Equally, it might be difficult to believe that something as simple as this algorithm can learn something. The only way to fix these things is to work through the algorithm by hand on an example or two, and to try to write the code and then see if it does what is expected. We will do both of those things next, first working through a simple example by hand.

3.3.4 An Example of Perceptron Learning: Logic Functions

The example we are going to use is something very simple that you already know about, the logical OR. This obviously isn't something that you actually need a neural network to learn about, but it does make a nice simple example. So what will our neural network look like? There are two input nodes (plus the bias input) and there will be one output. The inputs and the target are given in the table on the left of Figure 3.4; the right of the figure shows a plot of the function with the circles as the true outputs, and a cross as the false one. The corresponding neural network is shown in Figure 3.5.

As you can see from Figure 3.5, there are three weights. The algorithm tells us to initialise the weights to small random numbers, so we'll pick $w_0 = -0.05, w_1 = -0.02, w_2 = 0.02$. Now we feed in the first input, where both inputs are 0: $(0, 0)$. Remember that the input to the bias weight is always -1, so the value that reaches the neuron is $-0.05 \times -1 +$

$-0.02 \times 0 + 0.02 \times 0 = 0.05$. This value is above 0, so the neuron fires and the output is 1, which is incorrect according to the target. The update rule tells us that we need to apply Equation (3.3) to each of the weights separately (we'll pick a value of $\eta = 0.25$ for the example):

$$w_0 : -0.05 - 0.25 \times (1 - 0) \times -1 = 0.2, \tag{3.7}$$
$$w_1 : -0.02 - 0.25 \times (1 - 0) \times 0 = -0.02, \tag{3.8}$$
$$w_2 : 0.02 - 0.25 \times (1 - 0) \times 0 = 0.02. \tag{3.9}$$

Now we feed in the next input $(0, 1)$ and compute the output (check that you agree that the neuron does not fire, but that it should) and then apply the learning rule again:

$$w_0 : 0.2 - 0.25 \times (0 - 1) \times -1 = -0.05, \tag{3.10}$$
$$w_1 : -0.02 - 0.25 \times (0 - 1) \times 0 = -0.02, \tag{3.11}$$
$$w_2 : 0.02 - 0.25 \times (0 - 1) \times 1 = 0.27. \tag{3.12}$$

For the $(1, 0)$ input the answer is already correct (you should check that you agree with this), so we don't have to update the weights at all, and the same is true for the $(1, 1)$ input. So now we've been through all of the inputs once. Unfortunately, that doesn't mean we've finished—not all the answers are correct yet. We now need to start going through the inputs again, until the weights settle down and stop changing, which is what tells us that the algorithm has finished. For real-world applications the weights may never stop changing, which is why you run the algorithm for some pre-set number of iterations, T.

So now we carry on running the algorithm, which you should check for yourself either by hand or using computer code (which we'll discuss next), eventually getting to weight values that settle and stop changing. At this point the weights stop changing, and the Perceptron has correctly learnt all of the examples. Note that there are lots of different values that we can assign to the weights that will give the correct outputs; the ones that the algorithm finds depend on the learning rate, the inputs, and the initial starting values. We are interested in finding a set that works; we don't necessarily care what the actual values are, providing that the network generalises to other inputs.

3.3.5 Implementation

Turning the algorithm into code is fairly simple: we need to design some data structures to hold the variables, then write and test the program. Data structures are usually very basic for machine learning algorithms; here we need an array to hold the inputs, another to hold the weights, and then two more for the outputs and the targets. When we talked about the presentation of data to the neural network we used the term input vectors. The vector is a list of values that are presented to the Perceptron, with one value for each of the nodes in the network. When we turn this into computer code it makes sense to put these values into an array. However, the neural network isn't very exciting if we only show it one datapoint: we will need to show it lots of them. Therefore it is normal to arrange the data into a two-dimensional array, with each row of the array being a datapoint. In a language like C or Java, you then write a loop that runs over each row of the array to present the input, and a loop within it that runs over the number of input nodes (which does the computation on the current input vector).

Written this way in Python syntax (Appendix A provides a brief introduction to

Python), the recall code that is used after training for a set of `nData` datapoints arranged in the array `inputs` looks like (this code can be found on the book website):

```
for data in range(nData): # loop over the input vectors
    for n in range(N): # loop over the neurons
        # Compute sum of weights times inputs for each neuron
        # Set the activation to 0 to start
        activation[data][n] = 0
        # Loop over the input nodes (+1 for the bias node)
        for m in range(M+1):
            activation[data][n] += weight[m][n] * inputs[data][m]

        # Now decide whether the neuron fires or not
        if activation[data][n] > 0:
            activation[data][n] = 1
        else
            activation[data][n] = 0
```

However, Python's numerical library NumPy provides an alternative method, because it can easily multiply arrays and matrices together (MATLAB® and R have the same facility). This means that we can write the code with fewer loops, making it rather easier to read, and also means that we write less code. It can be a little confusing at first, though. To understand it, we need a little bit more mathematics, which is the concept of a matrix. In computer terms, matrices are just two-dimensional arrays. We can write the set of weights for the network in a matrix by making an `np.array` that has $m + 1$ rows (the number of input nodes + 1 for the bias) and n columns (the number of neurons). Now, the element of the matrix at location (i, j) contains the weight connecting input i to neuron j, which is what we had in the code above.

The benefit that we get from thinking about it in this way is that multiplying matrices and vectors together is well defined. You've probably seen this in high school or somewhere but, just in case, to be able to multiply matrices together we need the inner dimensions to be the same. This just means that if we have matrices $\mathbf{A}$ and $\mathbf{B}$ where $\mathbf{A}$ is size $m \times n$, then the size of $\mathbf{B}$ needs to be $n \times p$, where p can be any number. The n is called the inner dimension since when we write out the size of the matrices in the multiplication we get $(m \times n) \times (n \times p)$.

Now we can compute $\mathbf{AB}$ (but not necessarily $\mathbf{BA}$, since for that we'd need $m = p$, since the computation above would then be $(n \times p) \times (m \times n)$). The computation of the multiplication proceeds by picking up the first column of $\mathbf{B}$, rotating it by 90° anti-clockwise so that it is a row not a column, multiplying each element of it by the matching element in the first row of $\mathbf{A}$ and then adding them together. This is the first element of the answer matrix. The second element in the first row is made by picking up the second column of $\mathbf{B}$, rotating it to match the direction, and multiplying it by the first row of $\mathbf{A}$, and so on. As an example:

$$\begin{pmatrix} 3 & 4 & 5 \\ 2 & 3 & 4 \end{pmatrix} \times \begin{pmatrix} 1 & 3 \\ 2 & 4 \\ 3 & 5 \end{pmatrix} \tag{3.13}$$

$$= \begin{pmatrix} 3\times 1+4\times 2+5\times 3 & 3\times 3+4\times 4+5\times 5 \\ 2\times 1+3\times 2+4\times 3 & 2\times 3+3\times 4+4\times 5 \end{pmatrix} \tag{3.14}$$

$$= \begin{pmatrix} 26 & 50 \\ 20 & 38 \end{pmatrix} \tag{3.15}$$

NumPy can do this multiplication for us, using the `np.dot()` function (which is a rather strange name mathematically, but never mind). So to reproduce the calculation above, we use (where `>>>` denotes the Python command line, and so this is code to be typed in, with the answers provided by the Python interpreter shown afterwards):

```
>>> import numpy as np
>>> a = np.array([[3,4,5],[2,3,4]])
>>> b = np.array([[1,3],[2,4],[3,5]])
>>> np.dot(a,b)
array([[26, 50],
       [20, 38]])
```

The `np.array()` function makes the NumPy array, which is actually a matrix here, made up of an array of arrays: each row is a separate array, as you can see from the square brackets within square brackets. Note that we can enter the 2D array in one line of code by using commas between the different rows, but when it prints them out, NumPy puts each row of the matrix on a different line, which makes things easier to see.

This probably seems like a very long way from the Perceptron, but we are getting there, I promise! We can put the input vectors into a two-dimensional array of size $N \times m$, where N is the number of input vectors we have and m is the number of inputs. The weights array is of size $m \times n$, and so we can multiply them together. If we do, then the output will be an $N \times n$ matrix that holds the values of the sum that each neuron computes for each of the N input vectors. Now we just need to compute the activations based on these sums. NumPy has another useful function for us here, which is `np.where(condition,x,y)`, (`condition` is a logical condition and `x` and `y` are values) that returns a matrix that has value `x` where `condition` is true and value `y` everywhere else. So using the matrix `a` that was used above,

```
>>> np.where(a>3,1,0)
array([[0, 1, 1],
       [0, 0, 1]])
```

The upshot of this is that the entire section of code for the recall function of the Perceptron can be rewritten in two lines of code as:

```
# Compute activations
activations =  np.dot(inputs,self.weights)

# Threshold the activations
return np.where(activations>0,1,0)
```

The training section isn't that much harder really. You should notice that the first part of the training algorithm is the same as the recall computation, so we can put them into a function (I've called it `pcnfwd` in the code because it consists of running forwards through the network to get the outputs). Then we just need to compute the weight updates. The weights are in an $m \times n$ matrix, the activations are in an $N \times n$ matrix (as are the targets) and the inputs are in an $N \times m$ matrix. So to do the multiplication `np.dot(inputs,targets - activations)` we need to turn the `inputs` matrix around so that it is $m \times N$. This is done using the `np.transpose()` function, which swaps the rows and columns over (so using matrix `a` above again) we get:

```
>>> np.transpose(a)
array([[3, 2],
       [4, 3],
       [5, 4]])
```

Once we have that, the weight update for the entire network can be done in one line (where `eta` is the learning rate, η):

```
self.weights -= eta*np.dot(np.transpose(inputs),self.activations-targets)
```

Assuming that you make sure in advance that all your input matrices are the correct size (the `np.shape()` function, which tells you the number of elements in each dimension of the array, is helpful here), the only things that are needed are to add those extra -1's onto the input vectors for the bias node, and to decide what values we should put into the weights to start with. The first of these can be done using the `np.concatenate()` function, making a one-dimensional array that contains -1 as all of its elements, and adding it on to the inputs array (note that `nData` in the code is equivalent to N in the text):

```
inputs = np.concatenate((inputs,-np.ones((self.nData,1))),axis=1)
```

The last thing we need to do is to give initial values to the weights. It is possible to set them all to be zero, and the algorithm will get to the right answer. However, instead we will assign small random numbers to the weights, for reasons that will be discussed in Section 4.2.2. Again, NumPy has a nice way to do this, using the built-in random number generator (with `nin` corresponding to m and `nout` to n):

```
weights = np.random.rand(nIn+1,nOut)*0.1-0.05
```

At this point we have seen all the snippets of code that are required, and putting them together should not be a problem. The entire program is available from the book website as `pcn.py`. Note that this is a different version of the algorithm because it is a batch version: all of the inputs go forward through the algorithm, and then the error is computed and the weights are changed. This is different to the sequential version that was written down in the first algorithm. The batch version is simpler to write in Python and often works better.

We now move on to seeing the code working, starting with the OR example that was used in the hand-worked demonstration.

Making the OR data is easy, and then running the code requires importing it using its filename (`pcn`) and then calling the `pcntrain` function. The print-out below shows the instructions to set up the arrays and call the function, and the output of the weights for 5 iterations of a particular run of the program, starting from random initial points (note that the weights stop changing after the 1st iteration in this case, and that different runs will produce different values).

```
>>> import numpy as np
>>> inputs = np.array([[0,0],[0,1],[1,0],[1,1]])
>>> targets = np.array([[0],[1],[1],[1]])
>>> import pcn_logic_eg
>>>
>>> p = pcn_logic_eg.pcn(inputs,targets)
>>> p.pcntrain(inputs,targets,0.25,6)
Iteration:  0
[[-0.03755646]
 [ 0.01484562]
 [ 0.21173977]]
Final outputs are:
[[0]
 [0]
 [0]
 [0]]
Iteration:  1
[[ 0.46244354]
 [ 0.51484562]
 [-0.53826023]]
Final outputs are:
[[1]
 [1]
 [1]
 [1]]
Iteration:  2
[[ 0.46244354]
 [ 0.51484562]
 [-0.28826023]]
```

```
Final outputs are:
[[1]
 [1]
 [1]
 [1]]
Iteration:  3
[[ 0.46244354]
 [ 0.51484562]
 [-0.03826023]]
Final outputs are:
[[1]
 [1]
 [1]
 [1]]
Iteration:  4
[[ 0.46244354]
 [ 0.51484562]
 [ 0.21173977]]
Final outputs are:
[[0]
 [1]
 [1]
 [1]]
Iteration:  5
[[ 0.46244354]
 [ 0.51484562]
 [ 0.21173977]]
Final outputs are:
[[0]
 [1]
 [1]
 [1]]
```

We have trained the Perceptron on the four datapoints $(0,0),(1,0),(0,1)$, and $(1,1)$. However, we could put in an input like $(0.8,0.8)$ and expect to get an output from the neural network. Obviously, it wouldn't make any sense from the logic function point-of-view, but most of the things that we do with neural networks will be more interesting than that, anyway. Figure 3.6 shows the decision boundary, which shows when the decision about which class to categorise the input as changes from crosses to circles. We will see why this is a straight line in Section 3.4.

Before returning the weights, the Perceptron algorithm above prints out the outputs for the trained inputs. You can also use the network to predict the outputs for other values by using the `pcnfwd` function. However, you need to manually add the -1s on in this case, using:

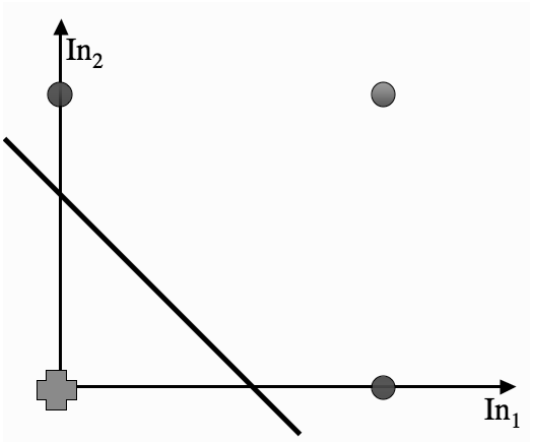

FIGURE 3.6 The decision boundary computed by a Perceptron for the OR function.

```
>>> # Add the inputs that match the bias node
>>> inputs_bias = np.concatenate((inputs,-np.ones((np.shape(inputs)[0],1))),
axis=1)
>>> pcn.pcnfwd(inputs_bias,weights)
```

The results on this test data are what you can use in order to compute the accuracy of the training algorithm using the methods that were described in Section 2.2.

In terms of learning about a set of data we have now reached the stage that neural networks were up to in 1969. Then, two researchers, Minsky and Papert, published a book called "Perceptrons." The purpose of the book was to stimulate neural network research by discussing the learning capabilities of the Perceptron, and showing what the network could and could not learn. Unfortunately, the book had another effect: it effectively killed neural network research for about 20 years. To see why, we need to think about how the Perceptron learns in a different way.

3.4 LINEAR SEPARABILITY

What does the Perceptron actually compute? For our one output neuron example of the OR data it tries to separate out the cases where the neuron should fire from those where it shouldn't. Looking at the graph on the right side of Figure 3.4, you should be able to draw a straight line that separates out the crosses from the circles without difficulty (it is done in Figure 3.6). In fact, that is exactly what the Perceptron does: it tries to find a straight line (in 2D, a plane in 3D, and a hyperplane in higher dimensions) where the neuron fires on one side of the line, and doesn't on the other. This line is called the decision boundary or discriminant function, and an example of one is given in Figure 3.7.

To see this, think about the matrix notation we used in the implementation, but consider just one input vector $\mathbf{x}$. The neuron fires if $\mathbf{x} \cdot \mathbf{w}^T \geq 0$ (where $\mathbf{w}$ is the row of $\mathbf{W}$ that connects the inputs to one particular neuron; they are the same for the OR example, since there is only one neuron, and $\mathbf{w}^T$ denotes the transpose of $\mathbf{w}$ and is used to make both of the vectors into column vectors). The $\mathbf{a} \cdot \mathbf{b}$ notation describes the inner or scalar product between two

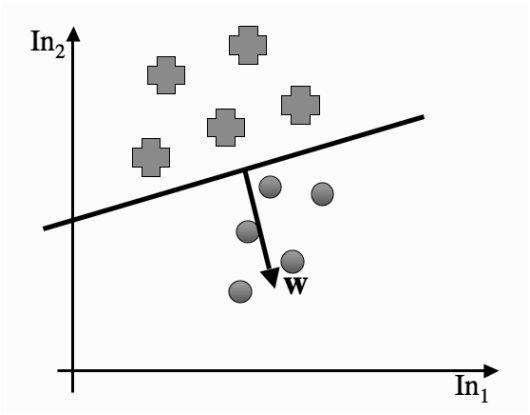

FIGURE 3.7 A decision boundary separating two classes of data.

vectors. It is computed by multiplying each element of the first vector by the matching element of the second and adding them all together. As you might remember from high school, $\mathbf{a} \cdot \mathbf{b} = \|a\|\|b\| \cos\theta$, where θ is the angle between $\mathbf{a}$ and $\mathbf{b}$ and $\|a\|$ is the length of the vector $\mathbf{a}$. So the inner product computes a function of the angle between the two vectors, scaled by their lengths. It can be computed in NumPy using the `np.inner()` function.

Getting back to the Perceptron, the boundary case is where we find an input vector $\mathbf{x}_1$ that has $\mathbf{x}_1 \cdot \mathbf{w}^T = 0$. Now suppose that we find another input vector $\mathbf{x}_2$ that satisfies $\mathbf{x}_2 \cdot \mathbf{w}^T = 0$. Putting these two equations together we get:

$$\mathbf{x}_1 \cdot \mathbf{w}^T = \mathbf{x}_2 \cdot \mathbf{w}^T \tag{3.16}$$

$$\Rightarrow (\mathbf{x}_1 - \mathbf{x}_2) \cdot \mathbf{w}^T = 0. \tag{3.17}$$

What does this last equation mean? In order for the inner product to be 0, either $\|a\|$ or $\|b\|$ or $\cos\theta$ needs to be zero. There is no reason to believe that $\|a\|$ or $\|b\|$ should be 0, so $\cos\theta = 0$. This means that $\theta = \pi/2$ (or $-\pi/2$), which means that the two vectors are at right angles to each other. Now $\mathbf{x}_1 - \mathbf{x}_2$ is a straight line between two points that lie on the decision boundary, and the weight vector $\mathbf{w}^T$ must be perpendicular to that, as in Figure 3.7.

So given some data, and the associated target outputs, the Perceptron simply tries to find a straight line that divides the examples where each neuron fires from those where it does not. This is great if that straight line exists, but is a bit of a problem otherwise. The cases where there is a straight line are called linearly separable cases. What happens if the classes that we want to learn about are not linearly separable? It turns out that making such a function is very easy: there is even one that matches a logic function. Before we have a look at it, it is worth thinking about what happens when we have more than one output neuron. The weights for each neuron separately describe a straight line, so by putting together several neurons we get several straight lines that each try to separate different parts of the space. Figure 3.8 shows an example of decision boundaries computed by a Perceptron with four neurons; by putting them together we can get good separation of the classes.

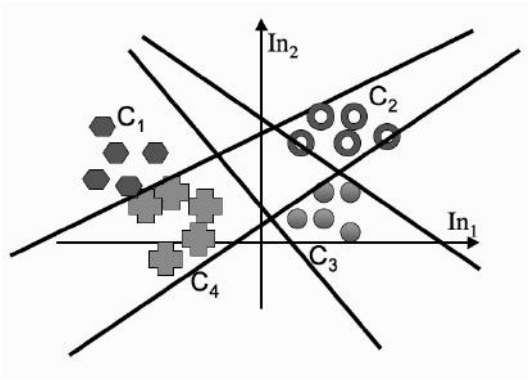

FIGURE 3.8 Different decision boundaries computed by a Perceptron with four neurons.

3.4.1 The Perceptron Convergence Theorem

Actually, it is not quite true that we have reached 1969. There is one more important fact that was known: Rosenblatt's 1962 proof that, given a linearly separable dataset, the Perceptron will converge to a solution that separates the classes, and that it will do it after a finite number of iterations. In fact, the number of iterations is bounded by $1/\gamma^2$, where γ is the distance between the separating hyperplane and the closest datapoint to it. The proof of this theorem only requires some algebra, and so we will work through it here. We will assume that the length of every input vector $\|\mathbf{x}\| \leq 1$, although it isn't strictly necessary provided that they are bounded by some constant R.

First, we know that there is some weight vector $\mathbf{w}^*$ that separates the data, since we have assumed that it is linearly separable. The Perceptron learning algorithm aims to find some vector $\mathbf{w}$ that is parallel to $\mathbf{w}^*$, or as close as possible. To see whether two vectors are parallel we use the inner product $\mathbf{w}^* \cdot \mathbf{w}$. When the two vectors are parallel, the angle between them is $\theta = 0$ and so $\cos\theta = 1$, and so the size of the inner product is a maximum. If we therefore show that at each weight update $\mathbf{w}^* \cdot \mathbf{w}$ increases, then we have nearly shown that the algorithm will converge. However, we do need a little bit more, because $\mathbf{w}^* \cdot \mathbf{w} = \|\mathbf{w}^*\|\|\mathbf{w}\|\cos\theta$, and so we also need to check that the length of $\mathbf{w}$ does not increase too much as well.

Hence, when we consider a weight update, there are two checks that we need to make: the value of $\mathbf{w}^* \cdot \mathbf{w}$ and the length of $\mathbf{w}$.

Suppose that at the tth iteration of the algorithm, the network sees a particular input $\mathbf{x}$ that should have output y, and that it gets this input wrong, so $y\mathbf{w}^{(t-1)} \cdot \mathbf{x} < 0$, where the $(t-1)$ index means the weights at the $(t-1)$st step. This means that the weights need to be updated. This weight update will be $\mathbf{w}^{(t)} = \mathbf{w}^{(t-1)} + y\mathbf{x}$ (where we have set $\eta = 1$ for simplicity, and because it is fine for the Perceptron.

To see how this changes the two values we are interested in, we need to do some computation:

$$\begin{aligned} \mathbf{w}^* \cdot \mathbf{w}^{(t)} &= \mathbf{w}^* \cdot \left(\mathbf{w}^{(t-1)} + y\mathbf{x}\right) \\ &= \mathbf{w}^* \cdot \mathbf{w}^{(t-1)} + y\mathbf{w}^* \cdot \mathbf{x} \\ &\geq \mathbf{w}^* \cdot \mathbf{w}^{(t-1)} + \gamma \end{aligned} \tag{3.18}$$

In_1	In_2	t
0	0	0
0	1	1
1	0	1
1	1	0

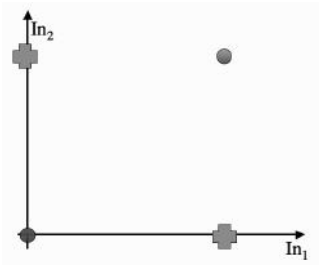

FIGURE 3.9 Data for the XOR logic function and a plot of the four datapoints.

where γ is that smallest distance between the optimal hyperplane defined by $\mathbf{w}^*$ and any datapoint.

This means that at each update of the weights, this inner product increases by at least γ, and so after t updates of the weights, $\mathbf{w}^* \cdot \mathbf{w}^{(t)} \geq t\gamma$. We can use this to put a lower bound on the length of $\|\mathbf{w}^{(t)}\|$ by using the Cauchy–Schwartz inequality, which tells us that $\mathbf{w}^* \cdot \mathbf{w}^{(t)} \leq \|\mathbf{w}^*\|\|\mathbf{w}^{(t)}\|$ and so $\|\mathbf{w}^{(t)}\| \geq t\gamma$.

The length of the weight vector after t steps is:

$$\begin{aligned} \|\mathbf{w}^{(t)}\|^2 &= \|\mathbf{w}^{(t-1)} + y\mathbf{x}\|^2 \\ &= \|\mathbf{w}^{(t-1)}\|^2 + y^2\|\mathbf{x}\|^2 + 2y\mathbf{w}^{(t-1)} \cdot \mathbf{x} \\ &\leq \|\mathbf{w}^{(t-1)}\|^2 + 1 \end{aligned} \tag{3.19}$$

where the last line follows because $y^2 = 1$, $\|\mathbf{x}\| \leq 1$, and the network made an error, so the $\mathbf{w}^{(t-1)}$ and $\mathbf{x}$ are perpendicular to each other. This tells us that after t steps, $\|\mathbf{w}^{(t)}\|^2 \leq k$.

We can put these two inequalities together to get that:

$$t\gamma \leq \|\mathbf{w}^{(t-1)}\| \leq \sqrt{t}, \tag{3.20}$$

and so $t \leq 1/\gamma^2$. Hence after we have made that many updates the algorithm must have converged.

We have shown that if the weights are linearly separable then the algorithm will converge, and that the time that this takes is a function of the distance between the separating hyperplane and the nearest datapoint. This is called the margin, and in Chapter 8 we will see an algorithm that uses this explicitly. Note that the Perceptron stops learning as soon as it gets all of the training data correct, and so there is no guarantee that it will find the largest margin, just that if there is a linear separator, it will find it. Further, we still don't know what happens if the data are not linearly separable. To see that, we will move on to just such an example.

3.4.2 The Exclusive Or (XOR) Function

The XOR has the same four input points as the OR function, but looking at Figure 3.9, you should be able to convince yourself that you can't draw a straight line on the graph that separates true from false (crosses from circles). In our new language, the XOR function is not linearly separable. If the analysis above is correct, then the Perceptron will fail to get the correct answer, and using the Perceptron code above we find:

```
>>> targets = np.array([[0],[1],[1],[0]])
>>> pcn.pcntrain(inputs,targets,0.25,15)
```

which gives the following output (the early iterations have been missed out):

```
Iteration:  11
[[ 0.45946905]
 [-0.27886266]
 [-0.25662428]]
Iteration:  12
[[-0.04053095]
 [-0.02886266]
 [-0.00662428]]
Iteration:  13
[[ 0.45946905]
 [-0.27886266]
 [-0.25662428]]
Iteration:  14
[[-0.04053095]
 [-0.02886266]
 [-0.00662428]]
Final outputs are:
[[0]
 [0]
 [0]
 [0]]
```

You can see that the algorithm does not converge, but keeps on cycling through two different wrong solutions. Running it for longer does not change this behaviour. So even for a simple logical function, the Perceptron can fail to learn the correct answer. This is what was demonstrated by Minsky and Papert in "Perceptrons," and the discovery that the Perceptron was not capable of solving even these problems, let alone more interesting ones, is what halted neural network development for so long. There is an obvious solution to the problem, which is to make the network more complicated—add in more neurons, with more complicated connections between them, and see if that helps. The trouble is that this makes the problem of training the network much more difficult. In fact, working out how to do that is the topic of the next chapter.

3.4.3 A Useful Insight

From the discussion in Section 3.4.2 you might think that the XOR function is impossible to solve using a linear function. In fact, this is not true. If we rewrite the problem in three dimensions instead of two, then it is perfectly possible to find a plane (the 2D analogue of a straight line) that can separate the two classes. There is a picture of this in Figure 3.10. Writing the problem in 3D means including a third input dimension that does not change the data when it is looked at in the (x, y) plane, but moves the point at $(0, 0)$ along a third

In_1	In_2	In_3	Output
0	0	1	1
0	1	0	0
1	0	0	0
1	1	0	1

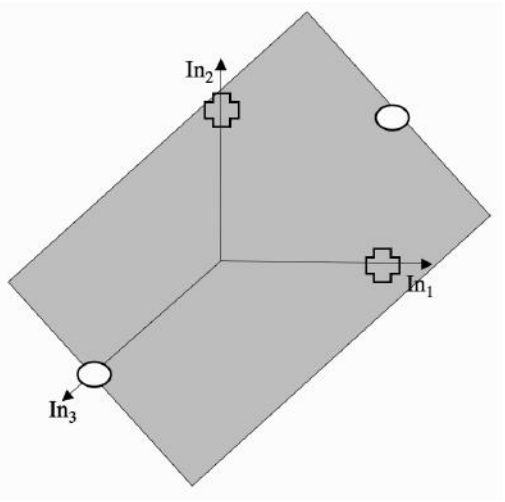

FIGURE 3.10 A decision boundary (the shaded plane) solving the XOR problem in 3D with the crosses below the surface and the circles above it.

dimension. So the truth table for the function is the one shown on the left side of Figure 3.10 (where 'In_3' has been added, and only affects the point at $(0, 0)$).

To demonstrate this, the following listing uses the same Perceptron code:

```
>>> inputs = np.array([[0,0,1],[0,1,0],[1,0,0],[1,1,0]])
>>> pcn.pcntrain(inputs,targets,0.25,15)
Iteration:  14
[[-0.27757663]
 [-0.21083089]
 [-0.23124407]
 [-0.53808657]]
Final outputs are:
[[0]
 [1]
 [1]
 [0]]
```

In fact, it is always possible to separate out two classes with a linear function, provided that you project the data into the correct set of dimensions. There is a whole class of methods for doing this reasonably efficiently, called kernel classifiers, which are the basis of Support Vector Machines, which are the subject of Chapter 8.

For now, it is sufficient to point out that if you want to make your linear Perceptron do non-linear things, then there is nothing to stop you making non-linear variables. For example, Figure 3.11 shows two versions of the same dataset. On the left side, the coordinates are x_1 and x_2, while on the right side the coordinates are x_1, x_2 and $x_1 \times x_2$. It is now easy to fit a plane (the 2D equivalent of a straight line) that separates the data.

Statistics has been dealing with problems of classification and regression for a long time, before we had computers in order to do difficult arithmetic for us, and so straight

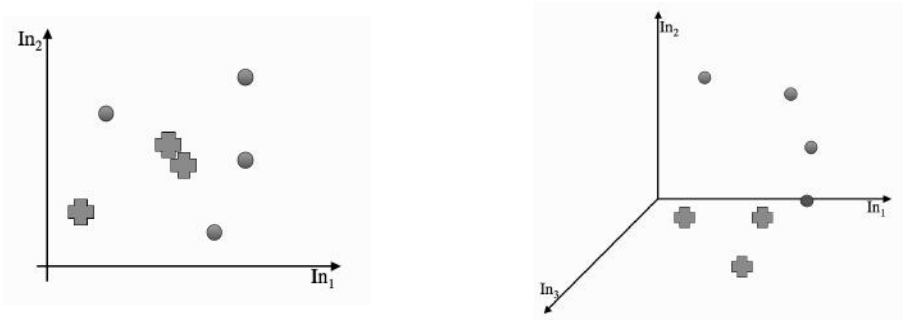

FIGURE 3.11 *Left:* Non-separable 2D dataset. *Right:* The same dataset with third coordinate $x_1 \times x_2$, which makes it separable.

line methods have been around in statistics for many years. They provide a different (and useful) way to understand what is happening in learning, and by using both statistical and computer science methods we can get a good understanding of the whole area. We will see the statistical method of linear regression in Section 3.5, but first we will work through another example of using the Perceptron. This is meant to be a tutorial example, so I will give some of the relevant code and results, but leave places for you to fill in the gaps.

3.4.4 Another Example: The Pima Indian Dataset

The UCI Machine Learning Repository (`http://archive.ics.uci.edu/ml/`) holds lots of datasets that are used to demonstrate and test machine learning algorithms. For the purposes of testing out the Perceptron and Linear Regressor, we are going to use one that is very well known. It provides eight measurements of a group of American Pima Indians living in Arizona in the USA, and the classification is whether or not each person had diabetes. The dataset is available from the UCI repository (called `Pima`) and there is a file inside the folder giving details of what the different variables mean.

Once you have downloaded it, import the relevant modules (NumPy to use the array methods, PyLab to plot the data, and the Perceptron from the book website) and then load the data into Python. This requires something like the following (where not all of the `import` lines are used immediately, but will be required as more code is developed):

```
>>> import os
>>> import pylab as pl
>>> import numpy as np
>>> import pcn

>>> os.chdir('/Users/srmarsla/Book/Datasets/pima')
>>> pima = np.loadtxt('pima-indians-diabetes.data',delimiter=',')
>>> np.shape(pima)
(768, 9)
```

where the path in the `os.chdir` line will obviously need to be changed to wherever you have saved the dataset. In the `np.loadtxt()` command the `delimiter` specifies which character is used to separate out the datapoints. The `np.shape()` method tells that there are 768

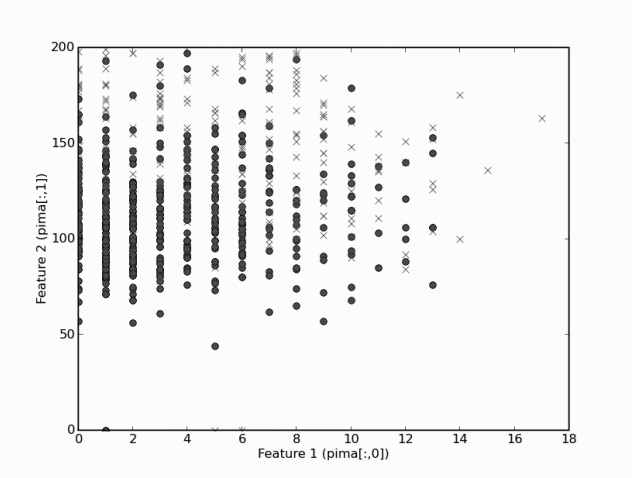

FIGURE 3.12 Plot of the first two dimensions of the Pima Indians dataset showing the two classes as 'x' and '○'.

datapoints, arranged as rows of the file, with each row containing nine numbers. These are the eight dimensions of data, with the class being the ninth element of each line (indexed as 8 since Python is zero-indexed). This arrangement, with each line of a file (or row of an array) being a datapoint is the one that will be used throughout the book.

You should have a look at the dataset. Obviously, you can't plot the whole thing at once, since that would require being able to visualise eight dimensions. But you can plot any two-dimensional subset of the data. Have a look at a few of them. In order to see the two different classes in the data in your plot, you will have to work out how to use the `np.where` command. Once you have worked that out, you will be able to plot them with different shapes and colours. The `pl.plot` command is in Matplotlib, so you'll need to import that (using `import pylab as pl`) beforehand. Assuming that you have worked out some way to store the indices of one class in `indices0` and the other in `indices1` you can use:

```
pl.ion()
pl.plot(pima[indices0,0],pima[indices0,1],'go')
pl.plot(pima[indices1,0],pima[indices1,1],'rx')
pl.show()
```

to plot the first two dimensions as green circles and red crosses, which (up to colour, of course) should look like Figure 3.12. The `pl.ion()` command ensures that the data is actually plotted, and might not be needed depending upon your precise software setup; this is also true of the `pl.show()` command, which ensures that the graph does not vanish when the program terminates. Clearly, there is no way that you can find a linear separation between these two classes with these features. However, you should have a look at some of the other combinations of features and see if you can find any that are better.

The next thing to do is to try using the Perceptron on the full dataset. You will need to try out different values for the learning rate and the number of iterations for the Perceptron, but you should find that you can get around 50-70% correct (use the confusion matrix

method `confmat()` to get the results). This isn't too bad, but it isn't that good, either. The results are quite unstable, too; sometimes the results have only 30% accuracy—worse than chance—which is rather depressing.

```
p = pcn.pcn(pima[:,:8],pima[:,8:9])
p.pcntrain(pima[:,:8],pima[:,8:9],0.25,100)
p.confmat(pima[:,:8],pima[:,8:9])
```

This is, of course, unfair testing, since we are testing the network on the same data we were training it on, and we have already seen that this is unfair in Section 2.2, but we will do something quick now, which is to use even-numbered datapoints for training, and odd-numbered datapoints for testing. This is very easy using the `:` operator, where we specify the start point, the end point, and the step size. NumPy will fill in any that we leave blank with the beginning or end of the array as appropriate.

```
trainin = pima[::2,:8]
testin = pima[1::2,:8]
traintgt = pima[::2,8:9]
testtgt = pima[1::2,8:9]
```

For now, rather than worrying about training and testing data, we are more interested in working out how to improve the results. And we can do better by preparing the data a little, or preprocessing it.

3.4.5 Preprocessing: Data Preparation

Machine learning algorithms tend to learn much more effectively if the inputs and targets are prepared for analysis before the network is trained. As the most basic example, the neurons that we are using give outputs of 0 and 1, and so if the target values are not 0 and 1, then they should be transformed so that they are. In fact, it is normal to scale the targets to lie between 0 and 1 no matter what kind of activation function is used for the output layer neurons. This helps to stop the weights from getting too large unnecessarily. Scaling the inputs also helps to avoid this problem.

The most common approach to scaling the input data is to treat each data dimension independently, and then to either make each dimension have zero mean and unit variance in each dimension, or simply to scale them so that maximum value is 1 and the minimum -1. Both of these scalings have similar effects, but the first is a little bit better as it does not allow outliers to dominate as much. These scalings are commonly referred to as data normalisation, or sometimes standardisation. While normalisation is not essential for every algorithm, but it is usually beneficial, and for some of the other algorithms that we will see, the normalisation will be essential.

In NumPy it is very easy to perform the normalisation by using the built-in `np.mean()` and `np.var()` functions; the only place where care is needed is along which axis the mean and variance are computed: `axis=0` sums down the columns and `axis=1` sums across the rows. Note that only the input variables are normalised in this code. This is not always true, but here the target variable already has values 0 and 1, which are the possible outputs for the Perceptron, and we don't want to change that.

```
data = (data - data.mean(axis=0))/data.var(axis=0)
targets = (targets - targets.mean(axis=0))/targets.var(axis=0)
```

There is one thing to be careful of, which is that if you normalise the training and testing sets separately in this way then a datapoint that is in both sets will end up being different in the two, since the mean and variance are probably different in the two sets. For this reason it is a good idea to normalise the dataset before splitting it into training and testing.

Normalisation can be done without knowing anything about the dataset in advance. However, there is often useful preprocessing that can be done by looking at the data. For example, in the Pima dataset, column 0 is the number of times that the person has been pregnant (did I mention that all the subjects were female?) and column 7 is the age of the person. Taking the pregnancy variable first, there are relatively few subjects that were pregnant 8 or more times, so rather than having the number there, maybe they should be replaced by an 8 for any of these values. Equally, the age would be better quantised into a set of ranges such as 21–30, 31–40, etc. (the minimum age is 21 in the dataset). This can be done using the `np.where` function again, as in this code snippet. If you make these changes and similar ones for the other values, then you should be able to get massively better results.

```
pima[np.where(pima[:,0]>8),0] = 8

pima[np.where(pima[:,7]<=30),7] = 1
pima[np.where((pima[:,7]>30) & (pima[:,7]<=40)),7] = 2
#You need to finish this data processing step
```

The last thing that we can do for now is to perform a basic form of feature selection and to try training the classifier with a subset of the inputs by missing out different features one at a time and seeing if they make the results better. If missing out one feature does improve the results, then leave it out completely and try missing out others as well. This is a simplistic way of testing for correlation between the output and each of the features. We will see better methods when we look at covariance in Section 2.4.2. We can also consider methods of dimensionality reduction, which produce lower dimensionsal representations of the data that still include the relevant information; see Chapter 6 for more details.

Now that we have seen how to use the Perceptron on a better example than the logic functions, we will look at another linear method, but coming from statistics, rather than neural networks.

3.5 LINEAR REGRESSION

As is common in statistics, we need to separate out regression problems, where we fit a line to data, from classification problems, where we find a line that separates out the classes, so that they can be distinguished. However, it is common to turn classification problems into regression problems. This can be done in two ways, first by introducing an indicator variable, which simply says which class each datapoint belongs to. The problem is now to use the data to predict the indicator variable, which is a regression problem. The second approach is to do repeated regression, once for each class, with the indicator value being 1

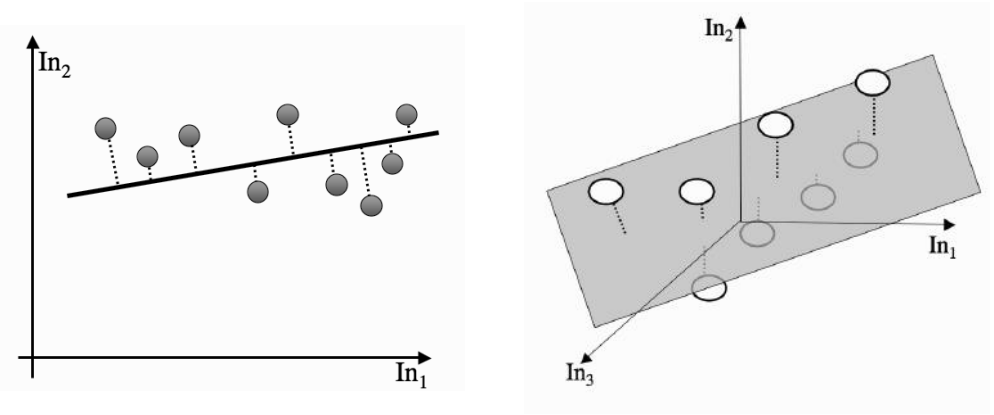

FIGURE 3.13 Linear regression in two and three dimensions.

for examples in the class and 0 for all of the others. Since classification can be replaced by regression using these methods, we'll think about regression here.

The only real difference between the Perceptron and more statistical approaches is in the way that the problem is set up. For regression we are making a prediction about an unknown value y (such as the indicator variable for classes or a future value of some data) by computing some function of known values x_i. We are thinking about straight lines, so the output y is going to be a sum of the x_i values, each multiplied by a constant parameter: $y = \sum_{i=0}^{M} \beta_i x_i$. The β_i define a straight line (plane in 3D, hyperplane in higher dimensions) that goes through (or at least near) the datapoints. Figure 3.13 shows this in two and three dimensions.

