Chapman & Hall/CRC
Machine Learning & Pattern Recognition Series

Machine Learning

An Algorithmic Perspective

Stephen Marsland

CRC Press
Taylor & Francis Group
A CHAPMAN & HALL BOOK

Machine Learning

An Algorithmic Perspective

Chapman & Hall/CRC
Machine Learning & Pattern Recognition Series

SERIES EDITORS

Ralf Herbrich and 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

MACHINE LEARNING: An Algorithmic Perspective
Stephen Marsland

Chapman & Hall/CRC
Machine Learning & Pattern Recognition Series

Machine Learning

An Algorithmic Perspective

Stephen Marsland
Massey University
Palmerston North, New Zealand

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

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

Chapman & Hall/CRC is an imprint of Taylor & Francis Group, an Informa business

No claim to original U.S. Government works
Printed in the United States of America on acid-free paper
10 9 8 7 6 5 4 3 2 1

International Standard Book Number-13: 978-1-4200-6718-7 (Hardcover)

Library of Congress Cataloging-in-Publication Data

Marsland, Stephen.
Machine learning : an algorithmic perspective / Stephen Marsland.
p. cm. -- (Chapman & Hall/CRC machine learning & pattern recognition series)
Includes bibliographical references and index.
ISBN 978-1-4200-6718-7 (hardcover : alk. paper)
1. Machine learning. 2. Algorithms. I. Title. II. Series.

Q325.5.M368 2009
006.3'1--dc22 2009007292

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

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

For my Monkle

Contents

Prologue

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://seat.massey.ac.nz/personal/s.r.marsland/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 16 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 Repository (`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, thanks to Zbygniew Nowicki, Joseph Marsland, Bob Hodgson, Patrick Rynhart, Gary Allen, Linda Chua, Mark Bebbington, JP Lewis, Tom Duckett, and Monika Nowicki. Thanks also to Jonathan Shapiro, who helped me discover machine learning and who may recognise some of his own examples.

Stephen Marsland
Ashhurst, New Zealand

Chapter 1

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, 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 incessently. 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 size and complexity of these datasets means 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

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

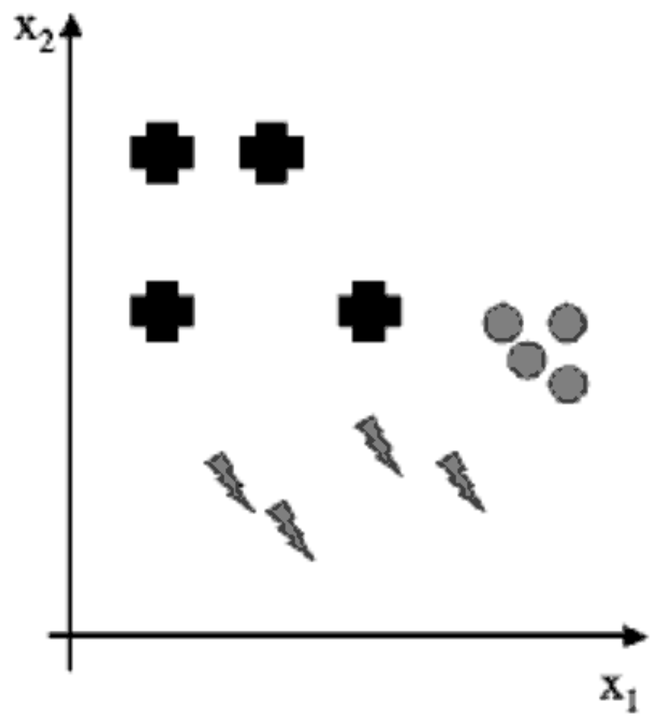


FIGURE 1.1: A set of datapoints as numerical values and as points plotted on a graph. It is easier for us to visualise data than to see it in a table, but if the data has more than three dimensions, we can't view it all at once.

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 that in 2006 about 160 exabytes (160×10^{18} bytes) of data were created and stored, and that this will increase to almost a zettabyte (10^{21} bytes) by 2010. However, to make a black hole the size of the earth would 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!

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.

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 1.5 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 quesion 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) are 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, 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).

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 data points. 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 data points. 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 data points. What about the plot on the top-right? It looks like it goes through all of the data points 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 data points, 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.

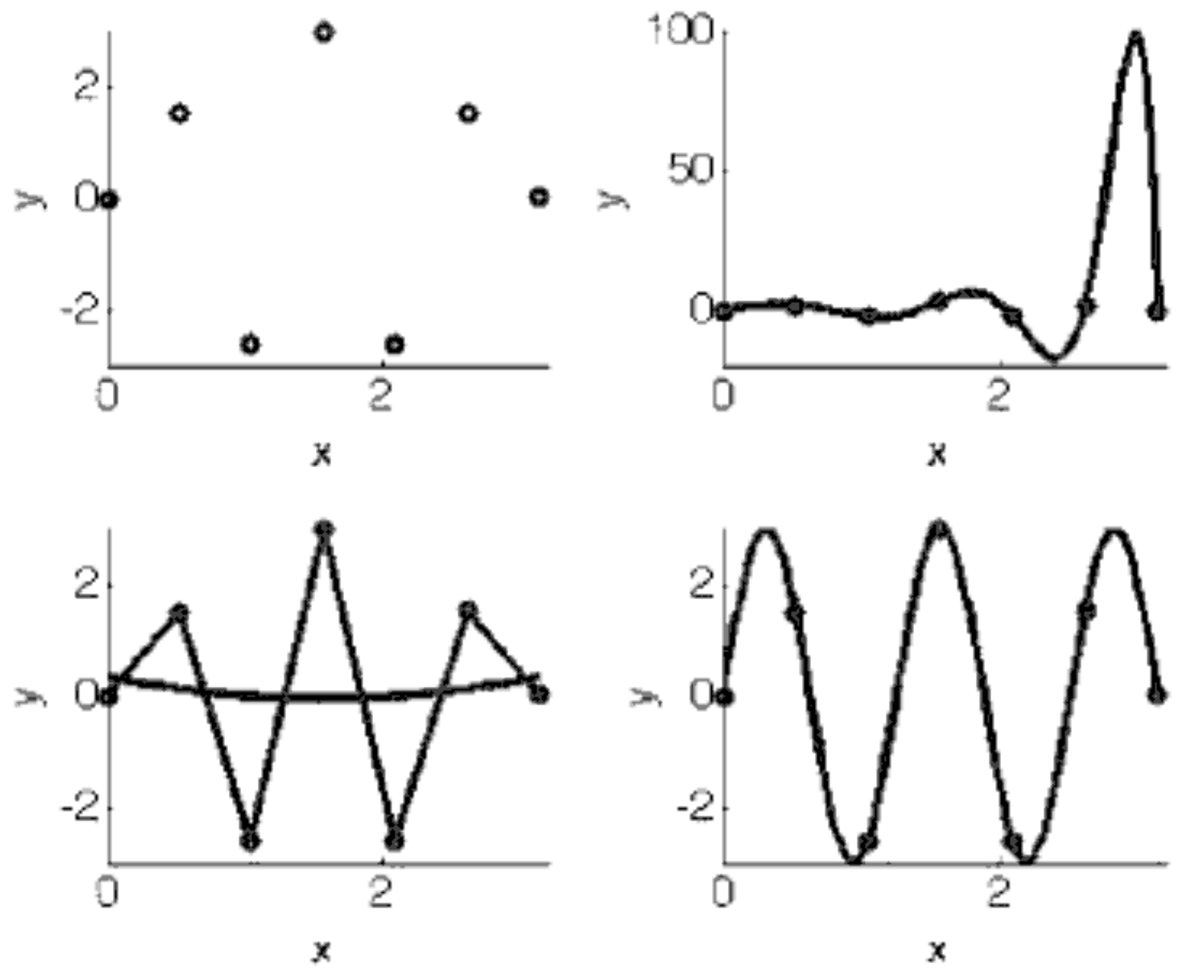


FIGURE 1.3: *Top left:* A few data points from a sample problem. *Bottom left:* Two possible ways to predict the values between the known datapoints: connecting the points with straight lines, or using a cubic approximation (which in this case misses all of the points). *Top and bottom right:* Two more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

So one thing that our machine learning algorithms can do is interpolate between data points. 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

FIGURE 1.4: The New Zealand coins.

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 faster; this is known as the curse of dimensionality and will be discussed in Section 4.3), 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

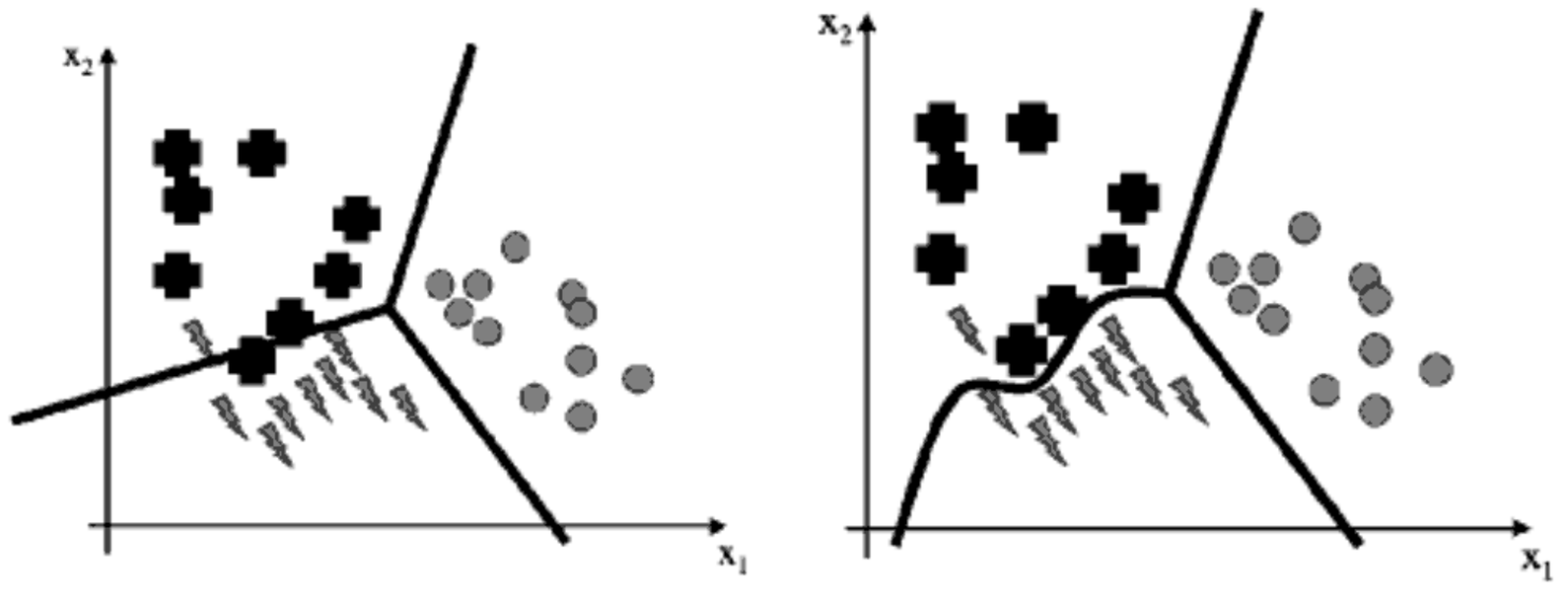


FIGURE 1.5: *Left:* A set of straight line decision boundaries for a classification problem. *Right:* An alternative set of decision boundaries that separate the plusses from the lightening strikes better, but requires a line that isn't straight.

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, it is time to return to our demonstration that learning is possible, which is the squishy thing that your skull protects.

1.5 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.

1.5.1 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.

1.5.2 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:

(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 1.6, 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

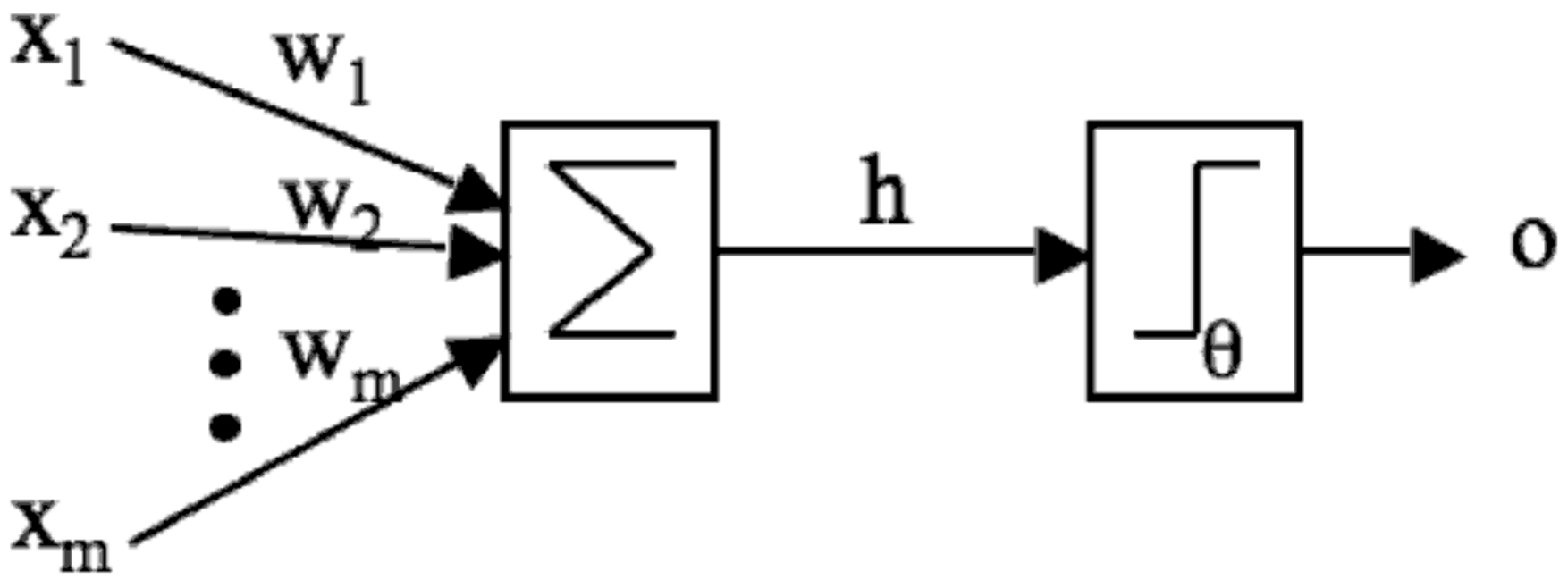


FIGURE 1.6: A picture of McCulloch and Pitt's mathematical model of a neuron. The inputs x_i are multiplied by the weights w_i, and the neurons sum their values. If this sum is greater than the threshold θ then the neuron fires, otherwise it does not.

$x_2 \times w_2$, 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

$$h = \sum_{i=1}^{m} w_i x_i, \tag{1.1}$$

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:

$$o = g(h) = \begin{cases} 1 & \text{if } h > \theta \\ 0 & \text{if } h \leq \theta. \end{cases} \tag{1.2}$$

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.

1.5.3 Limitations of the McCulloch and Pitt 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.

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.* MIT Press, Cambridge, MA, USA, 2nd edition, 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.

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.* Springer, Berlin, Germany, 2001.

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.* Wiley-Interscience, New York, USA, 2nd edition, 2001.
- Chapter 1 of S. Haykin. *Neural Networks: A Comprehensive Foundation.* Prentice-Hall, New Jersey, USA, 2nd edition, 1999.

Chapter 2

Linear Discriminants

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.

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 previous chapter 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 1.6) 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.

2.1 Preliminaries

Now is probably a good time to set up some terminology that we will use throughout the book. We've already seen a bit of it in the previous chapter. 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 much more in Section 4.1.1.

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 network. Written as $\mathbf{x}$, with elements x_i, where i runs from 1 to the number of input dimensions, m.

Weights w_{ij}, which is the weighted connection between nodes i and j. These weights are equivalent 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 $g(\cdot)$ 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 1.5.2.

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

2.2 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 2.1. 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 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 2.1 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

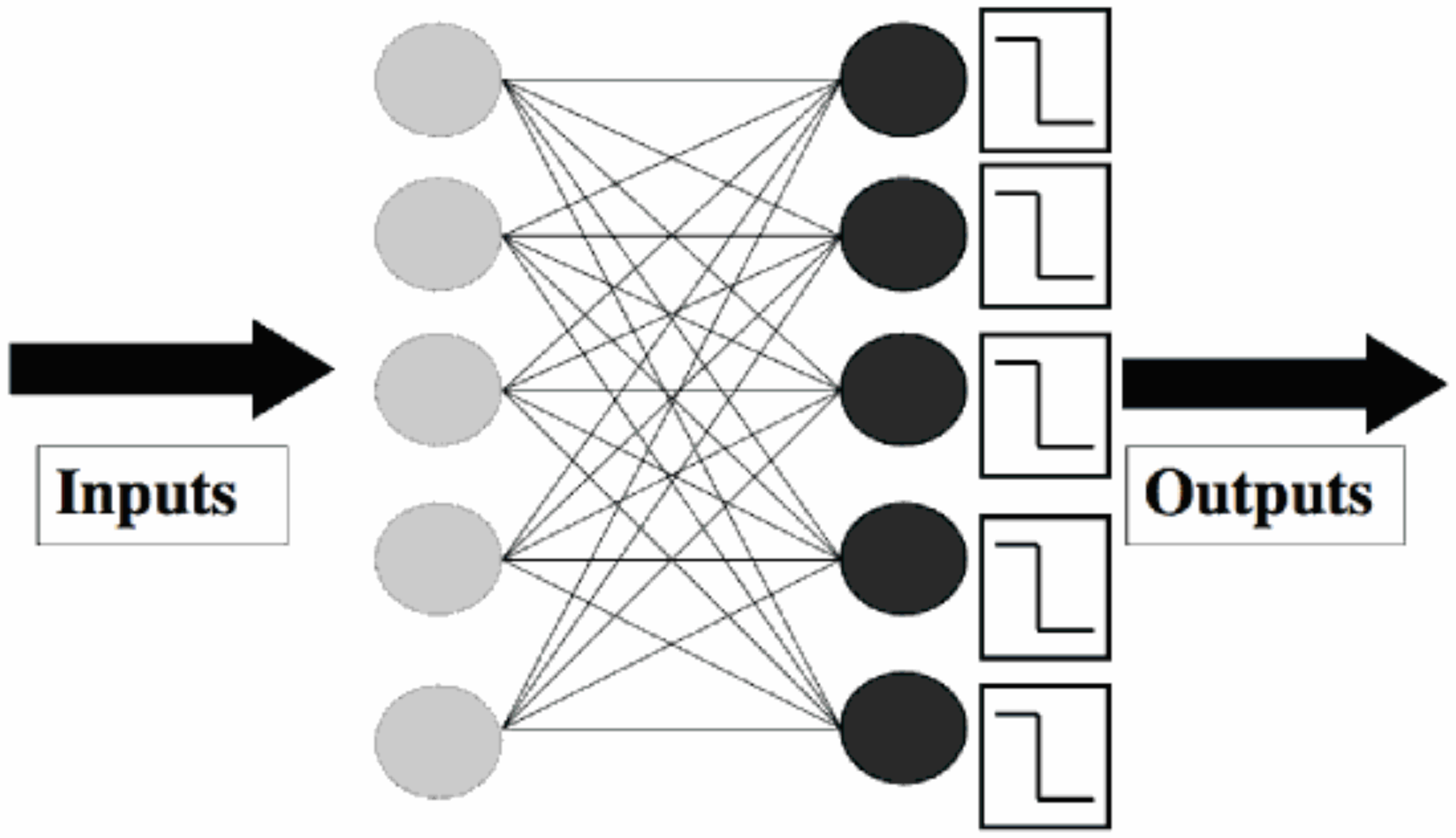


FIGURE 2.1: The Perceptron network, consisting of a set of input nodes (left) connected to McCulloch and Pitts neurons using weighted connections.

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 (1.1) and (1.2) 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 2.1, 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 3, 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 $t_k - y_k$ (the difference between the target for that neuron t_k, which is what the neuron should have done, and the output y_k, which is what the neuron did. This is a possible error function). If it is positive then the neuron should have fired and didn't, so we make the weights bigger, and vice versa if it is negative. 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: $\Delta w_{ik} = (t_k - y_k) \times x_i$, 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 2.2.2. 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}:

$$w_{ij} \leftarrow w_{ij} + \eta(t_j - y_j) \cdot x_i. \tag{2.1}$$

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 3.3.6), but for now we will predefine the maximum number of iterations, T.

2.2.1 The Learning Rate η

Equation (2.1) 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

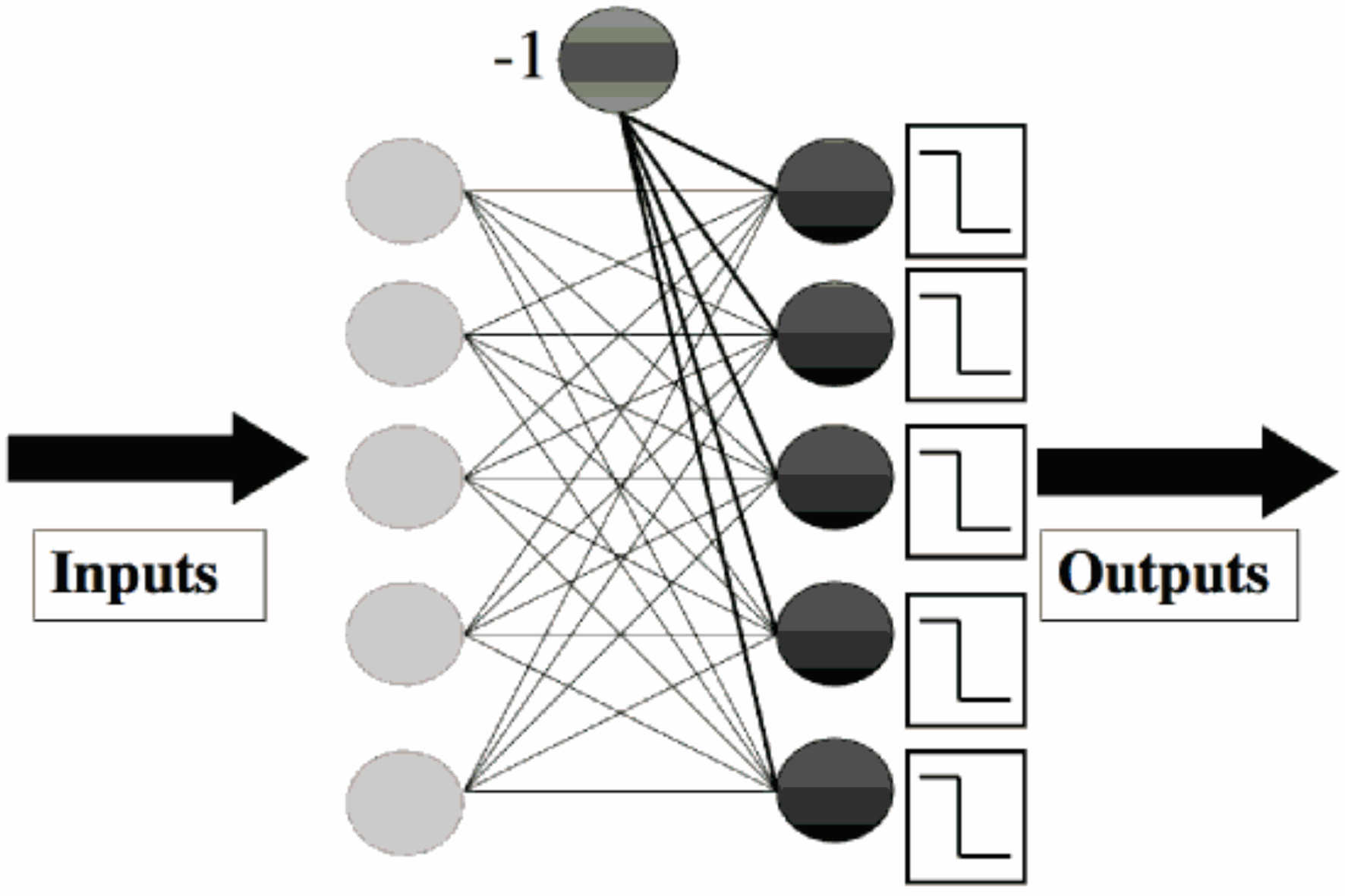


FIGURE 2.2: The Perceptron network again, showing the bias input.

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.

2.2.2 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 -1 is chosen). 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 often 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}.

2.2.3 The Perceptron Learning Algorithm

We are now ready to write our first learning algorithm. It might be useful to keep Figure 2.2 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:
    - for each input vector:
      - compute the activation of each neuron j using activation function g:

$$y_j = g\left(\sum_{i=0}^{m} w_{ij}x_i\right) = \begin{cases} 1 \text{ if } w_{ij}x_i > 0 \\ 0 \text{ if } w_{ij}x_i \leq 0 \end{cases} \tag{2.2}$$

      - update each of the weights individually using:

$$w_{ij} \leftarrow w_{ij} + \eta(t_j - y_j) \cdot x_i \tag{2.3}$$

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

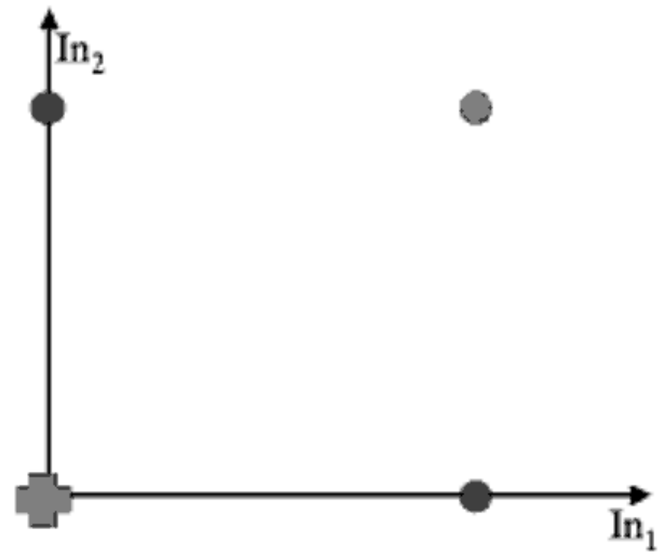


FIGURE 2.3: Data for the OR logic function and a plot of the four datapoints.

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

$$y_j = g\left(\sum_{i=0}^{m} w_{ij}x_i\right) = \begin{cases} 1 \text{ if } w_{ij}x_i > 0 \\ 0 \text{ if } w_{ij}x_i \leq 0 \end{cases} \tag{2.4}$$

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

2.2.4 An Example of Perceptron Learning

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 2.3; 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 2.4.

As you can see from Figure 2.4, 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,

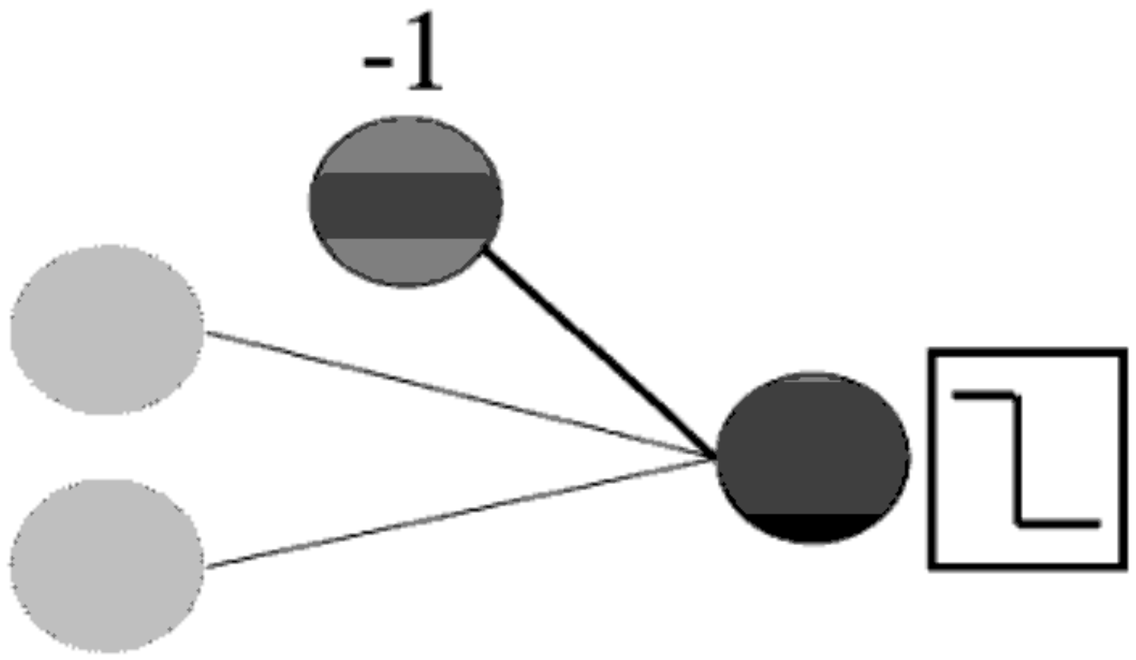


FIGURE 2.4: The Perceptron network for the example in Section 2.2.4.

so the value that reaches the neuron is $-0.05\times-1+-0.02\times0+-0.02\times0 = 0.05$. 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 (2.1) to each of the weights separately (we'll pick a value of $\eta = 0.25$ for the example):

$$w_0 : -0.05 + 0.25 \times (0 - 1) \times -1 = 0.2, \tag{2.5}$$
$$w_1 : -0.02 + 0.25 \times (0 - 1) \times 0 = -0.02, \tag{2.6}$$
$$w_2 : 0.02 + 0.25 \times (0 - 1) \times 0 = 0.02. \tag{2.7}$$

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:

$$w_0 : 0.2 + 0.25 \times (1 - 0) \times -1 = -0.05, \tag{2.8}$$
$$w_1 : -0.02 + 0.25 \times (1 - 0) \times 0 = -0.02, \tag{2.9}$$
$$w_2 : 0.02 + 0.25 \times (1 - 0) \times 1 = 0.27. \tag{2.10}$$

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.