The question is how we define the line (plane or hyperplane in higher dimensions) that best fits the data. The most common solution is to try to minimise the distance between each datapoint and the line that we fit. We can measure the distance between a point and a line by defining another line that goes through the point and hits the line. School geometry tells us that this second line will be shortest when it hits the line at right angles, and then we can use Pythagoras' theorem to know the distance. Now, we can try to minimise an error function that measures the sum of all these distances. If we ignore the square roots, and just minimise the sum-of-squares of the errors, then we get the most common minimisation, which is known as least-squares optimisation. What we are doing is choosing the parameters in order to minimise the squared difference between the prediction and the actual data value, summed over all of the datapoints. That is, we have:

$$\sum_{j=0}^{N} \left(t_j - \sum_{i=0}^{M} \beta_i x_{ij} \right)^2 . \tag{3.21}$$

This can be written in matrix form as:

$$(\mathbf{t} - \mathbf{X}\boldsymbol{\beta})^T (\mathbf{t} - \mathbf{X}\boldsymbol{\beta}), \tag{3.22}$$

where $\mathbf{t}$ is a column vector containing the targets and $\mathbf{X}$ is the matrix of input values (even including the bias inputs), just as for the Perceptron. Computing the smallest value of this means differentiating it with respect to the (column) parameter vector $\boldsymbol{\beta}$ and setting the derivative to 0, which means that $\mathbf{X}^T(\mathbf{t} - \mathbf{X}\boldsymbol{\beta}) = 0$ (to see this, expand out the brackets, remembering that $\mathbf{A}\mathbf{B}^T = \mathbf{B}^T\mathbf{A}$ and note that the term $\boldsymbol{\beta}^T\mathbf{X}^t\mathbf{t} = \mathbf{t}^T\mathbf{X}\boldsymbol{\beta}$ since they are

both a scalar term), which has the solution $\boldsymbol{\beta} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{t}$ (assuming that the matrix $\mathbf{X}^T\mathbf{X}$ can be inverted). Now, for a given input vector $\mathbf{z}$, the prediction is $\mathbf{z}\boldsymbol{\beta}$. The inverse of a matrix $\mathbf{X}$ is the matrix that satisfies $\mathbf{X}\mathbf{X}^{-1} = \mathbf{I}$, where $\mathbf{I}$ is the identity matrix, the matrix that has 1s on the leading diagonal and 0s everywhere else. The inverse of a matrix only exists if the matrix is square (has the same number of rows as columns) and its determinant is non-zero.

Computing this is very simple in Python, using the `np.linalg.inv()` function in NumPy. In fact, the entire function can be written as (where the ↩ symbol denotes a linebreak in the text, so that the command continues on the next line):

```
def linreg(inputs,targets):
        inputs = np.concatenate((inputs,-np.ones((np.shape(inputs)[0],1))),↩
        axis=1)
        beta = np.dot(np.dot(np.linalg.inv(np.dot(np.transpose(inputs),↩
        inputs)),np.transpose(inputs)),targets)

        outputs = np.dot(inputs,beta)
```

3.5.1 Linear Regression Examples

Using the linear regressor on the logical OR function seems a rather strange thing to do, since we are performing classification using a method designed explicitly for regression, trying to fit a surface to a set of 0 and 1 points. Worse, we will view it as an error if we get say 1.25 and the output should be 1, so points that are in some sense too correct will receive a penalty! However, we can do it, and it gives the following outputs:

```
[[ 0.25]
 [ 0.75]
 [ 0.75]
 [ 1.25]]
```

It might not be clear what this means, but if we threshold the outputs by setting every value less than 0.5 to 0 and every value above 0.5 to 1, then we get the correct answer. Using it on the XOR function shows that this is still a linear method:

```
[[ 0.5]
 [ 0.5]
 [ 0.5]
 [ 0.5]]
```

A better test of linear regression is to find a real regression dataset. The UCI database is useful here, as well. We will look at the `auto-mpg` dataset. This consists of a collection of a number of datapoints about certain cars (weight, horsepower, etc.), with the aim being to predict the fuel efficiency in miles per gallon (mpg). This dataset has one problem. There are

missing values in it (labelled with question marks '?'). The `np.loadtxt()` method doesn't like these, and we don't know what to do with them, anyway, so after downloading the dataset, manually edit the file and delete all lines where there is a ? in that line. The linear regressor can't do much with the names of the cars either, but since they appear in quotes (") we will tell `np.loadtxt` that they are comments, using:

```
auto = np.loadtxt('/Users/srmarsla/Book/Datasets/auto-mpg/auto-mpg.data.txt',
comments='"')
```

You should now separate the data into training and testing sets, and then use the training set to recover the β vector. Then you use that to get the predicted values on the test set. However, the confusion matrix isn't much use now, since there are no classes to enable us to analyse the results. Instead, we will use the sum-of-squares error, which consists of computing the difference between the prediction and the true value, squaring them so that they are all positive, and then adding them up, as is used in the definition of the linear regressor. Obviously, small values of this measure are good. It can be computed using:

```
beta = linreg.linreg(trainin,traintgt)

testin = np.concatenate((testin,-np.ones((np.shape(testin)[0],1))),axis=1)
testout = np.dot(testin,beta)
error = np.sum((testout - testtgt)**2)
```

Now you can test out whether normalising the data helps, and perform feature selection as we did for the Perceptron. There are other more advanced linear statistical methods. One of them, Linear Discriminant Analysis, will be considered in Section 6.1 once we have built up the understanding we need.

FURTHER READING

If you are interested in real brains and want to know more about them, then there are plenty of popular science books that should interest you, including:

- Susan Greenfield. *The Human Brain: A Guided Tour*. Orion, London, UK, 2001.
- S. Aamodt and S. Wang. *Welcome to Your Brain: Why You Lose Your Car Keys but Never Forget How to Drive and Other Puzzles of Everyday Life*. Bloomsbury, London, UK, 2008.

If you are looking for something a bit more formal, then the following is a good place to start (particularly the 'Roadmaps' at the beginning):

- Michael A. Arbib, editor. *The Handbook of Brain Theory and Neural Networks*, 2nd edition, MIT Press, Cambridge, MA, USA, 2002.

The original paper by McCulloch and Pitts is:

- W.S. McCulloch and W. Pitts. A logical calculus of ideas imminent in nervous activity. *Bulletin of Mathematics Biophysics*, 5:115–133, 1943.

There is a very nice motivation for neural network-based learning in:

- V. Braitenberg. *Vehicles: Experiments in Synthetic Psychology.* MIT Press, Cambridge, MA, USA, 1984.

If you want to know more about the history of neural networks, then the original paper on the Perceptron and the book that showed the requirement of linear separability (and that some people blame for putting the field back 20 years) still make interesting reads. Another paper that might be of interest is the review article written by Widrow and Lehr, which summarises some of the seminal work:

- F. Rosenblatt. The Perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, 1958.
- M.L. Minsky and S.A. Papert. *Perceptrons: An Introduction to Computational Geometry.* MIT Press, Cambridge MA, 1969.
- B. Widrow and M.A. Lehr. 30 years of adaptive neural networks: Perceptron, madaline, and backpropagation. *Proceedings of the IEEE*, 78(9):1415–1442, 1990.

Textbooks that cover the same material, although from different viewpoints, include:

- Chapter 5 of R.O. Duda, P.E. Hart, and D.G. Stork. *Pattern Classification*, 2nd edition, Wiley-Interscience, New York, USA, 2001.
- Sections 3.1–3.3 of T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*, 2nd edition, Springer, Berlin, Germany, 2008.

PRACTICE QUESTIONS

Problem 3.1 Consider a neuron with 2 inputs, 1 output, and a threshold activation function. If the two weights are $w_1 = 1$ and $w_2 = 1$, and the bias is $b = -1.5$, then what is the output for input $(0, 0)$? What about for inputs $(1, 0)$, $(0, 1)$, and $(1, 1)$?

Draw the discriminant function for this function, and write down its equation. Does it correspond to any particular logic gate?

Problem 3.2 Work out the Perceptrons that construct logical NOT, NAND, and NOR of their inputs.

Problem 3.3 The parity problem returns 1 if the number of inputs that are 1 is even, and 0 otherwise. Can a Perceptron learn this problem for 3 inputs? Design the network and try it.

Problem 3.4 Test out both the Perceptron and linear regressor code from the website on the parity problem.

Problem 3.5 The Perceptron code on the website is a batch update algorithm, where the whole of the dataset is fed in to find the errors, and then the weights are updated afterwards, as is discussed in Section 3.3.5. Convert the code to run as sequential updates and then compare the results of using the two versions.

Problem 3.6 Try to think of some interesting image processing tasks that cannot be performed by a Perceptron. (Hint: You need to think of tasks where looking at individual pixels isn't enough to allow classification.)

Problem 3.7 The decision boundary hyperplane found by the Perceptron has equation $y(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + b = 0$. For a point $\mathbf{x}'$, minimise $\|\mathbf{x} - \mathbf{x}'\|^2$ to show that the shortest distance from the point to the hyperplane is $|y(\mathbf{x}')|/\|\mathbf{w}\|$.

Problem 3.8 There is a link to a very large dataset of handwritten figures on the book website (the MNIST dataset). Download it and use a Perceptron to learn about the dataset.

Problem 3.9 For the prostate data available via the website, use both the Perceptron and logistic regressor and compare the results.

Problem 3.10 In the Perceptron Convergence Theorem proof we assumed that $\|\mathbf{x}\| \leq 1$. Modify the proof so that it only assumes that $\|\mathbf{x}\| \leq R$ for some constant R.

The Multi-layer Perceptron

In the last chapter we saw that while linear models are easy to understand and use, they come with the inherent cost that is implied by the word 'linear'; that is, they can only identify straight lines, planes, or hyperplanes. And this is not usually enough, because the majority of interesting problems are not linearly separable. In Section 3.4 we saw that problems can be made linearly separable if we can work out how to transform the features suitably. We will come back to this idea in Chapter 8, but in this chapter we will instead consider making more complicated networks.

We have pretty much decided that the learning in the neural network happens in the weights. So, to perform more computation it seems sensible to add more weights. There are two things that we can do: add some backwards connections, so that the output neurons connect to the inputs again, or add more neurons. The first approach leads into recurrent networks. These have been studied, but are not that commonly used. We will instead consider the second approach. We can add neurons between the input nodes and the outputs, and this will make more complex neural networks, such as the one shown in Figure 4.1.

We will think about why adding extra layers of nodes makes a neural network more powerful in Section 4.3.2, but for now, to persuade ourselves that it is true, we can check that a prepared network can solve the two-dimensional XOR problem, something that we have seen is not possible for a linear model like the Perceptron. A suitable network is shown in Figure 4.2. To check that it gives the correct answers, all that is required is to put in each input and work through the network, treating it as two different Perceptrons, first computing the activations of the neurons in the middle layer (labelled as C and D in Figure 4.2) and then using those activations as the inputs to the single neuron at the output. As an example, I'll work out what happens when you put in $(1, 0)$ as an input; the job of checking the rest is up to you.

Input $(1, 0)$ corresponds to node A being 1 and B being 0. The input to neuron C is therefore $-1 \times 0.5 + 1 \times 1 + 0 \times 1 = -0.5 + 1 = 0.5$. This is above the threshold of 0, and so neuron C fires, giving output 1. For neuron D the input is $-1 \times 1 + 1 \times 1 + 0 \times 1 = -1 + 1 = 0$, and so it does not fire, giving output 0. Therefore the input to neuron E is $-1 \times 0.5 + 1 \times 1 + 0 \times -1 = 0.5$, so neuron E fires. Checking the result of the inputs should persuade you that neuron E fires when inputs A and B are different to each other, but does not fire when they are the same, which is exactly the XOR function (it doesn't matter that the fire and not fire have been reversed).

So far, so good. Since this network can solve a problem that the Perceptron cannot, it seems worth looking into further. However, now we've got a much more interesting problem to solve, namely how can we train this network so that the weights are adapted to generate the correct (target) answers? If we try the method that we used for the Perceptron we need

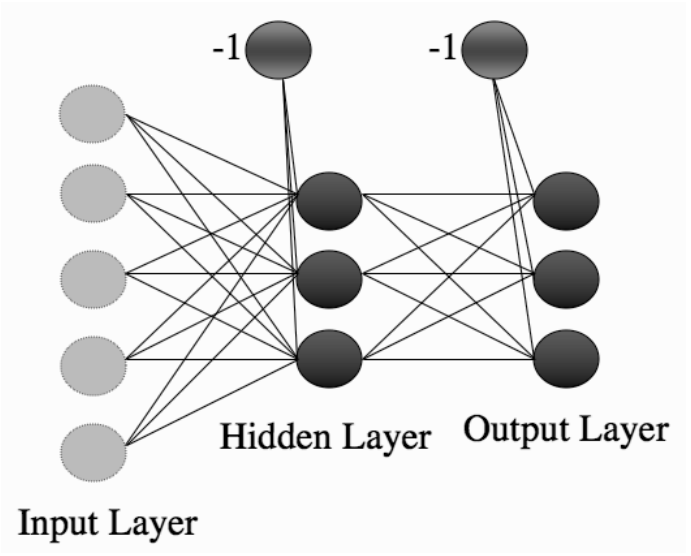

FIGURE 4.1 The Multi-layer Perceptron network, consisting of multiple layers of connected neurons.

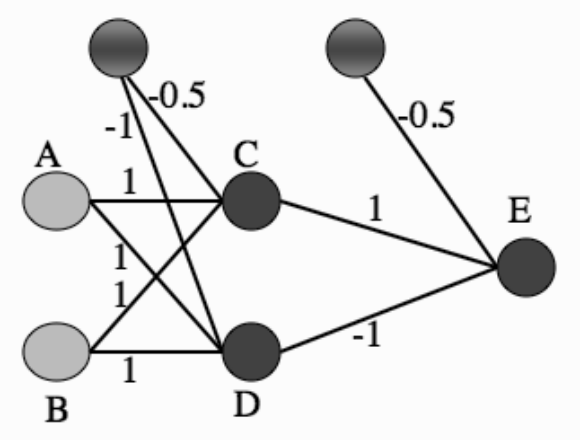

FIGURE 4.2 A Multi-layer Perceptron network showing a set of weights that solve the XOR problem.

to compute the error at the output. That's fine, since we know the targets there, so we can compute the difference between the targets and the outputs. But now we don't know which weights were wrong: those in the first layer, or the second? Worse, we don't know what the correct activations are for the neurons in the middle of the network. This fact gives the neurons in the middle of the network their name; they are called the hidden layer (or layers), because it isn't possible to examine and correct their values directly.

It took a long time for people who studied neural networks to work out how to solve this problem. In fact, it wasn't until 1986 that Rumelhart, Hinton, and McClelland managed it. However, a solution to the problem was already known by statisticians and engineers—they just didn't know that it was a problem in neural networks! In this chapter we are going to look at the neural network solution proposed by Rumelhart, Hinton, and McClelland, the Multi-layer Perceptron (MLP), which is still one of the most commonly used machine learning methods around. The MLP is one of the most common neural networks in use. It is often treated as a 'black box', in that people use it without understanding how it works, which often results in fairly poor results. Getting to the stage where we understand how it works and what we can do with it is going to take us into lots of different areas of statistics, mathematics, and computer science, so we'd better get started.

4.1 GOING FORWARDS

Just as it did for the Perceptron, training the MLP consists of two parts: working out what the outputs are for the given inputs and the current weights, and then updating the weights according to the error, which is a function of the difference between the outputs and the targets. These are generally known as going forwards and backwards through the network. We've already seen how to go forwards for the MLP when we saw the XOR example above, which was effectively the recall phase of the algorithm. It is pretty much just the same as the Perceptron, except that we have to do it twice, once for each set of neurons, and we need to do it layer by layer, because otherwise the input values to the second layer don't exist. In fact, having made an MLP with two layers of nodes, there is no reason why we can't make one with 3, or 4, or 20 layers of nodes (we'll discuss whether or not you might want to in Section 4.3.2). This won't even change our recall (forward) algorithm much, since we just work forwards through the network computing the activations of one layer of neurons and using those as the inputs for the next layer.

So looking at Figure 4.1, we start at the left by filling in the values for the inputs. We then use these inputs and the first level of weights to calculate the activations of the hidden layer, and then we use those activations and the next set of weights to calculate the activations of the output layer. Now that we've got the outputs of the network, we can compare them to the targets and compute the error.

4.1.1 Biases

We need to include a bias input to each neuron. We do this in the same way as we did for the Perceptron in Section 3.3.2, by having an extra input that is permanently set to -1, and adjusting the weights to each neuron as part of the training. Thus, each neuron in the network (whether it is a hidden layer or the output) has 1 extra input, with fixed value.

4.2 GOING BACKWARDS: BACK-PROPAGATION OF ERROR

It is in the backwards part of the algorithm that things get tricky. Computing the errors at the output is no more difficult than it was for the Perceptron, but working out what to do with those errors is more difficult. The method that we are going to look at is called back-propagation of error, which makes it clear that the errors are sent backwards through the network. It is a form of gradient descent (which is described briefly below, and also given its own section in Chapter 9; in that chapter, in Section 9.3.2, we will see how to use the general gradient descent algorithms for the MLP).

The best way to describe back-propagation properly is mathematically, but this can be intimidating and difficult to get a handle on at first. I've therefore tried to compromise by using words and pictures in the main text, but putting all of the mathematical details into Section 4.6. While you should look at that section and try to understand it, it can be skipped if you really don't have the background. Although it looks complicated, there are actually just three things that you need to know, all of which are from differential calculus: the derivative of $\frac{1}{2}x^2$, the fact that if you differentiate a function of x with respect to some other variable t, then the answer is 0, and the chain rule, which tells you how to differentiate composite functions.

When we talked about the Perceptron, we changed the weights so that the neurons fired when the targets said they should, and didn't fire when the targets said they shouldn't. What we did was to choose an error function for each neuron k: $E_k = y_k - t_k$, and tried to make it as small as possible. Since there was only one set of weights in the network, this was sufficient to train the network.

We still want to do the same thing—minimise the error, so that neurons fire only when they should—but, with the addition of extra layers of weights, this is harder to arrange. The problem is that when we try to adapt the weights of the Multi-layer Perceptron, we have to work out which weights caused the error. This could be the weights connecting the inputs to the hidden layer, or the weights connecting the hidden layer to the output layer. (For more complex networks, there could be extra weights between nodes in hidden layers. This isn't a problem—the same method works—but it is more confusing to talk about, so I'm only going to worry about one hidden layer here.)

The error function that we used for the Perceptron was $\sum_{k=1}^{N} E_k = \sum_{k=1}^{N} y_k - t_k$, where N is the number of output nodes. However, suppose that we make two errors. In the first, the target is bigger than the output, while in the second the output is bigger than the target. If these two errors are the same size, then if we add them up we could get 0, which means that the error value suggests that no error was made. To get around this we need to make all errors have the same sign. We can do this in a few different ways, but the one that will turn out to be best is the sum-of-squares error function, which calculates the difference between y and t for each node, squares them, and adds them all together:

$$E(\mathbf{t}, \mathbf{y}) = \frac{1}{2} \sum_{k=1}^{N} (y_k - t_k)^2. \tag{4.1}$$

You might have noticed the $\frac{1}{2}$ at the front of that equation. It doesn't matter that much, but it makes it easier when we differentiate the function, and that is the name of the game here: if we differentiate a function, then it tells us the gradient of that function, which is the direction along which it increases and decreases the most. So if we differentiate an error function, we get the gradient of the error. Since the purpose of learning is to minimise the error, following the error function downhill (in other words, in the direction of the negative gradient) will give us what we want. Imagine a ball rolling around on a surface that looks

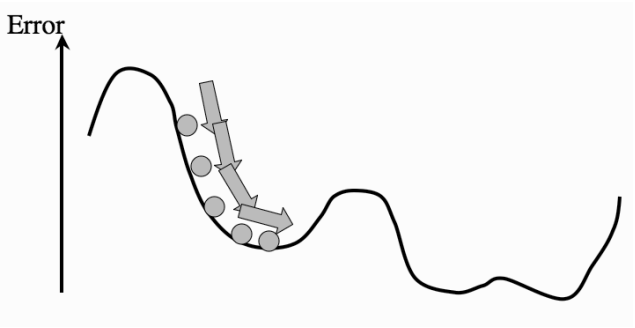

FIGURE 4.3 The weights of the network are trained so that the error goes downhill until it reaches a local minimum, just like a ball rolling under gravity.

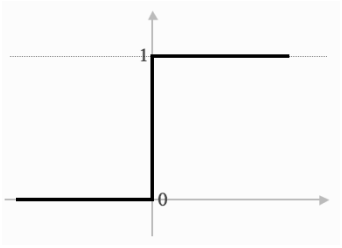

FIGURE 4.4 The threshold function that we used for the Perceptron. Note the discontinuity where the value changes from 0 to 1.

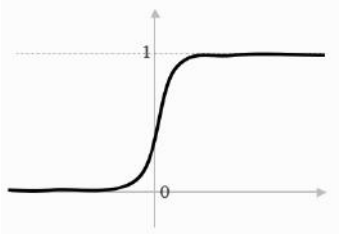

FIGURE 4.5 The sigmoid function, which looks qualitatively fairly similar, but varies smoothly and differentiably.

like the line in Figure 4.3. Gravity will make the ball roll downhill (follow the downhill gradient) until it ends up in the bottom of one of the hollows. These are places where the error is small, so that is exactly what we want. This is why the algorithm is called gradient descent. So what should we differentiate with respect to? There are only three things in the network that change: the inputs, the activation function that decides whether or not the node fires, and the weights. The first and second are out of our control when the algorithm is running, so only the weights matter, and therefore they are what we differentiate with respect to.

Having mentioned the activation function, this is a good time to point out a little problem with the threshold function that we have been using for our neurons so far, which is that it is discontinuous (see Figure 4.4; it has a sudden jump in the middle) and so differentiating it at that point isn't possible. The problem is that we need that jump between firing and not firing to make it act like a neuron. We can solve the problem if we can find an activation function that looks like a threshold function, but is differentiable so that we can compute the gradient. If you squint at a graph of the threshold function (for example, Figure 4.4) then it looks kind of S-shaped. There is a mathematical form of S-shaped functions, called sigmoid functions (see Figure 4.5). They have another nice property, which is that their derivative also has a nice form, as is shown in Section 4.6.3 for those who know some mathematics. The most commonly used form of this function (where β is some positive parameter) is:

$$a = g(h) = \frac{1}{1 + \exp(-\beta h)}. \tag{4.2}$$

In some texts you will see the activation function given a different form, as:

$$a = g(h) = \tanh(h) = \frac{\exp(h) - \exp(-h)}{\exp(h) + \exp(-h)}, \tag{4.3}$$

which is the hyperbolic tangent function. This is a different but similar function; it is still a sigmoid function, but it saturates (reaches its constant values) at ± 1 instead of 0 and 1, which is sometimes useful. It also has a relatively simple derivative: $\frac{d}{dx} \tanh x = (1 - \tanh^2(x))$. We can convert between the two easily, because if the saturation points are (± 1), then we can convert to $(0, 1)$ by using $0.5 \times (x + 1)$.

So now we've got a new form of error computation and a new activation function that decides whether or not a neuron should fire. We can differentiate it, so that when we change the weights, we do it in the direction that is downhill for the error, which means that we know we are improving the error function of the network. As far as an algorithm goes, we've fed our inputs forward through the network and worked out which nodes are firing. Now, at the output, we've computed the errors as the sum-squared difference between the outputs and the targets (Equation (4.1) above). What we want to do next is to compute the gradient of these errors and use them to decide how much to update each weight in the network. We will do that first for the nodes connected to the output layer, and after we have updated those, we will work *backwards* through the network until we get back to the inputs again. There are just two problems:

- for the output neurons, we don't know the inputs.
- for the hidden neurons, we don't know the targets; for extra hidden layers, we know neither the inputs nor the targets, but even this won't matter for the algorithm we derive.

So we can compute the error at the output, but since we don't know what the inputs were that caused it, we can't update those second layer weights the way we did for the Perceptron. If we use the chain rule of differentiation that you all (possibly) remember from high school then we can get around this problem. Here, the chain rule tells us that if we want to know how the error changes as we vary the weights, we can think about how the error changes as we vary the inputs to the weights, and multiply this by how those input values change as we vary the weights. This is useful because it lets us calculate all of the derivatives that we want to: we can write the activations of the output nodes in terms of the activations of the hidden nodes and the output weights, and then we can send the error calculations back through the network to the hidden layer to decide what the target outputs were for those neurons. Note that we can do exactly the same computations if the network has extra hidden layers between the inputs and the outputs. It gets harder to keep track of which functions we should be differentiating, but there are no new tricks needed.

All of the relevant equations are derived in Section 4.6, and you should read that section carefully, since it is quite difficult to describe exactly what is going on here in words. The important thing to understand is that we compute the gradients of the errors with respect to the weights, so that we change the weights so that we go downhill, which makes the errors get smaller. We do this by differentiating the error function with respect to the weights, but we can't do this directly, so we have to apply the chain rule and differentiate with respect to things that we know. This leads to two different update functions, one for each of the

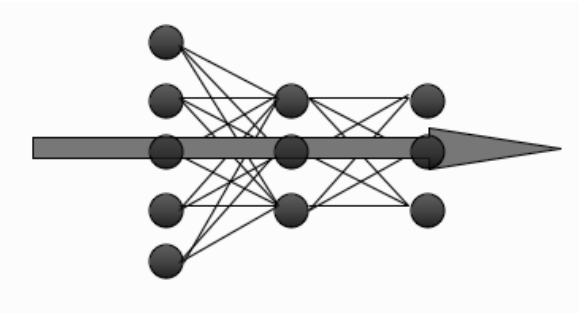

FIGURE 4.6 The forward direction in a Multi-layer Perceptron.

sets of weights, and we just apply these backwards through the network, starting at the outputs and ending up back at the inputs.

4.2.1 The Multi-layer Perceptron Algorithm

We'll get into the details of the basic algorithm here, and then, in the next section, have a look at some practical issues, such as how much training data is needed, how much training time is needed, and how to choose the correct size of network. We will assume that there are L input nodes, plus the bias, M hidden nodes, also plus a bias, and N output nodes, so that there are $(L+1) \times M$ weights between the input and the hidden layer and $(M+1) \times N$ between the hidden layer and the output. The sums that we write will start from 0 if they include the bias nodes and 1 otherwise, and run up to L, M, or N, so that $x_0 = -1$ is the bias input, and $a_0 = -1$ is the bias hidden node. The algorithm that is described could have any number of hidden layers, in which case there might be several values for M, and extra sets of weights between the hidden layers. We will also use i, j, k to index the nodes in each layer in the sums, and the corresponding Greek letters (ι, ζ, κ) for fixed indices.

Here is a quick summary of how the algorithm works, and then the full MLP training algorithm using back-propagation of error is described.

1. an input vector is put into the input nodes
2. the inputs are fed *forward* through the network (Figure 4.6)
    - the inputs and the first-layer weights (here labelled as v) are used to decide whether the hidden nodes fire or not. The activation function $g(\cdot)$ is the sigmoid function given in Equation (4.2) above
    - the outputs of these neurons and the second-layer weights (labelled as w) are used to decide if the output neurons fire or not
3. the *error* is computed as the sum-of-squares difference between the network outputs and the targets
4. this error is fed *backwards* through the network in order to
    - first update the second-layer weights
    - and then afterwards, the first-layer weights

The Multi-layer Perceptron Algorithm

- **Initialisation**
  - initialise all weights to small (positive and negative) random values
- **Training**
  - repeat:
    - for each input vector:

      Forwards phase:

      - compute the activation of each neuron j in the hidden layer(s) using:

$$h_\zeta = \sum_{i=0}^{L} x_i v_{i\zeta} \tag{4.4}$$

$$a_\zeta = g(h_\zeta) = \frac{1}{1+\exp(-\beta h_\zeta)} \tag{4.5}$$

      - work through the network until you get to the output layer neurons, which have activations (although see also Section 4.2.3):

$$h_\kappa = \sum_{j} a_j w_{j\kappa} \tag{4.6}$$

$$y_\kappa = g(h_\kappa) = \frac{1}{1+\exp(-\beta h_\kappa)} \tag{4.7}$$

      Backwards phase:

      - compute the error at the output using:

$$\delta_o(\kappa) = (y_\kappa - t_\kappa)\, y_\kappa (1 - y_\kappa) \tag{4.8}$$

      - compute the error in the hidden layer(s) using:

$$\delta_h(\zeta) = a_\zeta (1 - a_\zeta) \sum_{k=1}^{N} w_\zeta \delta_o(k) \tag{4.9}$$

      - update the output layer weights using:

$$w_{\zeta\kappa} \leftarrow w_{\zeta\kappa} - \eta \delta_o(\kappa) a_\zeta^{\text{hidden}} \tag{4.10}$$

      - update the hidden layer weights using:

$$v_\iota \leftarrow v_\iota - \eta \delta_h(\kappa) x_\iota \tag{4.11}$$

    - (if using sequential updating) randomise the order of the input vectors so that you don't train in exactly the same order each iteration
  - until learning stops (see Section 4.3.3)
- **Recall**
  - use the Forwards phase in the training section above

This provides a description of the basic algorithm. As with the Perceptron, a NumPy implementation can take advantage of various matrix multiplications, which makes things easy to read and faster to compute. The implementation on the website is a batch version of the algorithm, so that weight updates are made after all of the input vectors have been presented (as is described in Section 4.2.4). The central weight update computations for the algorithm can be implemented as:

```
deltao = (targets-self.outputs)*self.outputs*(1.0-self.outputs)
deltah = self.hidden*(1.0-self.hidden)*(np.dot(deltao,np.transpose(self.
weights2)))

updatew1 = np.zeros((np.shape(self.weights1)))
updatew2 = np.zeros((np.shape(self.weights2)))

updatew1 = eta*(np.dot(np.transpose(inputs),deltah[:,:-1]))
updatew2 = eta*(np.dot(np.transpose(self.hidden),deltao))
self.weights1 += updatew1
self.weights2 += updatew2
```

There are a few improvements that can be made to the algorithm, and there are some important things that need to be considered, such as how many training datapoints are needed, how many hidden nodes should be used, and how much training the network needs. We will look at the improvements first, and then move on to practical considerations in Section 4.3. There are lots of details that are given in this section because it is one of the early examples in the book; later on things will be skipped over more quickly.

The first thing that we can do is check that this MLP can indeed learn the logic functions, especially the XOR. We can do that with this code (which is function `logic` on the website):

```
import numpy as np
import mlp

anddata = np.array([[0,0,0],[0,1,0],[1,0,0],[1,1,1]])
xordata = np.array([[0,0,0],[0,1,1],[1,0,1],[1,1,0]])

p = mlp.mlp(anddata[:,0:2],anddata[:,2:3],2)
p.mlptrain(anddata[:,0:2],anddata[:,2:3],0.25,1001)
p.confmat(anddata[:,0:2],anddata[:,2:3])

q = mlp.mlp(xordata[:,0:2],xordata[:,2:3],2)
q.mlptrain(xordata[:,0:2],xordata[:,2:3],0.25,5001)
q.confmat(xordata[:,0:2],xordata[:,2:3])
```

The outputs that this produces is something like:

```
Iteration:  0  Error:  0.367917569871
Iteration:  1000  Error:  0.0204860723612
Confusion matrix is:
[[ 3.  0.]
 [ 0.  1.]]
Percentage Correct:  100.0
Iteration:  0  Error:  0.515798627074
Iteration:  1000  Error:  0.499568173798
```

```
Iteration:  2000  Error:  0.498271692284
Iteration:  3000  Error:  0.480839047738
Iteration:  4000  Error:  0.382706753191
Iteration:  5000  Error:  0.0537169253359
Confusion matrix is:
[[ 2.  0.]
 [ 0.  2.]]
Percentage Correct:  100.0
```

There are a few things to notice about this. One is that it does work, producing the correct answers, but the other is that even for the AND we need significantly more iterations than we did for the Perceptron. So the benefits of a more complex network come at a cost, because it takes substantially more computational time to fit those weights to solve the problem, even for linear examples. Sometimes, even 5000 iterations are not enough for the XOR function, and more have to be added.

4.2.2 Initialising the Weights

The MLP algorithm suggests that the weights are initialised to small random numbers, both positive and negative. The question is how small is small, and does it matter? One way to get a feeling for this would be to experiment with the code, setting all of the weights to 0, and seeing how well the network learns, then setting them all to large numbers and comparing the results. However, to understand why they should be small we can look at the shape of the sigmoid. If the initial weight values are close to 1 or -1 (which is what we mean by large here) then the inputs to the sigmoid are also likely to be close to ± 1 and so the output of the neuron is either 0 or 1 (the sigmoid has saturated, reached its maximum or minimum value). If the weights are very small (close to zero) then the input is still close to 0 and so the output of the neuron is just linear, so we get a linear model. Both of these things can be useful for the final network, but if we start off with values that are inbetween it can decide for itself.

Choosing the size of the initial values needs a little more thought, then. Each neuron is getting input from n different places (either input nodes if the neuron is in the hidden layer, or hidden neurons if it is in the output layer). If we view the values of these inputs as having uniform variance, then the typical input to the neuron will be $w\sqrt{n}$, where w is the initialisation value of the weights. So a common trick is to set the weights in the range $-1/\sqrt{n} < w < 1/\sqrt{n}$, where n is the number of nodes in the input layer to those weights. This makes the total input to a neuron have a maximum size of about 1. Further, if the weights are large, then the activation of a neuron is likely to be at, or close to, 0 or 1 already, which means that the gradients are small, and so the learning is very slow. There is an interplay here with the value of β in the logistic function, which means that small values of β (say $\beta = 3.0$ or less) are more effective. We use random values for the initialisation so that the learning starts off from different places for each run, and we keep them all about the same size because we want all of the weights to reach their final values at about the same time. This is known as uniform learning and it is important because otherwise the network will do better on some inputs than others.

4.2.3 Different Output Activation Functions

In the algorithm described above, we used sigmoid neurons in the hidden layer and the output layer. This is fine for classification problems, since there we can make the classes be 0 and 1. However, we might also want to perform regression problems, where the output needs to be from a continuous range, not just 0 or 1. The sigmoid neurons at the output are not very useful in that case. We can replace the output neurons with linear nodes that just sum the inputs and give that as their activation (so $g(h) = h$ in the notation of Equation (4.2)). This does not mean that we change the hidden layer neurons; they stay exactly the same, and we only modify the output nodes. They are not models of neurons anymore, since they don't have the characteristic fire/don't fire pattern. Even so, they enable us to solve regression problems, where we want a real number out, not just a 0/1 decision.

There is a third type of output neuron that is also used, which is the soft-max activation function. This is most commonly used for classification problems where the 1-of-N output encoding is used, as is described in Section 4.4.2. The soft-max function rescales the outputs by calculating the exponential of the inputs to that neuron, and dividing by the total sum of the inputs to all of the neurons, so that the activations sum to 1 and all lie between 0 and 1. As an activation function it can be written as:

$$y_\kappa = g(h_\kappa) = \frac{\exp(h_\kappa)}{\sum_{k=1}^{N} \exp(h_k)}. \tag{4.12}$$

Of course, if we change the activation function, then the derivative of the activation function will also change, and so the learning rule will be different. The changes that need to be made to the algorithm are in Equations (4.7) and (4.8), and are derived in Section 4.6.5. For the linear activation function the first is replaced by:

$$y_\kappa = g(h_\kappa) = h_\kappa, \tag{4.13}$$

while the second is replaced by:

$$\delta_o(\kappa) = (y_\kappa - t_\kappa)\,. \tag{4.14}$$

For the soft-max activation, the update equation that replaces (4.8) is

$$\delta_o(\kappa) = (y_\kappa - t_\kappa) y_\kappa (\delta_{\kappa K} - y_K), \tag{4.15}$$

where $\delta_{\kappa K} = 1$ if $\kappa = K$ and 0 otherwise; see Section 4.6.5 for further details. However, if we modify the error function as well, to have the cross-entropy form (where ln is the natural logarithm):

$$E_{\text{ce}} = -\sum_{k=1}^{N} t_k \ln(y_k), \tag{4.16}$$

then the delta term is Equation (4.14), just as for the linear output; for more details, see Section 4.6.6. Computing these update equations requires computing the error function that is being optimised, and then differentiating it. These additions can be added into the code by allowing the user to specify the type of output activation, which has to be done twice, once in the `mlpfwd` function, and once in the `mlptrain` function. In the former, the new piece of code can be written as:

```
# Different types of output neurons
if self.outtype == 'linear':
        return outputs
elif self.outtype == 'logistic':
    return 1.0/(1.0+np.exp(-self.beta*outputs))
elif self.outtype == 'softmax':
    normalisers = np.sum(np.exp(outputs),axis=1)*np.ones((1,np.shape(outputs)[
    0]))
    return np.transpose(np.transpose(np.exp(outputs))/normalisers)
else:
    print "error"
```

4.2.4 Sequential and Batch Training

The MLP is designed to be a batch algorithm. All of the training examples are presented to the neural network, the average sum-of-squares error is then computed, and this is used to update the weights. Thus there is only one set of weight updates for each epoch (pass through all the training examples). This means that we only update the weights once for each iteration of the algorithm, which means that the weights are moved in the direction that most of the inputs want them to move, rather than being pulled around by each input individually. The batch method performs a more accurate estimate of the error gradient, and will thus converge to the local minimum more quickly.

The algorithm that was described earlier was the sequential version, where the errors are computed and the weights updated after each input. This is not guaranteed to be as efficient in learning, but it is simpler to program when using loops, and it is therefore much more common. Since it does not converge as well, it can also sometimes avoid local minima, thus potentially reaching better solutions. While the description of the algorithm is sequential, the NumPy implementation on the book website is a batch version, because the matrix manipulation methods of NumPy make that easy. It is, however, relatively simple to modify it to use sequential update (making this change to the code is suggested as an exercise at the end of the chapter). In a sequential version, the order of the weight updates can matter, which is why the pseudocode version of the algorithm include a suggestion about randomising the order of the input vectors at each iteration. This can significantly improve the speed with which the algorithm learns. NumPy has a useful function that assists with this, `np.random.shuffle()`, which takes a list of numbers and reorders them. It can be used like this:

```
np.random.shuffle(change)
inputs = inputs[change,:]
targets = targets[change,:]
```

4.2.5 Local Minima

The driving force behind the learning rule is the minimisation of the network error by gradient descent (using the derivative of the error function to make the error smaller). This

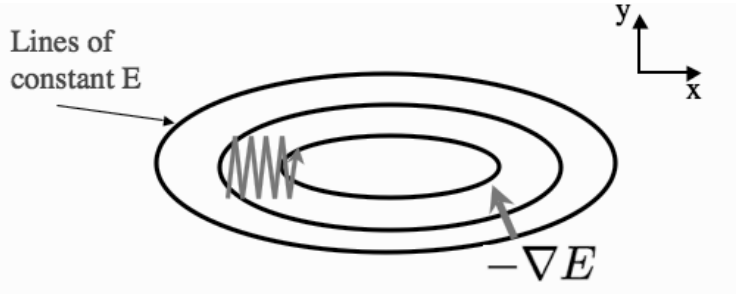

FIGURE 4.7 In 2D, downhill means at right angles to the lines of constant contour. Imagine walking down a hill with your eyes closed. If you find a direction that stays flat, then it is quite likely that perpendicular to that the ground goes uphill or downhill. However, this is not the direction that takes you directly towards the local minimum.

means that we are performing an optimisation: we are adapting the values of the weights in order to minimise the error function. As should be clear by now, the way that we are doing this is by approximating the gradient of the error and following it downhill so that we end up at the bottom of the slope. However, following the slope downhill only guarantees that we end up at a local minimum, a point that is lower than those close to it. If we imagine a ball rolling down a hill, it will settle at the bottom of a dip. However, there is no guarantee that it will have stopped at the lowest point—only the lowest point locally. There may be a much lower point over the next hill, but the ball can't see that, and it doesn't have enough energy to climb over the hill and find the global minimum (have another look at Figure 4.3 to see a picture of this).

Gradient descent works in the same way in two or more dimensions, and has similar (and worse) problems. The problem is that efficient downhill directions in two dimensions and higher are harder to compute locally. Standard contour maps provide beautiful images of gradients in our three-dimensional world, and if you imagine that you are walking in a hilly area aiming to get to the bottom of the nearest valley then you can get some idea of what is going on. Now suppose that you close your eyes, so that you can only feel which direction to go by moving one step and checking if you are higher up or lower down than you were. There will be places where going downwards as steeply as possible at the current point will not take you much closer to the valley bottom. There can be two reasons for this. The first is that you find a nearby local minimum, while the second is that sometimes the steepest direction is effectively across the valley, not towards the global minimum. This is shown in Figure 4.7.

All of these things are true for most of our optimisation problems, including the MLP. We don't know where the global minimum is because we don't know what the error landscape looks like; we can only compute local features of it for the place we are in at the moment. Which minimum we end up in depends on where we start. If we begin near the global minimum, then we are very likely to end up in it, but if we start near a local minimum we will probably end up there. In addition, how long it will take to get to the minimum that we do find depends upon the exact appearance of the landscape at the current point.

We can make it more likely that we find the global minimum by trying out several different starting points by training several different networks, and this is commonly done. However, we can also try to make it less likely that the algorithm will get stuck in local minima. There is a moderately effective way of doing this, which is discussed next.