2.2.5 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 (Chapter 16 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:

```
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 `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:

$$\begin{pmatrix} 3 & 4 & 5 \\ 2 & 3 & 4 \end{pmatrix} \times \begin{pmatrix} 1 & 3 \\ 2 & 4 \\ 3 & 5 \end{pmatrix} \tag{2.11}$$

$$= \begin{pmatrix} 3\times1+4\times2+5\times3 & 3\times3+4\times4+5\times5 \\ 2\times1+3\times2+4\times3 & 2\times3+3\times4+4\times5 \end{pmatrix} \tag{2.12}$$

$$= \begin{pmatrix} 26 & 50 \\ 20 & 38 \end{pmatrix} \tag{2.13}$$

NumPy can do this multiplication for us, using the `dot()` function (which is a rather strange name mathematically, but never mind). So to reproduce the calculation above, we use:

```
>>> from numpy import *
>>> a = array([[3,4,5],[2,3,4]])
>>> b = array([[1,3],[2,4],[3,5]])
```

```
>>> dot(a,b)
array([[26, 50],
       [20, 38]])
```

The `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 `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,

```
>>> 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 as:

```
activations = dot(inputs,weights)
activations = 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 `dot(inputs,targets - activations)` we need to turn the `inputs` matrix around so that it is $m \times N$. This is done using the `transpose()` function, which swaps the rows and columns over (so using matrix `a` above again) we get:

```
>>> 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):

```
weights += eta*dot(transpose(inputs),targets-activations)
```

Assuming that you make sure in advance that all your input matrices are the correct size (the `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 `concatenate()` function, making an 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 = concatenate((-ones((nData,1)),inputs),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 3.2.2. 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 = 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`. What is interesting is to see the code working, and that is what we will do next, 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 printout 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).

```
>>> from numpy import *
>>> inputs = array([[0,0],[0,1],[1,0],[1,1]])
>>> targets = 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
[[-0.22546505]
 [ 0.03035344]
 [ 0.2684798 ]]
Iteration:  1
[[ 0.02453495]
 [ 0.03035344]
 [ 0.2684798 ]]
Iteration:  2
[[ 0.02453495]
 [ 0.03035344]
 [ 0.2684798 ]]
Iteration:  3
[[ 0.02453495]
 [ 0.03035344]
 [ 0.2684798 ]]
Iteration:  4
[[ 0.02453495]
 [ 0.03035344]
 [ 0.2684798 ]]
Iteration:  5
[[ 0.02453495]
 [ 0.03035344]
 [ 0.2684798 ]]
 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 2.5 shows

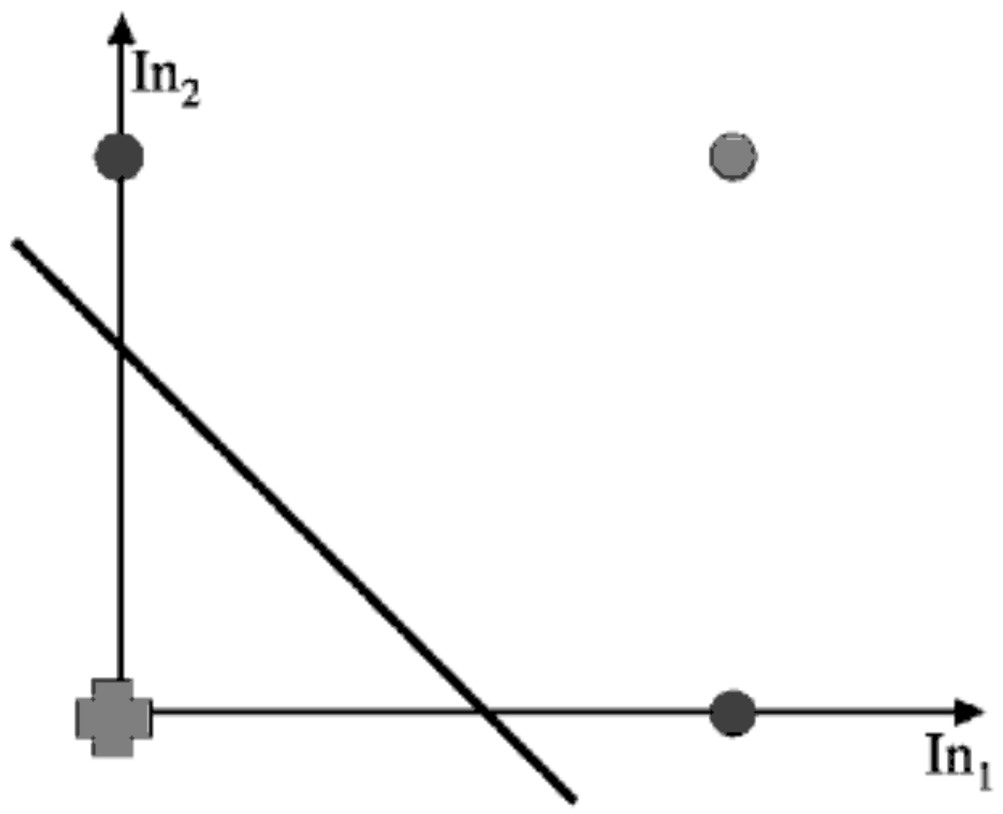


FIGURE 2.5: The decision boundary computed by a Perceptron for the OR function.

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 2.3.

2.2.6 Testing the Network

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:

```
>>> inputs_bias = concatenate((-ones((shape(inputs)[0],1)),
inputs),axis=1)
>>> pcn.pcnfwd(inputs_bias,weights)
```

This brings us to an interesting question, which is how do you decide whether or not the network has learnt well? The first thing that we can do is look at the error on the training set. We asked the Perceptron to learn about the OR data, and it got the predictions 100% correct. However, we want a neural network to generalise to examples that it has not seen in the training set, and we can't test this by using the training set. So we need some different data, a test set to test it on as well. This isn't very easy for this example, but for real datasets, you separate the data into a training set and a separate test set. This will be covered in more detail in Section 3.3.5.

Regardless of what data we use to test the network, 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. It is a nice simple idea, which is to make a square matrix that contains all

the possible classes in both the horizontal and vertical directions. We list the classes along the top of a table as the 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 into class i in the targets, but class j by the network. Anything on the leading diagonal 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. Writing the code to compute this is not too difficult, and for a small number of classes, it 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.

We've 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.

2.3 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 2.3, you should be able to draw a straight line that separates out the crosses from the circles without difficulty (it is done in Figure 2.5). 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 2.6.

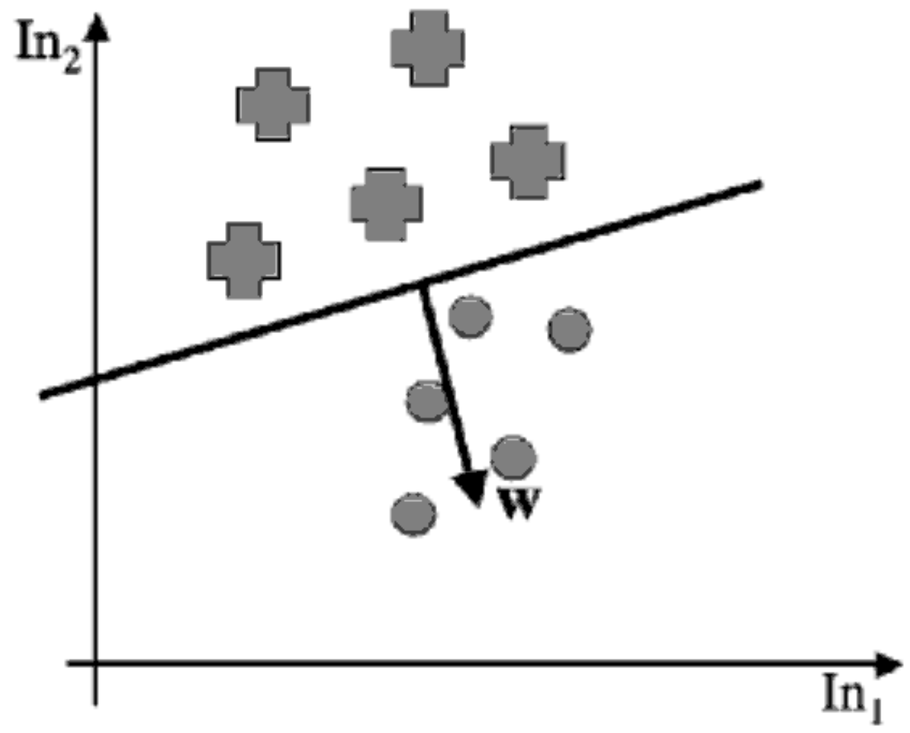


FIGURE 2.6: A decision boundary separating two classes of data.

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 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 `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:

$$\mathbf{x}_1 \cdot \mathbf{w}^T = \mathbf{x}_2 \cdot \mathbf{w}^T \tag{2.14}$$

$$\Rightarrow (\mathbf{x}_1 - \mathbf{x}_2) \cdot \mathbf{w}^T = 0. \tag{2.15}$$

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 2.6.

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

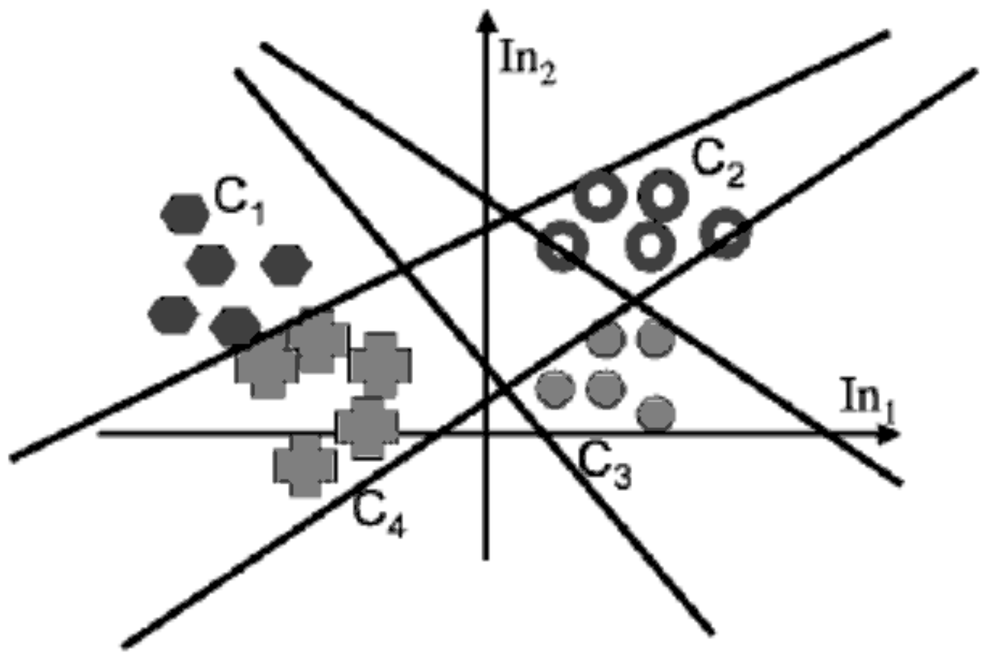


FIGURE 2.7: Different decision boundaries computed by a Perceptron with four neurons.

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

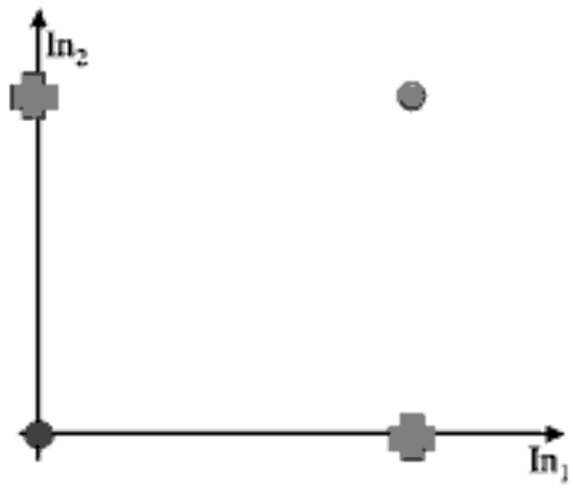


FIGURE 2.8: Data for the XOR logic function and a plot of the four datapoints.

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 2.7 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.

2.3.1 The Exclusive Or (XOR) Function

The XOR has the same four input points as the OR function, but looking at Figure 2.8, 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 = 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.

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

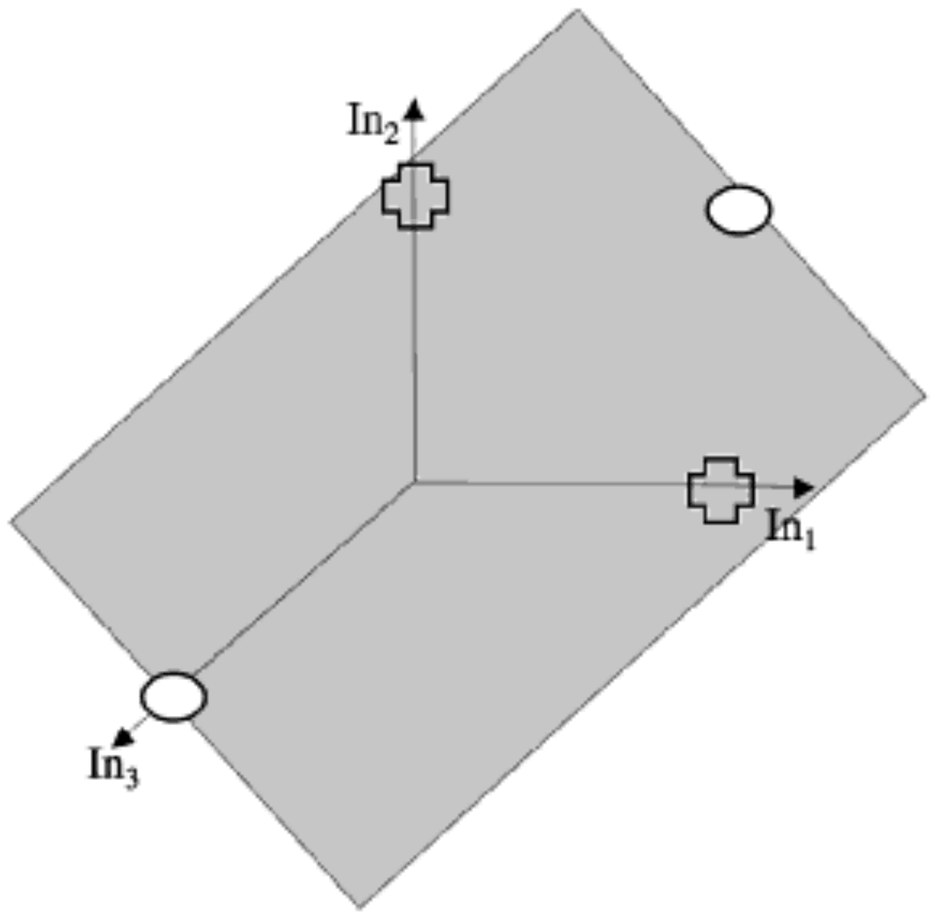


FIGURE 2.9: A decision boundary (the shaded plane) solving the XOR problem in 3D with the crosses below the surface and the circles above it.

2.3.2 A Useful Insight

From the discussion in Section 2.3.1 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 2.9. 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 dimension. So the truth table for the function is the one shown on the left side of Figure 2.9 (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 = 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 func-

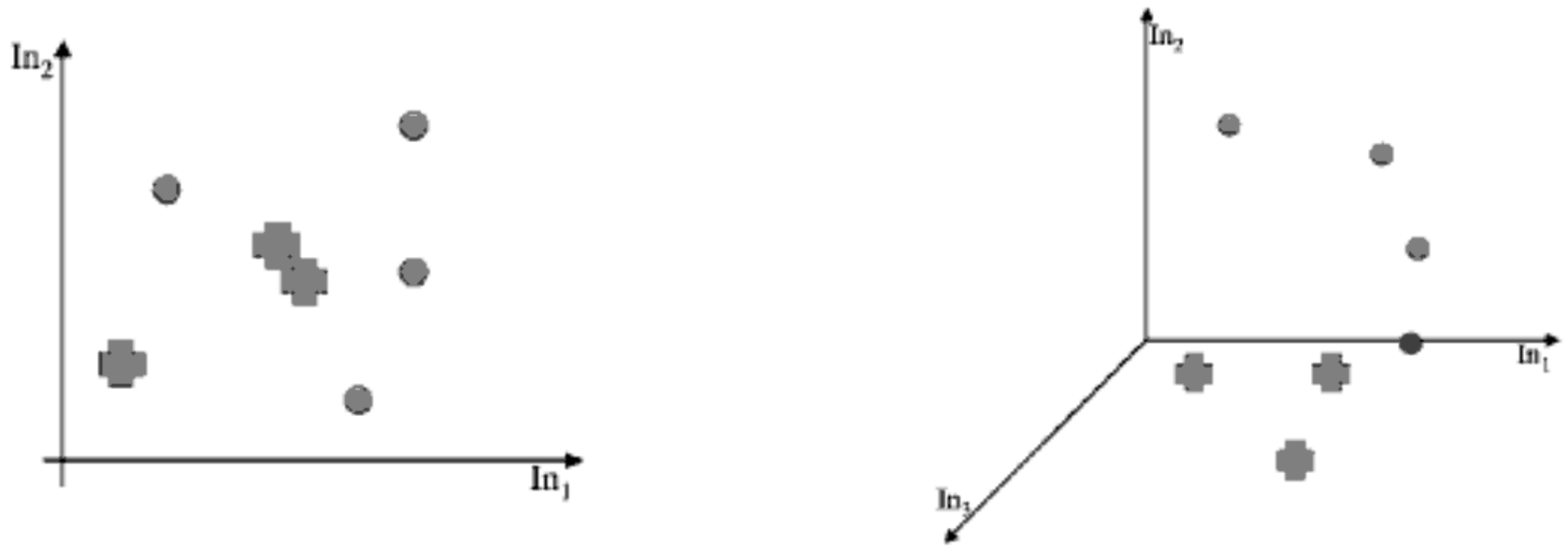


FIGURE 2.10: *Left:* Non-separable 2D dataset. *Right:* The same dataset with third coordinate $x_1 \times x_2$, which makes it separable.

tion, provided that you project the data into the correct set of dimensions. There are 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 5. 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 2.10 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 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 2.4, 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.

2.3.3 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:

```
import os
from pylab import *
from numpy import *
import pcn

os.chdir('~/Book/Datasets/pima')
pima = loadtxt('pima-indians-diabetes.data',delimiter=',')
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 `loadtxt()` command the `delimiter` specifies which character is used to separate out the datapoints. Note that PyLab is imported before NumPy. This should not matter, in general, but there are some commands in PyLab that overwrite some in NumPy, and we want to use the NumPy ones, so you need to import them in that order. The `shape()` method tells that there are 768 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 `where` command. Once you have worked that out, you will be able to plot them with different shapes and colours. The `plot` command is in Matplotlib, so you'll need to import that (as `pylab`) 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:

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

to plot the first two dimensions as green circles and red crosses, which (up to colour, of course) should look like Figure 2.11. The `ion()` command ensures that the data is actually plotted, while the `show()` command is only required if you are using Eclipse, and ensures that the graph does not vanish when

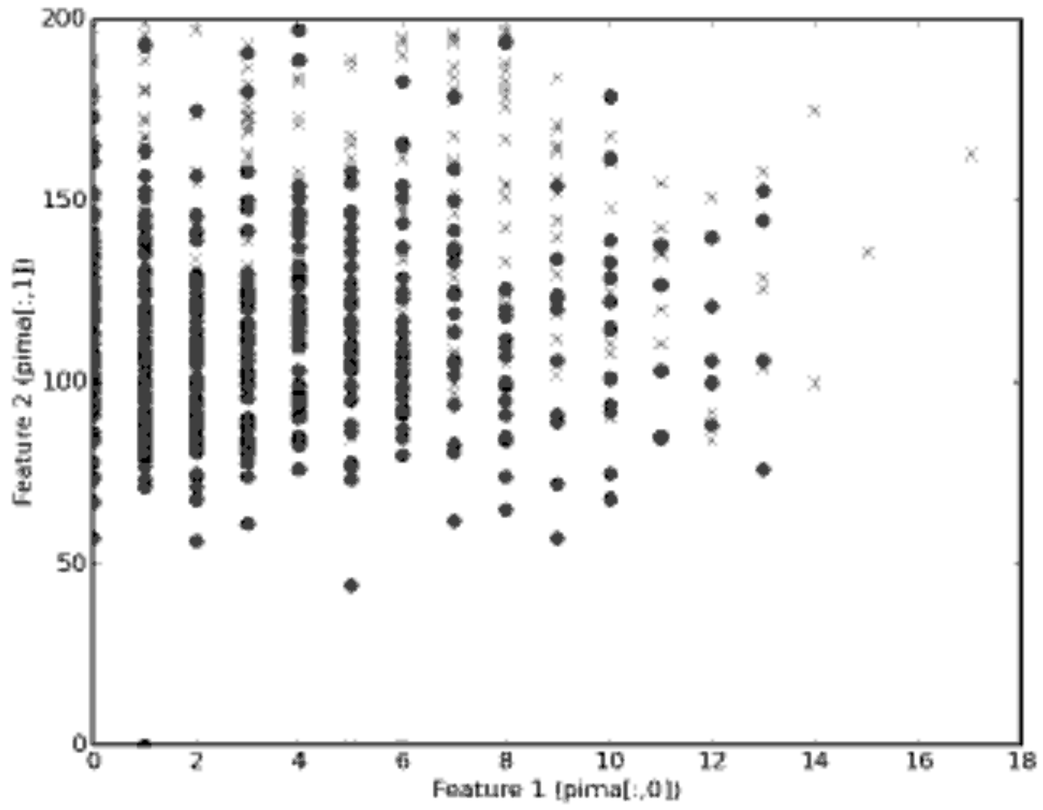


FIGURE 2.11: Plot of the first two dimensions of the Pima Indians dataset showing the two classes as 'x' and 'o'.

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. We will talk about this more in Section 3.3.5, 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. The first thing to do is to have a proper look at some of the data. For example, 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 `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[where(pima[:,0]>8),0] = 8

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

There is another thing that can improve the results markedly, which is to normalise the data, sometimes also known as standardisation. We will look at this more in Section 3.3.1, but the basic idea is to ensure that the values in the data are not too large, because then the weights will have to be very small. The most common method of normalisation is to subtract off the mean of each variable (so that they each have zero mean) and divide by the variance. This is very easy in NumPy once you have worked out which `axis` is which: `axis=0` sums down the columns and `axis=1` sums across the rows. Note that only the input variables are normalised here. 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.

```
pima[:,:8] = pima[:,:8]-pima[:,:8].mean(axis=0)
pima[:,:8] = pima[:,:8]/pima[:,:8].var(axis=0)
```

There is one thing to be careful of here, 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, if 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.

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 8.2.2.

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.

2.4 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 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 2.12 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 Pythagorus' theorem to know the distance. Now, we can try to minimise an error function that measures the sum of all these distances. If we

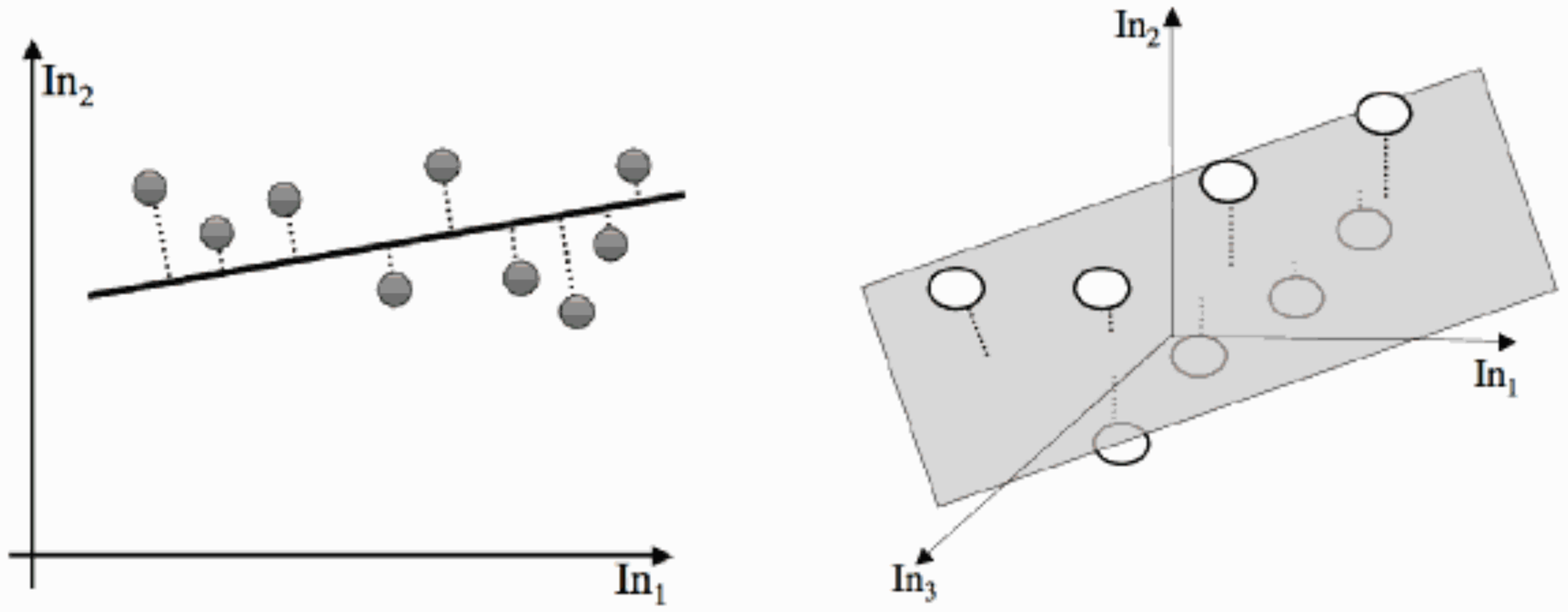


FIGURE 2.12: Linear regression in two and three dimensions.

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:

$$\sum_{j=0}^{N} \left(t_j - \sum_{i=0}^{M} \beta_i x_{ij} \right)^2 . \tag{2.16}$$

This can be written in matrix form as:

$$(\mathbf{t} - \mathbf{X}\boldsymbol{\beta})(\mathbf{t} - \mathbf{X}\boldsymbol{\beta})^T, \tag{2.17}$$

where $\mathbf{t}$ is 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 parameter vector $\boldsymbol{\beta}$ and setting the derivative to 0, which means that $\mathbf{X}^T(\mathbf{t}-\mathbf{X}\boldsymbol{\beta}) = 0$, which has the solution $\boldsymbol{\beta} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$ (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 `linalg.inv()` function that is available in NumPy. In fact, the entire function can be written as:

```
from numpy import *

def linreg(inputs,targets):

    inputs = concatenate((-ones((shape(inputs)[0],1)),inputs),
```

```
axis=1)
beta = dot(dot(linalg.inv(dot(transpose(inputs),inputs)),
transpose(inputs)),targets)

outputs = dot(inputs,beta)
```

2.4.1 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. 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 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 `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 `loadtxt` that they are comments, using:

```
auto = 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 $\boldsymbol{\beta}$ 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 = concatenate((-ones((shape(testin)[0],1)),testin),
axis=1)
testout = dot(testin,beta)
error = 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 10.1 once we have built up the understanding we need.

Further Reading

If you are interested in the historical aspects of the field, 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*. Wiley-Interscience, New York, USA, 2nd edition, 2001.
- Sections 3.1–3.3 of T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*. Springer, Berlin, Germany, 2001.

Practice Questions

Problem 2.1 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 2.2 Work out the Perceptrons that construct logical NOT, NAND, and NOR of their inputs.

Problem 2.3 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 2.4 Test out both the Perceptron and linear regressor code from the website on the parity problem.

Problem 2.5 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 2.6 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 2.7 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 2.8 For the prostate data available via the website, use both the Perceptron and logistic regressor and compare the results.

Chapter 3

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 2.3 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 5, 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 backward 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 3.1.

We will think about why adding extra layers of nodes makes a neural network more powerful in Section 3.3.3, 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 3.2. 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 3.2) 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

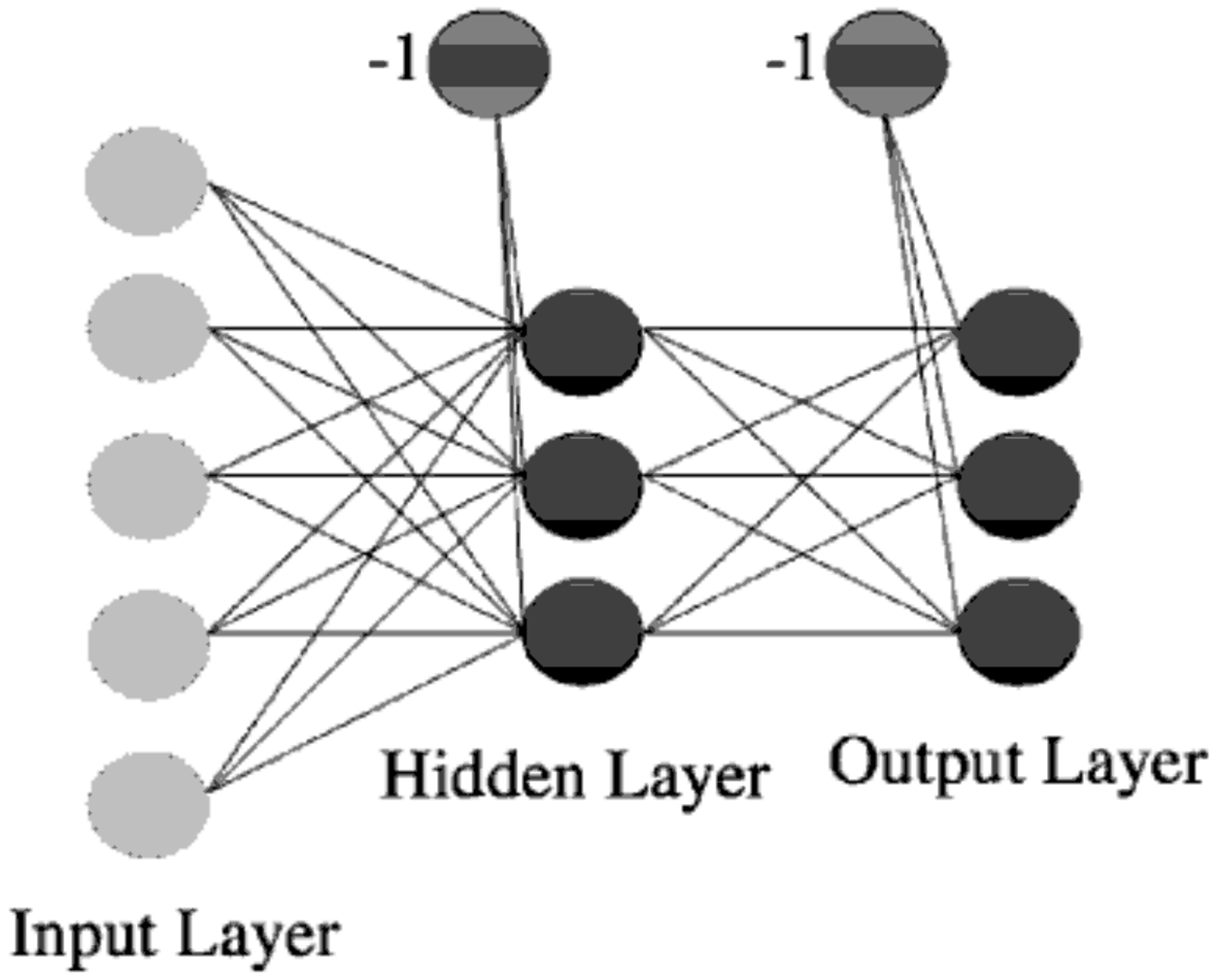


FIGURE 3.1: The Multi-Layer Perceptron network, consisting of multiple layers of connected neurons.

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 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. 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.

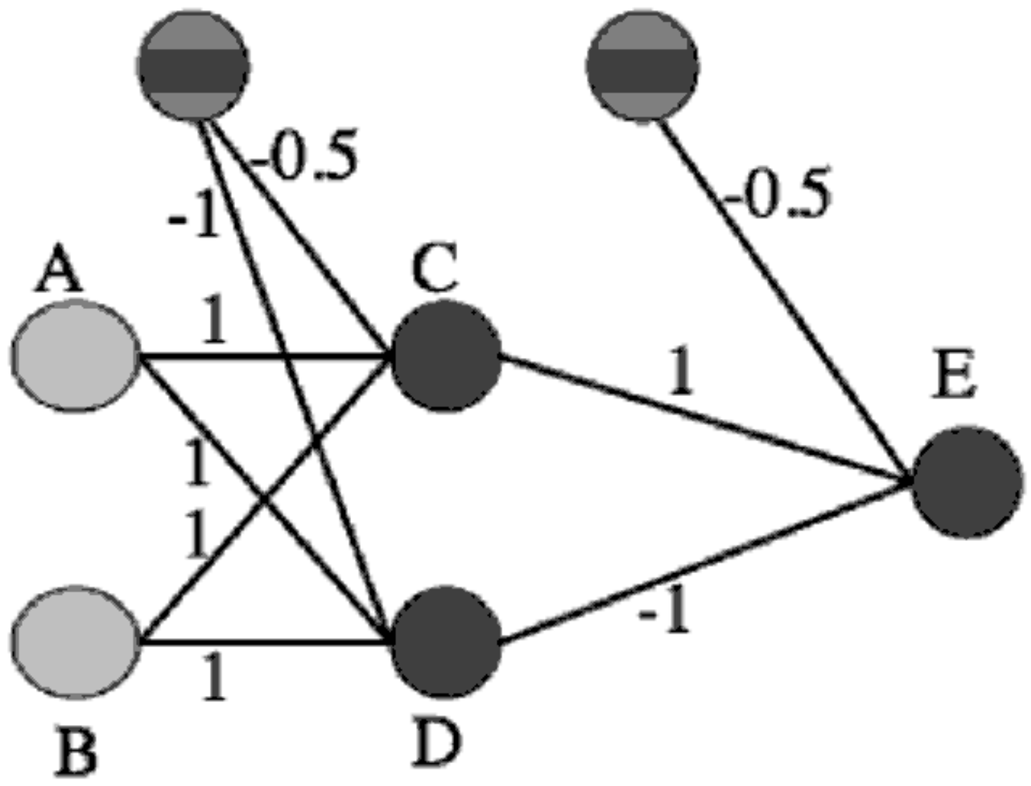


FIGURE 3.2: A Multi-Layer Perceptron network showing a set of weights that solve the XOR problem.

3.1 Going Forwards

Just like in 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 3.3.3). 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 3.1, 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.

3.1.1 Biases

Just like in the Perceptron case, we need to include a bias input to each neuron. We do this in the same way, 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.

3.2 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 11).

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 3.6. 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 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. Although we didn't say it like this then, what we did was to choose an error function $E = t - y$, 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).

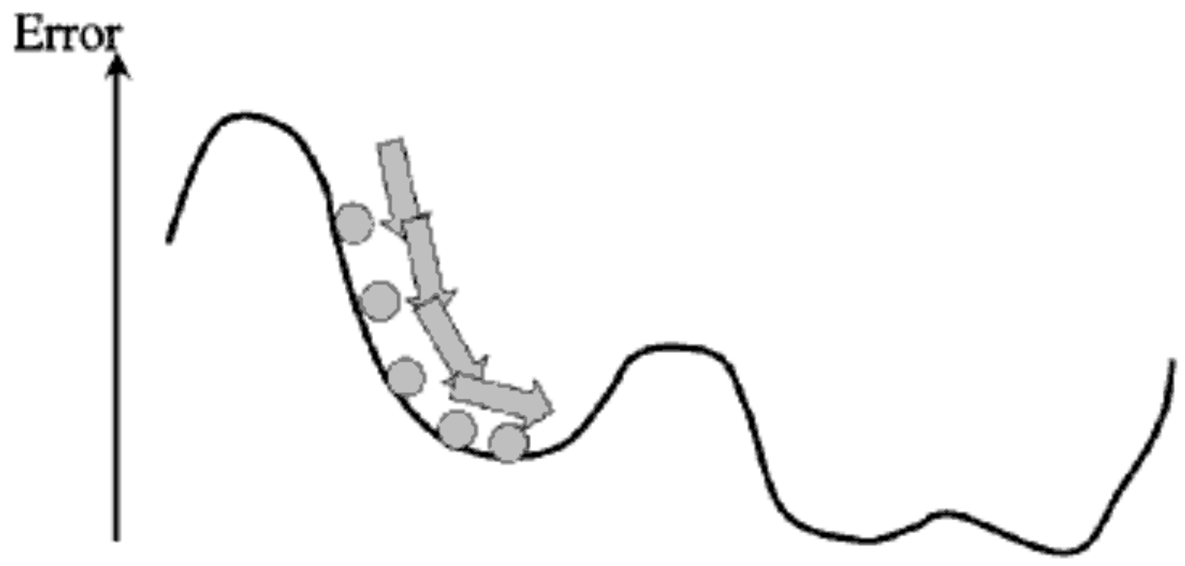


FIGURE 3.3: The weights of the network are trained so that the error goes downhill until it reaches a local minimum, just like a ball rolling under gravity.

The error function that we used for the Perceptron was $E = t - y$. 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 algorithm thinks 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 t and y for each node, squares them, and adds them all together:

$$E(\mathbf{t}, \mathbf{y}) = \frac{1}{2} \sum_{k=1}^{n} (t_k - y_k)^2. \tag{3.1}$$

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 like the line in Figure 3.3. 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

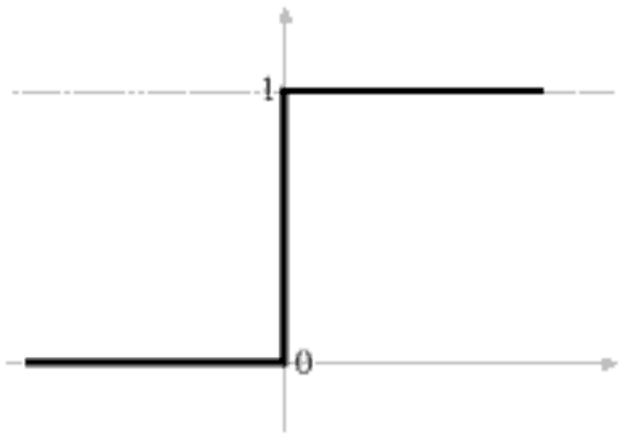


FIGURE 3.4: The threshold function that we used for the Perceptron. Note the discontinuity where the value changes from 0 to 1.

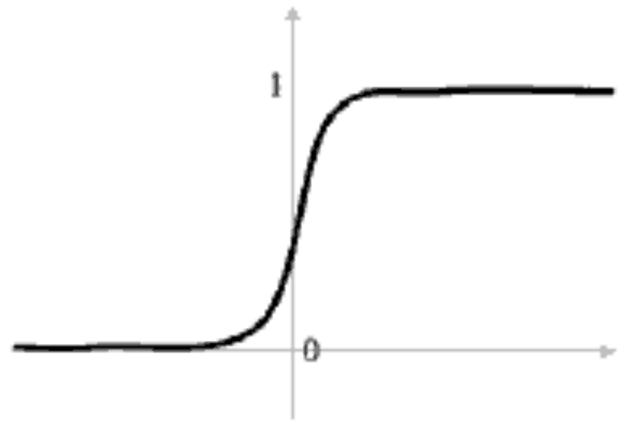


FIGURE 3.5: The sigmoid function, which looks qualitively fairly similar, but varies smoothly and differentiably.

our neurons so far, which is that it is discontinuous (see Figure 3.4; 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 3.4) then it looks kind of S-shaped. There is a mathematical form of S-shaped functions, called sigmoid functions (see Figure 3.5). They have another nice property, which is that their derivative also has a nice form, as is shown in Section 3.6.3 for those who know some mathematics. The most commonly used form of this function (where β is some positive parameter) is:

$$a = g(h) = \frac{1}{1 + \exp(-\beta h)}. \tag{3.2}$$

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

$$a = g(h) = \tanh(h) = \frac{\exp(h) - \exp(-h)}{\exp(h) + \exp^{(}-h)}, \tag{3.3}$$

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 targets and the outputs (Equation (3.1) 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 3.6, 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 sets of weights, and we just apply these backwards through the network, starting at the outputs and ending up back at the inputs.

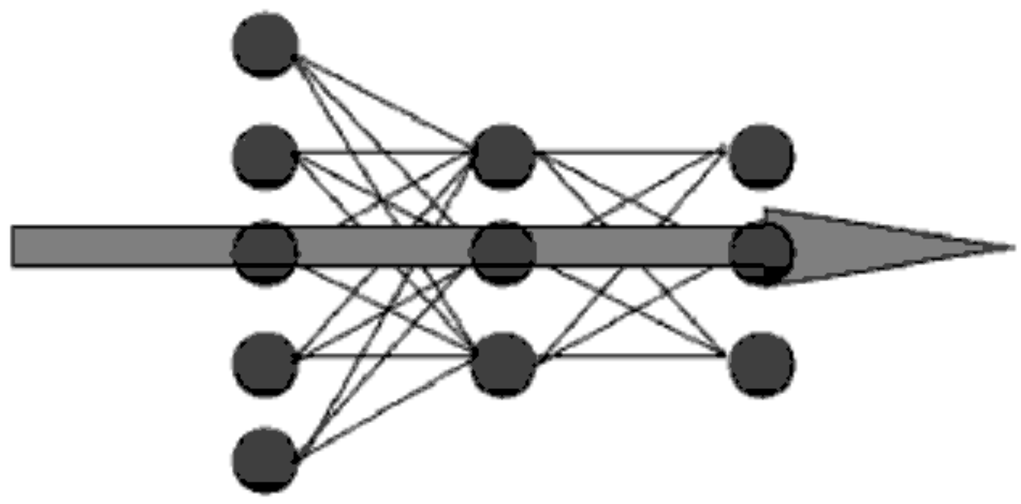


FIGURE 3.6: The forward direction in a Multi-Layer Perceptron.

3.2.1 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. 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. 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 3.6)
   - the inputs and the first-layer weights (here labelled as v) are used to decided whether the hidden nodes fire or not. The activation function $g(\cdot)$ is the sigmoid function given in Equation (3.2) 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 by using the δ_o errors
   - and then afterwards, the first-layer weights by using the δ_h errors

---

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:

$$h_j = \sum_i x_i v_{ij} \tag{3.4}$$

$$a_j = g(h_j) = \frac{1}{1 + \exp(-\beta h_j)} \tag{3.5}$$

      - work through the network until you get to the output layers, which have activations:

$$h_k = \sum_j a_j w_{jk} \tag{3.6}$$

$$y_k = g(h_k) = \frac{1}{1 + \exp(-\beta h_k)} \tag{3.7}$$

      Backwards phase:

      - compute the error at the output using:

$$\delta_{ok} = (t_k - y_k)\, y_k(1 - y_k) \tag{3.8}$$

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

$$\delta_{hj} = a_j(1 - a_j) \sum_k w_{jk}\delta_{ok} \tag{3.9}$$

      - update the output layer weights using:

$$w_{jk} \leftarrow w_{jk} + \eta\delta_{ok} a_j^{\text{hidden}} \tag{3.10}$$

      - update the hidden layer weights using:

$$v_{ij} \leftarrow v_{ij} + \eta\delta_{hj} x_i \tag{3.11}$$

    - 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 3.3.6)
- **Recall**
  - use the Forwards phase in the training section above

This provides a description of the basic algorithm. There are a couple of things that weren't mentioned beforehand, including the randomisation of the order of the input vectors. It turns out that this can significantly improve the speed with which the algorithm learns. NumPy has a useful function that assists with this, `random.shuffle()`, which takes a list of numbers and reorders them. It can be used like this:

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

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 3.2.4). 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)*(dot(deltao,transpose(
self.weights2)))

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

updatew1 = eta*(dot(transpose(inputs),deltah[:,:-1]))
updatew2 = eta*(dot(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 3.3. 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):

```
from numpy import *
import mlp

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

3.2.2 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. We use random values in this range 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.

3.2.3 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 (3.2)). This does not mean that we change the hidden layer neurons, they stay exactly the same. These output nodes are not models of neurons anymore, since they don't have the characteristic fire/don't fire pattern. Even so, they can be useful, for example for 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 3.4.2. 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:

$$y_k = g(h_k) = \frac{\exp(h_k)}{\sum_{\hat{h}_k} \exp(\hat{h}_k)}. \tag{3.12}$$

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 (3.7) and (3.8). For the linear activation function the first is replaced by:

$$y_k = g(h_k) = h_k, \tag{3.13}$$

while the second is replaced by:

$$\delta_{ok} = (t_k - y_k)\,. \tag{3.14}$$

For the soft-max activation, the update equation that replaces (3.8) is (3.14), just as for the linear output. 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:

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

3.2.4 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.

3.2.5 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 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 3.3 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 3.7.

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.

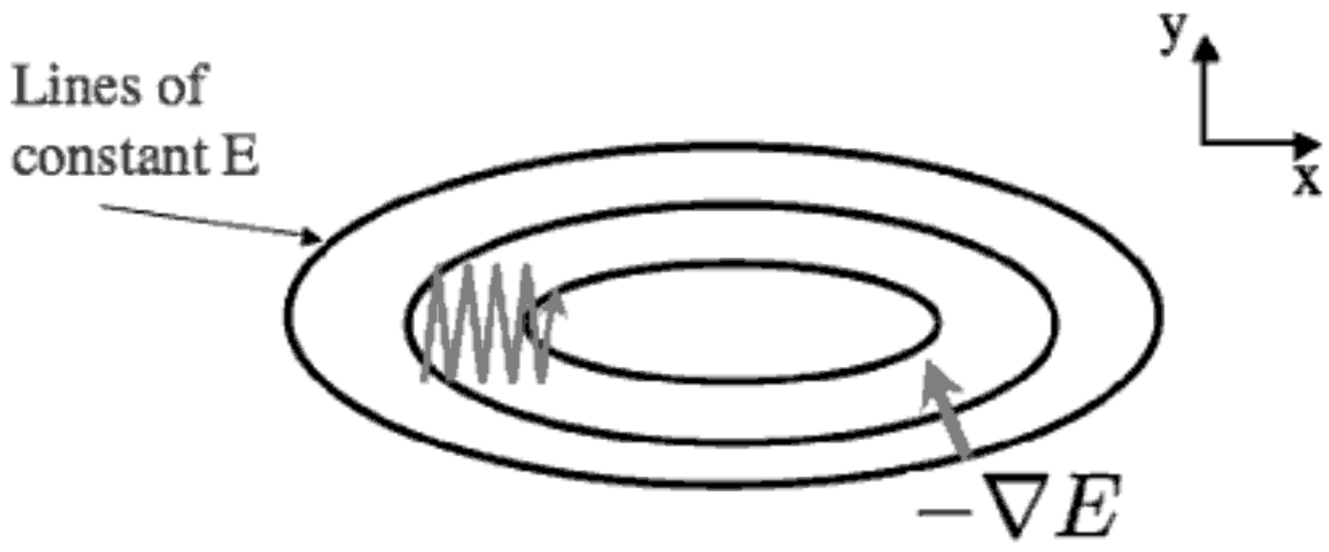


FIGURE 3.7: In 2D, downhill means at right angles to the lines of constant contour. Imagine walking down a hill with your eyes closed. If you find a direction that stays flat, then at right angles to that there may well be uphill or downhill. However, this is not the direction that takes you directly towards the local minimum.

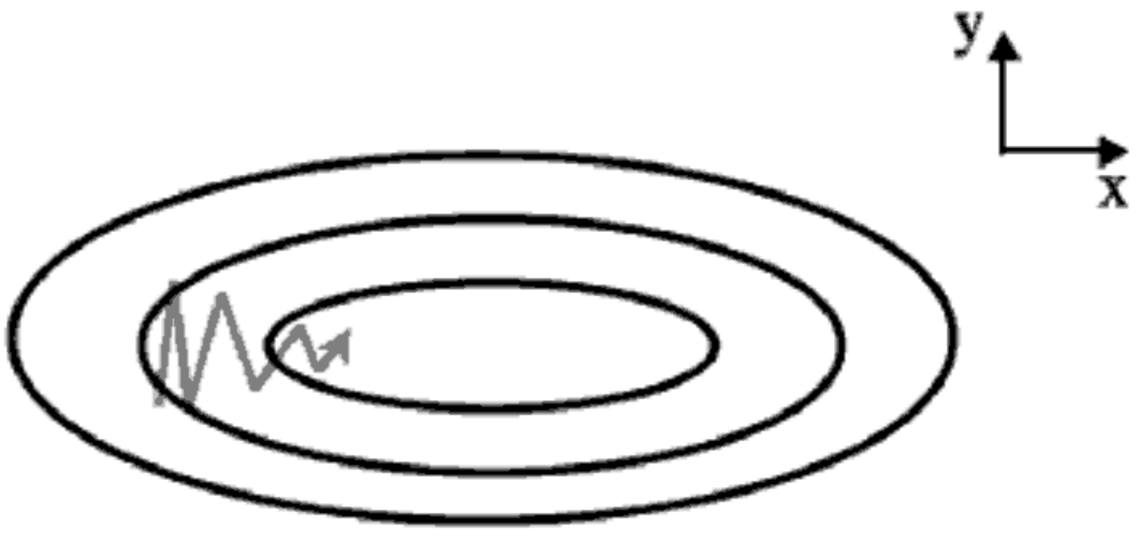


FIGURE 3.8: Adding momentum can help to avoid local minima, and also makes the dynamics of the optimisation more stable, improving convergence.

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.

3.2.6 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 3.8.

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 (3.10) and (3.11), where we need to add a second term to the weight updates so that they have the form:

$$w_{ij}^t \leftarrow w_{ij}^{t-1} + \eta\delta_o a_j^{\text{hidden}} + \alpha\Delta w_{ij}^{t-1}, \tag{3.15}$$

where t is used to indicate the current update and $t-1$ is the previous one. Δw_{ij}^{t-1} is the previous update that we made to the weights (so $\Delta w_{ij}^t = +\eta\delta_o a_j^{\text{hidden}} + \alpha\Delta w_{ij}^{t-1}$) 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*(dot(transpose(inputs),deltah[:,:-1])) +
momentum*updatew1
updatew2 = eta*(dot(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 changes the learning quite a lot, but since it makes the network simpler it can often produce improved results. Unfortunately, it isn't fail-safe: occasionally it can make the learning significantly worse, so it should be used with care. Setting the value is typically done experimentally.

3.2.7 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. Unfortunately, calculating the second derivatives is often computationally costly. These things are considered in more detail in Chapter 11.

3.3 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.

3.3.1 Data Preparation

The MLP, and indeed, pretty much every machine learning algorithm, tends to learn much more effectively if some preprocessing of the inputs and targets is performed before the network is trained. We saw some of these methods before, in Section 2.3.3, where we saw an example of using the Perceptron. For the targets it is fairly obvious that if using sigmoidal activation functions for the output, then the only possible target values should be 0 and 1. 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 data for the MLP 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. In fact, normalisation is not essential for the MLP, although it is usually beneficial. For some of the other networks that we will see, the normalisation will be essential.

In NumPy it is very easy to perform the normalisation by using the built-in `mean()` and `var()` functions; the only place where care is needed is along which axis the mean and variance are computed. For a dataset in the variable `data`, with each row being a datapoint, and targets `targets`:

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

3.3.2 Amount of Training Data

For the MLP with one hidden layer there are $(m+1) \times n + (n+1) \times p$ weights, where m, n, p 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.

3.3.3 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. For reasons that we will see shortly, two hidden layers is the most that you ever need for normal MLP learning, but the bad news for choosing the number of hidden nodes is that there is no theory to guide it. 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 3.4.

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 three layers (that is, two hidden layers 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 of this using pictures in Figure 3.9. 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 3.10. We will use this idea again when we look at approximating functions, for example using radial basis functions in Chapter 4.

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.

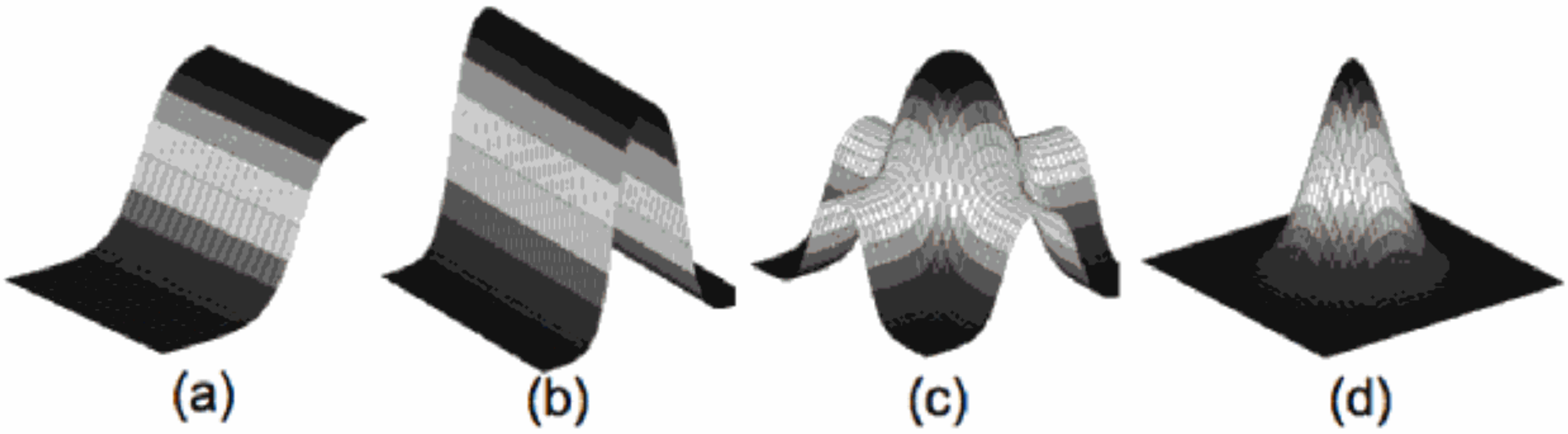


FIGURE 3.9: The learning of the MLP can be shown as (a) the output of a single sigmoidal neuron can be added to others (b), including reversed ones, to get a hill shape. Adding another hill at 90° produces (c) a bump, which can be sharpened (d) to any extent we want, with the bumps added together in the output layer. Thus the MLP learns a local representation of individual inputs.

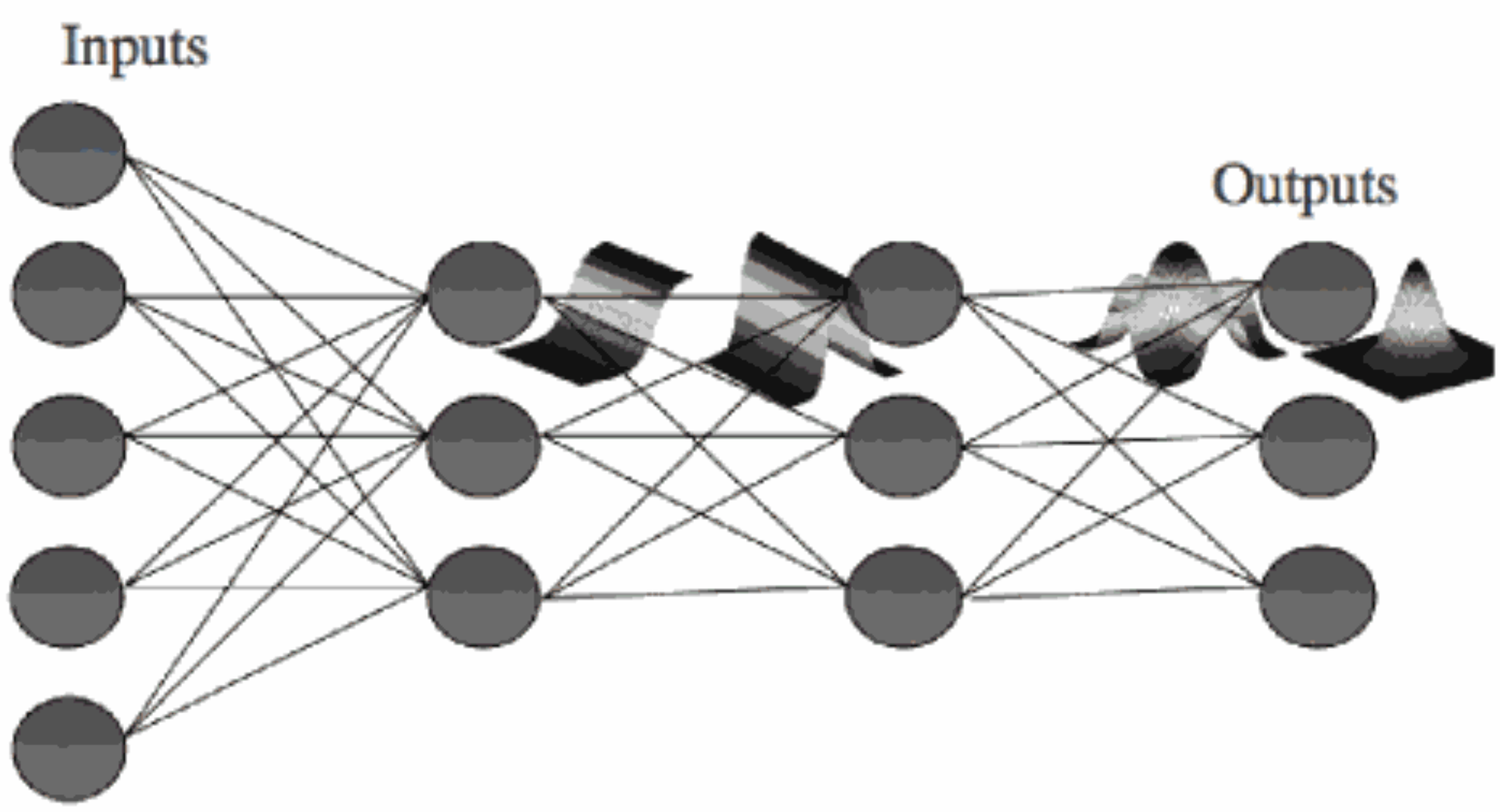


FIGURE 3.10: Schematic of the effective learning shape at each stage of the MLP.