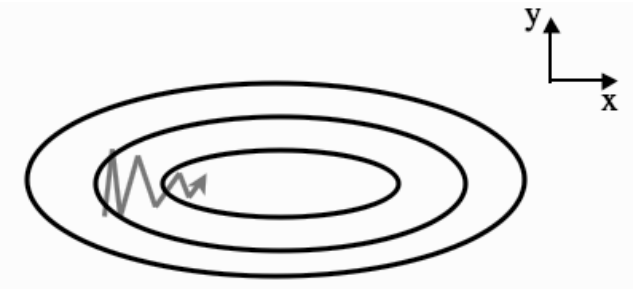

FIGURE 4.8 Adding momentum can help to avoid local minima, and also makes the dynamics of the optimisation more stable, improving convergence.

4.2.6 Picking Up Momentum

Let's go back to the analogy of the ball rolling down the hill. The reason that the ball stops rolling is because it runs out of energy at the bottom of the dip. If we give the ball some weight, then it will generate momentum as it rolls, and so it is more likely to overcome a small hill on the other side of the local minimum, and so more likely to find the global minimum. We can implement this idea in our neural network learning by adding in some contribution from the previous weight change that we made to the current one. In two dimensions it will mean that the ball rolls more directly towards the valley bottom, since on average that will be the correct direction, rather than being controlled by the local changes. This is shown in Figure 4.8.

There is another benefit to momentum. It makes it possible to use a smaller learning rate, which means that the learning is more stable. The only change that we need to make to the MLP algorithm is in Equations (4.10) and (4.11), where we need to add a second term to the weight updates so that they have the form:

$$w_{\zeta\kappa}^{t} \leftarrow w_{\zeta\kappa}^{t-1} + \eta\delta_o(\kappa)a_{\zeta}^{\text{hidden}} + \alpha\Delta w_{\zeta\kappa}^{t-1}, \tag{4.17}$$

where t is used to indicate the current update and $t-1$ is the previous one. $\Delta w_{\zeta\kappa}^{t-1}$ is the previous update that we made to the weights (so $\Delta w_{\zeta\kappa}^{t} = \eta\delta_o(\kappa)a_{\zeta}^{\text{hidden}} + \alpha\Delta w_{\zeta\kappa}^{t-1}$) and $0 < \alpha < 1$ is the momentum constant. Typically a value of $\alpha = 0.9$ is used. This is a very easy addition to the code, and can improve the speed of learning a lot.

```
updatew1 = eta*(np.dot(np.transpose(inputs),deltah[:,:-1])) + ↩
momentum*updatew1
updatew2 = eta*(np.dot(np.transpose(hidden),deltao)) + momentum*updatew2
```

Another thing that can be added is known as weight decay. This reduces the size of the weights as the number of iterations increases. The argument goes that small weights are better since they lead to a network that is closer to linear (since they are close to zero, they are in the region where the sigmoid is increasing linearly), and only those weights that are essential to the non-linear learning should be large. After each learning iteration through all of the input patterns, every weight is multiplied by some constant $0 < \epsilon < 1$. This makes the network simpler and can often produce improved results, but unfortunately, it

isn't fail-safe: occasionally it can make the learning significantly worse, so it should be used with care. Setting the value of ϵ is typically done experimentally.

4.2.7 Minibatches and Stochastic Gradient Descent

In Section 4.2.4 it was stated that the batch algorithm converges to a local minimum faster than the sequential algorithm, which computes the error for each input individually and then does a weight update, but that the latter is sometimes less likely to get stuck in local minima. The reason for both of these observations is that the batch algorithm makes a better estimate of the steepest descent direction, so that the direction it chooses to go is a good one, but this just leads to a local minimum.

The idea of a minibatch method is to find some happy middle ground between the two, by splitting the training set into random batches, estimating the gradient based on one of the subsets of the training set, performing a weight update, and then using the next subset to estimate a new gradient and using that for the weight update, until all of the training set have been used. The training set are then randomly shuffled into new batches and the next iteration takes place. If the batches are small, then there is often a reasonable degree of error in the gradient estimate, and so the optimisation has the chance to escape from local minima, albeit at the cost of heading in the wrong direction.

A more extreme version of the minibatch idea is to use just one piece of data to estimate the gradient at each iteration of the algorithm, and to pick that piece of data uniformly at random from the training set. So a single input vector is chosen from the training set, and the output and hence the error for that one vector computed, and this is used to estimate the gradient and so update the weights. A new random input vector (which could be the same as the previous one) is then chosen and the process repeated. This is known as stochastic gradient descent, and can be used for any gradient descent problem, not just the MLP. It is often used if the training set is very large, since it would be very expensive to use the whole dataset to estimate the gradient in that case.

4.2.8 Other Improvements

There are a few other things that can be done to improve the convergence and behaviour of the back-propagation algorithm. One is to reduce the learning rate as the algorithm progresses. The reasoning behind this is that the network should only be making large-scale changes to the weights at the beginning, when the weights are random; if it is still making large weight changes later on, then something is wrong.

Something that results in much larger performance gains is to include information about the second derivatives of the error with respect to the weights. In the back-propagation algorithm we use the first derivatives to drive the learning. However, if we have knowledge of the second derivatives as well, we can use them as well to improve the network. This will be described in more detail in Section 9.1.

4.3 THE MULTI-LAYER PERCEPTRON IN PRACTICE

The previous section looked at the design and implementation of the MLP network itself. In this section, we are going to look more at choices that can be made about the network in order to use it for solving real problems. We will then apply these ideas to using the MLP to find solutions to four different types of problem: regression, classification, time-series prediction, and data compression.

4.3.1 Amount of Training Data

For the MLP with one hidden layer there are $(L + 1) \times M + (M + 1) \times N$ weights, where L, M, N are the number of nodes in the input, hidden, and output layers, respectively. The extra +1s come from the bias nodes, which also have adjustable weights. This is a potentially huge number of adjustable parameters that we need to set during the training phase. Setting the values of these weights is the job of the back-propagation algorithm, which is driven by the errors coming from the training data. Clearly, the more training data there is, the better for learning, although the time that the algorithm takes to learn increases. Unfortunately, there is no way to compute what the minimum amount of data required is, since it depends on the problem. A rule of thumb that has been around for almost as long as the MLP itself is that you should use a number of training examples that is at least 10 times the number of weights. This is probably going to be a very large number of examples, so neural network training is a fairly computationally expensive operation, because we need to show the network all of these inputs lots of times.

4.3.2 Number of Hidden Layers

There are two other considerations concerning the number of weights that are inherent in the calculation above, which is the choice of the number of hidden nodes, and the number of hidden layers. Making these choices is obviously fundamental to the successful application of the algorithm. We will shortly see a pictorial demonstration of the fact that two hidden layers is the most that you ever need for normal MLP learning. In fact, this result can be strengthened: it is possible to show mathematically that one hidden layer with lots of hidden nodes is sufficient. This is known as the Universal Approximation Theorem; see the Further Reading section for more details. However, the bad news is that there is no theory to guide the choice of the number of hidden nodes. You just have to experiment by training networks with different numbers of hidden nodes and then choosing the one that gives the best results, as we will see in Section 4.4.

We can use the back-propagation algorithm for a network with as many layers as we like, although it gets progressively harder to keep track of which weights are being updated at any given time. Fortunately, as was mentioned above, we will never normally need more than two layers (that is, one hidden layer and the output layer). This is because we can approximate any smooth functional mapping using a linear combination of localised sigmoidal functions. There is a sketchy demonstration that two hidden layers are sufficient using pictures in Figure 4.9. The basic idea is that by combining sigmoid functions we can generate ridge-like functions, and by combining ridge-like functions we can generate functions with a unique maximum. By combining these and transforming them using another layer of neurons, we obtain a localised response (a 'bump' function), and any functional mapping can be approximated to arbitrary accuracy using a linear combination of such bumps. The way that the MLP does this is shown in Figure 4.10. We will use this idea again when we look at approximating functions, for example using radial basis functions in Chapter 5. Note that Figure 4.9 shows that two hidden layers are sufficient. In fact, they aren't necessary: one hidden layer will do, although it may require an arbitrarily large number of hidden nodes. This is known as the Universal Approximation Theorem, and the (mathematical) paper that shows this is provided in the references at the end of the chapter.

Two hidden layers are sufficient to compute these bump functions for different inputs, and so if the function that we want to learn (approximate) is continuous, the network can compute it. It can therefore approximate any decision boundary, not just the linear one that the Perceptron computed.

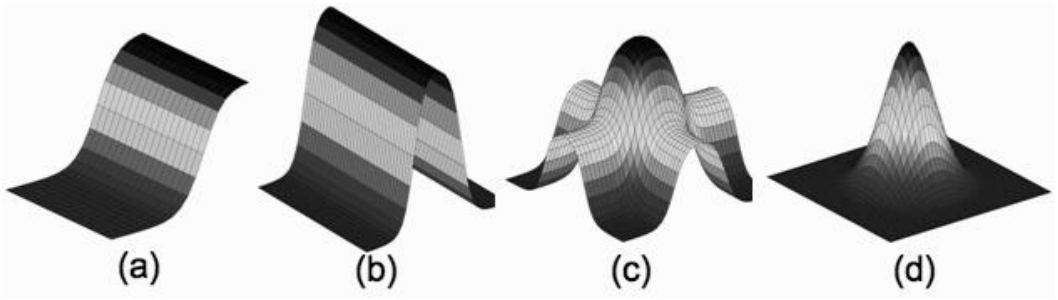

FIGURE 4.9 The learning of the MLP can be shown as the output of a single sigmoidal neuron (a), which can be added to others, including reversed ones, to get a hill shape (b). Adding another hill at 90° produces a bump (c) , which can be sharpened to any extent we want (d), with the bumps added together in the output layer. Thus the MLP learns a local representation of individual inputs.

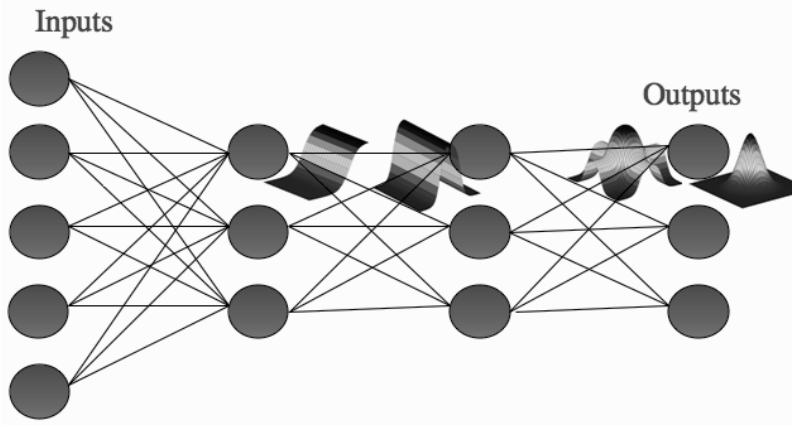

FIGURE 4.10 Schematic of the effective learning shape at each stage of the MLP.

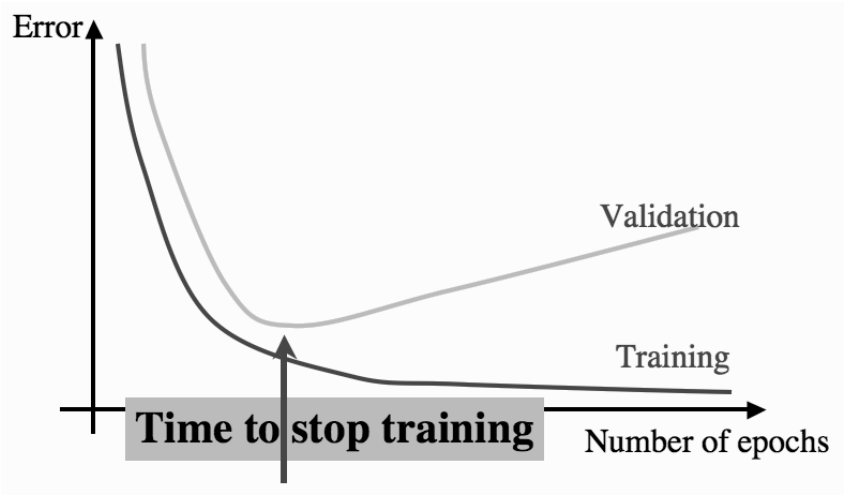

FIGURE 4.11 The effect of overfitting on the training and validation error curves, with the point at which early stopping will stop the learning marked.

4.3.3 When to Stop Learning

The training of the MLP requires that the algorithm runs over the entire dataset many times, with the weights changing as the network makes errors in each iteration. The question is how to decide when to stop learning, and this is a question that we are now ready to answer. It is unfortunate that the most obvious options are not sufficient: setting some predefined number N of iterations, and running until that is reached runs the risk that the network has overfitted by then, or not learnt sufficiently, and only stopping when some predefined minimum error is reached might mean the algorithm never terminates, or that it overfits. Using both of these options together can help, as can terminating the learning once the error stops decreasing.

However, the validation set gives us something rather more useful, since we can use it to monitor the generalisation ability of the network at its current stage of learning. If we plot the sum-of-squares error during training, it typically reduces fairly quickly during the first few training iterations, and then the reduction slows down as the learning algorithm performs small changes to find the exact local minimum. We don't want to stop training until the local minimum has been found, but, as we've just discussed, keeping on training too long leads to overfitting of the network. This is where the validation set comes in useful. We train the network for some predetermined amount of time, and then use the validation set to estimate how well the network is generalising. We then carry on training for a few more iterations, and repeat the whole process. At some stage the error on the validation set will start increasing again, because the network has stopped learning about the function that generated the data, and started to learn about the noise that is in the data itself (shown in Figure 4.11). At this stage we stop the training. This technique is called early stopping.

4.4 EXAMPLES OF USING THE MLP

This section is intended to be practical, so you should follow the examples at a computer, and add to them as you wish. The MLP is rather too complicated to enable us to work through the weight changes as we did with the Perceptron.

Instead, we shall look at some demonstrations of how to make the network learn about some data. As was mentioned above, we shall look at the four types of problems that are generally solved using an MLP: regression, classification, time-series prediction, and data compression/data denoising.

4.4.1 A Regression Problem

The regression problem we will look at is a very simple one. We will take a set of samples generated by a simple mathematical function, and try to learn the generating function (that describes how the data was made) so that we can find the values of any inputs, not just the ones we have training data for.

The function that we will use is a very simple one, just a bit of a sine wave. We'll make the data in the following way (make sure that you have NumPy imported as `np` first):

```
x = np.ones((1,40))*np.linspace(0,1,40)
t = np.sin(2*np.pi*x) + np.cos(4*np.pi*x) + np.random.randn(40)*0.2
x = x.T
t = t.T
```

The reason why we have to use the `reshape()` method is that NumPy defaults to lists for arrays that are $N \times 1$; compare the results of the `np.shape()` calls below, and the effect of the transpose operator `.T` on the array:

```
>>> x = np.linspace(0,1,40)
>>> np.shape(x)
(40,)
>>> np.shape(x.T)
(40,)
>>>
>>> x = np.linspace(0,1,40).reshape((1,40))
>>> np.shape(x)
(1, 40)
>>> np.shape(x.T)
(40, 1)
```

You can plot this data to see what it looks like (the results of which are shown in Figure 4.12) using:

```
>>> import pylab as pl
>>> pl.plot(x,t,'.')
```

We can now train an MLP on the data. There is one input value, `x` and one output value `t`, so the neural network will have one input and one output. Also, because we want the output to be the value of the function, rather than 0 or 1, we will use linear neurons at the output. We don't know how many hidden neurons we will need yet, so we'll have to experiment to see what works.

Before getting started, we need to normalise the data using the method shown in Section 3.4.5, and then separate the data into training, testing, and validation sets. For this example there are only 40 datapoints, and we'll use half of them as the training set, although that isn't very many and might not be enough for the algorithm to learn effectively. We can split the data in the ratio 50:25:25 by using the odd-numbered elements as training data, the even-numbered ones that do not divide by 4 for testing, and the rest for validation:

```
train = x[0::2,:]
test = x[1::4,:]
valid = x[3::4,:]
traintarget = t[0::2,:]
testtarget = t[1::4,:]
validtarget = t[3::4,:]
```

With that done, it is just a case of making and training the MLP. To start with, we will construct a network with three nodes in the hidden layer, and run it for 101 iterations with a learning rate of 0.25, just to see that it works:

```
>>> import mlp
>>> net = mlp.mlp(train,traintarget,3,outtype='linear')
>>> net.mlptrain(train,traintarget,0.25,101)
```

The output from this will look something like:

```
Iteration:  0  Error:  12.3704163654
Iteration:  100  Error:  8.2075961385
```

so we can see that the network is learning, since the error is decreasing. We now need to do two things: work out how many hidden nodes we need, and decide how long to train the network for. In order to solve the first problem, we need to test out different networks and see which get lower errors, but to do that properly we need to know when to stop training. So we'll solve the second problem first, which is to implement early stopping.

We train the network for a few iterations (let's make it 10 for now), then evaluate the validation set error by running the network forward (i.e., the recall phase). Learning should stop when the validation set error starts to increase. We'll write a Python program that does all the work for us. The important point is that we keep track of the validation error and stop when it starts to increase. The following code is a function within the MLP on the book website. It keeps track of the last two changes in validation error to ensure that small fluctuations in the learning don't change it from early stopping to premature stopping:

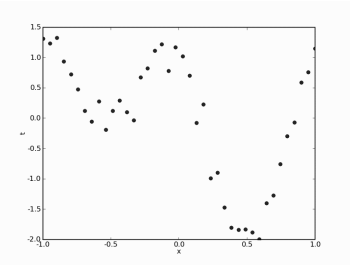

FIGURE 4.12 The data that we will learn using an MLP, consisting of some samples from a sine wave with Gaussian noise added.

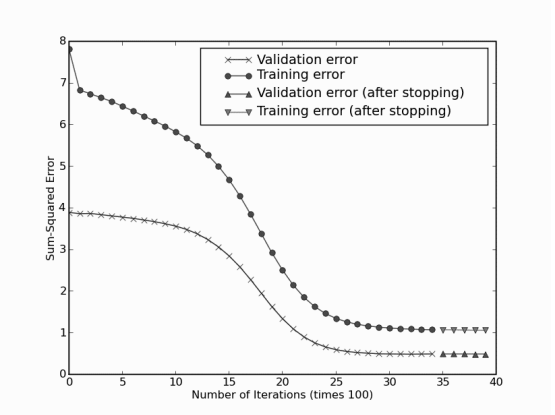

FIGURE 4.13 Plot of the error as the MLP learns (top line is total error on the training set; bottom line is on the validation set; it is larger on the training set because there are more datapoints in this set). Early-stopping halts the learning at the point where there is no line, where the crosses become triangles. The learning was continued to show that the error got slightly worse afterwards.

```
old_val_error1 = 100002
old_val_error2 = 100001
new_val_error = 100000

count = 0
while (((old_val_error1 - new_val_error) > 0.001) or ((old_val_error2 - ↩
old_val_error1)>0.001)):
    count+=1
    self.mlptrain(inputs,targets,0.25,100)
    old_val_error2 = old_val_error1
    old_val_error1 = new_val_error
    validout = self.mlpfwd(valid)
    new_val_error = 0.5*np.sum((validtargets-validout)**2)

print "Stopped", new_val_error,old_val_error1, old_val_error2
```

Figure 4.13 gives an example of the output of running the function. It plots the training and validation errors. The point at which early stopping makes the learning finish is the point where there is a missing validation datapoint. I ran it on after that so you could see that the validation error did not improve after that, and so early stopping found the correct point.

We can now return to the problem of finding the right size of network. There is one important thing to remember, which is that the weights are initialised randomly, and so

the fact that a particular size of network gets a good solution once does not mean it is the right size, it could have been a lucky starting point. So each network size is run 10 times, and the average is monitored. The following table shows the results of doing this, reporting the sum-of-squares validation error, for a few different sizes of network:

No. of hidden nodes	1	2	3	5	10	25	50
Mean error	2.21	0.52	0.52	0.52	0.55	1.35	2.56
Standard deviation	0.17	0.00	0.00	0.02	0.00	1.20	1.27
Max error	2.31	0.53	0.54	0.54	0.60	3.230	3.66
Min error	2.10	0.51	0.50	0.50	0.47	0.42	0.52

Based on these numbers, we would select a network with a small number of hidden nodes, certainly between 2 and 10 (and the smaller the better, in general), since their maximum error is much smaller than a network with just 1 hidden node. Note also that the error increases once too many hidden nodes are used, since the network has too much variation for the problem. You can also do the same kind of experimentation with more hidden layers.

4.4.2 Classification with the MLP

Using the MLP for classification problems is not radically different once the output encoding has been worked out. The inputs are easy: they are just the values of the feature measurements (suitably normalised). There are a couple of choices for the outputs. The first is to use a single linear node for the output, y, and put some thresholds on the activation value of that node. For example, for a four-class problem, we could use:

$$\text{Class is: } \begin{cases} C_1 & \text{if } y \leq -0.5 \\ C_2 & \text{if } -0.5 < y \leq 0 \\ C_3 & \text{if } 0 < y \leq 0.5 \\ C_4 & \text{if } y > 0.5 \end{cases} \tag{4.18}$$

However, this gets impractical as the number of classes gets large, and the boundaries are artificial; what about an example that is very close to a boundary, say $y = 0.5$? We arbitrarily guess that it belongs to class C_3, but the neural network doesn't give us any information about how close it was to the boundary in the output, so we don't know that this was a difficult example to classify. A more suitable output encoding is called 1-of-N encoding. A separate node is used to represent each possible class, and the target vectors consist of zeros everywhere except for in the one element that corresponds to the correct class, e.g., $(0, 0, 0, 1, 0, 0)$ means that the correct result is the 4th class out of 6. We are therefore using binary output values (we want each output to be either 0 or 1).

Once the network has been trained, performing the classification is easy: simply choose the element y_k of the output vector that is the largest element of $\mathbf{y}$ (in mathematical notation, pick the y_k for which $y_k > y_j \forall j \neq k$; $\forall$ means for all, so this statement says pick the y_k that is bigger than all other possible values y_j). This generates an unambiguous decision, since it is very unlikely that two output neurons will have identical largest output values. This is known as the hard-max activation function (since the neuron with the highest activation is chosen to fire and the rest are ignored). An alternative is the soft-max function, which we saw in Section 4.2.3, and which has the effect of scaling the output of each neuron according to how large it is in comparison to the others, and making the total output sum to 1. So if there is one clear winner, it will have a value near 1, while if there are several

values that are close to each other, they will each have a value of about $\frac{1}{p}$, where p is the number of output neurons that have similar values.

There is one other thing that we need to be aware of when performing classification, which is true for all classifiers. Suppose that we are doing two-class classification, and 90% of our data belongs to class 1. (This can happen: for example in medical data, most tests are negative in general.) In that case, the algorithm can learn to always return the negative class, since it will be right 90% of the time, but still a completely useless classifier! So you should generally make sure that you have approximately the same number of each class in your training set. This can mean discarding a lot of data from the over-represented class, which may seem rather wasteful. There is an alternative solution, known as novelty detection, which is to train the data on the data in the negative class only, and to assume that anything that looks different to that is a positive example. There is a reference about novelty detection in the readings at the end of the chapter.

4.4.3 A Classification Example: The Iris Dataset

As an example we are going to look at another example from the UCI Machine Learning repository. This one is concerned with classifying examples of three types of iris (flower) by the length and width of the sepals and petals and is called `iris`. It was originally worked on by R.A. Fisher, a famous statistician and biologist, who analysed it in the 1930s.

Unfortunately we can't currently load this into NumPy using `loadtxt()` because the class (which is the last column) is text rather than a number, and the `txt` in the function name doesn't mean that it reads text, only numbers in plaintext format. There are two alternatives. One is to edit the data in a text editor using search and replace, and the other is to use some Python code, such as this function:

```
def preprocessIris(infile,outfile):

    stext1 = 'Iris-setosa'
    stext2 = 'Iris-versicolor'
    stext3 = 'Iris-virginica'
    rtext1 = '0'
    rtext2 = '1'
    rtext3 = '2'

    fid = open(infile,"r")
    oid = open(outfile,"w")

    for s in fid:
        if s.find(stext1)>-1:
            oid.write(s.replace(stext1, rtext1))
        elif s.find(stext2)>-1:
            oid.write(s.replace(stext2, rtext2))
        elif s.find(stext3)>-1:
            oid.write(s.replace(stext3, rtext3))
    fid.close()
    oid.close()
```

You can then load it from the new file using `loadtxt()`. In the dataset, the last column is the class ID, and the others are the four measurements. We'll start by normalising the inputs, which we'll do in the same way as in Section 3.4.5, but using the maximum rather than the variance, and leaving the class IDs alone for now:

```
iris = np.loadtxt('iris_proc.data',delimiter=',')
iris[:,:4] = iris[:,:4]-iris[:,:4].mean(axis=0)
imax = np.concatenate((iris.max(axis=0)*np.ones((1,5)),np.abs(iris.min(0
axis=0))*np.ones((1,5))),axis=0).max(axis=0)
iris[:,:4] = iris[:,:4]/imax[:4]
```

The first few datapoints will then look like:

```
>>> print iris[0:5,:]
[[-0.36142626  0.33135215 -0.7508489  -0.76741803  0. ]
 [-0.45867099 -0.04011887 -0.7508489  -0.76741803  0. ]
 [-0.55591572  0.10846954 -0.78268251 -0.76741803  0. ]
 [-0.60453809  0.03417533 -0.71901528 -0.76741803  0. ]
 [-0.41004862  0.40564636 -0.7508489  -0.76741803  0. ]]
```

We now need to convert the targets into 1-of-N encoding, from their current encoding as class 1, 2, or 3. This is pretty easy if we make a new matrix that is initially all zeroes, and simply set one of the entries to be 1:

```
# Split into training, validation, and test sets
target = np.zeros((np.shape(iris)[0],3));
indices = np.where(iris[:,4]==0)
target[indices,0] = 1
indices = np.where(iris[:,4]==1)
target[indices,1] = 1
indices = np.where(iris[:,4]==2)
target[indices,2] = 1
```

We now need to separate the data into training, testing, and validation sets. There are 150 examples in the dataset, and they are split evenly amongst the three classes, so the three classes are the same size and we don't need to worry about discarding any datapoints. We'll split them into half training, and one quarter each testing and validation. If you look at the file, you will notice that the first 50 are class 1, the second 50 class 2, etc. We therefore need to randomise the order before we split them into sets, to ensure that there are not too many of one class in one of the sets:

```
# Randomly order the data
order = range(np.shape(iris)[0])
```

```
np.random.shuffle(order)
iris = iris[order,:]
target = target[order,:]

train = iris[::2,0:4]
traint = target[::2]
valid = iris[1::4,0:4]
validt = target[1::4]
test = iris[3::4,0:4]
testt = target[3::4]
```

We're now finally ready to set up and train the network. The commands should all be familiar from earlier:

```
>>> import mlp
>>> net = mlp.mlp(train,traint,5,outtype='softmax')
>>> net.earlystopping(train,traint,valid,validt,0.1)
>>> net.confmat(test,testt)
Confusion matrix is:
[[ 16.   0.   0.]
 [  0.  12.   2.]
 [  0.   1.   6.]]
Percentage Correct:  91.8918918919
```

This tells us that the algorithm got nearly all of the test data correct, misclassifying just two examples of class 2 and one of class 3.

4.4.4 Time-Series Prediction

There is a common data analysis task known as time-series prediction, where we have a set of data that show how something varies over time, and we want to predict how the data will vary in the future. It is quite a difficult task, but a fairly important one. It is useful in any field where there is data that appears over time, which is to say almost any field. Most notable (if often unsuccessful) uses have been in trying to predict stock markets and disease patterns. The problem is that even if there is some regularity in the time-series, it can appear over many different scales. For example, there is often seasonal variation—if we plotted average temperature over several years, we would notice that it got hotter in the summer and colder in the winter, but we might not notice if there was a overall upward or downward trend to the summer temperatures, because the summer peaks are spread too far apart in the data.

The other problems with the data are practical. How many datapoints should we look at to make the prediction (i.e., how many inputs should there be to the neural network) and how far apart in time should we space those inputs (i.e., should we use every second datapoint, every 10th, or all of them)? We can write this as an equation, where we are predicting y using a neural network that is written as a function $f(\cdot)$:

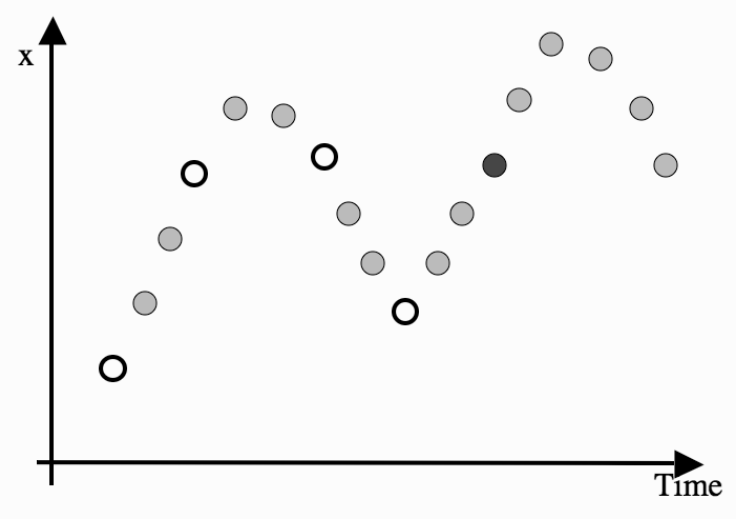

FIGURE 4.14 Part of a time-series plot, showing the datapoints and the meanings of τ and k.

$$y = x(t+\tau) = f(x(t), x(t-\tau), \ldots, x(t-k\tau)), \tag{4.19}$$

where the two questions about how many datapoints and how far apart they should be come down to choices about τ and k.

The target data for training the neural network is simple, because it comes from further up the time-series, and so training is easy. Suppose that $\tau = 2$ and $k = 3$. Then the first input data are elements $1, 3, 5$ of the dataset, and the target is element 7. The next input vector is elements $2, 4, 6$, with target 8, and then $3, 5, 7$ with target 9. You train the network by passing through the time-series (remembering to save some data for testing), and then press on into the future making predictions. Figure 4.14 shows an example of a time-series with $\tau = 3$ and $k = 4$, with a set of datapoints that make up an input vector marked as white circles, and the target coloured black.

The dataset I am going to use is available on the book website. It provides the daily measurement of the thickness of the ozone layer above Palmerston North in New Zealand (where I live) between 1996 and 2004. Ozone thickness is measured in Dobson Units, which are 0.01 mm thickness at 0 degrees Celsius and 1 atmosphere of pressure. I'm sure that I don't need to tell you that the reduction in stratospheric ozone is partly responsible for global warming and the increased incidence of skin cancer, and that in New Zealand we are fairly close to the large hole over Antarctica. What you might not know is that the thickness of the ozone layer varies naturally over the year. This should be obvious in the plot shown in Figure 4.15. A typical time-series problem is to predict the ozone levels into the future and see if you can detect an overall drop in the mean ozone level.

You can load the data using `PNoz = loadtxt('PNOz.dat')` (once you've downloaded it from the website), which will load the data and stick it into an array called PNoz. There are 4 elements to each vector: the year, the day of the year, and the ozone level and sulphur dioxide level, and there are 2855 readings. To just plot the ozone data so that you can see what it looks like, use `plot(arange(shape(PNoz)[0]),PNoz[:,2],'.')`.

The difficult bit is assembling the input vector from the time-series data. The first thing

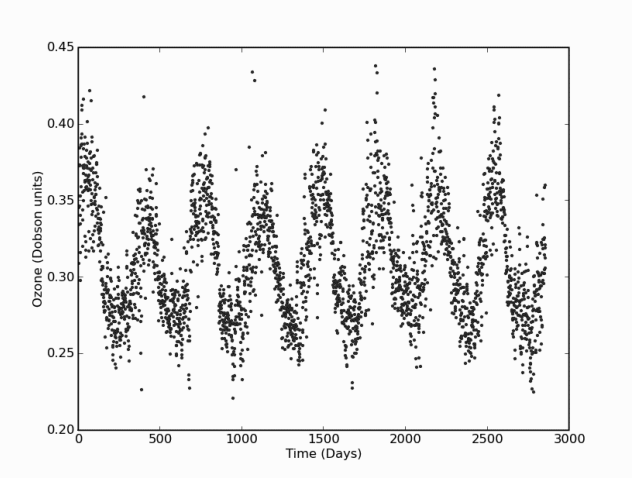

FIGURE 4.15 Plot of the ozone layer thickness above Palmerston North in New Zealand between 1996 and 2004.

is to choose values of τ and k. Then it is just a question of picking k values out of the array with spacing τ, which is a good use for the slice operator, as in this code:

```
test = inputs[-800:,:]
testtargets = targets[-800,:]
train = inputs[:-800:2,:]
traintargets = targets[:-800:2]
valid = inputs[1:-800:2,:]
validtargets = targets[1:-800:2]
```

You then need to assemble training, testing, and validation sets. However, some care is needed here since you need to ensure that they are not picked systematically into each group, (for example, if the inputs are the even-indexed datapoints, but some feature is only seen at odd datapoint times, then it will be completely missed). This can be averted by randomising the order of the datapoints first. However, it is also common to use the datapoints near the end as part of the test set; some possible results from using the MLP in this way are shown in Figure 4.16.

From here you can treat time-series as regression problems: the output nodes need to have linear activations, and you aim to minimise the sum-of-squares error. Since there are no classes, the confusion matrix is not useful. The only extra work is that in addition to testing MLPs with different numbers of input nodes and hidden nodes, you also need to consider different values of τ and k.

4.4.5 Data Compression: The Auto-Associative Network

We are now going to consider an interesting variation of the MLP. Suppose that we train the network to reproduce the inputs at the output layer (called auto-associative learning; sometimes the network is known as an autoencoder). The network is trained so that whatever

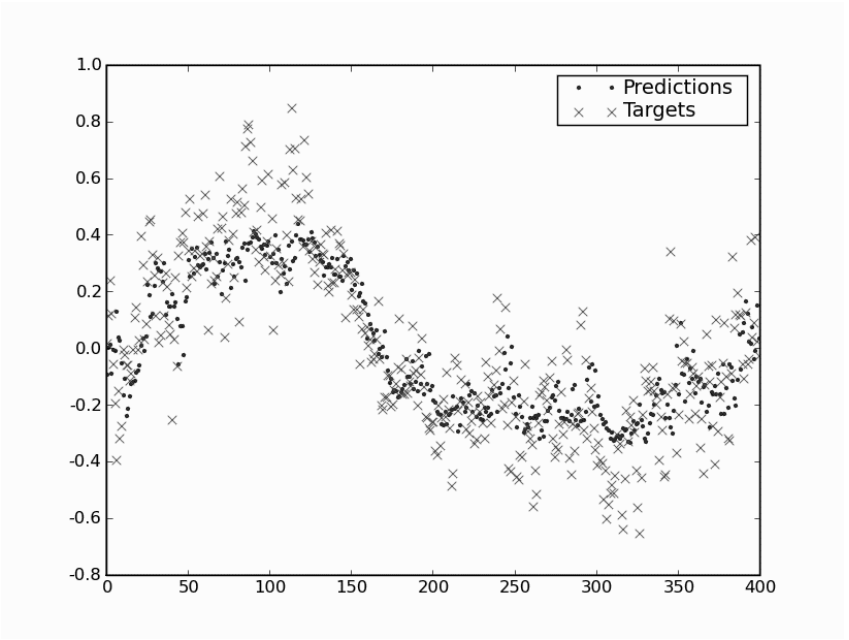

FIGURE 4.16 Plot of 400 predicted and actual output values of the ozone data using the MLP as a time-series predictor with $k = 3$ and $\tau = 2$.

you show it at the input is reproduced at the output, which doesn't seem very useful at first, but suppose that we use a hidden layer that has fewer neurons than the input layer (see Figure 4.17). This bottleneck hidden layer has to represent all of the information in the input, so that it can be reproduced at the output. It therefore performs some compression of the data, representing it using fewer dimensions than were used in the input. This gives us some idea of what the hidden layers of the MLP are doing: they are finding a different (often lower dimensional) representation of the input data that extracts important components of the data, and ignores the noise.

This auto-associative network can be used to compress images and other data. A schematic of this is shown in Figure 4.18: the 2D image is turned into a 1D vector of inputs by cutting the image into strips and sticking the strips into a long line. The values of this vector are the intensity (colour) values of the image, and these are the input values. The network learns to reproduce the same image at the output, and the activations of the hidden nodes are recorded for each image. After training, we can throw away the input nodes and first set of weights of the network. If we insert some values in the hidden nodes (their activations for a particular image; see Figure 4.19), then by feeding these activations forward through the second set of weights, the correct image will be reproduced on the output. So all we need to store are the set of second-layer weights and the activations of the hidden nodes for each image, which is the compressed version.

Auto-associative networks can also be used to denoise images, since, after training, the network will reproduce the trained image that best matches the current (noisy) input. We don't throw away the first set of weights this time, but if we feed a noisy version of the image into the inputs, then the network will produce the image that is closest to the noisy version at the outputs, which will be the version it learnt on, which is uncorrupted by noise.

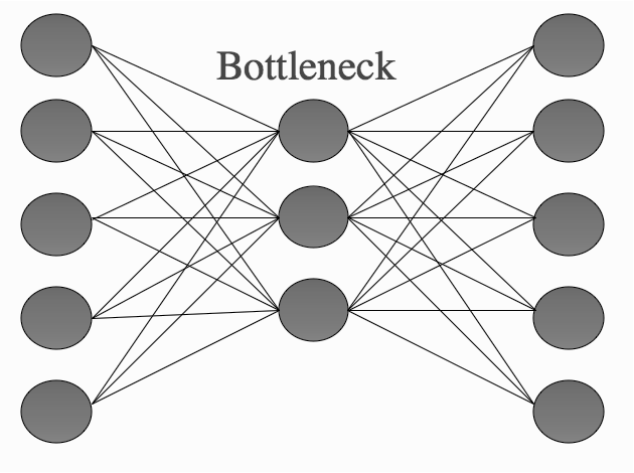

FIGURE 4.17 The auto-associative network. The network is trained to reproduce the inputs at the outputs, passing them through the bottleneck hidden layer that compresses the data.

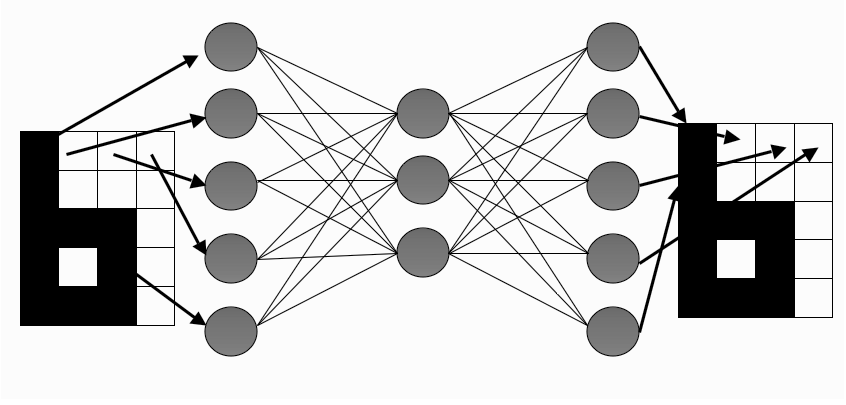

FIGURE 4.18 Schematic showing how images are fed into the auto-associative network for compression.

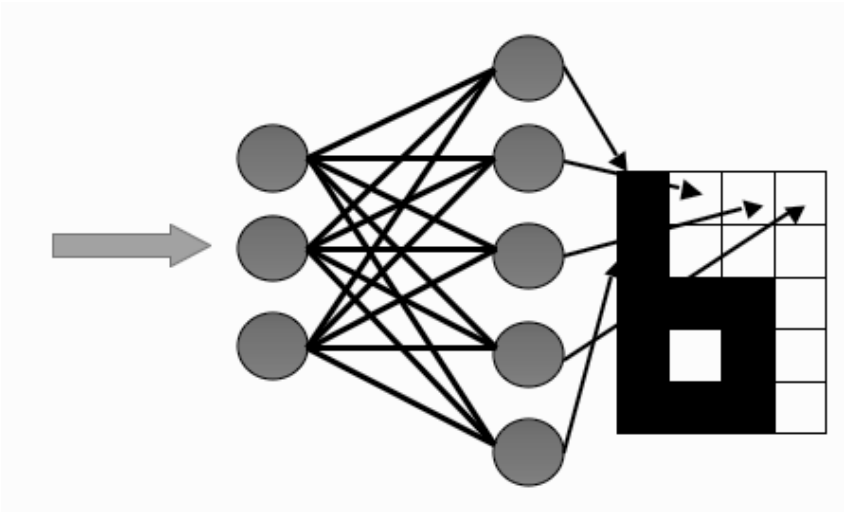

FIGURE 4.19 Schematic showing how the hidden nodes and second layer of weights can be used to regain the compressed images after the network has been trained.

You might be wondering what this representation in the hidden nodes looks like. In fact, if the nodes all have linear activation, then what the network learns to compute are the Principal Components of the input data. Principal Components Analysis (PCA) is a useful dimensionality reduction technique, and is described in Section 6.2.

4.5 A RECIPE FOR USING THE MLP

We have covered a lot in this chapter, so I'm going to give you a 'recipe' for how to use the Multi-layer Perceptron when presented with a dataset. This is, by necessity, a simplification of the problem, but it should serve to remind you of many of the important features.

Select inputs and outputs for your problem Before anything else, you need to think about the problem you are trying to solve, and make sure that you have data for the problem, both input vectors and target outputs. At this stage you need to choose what features are suitable for the problem (something we'll talk about more in other chapters) and decide on the output encoding that you will use — standard neurons, or linear nodes. These things are often decided for you by the input features and targets that you have available to solve the problem. Later on in the learning it can also be useful to re-evaluate the choice by training networks with some input feature missing to see if it improves the results at all.