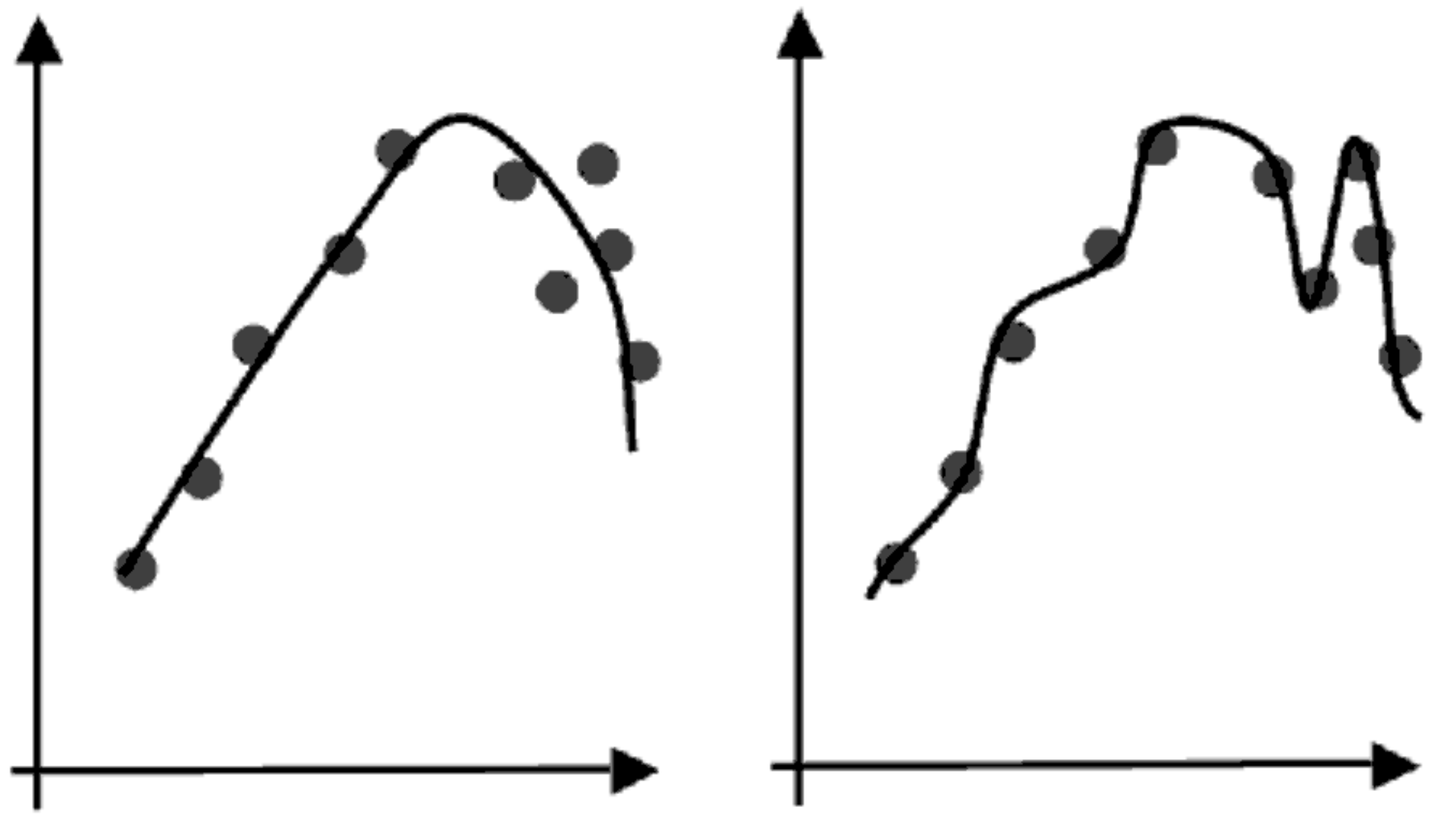


FIGURE 3.11: The effect of overfitting is that rather than finding the generating function (as shown on the left), the neural network matches the inputs perfectly, including the noise in them (on the right). This reduces the generalisation capabilities of the network.

3.3.4 Generalisation and Overfitting

The whole purpose of using a neural network is to generalise from the training examples to all possible inputs. We need to make sure that we do enough training that the network generalises well. However, there is at least as much danger in over-training the network as there is in under-training it. The number of degrees of variability in a neural network is huge — as discussed above, every weight can be varied. 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 have learnt will be much too complicated, and won't be able to generalise. Figure 3.11 shows this. Two different networks have learnt about the same data, but the one shown on the right has overfitted so that the curve goes through all of the datapoints, matching the noise as well as the underlying curve, which has been found in the graph on the left.

The solution to this problem is in two parts. The first is stopping the training before overfitting occurs (but not too early, so that the network has actually learnt something), while the second is working out when to do this, which requires thinking about something that is very important for the whole of machine learning: **testing**.

3.3.5 Training, Testing, and Validation

Whenever we train an MLP we are obviously going to test how well it works. We can compute the error of the trained network by computing the sum-of-squares error between the output and the target, just like we use to

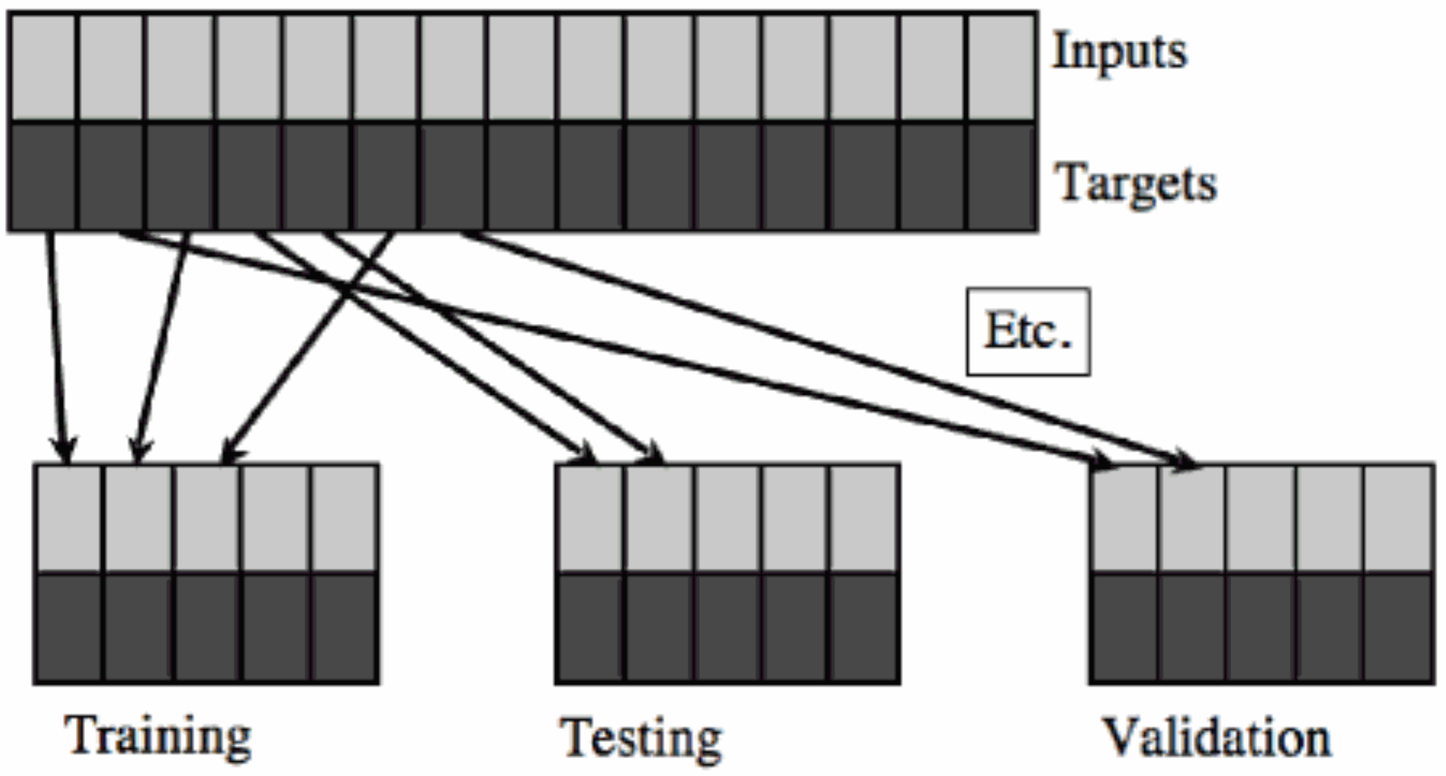


FIGURE 3.12: The dataset is split into different sets, some for training, some for validation, and some for testing.

drive the learning. What data should we test the network on? Clearly, it is not sensible to test using the same data that we trained on, since it would not tell us anything at all about how well the network generalises, nor anything about whether or not overfitting had occurred. We therefore need to keep a test set of (input, target) pairs in reserve that we don't use for training. The only problem with this is that it reduces the amount of data that we have available for testing, but that is something that we will just have to live with.

Things get more complicated when we consider checking how well the network is learning during training, so that we can decide when to stop. 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. We therefore keep a third set of data back, called the validation set because we're using it to validate the learning so far. This is known as cross-validation in statistics. 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 3.12.

If you are really short of training data, so that if you have a separate validation set there is a worry that the network won't be sufficiently trained, then it is possible to perform leave-some-out, multi-fold cross-validation. The idea is shown in Figure 3.13. The dataset is randomly partitioned into K subsets, and one subset is used as a validation set, while the neural network is trained on all of the others. A different subset is then left out and a new network is trained on that subset, repeating the same process for all of the

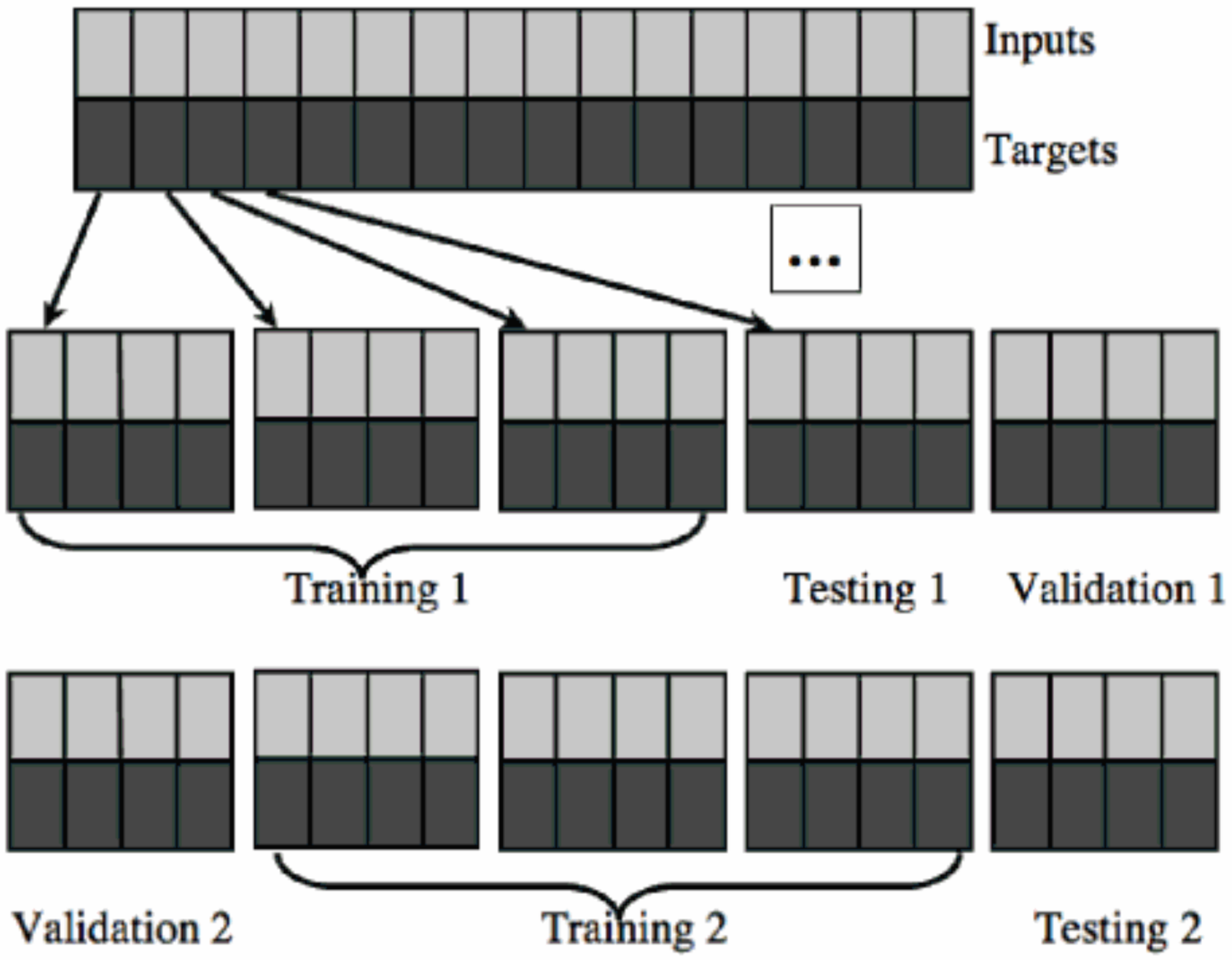


FIGURE 3.13: Leave-some-out, multi-fold cross-validation gets around the problem of data shortage by training many networks. It works by splitting the data into sets, training a network on most sets and holding one out for validation (and another for testing). Different networks are trained with different sets being held out.

different subsets. Finally, the network 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 networks instead of just one. In the most extreme case of this there is leave-one-out cross-validation, where the network is validated on just one piece of data, training on all of the rest.

3.3.6 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 we just created 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

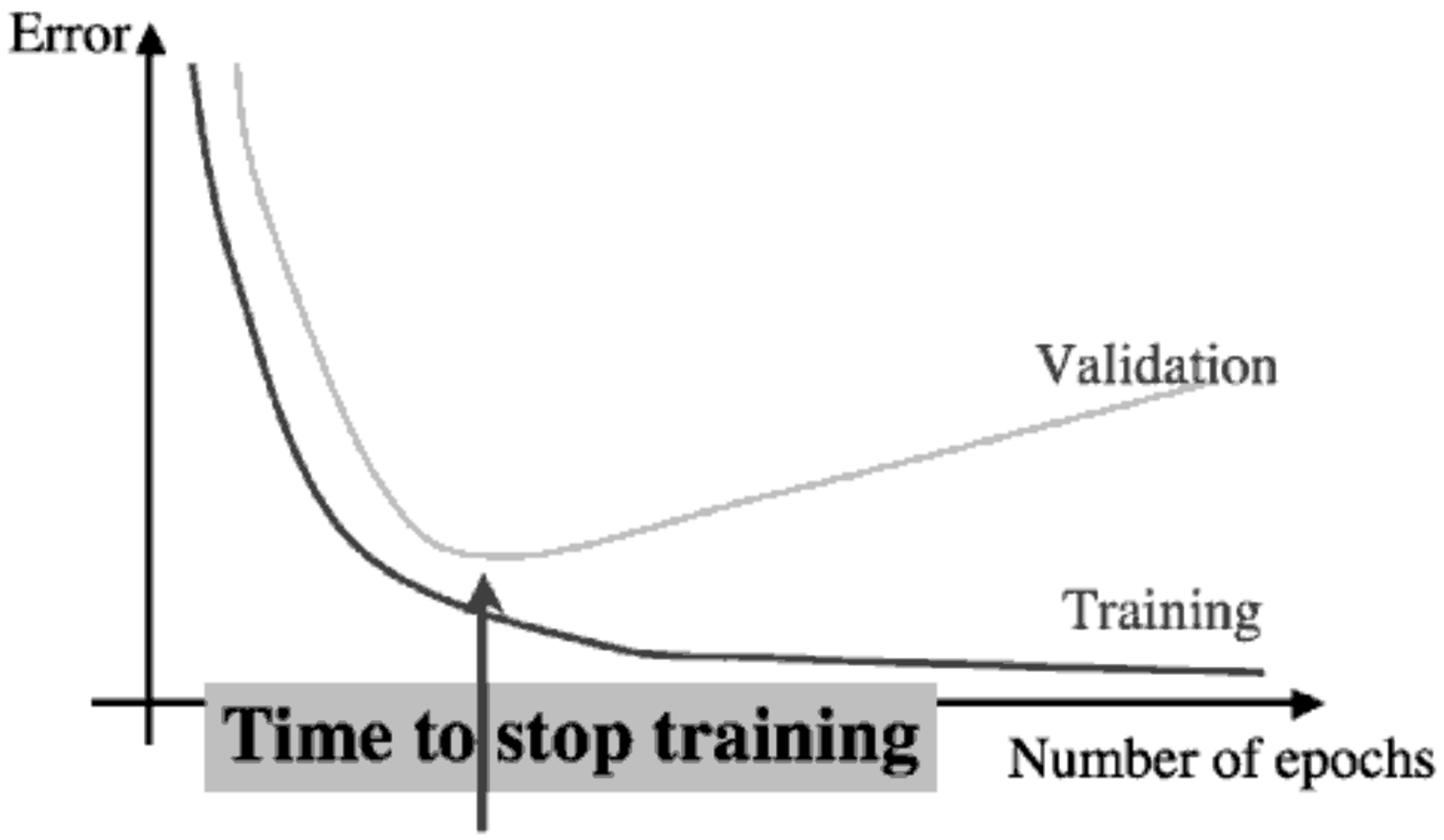


FIGURE 3.14: The effect of overfitting on the training and validation error curves, with the point at which early stopping will stop the learning marked.

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 3.14). At this stage we stop the training. This technique is called **early stopping**.

3.3.7 Computing and Evaluating the Results

We've now talked about everything that we need to run the algorithm and make it learn, but we haven't really considered how to report and analyse the results. This involves using the **test set** which has been held separate so far and running the network forward on this data. The error can then be evaluated by comparing the prediction with the target outputs. The question of what to do then depends upon the type of problem that we are solving. For a classification problem it is possible to compute the number of cases that the network predicted correctly for each class (in fact, we saw this in Section 2.2.6 with the **confusion matrix**), while for regression problems the only thing that is generally useful is the sum-of-squares error that we used to drive the training. We will see these methods being used as we look at examples.

In addition to the confusion matrix there is another way that we can evaluate the results of a classifier, which is known as an ROC curve, which stands for Receiver Operating Characteristic. This is a plot of the percentage of true positives (things corretly put into class 1) on the y axis against false positives (things incorrectly put into class 1) on the x axis. The true positive rate is sometimes known as the specificity and the false negative rate (things that were incorrectly put into class 2) is the sensitivity, so the false positive rate is 1-sensitivity. A single run of a classifier produces a single point on the ROC plot, and the closer to the top-left-hand corner it is, the better. If you were to use a fair coin to pick the class, then you would end up with a line on the diagonal from bottom-left to top-right.

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 true 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.

3.4 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.

3.4.1 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 (the reason why the `x` computation requires the `ones()` call is because of some idiosyncracies in the way that NumPy produces arrays). Make sure that you have NumPy imported first:

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

You can plot this data to see what it looks like (the results of which are shown in Figure 3.15) using:

```
>>> from pylab import *
>>> 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 3.3.1, 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. 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:

```
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*sum((validtargets-validout)**2)

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

Figure 3.16 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

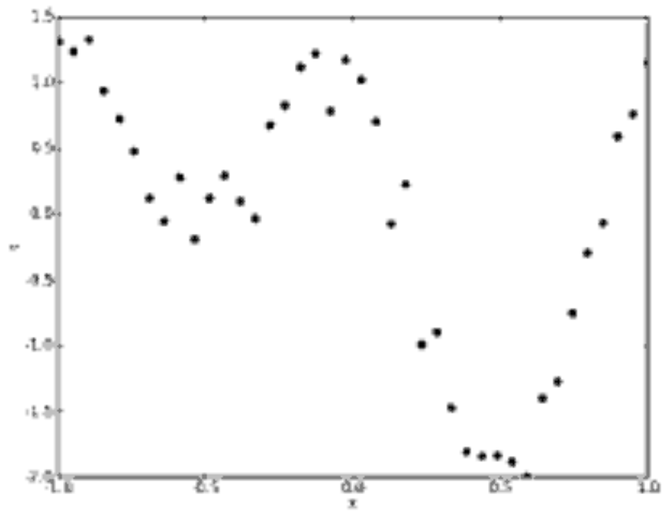


FIGURE 3.15: The data that we will learn using an MLP, consisting of some samples from a sine wave with Gaussian noise added.

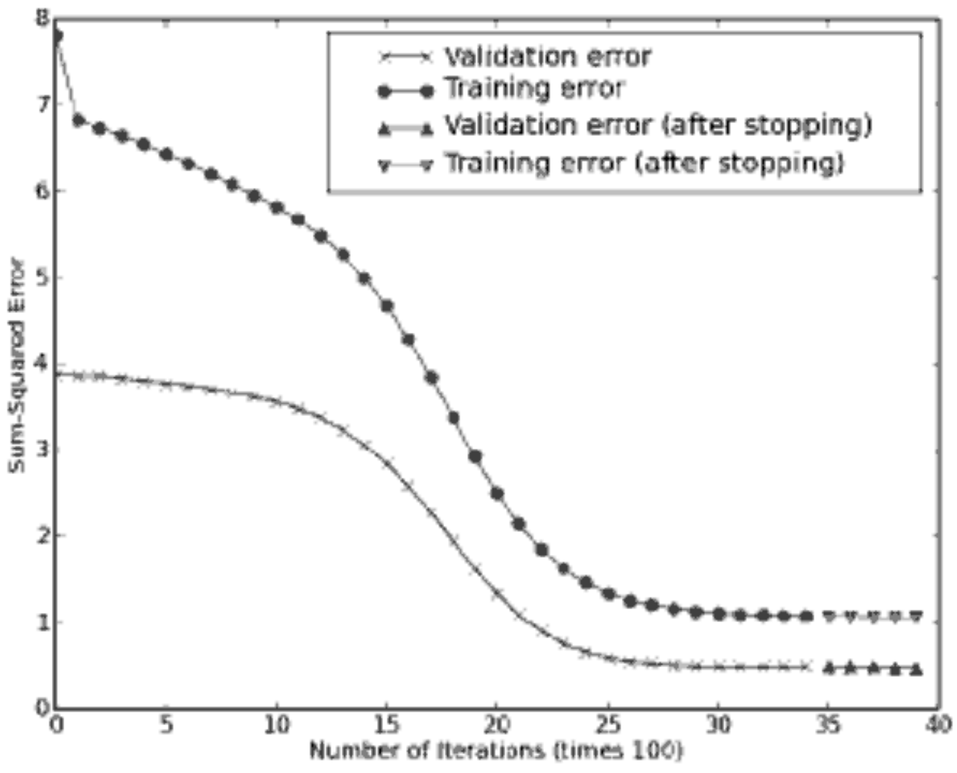


FIGURE 3.16: Plot of the error as the MLP learns (top line is total error on training set, bottom line is on validation set; the reason why it is larger on the training set is that there are more datapoints in this set). Early-stopping halts the learning at the point where there is no line, where the crosses become triangles. The learning was continued to show that the error got slightly worse afterwards.

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.

3.4.2 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:

$$\text{Class is: } \begin{cases} C_1 \text{ if } y \leq -0.5 \\ C_2 \text{ if } -0.5 < y \leq 0 \\ C_3 \text{ if } 0 < y \leq 0.5 \\ C_4 \text{ if } y > 0.5 \end{cases} \tag{3.16}$$

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 3.2.3, 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.

3.4.3 A Classification Example

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 3.3.1, but using the maximum rather than the variance, and leaving the class IDs alone for now:

```
iris = loadtxt('iris_proc.data',delimiter=',')
iris[:,:4] = iris[:,:4]-iris[:,:4].mean(axis=0)
imax = concatenate((iris.max(axis=0)*ones((1,5)),iris.min(
axis=0)*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 zero, and simply set one of the entries to be 1:

```
target = zeros((shape(iris)[0],3))
indices = where(iris[:,4]==0)
target[indices,0] = 1
indices = where(iris[:,4]==1)
target[indices,1] = 1
indices = 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:

```
order = range(shape(iris)[0])
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.

3.4.4 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.

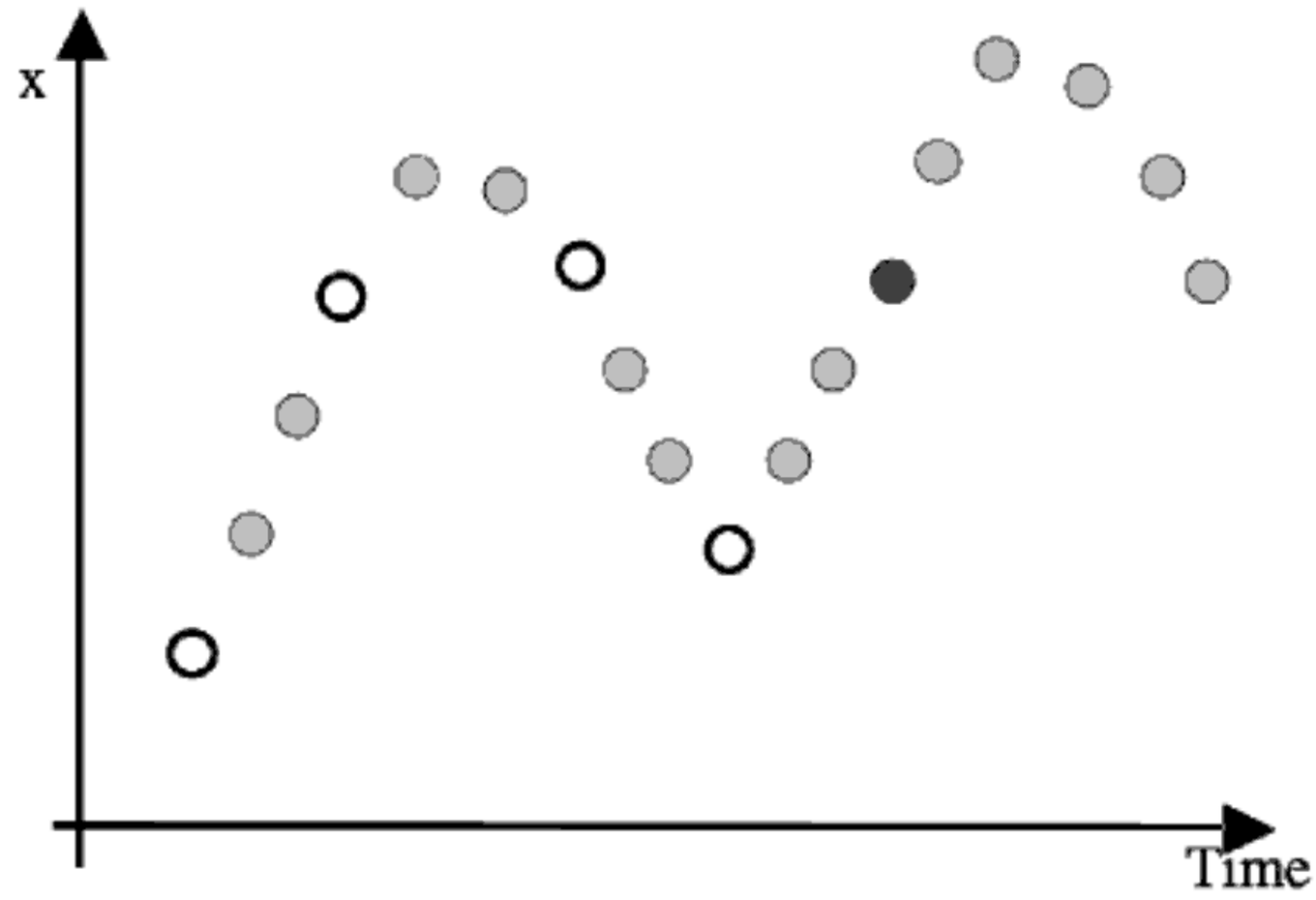


FIGURE 3.17: Part of a time-series plot, showing the datapoints and the meanings of τ and k.

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 $f(\cdot)$:

$$y = x(t + \tau) = f(x(t), x(t - \tau), \ldots, x(t - k\tau)), \tag{3.17}$$

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 3.17 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 Celcius 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

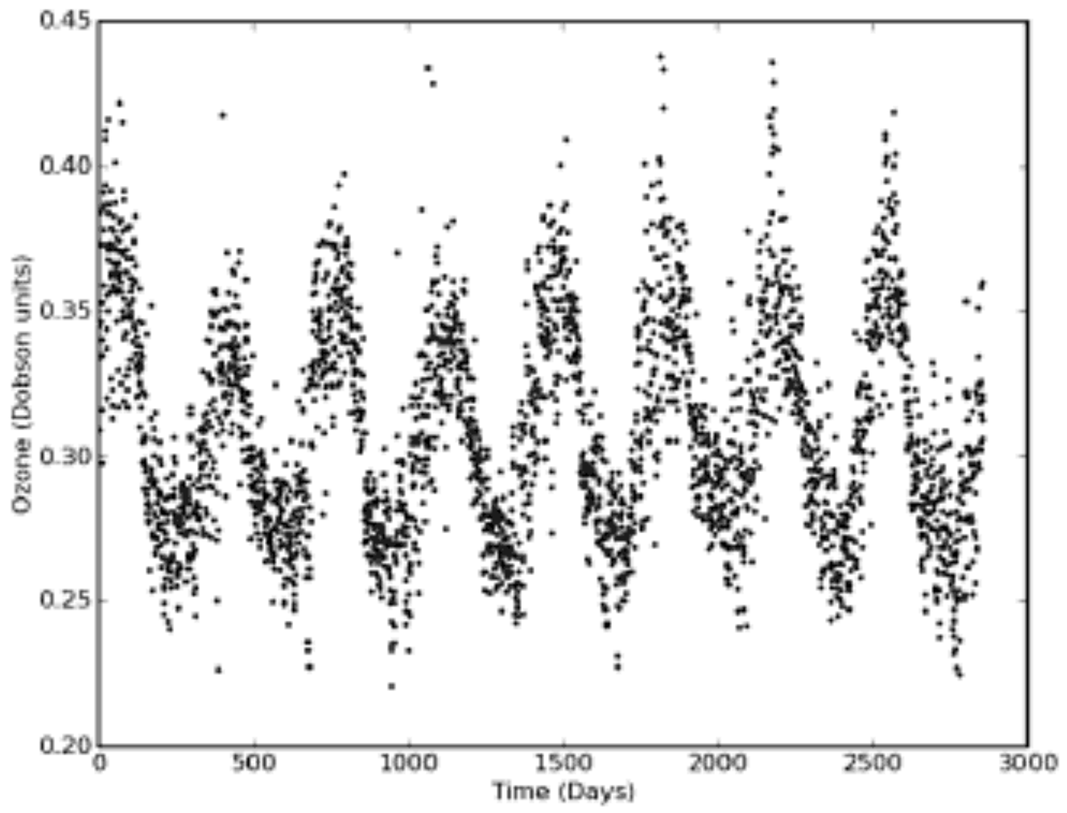


FIGURE 3.18: Plot of the ozone layer thickness above Palmerston North in New Zealand between 1996 and 2004.

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 3.18. 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 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

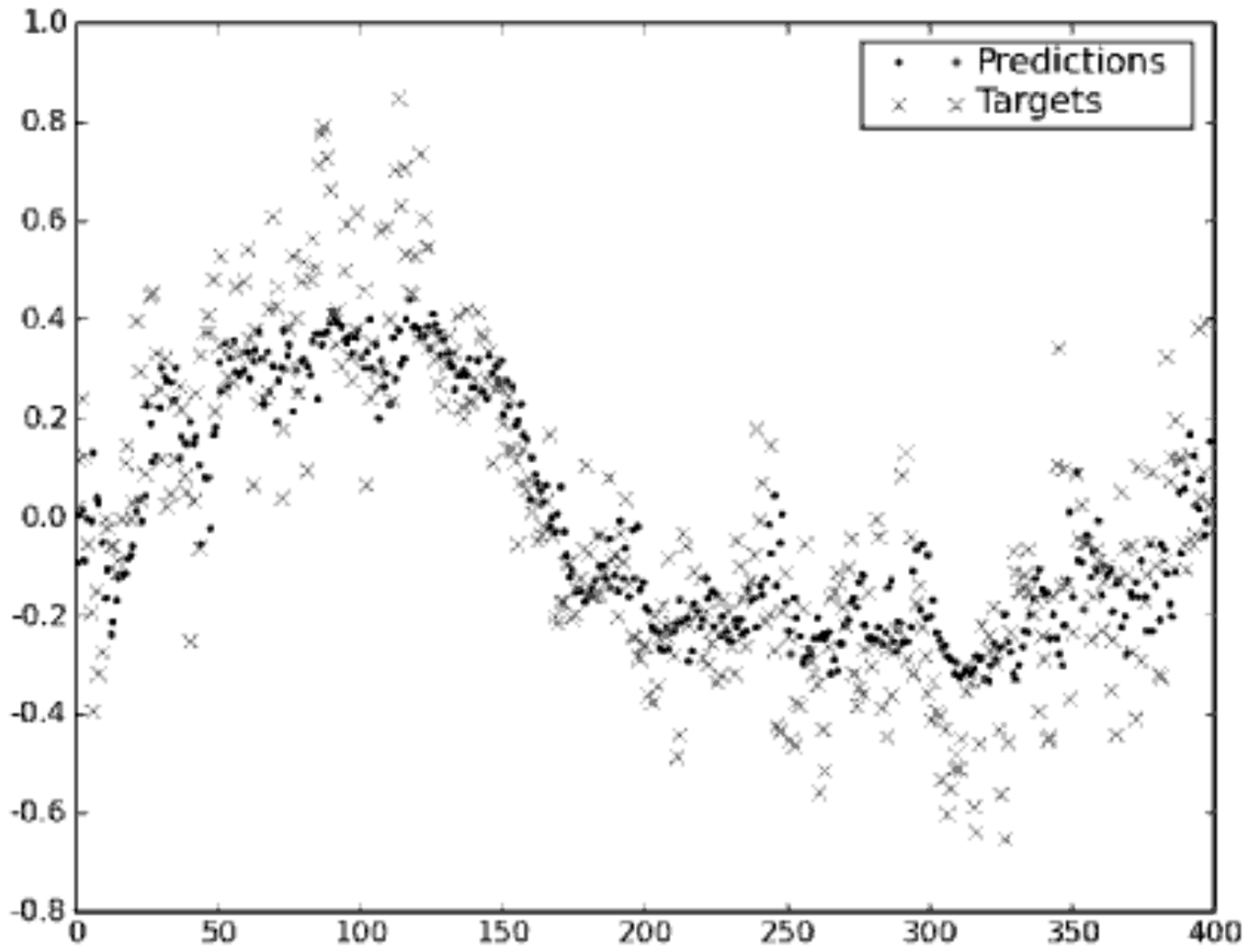


FIGURE 3.19: Plot of 400 predicted and actual output values of the ozone data using the MLP as a time-series predictor with $k = 3$ and $\tau = 2$.

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 3.19.

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.

3.4.5 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 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 3.20). 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

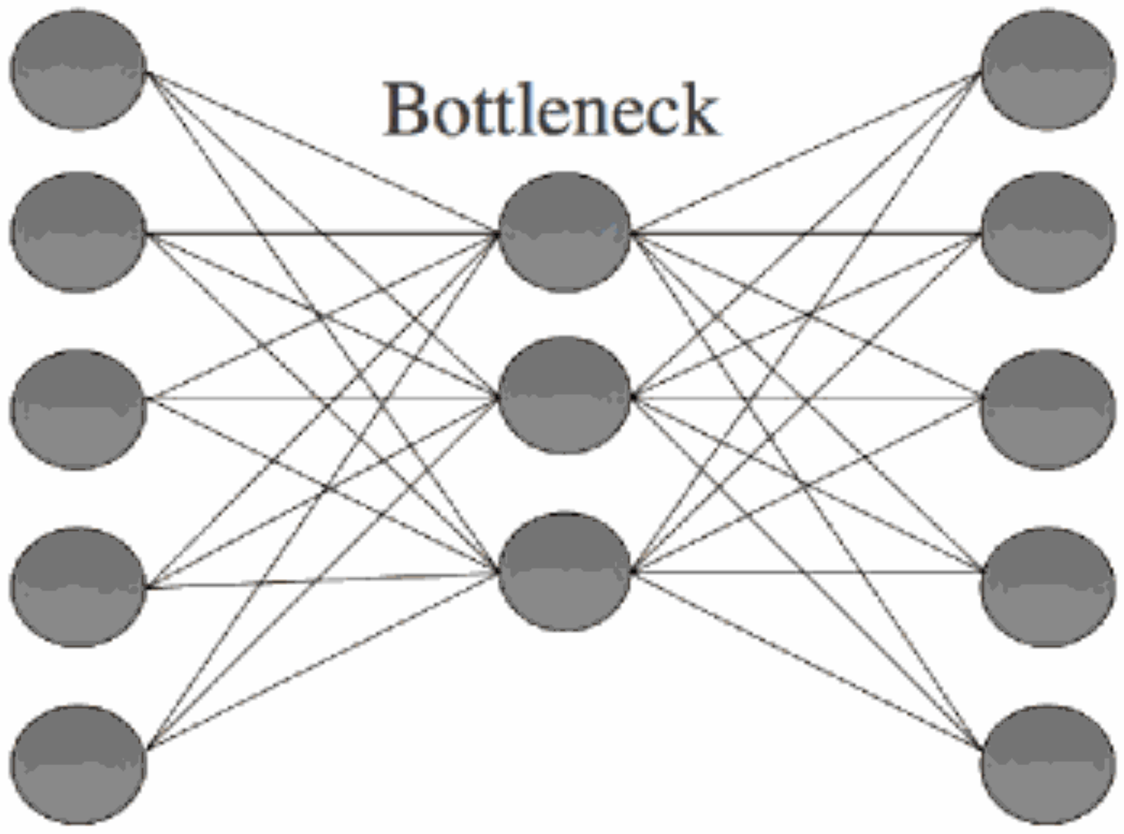


FIGURE 3.20: The auto-associative network. The network is trained to reproduce the inputs at the outputs, passing them through the bottleneck hidden layer that compresses the data.

the noise.

This auto-associative network can be used to compress images and other data. A schematic of this is shown in Figure 3.21: 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 3.22), 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 in 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.

You might be wondering what this representation in the hidden nodes looks like. In fact, what the network learns to compute are the Principal Components of the input data. Principal Component Analysis (PCA) is a useful dimensionality reduction technique, and is described in Section 10.2.

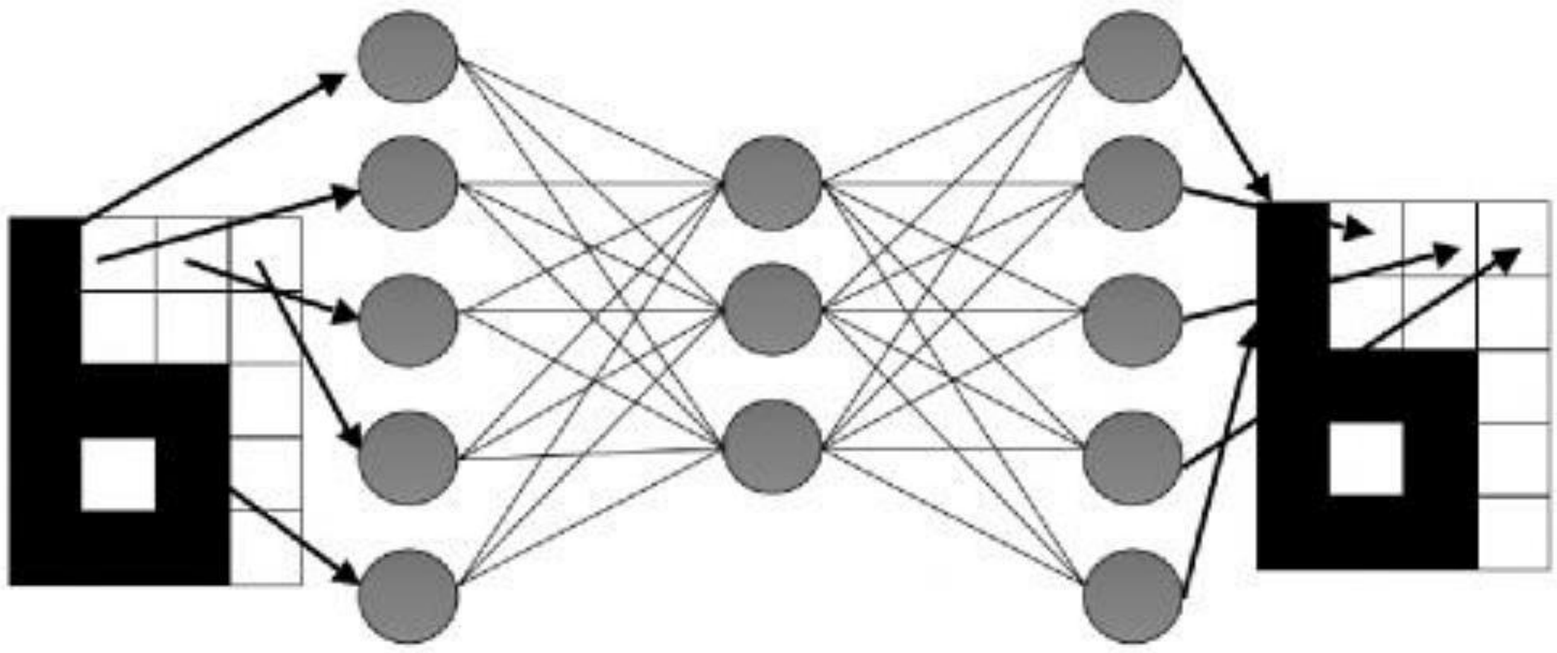


FIGURE 3.21: Schematic showing how images are fed into the auto-associative network for compression.

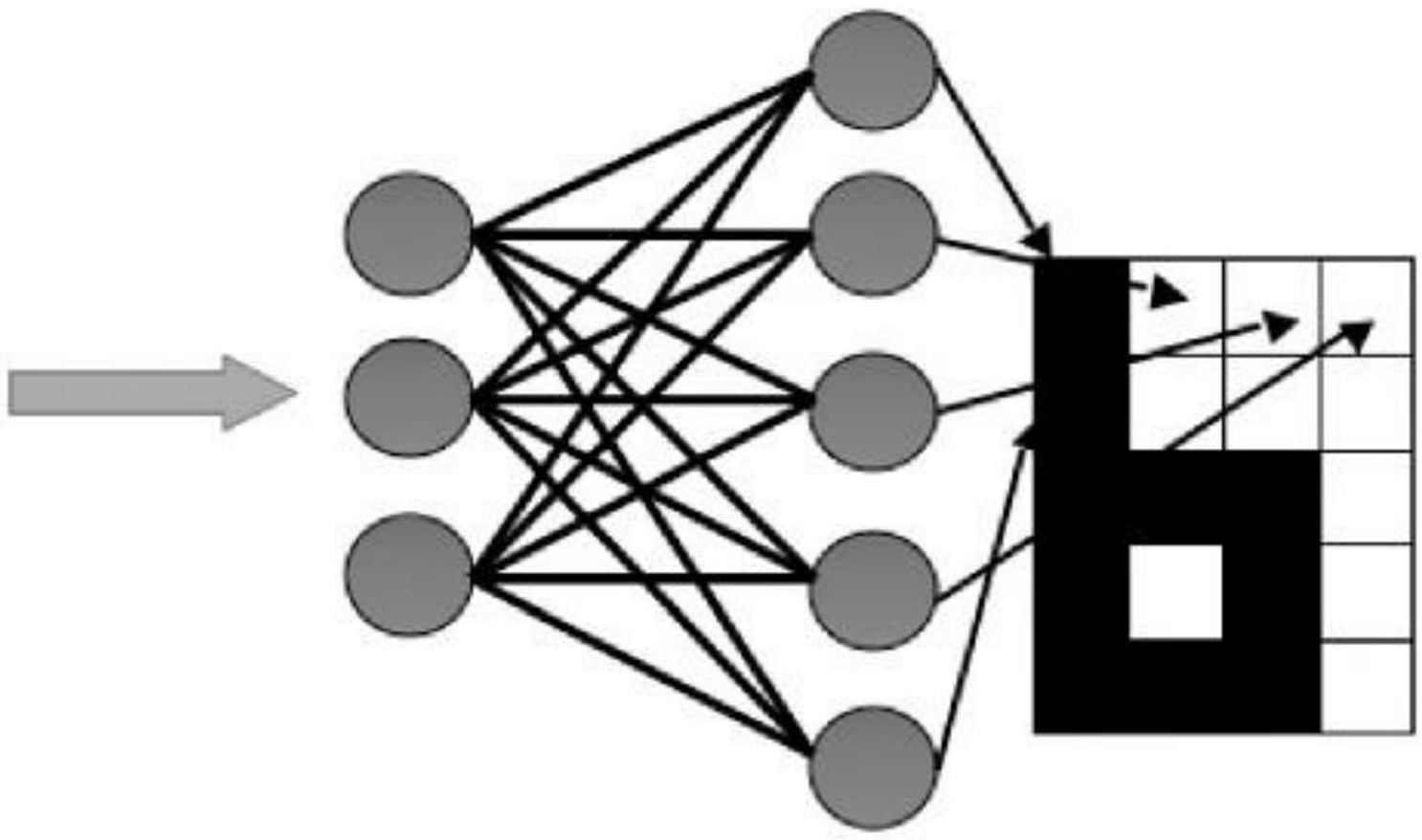


FIGURE 3.22: Schematic showing how the hidden nodes and second layer of weights can be used to regain the compressed images after the network has been trained.

3.5 Overview

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 in practice. 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.

Where there is not enough data for three sets, a technique called cross-validation can be useful. In its most extreme form, leave-one-out cross-validation, this consists of training the network on all but one piece of the training data and then validating it on the final piece. You then train another network on the training data again, but leaving out a different piece of data. You select one of the networks that gets the final piece correct. You still need a separate test set.

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.

3.6 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.

3.6.1 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 3.6.3, 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.

3.6.2 The Error of the Network

When we discussed the Perceptron learning rule in the previous chapter we motivated it by minimising the error function $E = \sum_{i=1}^{N} \mathbf{t}_i - \mathbf{y}_i$. 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 targets $\mathbf{t}$ and the outputs $\mathbf{y}$, 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 $E = \sum_{i=1}^{N} \mathbf{t}_i - \mathbf{y}_i$, but there are some problems with this: if $\mathbf{t}_i > \mathbf{y}_i$, the sign of the error is different to if $\mathbf{y}_i > \mathbf{t}_i$, 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 $\mathbf{t}_i$ and $\mathbf{y}_i$ for each node, squares them, and adds them together (I've missed out the $\mathbf{v}$ in $E(\mathbf{w})$ because we don't use them here):

$$E(\mathbf{w}) = \frac{1}{2} \sum_{k=1}^{N} (t_k - y_k)^2 \tag{3.18}$$

$$= \frac{1}{2} \sum_{k} \left[t_k - g\left(\sum_{j} w_{jk} a_j \right) \right]^2 \tag{3.19}$$

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, so Equation (3.19) will be replaced by:

$$\frac{1}{2}\sum_k \left[t_k - g\left(\sum_j w_{jk} x_j \right) \right]^2 \tag{3.20}$$

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 $y_k = \sum_j w_{jk} x_j$ where x_j is the value of an input node.

We are going to use a gradient descent algorithm that adjusts each weight w_{jk} in the direction of the 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:

$$\frac{\partial E}{\partial w_{ik}} = \frac{\partial}{\partial w_{ik}} \left(\frac{1}{2} \sum_k (t_k - y_k)^2 \right) \tag{3.21}$$

$$= \frac{1}{2} \sum_k 2(t_k - y_k) \frac{\partial}{\partial w_{ik}} \left(t_k - \sum_j w_{jk} x_j \right) \tag{3.22}$$

t_k is not a function of w_{ik}, so $\dfrac{\partial t_k}{\partial w_{ik}} = 0$,

and the only part of $\sum_j w_{jk} x_j$ that is a function of w_{ik} is when

$i = j$, that is, w_{ik} itself. Hence:

$$\frac{\partial E}{\partial w_{ik}} = \sum_k (t_k - y_k)(-x_i). \tag{3.23}$$

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

$$w_{ik} \leftarrow w_{ik} + \eta(t_k - y_k)x_i, \tag{3.24}$$

which hopefully looks familiar (see Equation (2.1)). It isn't actually identical, because we are computing y_k differently, and for the Perceptron, we used the

threshold activation threshold function, whereas in the work above we ignored the thresholder. 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.

3.6.3 A Suitable Activation Function

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 3.5) that satisfy all those criteria perfectly. The form in which it is generally used is:

$$a = g(h) = \frac{1}{1 + \exp(-\beta h)}, \tag{3.25}$$

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

$$g'(h) = \frac{dg}{dh} = \frac{d}{dh}(1 + e^{-\beta h})^{-1} \tag{3.26}$$

$$= -1(1 + e^{-\beta h})^{-2}\frac{de^{-\beta h}}{dh} \tag{3.27}$$

$$= -1(1 + e^{-\beta h})^{-2}(-\beta e^{-\beta h}) \tag{3.28}$$

$$= \frac{\beta e^{-\beta h}}{(1 + e^{-\beta h})^2} \tag{3.29}$$

$$= \beta g(h)(1 - g(h)) \tag{3.30}$$

$$= \beta a(1 - a) \tag{3.31}$$

We'll be using this derivative later, except that we can ignore the factor of β, since this is just a scaling. So we've now got an error function and an activation function that we can compute derivatives of. The next things to do is work out how to use them in order to adjust the weights of the network.

3.6.4 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:

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial E}{\partial h_k}\frac{\partial h_k}{\partial w_{jk}}, \tag{3.32}$$

where $h_k = \sum_l w_{lk}a_l$ is the input to output-layer neuron k, that is, the sum of the activations of the hidden-layer neurons multiplied by the relevant (second-layer) weights. So what does Equation (3.32) 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 $\frac{\partial w_{lk}}{\partial w_{jk}} = 0$ for all values of l except $l = j$):

$$\frac{\partial h_k}{\partial w_{jk}} = \frac{\partial \sum_l w_{lk}a_l}{\partial w_{jk}} \tag{3.33}$$

$$= \sum_l \frac{\partial w_{lk}a_l}{\partial w_{jk}} \tag{3.34}$$

$$= a_j. \tag{3.35}$$

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

$$\delta_o = \frac{\partial E}{\partial h_k}. \tag{3.36}$$

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:

$$\delta_o = \frac{\partial E}{\partial h_k} = \frac{\partial E}{\partial y_k}\frac{\partial y_k}{\partial h_k}. \tag{3.37}$$

Now the output of output layer neuron k is

$$y_k = g(h_k^{\text{output}}) = g\left(\sum_j w_{jk}a_j^{\text{hidden}}\right), \tag{3.38}$$

where $g(\cdot)$ is the activation function. We've chosen to use the sigmoid function given in Equation (3.25), but for now I'm going to leave it as a function to make it a little bit more general. 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 (3.41) I've substituted in the expression for the error at the output, which we computed in Equation (3.19):