Normalise inputs Rescale the data by subtracting the mean value from each element of the input vector, and divide by the variance (or alternatively, either the maximum or minus the minimum, whichever is greater).

Split the data into training, testing, and validation sets You cannot test the learning ability of the network on the same data that you trained it on, since it will generally fit that data very well (often too well, overfitting and modelling the noise in the data as well as the generating function). We generally split the data into three sets, one for training, one for testing, and then a third set for validation, which is testing how

well the network is learning during training. The ratio between the sizes of the three groups depends on how much data you have, but is often around 50:25:25. If you do not have enough data for this, use cross-validation instead.

Select a network architecture You already know how many input nodes there will be, and how many output neurons. You need to consider whether you will need a hidden layer at all, and if so how many neurons it should have in it. You might want to consider more than one hidden layer. The more complex the network, the more data it will need to be trained on, and the longer it will take. It might also be more subject to overfitting. The usual method of selecting a network architecture is to try several with different numbers of hidden nodes and see which works best.

Train a network The training of the neural network consists of applying the Multi-layer Perceptron algorithm to the training data. This is usually run in conjunction with early stopping, where after a few iterations of the algorithm through all of the training data, the generalisation ability of the network is tested by using the validation set. The neural network is very likely to have far too many degrees of freedom for the problem, and so after some amount of learning it will stop modelling the generating function of the data, and start to fit the noise and inaccuracies inherent in the training data. At this stage the error on the validation set will start to increase, and learning should be stopped.

Test the network Once you have a trained network that you are happy with, it is time to use the test data for the first (and only) time. This will enable you to see how well the network performs on some data that it has not seen before, and will tell you whether this network is likely to be usable for other data, for which you do not have targets.

4.6 DERIVING BACK-PROPAGATION

This section derives the back-propagation algorithm. This is important to understand how and why the algorithm works. There isn't actually that much mathematics involved except some slightly messy algebra. In fact, there are only three things that you really need to know. One is the derivative (with respect to x) of $\frac{1}{2}x^2$, which is x, and another is the chain rule, which says that $\frac{dy}{dx} = \frac{dy}{dt}\frac{dt}{dx}$. The third thing is very simple: $\frac{dy}{dx} = 0$ if y is not a function of x. With those three things clear in your mind, just follow through the algebra, and you'll be fine. We'll work in simple steps.

4.6.1 The Network Output and the Error

The output of the neural network (the end of the forward phase of the algorithm) is a function of three things:

- the current input ($\mathbf{x}$)
- the activation function $g(\cdot)$ of the nodes of the network
- the weights of the network ($\mathbf{v}$ for the first layer and $\mathbf{w}$ for the second)

We can't change the inputs, since they are what we are learning about, nor can we change the activation function as the algorithm learns. So the weights are the only things that we can vary to improve the performance of the network, i.e., to make it learn. However, we do need to think about the activation function, since the threshold function that we used for

the Perceptron is not differentiable (it has a discontinuity at 0). We'll think about a better one in Section 4.6.3, but first we'll think about the error of the network. Remember that we have run the algorithm forwards, so that we have fed the inputs ($\mathbf{x}$) into the algorithm, used the first set of weights ($\mathbf{v}$) to compute the activations of the hidden neurons, then those activations and the second set of weights ($\mathbf{w}$) to compute the activations of the output neurons, which are the outputs of the network ($\mathbf{y}$). Note that I'm going to use i to be an index over the input nodes, j to be an index over the hidden layer neurons, and k to be an index over the output neurons.

4.6.2 The Error of the Network

When we discussed the Perceptron learning rule in the previous chapter we motivated it by minimising the error function $E = \sum_{k=1}^{N} y_k - t_k$. We then invented a learning rule that made this error smaller. We are going to do much better this time, because everything is computed from the principles of gradient descent.

To begin with, let's think about the error of the network. This is obviously going to have something to do with the difference between the outputs $\mathbf{y}$ and the targets $\mathbf{t}$, but I'm going to write it as $E(\mathbf{v}, \mathbf{w})$ to remind us that the only things that we can change are the weights $\mathbf{v}$ and $\mathbf{w}$, and that changing the weights changes the output, which in turn changes the error.

For the Perceptron we computed the error as $E = \sum_{k=1}^{N} y_k - t_k$, but there are some problems with this: if $t_k > y_k$, then the sign of the error is different to when $y_k > t_k$, so if we have lots of output nodes that are all wrong, but some have positive sign and some have negative sign, then they might cancel out. Instead, we'll choose the sum-of-squares error function, which calculates the difference between y_k and t_k for each node k, squares them, and adds them together (I've missed out the $\mathbf{v}$ in $E(\mathbf{w})$ because we don't use them here):

$$E(\mathbf{w}) = \frac{1}{2}\sum_{k=1}^{N}(y_k - t_k)^2 \tag{4.20}$$

$$= \frac{1}{2}\sum_{k=1}^{N}\left[g\left(\sum_{j=0}^{M} w_{jk}a_j\right) - t_k\right]^2 \tag{4.21}$$

The second line adds in the input from the hidden layer neurons and the second-layer weights to decide on the activations of the output neurons. For now we're going to think about the Perceptron and index the input nodes by i and the output nodes by k, so Equation (4.21) will be replaced by:

$$\frac{1}{2}\sum_{k=1}^{N}\left[g\left(\sum_{i=0}^{L} w_{ik}x_i\right) - t_k\right]^2. \tag{4.22}$$

Now we can't differentiate the threshold function, which is what the Perceptron used for $g(\cdot)$, because it has a discontinuity (sudden jump) at the threshold value. So I'm going to miss it out completely for the moment. Also, for the Perceptron there are no hidden neurons, and so the activation of an output neuron is just $y_\kappa = \sum_{i=0}^{L} w_{i\kappa}x_i$ where x_i is the value of an input node, and the sum runs over the number of input nodes, including the bias node.

We are going to use a gradient descent algorithm that adjusts each weight $w_{\iota\kappa}$ for fixed

values of ι and κ, in the direction of the negative gradient of $E(\mathbf{w})$. In what follows, the notation ∂ means the partial derivative, and is used because there are lots of different functions that we can differentiate E with respect to: all of the different weights. If you don't know what a partial derivative is, think of it as being the same as a normal derivative, but taking care that you differentiate in the correct direction. The gradient that we want to know is how the error function changes with respect to the different weights:

$$\begin{aligned}\frac{\partial E}{\partial w_{\iota\kappa}} &= \frac{\partial}{\partial w_{\iota\kappa}}\left(\frac{1}{2}\sum_{k=1}^{N}(y_k - t_k)^2\right) \\ &= \frac{1}{2}\sum_{k=1}^{N} 2(y_k - t_k)\frac{\partial}{\partial w_{\iota\kappa}}\left(y_k - \sum_{i=0}^{L} w_{i\kappa}x_i\right) \qquad (4.23)\end{aligned}$$

$$(4.24)$$

Now t_k is not a function of any of the weights, since it is a value given to the algorithm, so $\frac{\partial t_k}{\partial w_{\iota\kappa}} = 0$ for all values of k, ι, κ, and the only part of $\sum_{i=0}^{L} w_{i\kappa}x_i$ that is a function of $w_{\iota\kappa}$ is when $i = \iota$, that is $w_{\iota\kappa}$ itself, which has derivative 1. Hence:

$$\frac{\partial E}{\partial w_{\iota\kappa}} = \sum_{k=1}^{N}(t_k - y_k)(-x_\iota). \qquad (4.25)$$

Now the idea of the weight update rule is that we follow the gradient downhill, that is, in the direction $-\frac{\partial E}{\partial w_{\iota\kappa}}$. So the weight update rule (when we include the learning rate η) is:

$$w_{\iota\kappa} \leftarrow w_{\iota\kappa} + \eta(t_\kappa - y_\kappa)x_\iota, \qquad (4.26)$$

which hopefully looks familiar (see Equation (3.3)). Note that we are computing y_κ differently: for the Perceptron we used the threshold activation function, whereas in the work above we ignored the threshold function. This isn't very useful if we want units that act like neurons, because neurons either fire or do not fire, rather than varying continuously. However, if we want to be able to differentiate the output in order to use gradient descent, then we need a differentiable activation function, so that's what we'll talk about now.

4.6.3 Requirements of an Activation Function

In order to model a neuron we want an activation function that has the following properties:

- it must be differentiable so that we can compute the gradient
- it should saturate (become constant) at both ends of the range, so that the neuron either fires or does not fire
- it should change between the saturation values fairly quickly in the middle

There is a family of functions called sigmoid functions because they are S-shaped (see Figure 4.5) that satisfy all those criteria perfectly. The form in which it is generally used is:

$$a = g(h) = \frac{1}{1 + \exp(-\beta h)}, \qquad (4.27)$$

where β is some positive parameter. One happy feature of this function is that its derivative has an especially nice form:

$$
\begin{aligned}
g'(h) = \frac{dg}{dh} &= \frac{d}{dh}(1+e^{-\beta h})^{-1} && (4.28)\\
&= -1(1+e^{-\beta h})^{-2}\frac{de^{-\beta h}}{dh} && (4.29)\\
&= -1(1+e^{-\beta h})^{-2}(-\beta e^{-\beta h}) && (4.30)\\
&= \frac{\beta e^{-\beta h}}{(1+e^{-\beta h})^2} && (4.31)\\
&= \beta g(h)(1-g(h)) && (4.32)\\
&= \beta a(1-a) && (4.33)
\end{aligned}
$$

We'll be using this derivative later. So we've now got an error function and an activation function that we can compute derivatives of. We will consider some other possible activation functions for the output neurons in Section 4.6.5 and an alternative error function in Section 4.6.6. The next thing to do is work out how to use them in order to adjust the weights of the network.

4.6.4 Back-Propagation of Error

It is now that we'll need the chain rule that I reminded you of earlier. In the form that we want, it looks like this:

$$\frac{\partial E}{\partial w_{\zeta\kappa}} = \frac{\partial E}{\partial h_\kappa}\frac{\partial h_\kappa}{\partial w_{\zeta\kappa}}, \tag{4.34}$$

where $h_\kappa = \sum_{j=0}^{M} w_{j\kappa}a_\zeta$ is the input to output-layer neuron κ; that is, the sum of the activations of the hidden-layer neurons multiplied by the relevant (second-layer) weights. So what does Equation (4.34) say? It tells us that if we want to know how the error at the output changes as we vary the second-layer weights, we can think about how the error changes as we vary the input to the output neurons, and also about how those input values change as we vary the weights.

Let's think about the second term first (in the third line we use the fact that $\frac{\partial w_{j\kappa}}{\partial w_{\zeta\kappa}} = 0$ for all values of j except $j = \zeta$, when it is 1):

$$
\begin{aligned}
\frac{\partial h_\kappa}{\partial w_{\zeta\kappa}} &= \frac{\partial \sum_{j=0}^{M} w_{j\kappa}a_j}{\partial w_{\zeta\kappa}} && (4.35)\\
&= \sum_{j=0}^{M}\frac{\partial w_{j\kappa}a_j}{\partial w_{\zeta\kappa}} && (4.36)\\
&= a_\zeta. && (4.37)
\end{aligned}
$$

Now we can worry about the $\frac{\partial E}{\partial h_\kappa}$ term. This term is important enough to get its own term, which is the error or delta term:

$$\delta_o(\kappa) = \frac{\partial E}{\partial h_\kappa}. \tag{4.38}$$

Let's start off by trying to compute this error for the output. We can't actually compute

it directly, since we don't know much about the inputs to a neuron, we just know about its output. That's fine, because we can use the chain rule again:

$$\delta_o(\kappa) = \frac{\partial E}{\partial h_\kappa} = \frac{\partial E}{\partial y_\kappa}\frac{\partial y_\kappa}{\partial h_\kappa}. \tag{4.39}$$

Now the output of output layer neuron κ is

$$y_\kappa = g(h_\kappa^{\text{output}}) = g\left(\sum_{j=0}^{M} w_{j\kappa} a_j^{\text{hidden}}\right), \tag{4.40}$$

where $g(\cdot)$ is the activation function. There are different possible choices for $g(\cdot)$ including the sigmoid function given in Equation (4.27), so for now I'm going to leave it as a function. I've also started labelling whether h refers to an output or hidden layer neuron, just to avoid any possible confusion. We don't need to worry about this for the activations, because we use y for the activations of output neurons and a for hidden neurons. In Equation (4.43) I've substituted in the expression for the error at the output, which we computed in Equation (4.21):

$$\begin{aligned}
\delta_o(\kappa) &= \frac{\partial E}{\partial g\left(h_\kappa^{\text{output}}\right)}\frac{\partial g\left(h_\kappa^{\text{output}}\right)}{\partial h_\kappa^{\text{output}}} && (4.41)\\
&= \frac{\partial E}{\partial g\left(h_\kappa^{\text{output}}\right)} g'\left(h_\kappa^{\text{output}}\right) && (4.42)\\
&= \frac{\partial}{\partial g\left(h_\kappa^{\text{output}}\right)}\left[\frac{1}{2}\sum_{k=1}^{N}\left(g(h_k^{\text{output}}) - t_k\right)^2\right] g'\left(h_\kappa^{\text{output}}\right) && (4.43)\\
&= \left(g(h_\kappa^{\text{output}}) - t_\kappa\right) g'(h_\kappa^{\text{output}}) && (4.44)\\
&= (y_\kappa - t_\kappa) g'(h_\kappa^{\text{output}}), && (4.45)
\end{aligned}$$

where $g'(h_\kappa)$ denotes the derivative of g with respect to h_κ. This will change depending upon which activation function we use for the output neurons, so for now we will write the update equation for the output layer weights in a slightly general form and pick it up again at the end of the section:

$$\begin{aligned}
w_{\zeta\kappa} &\leftarrow w_{\zeta\kappa} - \eta\frac{\partial E}{\partial w_{\zeta\kappa}}\\
&= w_{\zeta\kappa} - \eta\delta_o(\kappa)a_\zeta. && (4.46)
\end{aligned}$$

where we are using the minus sign because we want to go downhill to minimise the error.

We don't actually need to do too much more work to get to the first layer weights, v_ι, which connects input ι to hidden node ζ. We need the chain rule (Equation (4.34)) one more time to get to these weights, remembering that we are working backwards through the network so that k runs over the output nodes. The way to think about this is that each hidden node contributes to the activation of all of the output nodes, and so we need to consider all of these contributions (with the relevant weights).

$$\delta_h(\zeta) = \sum_{k=1}^{N} \frac{\partial E}{\partial h_k^{\text{output}}} \frac{\partial h_k^{\text{output}}}{\partial h_\zeta^{\text{hidden}}} \tag{4.47}$$

$$= \sum_{k=1}^{N} \delta_o(k) \frac{\partial h_k^{\text{output}}}{\partial h_\zeta^{\text{hidden}}}, \tag{4.48}$$

where we obtain the second line by using Equation (4.38). We now need a nicer expression for that derivative. The important thing that we need to remember is that inputs to the output layer neurons come from the activations of the hidden layer neurons multiplied by the second layer weights:

$$h_\kappa^{\text{output}} = \sum_{j=0}^{M} w_{j\kappa} g\left(h_j^{\text{hidden}}\right), \tag{4.49}$$

which means that:

$$\frac{\partial h_\kappa^{\text{output}}}{\partial h_\zeta^{\text{hidden}}} = \frac{\partial g\left(\sum_{j=0}^{M} w_{j\kappa} h_j^{\text{hidden}}\right)}{\partial h_j^{\text{hidden}}}. \tag{4.50}$$

We can now use a fact that we've used before, which is that $\frac{\partial h_\zeta}{\partial h_j} = 0$ unless $j = \zeta$, when it is 1. So:

$$\frac{\partial h_\kappa^{\text{output}}}{\partial h_\zeta^{\text{hidden}}} = w_{\zeta\kappa} g'(a_\zeta). \tag{4.51}$$

The hidden nodes always have sigmoidal activation functions, so that we can use the derivative that we computed in Equation (4.33) to get that $g'(a_\zeta) = \beta a_\zeta (1 - a_\zeta)$, which allows us to compute:

$$\delta_h(\zeta) = \beta a_\zeta (1 - a_\zeta) \sum_{k=1}^{N} \delta_o(k) w_\zeta. \tag{4.52}$$

This means that the update rule for v_ι is:

$$v_\iota \leftarrow v_\iota - \eta \frac{\partial E}{\partial v_\iota}$$

$$= v_\iota - \eta a_\zeta (1 - a_\zeta) \left(\sum_{k=1}^{N} \delta_o(k) w_\zeta \right) x_\iota. \tag{4.53}$$

Note that we can do exactly the same computations if the network has extra hidden layers between the inputs and the outputs. It gets harder to keep track of which functions we should be differentiating, but there are no new tricks needed.

4.6.5 The Output Activation Functions

The sigmoidal activation function that we have created is aimed at making the nodes act a bit like neurons, either firing or not firing. This is very important in the hidden layer, but earlier in the chapter we have observed two cases where it is not suitable for the output neurons. One was regression, where we want the output to be continuous, and one was multi-class classification, where we want only one of the output neurons to fire. We identified possible activation functions for these cases, and here we will derive the delta term δ_o for them. As a reminder, the three functions are:

Linear $y_\kappa = g(h_\kappa) = h_\kappa$

Sigmoidal $y_\kappa = g(h_\kappa) = 1/(1 + \exp(-\beta h_\kappa)$

Soft-max $y_\kappa = g(h_\kappa) = \exp(h_\kappa)/\sum_{k=1}^{N} \exp(h_k)$

For each of these we need the derivative with respect to each of the output weights so that we can use Equation (4.45).

This is easy for the first two cases, and tells us that for linear outputs $\delta_o(\kappa) = (y_\kappa - t(\kappa))y_\kappa$, while for sigmoidal outputs it is $\delta_o(\kappa) = \beta(y_\kappa - t(\kappa))y_\kappa(1 - y_\kappa)$.

However, we have to do some more work for the soft-max case, since we haven't differentiated it yet. If we write it as:

$$\frac{\partial}{\partial h_K} y_\kappa = \frac{\partial}{\partial h_K} \left(\exp(h_\kappa) \left(\sum_{k=1}^{N} \exp(h_k) \right)^{-1} \right) \tag{4.54}$$

then the problem becomes clear: we have a product of two things to differentiate, and three different indices to worry about. Further, the k index runs over all the output nodes, and so includes K and κ within it. There are two cases: either $K = \kappa$, or it does not. If they are the same, then we can write that $\frac{\partial \exp(h_\kappa)}{\partial h_{kappa}} = \exp(h_\kappa)$ to get (where the last term in the first line comes from the use of the chain rule):

$$\begin{aligned} & \frac{\partial}{\partial h_\kappa} \left(\exp(h_\kappa) \left(\sum_{k=1}^{N} \exp(h_k) \right)^{-1} \right) \\ = \quad & \exp(h_\kappa) \left(\sum_{k=1}^{N} \exp(h_k) \right)^{-1} - \exp(h_\kappa) \left(\sum_{k=1}^{N} \exp(h_k) \right)^{-2} \exp(h_\kappa) \\ = \quad & y_\kappa(1 - y_\kappa). \end{aligned} \tag{4.55}$$

For the case where $K \neq \kappa$ things are a little easier, and we get:

$$\begin{aligned} \frac{\partial}{\partial h_K} \exp(h_\kappa) \left(\sum_{k=1}^{N} \exp(h_k) \right)^{-1} \quad &= \quad - \exp(h_\kappa) \exp(h_K) \left(\sum_{k=1}^{N} \exp(h_k) \right)^{-2} \\ &= \quad -y_\kappa y_K. \end{aligned} \tag{4.56}$$

Using the Kronecker delta function δ_{ij}, which is 1 if $i = j$ and 0 otherwise, we can write the two cases in one equation to get the delta term:

$$\delta_o(\kappa) = (y_\kappa - t_\kappa) y_\kappa (\delta_{\kappa K} - y_K). \tag{4.57}$$

The very last thing to think about is whether or not the sum-of-squares error function is always the best one to use.

4.6.6 An Alternative Error Function

We have been using the sum-of-squares error function throughout this chapter. It is easy to compute and works well in general; we will see another benefit of it in Section 9.2. However, for classification tasks we are assuming that the outputs represent different, independent classes, and this means that we can think of the activations of the nodes as giving us a probability that each class is the correct one.

In this probabilistic interpretation of the outputs, we can ask how likely we are to see each target given the set of weights that we are using. This is known as the likelihood and the aim is to maximise it, so that we predict the targets as well as possible. If we have a 1 output node, taking values 0 or 1, then the likelihood is:

$$p(t|\mathbf{w}) = y_k^{t_k} \left(1 - y_k\right)^{1-t_k}. \tag{4.58}$$

In order to turn this into a minimisation function we put a minus sign in front, and it will turn out to be useful to take the logarithm of it as well, which produces the cross-entropy error function, which is (for N output nodes):

$$E_{\text{ce}} = -\sum_{k=1}^{N} t_k \ln(y_k), \tag{4.59}$$

where ln is the natural logarithm. This error function has the nice property that when we use the soft-max function the derivatives are very easy because the exponential and logarithm are inverse functions, and so the delta term is simply $\delta_o(\kappa) = y_\kappa - t_\kappa$.

FURTHER READING

The original papers describing the back-propagation algorithm are listed here, along with a well-known introduction to neural networks:

- D.E. Rumelhart, G.E. Hinton, and R.J. Williams. Learning internal representations by back-propagating errors. *Nature*, 323(99):533–536, 1986a.
- D.E. Rumelhart, J.L. McClelland, and the PDP Research Group, editors. *Parallel Distributed Processing*. MIT Press, Cambridge, MA, 1986b.
- R. Lippmann. An introduction to computing with neural nets. *IEEE ASSP Magazine*, pages 4–22, 1987.

For more on the Universal Approximation Theorem, which shows that one hidden layer is sufficient, some references (which are not for the mathematically faint-hearted) are:

- G. Cybenko. Approximations by superpositions of sigmoidal functions. *Mathematics of Control, Signals, and Systems*, 2(4):303–314, 1989.
- Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4(2):251–257, 1991.

If you are interested in novelty detection, then a review article is:

- S. Marsland. Novelty detection in learning systems. *Neural Computing Surveys*, 3: 157–195, 2003.

The topics in this chapter are covered in any book on machine learning and neural networks. Different treatments are given by:

- Sections 5.1–5.3 of C.M. Bishop. *Pattern Recognition and Machine Learning*. Springer, Berlin, Germany, 2006.
- Section 5.4 of J. Hertz, A. Krogh, and R.G. Palmer. *Introduction to the Theory of Neural Computation*. Addison-Wesley, Redwood City, CA, USA, 1991.
- Sections 4.4–4.7 of T. Mitchell. *Machine Learning*. McGraw-Hill, New York, USA, 1997.

PRACTICE QUESTIONS

Problem 4.1 Work through the MLP shown in Figure 4.2 to ensure that it does solve the XOR problem.

Problem 4.2 Suppose that the local power company wants to predict electricity demand for the next 5 days. They have the data about daily demand for the last 5 years. Typically, the demand will be a number between 80 and 400.

1. Describe how you could use an MLP to make the prediction. What parameters would you have to choose, and what do you think would be sensible values for them?
2. If the weather forecast for the next day, being the estimated temperatures for daytime and nighttime, was available, how would you add that into your system?
3. Do you think that this system would work well for predicting power consumption? Are there demands that it would not be able to predict?

Problem 4.3 Design an MLP that would learn to hyphenate words correctly. You would have a dictionary that shows correct hyphenation examples for lots of words, and you need to choose methods of encoding the inputs and outputs that say whether a hyphen is allowed between each pair of letters. You should also describe how you would perform training and testing.

Problem 4.4 Would the previous system be better than just using the dictionary?

Problem 4.5 Modify the code on the book website to work sequentially rather than in batch mode. Compare the results on the iris dataset.

Problem 4.6 Modify the code so that it performs minibatch optimisation and then stochastic gradient descent (both described in Section 4.2.7) and compare the results with using the standard algorithm on the Pima Indian dataset that was described in Section 3.4.4. Experiment with different sizes for the minibatches.

Problem 4.7 Modify the code to allow another hidden layer to be used. You will have to work out the gradient as well in order to compute the weight updates for the extra layer of weights. Test this new network on the Pima Indian dataset that was described in Section 3.4.4.

Problem 4.8 Modify the code to use the alternative error term in Section 4.6.6 and see what difference it makes for classification problems.

Problem 4.9 A hospital manager wants to predict how many beds will be needed in the geriatric ward. He asks you to design a neural network method for making this prediction. He has data for the last 5 years that cover:

- The number of people in the geriatric ward each week.
- The weather (average day and night temperatures).
- The season of the year (spring, summer, autumn, winter).
- Whether or not there was an epidemic on (use a binary variable: yes or no).

Design a suitable MLP for this problem, considering how you would choose the number of hidden neurons, the inputs (and whether there are any other inputs you need) and the preprocessing, and whether or not you would expect the system to work.

Problem 4.10 Look into the MNIST dataset that is available via the book website. Implement an MLP to learn about them and test different numbers of hidden nodes.

Problem 4.11 A recurrent network has some of its outputs connected to its own inputs, so that the outputs at time t are fed back into the network at time $t + 1$. This can be a different way to deal with time-series data. Modify the MLP code so that it acts as a recurrent network, and test it out on the Palmerston North ozone data on the book website.

Problem 4.12 The alternative activation function that can be used in $\tanh(h)$. Show that $\tanh(h) = 2g(2h) - 1$, where g is given by Equation (4.2). Use this to show that there is an exactly equivalent MLP using the tanh activation function. Modify the code to implement it.

CHAPTER 5

Radial Basis Functions and Splines

In the Multi-layer Perceptron, the activations of the hidden nodes were decided by whether the inputs times the weights were above a threshold that made the neuron fire. While we had to sacrifice some of this ideal to the requirement for differentiability, it was still the case that the product of the inputs and the weights was summed, and if it was well above the threshold then the neuron fired, if it was well below the threshold it did not, and between those values it acted linearly. For any input vector several of the neurons could fire, and the outputs of these neurons times their weights were then summed in the second layer to decide which neurons should fire there. This has the result that the activity in the hidden layer is distributed over the neurons there, and it is this pattern of activation that was used as the inputs to the next layer.

In this chapter we are going to consider a different approach, which is to use local neurons, where each neuron only responds to inputs in one particular part of the input space. The argument is that if inputs are similar, then the responses to those inputs should also be similar, and so the same neuron should respond. Extending this a little, if an input is between two others, then the neurons that respond to each of the inputs should both fire to some extent. We can justify this by thinking about a typical classification task, since if two input vectors are similar, then they should presumably belong to the same class. In order to understand this better we are going to need two concepts, one from machine learning, weight space, and one from neuroscience, receptive fields.

5.1 RECEPTIVE FIELDS

In Section 2.1.1 we argued that one way to compute the activations of neurons was to use the concept of weight space to abstractly plot them in the same set of dimensions as the inputs, and to have neurons that were 'closer' to the input being more highly activated. To see why this might be a good idea, we need to look at the idea of receptive fields.

Suppose that we have a set of 'nodes' (since they are no longer models of neurons in any sense, the terminology changes to call the components of the network 'nodes'). As in Section 2.1.1, these nodes are imagined to be sitting in weight space, and we can change their locations by adjusting the weights. We want to decide how strongly a node matches the current input, so just like in Section 2.1.1 we pretend that input space and weight space are the same, and measure the distance between the input vector position and the position of

each node. The activation of these nodes can then be computed according to their distance to the current input, in ways that we'll get to later.

To put this idea of nodes firing when they are 'close' to the input into some sort of context, we are going to have a quick digression into the idea of receptive fields. Imagine the back of your eye. Light comes through the pupil and hits the retina, which has light-sensitive cells (rods and cones) spread across it. Now suppose that you look at the night sky with one bright star in it. How will you see the star, or to put it another way, which rods on your retina will detect the light of the star? The obvious answer is that there will be one localised area of your retina that picks up the light, and a few rods that are close together will detect it, while the rest don't see anything except the dark night sky. However, if you looked at the sky again a few hours later, when the position of the star in the sky had changed, then different rods would detect it (assuming that your head is in the same position, of course). So even though the appearance of the star is the same, because the relative position of the star has changed because the earth has rotated, so the rods that you use to detect it have changed. The receptive field of a particular rod within your eye is the area on your retina that it responds to light from. We can extend this to particular sensory neurons as well, so that the response of particular neurons may depend on the location of the stimulus.

We might want to know what shape these receptive fields are, and how the response of the rod (or neuron) changes as the stimulus moves away from the area that matches the rod. If we were equipped with a neuroscience lab with electrodes and measurement devices (and animals, and ethics approval) then we could measure exactly this. We could show pictures of light blobs on dark backgrounds to animals and measure the amount of neuronal activity in particular neurons as the position of the blob moved. And people have done exactly this.

For now, let's just try a thought experiment: it's simpler and cheaper, and nothing gets hurt. If we are looking at our star again, then we have already worked out that there is a set of rods that is detecting the light, and plenty of others that aren't. What about a rod that is just at the boundary where the light from the star stops being visible? Let's pick one where its receptive field stops just to the left of this boundary, so that the neuron is not firing. Now move your head slightly to the right, so that it is just inside. What happens? For a real neuron it would start to spike. Assuming that the number of times the neuron spikes says how bright the light that it detects is (which probably isn't exactly true), then it wouldn't spike very often. As you move your head to the right again so that the light on that particular neuron gets brighter and brighter, that neuron will spike more and more often, until once you've moved your head past the light and the spiking slows down, and eventually stops. The left-hand graph of Figure 5.1 shows this, with the points plotted and a smooth curve that goes through them.

Now suppose that you repeat the experiment, but this time you start with the star below your vision and move your head down until you can see it, and then keep on moving your head further down. The exact same thing happens. The graph in the middle of Figure 5.1 shows this. So for this example, it doesn't matter where the point of light is with regard to the neuron, just how far away it is. In other words, if we were to put the star on a wire circle centred on one particular rod within our eye (a bit painful, but that's the good thing about thought experiments), then as we moved the light along the wire the activation of the rod would not change. Only the radius of the circle matters, which is why functions that model this are known as radial functions. Mathematically, we say that they only depend on the two-norm $\|\mathbf{x}_i - \mathbf{x}\|_2$, that is the Euclidean distance between the point and the centre of the circle.

The main thing that we have not decided yet is how the drop-off should occur from response to maximum brightness to nothing. For real neurons the drop-off has to change

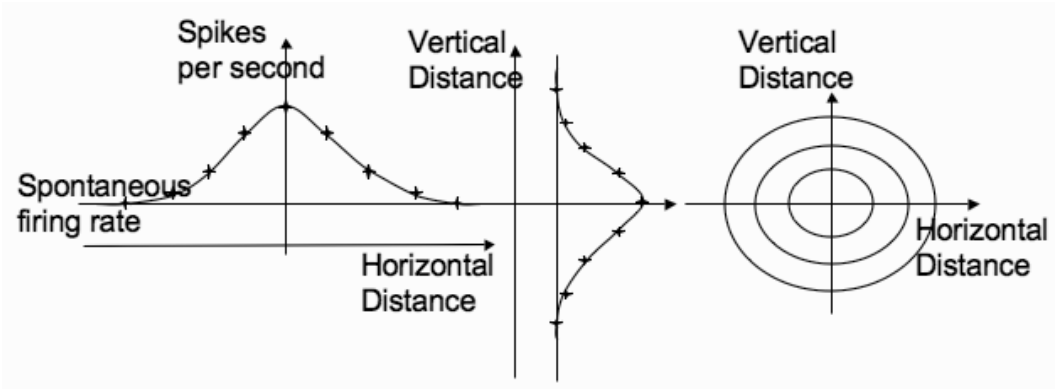

FIGURE 5.1 *Left:* Count of the number of spikes per second as the distance of a rod from the light varies horizontally. Note that it does not go to zero, but to the spontaneous firing rate of the neuron, which is how often it fires without input. *Centre:* The same thing for vertical motion. *Right:* The combination of the two makes a set of circles.

between integer values, but for our mathematical model it doesn't: we can make it decrease smoothly, so that we can use well-behaved (that is, differentiable) mathematical functions. Then we can pick any function that we can differentiate, that decreases symmetrically (in all directions, or radially) from a maximum to zero. There are obviously lots of possible functions with this property that we can pick, but for now we'll go with by far the most common one in statistics, the Gaussian, something that was important enough to get its own section earlier on (Section 2.4.3). It doesn't really go to 0, but if we truncate it a little, then the output value becomes 0 fairly quickly as we move away from the centre. We do not typically use a real Gaussian function for the activation function, ignoring the normalisation to get an approximation to it written as:

$$g(\mathbf{x}, \mathbf{w}, \sigma) = \exp\left(\frac{-\|\mathbf{x} - \mathbf{w}\|^2}{2\sigma^2}\right). \tag{5.1}$$

The choice of σ in this equation is quite important, since it controls the width of the Gaussian. If we make it infinitely large, then the neuron responds to every input. Suppose instead that we make σ smaller and smaller, so that the Gaussian gets thinner and thinner. This means that the receptive field gets narrower and narrower. Eventually, this neuron will respond to exactly one stimulus, and even then, if the input is corrupted by noise, it won't recognise it. This function is sometimes known as an indicator or delta function. Picking the value of σ for each individual node needs to be part of the algorithm.

So, we can use Gaussians to model these receptive fields for neurons so that nodes will fire strongly if the input is close to them, less strongly if the input is further away, and not at all if it is even further away. We are going to see several neural networks in different chapters that use these ideas, mostly for unsupervised learning, but first we will see a supervised one, the radial basis function (RBF) network. Figure 5.2 shows a set of nodes that represent radial bases in weight space. They are often known as centres because they each form the centre of their own circle or ellipse.

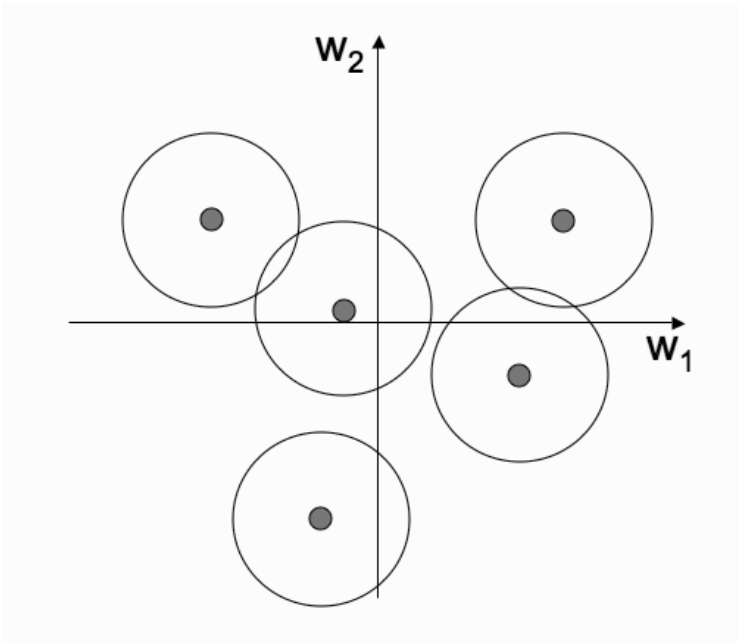

FIGURE 5.2 The effect of radial basis functions in weight space. The points show the position of the RBF in weight space, while the circle around each point shows the receptive field of the node. In higher dimensions these circles become hyperspheres.

5.2 THE RADIAL BASIS FUNCTION (RBF) NETWORK

The argument that started this chapter was that inputs that are close together should generate the same output, whereas inputs that are far apart should not. We have seen that using Gaussian activations, where the output of a neuron is proportional to the distance between the input and the weight, gives us receptive fields. The Gaussian activations mean that normalising the input vectors is very important for the RBF network; Section 14.1.3 will make the reason for this clearer. For any input that we present to a set of these neurons, some of them will fire strongly, some weakly, and some will not fire at all, depending upon the distance between the weights and the particular input in weight space. We can treat these nodes as a hidden layer, just as we did for the MLP, and connect up some output nodes in a second layer. This simply requires adding weights from each hidden (RBF) neuron to a set of output nodes. This is known as an RBF network, and a schematic is shown in Figure 5.3. In the figure, the nodes in both the hidden and output layer are drawn the same, but we haven't decided what kind of nodes to use in the output layer—they don't need to have Gaussian activations. The simplest solution is to use McCulloch and Pitts neurons, in which case this second part of the network is simply a Perceptron network. Note that there is a bias input for the output layer, which deals with the situation when none of the RBF neurons fire. Since we already know exactly how to train the Perceptron, training this second part of the network is easy. The questions that we need to ask are whether or not it is any better than using a Perceptron, and how to train the first layer weights that position the RBF neurons.

A little thought should persuade you that this network is better than just a Perceptron, since the inputs that are given to the Perceptron are non-linear functions of the inputs. In

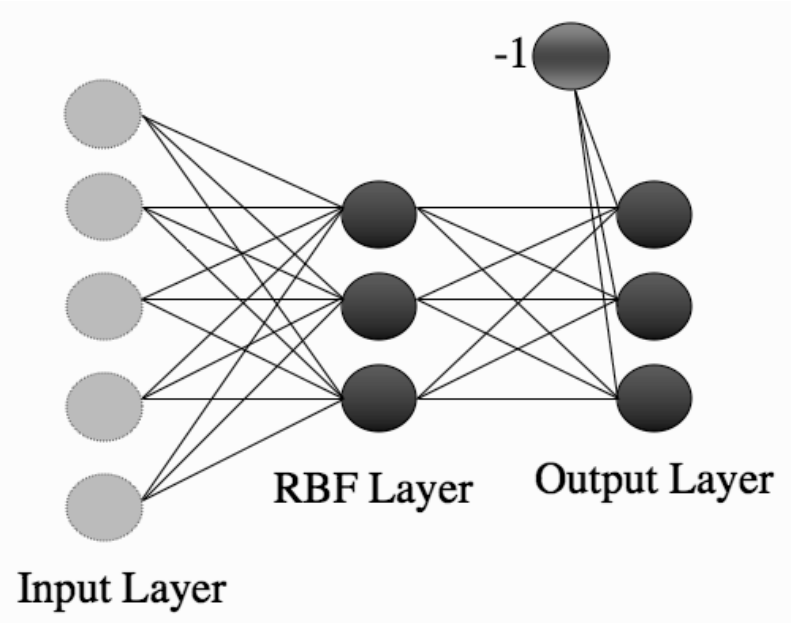

FIGURE 5.3 The Radial Basis Function network consists of input nodes connected by weights to a set of RBF neurons, which fire proportionally to the distance between the input and the neuron in weight space. The activations of these nodes are used as inputs to the second layer, which consists of linear nodes. The schematic looks very similar to the MLP except for the lack of a bias in the hidden layer.

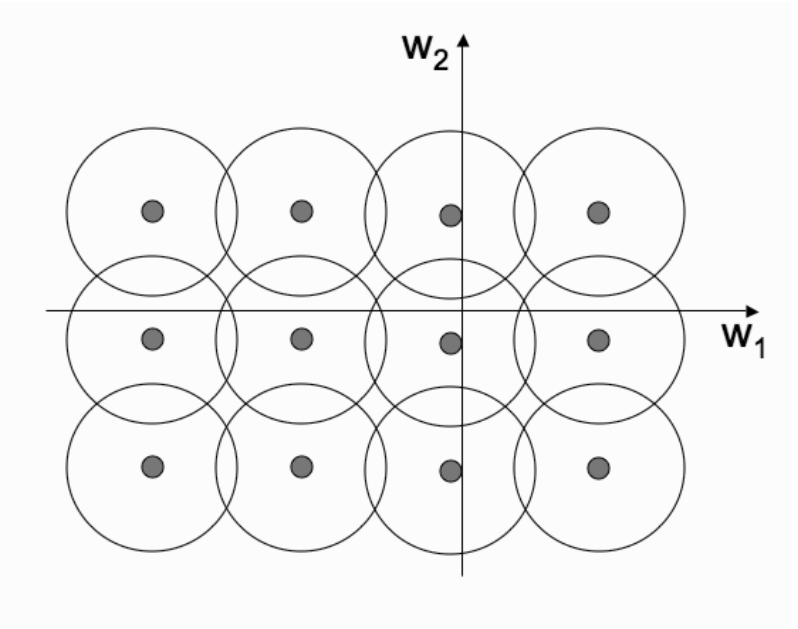

FIGURE 5.4 We can space out RBF nodes to cover the whole of space by continuing this pattern everywhere, so that the network acts as a universal approximator, since there is an output for every possible input.

fact, the RBF network is a universal approximator, just like the MLP. To see this, imagine that we fill the entire space with RBF nodes equally spaced in all directions, so that their receptive fields just overlap, as in Figure 5.4. Now, no matter what the input, there is an RBF node that recognises it and can respond appropriately to it. If we need to make the outputs more finely grained, then we just add more RBFs at the relevant positions and reduce the radius of the receptive fields; and if we don't care, we can just make the receptive fields of each node bigger and use fewer of them.

RBF networks never have more than one layer of non-linear neurons, in contrast to the MLP. However, there are many similarities between the two networks: they are both supervised learning algorithms that form universal approximators. In fact, it turns out that you can turn one into the other because the two types of neuron firing rules (RBFs based on distance and MLPs on inner product) are related. This fact will turn up in another form in Section 14.1.3. The most important difference between them is the fact that the MLP uses the hidden nodes to separate the space using hyperplanes, which are global, while the RBF uses them to match functions locally.

In an RBF network, when we see an input several of the nodes will activate to some degree or other, according to how close they are to the input, and the combination of these activations will enable the network to decide how to respond. Using the analogy we had earlier of looking at a star, suppose that the star is replaced by a torch, and somebody is signalling directions to us. If the torch is high (at 12 o'clock) we go forwards, low (6 o'clock) we go backwards, and left and right (9 and 3 o'clock, respectively) mean that we turn. The RBF network would work in such a way that if the torch was at 2 o'clock or thereabouts, then we would do some of the 12 o'clock action and a bit more of the 3 o'clock action, but none of the 6 or 9 o'clock actions. So we would move forwards and to the right. This adding

up of the contributions from the different basis functions according to how active they are means that our responses are local.

5.2.1 Training the RBF Network

In the MLP we used back-propagation of error to adjust first the output layer weights, and then the hidden layer weights. We can do exactly the same thing with the RBF network, by differentiating the relevant activation functions. However, there are simpler and better alternatives for RBF networks. They do not need to compute gradients for the hidden nodes and so they are significantly faster. The important thing to notice is that the two types of node provide different functions, and so they do not need to be trained together. The purpose of the RBF nodes in the hidden layer is to find a non-linear representation of the inputs, while the purpose of the output layer is to find a linear combination of those hidden nodes that does the classification. So we can split the training into two parts: position the RBF nodes, and then use the activations of those nodes to train the linear outputs. This makes things much simpler. For the linear outputs we can use an algorithm that we already know: the Perceptron (Section 3.3).

However, we need to work out something different for the first layer weights, which control the positions of the RBF nodes. One thing that we can do is to avoid the problem of training completely by randomly picking some of the datapoints to act as basis locations. Provided that our training data are representative of the full dataset, this often turns out to be a good solution. The other thing that we can do is to try to position the nodes so that they are representative of typical inputs. This is precisely the problem solved by several unsupervised learning methods, and we are going to see several algorithms for doing this in Chapter 14. For the RBF network, the most common one is the k-means algorithm that is described in Section 14.1. Thus, training an RBF network can be reduced to using two other algorithms that are commonly used in machine learning, one after the other. This is known as a hybrid algorithm, since it combines supervised and unsupervised learning.

The Radial Basic Function Algorithm

- Position the RBF centres by either:
  - using the k-means algorithm to initialise the positions of the RBF centres OR
  - setting the RBF centres to be randomly chosen datapoints
- Calculate the actions of the RBF nodes using Equation (5.1)
- Train the output weights by either:
  - using the Perceptron OR
  - computing the pseudo-inverse of the activations of the RBF centres (this will be described shortly)

To implement this in Python we can simply `import` the other algorithms, and use them directly (if they are in different directories, then you need to add them to the `PYTHONPATH` variable; precisely how to do this varies between programming IDEs. The training is then very simple:

```
    def rbftrain(self,inputs,targets,eta=0.25,niterations=100):

        if self.usekmeans==0:
            # Version 1: set RBFs to be datapoints
            indices = range(self.ndata)
            np.random.shuffle(indices)
            for i in range(self.nRBF):
                self.weights1[:,i] = inputs[indices[i],:]
        else:
            # Version 2: use k-means
            self.weights1 = np.transpose(self.kmeansnet.kmeanstrain(inputs))

        for i in range(self.nRBF):
            self.hidden[:,i] = np.exp(-np.sum((inputs - np.ones((1,self.nin))↩
            *self.weights1[:,i])**2,axis=1)/(2*self.sigma**2))
        if self.normalise:
            self.hidden[:,:-1] /= np.transpose(np.ones((1,np.shape(self.↩
            hidden)[0]))*self.hidden[:,:-1].sum(axis=1))

        # Call Perceptron without bias node (since it adds its own)
        self.perceptron.pcntrain(self.hidden[:,:-1],targets,eta,niterations)
```

In fact, because of this separation of the two learning parts, we can do better than a Perceptron for training the outputs weights. For each input vector, we compute the activation of all the hidden nodes, and assemble them into a matrix $\mathbf{G}$. So each element of $\mathbf{G}$, say $\mathbf{G}_{ij}$, consists of the activation of hidden node j for input i. The outputs of the network can then be computed as $\mathbf{y} = \mathbf{GW}$ for set of weights $\mathbf{W}$. Except that we don't know what the weights are—that is what we set out to compute—and we want to choose them using the target outputs $\mathbf{t}$.

If we were able to get all of the outputs correct, then we could write $\mathbf{t} = \mathbf{GW}$. Now we just need to calculate the matrix inverse of $\mathbf{G}$, to get $\mathbf{W} = \mathbf{G}^{-1}\mathbf{t}$. Unfortunately, there is a little problem here. The matrix inverse is only defined if a matrix is square, and this one probably isn't—there is no reason why the number of hidden nodes should be the same as the number of training inputs. In fact, we hope it isn't, since that would probably be serious overfitting.

Fortunately, there is a well-defined pseudo-inverse $\mathbf{G}^+$ of a matrix, which is $\mathbf{G}^+ = (\mathbf{G}^T\mathbf{G})^{-1}\mathbf{G}^T$. Since the point of the inverse $\mathbf{G}^{-1}$ to a matrix $\mathbf{G}$ is that $\mathbf{G}^{-1}\mathbf{G} = \mathbf{I}$, where $\mathbf{I}$ is the identity matrix, the pseudo-inverse is the matrix that satisfies $\mathbf{G}^+\mathbf{G} = \mathbf{I}$. If $\mathbf{G}$ is a square, non-singular (i.e., with non-zero determinant) matrix then $\mathbf{G}^+ = \mathbf{G}^{-1}$. In NumPy the pseudo-inverse is `np.linalg.pinv()`. This gives us an alternative to the Perceptron network that is even faster, since the training only needs one iteration:

```
self.weights2 = np.dot(np.linalg.pinv(self.hidden),targets)
```

There is one thing that we haven't considered yet, and that is the size of the receptive fields σ for the nodes. We can avoid the problem by giving all of the nodes the same size, and

testing lots of different sizes using a validation set to select one that works. Alternatively, we can select it in advance by arguing that the important thing is that the whole space is covered by the receptive fields of the entire set of basis functions, and so the width of the Gaussians should be set according to the maximum distance between the locations of the hidden nodes (d) and the number of hidden nodes. The most common choice is to pick the width of the Gaussian as $\sigma = d/\sqrt{2M}$, where M is the number of RBFs.

There is another way to deal with the fact that there may be inputs that are outside the receptive fields of all nodes, and that is to use normalised Gaussians, so that there is always at least one input firing: the node that is closest to the current input, even if that is a long way off. It is a modification of Equation (5.1) and it looks like the soft-max function:

$$g(\mathbf{x}, \mathbf{w}, \sigma) = \frac{\exp(-\|\mathbf{x} - \mathbf{w}\|/2\sigma^2)}{\sum_i \exp(-\|\mathbf{x} - \mathbf{w}_i\|/2\sigma^2)}. \tag{5.2}$$

Using the RBF network on the `iris` dataset that was used in Section 4.4.3 with five RBF centres gives similar results to the MLP, with well over 90% classification accuracy.

With the MLP, one question that we failed to find a nice answer to was how to pick the number of hidden nodes, and we were reduced to training lots of networks with different numbers of nodes and using the one that performed best on the validation set. The same problem occurs with the RBF network.

In the RBF network the activations of the hidden nodes is based on the distance between the current input and the weights. There are various measures of distance that we can use, as will be discussed in Section 7.2.3; we generally use the Euclidean distance. These distances can be computed for any number of dimensions, but as the number of dimensions increases, something rather worrying happens, which is that we start needing more RBF nodes to cover the space. The number of input dimensions has a profound effect on learning, something that has a suitably impressive name that we have already seen—the curse of dimensionality (Section 2.1.2). For RBFs, the effect of the curse is that the amount of the space covered by an RBF with a fixed receptive field will decrease, and so we will need many more of them to cover the space.

5.3 INTERPOLATION AND BASIS FUNCTIONS

One of the problems that we looked at in Chapter 1 was that of function approximation: given some data, find a function that goes through the data without overfitting to the noise, so that values between the known datapoints can be inferred or interpolated. The RBF network solves this problem by each of the basis functions making a contribution to the output whenever the input is within its receptive field. So several RBF nodes will probably respond for each input.

We are now going to make the problem a bit simpler. We won't allow the receptive fields to overlap, and we'll space them out so that they just meet up with each other. Obviously, we won't need the Gaussian part that decides how much each one matches now, either. If the datapoint is within the receptive field of this function then we listen only to this function, otherwise we ignore it and listen to some other function. If each function just returns the average value within its patch, then for one-dimensional data we get a histogram output, as is shown in Figure 5.5. We can extend this a bit further so that the lines are not horizontal, but instead reflect the first derivative of the curve at that point, as is shown in Figure 5.6. This is all right, but we might want the output to be continuous, so that the line within the first bin meets up with the line in the second bin at the boundary, so we can add the extra constraint that the lines have to meet up as well. This gives the curve in Figure 5.7.

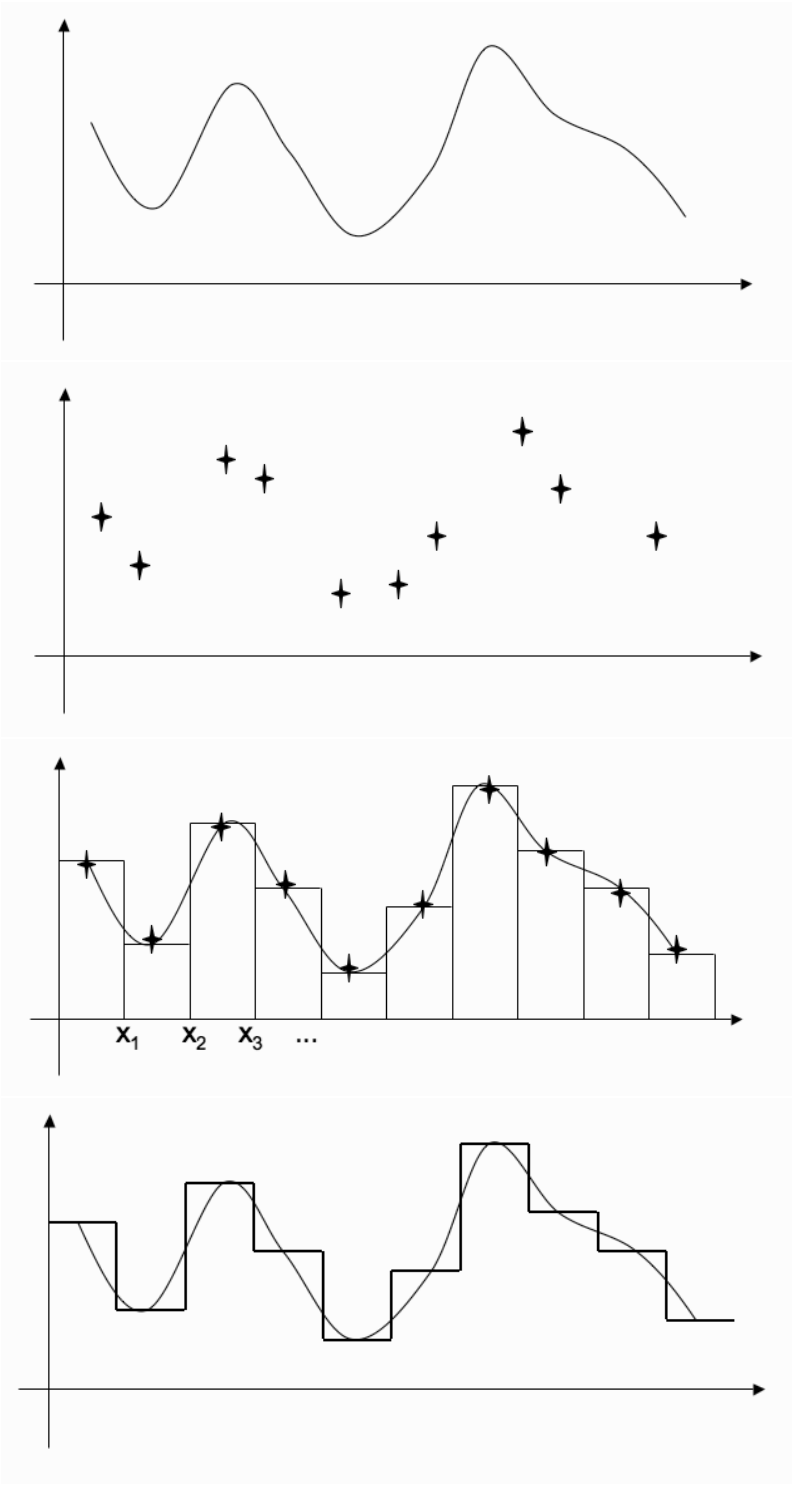

FIGURE 5.5 *Top:* Curve showing a function. *Second:* A set of datapoints from the curve. *Third:* Putting a straight horizontal line through each point creates a histogram that describes an approximation to the curve. *Bottom:* That approximation.

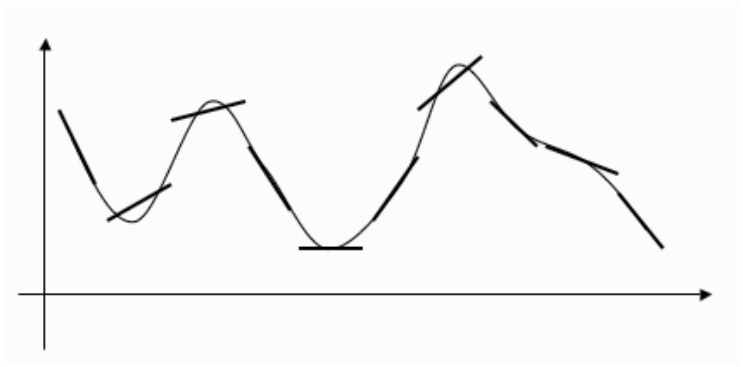

FIGURE 5.6 Representing the points by straight lines that aren't necessarily horizontal (so that their first derivative matches at the point) gives a better approximation.

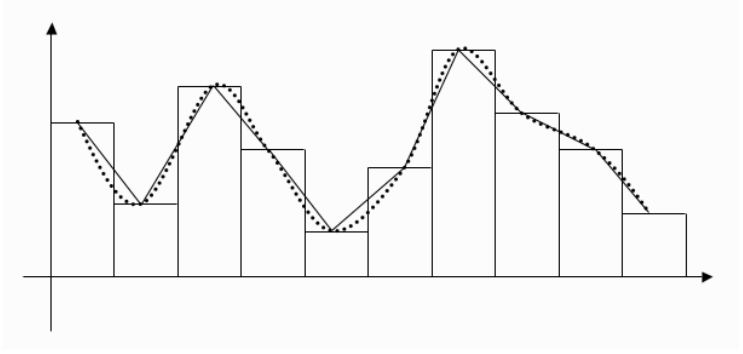

FIGURE 5.7 Making the straight lines meet so that the function is continuous gives a better approximation.

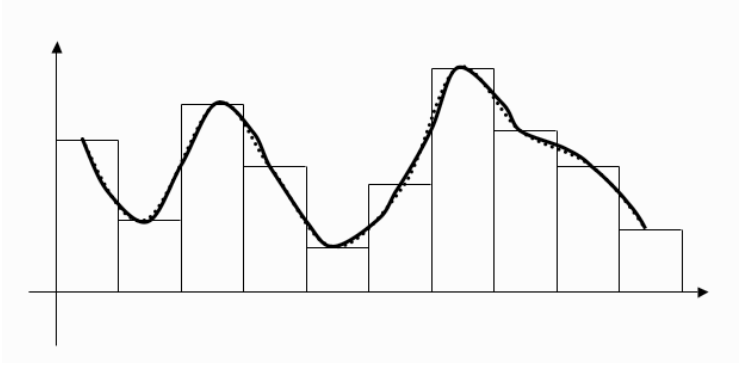

FIGURE 5.8 Using cubic functions to connect the points gives an even better approximation, and the curve is also continuous at the points where the sections join up (known as knotpoints).

Of course, there is no reason why the functions should be linear at all—if we use cubic functions (i.e., polynomials with x^3, x^2, x and constant components) to approximate each piece of data, then we can get results like those shown in Figure 5.8. We can continue to make the functions more complicated, with the important point being how many degrees of continuity we require at the boundaries between the points. These functions are known as splines, and the most common one to use is the cubic spline. To reach the stage where we can understand it, we need to go back and think about some theory.

5.3.1 Bases and Basis Expansion

Radial basis functions and several other machine learning algorithms can be written in this form:

$$f(\mathbf{x}) = \sum_{i=1}^{n} \alpha_i \Phi_i(\mathbf{x}), \tag{5.3}$$

where $\Phi_i(\mathbf{x})$ is some function of the input value $\mathbf{x}$ and the α_i are the parameters we can solve for in order to make the model fit the data. We will consider the input being scalar values x rather than vector values $\mathbf{x}$ in what follows. The $\Phi_i(x)$ are known as basis functions and they are parameters of the model that are chosen. The first thing we need to think about is where each Φ_i is defined. Looking at the third graph of Figure 5.5 we see that the first function should only be defined between 0 and x_1, the next between x_1 and x_2, and so on. These points x_i are called knotpoints and they are generally evenly spaced, but choosing how many of them there should be is not necessarily easy. The more knotpoints there are, the more complex the model can be, in which case the model is more likely to overfit, and needs more training data, just like the neural networks that we have seen.

We can choose the Φ_i in any way we like. Suppose that we simply use a constant function $\Phi(x) = 1$. Now the model would have value α_1 to the left of x_1, value α_2 between x_1 and x_2, etc. So depending upon how we fit the spline model to the data, the model will have different values, but it will certainly be constant in each region. This is sufficient to make the straight line approximation shown at the bottom of Figure 5.5. However, we might decide that a constant value is not enough, and we use a function that varies linearly (a linear function that has value $\Phi(x) = x$ within the region). In this case, we can make Figure 5.6, where each point is represented by a straight line that is not necessarily horizontal. This represents the line close to each point fairly well, but looks messy because the line segments do not meet up.

The question then is how to extend the model to include matching at the knotpoints, where one line segment stops and the next one starts. In fact, this is easy. We just insist that the α_i have to be chosen so that at the knotpoint the value of $f(x_1)$ is the same whether we come from the left of x_1 or the right. These are often written as $f(x_1^-)$ and $f(x_1^+)$. Now we just need to work out which α values are involved in the x_1 knotpoint from each side. There are going to be four of them: two for the constant part, and two for the linear part. The ones connected with the constant are obvious: α_1 and α_2. Now suppose that the linear ones are α_{11} and α_{12} (which would mean that there were 10 regions and therefore 9 knotpoints, since then $\alpha_1 \ldots \alpha_{10}$ correspond to the constant functions for each region). In that case, $f(x_1^-) = \alpha_1 + x_1\alpha_{11}$ and $f(x_1^+) = \alpha_2 + x_1\alpha_{12}$. This is an extra constraint that we will need to include when we solve for the values of the α_i.

There is a simpler way to encode this, which is to add some extra basis functions. As well as $\Phi_1(x) = 1$, $\Phi_2(x) = x$, we add some basis functions that insist that the value is 0 at the boundary with x_1: $\Phi_3(x) = (x - x_1)_+$, and the next with the boundary at

x_2: $\Phi_4(x) = (x - x_2)_+$, etc., where $(x)_+ = x$ if $x > 0$ and 0 otherwise. These functions are sufficient to insist that the knotpoint values are enforced, since one is defined on each knotpoint. This is then enough for us to construct the approximation shown in Figure 5.7.

5.3.2 The Cubic Spline

We can carry on adding extra powers of x, but it turns out that the cubic spline is generally sufficient. This has four basic basis functions ($\Phi_1(x) = 1, \Phi_2(x) = x, \Phi_3(x) = x^2, \Phi_4(x) = x^3$), and then as many extras as there are knotpoints, each of the form $\Phi_{4+i}(x) = (x - x_i)^3_+$. This function constrains the function itself and also its first two derivatives to meet at each knotpoint. Notice that while the Φs are not linear, we are simply adding up a weighted sum of them, and so the model is linear in them. We can then produce curves like Figure 5.8, which represent the data very well.

5.3.3 Fitting the Spline to the Data

Having defined the functions, we need to work out how to choose the α_i in order to make the model fit the data. We will continue to define the sum-of-squares error and to minimise that, which is known in the statistical literature as least-squares fitting, and will be described in more detail in Section 9.2. The important point is that everything is linear in the basis functions, so computing the least-squares fit is a linear problem. As with the MLP, the error that we are trying to minimise is:

$$E(y, f(x)) = \sum_{i=1}^{N} (y_i - f(x_i))^2. \tag{5.4}$$

NumPy already has a method defined for computing linear least-squares optimisation: the function `np.linalg.lstsq()`. As a simple example of how to use it we will make some noisy data from a couple of Gaussians and then fit the model parameters, which are `2.5` and `3.2`. The final line gives the result, which isn't too far from the correct one, and Figure 5.9 shows the results.

```
import numpy as np
import pylab as pl

x = np.arange(-3,10,0.05)
y = 2.5 * np.exp(-(x)**2/9) + 3.2 * np.exp(-(x-0.5)**2/4) + np.random.normal(0.0, 1.0, len(x))
nParam = 2
A = np.zeros((len(x),nParam), dtype=float)
A[:,0] = np.exp(-(x)**2/9)
A[:,1] = np.exp(-(x-0.5)**2/4)
(p, residuals, rank, s) = np.linalg.lstsq(A,y)

pl.plot(x,y,'.')
pl.plot(x,p[0]*A[:,0]+p[1]*A[:,1],'x')

p
```

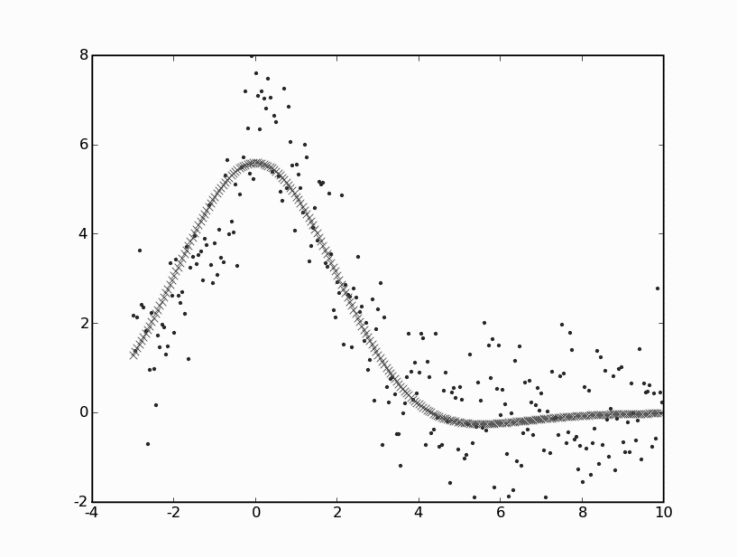

FIGURE 5.9 Using linear least-squares to fit parameters for two Gaussians produces the line from the noisy datapoints plotted as circles.

```
>>> array([ 2.00101406,  3.09626831])
```

5.3.4 Smoothing Splines

The way that we constructed the splines in Section 5.3 was to insist that they went through each knotpoint exactly. This was a good way to describe our constraints, but it is not necessarily realistic: almost all of the data that we ever see will be noisy, and insisting that the data goes through the knotpoints therefore overfits: imagine that the line in Figure 5.9 went through each datapoint. As we try to make the spline model match the data more and more accurately, we will add further knotpoints, which leads to further overfitting. We can deal with this by using regularisation. This is a very important idea in optimisation. In essence, it means adding an extra constraint that makes the problem simpler to solve by providing some way to choose from amongst the set of possible solutions.

The most common regulariser that is used for splines is to make the spline model as 'smooth' as possible, where the smoothness is measured by computing the second derivative of the curve at each point, squaring it so that it is always positive, and integrating it along the curve. In this way, a straight line is perfectly smooth, but probably won't be a good match for the data, so we introduce a parameter λ that describes the tradeoff between the two parts. We regain the interpolating spline of Section 5.3 for $\lambda = 0$, whereas for $\lambda \to \infty$ we get the least-squares straight line. This type of spline is known as a smoothing spline. The cubic smoothing spline is often used. While there are automated methods of choosing λ, it is more normal to use cross-validation to find a value that seems to work well. The form of the optimisation is now:

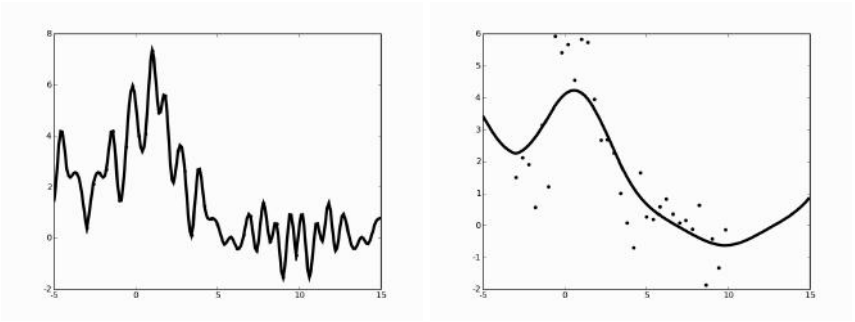

FIGURE 5.10 B-spline fitting of the data shown in Figure 5.9 with *left:* $\lambda = 0$ and *right:* $\lambda = 100$.

$$E(y, f(x), \lambda) = \sum_{i=1}^{N} (y_i - f(x_i))^2 + \lambda \int \left(\frac{d^2 f}{dt^2} \right)^2 dt. \tag{5.5}$$

SciPy already has functions to perform this in Python, the output of two different values of the smoothing parameter are shown in Figure 5.10.

```
import scipy.signal as sig
# Fit spline
spline = sig.cspline1d(y,100)
xbar = np.arange(-5,15,0.1)
# Evaluate spline
ybar = sig.cspline1d_eval(spline, xbar, dx=x[1]-x[0],x0=x[0])
```

5.3.5 Higher Dimensions

Everything that we have done so far is aimed at one spatial dimension and all of our effort has gone into the cubic spline. However, it is not very clear what to do with higher-dimensional data. One common thing that is done is to take a set of independent basis functions in each different coordinate (x, y, and z in 3D) and then to combine them in all possible combinations ($\Phi_{xi}(x)\Phi_{yj}(y)\Phi_{zk}(z)$). This is known as the tensor product basis, and suffers from the curse of dimensionality very quickly, but works well in 2D and 3D, where the B-spline is built up in this way. Figure 5.11 shows a grid of knotpoints and a set of points inbetween that can be interpolated in the x_1 and x_2 directions separately.

However, for the smoothing spline there is another problem: what is the higher-dimensional analogue of the curvature measurement that was computed with the second derivative in Equation (5.5)? In two dimensions, one possibility is to consider the bending energy. This measures how much energy is required to bend a thin plate so that it passes through a set of points without gravity. It leads to a penalty term that consists of:

$$\int \int_{\mathbb{R}^2} \left(\frac{\partial^2 f}{\partial x_1^2} \right)^2 + 2 \left(\frac{\partial^2 f}{\partial x_1 \partial x_2} \right)^2 + \left(\frac{\partial^2 f}{\partial x_2^2} \right)^2 dx_1 dx_2. \tag{5.6}$$

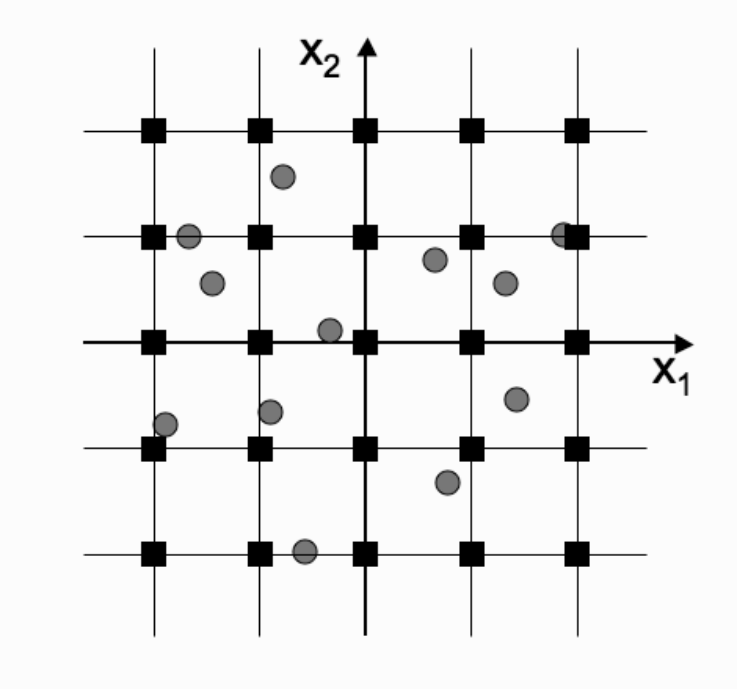

FIGURE 5.11 In 2D the knotpoints (black squares) can be used to interpolate other points (grey circles) in each dimension individually.

Computing the optimal values under this penalty leads to thin-plate splines, which are radial basis functions of the form $f(x, y) = f(r) = r^2 \log |r|$, where r is the radial distance between x and y, which was first published by Duchon in 1978, but popularised by Bookstein, who uses it to look at what he calls morphometrics, which is the study of how shape changes as animals are growing. The fitting is no different, it is just the basis functions that have changed.

5.3.6 Beyond the Bounds

There is an interesting extra feature to consider. We are fitting our spline to the training data in order to predict the values for other datapoints that we do not know target values for. We assume that our training data are representative of the entire training set, but that does not mean that it contains the lowest possible values, nor the highest. The spline model that we have built has constraints to ensure that the pieces of the spline match up continuously at the knotpoints, but we haven't done anything at all regarding thinking about what happens before the first knotpoint, or after the last. For the polynomials that we are using here, this turns out to be a serious problem, which means that guesses outside the boundaries (extrapolations) often turn out to be very inaccurate. Since we don't have any data, it is hard to do much, but one thing that is sometimes done is to insist that outside the boundary knotpoints the function is linear. This is known as the natural spline.

FURTHER READING

The original paper on radial basis function neural networks is:

- J. E. Moody and C. Darken. Fast learning in networks of locally-tuned processing units. *Neural Computation*, 1:281–294, 1989.

For more information on splines, not necessarily from the machine learning viewpoint, try:

- C. de Boor. *A Practical Guide to Splines*. Springer, Berlin, Germany, 1978.
- G. Wahba. *Spline Models for Observational Data*. SIAM, Philadelphia, USA, 1990.
- F. Girosi, M. Jones, and T. Poggio. Regularization theory and neural network architectures. *Neural Computation*, 7:219–269, 1995.
- Chapter 5 and Section 6.7 of T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*, 2nd edition, Springer, Berlin, Germany, 2008.
- Chapter 5 of S. Haykin. *Neural Networks: A Comprehensive Foundation*, 2nd edition, Prentice-Hall, New Jersey, USA, 1999.

The field of morphometrics, studying how shape changes as organisms grow, is a very interesting one. A possible place to start studying this topic would be:

- F.L. Bookstein. *Morphometric Tools for Landmark Data: Geometry and Biology*. Cambridge University Press, Cambridge, UK, 1991.

PRACTICE QUESTIONS

Problem 5.1 Create an RBF network that solves the XOR function.

Problem 5.2 Apply the RBF network to the Pima Indian dataset and the classification of the MNIST letters. Can you identify differences in the results between the RBF and the MLP?

Problem 5.3 The RBF code that is available on the website uses the hybrid approach. You should be able to change the code so that it uses the fixed centres or full gradient descent method, and then you can experiment with them and see which one works better. In particular, you should be able to find examples where the fixed centres one does not work well if the order of the inputs is poorly chosen.

Problem 5.4 The following function creates some noisy data from a sinusoidal function:

```
def gendata(npoints):
    x = np.arange(0,4*np.pi,1./npoints)

    data = x*np.sin(x) + np.random.normal(0,2,np.size(x))
    print data
    pl.plot(x,x*np.sin(x),'k-',x,data,'k.')
    pl.show()
    return x,data
```

Fit a spline to this data using both the interpolating and smoothing versions of the B-spline. Which makes more sense here? Experiment with different values of the smoothing parameter. Can you work out an algorithm that will attempt to set it based on a validation set?

Problem 5.5 Implement the B-spline in 2D by convolving two 1D cubic splines in orthogonal directions. Can you use it to warp images?

CHAPTER 6

Dimensionality Reduction

When looking at data and plotting results, we can never go beyond three dimensions in our data, and usually find two dimensions easier to interpret. In addition, we have already seen the curse of dimensionality (Section 2.1.2) means that the higher the number of dimensions we have, the more training data we need. Further, the dimensionality is an explicit factor for the computational cost of many algorithms. These are some of the reasons why dimensionality reduction is useful. However, it can also remove noise, significantly improve the results of the learning algorithm, make the dataset easier to work with, and make the results easier to understand. In extreme cases such as the Self-Organising Map that we will see in Section 14.3, where the number of dimensions becomes three or fewer, we can also plot the data, which makes it much easier to understand and interpret.

With this many good things to say about dimensionality reduction, clearly it is something that we need to understand. The importance of the field for machine learning and other forms of data analysis can be seen from the fact that in the year 2000 there were three articles related to dimensionality reduction published together in the prestigious journal *Science*. At the end of the chapter we are going to see two of the algorithms that were described in those papers: Locally Linear Embedding and Isomap.

There are three different ways to do dimensionality reduction. The first is feature selection, which typically means looking through the features that are available and seeing whether or not they are actually useful, i.e., correlated to the output variables. While many people use neural networks precisely because they don't want to 'get their hands dirty' and look at the data themselves, as we have already seen, the results will be better if you check for correlations and other simple things before using the neural network or other learning algorithm. The second method is feature derivation, which means deriving new features from the old ones, generally by applying transforms to the dataset that simply change the axes (coordinate system) of the graph by moving and rotating them, which can be written simply as a matrix that we apply to the data. The reason this performs dimensionality reduction is that it enables us to combine features, and to identify which are useful and which are not. The third method is simply to use clustering in order to group together similar datapoints, and to see whether this allows fewer features to be used.

To see how choosing the right features can make a problem significantly simpler, have a look at the table on the left of Figure 6.1. It shows the x and y coordinates of 4 points. Looking at the numbers it is hard to see any correlation between the points, and even when they are plotted it simply looks like they might form corners of a rotated rectangle. However, the plot on the right of the figure shows that they are simply a set of four points from a circle, (in fact, the points at $(\pi/6, 4\pi/6, 7\pi/6, 11\pi/6)$) and using this one coordinate, the angle, makes the data a lot easier to understand and analyse.

x	y
2.00	-1.43
2.37	-2.80
1.00	-3.17
0.63	-1.80

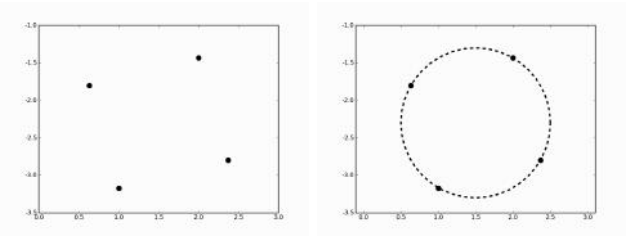

FIGURE 6.1 Three views of the same four points. *Left:* As numbers, where the links are unclear. *Centre:* As four plotted points. *Right:* As four points that lie on a circle.

Once we have worked out how to represent the data, we can suppress dimensions that aren't useful to the algorithm. Even before we get into any form of analysis at all, we can try to perform feature selection, looking at the possible inputs that we have for the problem, and deciding which are useful. Many of the methods that we will see in this chapter merge this idea with transformations of the data, so that combinations of the different inputs, rather than the inputs themselves, are used. However, even before using any of the algorithms identified here, input features can be ignored if they do not seem to be useful.

We will see another method of doing feature selection later, since it is inherent to the way that the decision tree (Chapter 12) works: at each stage of the algorithm it decides which feature to add next. This is the constructive way to decide on the features: start with none, and then iteratively add more, testing the error at each stage to see whether or not it changed at all when that feature was added. The destructive method is to prune the decision tree, lopping off branches and checking whether the error changed at all.

In general, selecting the features is a search problem. We take the best system so far, and then search over the set of possible next features to add. This can be computationally very expensive, since for d features there are $2^d - 1$ possible sets of features to search over, from any individual feature up to the full set. In general, greedy methods (Section 9.4) are employed, although backtracking can also be employed to check whether the search gets stuck.

Many of the algorithms that we will see in this chapter are unsupervised. The disadvantage of this is that we are not then able to use the knowledge of their classes in order to reduce the problem further. However, we will start off by considering a method of dimensionality reduction that is aimed at supervised learning, Linear Discriminant Analysis. This method is credited to one of the best-known statisticians of the 20th century, R.A. Fisher, and dates from 1936.

6.1 LINEAR DISCRIMINANT ANALYSIS (LDA)

Figure 6.2 shows a simple two-dimensional dataset consisting of two classes. We can compute various statistics about the data, but we will settle for the means of the two classes in the data, $\boldsymbol{\mu}_1$ and $\boldsymbol{\mu}_2$, the mean of the entire dataset ($\boldsymbol{\mu}$), and the covariance of each class with itself (see Section 2.4.2 for a description of covariance), which is $\sum_j (\mathbf{x}_j - \boldsymbol{\mu})(\mathbf{x}_j - \boldsymbol{\mu})^T$. The question is what we can do with these pieces of data. The principal insight of LDA is that the covariance matrix can tell us about the scatter within a dataset, which is the amount of spread that there is within the data. The way to find this scatter is to multiply the covariance by the p_c, the probability of the class (that is, the number of datapoints there are in that class divided by the total number). Adding the values of this for all of the classes gives us a measure of the within-class scatter of the dataset:

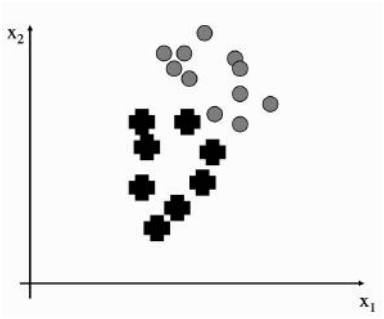

FIGURE 6.2 A set of datapoints in two dimensions, with two classes.

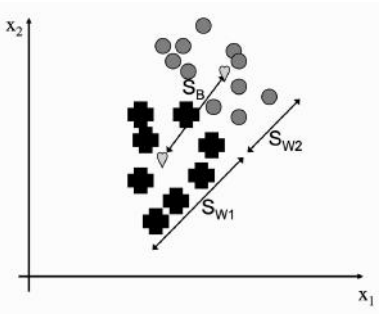

FIGURE 6.3 The meaning of the between-class and within-class scatter. The hearts mark the means of the two classes.

$$S_W = \sum_{\text{classes } c} \sum_{j \in c} p_c(\mathbf{x}_j - \boldsymbol{\mu}_c)(\mathbf{x}_j - \boldsymbol{\mu}_c)^T. \tag{6.1}$$

If our dataset is easy to separate into classes, then this within-class scatter should be small, so that each class is tightly clustered together. However, to be able to separate the data, we also want the distance *between* the classes to be large. This is known as the between-classes scatter and is a significantly simpler computation, simply looking at the difference in the means:

$$S_B = \sum_{\text{classes } c} (\boldsymbol{\mu}_c - \boldsymbol{\mu})(\boldsymbol{\mu}_c - \boldsymbol{\mu})^T. \tag{6.2}$$

The meanings of these two measurements is shown in Figure 6.3. The argument about good separation suggests that datasets that are easy to separate into the different classes (i.e., the classes are discriminable) should have S_B/S_W as large as possible. This seems perfectly reasonable, but it hasn't told us anything about dimensionality reduction. However, we can say that the rule about making S_B/S_W as large as possible is something that we want to be true for our data when we reduce the number of dimensions. Figure 6.4 shows two projections of the dataset onto a straight line. For the projection on the left it is clear that we can't separate out the two classes, while for the one on the right we can. So we just need to find a way to compute a suitable projection.

Remember from Chapter 3 that any line can be written as a vector $\mathbf{w}$ (which we used as our weight vector in Section 3.4; it is one row of weight matrix $\mathbf{W}$). The projection of the data can be written as $z = \mathbf{w}^T \cdot \mathbf{x}$ for datapoint $\mathbf{x}$. This gives us a scalar that is the distance along the $\mathbf{w}$ vector that we need to go to find the projection of point $\mathbf{x}$. To see this, remember that $\mathbf{w}^T \cdot \mathbf{x}$ is the sum of the vectors multiplied together element-wise, and is equal to the size of $\mathbf{w}$ times the size of $\mathbf{x}$ times the cosine of the angle between them. We can make the size of $\mathbf{w}$ be 1, so that we don't have to worry about it, and all that is then described is the amount of $\mathbf{x}$ that lies along $\mathbf{w}$.

So we can compute the projection of our data along $\mathbf{w}$ for every point, and we will have projected our data onto a straight line, as is shown in the two examples in Figure 6.4. Since the mean can be treated as a datapoint, we can project that as well: $\mu'_c = \mathbf{w}^T \cdot \boldsymbol{\mu}_c$. Now we just need to work out what happens to the within-class and between-class scatters.

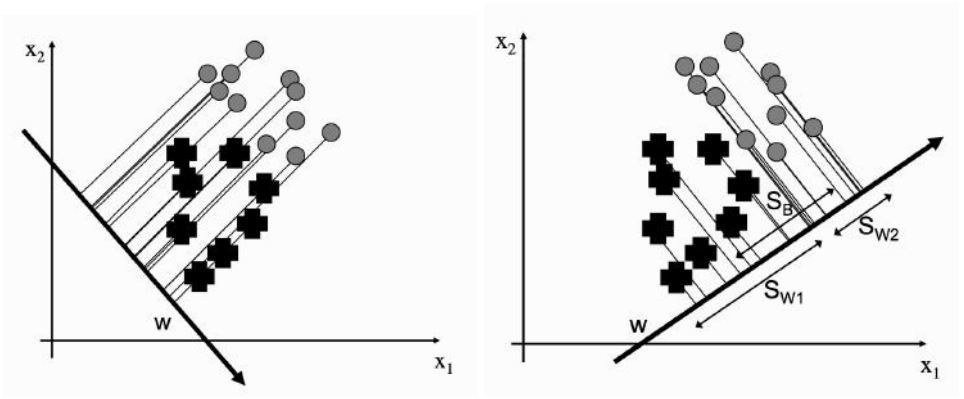

FIGURE 6.4 Two different possible projection lines. The one on the left fails to separate the classes.