$$\delta_o = \frac{\partial E}{\partial g\left(h_k^{\text{output}}\right)} \frac{\partial g\left(h_k^{\text{output}}\right)}{\partial h_k^{\text{output}}} \tag{3.39}$$

$$= \frac{\partial E}{\partial g\left(h_k^{\text{output}}\right)} g'\left(h_k^{\text{output}}\right) \tag{3.40}$$

$$= \frac{\partial}{\partial g\left(h_k^{\text{output}}\right)} \left[\frac{1}{2}\sum_k \left(g(h^{\text{output}}) - t_k\right)^2\right] g'\left(h_k^{\text{output}}\right) \tag{3.41}$$

$$= \left(g(h_k^{\text{output}}) - t_k\right) g'(h_k^{\text{output}}) \tag{3.42}$$

$$= (y_k - t_k) g'(h_k^{\text{output}}), \tag{3.43}$$

where $g'(h_k)$ denotes the derivative of g with respect to h_k. We know exactly what that is for the sigmoid functions that we are using: we computed it in Equation (3.31). So we can put everything together to compute the precise update rule for the second-layer weights, $w_{jk} \leftarrow w_{jk} - \eta \frac{\partial E}{\partial w_{jk}}$ (we are using the minus sign because we want to go downhill), where:

$$\frac{\partial E}{\partial w_{jk}} = \delta_o a_j \tag{3.44}$$

$$= (y_k - t_k) y_k (1 - y_k) a_j. \tag{3.45}$$

Having got through all this, we don't actually need to do too much more work to get to the first layer of v_{jk} weights, which connect the inputs to the hidden nodes. We need the chain rule (Equation (3.32)) one more time to get to these weights, remembering that we are working backwards through the network:

$$\delta_h = \sum_k \frac{\partial E}{\partial h_k^{\text{output}}} \frac{\partial h_k^{\text{output}}}{\partial h_j^{\text{hidden}}} \tag{3.46}$$

$$= \sum_k \delta_o \frac{\partial h_k^{\text{output}}}{\partial h_j^{\text{hidden}}}. \tag{3.47}$$

In the first line, k runs over the output nodes, and we obtain the second line by using Equation (3.40). 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:

$$h_k^{\text{output}} = g\left(\sum_l w_{lk} h_l^{\text{hidden}}\right), \tag{3.48}$$

which means that:

$$\frac{\partial h_k^{\text{output}}}{\partial h_j^{\text{hidden}}} = \frac{\partial g\left(\sum_l w_{lk} h_l^{\text{hidden}}\right)}{\partial h_j^{\text{hidden}}}. \tag{3.49}$$

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

$$\frac{\partial h_k^{\text{output}}}{\partial h_j^{\text{hidden}}} = w_{jk} g'(a_j) \tag{3.50}$$

$$= w_{jk} a_j (1 - a_j) \tag{3.51}$$

which allows us to compute:

$$\delta_h = a_j(1 - a_j) \sum_k \delta_o w_{jk} \tag{3.52}$$

and so get to the update rule for $v_{ij} \leftarrow v_{ij} - \eta \frac{\partial E}{\partial v_{ij}}$, by computing:

$$\frac{\partial E}{\partial v_{ij}} = a_j(1 - a_j) \left(\sum_k \delta_o w_{jk}\right) x_i. \tag{3.53}$$

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.

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.

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 3.1 Work through the MLP shown in Figure 3.2 to ensure that it does solve the XOR problem.

Problem 3.2 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 estimate 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 3.3 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 3.4 Would the previous system be better than just using the dictionary?

Problem 3.5 Modify the code on the book website to work sequentially rather than in batch mode. Compare the results on the iris dataset.

Problem 3.6 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 2.3.3.

Problem 3.7 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 3.8 The book website contains a set of datafiles that are very basic (ASCII-style) 16×16 images of numbers (in the file `numbers.dat`. Modify the MLP code in order to turn it into an auto-associative network and then train it on these number images. You can then store the values in the hidden nodes (the compressed version) and use it to recreate the number images themselves. There are also some corrupted versions of the images (in `badnumbers.dat`). Experiment to see how well they can be cleaned up. Then move on to the MNIST dataset that is available via the book website. How much does the extra dimensions of these images affect the results?

Problem 3.9 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 3.10 The alternative activation function that can be used in $\tanh(h)$. Show that $\tanh(h) = 2g(2h) - 1$, where g is given by Equation (3.2). Use this to show that there is an exactly equivalent MLP using the tanh activation function. Modify the code to implement it.

Chapter 4

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.

4.1 Concepts

4.1.1 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

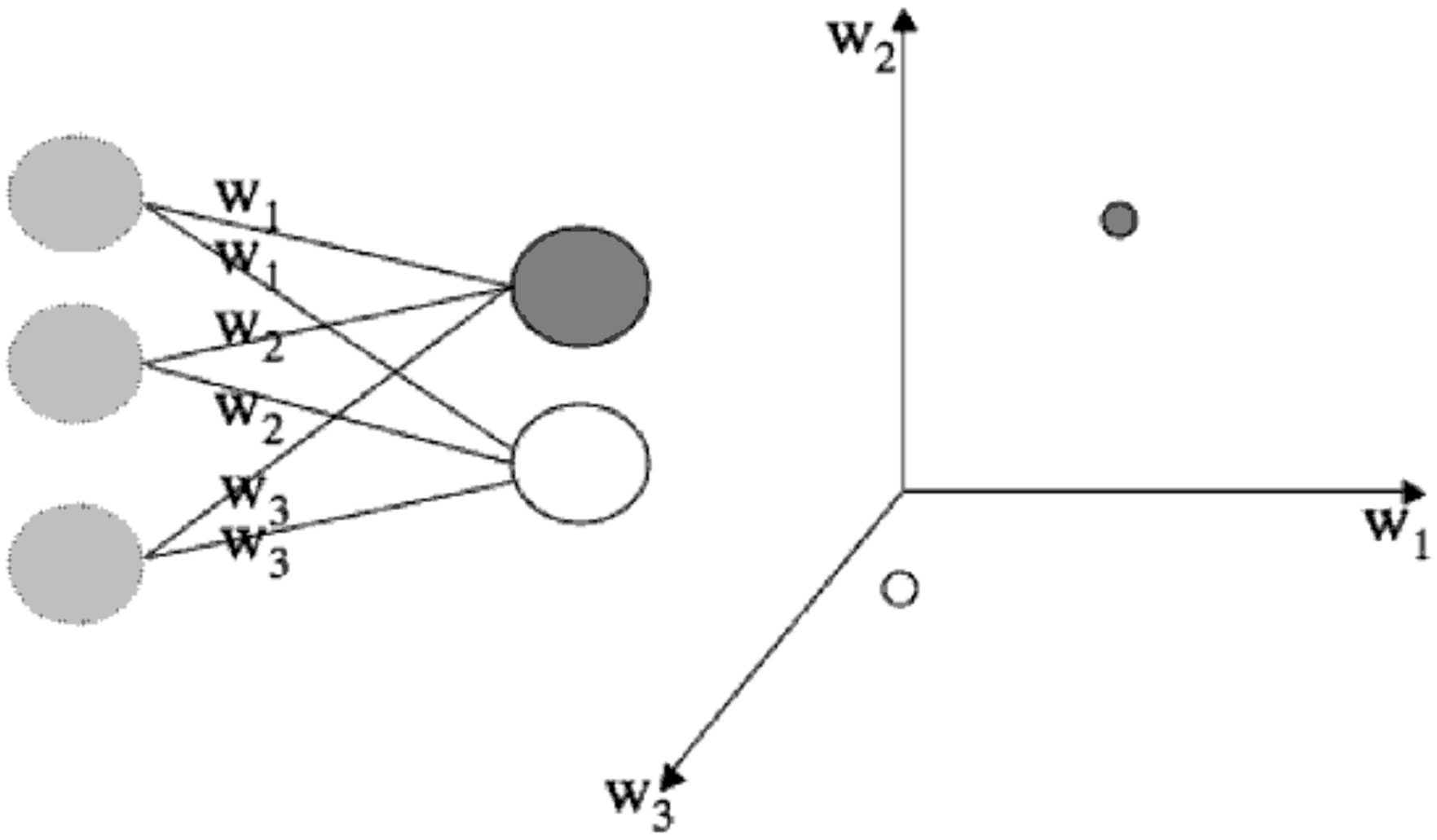


FIGURE 4.1: The position of two neurons in weight space. The labels on the network refer to the dimension in which that weight is plotted, not its value. Note that there is no bias node.

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 the position of a neuron in weight space. If we think about the weights that connect into a particular neuron, we can 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. There is a schematic of this in Figure 4.1.

Now 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 positions of 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, 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:

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}. \tag{4.1}$$

So we can use the idea of neurons and inputs being 'close together' in order to decide 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. To see why this might be a good idea, we need to look at the idea of receptive fields.

4.1.2 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 4.1.1, 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 4.1.1 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 position of the star has changed, 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 are a set of rods that are 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, so 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 4.2 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 4.2 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 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

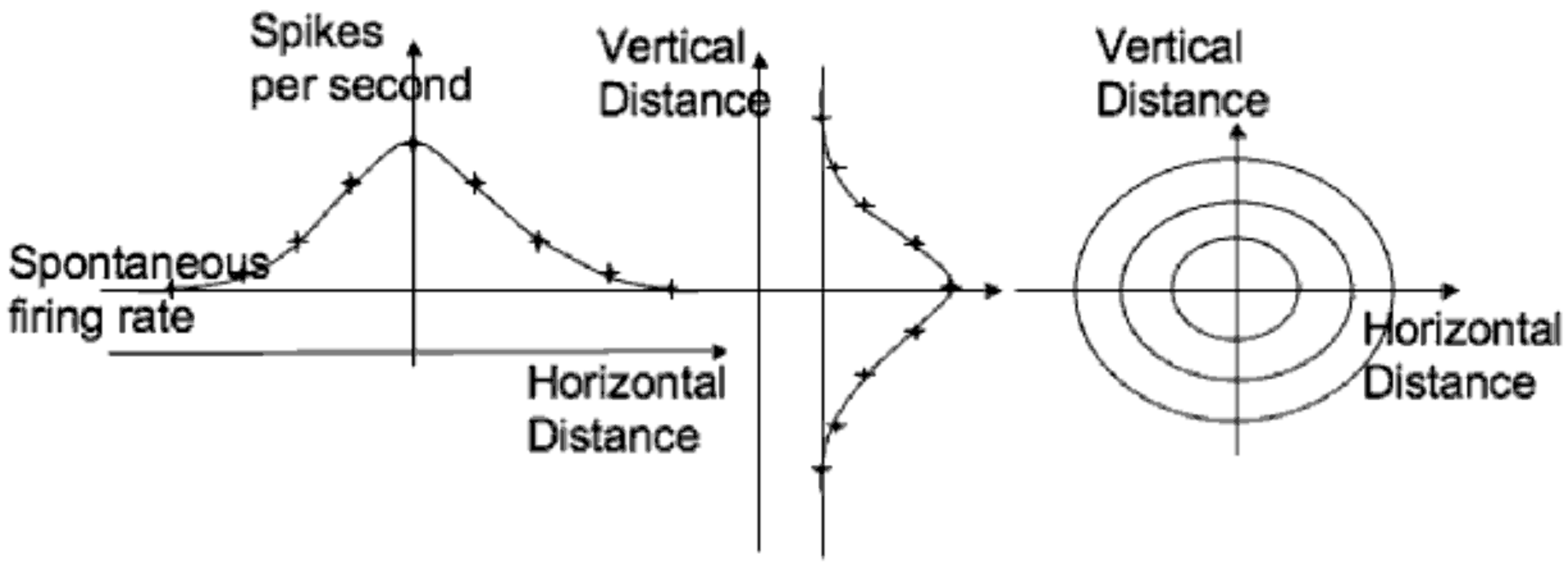


FIGURE 4.2: *Left:* Count of the number of spikes per second as the distance of a rod from the light varies horizontally. Note that it does not go to zero, but to the spontaneous firing rate of the neuron, which is how often it fires without input. *Centre:* The same thing for vertical motion. *Right:* The combination of the two makes a set of circles.

with by far the most common one in statistics, the Gaussian, something that is important enough to get its own section later on (Section 8.2.3). 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:

$$g(\mathbf{x}, \mathbf{w}, \sigma) = \exp\left(\frac{-\|\mathbf{x} - \mathbf{w}\|^2}{2\sigma^2}\right). \tag{4.2}$$

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 4.3 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.

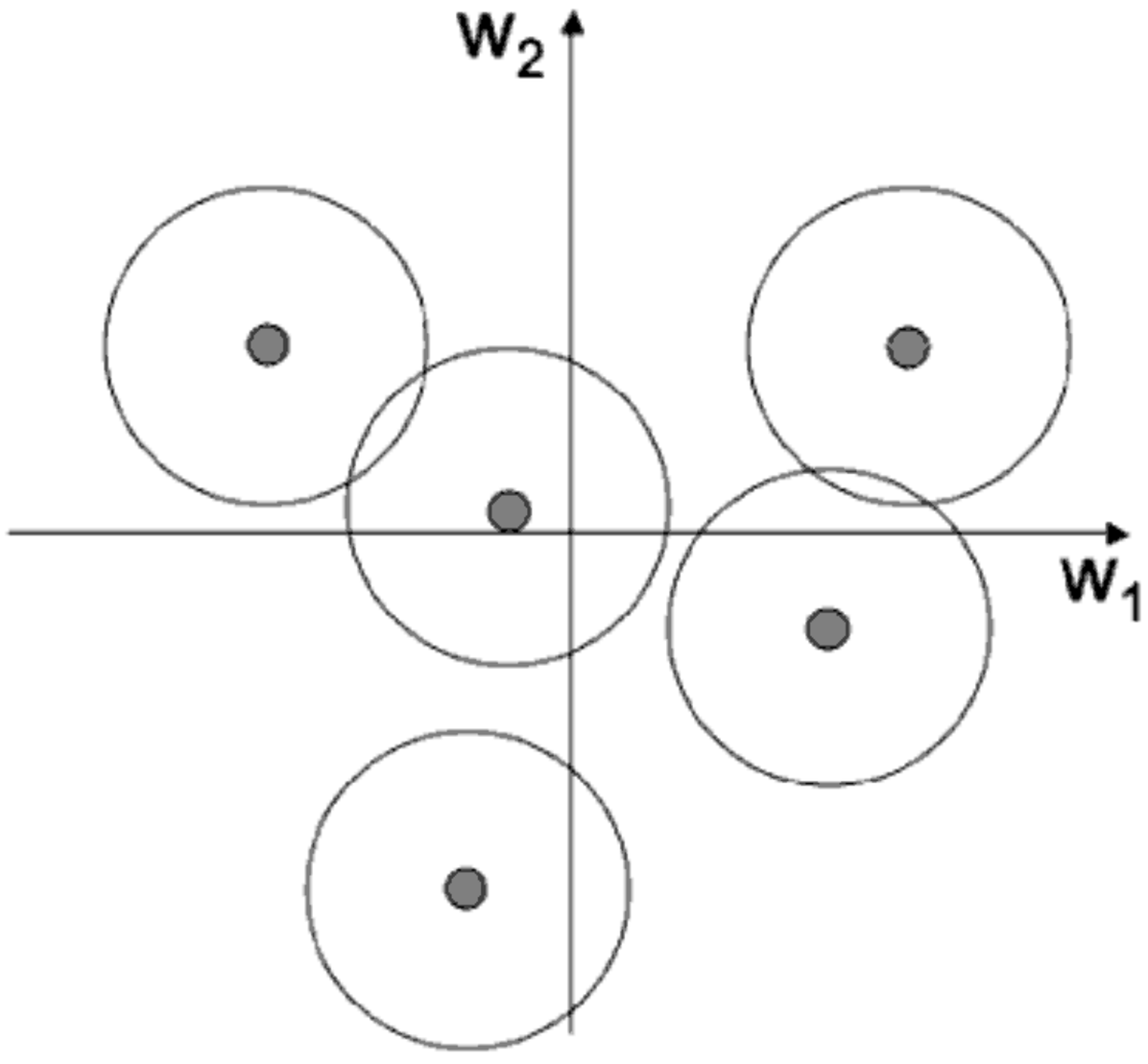


FIGURE 4.3: The effect of radial basis functions in weight space. The points show the position of the RBF in weight space, while the circle around each point shows the receptive field of the node. In higher dimensions these circles become hyperspheres.

4.2 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 9.1.3 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 4.4. 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

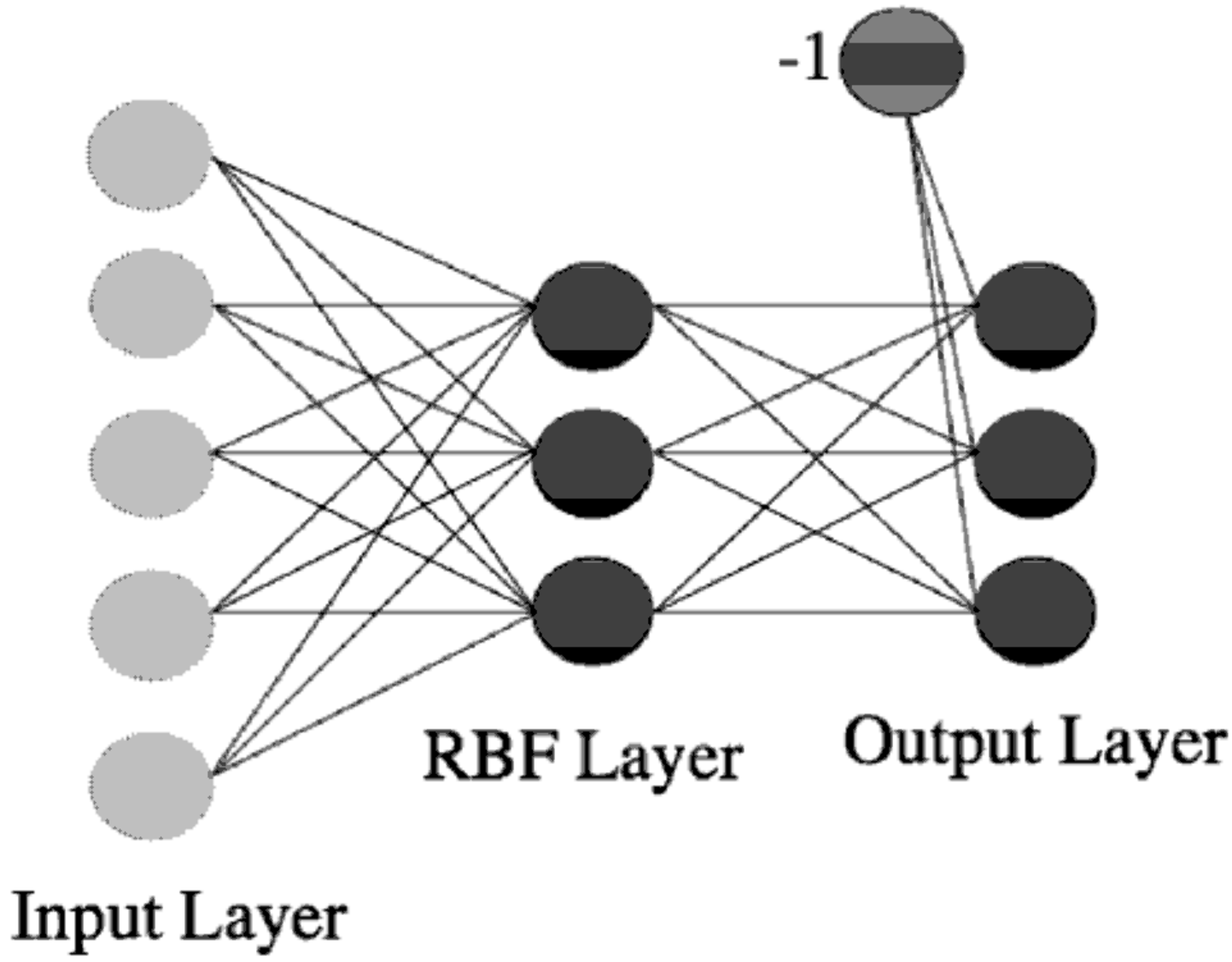


FIGURE 4.4: The Radial Basis Function network consists of input nodes connected by weights to a set of RBF neurons, which fire proportionally to the distance between the input and the neuron in weight space. The activations of these nodes are used as inputs to the second layer, which consists of linear nodes. The schematic looks very similar to the MLP except for the lack of a bias in the hidden layer.

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 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 4.5. 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 in 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

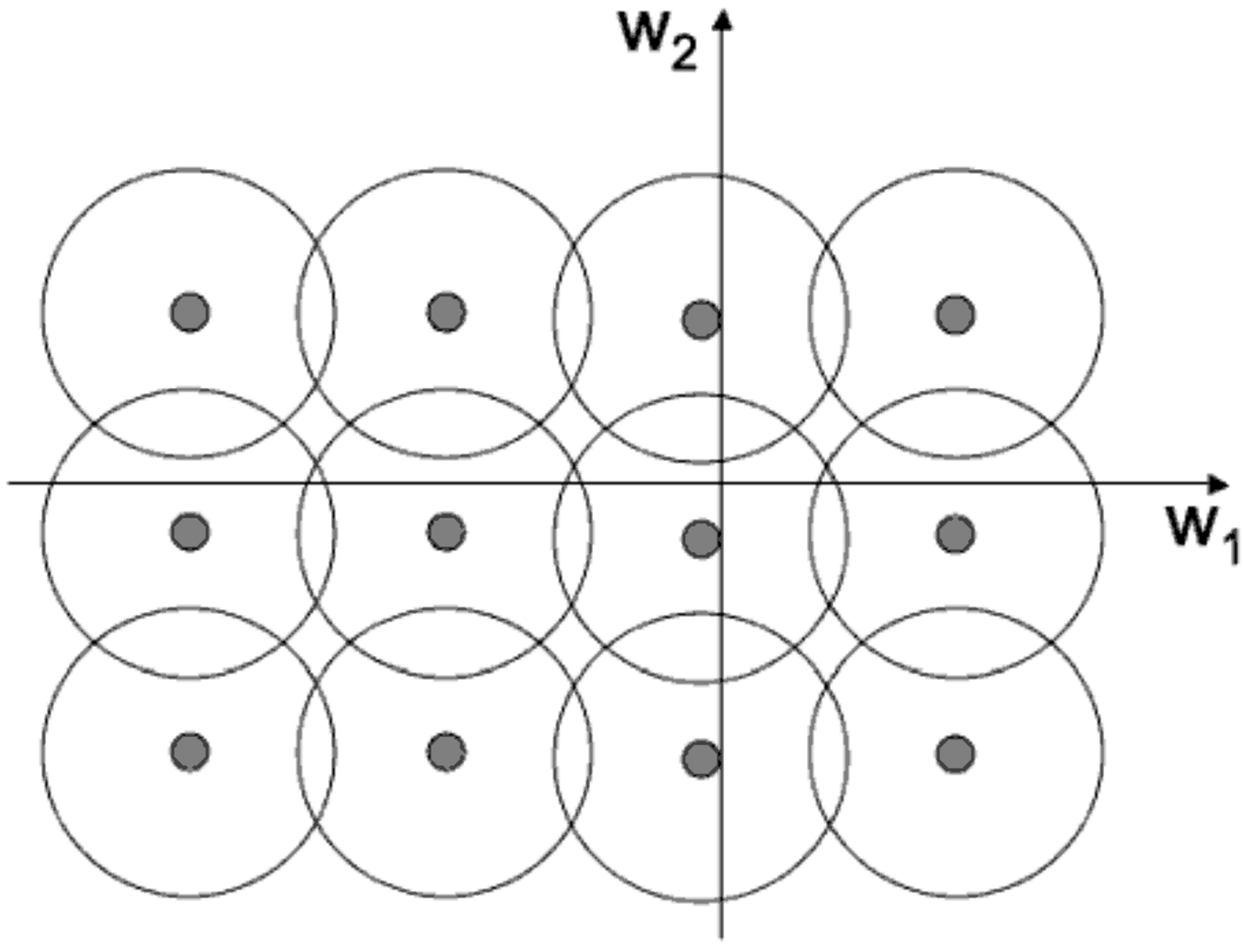


FIGURE 4.5: We can space out RBF nodes to cover the whole of space by continuing this pattern everywhere, so that the network acts as a universal approximator, since there is an output for every possible input.

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 9.1.3. 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.

4.2.1 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 2.2). 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 9. For the RBF network, the most common one is the *k*-**means algorithm** that is described in Section 9.1. 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 (4.2)
- 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; in Eclipse this is done by selecting the project, accessing its properties, and adding the relevant source folder into the `pydev - PYTHONPATH` entry). The training is then very simple:

```
def rbftrain(self,inputs,targets,eta=0.25,niterations=10):

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

    for i in range(self.nRBF):
        self.hidden[:,i] = exp(-sum((inputs - ones((1,self.
        nin))*self.weights1[:,i])**2,axis=1)/(2*self.sigma**2)
        )
    if self.normalise:
        self.hidden[:,:-1] /= transpose(ones((1,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 `linalg.pinv()`. This gives us an alternative to the Perceptron network that is even faster, since the training only needs one iteration:

```
self.weights2 = dot(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 out 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 (4.2) and it looks like the soft-max function:

$$g(\mathbf{x}, \mathbf{w}, \sigma) = \frac{\exp(-\|\mathbf{x} - \mathbf{w}\|/2\sigma^2)}{\sum_i \exp(-\|\mathbf{x} - \mathbf{w}_i\|/2\sigma^2)}. \tag{4.3}$$

Using the RBF network on the `iris` dataset that was used in Section 3.4.3 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 8.4.3; 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

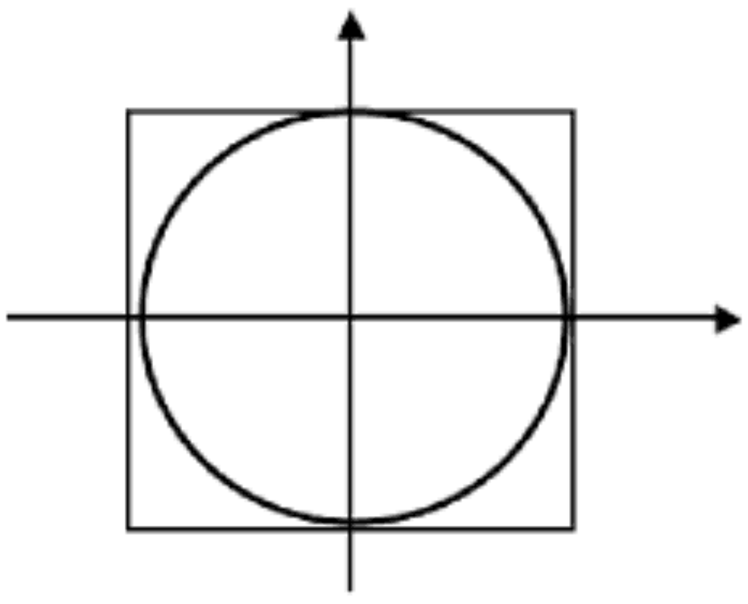


FIGURE 4.6: The unit circle in 2D with its bounding box.

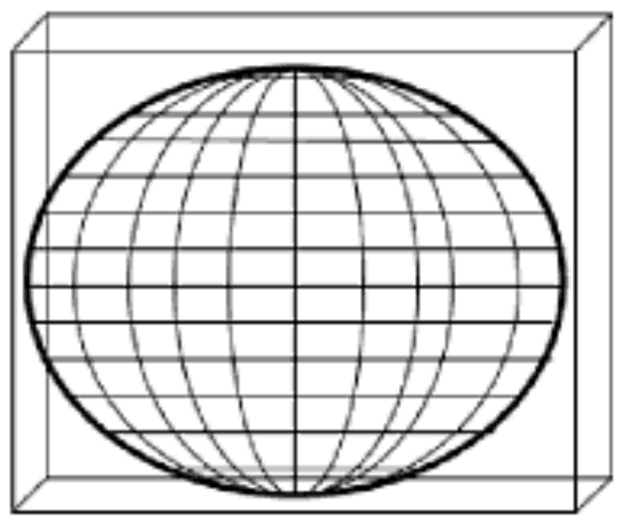


FIGURE 4.7: The unit sphere in 3D with its bounding cube. The sphere does not reach as far into the corners as the circle does, and this gets more noticeable as the number of dimensions increases.

the space. The number of input dimensions has a profound effect on learning, something that has a suitably impressive name: the curse of dimensionality.

4.3 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 4.6), and in 3D we get a sphere around $(0, 0, 0)$ (Figure 4.7). 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 4.8 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.

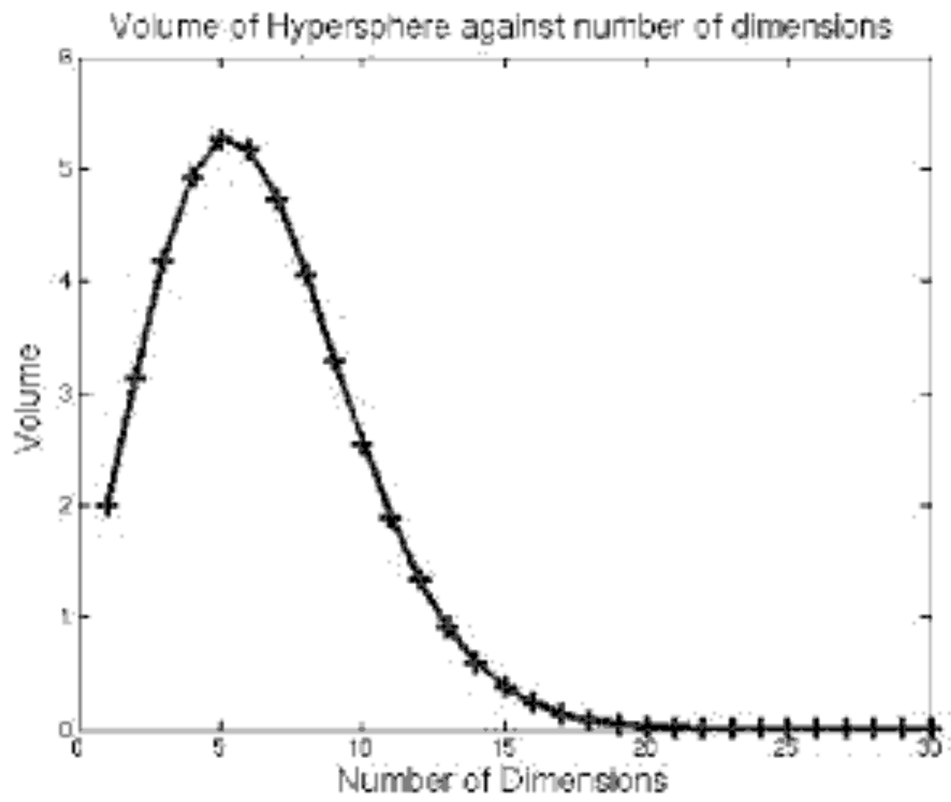


FIGURE 4.8: The volume of the unit hypersphere for different numbers of dimensions.

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 4.6) The same is true in 3D (Figure 4.7), 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 4.8 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 is $v_n = (2\pi/n)v_{n-2}$. 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, so 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 RBFs, 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.

4.4 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 4.9. 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 4.10. 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 4.11.

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 4.12. 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.

4.4.1 Bases and Basis Expansion

Radial basis functions and several other machine learning algorithms can be written in this form:

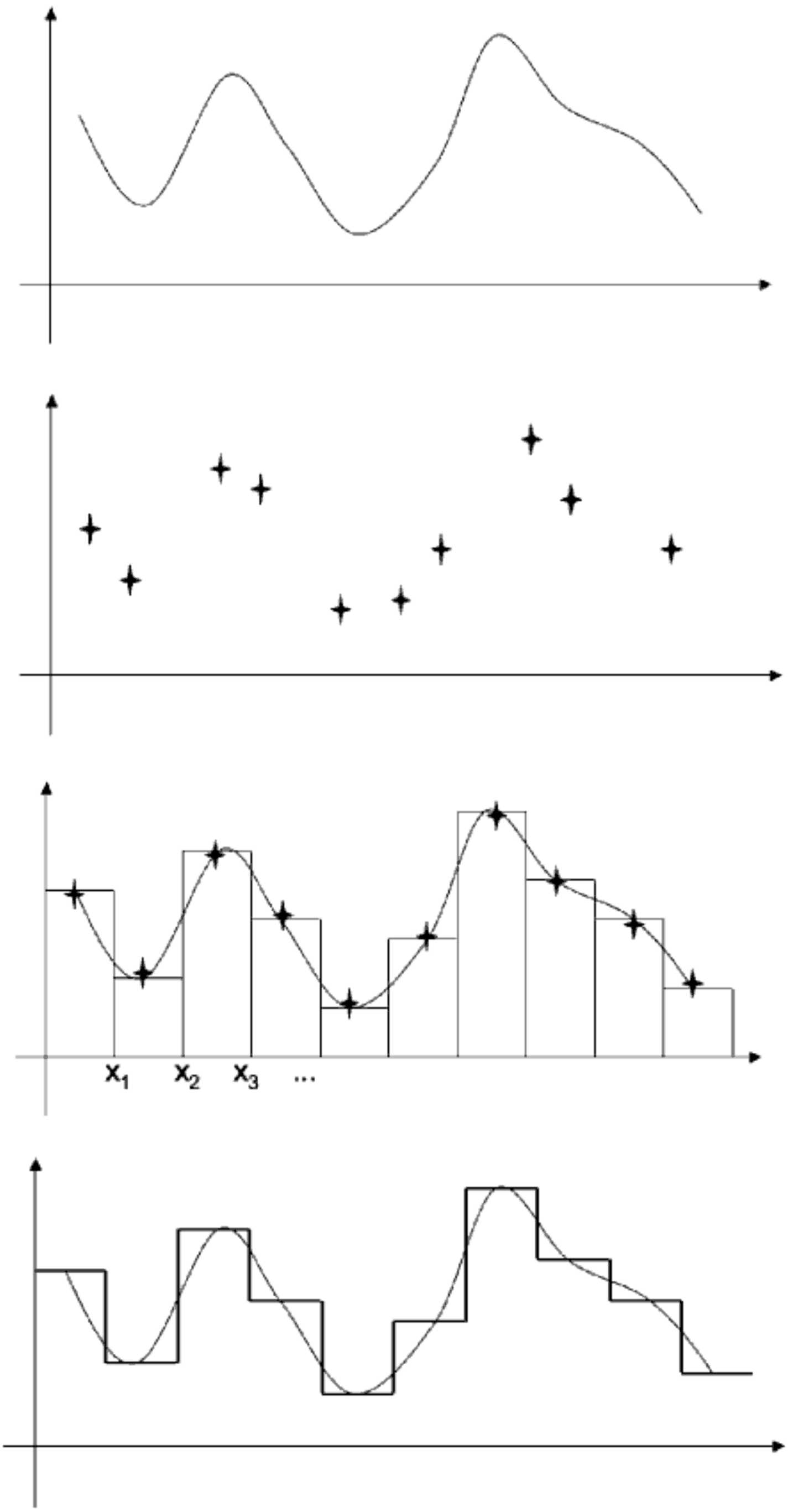


FIGURE 4.9: *Top:* Curve showing a function. *Second:* A set of datapoints from the curve. *Third:* Putting a straight horizontal line through each point creates a histogram that describes an approximation to the curve. *Bottom:* That approximation.

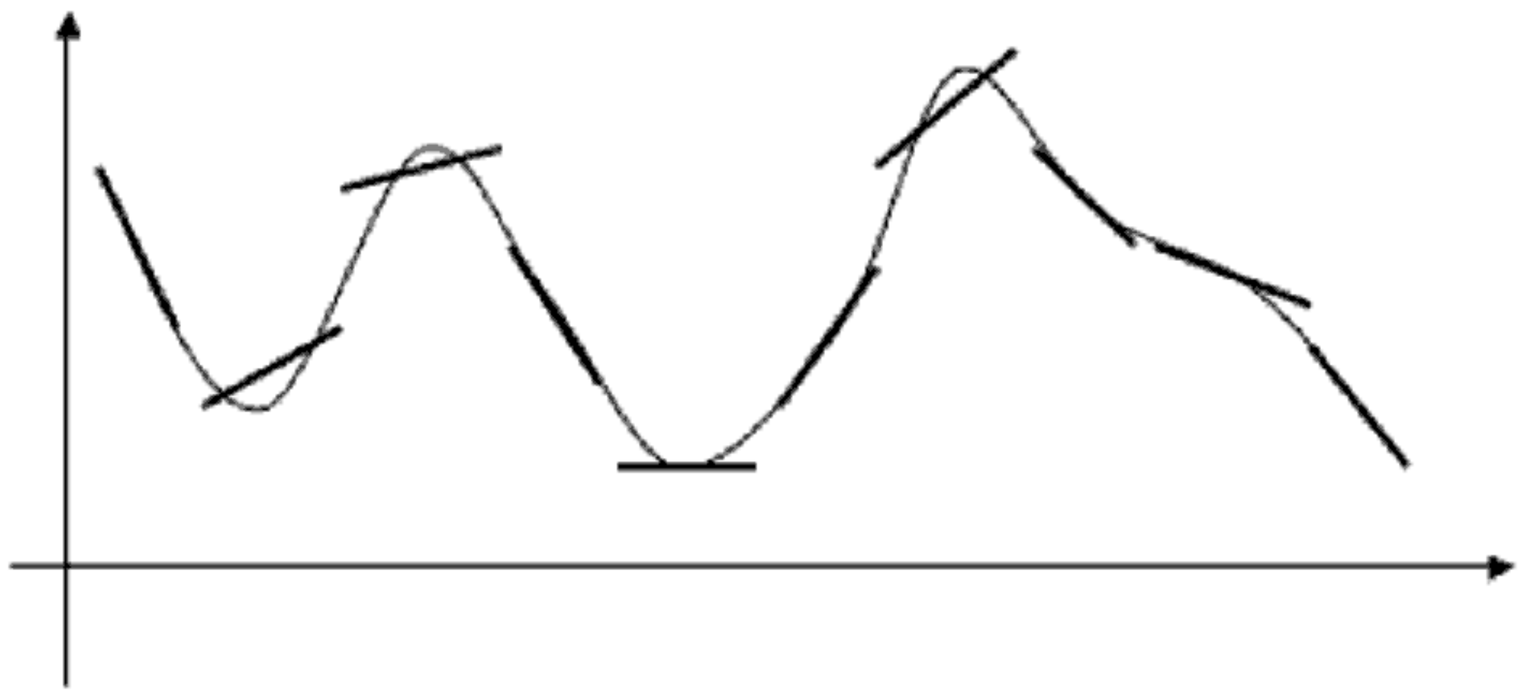


FIGURE 4.10: Representing the points by straight lines that aren't necessarily horizontal (so that their first derivative matches at the point) gives a better approximation.

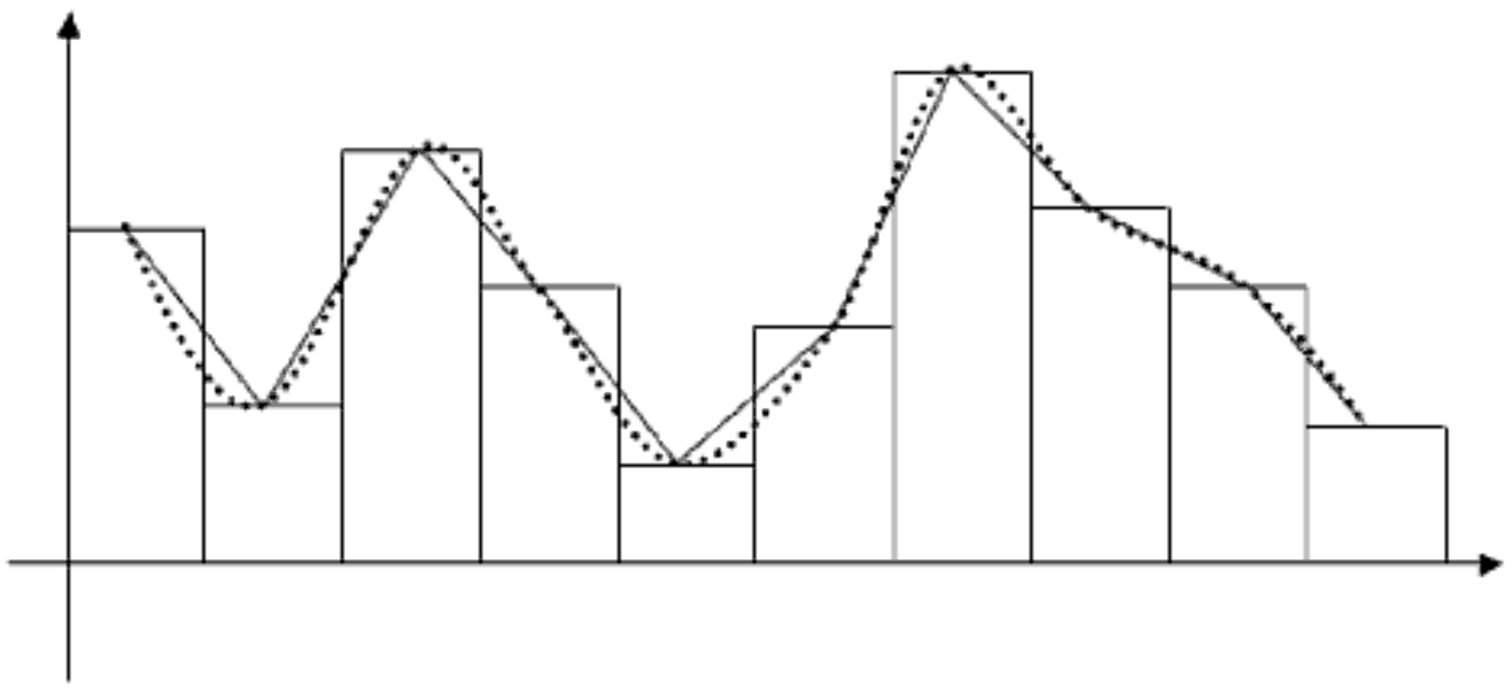


FIGURE 4.11: Making the straight lines meet so that the function is continuous gives a better approximation.

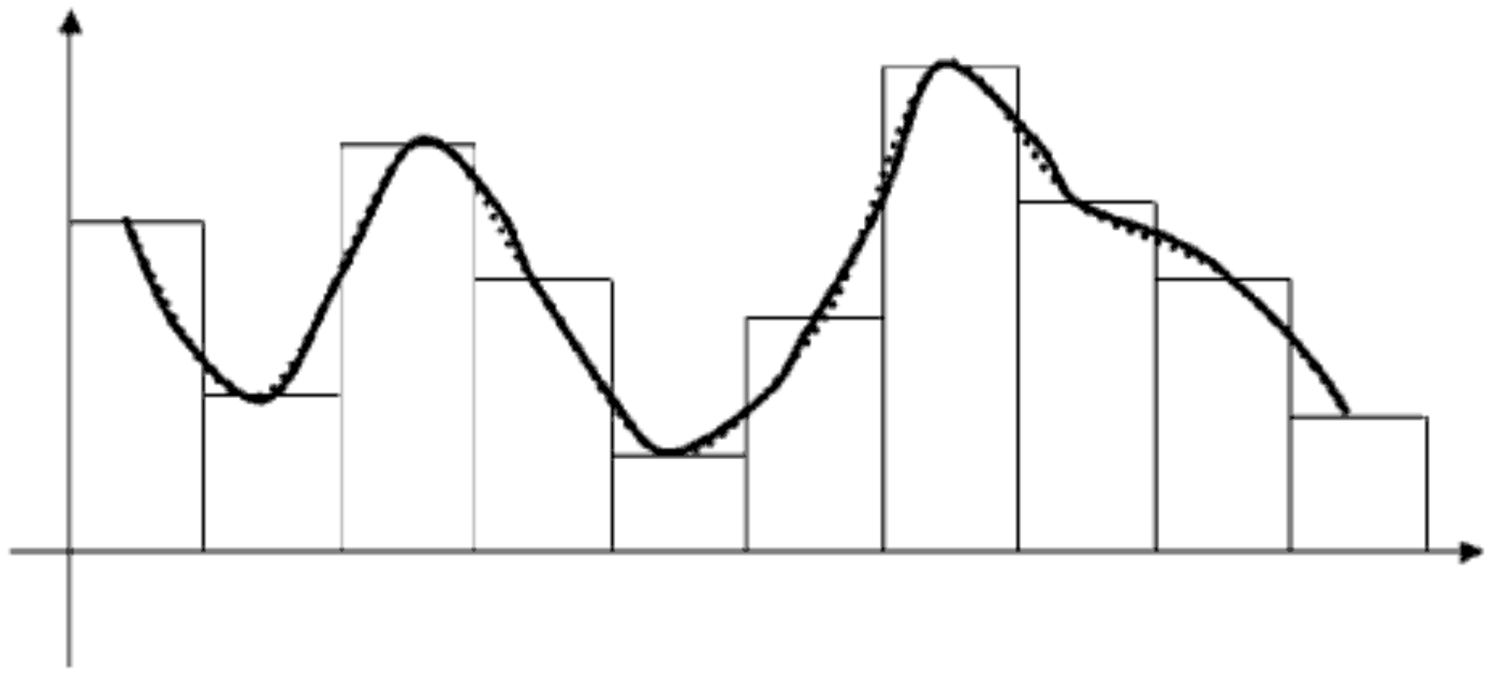


FIGURE 4.12: Using cubic functions to connect the points gives an even better approximation, and the curve is also continuous at the points where the sections join up (known as knotpoints).

$$f(\mathbf{x}) = \sum_{i=1}^{n} \alpha_i \Phi_i(\mathbf{x}), \tag{4.4}$$

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. Note that this is more general than a neural network, where the Φ would correspond to the neuronal activations and are fixed, while the α_i correspond to the network weights. 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 4.9 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 4.9. 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 4.10, 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 4.11.

4.4.2 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 4.12, which represent the data very well.

4.4.3 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 11.2. 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:

$$E(y, f(x)) = \sum_{i=1}^{N} (y_i - f(x_i))^2. \tag{4.5}$$

NumPy already has a method defined for computing linear least-squares optimisation: the function `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 4.13 shows the results.

```
from pylab import *
from numpy import *

x = arange(-3,10,0.05)
y = 2.5 * exp(-(x)**2/9) + 3.2 * exp(-(x-0.5)**2/4) + random.
normal(0.0, 1.0, len(x))
nParam = 2
A = zeros((len(x),nParam), float)
A[:,0] = exp(-(x)**2/9)
A[:,1] = exp(-(x*0.5)**2/4)
```

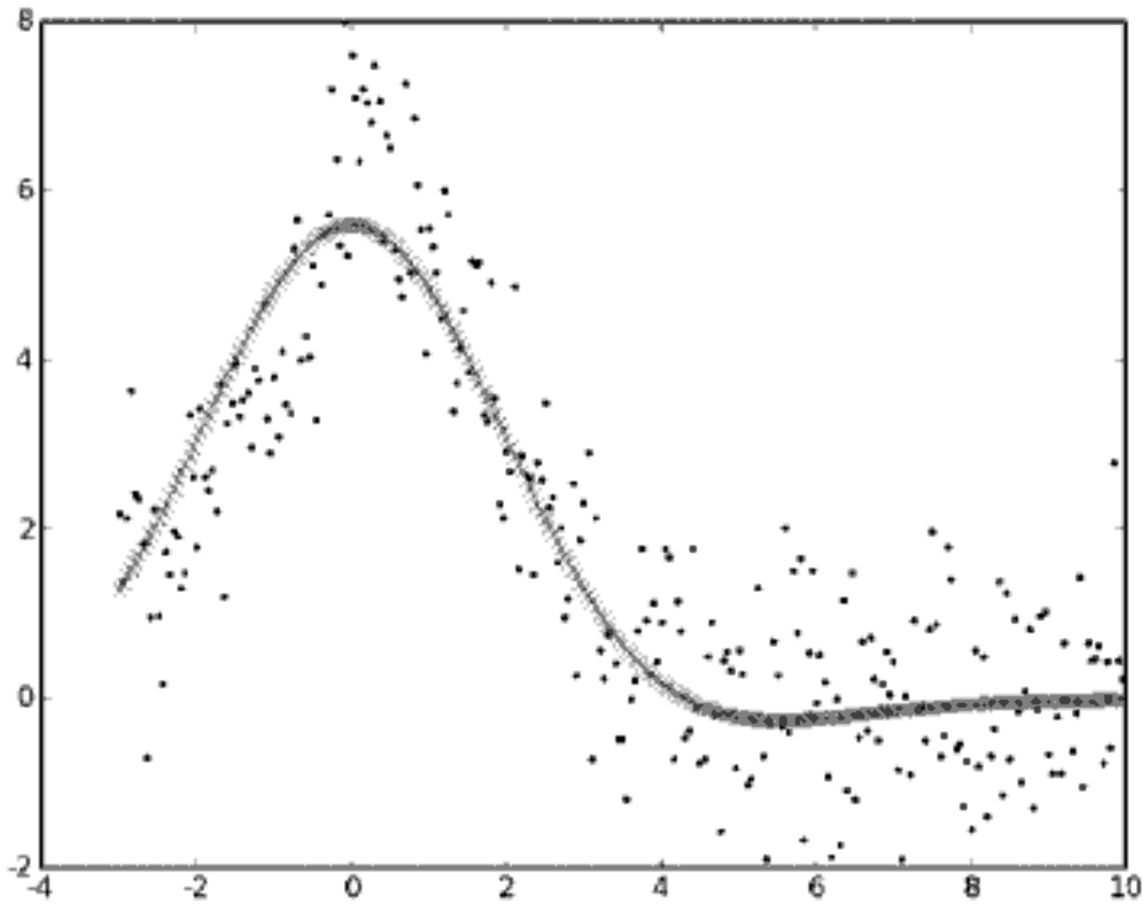


FIGURE 4.13: Using linear least-squares to fit parameters for two Gaussians produces the line from the noisy datapoints plotted as circles.

```
(p, residuals, rank, s) = linalg.lstsq(A,y)

plot(x,y,'.')
plot(x,p[0]*A[:,0]+p[1]*A[:,1],'x')

p
>>> array([ 2.00101406,  3.09626831])
```

4.4.4 Smoothing Splines

The way that we constructed the splines in Section 4.4 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 4.13 went through each datapoint. As we try to make the spline model match the data more and more accurately, we will add further knotpoints in, 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

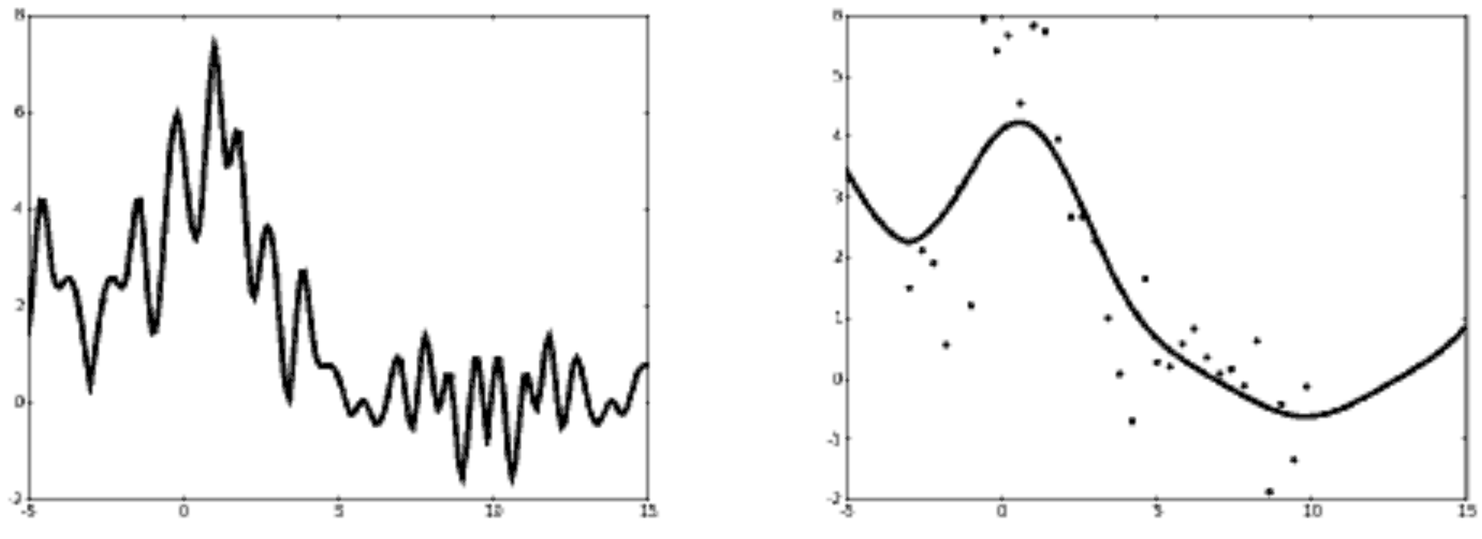


FIGURE 4.14: B-spline fitting of the data shown in Figure 4.13 with *left:* $\lambda = 0$ and *right:* $\lambda = 100$.

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 trade-off between the two parts. We regain the interpolating spline of Section 4.4 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:

$$E(y, f(x), \lambda) = \sum_{i=1}^{N} (y_i - f(x_i))^2 + \lambda \int \left(\frac{d^2 f}{dt^2} \right)^2 dt. \tag{4.6}$$

SciPy already has functions to perform this in Python, the output of two different values of the smoothing parameter are shown in Figure 4.14.

```
# Fit spline
spline = cspline1d(y,100)
xbar = arange(-5,15,0.1)
# Evaluate spline
ybar = cspline1d_eval(spline, xbar, dx=x[1]-x[0],x0=x[0])
```

4.4.5 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,

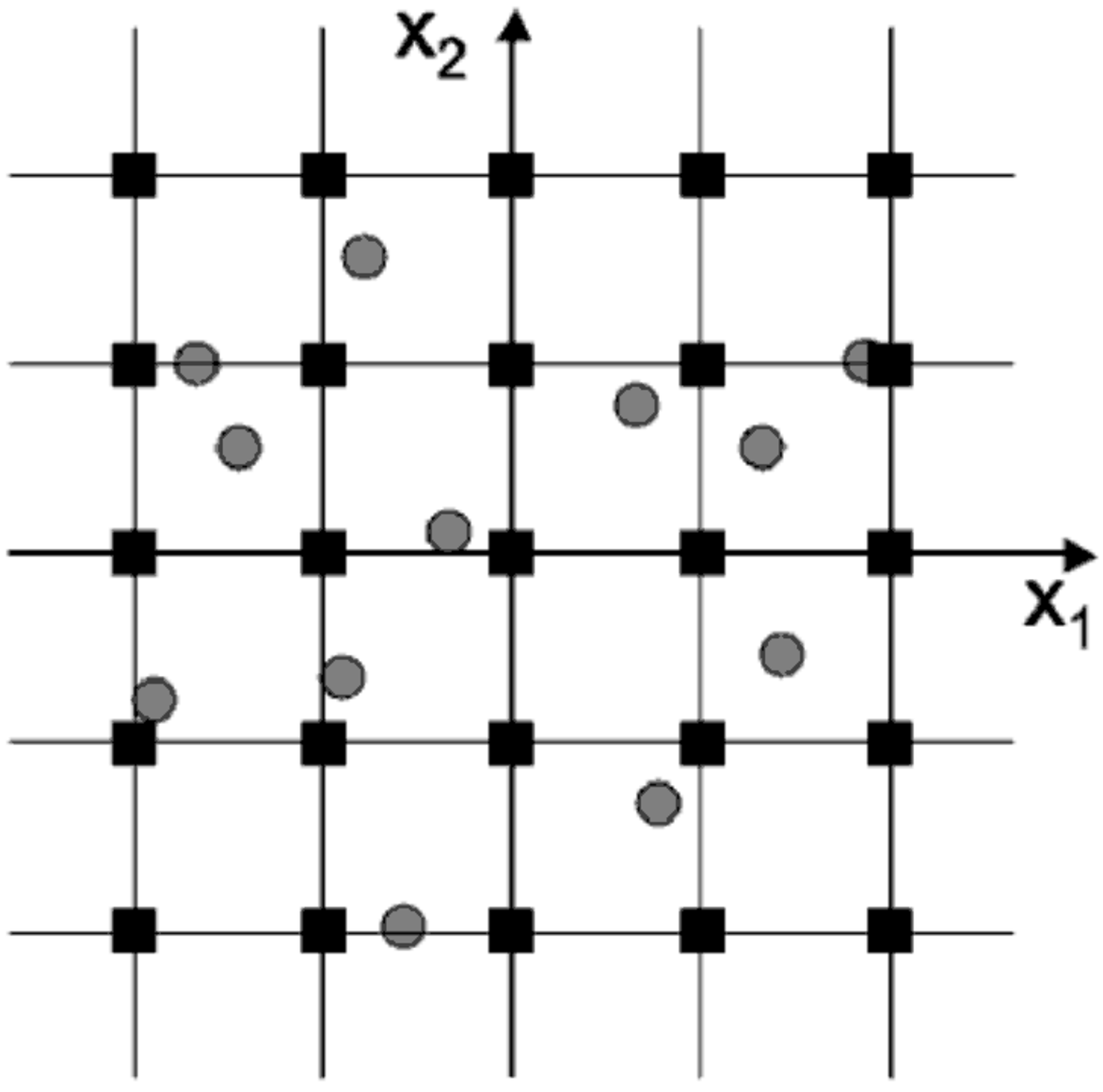


FIGURE 4.15: In 2D the knotpoints (black squares) can be used to interpolate other points (grey circles) in each dimension individually.

where the B-spline is built up in this way. Figure 4.15 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 (4.6)? 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:

$$\int\int_{\mathbb{R}^2} \left(\frac{\partial^2 f}{\partial x_1^2}\right)^2 + 2\left(\frac{\partial^2 f}{\partial x_1 \partial x_2}\right)^2 + \left(\frac{\partial^2 f}{\partial x_2^2}\right)^2 dx_1 dx_2. \tag{4.7}$$

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.

4.4.6 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.* Springer, Berlin, Germany, 2001.
- Chapter 5 of S. Haykin. *Neural Networks: A Comprehensive Foundation.* Prentice-Hall, New Jersey, USA, 2nd edition, 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 4.1 Create an RBF network that solves the XOR function.

Problem 4.2 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 4.3 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 4.4 The following function creates some noisy data from a sinusoidal function:

```
def gendata(npoints):
    x = arange(0,4*pi,1./npoints)

    data = x*sin(x) + random.normal(0,2,size(x))
    print data
    plot(x,x*sin(x),'k-',x,data,'k.')
    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 4.5 Implement the B-spline in 2D by convolving two 1D cubic splines in orthogonal directions. Can you use it to warp images?

Chapter 5

Support Vector Machines

Back in Chapter 2 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 the difference between the two truth classes of the 2D XOR function. However, in Section 2.3.2, 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 4.4, 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 2.3.2: we added in 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. They were introduced by Vapnik in 1992 and have taken off radically since then, principally because they often (but not always) provide significantly better classification performance than other machine learning algorithms on reasonably sized datasets (they do not work well on extremely large datasets, since they involve a data matrix inversion, which is computationally very expensive). This should be sufficient motivation to master the (quite complex) concepts that are needed to understand the algorithm. We won't be using any code in this chapter, since implementing an SVM is probably something you wouldn't want to implement for yourself: some of the details of the algorithm, particularly the optimisation routine, are difficult to implement. There are several different implementations of the SVM available on the Internet,

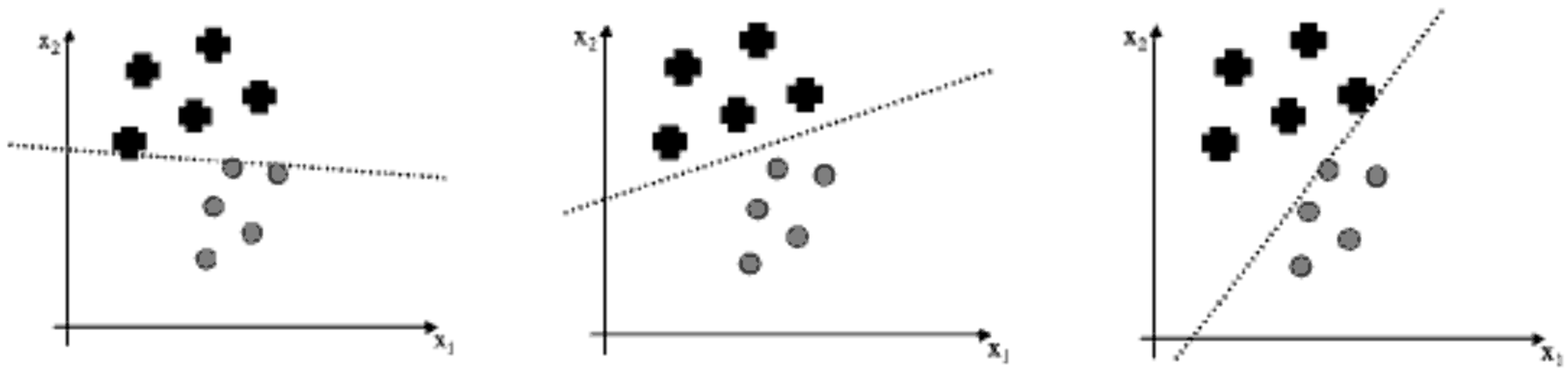


FIGURE 5.1: Three different classification lines. Is there any reason why one is better than the others?

and there are references to some of the more popular ones at the end of the chapter. Some of them include wrappers for Python 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 these advanced algorithms to be derived, so that is where we will start.

5.1 Optimal Separation

Figure 5.1 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.' 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 5.1 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.'

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 5.2), 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 classifier in the middle of Figure 5.1 has the largest margin of the three. It 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 a 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 data except for the support vectors, and use them for classification.

Now that we've got a measurement that we can use to find the optimal classification line, 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 2. We can use the standard equation of a straight line to write down our classifier, it is $y = \mathbf{w} \cdot \mathbf{x} + b$, where we are using the same notation as in Chapter 2: the $\mathbf{w}$ is the weight vector (it is a vector, not a matrix, since there is only one output) and $\mathbf{x}$ is the particular input vector, 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 'o' 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. Remember that $\mathbf{w} \cdot \mathbf{x}$ is the inner or scalar product, $\mathbf{w} \cdot \mathbf{x} = \sum_i w_i x_i$.