Replacing $\mathbf{x}_j$ with $\mathbf{w}^T \cdot \mathbf{x}_j$ in Equations (6.1) and (6.2) we can use some linear algebra (principally the fact that $(\mathbf{A}^T\mathbf{B})^T = \mathbf{B}^T\mathbf{A}^{T^T} = \mathbf{B}^T\mathbf{A}$) to get:

$$\sum_{\text{classes } c} \sum_{j \in c} p_c(\mathbf{w}^T \cdot (\mathbf{x}_j - \boldsymbol{\mu}_c))(\mathbf{w}^T \cdot (\mathbf{x}_j - \boldsymbol{\mu}_c))^T = \mathbf{w}^T S_W \mathbf{w} \tag{6.3}$$

$$\sum_{\text{classes } c} \mathbf{w}^T(\boldsymbol{\mu}_c - \boldsymbol{\mu})(\boldsymbol{\mu}_c - \boldsymbol{\mu})^T\mathbf{w} = \mathbf{w}^T S_B \mathbf{w}. \tag{6.4}$$

So our ratio of within-class and between-class scatter looks like $\frac{\mathbf{w}^T S_W \mathbf{w}}{\mathbf{w}^T S_B \mathbf{w}}$.

In order to find the maximum value of this with respect to $\mathbf{w}$, we differentiate it and set the derivative equal to 0. This tells us that:

$$\frac{S_B\mathbf{w}(\mathbf{w}^T S_W \mathbf{w}) - S_W\mathbf{w}(\mathbf{w}^T S_B \mathbf{w})}{(\mathbf{w}^T S_W \mathbf{w})^2} = 0. \tag{6.5}$$

So we just need to solve this equation for $\mathbf{w}$ and we are done. We start with a little bit of rearranging to get:

$$S_W\mathbf{w} = \frac{\mathbf{w}^T S_W \mathbf{w}}{\mathbf{w}^T S_B \mathbf{w}} S_B\mathbf{w}. \tag{6.6}$$

If there are only two classes in the data, then we can rewrite Equation (6.2) as $S_B = (\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2)(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2)^T$. To see this, consider that there are N_1 examples of class 1 and N_2 examples of class 2. Then substitute $(N_1 + N_2)\mu = N_1\mu_1 + N_2\mu_2)$ into Equation (6.2). The rewritten $S_B\mathbf{w}$ is in the direction $(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2)$, and so $\mathbf{w}$ is in the direction of $S_W^{-1}(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2)$, as can be seen by recalling that the scalar product is independent of order, and so after substituting in the new expression for S_B, the order of the bracketed terms can be changed. Note that we can ignore the ratio of within-class and between-class scatter, since it is a scalar and therefore does not affect the direction of the vector.

Unfortunately, this does not work for the general case. There, finding the minimum is not simple, and requires computing the generalised eigenvectors of $S_W^{-1}S_B$, assuming that S_W^{-1} exists. We will be discussing eigenvectors in the next section if you are not sure what they are.

Turning this into an algorithm is very simple. You simply have to compute the between-class and within-class scatters, and then find the value of **w**. In NumPy, the entire algorithm can be written as (where the generalised eigenvectors are computed in SciPy rather than NumPy, which was imported using `from scipy import linalg as la`):

```
C = np.cov(np.transpose(data))

# Loop over classes
classes = np.unique(labels)
for i in range(len(classes)):
    # Find relevant datapoints
    indices = np.squeeze(np.where(labels==classes[i]))
    d = np.squeeze(data[indices,:])
    classcov = np.cov(np.transpose(d))
    Sw += np.float(np.shape(indices)[0])/nData * classcov

Sb = C - Sw
# Now solve for W and compute mapped data
# Compute eigenvalues, eigenvectors and sort into order
evals,evecs = la.eig(Sw,Sb)
indices = np.argsort(evals)
indices = indices[::-1]
evecs = evecs[:,indices]
evals = evals[indices]
w = evecs[:,:redDim]
newData = np.dot(data,w)
```

As an example of using the algorithm, Figure 6.5 shows a plot of the first two dimensions of the iris data (with the classes shown as three different symbols) before and after applying LDA, with the number of dimensions being set to two. While one of the classes (the circles) can already be separated from the others, all three are readily distinguishable after LDA has been applied (and only one dimension, the y one, is required for this).

6.2 PRINCIPAL COMPONENTS ANALYSIS (PCA)

The next few methods that we are going to look at are also involved in computing transformations of the data in order to identify a lower-dimensional set of axes. However, unlike LDA, they are designed for unlabelled data. This does not stop them being used for labelled data, since the learning that takes place in the lower dimensional space can still use the target data, although it does mean that they miss out on any information that is contained in the targets. The idea is that by finding particular sets of coordinate axes, it will become clear that some of the dimensions are not required. This is demonstrated in Figure 6.6, which shows two versions of the same dataset. In the first the data are arranged in an ellipse that runs at 45° to the axes, while in the second the axes have been moved so that the data now runs along the $x-$axis and is centred on the origin. The potential for dimensionality reduction is in the fact that the y dimension does not now demonstrate much variability, and so it might be possible to ignore it and use the x axis values alone

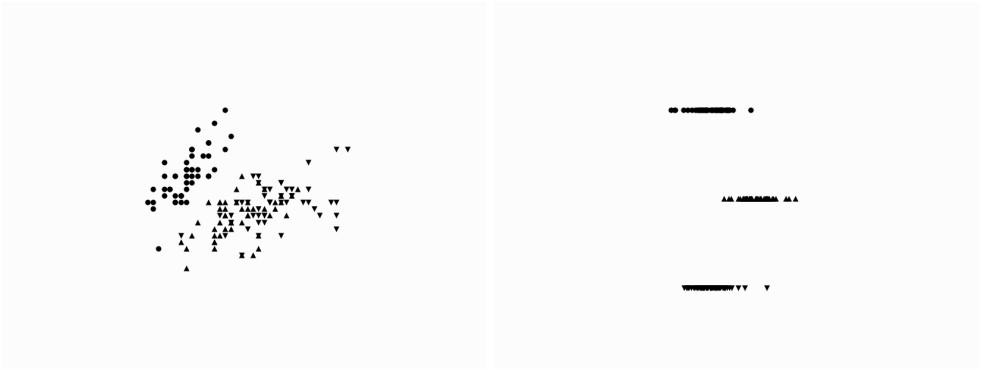

FIGURE 6.5 Plot of the iris data showing the three classes *left:* before and *right:* after LDA has been applied.

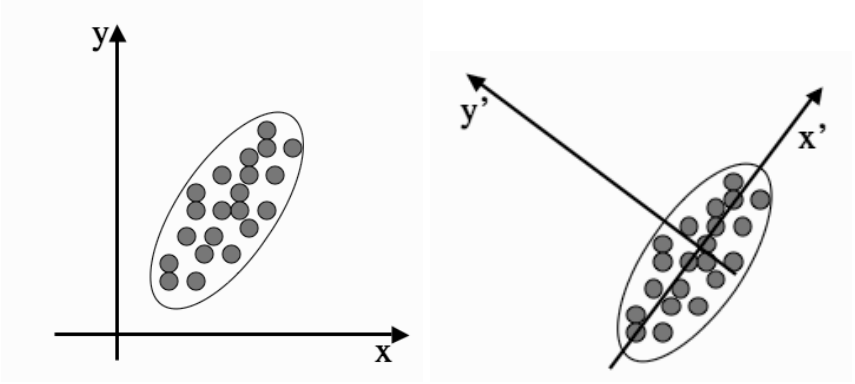

FIGURE 6.6 Two different sets of coordinate axes. The second consists of a rotation and translation of the first and was found using Principal Components Analysis.

without compromising the results of a learning algorithm. In fact, it can make the results better, since we are often removing some of the noise in the data.

The question is how to choose the axes. The first method we are going to look at is Principal Components Analysis (PCA). The idea of a principal component is that it is a direction in the data with the largest variation. The algorithm first centres the data by subtracting off the mean, and then chooses the direction with the largest variation and places an axis in that direction, and then looks at the variation that remains and finds another axis that is orthogonal to the first and covers as much of the remaining variation as possible. It then iterates this until it has run out of possible axes. The end result is that all the variation is along the axes of the coordinate set, and so the covariance matrix is diagonal—each new variable is uncorrelated with every variable except itself. Some of the axes that are found last have very little variation, and so they can be removed without affecting the variability in the data.

Putting this in more formal terms, we have a data matrix $\mathbf{X}$ and we want to rotate it so that the data lies along the directions of maximum variation. This means that we multiply our data matrix by a rotation matrix (often written as $\mathbf{P}^T$) so that $\mathbf{Y} = \mathbf{P}^T\mathbf{X}$, where $\mathbf{P}$ is chosen so that the covariance matrix of $\mathbf{Y}$ is diagonal, i.e.,

$$\text{cov}(\mathbf{Y}) = \text{cov}(\mathbf{P}^T\mathbf{X}) = \begin{pmatrix} \lambda_1 & 0 & 0 & \dots & 0 \\ 0 & \lambda_2 & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & \dots & \lambda_N \end{pmatrix}. \tag{6.7}$$

We can get a different handle on this by using some linear algebra and the definition of covariance to see that:

$$\begin{aligned} \text{cov}(\mathbf{Y}) &= E[\mathbf{Y}\mathbf{Y}^T] && (6.8)\\ &= E[(\mathbf{P}^T\mathbf{X})(\mathbf{P}^T\mathbf{X})^T] && (6.9)\\ &= E[(\mathbf{P}^T\mathbf{X})(\mathbf{X}^T\mathbf{P})] && (6.10)\\ &= \mathbf{P}^T E(\mathbf{X}\mathbf{X}^T)\mathbf{P} && (6.11)\\ &= \mathbf{P}^T\text{cov}(\mathbf{X})\mathbf{P}. && (6.12) \end{aligned}$$

The two extra things that we needed to know were that $(\mathbf{P}^T\mathbf{X})^T = \mathbf{X}^T\mathbf{P}^{T^T} = \mathbf{X}^T\mathbf{P}$ and that $E[\mathbf{P}] = \mathbf{P}$ (and obviously the same for $\mathbf{P}^T$) since it is not a data-dependent matrix. This then tells us that:

$$\mathbf{P}\text{cov}(\mathbf{Y}) = \mathbf{P}\mathbf{P}^T\text{cov}(\mathbf{X})\mathbf{P} = \text{cov}(\mathbf{X})\mathbf{P}, \tag{6.13}$$

where there is one tricky fact, namely that for a rotation matrix $\mathbf{P}^T = \mathbf{P}^{-1}$. This just says that to invert a rotation we rotate in the opposite direction by the same amount that we rotated forwards.

As $\text{cov}(\mathbf{Y})$ is diagonal, if we write $\mathbf{P}$ as a set of column vectors $\mathbf{P} = [\mathbf{p}_1, \mathbf{p}_2, \dots, \mathbf{p}_N]$ then:

$$\mathbf{P}\text{cov}(\mathbf{Y}) = [\lambda_1\mathbf{p}_1, \lambda_2\mathbf{p}_2, \dots, \lambda_N\mathbf{p}_N], \tag{6.14}$$

which (by writing the λ variables in a matrix as $\boldsymbol{\lambda} = (\lambda_1, \lambda_2, \dots, \lambda_N)^T$ and $\mathbf{Z} = \text{cov}(\mathbf{X})$) leads to a very interesting equation:

$$\boldsymbol{\lambda}\mathbf{p}_i = \mathbf{Z}\mathbf{p}_i \text{ for each } \mathbf{p}_i. \tag{6.15}$$

At first sight it doesn't look very interesting, but the important thing is to realise that $\boldsymbol{\lambda}$ is a column vector, while $\mathbf{Z}$ is a full matrix, and it can be applied to each of the $\mathbf{p}_i$ vectors that make up $\mathbf{P}$. Since $\boldsymbol{\lambda}$ is only a column vector, all it does is rescale the $\mathbf{p}_i$s; it cannot rotate it or do anything complicated like that. So this tells us that somehow we have found a matrix $\mathbf{P}$ so that for the directions that $\mathbf{P}$ is written in, the matrix $\mathbf{Z}$ does not twist or rotate those directions, but just rescales them. These directions are special enough that they have a name: they are eigenvectors, and the amount that they rescale the axes (the $\boldsymbol{\lambda}$s) by are known as eigenvalues.

All eigenvectors of a square symmetric matrix $\mathbf{A}$ are orthogonal to each other. This tells us that the eigenvectors define a space. If we make a matrix $\mathbf{E}$ that contains the (normalised) eigenvectors of a matrix $\mathbf{A}$ as columns, then this matrix will take any vector and rotate it into what is known as the eigenspace. Since $\mathbf{E}$ is a rotation matrix, $\mathbf{E}^{-1} = \mathbf{E}^T$, so that rotating the resultant vector back out of the eigenspace requires multiplying it by $\mathbf{E}^T$, where by 'normalised', I mean that the eigenvectors are made unit length. So what should we do between rotating the vector into the eigenspace, and rotating it back out? The answer is that we can stretch the vectors along the axes. This is done by multiplying the vector by a

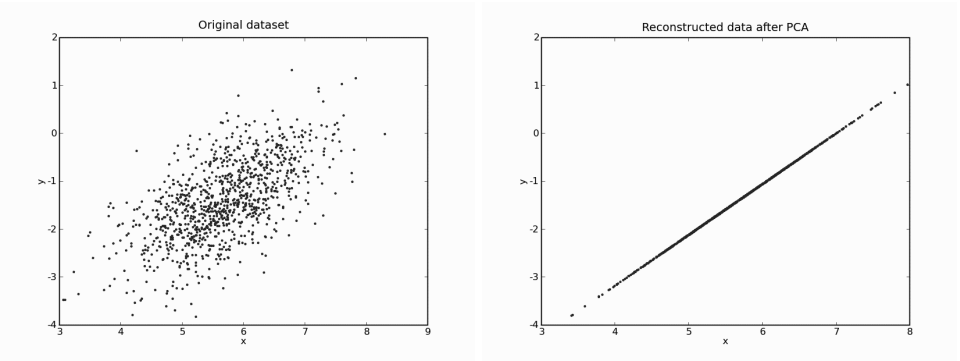

FIGURE 6.7 Computing the principal components of the 2D dataset on the left and using only the first one to reconstruct it produces the line of data shown on the right, which is along the principal axis of the ellipse that the data was sampled from.

diagonal matrix that has the eigenvalues along its diagonal, $\mathbf{D}$. So we can decompose any square symmetric matrix $\mathbf{A}$ into the following set of matrices: $\mathbf{A} = \mathbf{EDE}^T$, and this is what we have done to our covariance matrix above. This is called the spectral decomposition.

Before we get on to the algorithm, there is one other useful thing to note. The eigenvalues tell us how much stretching we need to do along their corresponding eigenvector dimensions. The more of this rescaling is needed, the larger the variation along that dimension (since if the data was already spread out equally then the eigenvalue would be close to 1), and so the dimensions with large eigenvalues have lots of variation and are therefore useful dimensions, while for those with small eigenvalues, all the datapoints are very tightly bunched together, and there is not much variation in that direction. This means that we can throw away dimensions where the eigenvalues are very small (usually smaller than some chosen parameter).

It is time to see the algorithm that we need.

The Principal Components Analysis Algorithm

- Write N datapoints $\mathbf{x}_i = (\mathbf{x}_{1i}, \mathbf{x}_{2i}, \ldots, \mathbf{x}_{Mi})$ as row vectors
- Put these vectors into a matrix $\mathbf{X}$ (which will have size $N \times M$)
- Centre the data by subtracting off the mean of each column, putting it into matrix $\mathbf{B}$
- Compute the covariance matrix $\mathbf{C} = \frac{1}{N}\mathbf{B}^T\mathbf{B}$
- Compute the eigenvalues and eigenvectors of $\mathbf{C}$, so $\mathbf{V}^{-1}\mathbf{CV} = \mathbf{D}$, where $\mathbf{V}$ holds the eigenvectors of $\mathbf{C}$ and $\mathbf{D}$ is the $M \times M$ diagonal eigenvalue matrix
- Sort the columns of $\mathbf{D}$ into order of decreasing eigenvalues, and apply the same order to the columns of V
- Reject those with eigenvalue less than some η, leaving L dimensions in the data

NumPy can compute the eigenvalues and eigenvectors for us. They are both returned in `evals,evecs = np.linalg.eig(x)`. This makes the entire algorithm fairly easy to implement:

```
def pca(data,nRedDim=0,normalise=1):

    # Centre data
    m = np.mean(data,axis=0)
    data -= m

    # Covariance matrix
    C = np.cov(np.transpose(data))

    # Compute eigenvalues and sort into descending order
    evals,evecs = np.linalg.eig(C)
    indices = np.argsort(evals)
    indices = indices[::-1]
    evecs = evecs[:,indices]
    evals = evals[indices]

    if nRedDim>0:
        evecs = evecs[:,:nRedDim]

    if normalise:
        for i in range(np.shape(evecs)[1]):
            evecs[:,i] / np.linalg.norm(evecs[:,i]) * np.sqrt(evals[i])

    # Produce the new data matrix
    x = np.dot(np.transpose(evecs),np.transpose(data))
    # Compute the original data again
    y=np.transpose(np.dot(evecs,x))+m
    return x,y,evals,evecs
```

Two different examples of using PCA are shown in Figures 6.7 and 6.8. The former shows two-dimensional data from an ellipse being mapped into one principal component, which lies along the principal axis of the ellipse. Figure 6.8 shows the first two dimensions of the iris data, and shows that the three classes are clearly distinguishable after PCA has been applied.

6.2.1 Relation with the Multi-layer Perceptron

We will see (in Section 14.3.2) that PCA can be used in the SOM algorithm to initialise the weights, thus reducing the amount of learning that is required, and that it is very useful for dimensionality reduction. However, there is another reason why people who are interested in neural networks are interested in PCA. We already mentioned it when we talked about the auto-associative MLP in Section 4.4.5. The auto-associative MLP actually computes something very similar to the principal components of the data in the hidden nodes, and this is one of the ways that we can understand what the network is doing. Of course, computing the principal components with a neural network isn't necessarily a good idea. PCA is linear (it just rotates and translates the axes, it can't do anything more complicated). This is clear if we think about the network, since it is the hidden nodes that

FIGURE 6.8 Plot of the first two principal components of the iris data, showing that the three classes are clearly distinguishable.

are computing PCA, and they are effectively a bit like a Perceptron—they can only perform linear tasks. It is the extra layers of neurons that allow us to do more.

So suppose we do just that and use a more complicated MLP network with four layers of neurons instead of two. We still use it as an auto-associator, so that the targets are the same as the inputs. What will the middle hidden layer look like then? A full answer is complicated, but we can speculate that the first layer is computing some non-linear transformation of the data, while the second (bottleneck) layer is computing the PCA of those non-linear functions. Then the third layer reconstructs the data, which appears again in the fourth layer. So the network is still doing PCA, just on a non-linear version of the inputs. This might be useful, since now we are not assuming that the data are linearly separable. However, to understand it better we will look at it from a different viewpoint, thinking of the actions of the first layer as kernels, which are described in Section 8.2.

6.2.2 Kernel PCA

One problem with PCA is that it assumes that the directions of variation are all straight lines. This is often not true. We can use the auto-associator with multiple hidden layers as just discussed, but there is a very nice extension to PCA that uses the kernel trick (which is described in Section 8.2) to get around this problem, just as the SVM got around it for the Perceptron. Just as is done there, we apply a (possibly non-linear) function $\Phi(\cdot)$ to each datapoint $\mathbf{x}$ that transforms the data into the kernel space, and then perform normal linear PCA in that space. The covariance matrix is defined in the kernel space and is:

$$\mathbf{C} = \frac{1}{N}\sum_{n=1}^{N}\Phi(\mathbf{x}_n)\Phi(\mathbf{x}_n)^T, \tag{6.16}$$

which produces the eigenvector equation:

$$\lambda\left(\Phi(\mathbf{x}_i)\mathbf{V}\right) = \left(\Phi(\mathbf{x}_i)\mathbf{C}\mathbf{V}\right) \quad i = 1\ldots N, \tag{6.17}$$

where $\mathbf{V} = \sum_{j=1}^{N}\boldsymbol{\alpha}_j\Phi(\mathbf{x}_j)$ are the eigenvectors of the original problem and the $\boldsymbol{\alpha}_j$ will turn out to be the eigenvectors of the 'kernelized' problem. It is at this point that we can apply the kernel trick and produce an $N \times N$ matrix $\mathbf{K}$, where:

$$\mathbf{K}_{(i,j)} = (\Phi(\mathbf{x}_i) \cdot \Phi(\mathbf{x}_j)) . \tag{6.18}$$

Putting these together we get the equation $N\lambda\mathbf{K}\boldsymbol{\alpha} = \mathbf{K}^2\boldsymbol{\alpha}$, and we left-multiply by $\mathbf{K}^{-1}$ to reduce it to $N\lambda\boldsymbol{\alpha} = \mathbf{K}\boldsymbol{\alpha}$. Computing the projection of a new point $\mathbf{x}$ into the kernel PCA space requires:

$$\left(\mathbf{V}^k \cdot \Phi(\mathbf{x})\right) = \sum_{i=1}^{N} \boldsymbol{\alpha}_i^k \left(\Phi(\mathbf{x}_i) \cdot \Phi(\mathbf{x}_j)\right) . \tag{6.19}$$

This is all there is to the algorithm.

The Kernel PCA Algorithm

- Choose a kernel and apply it to all pairs of points to get matrix $\mathbf{K}$ of distances between the pairs of points in the transformed space
- Compute the eigenvalues and eigenvectors of $\mathbf{K}$
- Normalise the eigenvectors by the square root of the eigenvalues
- Retain the eigenvectors corresponding to the largest eigenvalues

The only tricky part of the implementation is in the diagonalisation of $\mathbf{K}$, which is generally done using some well-known linear algebra identities, leading to:

```
K = kernelmatrix(data,kernel)

# Compute the transformed data
D = np.sum(K,axis=0)/nData
E = np.sum(D)/nData
J = np.ones((nData,1))*D
K = K - J - np.transpose(J) + E*np.ones((nData,nData))

# Perform the dimensionality reduction
evals,evecs = np.linalg.eig(K)
indices = np.argsort(evals)
indices = indices[::-1]
evecs = evecs[:,indices[:redDim]]
evals = evals[indices[:redDim]]

sqrtE = np.zeros((len(evals),len(evals)))
for i in range(len(evals)):
    sqrtE[i,i] = np.sqrt(evals[i])

newData = np.transpose(np.dot(sqrtE,np.transpose(evecs)))
```

This is a computationally expensive algorithm, since it requires computing the kernel

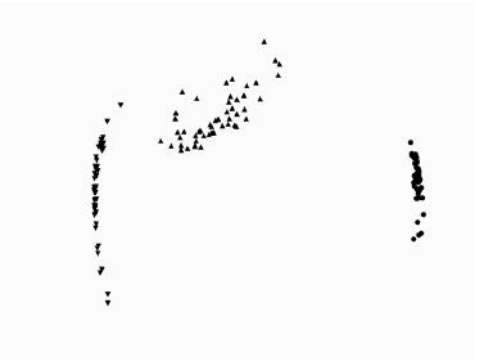

FIGURE 6.9 Plot of the first two non-linear principal components of the iris data, (using the Gaussian kernel) showing that the three classes are clearly distinguishable.

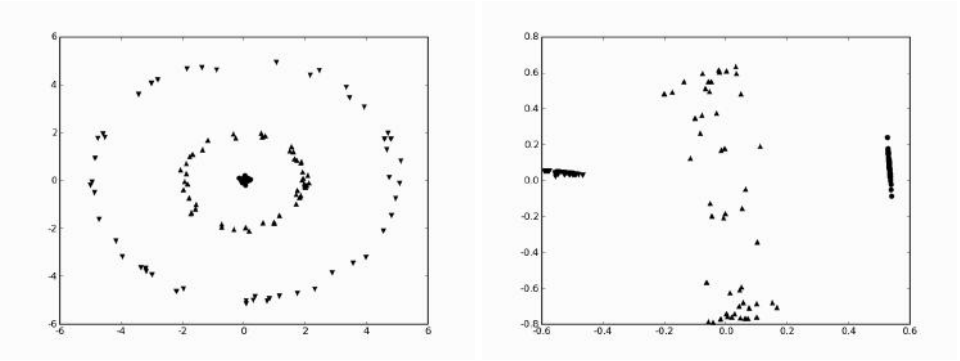

FIGURE 6.10 A very definitely non-linear dataset consisting of three concentric circles, and the (Gaussian) kernel PCA mapping of the iris data, which requires only one component to separate the data.

matrix and then the eigenvalues and eigenvectors of that matrix. The naïve implementation on the algorithm on the website is $\mathcal{O}(n^3)$, but with care it is possible to take advantage of the fact that not all of the eigenvalues are needed, which can lead to an $\mathcal{O}(n^2)$ algorithm.

Figure 6.9 shows the output of kernel PCA when applied to the iris dataset. The fact that it can separate this data well is not very surprising since the linear methods that we have already seen can do it, but it is a useful check of the method. A rather more difficult example is shown in Figure 6.10. Data are sampled from three concentric circles. Clearly, linear PCA would not be able to separate this data, but applying kernel PCA to this example separates the data using only one component.

6.3 FACTOR ANALYSIS

The idea of factor analysis is to ask whether the data that is observed can be explained by a smaller number of uncorrelated factors or latent variables. The assumption is that the data comes from some underlying data source (or set of data sources) that are not directly known. The problem of factor analysis is to find those independent factors, and the noise that is inherent in the measurements of each factor. Factor analysis is commonly used in psychology and other social sciences, and the factors are generally chosen to have some particular meanings: in psychology, they can be related to IQ and other tests.

Suppose that we have a dataset in the usual $N \times M$ matrix $\mathbf{X}$, i.e., each row of $\mathbf{X}$ is an M-dimensional datapoint, and $\mathbf{X}$ has covariance matrix $\mathbf{\Sigma}$. As, with PCA, we centre the data by subtracting off the mean of each variable (i.e., each column): $\mathbf{b}_j = \mathbf{x}_j - \boldsymbol{\mu}_j$, $j = 1 \dots M$, so that the mean $E[\mathbf{b}_i] = 0$. Which we've done before, for example for the MLP and many times since.

We can write the model that we are assuming as:

$$\mathbf{X} = \mathbf{WY} + \boldsymbol{\epsilon}, \tag{6.20}$$

where $\mathbf{X}$ are the observations and $\boldsymbol{\epsilon}$ is the noise. Since the factors $\mathbf{b}_i$ that we want to find should be independent, so $\text{cov}(\mathbf{b}_i, \mathbf{b}_j) = 0$ if $i \neq j$. Factor analysis takes explicit notice of the noise in the data, using the variable $\boldsymbol{\epsilon}$. In fact, it assumes that the noise is Gaussian with zero mean and some known variance: Ψ, with the variance of each element being $\Psi_i = \text{var}(\epsilon_i)$. It also assumes that these noise measurements are independent of each other, which is equivalent to the assumption that the data come from a set of separate (independent) physical processes, and seems reasonable if we don't know otherwise.

The covariance matrix of the original data, $\mathbf{\Sigma}$, can now be broken down into $\text{cov}(\mathbf{Wb} + \boldsymbol{\epsilon}) = \mathbf{WW}^T + \Psi$, where Ψ is the matrix of noise variances and we have used the fact that $\text{cov}(\mathbf{b}) = \mathbf{I}$ since the factors are uncorrelated.

With all of that set up, the aim of factor analysis is to try to find a set of factor loadings $\mathbf{W}_{ij}$ and values for the variance of the noise parameters Ψ, so that the data in $\mathbf{X}$ can be reconstructed from the parameters, or so that we can perform dimensionality reduction.

Since we are looking at adding additional variables, the natural formulation is an EM algorithm (as described in Section 7.1.1) and this is how the computations are usually performed to produce the maximum likelihood estimate. Getting to the EM algorithm takes some effort. We first define the log likelihood (where $\boldsymbol{\theta}$ is the data we are trying to fit) as:

$$Q(\boldsymbol{\theta}_t | \boldsymbol{\theta}_{t-1}) = \int p(\mathbf{x}|\mathbf{y}, \boldsymbol{\theta}_{t-1}) \log(p(\mathbf{y}|\mathbf{x}, \boldsymbol{\theta}_t) p(\mathbf{x})) d\mathbf{x}. \tag{6.21}$$

We can replace several of the terms in here with values, and we can also ignore any terms that do not depend on $\boldsymbol{\theta}$. The end result of this is a new version of Q, which forms the basis of the E-step:

$$Q(\boldsymbol{\theta}_t | \boldsymbol{\theta}_{t-1}) = \frac{1}{2} \int p(\mathbf{x}|\mathbf{y}, \boldsymbol{\theta}_{t-1}) \log(\det(\Psi^{-1})) - \left(\mathbf{y} - W\mathbf{x}\right)^T \Psi^{-1} (\mathbf{y} - W\mathbf{x}) \Big) \, d\mathbf{x}. \tag{6.22}$$

For the EM algorithm we now have to differentiate this with respect to $\mathbf{W}$ and the individual elements of Ψ, and apply some linear algebra, to get update rules:

$$\mathbf{W}_{new} = \left(\mathbf{y}E(\mathbf{x}|\mathbf{y})^T\right)\left(E(\mathbf{x}\mathbf{x}^T|\mathbf{y})\right)^{-1}, \quad (6.23)$$

$$\Psi_{new} = \frac{1}{N}\text{diagonal}\left(\mathbf{x}\mathbf{x}^T - WE(\mathbf{x}|\mathbf{y})\mathbf{y}^T\right), \quad (6.24)$$

where diagonal() ensures that the matrix retains values only on the diagonal and the expectations are:

$$E(\mathbf{x}|\mathbf{y}) = \mathbf{W}^T(\mathbf{W}\mathbf{W}^T + \Psi)^{-1}\mathbf{b} \quad (6.25)$$

$$E(\mathbf{x}\mathbf{x}^T|\mathbf{x}) - E(\mathbf{x}|\mathbf{y})E(\mathbf{x}|\mathbf{y})^T = \mathbf{I} - \mathbf{W}^T(\mathbf{W}\mathbf{W}^T + \Psi)^{-1}W. \quad (6.26)$$

The only other things that we need to add to the algorithm is some way to decide when to stop, which involves computing the log likelihood and stopping the algorithm when it stops descending. This leads to an algorithm where the basic steps in the loop are:

```
# E-step
A = np.dot(W,np.transpose(W)) + np.diag(Psi)
logA = np.log(np.abs(np.linalg.det(A)))
A = np.linalg.inv(A)

WA = np.dot(np.transpose(W),A)
WAC = np.dot(WA,C)
Exx = np.eye(nRedDim) - np.dot(WA,W) + np.dot(WAC,np.transpose(WA))

# M-step
W = np.dot(np.transpose(WAC),np.linalg.inv(Exx))
Psi = Cd - (np.dot(W,WAC)).diagonal()

tAC = (A*np.transpose(C)).sum()

L = -N/2*np.log(2.*np.pi) -0.5*logA - 0.5*tAC
if (L-oldL)<(1e-4):
    print "Stop",i
    break
```

The output of using factor analysis on the iris dataset are shown in Figure 6.11.

6.4 INDEPENDENT COMPONENTS ANALYSIS (ICA)

There is a related approach to factor analysis that is known as Independent Components Analysis. When we looked at PCA above, the components were chosen so that they were orthogonal and uncorrelated (so that the covariance matrix was diagonal, i.e., so $\text{cov}(\mathbf{b}_i, \mathbf{b}_j) = 0$ if $i \neq j$). If, instead, we require that the components are statistically independent (so that for $E[\mathbf{b}_i, \mathbf{b}_j] = E[\mathbf{b}_i]E[\mathbf{b}_j]$ as well as the $\mathbf{b}_i$ being uncorrelated), then we get ICA.

The common motivation for ICA is the problem of blind source separation. As with factor analysis, the assumption is that the data we see are actually created by a set of underlying

FIGURE 6.11 Plot of the first two factor analysis components of the iris data, showing that the three classes are clearly distinguishable.

physical processes that are independent. The reason why the data we see are correlated is because of the way the outputs from different processes have been mixed together. So given some data, we want to find a transformation that turns it into a mixture of independent sources or components.

The most popular way to describe blind source separation is known as the cocktail party problem. If you are at a party, then your ears hear lots of different sounds coming from lots of different locations (different people talking, the clink of glasses, background music, etc.) but you are somehow able to focus on the voice of the people you are talking to, and can in fact separate out the sounds from all of the different sources even though they are mixed together. The cocktail party problem is the challenge of separating out these sources, although there is one wrinkle: for the algorithm to work, you need as many ears as there are sources. This is because the algorithm does not have the information we have about what things sound like.

Suppose that we have two sources making noise (s_1^t, s_2^t) where the top index covers the fact that there are lots of datapoints appearing over time, and two microphones that hear things, giving inputs (x_1^t, x_2^t). The sounds that are heard come from the sources as:

$$x_1 = as_1 + bs_2, \tag{6.27}$$

$$x_2 = cs_1 + ds_2, \tag{6.28}$$

which can be written in matrix form as:

$$\mathbf{x} = \mathbf{A}\mathbf{s}, \tag{6.29}$$

where $\mathbf{A}$ is known as the mixing matrix. Reconstructing $\mathbf{s}$ looks easy now: we just compute $\mathbf{s} = \mathbf{A}^{-1}\mathbf{x}$. Except that, unfortunately, we don't know $\mathbf{A}$. The approximation to $\mathbf{A}^{-1}$ that we work out is generally labelled as $\mathbf{W}$, and it is a square matrix since we have the same number of microphones as we do sources.

At this point we need to work out what we actually know about the sources and the signals. There are three things:

- the mixtures are not independent, even though the sources are

- the mixtures will look like normal distributions even if the sources are not (this is because of the Central Limit Theorem, something that we won't look at further here)
- the mixtures will look more complicated than the sources

We can use the first fact to say that if we find factors that are independent of each other then they are probably sources, and the second to say that if we find factors that are not Gaussian then they are probably sources. We can measure the amount of independence between two variables by using the mutual information, which we will see in Section 12.2.1 when we look at entropy. In fact, the most common approach is to use what is rather uglily known as negentropy: $J(y) = H(z) - H(y)$, which maximises the deviations from Gaussianness (where $H(\cdot)$ is the entropy):

$$H(y) = -\int g(y) \log g(y) dy. \tag{6.30}$$

One common approximation is $J(y) = (E[G(y)] - E[G(z)])^2$, where $g(u) = \frac{1}{a} \log \cosh(au)$, so $g'(u) = \tanh(au)$ $1 \leq a \leq 2$. Implementing ICA is actually quite tricky because of some numerical issues, so we won't do it ourselves. There are a few well-used ICA implementations out there, of which the most popular is known as FastICA, which is available in Python as part of the MDP package.

6.5 LOCALLY LINEAR EMBEDDING

Two relatively recent methods of computing dimensionality reduction were mentioned in the introduction because they were published in the journal *Science*. Both are non-linear, and both attempt to preserve the neighbourhood relations in the data (as will be discussed for the SOM in Section 14.3) but they use different approaches. The first tries to approximate the data by sticking together sets of locally flat patches that cover the dataset, while the second uses the shortest distances (geodesics) on the non-linear space to find a globally optimal solution.

We will look first at the locally linear algorithm, which is called Locally Linear Embedding (LLE). It was introduced by Roweis and Saul in 2000. The idea is to say that by making linear approximations we will make some errors, so we should make these errors as small as possible by making the patches small where there is lots of non-linearity in the data. The error is known as the reconstruction error and is simply the sum-of-squares of the distance between the original point and its reconstruction:

$$\epsilon = \sum_{i=1}^{N} \left(\mathbf{x}_i - \sum_{j=1}^{N} \mathbf{W}_{ij} \mathbf{x}_j \right)^2. \tag{6.31}$$

The weights $\mathbf{W}_{ij}$ say how much effect the jth datapoint has on the reconstruction of the ith one. The question is which points can be usefully used to reconstruct a particular datapoint. If another point is a long way off, then it probably isn't very useful: only those points that are close to the current datapoint (that are in its neighbourhood) are used. There are two common ways to create neighbourhoods:

- Points that are less than some predefined distance d to the current point are neighbours (so we don't know how many neighbours there are, but they are all close)
- The k nearest points are neighbours (so we know how many there are, but some could be far away)

Solving for the weights $\mathbf{W}_{ij}$ is a least-squares problem, which we can simplify by enforcing the constraints that for any point $\mathbf{x}_j$ that is a long way from the current point $\mathbf{x}_i$, $\mathbf{W}_{ij} = 0$, and that $\sum_j \mathbf{W}_{ij} = 1$. This produces a reconstruction of the data, but it does not reduce the dimensionality at all. For this we have to reapply the same basic cost function, but minimise it according to the positions $\mathbf{y}_i$ of the points in some lower dimensional space (dimension L):

$$\mathbf{y}_i = \sum_{i=1}^{N} \left(\mathbf{y}_i - \sum_{j=1}^{L} \mathbf{W}_{ij} \mathbf{y}_j \right)^2 . \tag{6.32}$$

Solving this is rather more complicated, so we won't go into details, but it turns out that the solution is the eigenvalues of the quadratic form matrix $\mathbf{M}_{ij} = \delta_{ij} - \mathbf{W}_{ij} - \mathbf{W}_{ji} + \sum_k \mathbf{W}_{ji}\mathbf{W}_{kj}$, where δ_{ij} is the Kronecker delta function, so $\delta_{ij} = 1$ if $i = j$ and 0 otherwise. This leads to the following algorithm:

The Locally Linear Embedding Algorithm

- Decide on the neighbours of each point (e.g., K nearest neighbours):
  - compute distances between every pair of points
  - find the k smallest distances
  - set $\mathbf{W}_{ij} = 0$ for other points
  - for each point $\mathbf{x}_i$:
    - create a list of its neighbours' locations $\mathbf{z}_i$
    - compute $\mathbf{z}_i = \mathbf{z}_i - \mathbf{x}_i$
- Compute the weights matrix $\mathbf{W}$ that minimises Equation (6.31) according to the constraints:
  - compute local covariance $\mathbf{C} = \mathbf{Z}\mathbf{Z}^T$, where $\mathbf{Z}$ is the matrix of $\mathbf{z}_i$s
  - solve $\mathbf{C}\mathbf{W} = \mathbf{I}$ for $\mathbf{W}$, where $\mathbf{I}$ is the $N \times N$ identity matrix
  - set $\mathbf{W}_{ij} = 0$ for non-neighbours
  - set other elements to $\mathbf{W}/\sum(\mathbf{W})$
- Compute the lower dimensional vectors $\mathbf{y}_i$ that minimise Equation (6.32):
  - create $\mathbf{M} = (\mathbf{I} - \mathbf{W})^T(\mathbf{I} - \mathbf{W})$
  - compute the eigenvalues and eigenvectors of $\mathbf{M}$
  - sort the eigenvectors into order by size of eigenvalue
  - set the qth row of $\mathbf{y}$ to be the $q+1$ eigenvector corresponding to the qth smallest eigenvalue (ignore the first eigenvector, which has eigenvalue 0)

There are a couple of things in there that are a bit tricky to implement, and there is a function that we haven't used before, `np.kron()`, which takes two matrices and multiplies each element of the first one by all the elements of the second, putting all of the results together into one multi-dimensional output array. It is used to construct the set of neighbourhood locations for each point.

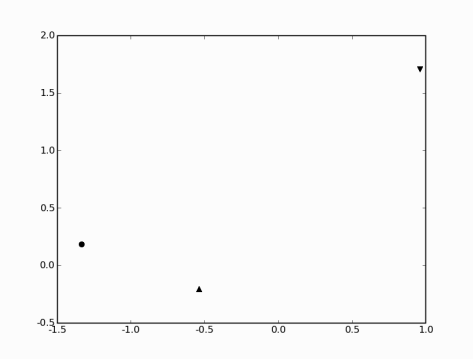

FIGURE 6.12 The Locally Linear Embedding algorithm with $k = 12$ neighbours transforms the iris dataset into three points, separating the data perfectly.

```
for i in range(ndata):
    Z  = data[neighbours[i,:],:] - np.kron(np.ones((K,1)),data[i,:])
    C = np.dot(Z,np.transpose(Z))
    C = C+np.identity(K)*1e-3*np.trace(C)
    W[:,i] = np.transpose(np.linalg.solve(C,np.ones((K,1))))
    W[:,i] = W[:,i]/np.sum(W[:,i])

M = np.eye(ndata,dtype=float)
for i in range(ndata):
    w = np.transpose(np.ones((1,np.shape(W)[0]))*np.transpose(W[:,i]))
    j = neighbours[i,:]
    #print shape(w), np.shape(np.dot(w,np.transpose(w))), np.shape(M[i,j])
    ww = np.dot(w,np.transpose(w))
    for k in range(K):
        M[i,j[k]] -= w[k]
        M[j[k],i] -= w[k]
        for l in range(K):
            M[j[k],j[l]] += ww[k,l]

evals,evecs = np.linalg.eig(M)
ind = np.argsort(evals)
y = evecs[:,ind[1:nRedDim+1]]*np.sqrt(ndata)
```

The LLE algorithm produces a very interesting result on the iris dataset: it separates the three groups into three points (Figure 6.12). This shows that the algorithm works very well on this type of data, but doesn't give us any hints as to what else it can do. Figure 6.13 shows a common demonstration dataset for these algorithms. Known as the swissroll for obvious reasons, it is tricky to find a 2D representation of the 3D data because it is rolled up. The right of Figure 6.13 shows that LLE can successfully unroll it.

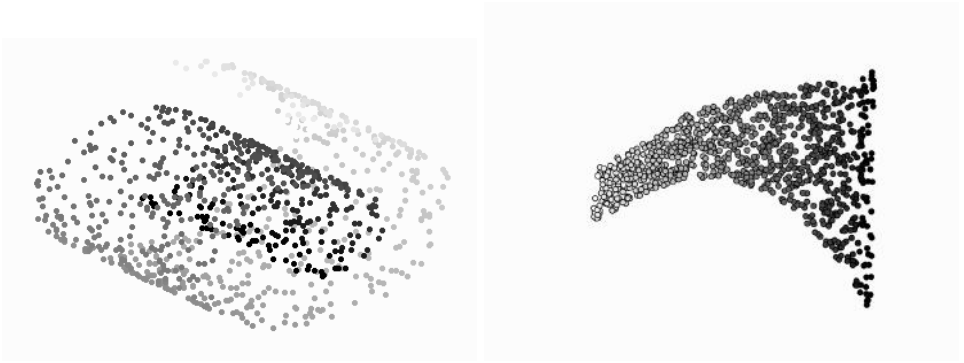

FIGURE 6.13 A common example used to demonstrate LLE is the swissroll dataset shown on the left. To produce a useful 2D representation of this data requires unrolling the data, which the LLE does successfully, as is shown on the right. The shades are used to identify neighbouring points, and do not have any other purpose.

6.6 ISOMAP

The other algorithm was proposed by Tenenbaum et al., also in 2000. It tries to minimise the global error by looking at all of the pairwise distances and computing global geodesics. It is a variant of the standard multi-dimensional scaling (MDS) algorithm, so we'll talk about that first.

6.6.1 Multi-Dimensional Scaling (MDS)

Like PCA, MDS tries to find a linear approximation to the full dataspace that embeds the data into a lower dimensionality. In the case of MDS the embedding tries to preserve the distances between all pairs of points (however these distances are measured). It turns out that if the space is Euclidean, then the two methods are identical. We use the same notational setup as previously, starting with datapoints $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N \in \mathbb{R}^M$. We choose a new dimensionality $L < M$ and compute the embedding so that the datapoints are $\mathbf{z}_1, \mathbf{z}_2, \ldots \mathbf{z}_N \in \mathbb{R}^L$. As usual, we need a cost function to minimise. There are lots of choices for MDS cost functions, but the more common ones are:

Kruskal–Shephard scaling (also known as least-squares) $S_{KS}(\mathbf{z}_1, \mathbf{z}_2, \ldots \mathbf{z}_N) = \sum_{i \neq i'} (d_{ii'} - \|\mathbf{z}_i - \mathbf{z}'_i\|)^2$

Sammon mapping $S_{SM}(\mathbf{z}_1, \mathbf{z}_2, \ldots \mathbf{z}_N) = \sum_{i \neq i'} \frac{(d_{ii'} - \|\mathbf{z}_i - \mathbf{z}'_i\|)^2}{d_{ii'}}$. This puts more weight onto short distances, so that neighbouring points stay the correct distance apart.

In either case, gradient descent can be used to minimise the distances. There is another version of MDS called classical MDS that uses similarities between datapoints rather than distances. These can be constructed from a set of distances by using the centred inner product $s_{ii'} = (\mathbf{x}_i - \bar{\mathbf{x}}), (\mathbf{x}'_i - \bar{\mathbf{x}})^T$. By doing this it is possible to construct a direct algorithm that does not have to use gradient descent. The function that needs to be minimised is $\sum_{i \neq i'} \left(s_{ii'} - (\mathbf{z}_i - \bar{\mathbf{z}}), (\mathbf{z}'_i - \bar{\mathbf{z}})^T\right)^2$. The computations that are needed are:

The Multi-Dimensional Scaling (MDS) Algorithm

- Compute the matrix of squared pairwise similarities $\mathbf{D}$, $\mathbf{D}_{ij} = \|\mathbf{x}_i - \mathbf{x}_j\|^2$
- Compute $\mathbf{J} = \mathbf{I}_N - 1/N$ (where $\mathbf{I}_N$ is the $N \times N$ identity function and N is the number of datapoints)
- Compute $\mathbf{B} = -\frac{1}{2}\mathbf{J}\mathbf{D}\mathbf{J}^T$
- Find the L largest eigenvalues λ_i of $\mathbf{B}$, together with the corresponding eigenvectors $\mathbf{e}_i$
- Put the eigenvalues into a diagonal matrix $\mathbf{V}$ and set the eigenvectors to be columns of matrix $\mathbf{P}$
- Compute the embedding as $\mathbf{X} = \mathbf{P}\mathbf{V}^{1/2}$

This classical MDS algorithm works fine on flat manifolds (dataspaces). However, we are interested in manifolds that are not flat, and this is where Isomap comes in. The algorithm has to construct the distance matrix for all pairs of datapoints on the manifold, but there is no information about the manifold, and so the distances can't be computed exactly. Isomap approximates them by assuming that the distances between pairs of points that are close together are good, since over a small distance the non-linearity of the manifold won't matter. It builds up the distances between points that are far away by finding paths that run through points that are close together, i.e., that are neighbours, and then uses normal MDS on this distance matrix:

The Isomap Algorithm

- Construct the pairwise distances between all pairs of points
- Identify the neighbours of each point to make a weighted graph G
- Estimate the geodesic distances d_G by finding shortest paths
- Apply classical MDS to the set of d_G

Floyd's and Dijkstra's algorithms are well-known algorithms for finding shortest paths on graphs. They are of $\mathcal{O}(N^3)$ and $\mathcal{O}(N^2)$ time complexity, respectively. Any good algorithms textbook provides the details if you don't know them.

There is one practical aspect of Isomap, which is that getting the number of neighbours right can be important, otherwise the graph splits into separate components (that is, segments of the graph that are not linked to each other), which have infinite distance between them. You then have to be careful to deal only with the largest component, which means that you end up with less data than you started with. Otherwise the implementation is fairly simple.

Figure 6.14 shows the results of applying Isomap to the iris dataset. Here, the default neighbourhood size of 12 produced a largest component that held only one of the three classes, and the other two were deleted. By increasing the neighbourhood size over 50, so that each point had more neighbours than were in its class, the results shown in the figure were produced. On the swissroll dataset shown on the left of Figure 6.13, Isomap produces qualitatively similar results to LLE, as can be seen in Figure 6.15.

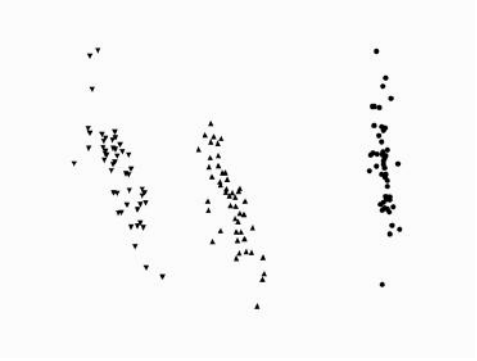

FIGURE 6.14 Isomap transforms the iris data in a similar way to factor analysis, provided that the neighbourhood size is large enough to avoid points becoming disconnected.

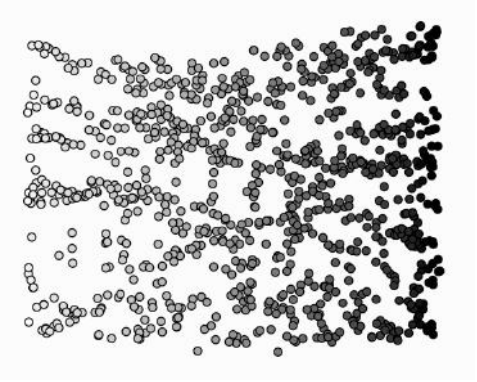

FIGURE 6.15 Isomap also produces a good remapping of the swissroll dataset.

Although the two algorithms produce similar mappings of the swissroll dataset, they are based on different principles. Isomap attempts to find a mapping that preserves the distances between pairs of points within the manifold, no matter how far apart they are, while LLE focuses only on local regions of the manifold. This means that the computational cost of LLE is significantly less, but it can make errors by putting points close together that should be far apart. The choice of which algorithm to use often depends upon the dataset, and trying both of them out for your particular dataset is often a good idea.

FURTHER READING

Surveys of the area of dimensionality reduction include:

- L.J.P. van der Maaten. An introduction to dimensionality reduction using MATLAB. Technical Report MICC 07-07, Maastricht University, Maastricht, the Netherlands, 2007.
- F. Camastra. Data dimensionality estimation methods: a survey. *Pattern Recognition*, 36:2945–2954, 2003.

For more information about many of the methods desribed here, there are books or papers that contain a lot of information. Notable references include:

- (for LDA) Section 4.3 of T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*, 2nd edition, Springer, Berlin, Germany, 2008.
- (for PCA) I.T. Jolliffe. *Principal Components Analysis*. Springer, Berlin, Germany, 1986.
- (for kernel PCA) J. Shawe-Taylor and N. Cristianini. *Kernel Methods for Pattern Analysis*. Cambridge University Press, Cambridge, UK, 2004.
- (for ICA) J.V. Stone. *Independent Components Analysis: A Tutorial Introduction*. MIT Press, Cambridge, MA, USA, 2004.
- (for ICA) A. Hyvrinen and E. Oja. Independent components analysis: Algorithms and applications. *Neural Networks*, 13(4–5):411–430, 2000.
- (for LLE) S. Roweis and L. Saul. Nonlinear dimensionality reduction by locally linear embedding. *Science*, 290(5500):2323–2326, 2000.
- (for MDS) T.F. Cox and M.A.A. Cox. *Multidimensional Scaling*. Chapman & Hall, London, UK, 1994.
- (for Isomap) J.B. Tenenbaum, V. de Silva, and J.C. Langford. A global geometric framework for nonlinear dimensionality reduction. *Science*, 290(5500):2319–2323, 2000.
- Chapter 12 of C.M. Bishop. *Pattern Recognition and Machine Learning*. Springer, Berlin, Germany, 2006.

PRACTICE QUESTIONS

Problem 6.1 Use LDA on the iris dataset (which is what Fisher originally tested LDA on).

Problem 6.2 Compare the results with using PCA, which is not supervised and will not therefore be able to find the same space.

Problem 6.3 Compute the eigenvalues and eigenvectors of:

$$\begin{pmatrix} 5 & 7 \\ -2 & -4 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 2 & 1 \\ 6 & -1 & 0 \\ -1 & -2 & -1 \end{pmatrix} \tag{6.33}$$

Problem 6.4 Compare the algorithms described in this chapter on a variety of different datasets, including the `yeast` dataset and the `wine` dataset. Input the results of the data reduction method to the MLP and SOM. Are the results better than before this preprocessing?

Problem 6.5 Modify the Isomap code to use Dijkstra's algorithm rather than Floyd's algorithm.

Problem 6.6 Another dataset that the Isomap and LLE algorithms are commonly demonstrated on is the 'S' shape that is available on the website. Download it and test various algorithms, not just Isomap and LLE on it. For Isomap and LLE, try different numbers of neighbours to see the effect that this has.

Probabilistic Learning

One criticism that is often made of neural networks—especially the MLP—is that it is not clear exactly what it is doing: while we can go and have a look at the activations of the neurons and the weights, they don't tell us much. We've already seen some methods that don't have this problem, principally the decision tree in Chapter 12. In this chapter we are going to look at methods that are based on statistics, and that are therefore more transparent, in that we can always extract and look at the probabilities and see what they are, rather than having to worry about weights that have no obvious meaning.

We will look at how to perform classification by using the frequency with which examples appear in the training data, and then we will see how we can deal with our first example of unsupervised learning, when the labels are not present for the training examples. If the data comes from known probability distributions, then we will see that it is possible to solve this problem with a very neat algorithm, the EM algorithm, which we will also see in other guises in later chapters. Finally, we will have a look at a rather different way of using the dataset when we look at nearest neighbour methods.

7.1 GAUSSIAN MIXTURE MODELS

For the Bayes' classifier that we saw in Section 2.3.2 the data had target labels, and so we could do supervised learning, learning the probabilities from the labelled data. However, suppose that we have the same data, but without target labels. This requires unsupervised learning, and we will see lots of ways to deal with this in Chapters 14 and 6, but here we will look at one special case. Suppose that the different classes each come from their own Gaussian distribution. This is known as multi-modal data, since there is one distribution (mode) for each different class. We can't fit one Gaussian to the data, because it doesn't look Gaussian overall.

There is, however, something we can do. If we know how many classes there are in the data, then we can try to estimate the parameters for that many Gaussians, all at once. If we don't know, then we can try different numbers and see which one works best. We will talk about this issue more for a different method (the k-means algorithm) in Section 14.1. It is perfectly possible to use any other probability distribution instead of a Gaussian, but Gaussians are by far the most common choice. Then the output for any particular datapoint that is input to the algorithm will be the sum of the values expected by all of the M Gaussians:

$$f(\mathbf{x}) = \sum_{m=1}^{M} \alpha_m \phi(\mathbf{x}; \boldsymbol{\mu}_m, \boldsymbol{\Sigma}_m), \tag{7.1}$$

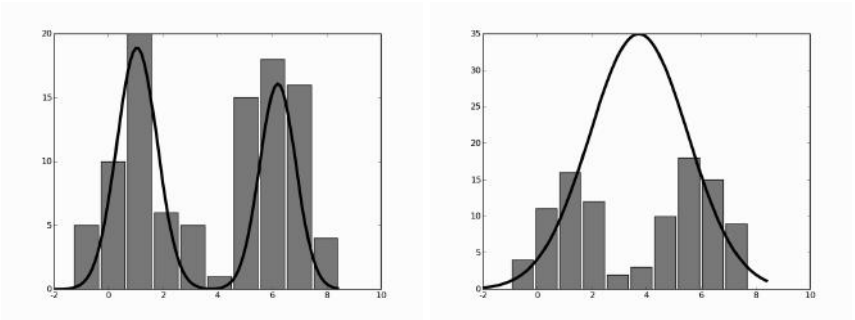

FIGURE 7.1 Histograms of training data from a mixture of two Gaussians and two fitted models, shown as the line plot. The model shown on the left fits well, but the one on the right produces two Gaussians right on top of each other that do not fit the data well.

where $\phi(\mathbf{x}; \boldsymbol{\mu}_m, \boldsymbol{\Sigma}_m)$ is a Gaussian function with mean $\boldsymbol{\mu}_m$ and covariance matrix $\boldsymbol{\Sigma}_m$, and the α_m are weights with the constraint that $\sum_{m=1}^{M} \alpha_m = 1$.

Figure 7.1 shows two examples, where the data (shown by the histograms) comes from two different Gaussians, and the model is computed as a sum or mixture of the two Gaussians together. The figure also gives you some idea of how to use the mixture model once it has been created. The probability that input $\mathbf{x}_i$ belongs to class m can be written as (where a hat on a variable ($\hat{\cdot}$) means that we are estimating the value of that variable):

$$p(\mathbf{x}_i \in c_m) = \frac{\hat{\alpha}_m \phi(\mathbf{x}_i; \hat{\boldsymbol{\mu}}_m; \hat{\boldsymbol{\Sigma}}_m)}{\sum_{k=1}^{M} \hat{\alpha}_m \phi(\mathbf{x}_i; \hat{\boldsymbol{\mu}}_k; \hat{\boldsymbol{\Sigma}}_k)}. \tag{7.2}$$

The problem is how to choose the weights α_m. The common approach is to aim for the maximum likelihood solution (the likelihood is the conditional probability of the data given the model, and the maximum likelihood solution varies the model to maximise this conditional probability). In fact, it is common to compute the log likelihood and then to maximise that; it is guaranteed to be negative, since probabilities are all less than 1, and the logarithm spreads out the values, making the optimisation more effective. The algorithm that is used is an example of a very general one known as the expectation-maximisation (or more compactly, EM) algorithm. The reason for the name will become clearer below. We will see another example of an EM algorithm in Section 16.3.3, but here we see how to use it for fitting Gaussian mixtures, and get a very approximate idea of how the algorithm works for more general examples. For more details, see the Further Reading section.

7.1.1 The Expectation-Maximisation (EM) Algorithm

The basic idea of the EM algorithm is that sometimes it is easier to add extra variables that are not actually known (called hidden or latent variables) and then to maximise the function over those variables. This might seem to be making a problem much more complicated than it needs to be, but it turns out for many problems that it makes finding the solution significantly easier.

In order to see how it works, we will consider the simplest interesting case of the Gaussian mixture model: a combination of just two Gaussian mixtures. The assumption now is that

data were created by randomly choosing one of two possible Gaussians, and then creating a sample from that Gaussian. If the probability of picking Gaussian one is p, then the entire model looks like this (where $\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\sigma}^2)$ specifies a Gaussian distribution with mean $\boldsymbol{\mu}$ and standard deviation $\boldsymbol{\sigma}$):

$$\begin{aligned} G_1 &= \mathcal{N}(\boldsymbol{\mu}_1, \boldsymbol{\sigma}_1^2) \\ G_2 &= \mathcal{N}(\boldsymbol{\mu}_2, \boldsymbol{\sigma}_2^2) \\ y &= pG_1 + (1-p)G_2. \end{aligned} \tag{7.3}$$

If the probability distribution of p is written as $\boldsymbol{\pi}$, then the probability density is:

$$P(\mathbf{y}) = \boldsymbol{\pi}\phi(\mathbf{y}; \boldsymbol{\mu}_1, \boldsymbol{\sigma}_1) + (1-\boldsymbol{\pi})\phi(\mathbf{y}; \boldsymbol{\mu}_2, \boldsymbol{\sigma}_2). \tag{7.4}$$

Finding the maximum likelihood solution (actually the maximum log likelihood) to this problem is then a case of computing the sum of the logarithm of Equation (7.4) over all of the training data, and differentiating it, which would be rather difficult. Fortunately, there is a way around it. The key insight that we need is that if we knew which of the two Gaussian components the datapoint came from, then the computation would be easy. The mean and standard deviation for each component could be computed from the datapoints that belong to that component, and there would not be a problem. Although we don't know which component each datapoint came from, we can pretend we do, by introducing a new variable f. If $f = 0$ then the data came from Gaussian one, if $f = 1$ then it came from Gaussian two.

This is the typical initial step of an EM algorithm: adding latent variables. Now we just need to work out how to optimise over them. This is the time when the reason for the algorithm being called expectation-maximisation becomes clear. We don't know much about variable f (hardly surprising, since we invented it), but we can compute its expectation (that is, the value that we 'expect' to see, which is the mean average) from the data:

$$\begin{aligned} \boldsymbol{\gamma}_i(\hat{\boldsymbol{\mu}}_1, \hat{\boldsymbol{\mu}}_2, \hat{\boldsymbol{\sigma}}_1, \hat{\boldsymbol{\sigma}}_2, \hat{\boldsymbol{\pi}}) &= E(f|\hat{\boldsymbol{\mu}}_1, \hat{\boldsymbol{\mu}}_2, \hat{\boldsymbol{\sigma}}_1, \hat{\boldsymbol{\sigma}}_2, \hat{\boldsymbol{\pi}}, D) \\ &= P(f=1|\hat{\boldsymbol{\mu}}_1, \hat{\boldsymbol{\mu}}_2, \hat{\boldsymbol{\sigma}}_1, \hat{\boldsymbol{\sigma}}_2, \hat{\boldsymbol{\pi}}, D), \end{aligned} \tag{7.5}$$

where D denotes the data. Note that since we have set $f = 1$ this means that we are choosing Gaussian two.

Computing the value of this expectation is known as the E-step. Then this estimate of the expectation is maximised over the model parameters (the parameters of the two Gaussians and the mixing parameter $\boldsymbol{\pi}$), the M-step. This requires differentiating the expectation with respect to each of the model parameters. These two steps are simply iterated until the algorithm converges. Note that the estimate never gets any smaller, and it turns out that EM algorithms are guaranteed to reach a local maxima.

To see how this looks for the two-component Gaussian mixture, we'll take a closer look at the algorithm:

The Gaussian Mixture Model EM Algorithm

- Initialisation
  - set $\hat{\mu}_1$ and $\hat{\mu}_2$ to be randomly chosen values from the dataset
  - set $\hat{\sigma}_1 = \hat{\sigma}_2 = \sum_{i=1}^{N}(y_i - \bar{y})^2/N$ (where $\bar{y}$ is the mean of the entire dataset)
  - set $\hat{\pi} = 0.5$
- Repeat until convergence:
  - (E-step) $\hat{\gamma}_i = \frac{\hat{\pi}\phi(y_i;\hat{\mu}_1,\hat{\sigma}_1)}{\hat{\pi}\phi(y_i;\hat{\mu}_1,\hat{\sigma}_1)+(1-\hat{\pi})\phi(y_i;\hat{\mu}_2,\hat{\sigma}_2)}$ for $i = 1 \ldots N$
  - (M-step 1) $\hat{\mu}_1 = \frac{\sum_{i=1}^{N}(1-\hat{\gamma}_i)y_i}{\sum_{i=1}^{N}(1-\hat{\gamma}_i)}$
  - (M-step 2) $\hat{\mu}_2 = \frac{\sum_{i=1}^{N}\hat{\gamma}_i y_i}{\sum_{i=1}^{N}\hat{\gamma}_i}$
  - (M-step 3) $\hat{\sigma}_1 = \frac{\sum_{i=1}^{N}(1-\hat{\gamma}_i)(y_i-\hat{\mu}_1)^2}{\sum_{i=1}^{N}(1-\hat{\gamma}_i)}$
  - (M-step 4) $\hat{\sigma}_2 = \frac{\sum_{i=1}^{N}\hat{\gamma}_i(y_i-\hat{\mu}_2)^2}{\sum_{i=1}^{N}\hat{\gamma}_i}$
  - (M-step 5) $\hat{\pi} = \sum_{i=1}^{N}\frac{\hat{\gamma}_i}{N}$

Turning this into Python code does not require any new techniques:

```
while count<nits:
    count = count + 1

    # E-step
    for i in range(N):
        gamma[i] = pi*np.exp(-(y[i]-mu1)**2/(2*s1))/ (pi * np.exp(-(y[i]-
        mu1)**2/(2*s1)) + (1-pi)* np.exp(-(y[i]-mu2)**2/2*s2))

    # M-step
    mu1 = np.sum((1-gamma)*y)/np.sum(1-gamma)
    mu2 = np.sum(gamma*y)/np.sum(gamma)
    s1 = np.sum((1-gamma)*(y-mu1)**2)/np.sum(1-gamma)
```

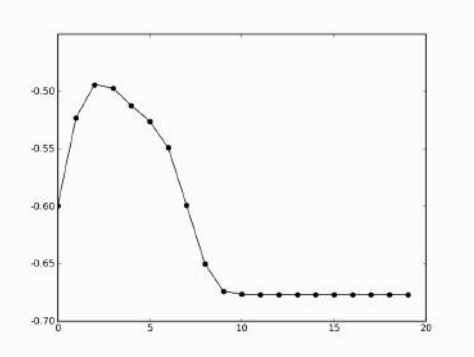

FIGURE 7.2 Plot of the log likelihood changing as the Gaussian Mixture Model EM algorithm learns to fit the two Gaussians shown on the left of Figure 7.1.

```
s2 = np.sum(gamma*(y-mu2)**2)/np.sum(gamma)
pi = np.sum(gamma)/N

ll[count-1] = np.sum(np.log(pi*np.exp(-(y[i]-mu1)**2/(2*s1)) + (1-pi)*
*np.exp(-(y[i]-mu2)**2/(2*s2))))
```

Figure 7.2 shows the log likelihood dropping as the algorithm learns for the example on the left of Figure 7.1. The computational costs of this model are very good for classifying a new datapoint, since it is $\mathcal{O}(M)$, where M is the number of Gaussians, which is often of the order of $\log N$ (where N is the number of datapoints). The training is, however, fairly expensive: $\mathcal{O}(NM^2 + M^3)$.

The general algorithm has pretty much exactly the same steps (the parameters of the model are written as $\boldsymbol{\theta}$, $\boldsymbol{\theta}'$ is a dummy variable, D is the original dataset, and D' is the dataset with the latent variables included):

The General Expectation-Maximisation (EM) Algorithm

- Initialisation
  - guess parameters $\hat{\boldsymbol{\theta}}^{(0)}$
- Repeat until convergence:
  - (E-step) compute the expectation $Q(\boldsymbol{\theta}', \hat{\boldsymbol{\theta}}^{(j)}) = E(f(\boldsymbol{\theta}'; D')|D, \hat{\boldsymbol{\theta}}^{(j)})$
  - (M-step) estimate the new parameters $\hat{\boldsymbol{\theta}}^{(j+1)}$ as $\max_{\boldsymbol{\theta}'} Q(\boldsymbol{\theta}', \hat{\boldsymbol{\theta}}^{(j)})$

The trick with applying EM algorithms to problems is in identifying the correct latent variables to include, and then simply working through the steps. They are very powerful methods for a wide variety of statistical learning problems.

We are now going to turn our attention to something much simpler, which is how we can use information about nearby datapoints to decide on classification output. For this we don't use a model of the data at all, but directly use the data that is available.

7.1.2 Information Criteria

The likelihood of the data given the model has another useful function as well. Back in Section 2.2.2 we identified the need to use model selection in order to identify the right time to stop learning. In that section we introduced the idea of a validation set, or using cross-validation if there was not enough data. However, this replaces data with computation time, as many models are trained on different datasets.

An alternative idea is to identify some measure that tells us about how well we can expect this trained model to perform. There are two such information criteria that are commonly used:

Aikake Information Criterium

$$\text{AIC} = \ln(\mathcal{L}) - k \tag{7.6}$$

Bayesian Information Criterium

$$\text{BIC} = 2\ln(\mathcal{L}) - k\ln N \tag{7.7}$$

In these equations, k is the number of parameters in the model, N is the number of training examples, and $\mathcal{L}$ is the best (largest) likelihood of the model. In both cases, based on the way that they are written here, the model with the largest value is taken. Both of the measures will favour simple models, which is a form of Occam's razor.

7.2 NEAREST NEIGHBOUR METHODS

Suppose that you are in a nightclub and decide to dance. It is unlikely that you will know the dance moves for the particular song that is playing, so you will probably try to work out what to do by looking at what the people close to you are doing. The first thing you could do would be just to pick the person closest to you and copy them. However, since most of the people who are in the nightclub are also unlikely to know all the moves, you might decide to look at a few more people and do what most of them are doing. This is pretty much exactly the idea behind nearest neighbour methods: if we don't have a model that describes the data, then the best thing to do is to look at similar data and choose to be in the same class as them.

We have the datapoints positioned within input space, so we just need to work out which of the training data are close to it. This requires computing the distance to each datapoint in the training set, which is relatively expensive: if we are in normal Euclidean space, then we have to compute d subtractions and d squarings (we can ignore the square root since we only want to know which points are the closest, not the actual distance) and this has to be done $\mathcal{O}(N^2)$ times. We can then identify the k nearest neighbours to the test point, and then set the class of the test point to be the most common one out of those for the nearest neighbours. The choice of k is not trivial. Make it too small and nearest neighbour methods are sensitive to noise, too large and the accuracy reduces as points that are too far away are considered. Some possible effects of changing the size of k on the decision boundary are shown in Figure 7.3.

This method suffers from the curse of dimensionality (Section 2.1.2). First, as shown above, the computational costs get higher as the number of dimensions grows. This is not as bad as it might appear at first: there are sets of methods such as KD-Trees (see Section 7.2.2 for more details) that compute this in $\mathcal{O}(N \log N)$ time. However, more importantly, as the number of dimensions increases, so the distance to other datapoints tends to increase. In

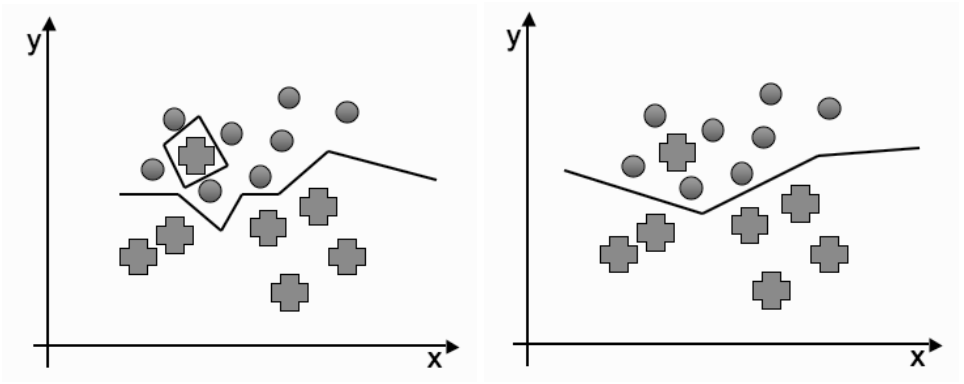

FIGURE 7.3 The nearest neighbours decision boundary with *left:* one neighbour and *right:* two neighbours.

addition, they can be far away in a variety of different directions—there might be points that are relatively close in some dimensions, but a long way in others. There are methods for dealing with these problems, known as adaptive nearest neighbour methods, and there is a reference to them in the Further Reading section at the end of the chapter.

The only part of this that requires any care during the implementation is what to do when there is more than one class found in the closest points, but even with that the implementation is nice and simple:

```
def knn(k,data,dataClass,inputs):

        nInputs = np.shape(inputs)[0]
        closest = np.zeros(nInputs)

        for n in range(nInputs):
                # Compute distances
                distances = np.sum((data-inputs[n,:])**2,axis=1)

                # Identify the nearest neighbours
                indices = np.argsort(distances,axis=0)

                classes = np.unique(dataClass[indices[:k]])
                if len(classes)==1:
                        closest[n] = np.unique(classes)
                else:
                        counts = np.zeros(max(classes)+1)
                        for i in range(k):
                                counts[dataClass[indices[i]]] += 1
                        closest[n] = np.max(counts)

        return closest
```

We are going to look next at how we can use these methods for regression, before we turn to the question of how to perform the distance calculations as efficiently as possible, something that is done simply but inefficiently in the code above. We will then consider briefly whether or not the Euclidean distance is always the most useful way to calculate distances, and what alternatives there are.

For the k-nearest neighbours algorithm the bias-variance decomposition can be computed as:

$$E((\mathbf{y} - \hat{f}(\mathbf{x}))^2) = \sigma^2 + \left[f(\mathbf{x}) - \frac{1}{k} \sum_{i=0}^{k} f(\mathbf{x}_i) \right]^2 + \frac{\sigma^2}{k}. \tag{7.8}$$

The way to interpret this is that when k is small, so that there are few neighbours considered, the model has flexibility and can represent the underlying model well, but that it makes mistakes (has high variance) because there is relatively little data. As k increases, the variance decreases, but at the cost of less flexibility and so more bias.

7.2.1 Nearest Neighbour Smoothing

Nearest neighbour methods can also be used for regression by returning the average value of the neighbours to a point, or a spline or similar fit as the new value. The most common methods are known as kernel smoothers, and they use a kernel (a weighting function between pairs of points) that decides how much emphasis (weight) to put onto the contribution from each datapoint according to its distance from the input. We will see kernels in a different context in Section 8.2, but here we shall simply use two kernels that are used for smoothing.

Both of these kernels are designed to give more weight to points that are closer to the current input, with the weights decreasing smoothly to zero as they pass out of the range of the current input, with the range specified by a parameter λ. They are the Epanechnikov quadratic kernel:

$$K_{E,\lambda}(x_0, x) = \begin{cases} 0.75\left(1 - (x_0 - x)^2/\lambda^2\right) & \text{if } |x - x_0| < \lambda \\ 0 & \text{otherwise} \end{cases}, \tag{7.9}$$

and the tricube kernel:

$$K_{T,\lambda}(x_0, x) = \begin{cases} \left(1 - \left|\frac{x_0 - x}{\lambda}\right|^3\right)^3 & \text{if } |x - x_0| < \lambda \\ 0 & \text{otherwise} \end{cases}. \tag{7.10}$$

The results of using these kernels are shown in Figure 7.4 on a dataset that consists of the time between eruptions (technically known as the repose) and the duration of the eruptions of Mount Ruapehu, the large volcano in the centre of New Zealand's north island. Values of λ of 2 and 4 were used here. Picking λ requires experimentation. Large values average over more datapoints, and therefore produce lower variance, but at the cost of higher bias.

7.2.2 Efficient Distance Computations: the KD-Tree

As was mentioned above, computing the distances between all pairs of points is very computationally expensive. Fortunately, as with many problems in computer science, designing an efficient data structure can reduce the computational overhead a lot. For the problem of finding nearest neighbours the data structure of choice is the KD-Tree. It has been around since the late 1970s, when it was devised by Friedman and Bentley, and it reduces the cost of finding a nearest neighbour to $\mathcal{O}(\log N)$ for $\mathcal{O}(N)$ storage. The construction of the tree

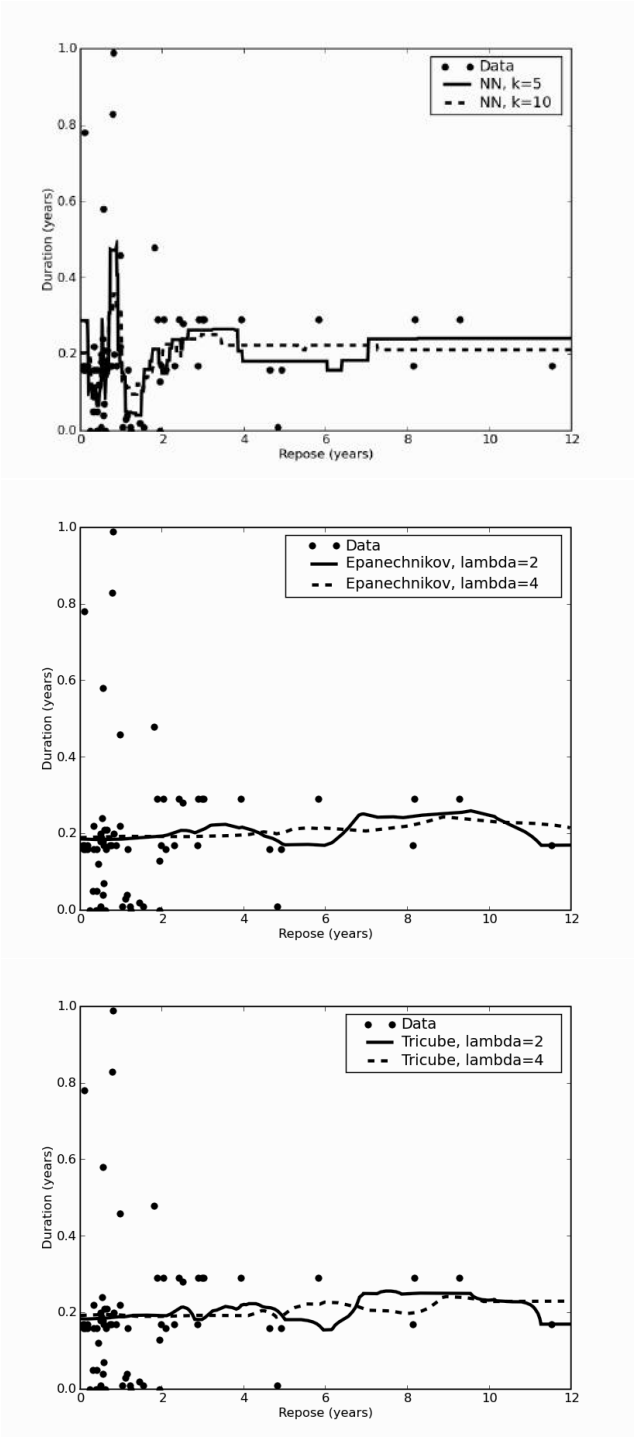

FIGURE 7.4 Output of the nearest neighbour method and two kernel smoothers on the data of duration and repose of eruptions of Mount Ruapehu 1860–2006.

is $\mathcal{O}(N \log^2 N)$, with much of the computational cost being in the computation of the median, which with a naïve algorithm requires a sort and is therefore $\mathcal{O}(N \log N)$, or can be computed with a randomised algorithm in $\mathcal{O}(N)$ time.

The idea behind the KD-tree is very simple. You create a binary tree by choosing one dimension at a time to split into two, and placing the line through the median of the point coordinates of that dimension. Not that different to a decision tree (Chapter 12), really. The points themselves end up as leaves of the tree. Making the tree follows pretty much the same steps as usual for constructing a binary tree: we identify a place to split into two choices, left and right, and then carry on down the tree. This makes it natural to write the algorithm recursively. The choice of what to split and where is what makes the KD-tree special. Just one dimension is split in each step, and the position of the split is found by computing the median of the points that are to be split in that one dimension, and putting the line there. In general, the choice of which dimension to split alternates through the different choices, or it can be made randomly. The algorithm below cycles through the possible dimensions based on the depth of the tree so far, so that in two dimensions it alternates horizontal and vertical splits.

The centre of the construction method is simply a recursive function that picks the axis to split on, finds the median value on that axis, and separates the points according to that value, which in Python can be written as:

```
# Pick next axis to split on
whichAxis = np.mod(depth,np.shape(points)[1])

# Find the median point
indices = np.argsort(points[:,whichAxis])
points = points[indices,:]
median = np.ceil(float(np.shape(points)[0]-1)/2)

# Separate the remaining points
goLeft = points[:median,:]
goRight = points[median+1:,:]

# Make a new branching node and recurse
newNode = node()
newNode.point = points[median,:]
newNode.left = makeKDtree(goLeft,depth+1)
newNode.right = makeKDtree(goRight,depth+1)
return newNode
```

Suppose that we had seven two-dimensional points to make a tree from: $(5, 4), (1, 6), (6, 1), (7, 5), (2, 7), (2, 2), (5, 8)$ (as plotted in Figure 7.5). The algorithm will pick the first coordinate to split on initially, and the median point here is 5, so the split is through $x = 5$. Of those on the left of the line, the median y coordinate is 6, and for those on the right it is 5. At this point we have separated all the points, and so the algorithm terminates with the split shown in Figure 7.6 and the tree shown in Figure 7.7.

Searching the tree is the same as any other binary tree; we are more interested in finding the nearest neighbours of a test point. This is fairly easy: starting at the root of the tree you recurse down through the tree comparing just one dimension at a time until you find a

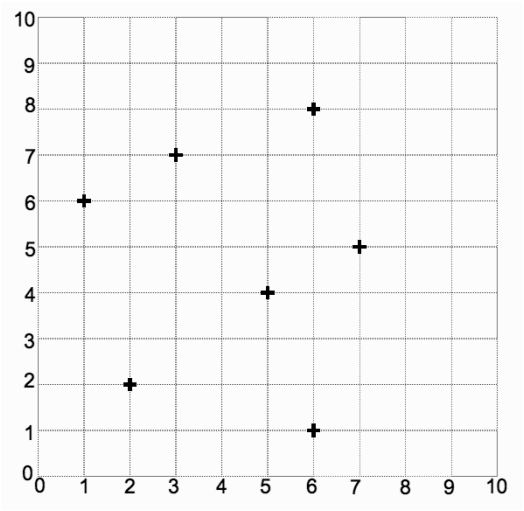

FIGURE 7.5 The initial set of 2D data.

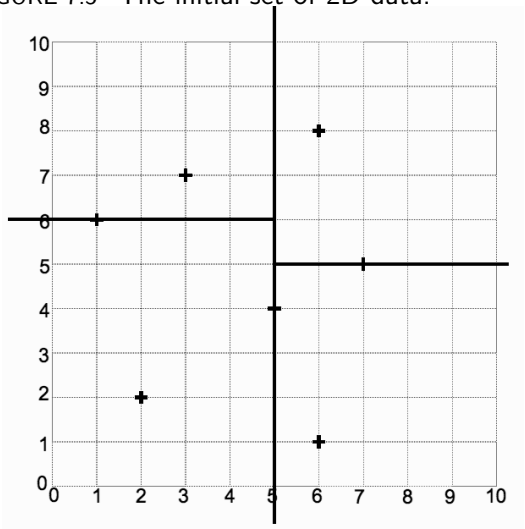

FIGURE 7.6 The splits and leaf points found by the KD-tree.

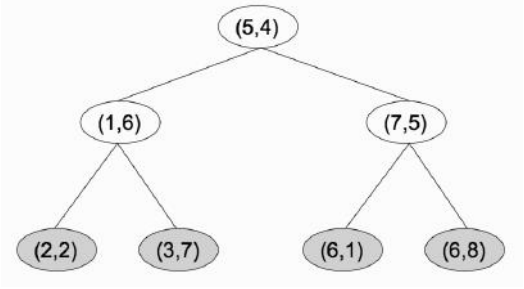

FIGURE 7.7 The KD-tree that made the splits.

leaf node that is in the region containing the test point. Using the tree shown in Figure 7.7 we introduce the test point $(3, 5)$, which finds $(2, 2)$ as the leaf for the box that $(3, 5)$ is in. However, looking at Figure 7.8 we see that this is not the closest point at all, so we need to do some more work.

The first thing we do is label the leaf we have found as a potential nearest neighbour, and compute the distance between the test point and this point, since any other point has to be closer. Now we need to check any other boxes that could contain something closer. Looking at Figure 7.8 you can see that point $(3, 7)$ is closer, and that is the label of the leaf for the sibling box to the one that was returned, so the algorithm also needs to check the sibling box. However, suppose that we used $(4.5, 2)$ as the test point. In that case the sibling is too far away, but another point $(6, 1)$ is closer. So just checking the sibling is not enough — we also need to check the siblings of the parent node, together with its descendants (the cousins of the first point). A look at the figure again should convince you that the algorithm can then terminate in most cases; very occasionally it can be necessary to go even further afield, but it is easy to see which branches to prune. This leads to the following Python program:

```
def returnNearest(tree,point,depth):
  if tree.left is None:
      # Have reached a leaf
      distance = np.sum((tree.point-point)**2)
      return tree.point,distance,0
  else:
      # Pick next axis to split on
      whichAxis = np.mod(depth,np.shape(point)[0])

      # Recurse down the tree
      if point[whichAxis]<tree.point[whichAxis]:
          bestGuess,distance,height = returnNearest(tree.left,point,depth+1)
      else:
          bestGuess,distance,height = returnNearest(tree.right,point,depth+1)

    if height<=2:
      # Check the sibling
      if point[whichAxis]<tree.point[whichAxis]:
          bestGuess2,distance2,height2 =                                    returnNear-
est(tree.right,point,depth+
          1)
      else:
          bestGuess2,distance2,height2 =                                    returnNear-
est(tree.left,point,depth+1)

    # Check this node
    distance3 = np.sum((tree.point-point)**2)
    if (distance3<distance2):
      distance2 = distance3
          bestGuess2 = tree.point
```

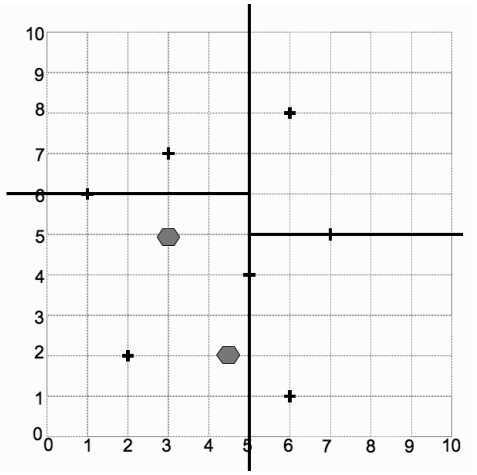

FIGURE 7.8 Two test points for the example KD-tree.

```
if (distance2<distance):
     distance = distance2
     bestGuess = bestGuess2
return bestGuess,distance,height+1
```

7.2.3 Distance Measures

We have computed the distance between points as the Euclidean distance, which is something that you learnt about in high school. However, it is not the only option, nor is it necessarily the most useful. In this section we will look at the underlying idea behind distance calculations and possible alternatives.

If I were to ask you to find the distance between my house and the nearest shop, then your first guess might involve taking a map of my town, locating my house and the shop, and using a ruler to measure the distance between them. By careful application of the map scale you can now tell me how far it is. However, when I set out to buy some milk I'm liable to find that I have to walk rather further than you've told me, since the direct line that you measured would involve walking through (or over) several houses, and some serious fence-scaling. Your 'as the crow flies' distance is the shortest possible path, and it is the straight-line, or Euclidean, distance. You can measure it on the map by just using a ruler, but it essentially consists of measuring the distance in one direction (we'll call it north-south) and then the distance in another direction that is perpendicular to the first (let's call it east-west) and then squaring them, adding them together, and then taking the square root of that. Writing that out, the Euclidean distance that we are all used to is:

$$d_E = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}, \tag{7.11}$$

where (x_1, y_1) is the location of my house in some coordinate system (say by using a GPS tracker) and (x_2, y_2) is the location of the shop.

If I told you that my town was laid out on a grid block system, as is common in towns

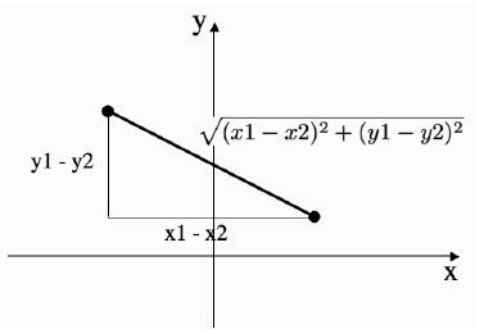

FIGURE 7.9 The Euclidean and city-block distances between two points.

that were built in the interval between the invention of the motor car and the invention of innovative town planners, then you would probably use a different measure. You would measure the distance between my house and the shop in the 'north-south' direction and the distance in the 'east-west' direction, and then add the two distances together. This would correspond to the distance I actually had to walk. It is often known as the city-block or Manhattan distance and looks like:

$$d_C = |x_1 - x_2| + |y_1 - y_2|. \tag{7.12}$$

The point of this discussion is to show that there is more than one way to measure a distance, and that they can provide radically different answers. These two different distances can be seen in Figure 7.9. Mathematically, these distance measures are known as metrics. A metric function or norm takes two inputs and gives a scalar (the distance) back, which is positive, and 0 if and only if the two points are the same, symmetric (so that the distance to the shop is the same as the distance back), and obeys the triangle inequality, which says that the distance from a to b plus the distance from b to c should not be less than the direct distance from a to c.

Most of the data that we are going to have to analyse lives in rather more than two dimensions. Fortunately, the Euclidean distance that we know about generalises very well to higher dimensions (and so does the city-block metric). In fact, these two measures are both instances of a class of metrics that work in any number of dimensions. The general measure is the Minkowski metric and it is written as:

$$L_k(\mathbf{x}, \mathbf{y}) = \left(\sum_{i=1}^{d} |x_i - y_i|^k \right)^{\frac{1}{k}}. \tag{7.13}$$

If we put $k = 1$ then we get the city-block distance (Equation (7.12)), and $k = 2$ gives the Euclidean distance (Equation (7.11)). Thus, you might possibly see the Euclidean metric written as the L_2 norm and the city-block distance as the L_1 norm. These norms have another interesting feature. Remember that we can define different averages of a set of numbers. If we define the average as the point that minimises the sum of the distance to every datapoint, then it turns out that the mean minimises the Euclidean distance (the sum-of-squares distance), and the median minimises the L_1 metric. We met another distance measure earlier: the Mahalanobis distance in Section 2.4.2.

There are plenty of other possible metrics to choose, depending upon the dataspace. We generally assume that the space is flat (if it isn't, then none of these techniques work, and

we don't want to worry about that). However, it can still be beneficial to look at other metrics. Suppose that we want our classifier to be able to recognise images, for example of faces. We take a set of digital photos of faces and use the pixel values as features. Then we use the nearest neighbour algorithm that we've just seen to identify each face. Even if we ensure that all of the photos are taken fully face-on, there are still a few things that will get in the way of this method. One is that slight variations in the angle of the head (or the camera) could make a difference; another is that different distances between the face and the camera (scaling) will change the results; and another is that different lighting conditions will make a difference. We can try to fix all of these things in preprocessing, but there is also another alternative: use a different metric that is invariant to these changes, i.e., it does not vary as they do. The idea of invariant metrics is to find measures that ignore changes that you don't want. So if you want to be able to rotate shapes around and still recognise them, you need a metric that is invariant to rotation.

A common invariant metric in use for images is the tangent distance, which is an approximation to the Taylor expansion in first derivatives, and works very well for small rotations and scalings; for example, it was used to halve the final error rate on nearest neighbour classification of a set of handwritten letters. Invariant metrics are an interesting topic for further study, and there is a reference for them in the Further Reading section if you are interested.

FURTHER READING

For more on nearest neighbour methods, see:

- T. Hastie and R. Tibshirani. Discriminant adaptive nearest neighbor classification and regression. In David S. Touretzky, Michael C. Mozer, and Michael E. Hasselmo, editors, *Advances in Neural Information Processing Systems*, volume 8, pages 409–415. The MIT Press, 1996.
- N.S. Altman. An introduction to kernel and nearest-neighbor nonparametric regression. *The American Statistician*, 46:175–185, 1992.

The original description of KD-trees is:

- A. Moore. A tutorial on KD-trees. Extract from PhD Thesis, 1991. Available from http://www.cs.cmu.edu/*sim*awm/papers.html.

A reference on the tangent distance is:

- P.Y. Simard, Y.A. Le Cun, J.S. Denker, and B. Victorri. Transformation invariance in pattern recognition: Tangent distance and propagation. *International Journal of Imaging Systems and Technology*, 11:181–194, 2001.

Some of the material in the chapter is covered in:

- Section 9.2 of C.M. Bishop. *Pattern Recognition and Machine Learning.* Springer, Berlin, Germany, 2006.
- Chapter 6 (especially Sections 6.1–6.3) of T. Mitchell. *Machine Learning.* McGraw-Hill, New York, USA, 1997.
- Section 13.3 of T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*, 2nd edition, Springer, Berlin, Germany, 2008.

PRACTICE QUESTIONS

Problem 7.1 Extend the Gaussian Mixture Model algorithm to allow for more than two classes in the data. This is not trivial, since it involves modifying the EM algorithm.

Problem 7.2 Modify the KD-tree algorithm so that it works on spheres in the data, rather than rectangles. Since they no longer cover the space you will have to add some cases that fail to return a leaf at all. However, this means that the algorithm will not return points that are far away, which will make the results more accurate. Now modify it so that it does not use the Euclidean distance, but rather the L_1 distance. Compare the results of using these two methods on the `iris` dataset.

Problem 7.3 Use the small figures of numbers that are available on the book website in order to compute the tangent distance. You will have to write code that rotates the numbers by small amounts in order to check that you have written it correctly. What happens when you make large rotations (particularly of a 6 or 9)? Compare using nearest neighbours with Euclidean distance and the tangent distance to verify the results claimed in the chapter. Extend the experiment to the MNIST dataset.

Support Vector Machines

Back in Chapter 3 we looked at the Perceptron, a set of McCulloch and Pitts neurons arranged in a single layer. We identified a method by which we could modify the weights so that the network learned, and then saw that the Perceptron was rather limited in that it could only identify straight line classifiers, that is, it could only separate out groups of data if it was possible to draw a straight line (hyperplane in higher dimensions) between them. This meant that it could not learn to distinguish between the two truth classes of the 2D XOR function. However, in Section 3.4.3, we saw that it was possible to modify the problem so that the Perceptron could solve the problem, by changing the data so that it used more dimensions than the original data.

This chapter is concerned with a method that makes use of that insight, amongst other things. The main idea is one that we have seen before, in Section 5.3, which is to modify the data by changing its representation. However, the terminology is different here, and we will introduce kernel functions rather than bases. In principle, it is always possible to transform any set of data so that the classes within it can be separated linearly. To get a bit of a handle on this, think again about what we did with the XOR problem in Section 3.4.3: we added an extra dimension and moved a point that we could not classify properly into that additional dimension so that we could linearly separate the classes. The problem is how to work out which dimensions to use, and that is what kernel methods, which is the class of algorithms that we will talk about in this chapter, do.

We will focus on one particular algorithm, the Support Vector Machine (SVM) , which is one of the most popular algorithms in modern machine learning. It was introduced by Vapnik in 1992 and has taken off radically since then, principally because it often (but not always) provides very impressive classification performance on reasonably sized datasets. SVMs do not work well on extremely large datasets, since (as we shall see) the computations don't scale well with the number of training examples, and so become computationally very expensive. This should be sufficient motivation to master the (quite complex) concepts that are needed to understand the algorithm.

We will develop a simple SVM in this chapter, using `cvxopt`, a freely available solver with a Python interface, to do the heavy work. There are several different implementations of the SVM available on the Internet, and there are references to some of the more popular ones at the end of the chapter. Some of them include wrappers so that they can be used from within Python.

There is rather more to the SVM than the kernel method; the algorithm also reformulates the classification problem in such a way that we can tell a good classifier from a bad one, even if they both give the same results on a particular dataset. It is this distinction that enables the SVM algorithm to be derived, so that is where we will start.

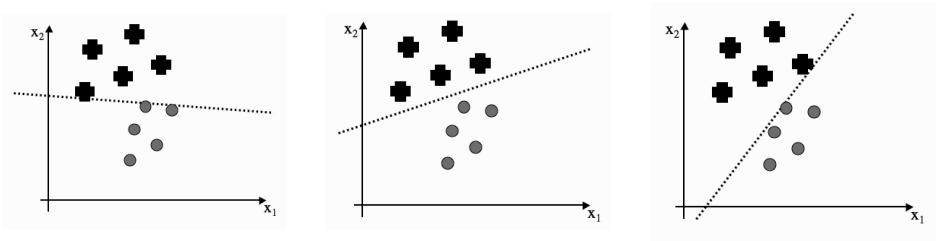

FIGURE 8.1 Three different classification lines. Is there any reason why one is better than the others?

8.1 OPTIMAL SEPARATION

Figure 8.1 shows a simple classification problem with three different possible linear classification lines. All three of the lines that are drawn separate out the two classes, so in some sense they are 'correct', and the Perceptron would stop its training if it reached any one of them. However, if you had to pick one of the lines to act as the classifier for a set of test data, I'm guessing that most of you would pick the line shown in the middle picture. It's probably hard to describe exactly why you would do this, but somehow we prefer a line that runs through the middle of the separation between the datapoints from the two classes, staying approximately equidistant from the data in both classes. Of course, if you were feeling smart, then you might have asked what criteria you were meant to pick a line based on, and why one of the lines should be any better than the others.

To answer that, we are going to try to define why the line that runs halfway between the two sets of datapoints is better, and then work out some way to quantify that so we can identify the 'optimal' line, that is, the best line according to our criteria. The data that we have used to identify the classification line is our training data. We believe that these data are indicative of some underlying process that we are trying to learn, and that the testing data that the algorithm will be evaluated on after training comes from the same underlying process. However, we don't expect to see exactly the same datapoints in the test dataset, and inevitably some of the points will be closer to the classifier line, and some will be further away. If we pick the lines shown in the left or right graphs of Figure 8.1, then there is a chance that a datapoint from one class will be on the wrong side of the line, just because we have put the line tight up against some of the datapoints we have seen in the training set. The line in the middle picture doesn't have this problem; like the baby bear's porridge in Goldilocks, it is 'just right'.

8.1.1 The Margin and Support Vectors

How can we quantify this? We can measure the distance that we have to travel away from the line (in a direction perpendicular to the line) before we hit a datapoint. Imagine that we put a 'no-man's land' around the line (shown in Figure 8.2), so that any point that lies within that region is declared to be too close to the line to be accurately classified. This region is symmetric about the line, so that it forms a cylinder about the line in 3D, and a hyper-cylinder in higher dimensions. How large could we make the radius of this cylinder until we started to put points into a no-man's land, where we don't know which class they are from? This largest radius is known as the margin, labelled M. The margin was mentioned briefly in Section 3.4.1, where it affected the speed at which the Perceptron converged. The classifier in the middle of Figure 8.1 has the largest margin of the three. It

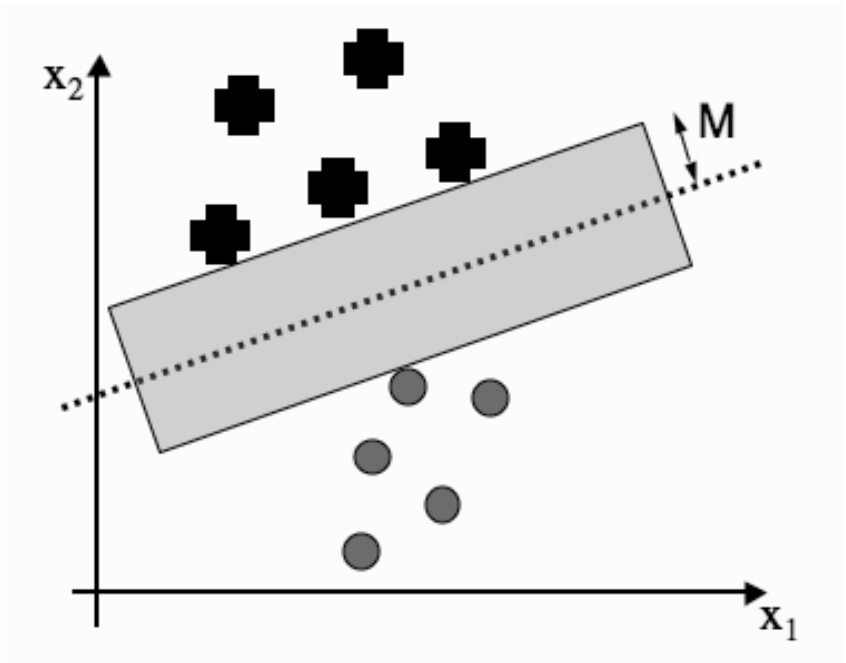

FIGURE 8.2 The margin is the largest region we can put that separates the classes without there being any points inside, where the box is made from two lines that are parallel to the decision boundary.

has the imaginative name of the maximum margin (linear) classifier. The datapoints in each class that lie closest to the classification line have a name as well. They are called support vectors. Using the argument that the best classifier is the one that goes through the middle of no-man's land, we can now make two arguments: first that the margin should be as large as possible, and second that the support vectors are the most useful datapoints because they are the ones that we might get wrong. This leads to an interesting feature of these algorithms: after training we can throw away all of the data except for the support vectors, and use them for classification, which is a useful saving in data storage.

Now that we've got a measurement that we can use to find the optimal decision boundary, we just need to work out how to actually compute it from a given set of datapoints. Let's start by reminding ourselves of some of the things that we worked out in Chapter 3. We have a weight vector (a vector, not a matrix, since there is only one output) and an input vector $\mathbf{x}$. The output we used in Chapter 3 was $y = \mathbf{w} \cdot \mathbf{x} + b$, with b being the contribution from the bias weight. We use the classifier line by saying that any $\mathbf{x}$ value that gives a positive value for $\mathbf{w} \cdot \mathbf{x} + b$ is above the line, and so is an example of the '+' class, while any $\mathbf{x}$ that gives a negative value is in the '○' class. In our new version of this we want to include our no-man's land. So instead of just looking at whether the value of $\mathbf{w} \cdot \mathbf{x} + b$ is positive or negative, we also check whether the absolute value is less than our margin M, which would put it inside the grey box in Figure 8.2. Remember that $\mathbf{w} \cdot \mathbf{x}$ is the inner or scalar product, $\mathbf{w} \cdot \mathbf{x} = \sum_i w_i x_i$. This can also be written as $\mathbf{w}^T \mathbf{x}$, since this simply means that we treat the vectors as degenerate matrices and use the normal matrix multiplication rules. This notation will turn out to be simpler, and so will be used from here on.

For a given margin value M we can say that any point $\mathbf{x}$ where $\mathbf{w}^T\mathbf{x} + b \geq M$ is a plus, and any point where $\mathbf{w}^T\mathbf{x} + b \leq -M$ is a circle. The actual separating hyperplane is specified by $\mathbf{w}^T\mathbf{x} + b = 0$. Now suppose that we pick a point $\mathbf{x}^+$ that lies on the '+' class boundary line, so that $\mathbf{w}^T\mathbf{x}^+ = M$. This is a support vector. If we want to find the closest point that lies on the boundary line for the '$\circ$' class, then we travel perpendicular to the '+' boundary line until we hit the '$\circ$' boundary line. The point that we hit is the closest point, and we'll call it $\mathbf{x}^-$. How far did we have to travel in this direction? Figure 8.2 hopefully makes it clear that the distance we travelled is M to get to the separating hyperplane, and then M from there to the opposing support vector. We can use this fact to write down the margin size M in terms of $\mathbf{w}$ if we remember one extra thing from Chapter 3, namely that the weight vector $\mathbf{w}$ is perpendicular to the classifier line. If it is perpendicular to the classifier line, then it is obviously perpendicular to the '+' and '$\circ$' boundary lines too, so the direction we travelled from $\mathbf{x}^+$ to $\mathbf{x}^-$ is along $\mathbf{w}$. Now we need to make $\mathbf{w}$ a unit vector $\mathbf{w}/\|w\|$, and so we see that the margin is $1/\|w\|$. In some texts the margin is actually written as the total distance between the support vectors, so that it would be twice the one that we have computed.

So now, given a classifier line (that is, the vector $\mathbf{w}$ and scalar b that define the line $\mathbf{w}^T\mathbf{x} + b$) we can compute the margin M. We can also check that it puts all of the points on the right side of the classification line. Of course, that isn't actually what we want to do: we want to find the $\mathbf{w}$ and b that give us the biggest possible value of M. Our knowledge that the width of the margin is $1/\|w\|$ tells us that making M as large as possible is the same as making $\mathbf{w}^T\mathbf{w}$ as small as possible. If that was the only constraint, then we could just set $\mathbf{w} = \mathbf{0}$, and the problem would be solved, but we also want the classification line to separate out the '+' data from the '$\circ$', that is, actually act as a classifier. So we are going to need to try to satisfy two problems simultaneously: find a decision boundary that classifies well, while also making $\mathbf{w}^T\mathbf{w}$ as small as possible. Mathematically, we can write these requirements as: minimise $\frac{1}{2}\mathbf{w}^T\mathbf{w}$ (where the half is there for convenience as in so many other cases) subject to some constraint that says that the data are well matched. The next thing is to work out what these constraints are.

8.1.2 A Constrained Optimisation Problem

How do we decide whether or not a classifier is any good? Obviously, the fewer mistakes that it makes, the better. So we can write down a set of constraints that say that the classifier should get the answer right. To do this we make the target answers for our two classes be ± 1, rather than 0 and 1. We can then write down $t_i \times y_i$, that is, the target multiplied by the output, and this will be positive if the two are the same and negative otherwise. We can write down the equation of the straight line again, which is how we computed y, to see that we require that $t_i(\mathbf{w}^T\mathbf{x} + b) \geq 1$. This means that the constraints just need to check each datapoint for this condition. So the full problem that we wish to solve is:

$$\text{minimise } \frac{1}{2}\mathbf{w}^T\mathbf{w} \text{ subject to } t_i(\mathbf{w}^T\mathbf{x}_i + b) \geq 1 \text{ for all } i = 1, \ldots n. \tag{8.1}$$

We've put in a lot of effort to write down this equation, but we don't know how to solve it. We could try and use gradient descent, but we would have to put a lot of effort into making it enforce the constraints, and it would be very, very slow and inefficient for the problem. There is a method that is much better suited, which is quadratic programming, which takes advantage of the fact that the problem we have described is quadratic and therefore convex, and has linear constraints. A convex problem is one where if we take any two points on the line and join them with a straight line, then every point on the line will

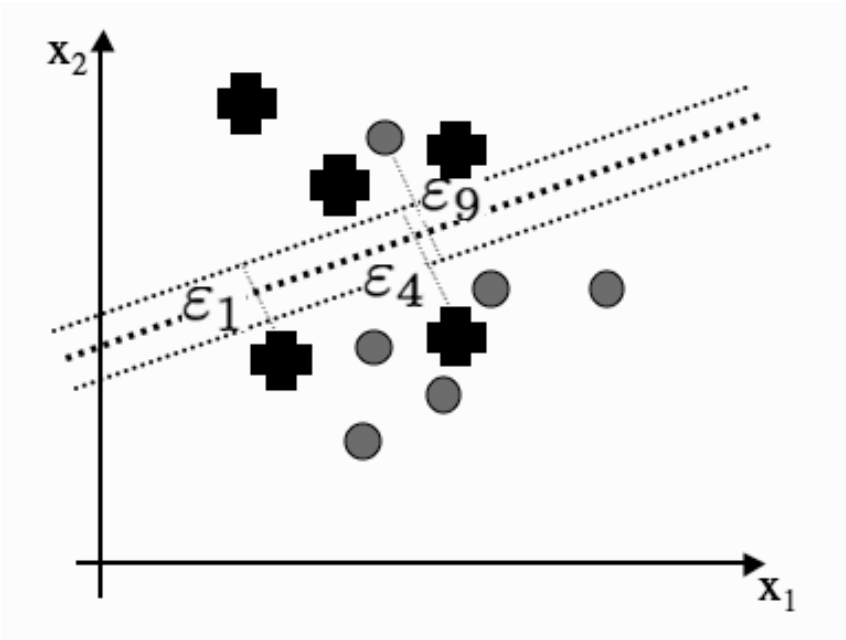

FIGURE 8.3 If the classifier makes some errors, then the distance by which the points are over the border should be used to weight each error in order to decide how bad the classifier is.

be above the curve. Figure 8.4 shows an example of a convex and a non-convex function. Convex functions have a unique minimum, which is fairly easy to see in one dimension, and remains true in any number of dimensions.

The practical upshot of these facts for us is that the types of problem that we are interested in can be solved directly and efficiently (i.e., in polynomial time). There are very effective quadratic programming solvers available, but it is not an algorithm that we will consider writing ourselves. We will, however, work out how to formulate the problem so that it can be presented to a quadratic program solver, and then use one of the programs that other people have been nice enough to prepare and make freely available.

Since the problem is quadratic, there is a unique optimum. When we find that optimal solution, the Karush–Kuhn–Tucker (KKT) conditions will be satisfied. These are (for all values of i from 1 to n, and where the $*$ denotes the optimal value of each parameter):

$$\lambda_i^*(1 - t_i(\mathbf{w}^{*T}\mathbf{x}_i + b^*)) = 0 \tag{8.2}$$

$$1 - t_i(\mathbf{w}^{*T}\mathbf{x}_i + b^*) \leq 0 \tag{8.3}$$

$$\lambda_i^* \geq 0, \tag{8.4}$$

where the λ_i are positive values known as Lagrange multipliers, which are a standard approach to solving equations with equality constraints.

The first of these conditions tells us that if $\lambda_i \neq 0$ then $(1 - t_i(\mathbf{w}^{*T}\mathbf{x}_i + b^*)) = 0$. This is only true for the support vectors (the SVMs provide a sparse representation of the data), and so we only have to consider them, and can ignore the rest. In the jargon, the support vectors

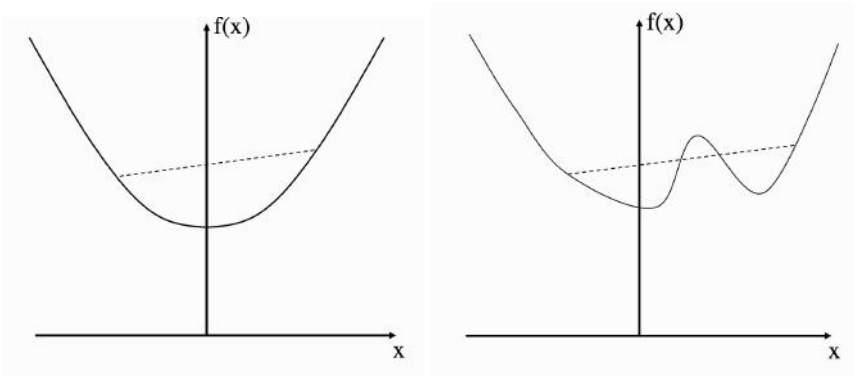

FIGURE 8.4 A function is convex if every straight line that links two points on the curve does not intersect the curve anywhere else. The function on the left is convex, but the one on the right is not, as the dashed line shows.

are those vectors in the active set of constraints. For the support vectors the constraints are equalities instead of inequalities. We can therefore solve the Lagrangian function:

$$\mathcal{L}(\mathbf{w}, b, \boldsymbol{\lambda}) = \frac{1}{2}\mathbf{w}^T\mathbf{w} + \sum_{i=1}^{n} \lambda_i (1 - t_i(\mathbf{w}^T\mathbf{x}_i + b)), \tag{8.5}$$

We differentiate this function with respect to the elements of $\mathbf{w}$ and b:

$$\nabla_{\mathbf{w}}\mathcal{L} = \mathbf{w} - \sum_{i=1}^{n} \lambda_i t_i \mathbf{x}_i, \tag{8.6}$$

and

$$\frac{\partial \mathcal{L}}{\partial b} = -\sum_{i=1}^{n} \lambda_i t_i. \tag{8.7}$$

If we set the derivatives to be equal to zero, so that we find the saddle points (maxima) of the function, we see that:

$$\mathbf{w}^* = \sum_{i=1}^{n} \lambda_i t_i \mathbf{x}_i, \sum_{i=1}^{n} \lambda_i t_i = 0. \tag{8.8}$$

We can substitute these expressions at the optimal values of $\mathbf{w}$ and b into Equation (8.5) and, after a little bit of rearranging, we get (where $\boldsymbol{\lambda}$ is the vector of the λ_i):

$$\mathcal{L}(\mathbf{w}^*, b^*, \boldsymbol{\lambda}) = \sum_{i=1}^{n} \lambda_i - \sum_{i=1}^{n} \lambda_i t_i - \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} \lambda_i \lambda_j t_i t_j \mathbf{x}_i^T \mathbf{x}_j, \tag{8.9}$$

and we can notice that using the derivative with respect to b we can treat the middle term as 0. This equation is known as the dual problem, and the aim is to maximise it with respect to the λ_i variables. The constraints are that $\lambda_i \geq 0$ for all i, and $\sum_{i=1}^{n} \lambda_i t_i = 0$.

Equation (8.8) gives us an expression for $\mathbf{w}^*$, but we also want to know what b^* is. We know that for a support vector $t_i(\mathbf{w}^T\mathbf{x}_i + b) = 1$, and we can substitute the expression for

$\mathbf{w}^*$ into there and substitute in the $(\mathbf{x}, t)$ of one of the support vectors. However, in case of errors this is not very stable, and so it is better to average it over the whole set of N_s support vectors:

$$b^* = \frac{1}{N_s} \sum_{\text{support vectors } j} \left(t_j - \sum_{i=1}^{n} \lambda_i t_i \mathbf{x}_i^T \mathbf{x}_j \right). \tag{8.10}$$

We can also use Equation (8.8) to see how to make a prediction, since for a new point $\mathbf{z}$:

$$\mathbf{w}^{*T}\mathbf{z} + b^* = \left(\sum_{i=1}^{n} \lambda_i t_i \mathbf{x}_i \right)^T \mathbf{z} + b^*. \tag{8.11}$$

This means that to classify a new point, we just need to compute the inner product between the new datapoint and the support vectors.

8.1.3 Slack Variables for Non-Linearly Separable Problems

Everything that we have done so far has assumed that the datatset is linearly separable. We know that this is not always the case, but if we have a non-linearly separable dataset, then we cannot satisfy the constraints for all of the datapoints. The solution is to introduce some slack variables $\eta_i \geq 0$ so that the constraints become $t_i(\mathbf{w}^T\mathbf{x}_i + b) \geq 1 - \eta_i$. For inputs that are correct, we set $\eta_i = 0$.

These slack variables are telling us that, when comparing classifiers, we should consider the case where one classifier makes a mistake by putting a point just on the wrong side of the line, and another puts the same point a long way onto the wrong side of the line. The first classifier is better than the second, because the mistake was not as serious, so we should include this information in our minimisation criterion. We can do this by modifying the problem. In fact, we have to do major surgery, since we want to add a term into the minimisation problem so that we will now minimise $\mathbf{w}^T\mathbf{w} + C\times$ (distance of misclassified points from the correct boundary line). Here, C is a tradeoff parameter that decides how much weight to put onto each of the two criteria: small C means we prize a large margin over a few errors, while large C means the opposite. This transforms the problem into a soft-margin classifier, since we are allowing for a few mistakes. Writing this in a more mathematical way, the function that we want to minimise is:

$$L(\mathbf{w}, \boldsymbol{\epsilon}) = \mathbf{w}^T\mathbf{w} + C\sum_{i=1}^{n} \epsilon_i. \tag{8.12}$$

The derivation of the dual problem that we worked out earlier still holds, except that $0 \leq \lambda_i \leq C$, and the support vectors are now those vectors with $\lambda_i > 0$. The KKT conditions are slightly different, too:

$$\lambda_i^*(1 - t_i(\mathbf{w}^{*T}\mathbf{x}_i + b^*) - \eta_i) = 0 \tag{8.13}$$

$$(C - \lambda_i^*)\eta_i = 0 \tag{8.14}$$

$$\sum_{i=1}^{n} \lambda_i^* t_i = 0. \tag{8.15}$$

The second condition tells us that, if $\lambda_i < C$, then $\eta_i = 0$, which means that these are

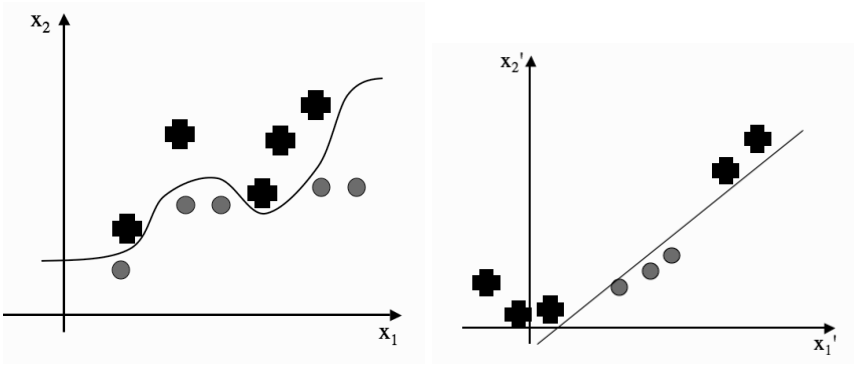

FIGURE 8.5 By modifying the features we hope to find spaces where the data are linearly separable.

the support vectors. If $\lambda_i = C$, then the first condition tells us that if $\eta_i > 1$ then the classifier made a mistake. The problem with this is that it is not as clear how to choose a limited set of vectors, and so most of our training set will be support vectors.

We have now built an optimal linear classifier. However, since most problems are non-linear we seem to have done a lot of work for a case that we could already solve, albeit not as effectively. So while the decision boundary that is found could be better than that found by the Perceptron, if there is not a straight line solution, then the method doesn't work much better than the Perceptron. Not ideal for something that's taken lots of effort to work out! It's time to pull our extra piece of magic out of the hat: transformation of the data.

8.2 KERNELS

To see the idea, have a look at Figure 8.5. Basically, we see that if we modify the features in some way, then we might be able to linearly separate the data, as we did for the XOR problem in Section 3.4.3; if we can use more dimensions, then we might be able to find a linear decision boundary that separates the classes. So all that we need to do is work out what extra dimensions we can use. We can't invent new data, so the new features will have to be derived from the current ones in some way. Just like in Section 5.3, we are going to introduce new functions $\phi(\mathbf{x})$ of our input features.

The important thing is that we are just transforming the data, so that we are making some function $\phi(\mathbf{x}_i)$ from input $\mathbf{x}_i$. The reason why this matters is that we want to be able to use the SVM algorithm that we worked out above, particularly Equation (8.11). The good news is that it isn't any worse, since we can replace $\mathbf{x}_i$ by $\phi(\mathbf{x}_i)$ (and $\mathbf{z}$ by $\phi(\mathbf{z})$) and get a prediction quite easily:

$$\mathbf{w}^T\mathbf{x} + b = \left(\sum_{i=1}^{n} \lambda_i t_i \phi(\mathbf{x}_i)\right)^T \phi(\mathbf{z}) + b. \tag{8.16}$$

We still need to pick what functions to use, of course. If we knew something about the data, then we might be able to identify functions that would be a good idea, but this kind of domain knowledge is not always going to be around, and we would like to automate the algorithm. For now, let's think about a basis that consists of the polynomials of everything up to degree 2. It contains the constant value 1, each of the individual (scalar)

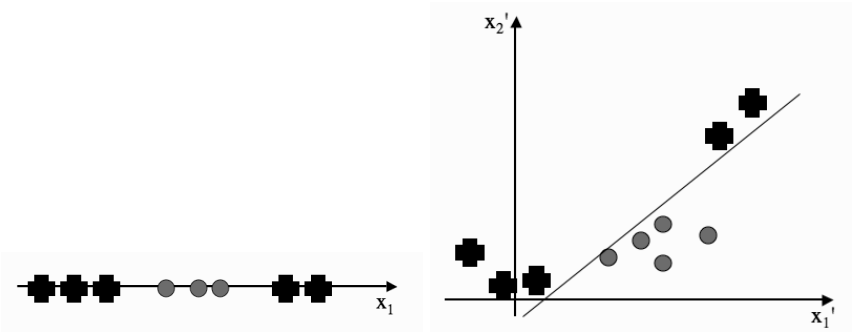

FIGURE 8.6 Using x_1^2 as well as x_1 allows these two classes to be separated.

input elements $x_1, x_2, \ldots, x_d$, and then the squares of each input element $x_1^2, x_2^2, \ldots, x_d^2$, and finally, the products of each pair of elements $x_1x_2, x_1x_3, \ldots, x_{d-1}x_d$. The total input vector made up of all these things is generally written as $\Phi(\mathbf{x})$; it contains about $d^2/2$ elements. The right of Figure 8.6 shows a 2D version of this (with the constant term suppressed), and I'm going to write it out for the case $d = 3$, with a set of $\sqrt{2}$s in there (the reasons for them will become clear soon):

$$\Phi(\mathbf{x}) = (1, \sqrt{2}x_1, \sqrt{2}x_2, \sqrt{2}x_3, x_1^2, x_2^2, x_3^2, \sqrt{2}x_1x_2, \sqrt{2}x_1x_3, \sqrt{2}x_2x_3). \tag{8.17}$$

If there was just one feature, x_1, then we would have changed this from a one-dimensional problem into a three-dimensional one $(1, x_1, x_1^2)$.

The only thing that this has cost us is computational time: the function $\Phi(\mathbf{x}_i)$ has $d^2/2$ elements, and we need to multiply it with another one the same size, and we need to do this many times. This is rather computationally expensive, and if we need to use the powers of the input vector greater than 2 it will be even worse. There is one last piece of trickery that will get us out of this hole: it turns out that we don't actually have to compute $\Phi(\mathbf{x}_i)^T\Phi(\mathbf{x}_j)$. To see how this works, let's work out what $\Phi(\mathbf{x})^T\Phi(\mathbf{y})$ actually is for the example above (where $d = 3$ to match perfectly):

$$\Phi(\mathbf{x})^T\Phi(\mathbf{y}) = 1 + 2\sum_{i=1}^{d} x_iy_i + \sum_{i=1}^{d} x_i^2y_i^2 + 2\sum_{i,j=1;i<j}^{d} x_ix_jy_iy_j. \tag{8.18}$$

You might not recognise that you can factorise this equation, but fortunately somebody did: it can be written as $(1+\mathbf{x}^T\mathbf{y})^2$. The dot product here is in the original space, so it only requires d multiplications, which is obviously much better—this part of the algorithm has now been reduced from $\mathcal{O}(d^2)$ to $\mathcal{O}(d)$. The same thing holds true for the polynomials of any degree s that we are making here, where the cost of the naïve algorithm is $\mathcal{O}(d^s)$. The important thing is that we remove the problem of computing the dot products of all the extended basis vectors, which is expensive, with the computation of a kernel matrix (also known as the Gram matrix) $\mathbf{K}$ that is made from the dot product of the original vectors, which is only linear in cost. This is sometimes known as the kernel trick. It means that you don't even have to know what $\Phi(\cdot)$ is, provided you know a kernel. These kernels are the fundamental reason why these methods work, and the reason why we went to all that effort to produce the dual formulation of the problem. They produce a transformation of the data so that they are in a higher-dimensional space, but because the datapoints only

appear inside those inner products, we don't actually have to do any computations in those higher-dimensional spaces, only in the original (relatively cheap) low-dimensional space.

8.2.1 Choosing Kernels

So how do we go about finding a suitable kernel? Any symmetric function that is positive definite (meaning that it enforces positivity on the integral of arbitrary functions) can be used as a kernel. This is a result of Mercer's theorem, which also says that it is possible to convolve kernels together and the result will be another kernel. However, there are three different types of basis functions that are commonly used, and they have nice kernels that correspond to them:

- polynomials up to some degree s in the elements x_k of the input vector (e.g., x_3^3 or $x_1 \times x_4$) with kernel:
$$\mathbf{K}(\mathbf{x}, \mathbf{y}) = (1 + \mathbf{x}^T\mathbf{y})^s \tag{8.19}$$
For $s = 1$ this gives a linear kernel
- sigmoid functions of the x_ks with parameters κ and δ, and kernel:
$$\mathbf{K}(\mathbf{x}, \mathbf{y}) = \tanh(\kappa\mathbf{x}^T\mathbf{y} - \delta) \tag{8.20}$$
- radial basis function expansions of the x_ks with parameter σ and kernel:
$$\mathbf{K}(\mathbf{x}, \mathbf{y}) = \exp\left(-(\mathbf{x} - \mathbf{y})^2/2\sigma^2\right) \tag{8.21}$$

Choosing which kernel to use and the parameters in these kernels is a tricky problem. While there is some theory based on something known as the Vapnik–Chernik dimension that can be applied, most people just experiment with different values and find one that works, using a validation set as we did for the MLP in Chapter 4.

There are two things that we still need to worry about for the algorithm. One is something that we've discussed in the context of other machine learning algorithms: overfitting, and the other is how we will do testing. The second one is probably worth a little explaining. We used the kernel trick in order to reduce the computations for the training set. We still need to work out how to do the same thing for the testing set, since otherwise we'll be stuck with doing the $\mathcal{O}(d^s)$ computations. In fact, it isn't too hard to get around this problem, because the forward computation for the weights is $\mathbf{w}^T\Phi(\mathbf{x})$, where:

$$\mathbf{w} = \sum_{i \text{ where } \lambda_i > 0} \lambda_i t_i \Phi(\mathbf{x}_i). \tag{8.22}$$

So we still have the computation of $\Phi(\mathbf{x}_i)^T\Phi(\mathbf{x}_j)$, which we can replace using the kernel as before.

The overfitting problem goes away because of the fact that we are still optimising $\mathbf{w}^T\mathbf{w}$ (remember that from somewhere a long way back?), which tries to keep $\mathbf{w}$ small, which means that many of the parameters are kept close to 0.