For a given margin value M we can say that any point $\mathbf{x}$ where $\mathbf{w} \cdot \mathbf{x} + b \geq M$ is a plus, and any point where $\mathbf{w} \cdot \mathbf{x} + b \leq -M$ is a circle. Now suppose that we pick a point $\mathbf{x}^+$ that lies on the '+' class boundary line, so that $\mathbf{w} \cdot \mathbf{x}^+ = M$. This is a support vector. If we want to find the closest point that lies on the boundary line for the 'o' class, then we travel perpendicular to the '+'

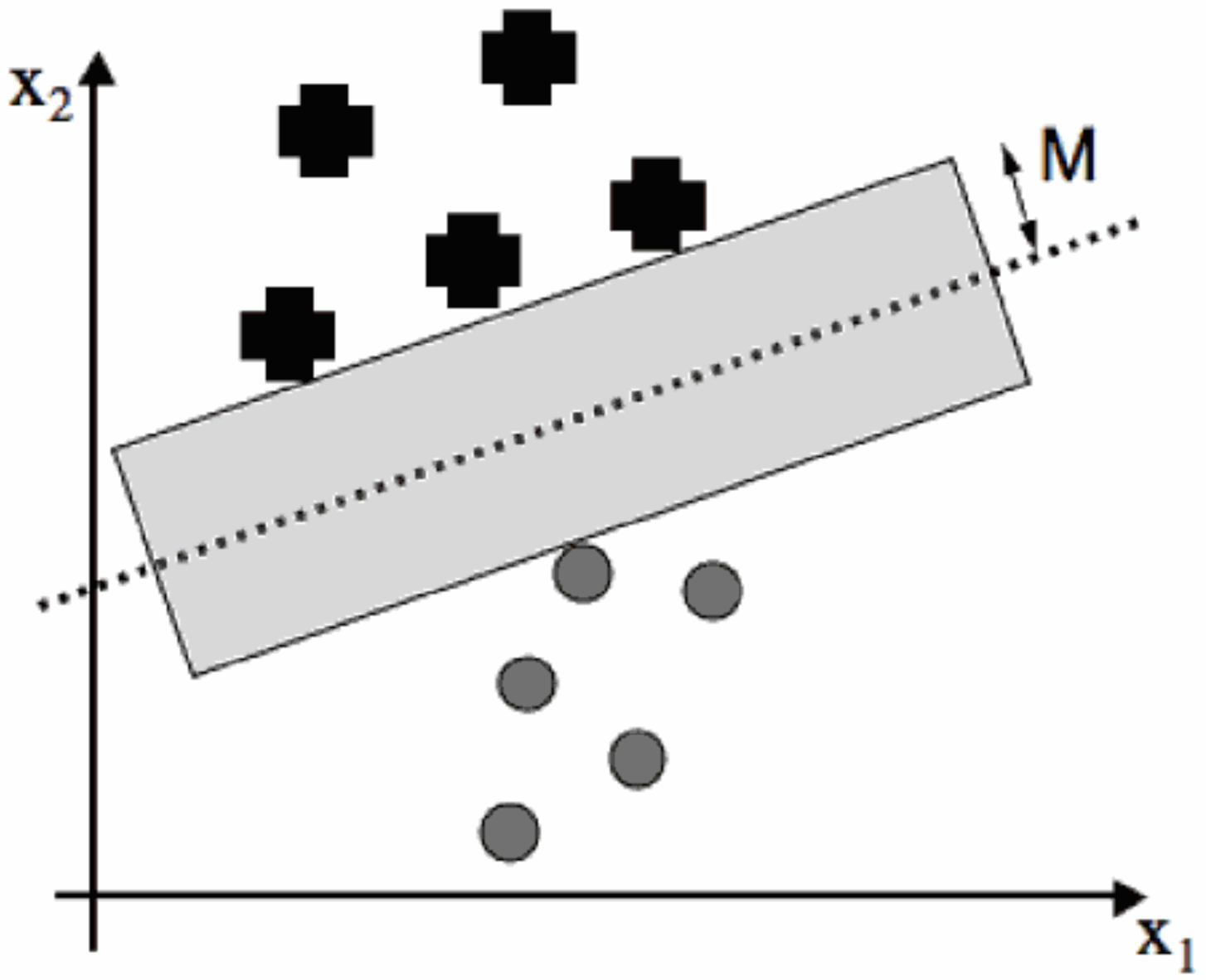


FIGURE 5.2: The margin is the largest region we can put that separates the classes without there being any points inside.

boundary line until we hit the 'o' 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 5.2 hopefully makes it clear that the distance we travelled is $2M$. 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 2, 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 'o' boundary lines too, so the direction we travelled in from $\mathbf{x}^+$ to $\mathbf{x}^-$ is along $\mathbf{w}$, or writing it as an equation (where ν is some distance along the line):

$$\mathbf{x}^- = \mathbf{x}^+ + \nu\mathbf{w}. \tag{5.1}$$

We know that $|\mathbf{x}^- - \mathbf{x}^+| = 2M$, and so we can use the equation above to compute that:

$$M = \frac{1}{2|\mathbf{w}|} = \frac{1}{2\sqrt{\mathbf{w} \cdot \mathbf{w}}}. \tag{5.2}$$

So now, given a classifier line (that is, the vector $\mathbf{w}$ and scalar b that define the line $\mathbf{w} \cdot \mathbf{x} + b$) 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. Equation (5.2) tells us that making M as large as possible is the same as making $\mathbf{w} \cdot \mathbf{w}$ as small as possible. If

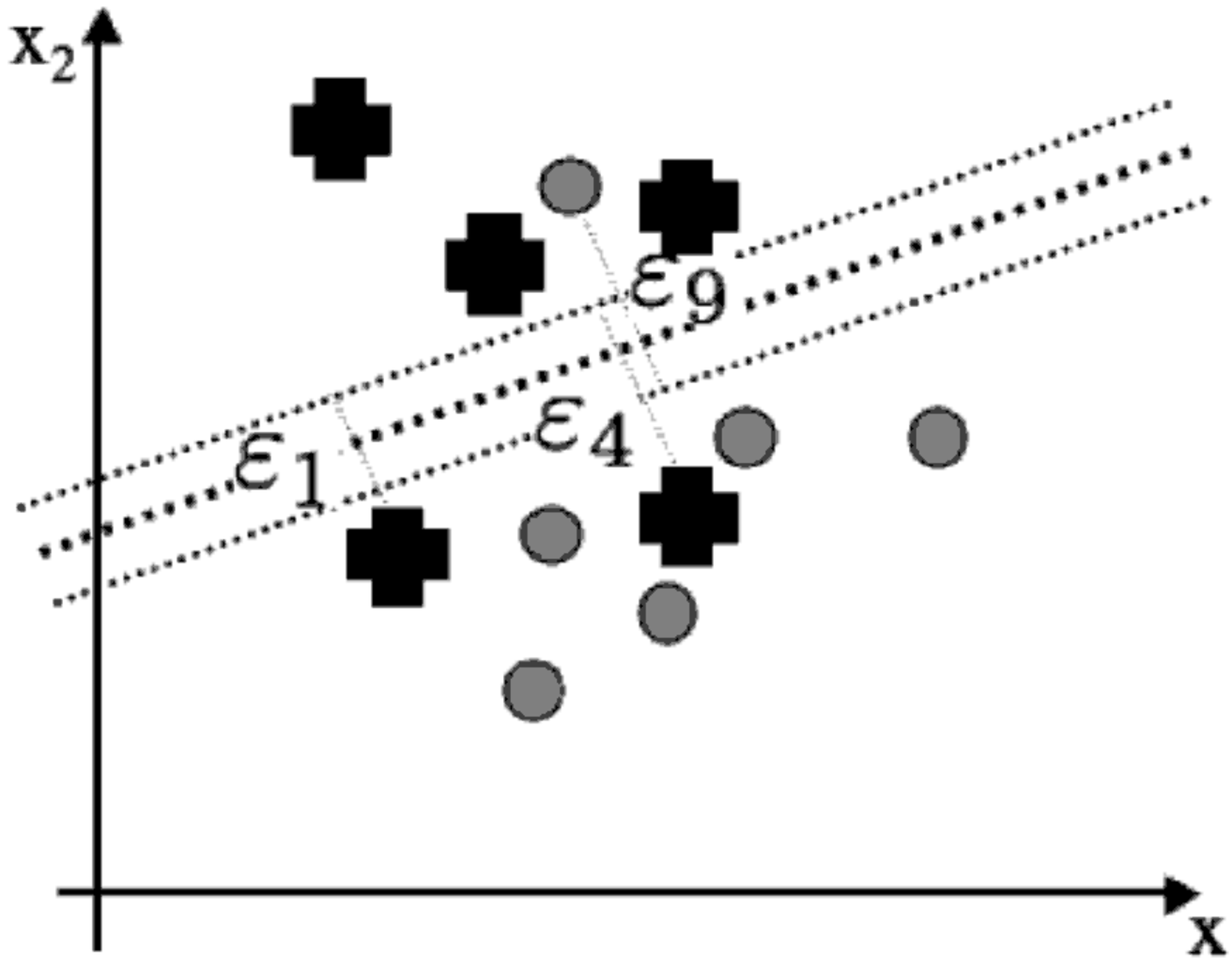


FIGURE 5.3: If the classifier makes some errors, then the distance by which the points are over the border should be used to weight each error in order to decide how bad the classifier is.

that was the only constraint, then we could just set $\mathbf{w} = \mathbf{0}$, and problem would be solved, but we also want the classification line to separate out the '+' data from the '∘', 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 $\mathbf{w} \cdot \mathbf{w}$ as small as possible.

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 $t_i(\mathbf{w} \cdot \mathbf{x} + b) \geq 1$.

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. It can be argued that 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 $\mathbf{w} \cdot \mathbf{w} + \lambda \times$ (distance of misclassified points from the correct boundary line). Here, λ is a trade-off parameter that decides how much weight to put onto each of the two criteria—small λ means

we prize a large margin over a few errors, while large λ means the opposite. This actually 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:

$$L(\mathbf{w}, \boldsymbol{\epsilon}) = \mathbf{w} \cdot \mathbf{w} + \lambda \sum_{i=1}^{R} \epsilon_i, \tag{5.3}$$

where R is the number of misclassified data points, and each ϵ_i is the distance to the correct boundary line for the missing point, which is sometimes known as a slack variable. The constraints don't quite work anymore, either, since they don't mention the possibility of getting something wrong. We now want to say for each point that $\mathbf{w} \cdot \mathbf{x}_i \geq 1 - \epsilon_i$ if the target is 1 and $\mathbf{w} \cdot \mathbf{x}_i \leq -1 + \epsilon_i$ if the target is -1. The other thing to notice is that ϵ_i is a distance, and therefore has to be a positive number. This has to be specified as an additional constraint for each i.

We've made a lot of effort to write down this equation, but we don't know how to solve it. We could 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. If you want to understand these terms, and don't, then a book on numerical optimisation would be a good start; a couple are given in the references at the end of the chapter. The practical upshot of these facts for us is that the problem can be solved directly and efficiently (i.e., in polynomial time) for the problems that we wish to solve. There are very effective quadratic programming solvers available, but it is not an algorithm that we will consider writing ourselves, being well beyond the scope of this book.

We will, however, work out how to formulate the problem so that it can be presented to a quadratic program solver. This involves transforming the form of the problem by using the technique of Lagrange multipliers, which means that we treat λ in Equation (5.3) as a parameter instead of a constant, and find the minimiser by looking at the derivatives of the function with respect to each of the parameters independently and setting them to zero. The method of Lagrange multipliers is a very useful one for optimisation, and good introductions to the method can be found in many textbooks.

There is another modification to the problem that we will make as well, which is to change it into a maximisation problem by finding its dual. This is an alternative representation of a constraint problem that has the same solution, but swaps the constraints and the objective, producing a version of the problem that is more efficient for quadratic optimisation, and will have certain other benefits later on. Constructing the dual function requires that we use the Karush-Kuhn-Tucker construction to eliminate the $\mathbf{w}$ from Equation (5.3)

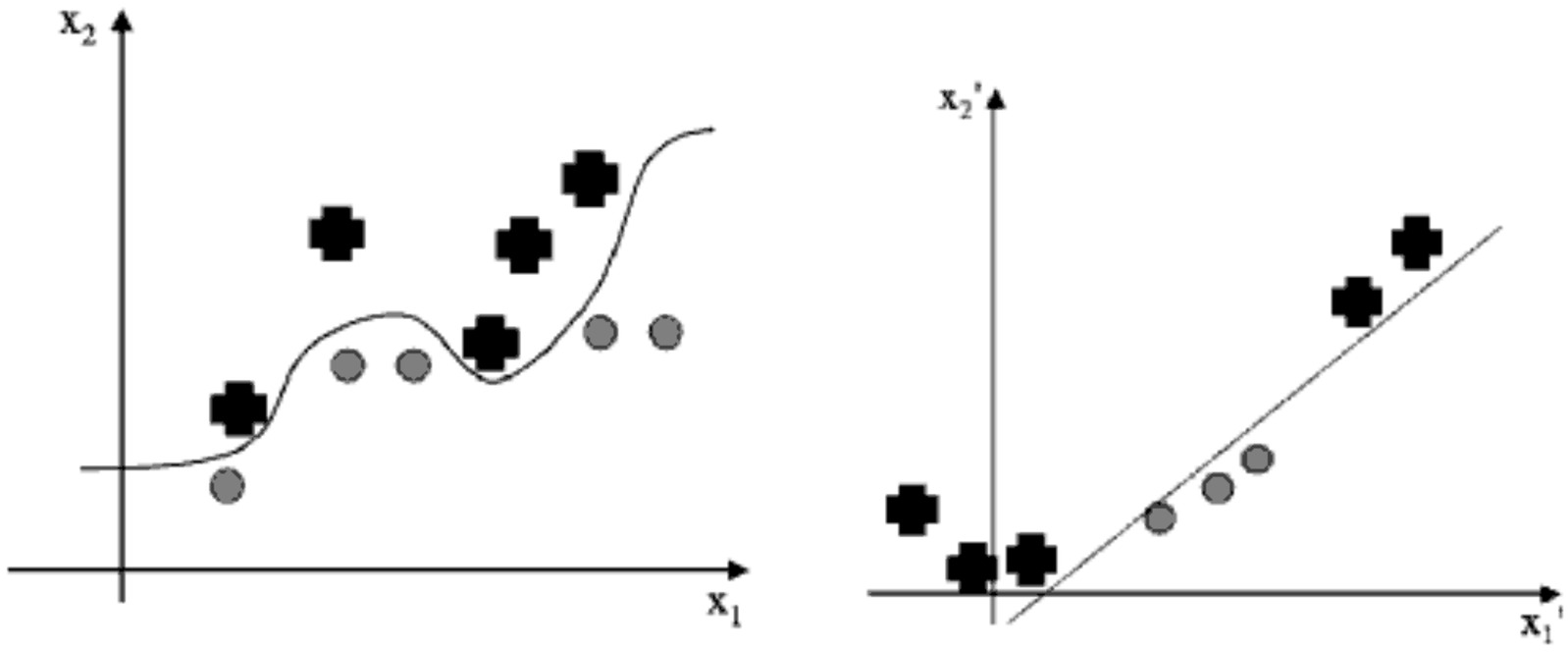


FIGURE 5.4: By modifying the features we hope to find spaces where the data are linearly separable.

(by differentiating with respect to it and setting the derivatives equal to 0) to get:

$$L(\boldsymbol{\epsilon}) = \max \sum_{i=1}^{R} \alpha_i - \frac{1}{2} \sum_{i=1}^{R} \sum_{j=1}^{R} \alpha_i \alpha_j t_i t_j \mathbf{x}_i \cdot \mathbf{x}_j, \tag{5.4}$$

subject to the constraints $0 \leq \alpha_i \leq \lambda$ and $\sum_{i=1}^{R} \alpha_i \mathbf{x}_k = 0$.

5.2 Kernels

Although we've done lots of work up to this point, we haven't actually changed things very much, because we are still finding straight line boundary conditions in the input space of the data. So while the decision boundary that is found could be better than that found by the Perceptron, if there isn't a straight line solution then, just like the Perceptron, our current method won't work. 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. To see the idea, have a look at Figure 5.4. It is basically the idea 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; if we can use more dimensions then we might be able to find a linear decision boundary that separates the classes. What extra dimensions can we 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 4.4, we are going to introduce new functions $\phi(\mathbf{x})$ of our input features.

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

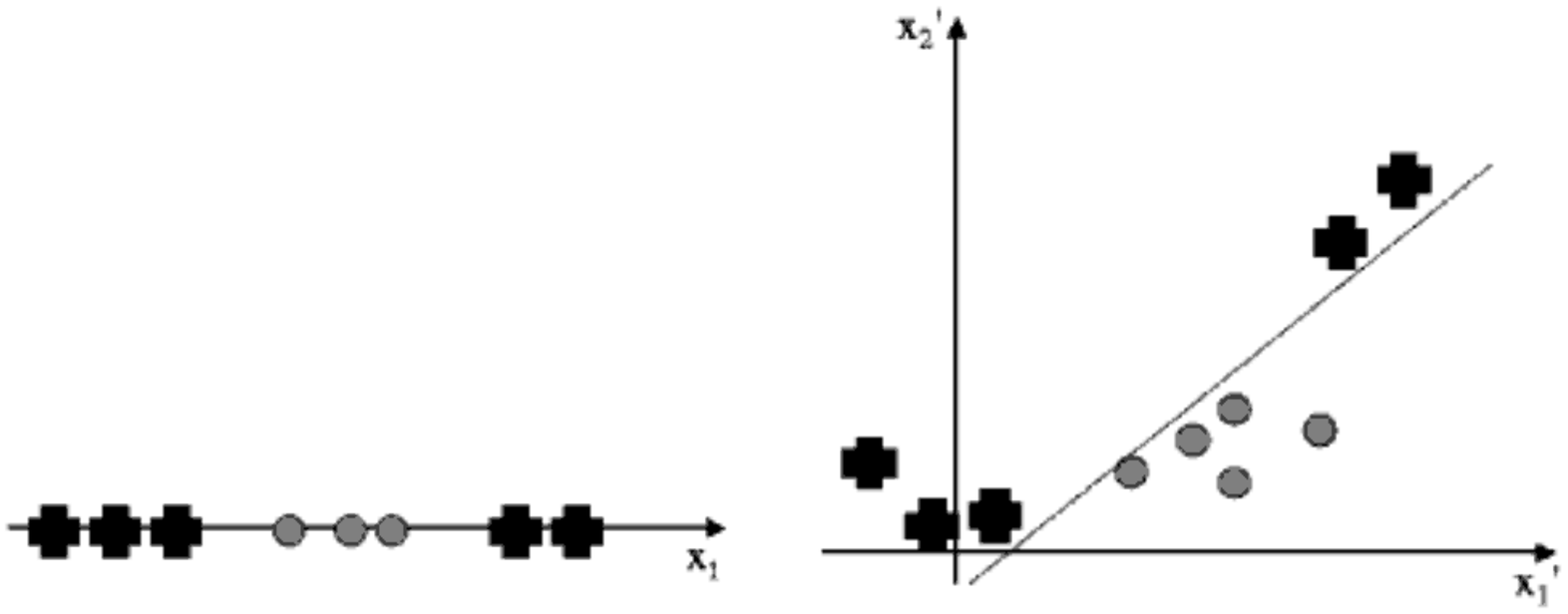


FIGURE 5.5: Using x_1^2 as well as x_1 allows these two classes to be separated.

around, and we would like to automate the algorithm. For now, let's think about a basis that consists of the polynomial of everything up to degree 2. It contains the constant value 1, each of the individual (scalar) 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 5.5 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):

$$\Phi(\mathbf{x}) = (1, \sqrt{2}x_1, \sqrt{2}x_2, \sqrt{2}x_3, x_1^2, x_2^2, x_3^2, \sqrt{2}x_1x_2, \sqrt{2}x_1x_3, \sqrt{2}x_2x_3). \quad (5.5)$$

If there was just one feature, x_1, then we would have changed this from a one-dimensional problem into a three-dimensional one $(1, x_1, x_1^2)$.

Have another look at Equation (5.4). There is no reason why we can't modify it so that the $\mathbf{x}$ variables look like $\Phi(\mathbf{x}_i) \cdot \Phi(\mathbf{x}_j)$. (Note the slight notational intricacy here: $\mathbf{x}_i$ is the ith input vector, while x_i is the ith element of an input vector.) This seems great, since we don't have to modify our derivation of that equation at all. Except that now 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 R^2 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 $\Phi(\mathbf{x}_i) \cdot \Phi(\mathbf{x}_j)$. To see how this works, let's work out what $\Phi(\mathbf{x}) \cdot \Phi(\mathbf{y})$ actually is:

$$\Phi(\mathbf{x}) \cdot \Phi(\mathbf{y}) = 1 + 2\sum_{i=1}^{d} x_i y_i + \sum_{i=1}^{d} x_i^2 y_i^2 + 2\sum_{i=1}^{m} x_i x_j y_i y_j. \quad (5.6)$$

You might not recognise that you can factorise this equation, but fortunately somebody did: it can be written as $(1 + \mathbf{x} \cdot \mathbf{y})^2$. 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.

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:

$$\mathbf{K}(\mathbf{x}, \mathbf{y}) = (1 + \mathbf{x} \cdot \mathbf{y})^s \tag{5.7}$$

- sigmoid functions of the x_ks with parameters κ and δ, and kernel:

$$\mathbf{K}(\mathbf{x}, \mathbf{y}) = \tanh(\kappa \mathbf{x} \cdot \mathbf{y} - \delta) \tag{5.8}$$

- radial basis function expansions of the x_ks with parameter σ and kernel:

$$\mathbf{K}(\mathbf{x}, \mathbf{y}) = \exp\left(-(\mathbf{x} - \mathbf{y})^2 / 2\sigma^2\right) \tag{5.9}$$

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 3.

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 $\mathbf{w} \cdot \Phi(\mathbf{x})$, where:

$$\mathbf{w} = \sum_{i \text{ where } \alpha_i > 0} \alpha_i y_i \Phi(\mathbf{x}_i). \tag{5.10}$$

So we still have the computation of $\Phi(\mathbf{x}_i) \cdot \Phi(\mathbf{x}_j)$, which we can replace using the kernel as before.

The overfitting problem goes away because of the fact that we are still optimising $\mathbf{w} \cdot \mathbf{w}$ (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.

5.2.1 Example: XOR

We motivated the SVM by thinking about how we solved the XOR function in Section 2.3.2. So will the SVM actually solve the problem? We'll need to modify the problem to have targets -1 and 1 rather than 0 and 1, but that is not difficult. Then we'll introduce a basis of all terms up to quadratic in our two features: $1, \sqrt{2}x_1, \sqrt{2}x_2, x_1x_2, x_1^2, x_2^2$, where the $\sqrt{2}$ is to keep the multiplications simple. Then Equation (5.4) looks like:

$$\sum_{i=1}^{4} \alpha_i - \sum_{i,j}^{4} \alpha_i \alpha_j t_i t_j \Phi(\mathbf{x}_i) \cdot \Phi(\mathbf{x}_j), \tag{5.11}$$

subject to the constraints that $\alpha_1 - \alpha_2 + \alpha_3 - \alpha_4 = 0, \alpha_i \geq 0 \;\; i = 1 \ldots 4$. Solving this (which can be done algebraically) tells us that the classifier line is at $x_1x_2 = 0$. The margin that corresponds to this is $\sqrt{2}$. Unfortunately we can't plot it, since our four points have been transferred into a six-dimensional space. We know that this is not the smallest number that it can be solved in, since we did it in three, but the dimensionality of the kernel space doesn't matter, as all the computations are in the 2D space anyway.

5.2.2 Extensions to the Support Vector Machine

We've talked about SVMs in terms of two-class classification. You might be wondering how to use them for more classes, since we can't use the same methods as we have done to work out the current algorithm. In fact, you can't actually do it in a consistent way. The SVM only works for two classes. This might seem like a major problem, but with a little thought it is possible to find ways around the problem. For the problem of N-class classification,

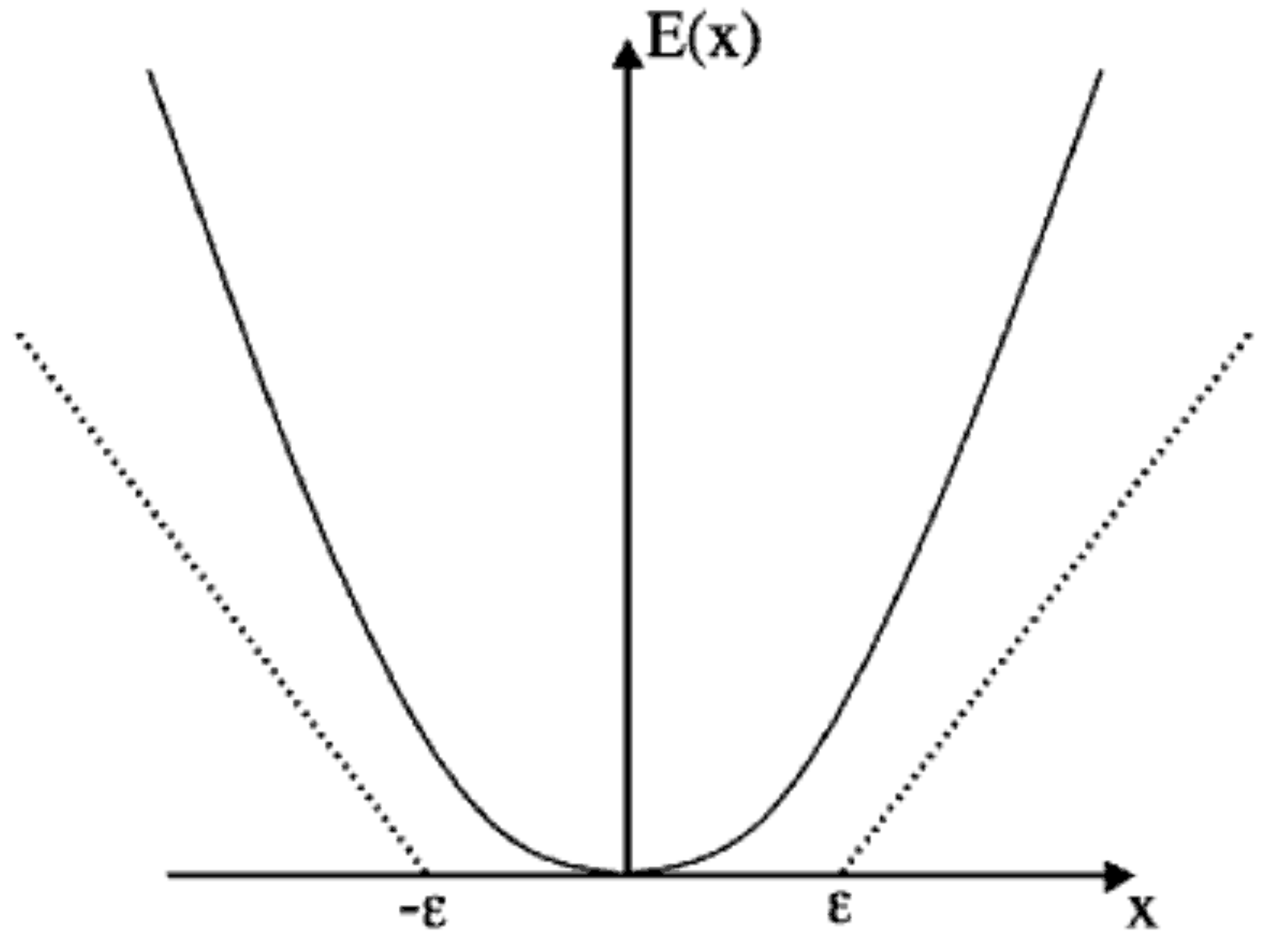


FIGURE 5.6: The ϵ-insensitive error function is zero for any error below ϵ.

you train an SVM that learns to classify class one from all other classes, then another that classifies class two from all the others. So for N-classes, we have N SVMs. This still leaves one problem: how do we decide which of these SVMs is the one that recognises the particular input? The answer is just to choose the one that makes the strongest prediction, that is, the one where the basis vector input point is the furthest into the positive class region.

Interestingly, it is also possible to use the SVM for regression. The key is to take the usual least-squares error function (with the regulariser that keeps the norm of the weights small):

$$\frac{1}{2}\sum_{i=1}^{N}(t_i - y_i)^2 + \frac{1}{2}\lambda\|\mathbf{w}\|^2, \tag{5.12}$$

and transform it using what is known as a ϵ-insensitive error function (E_ϵ) that gives 0 if the difference between the target and output is less than ϵ (and subtracts ϵ in any other case for consistency). Figure 5.6 shows the form of this error function, which is:

$$\sum_{i=1}^{N} E_\epsilon(t_i - y_i) + \lambda\frac{1}{2}\|\mathbf{w}\|^2. \tag{5.13}$$

You might see this written in other texts with the constant λ in front of the second term replaced by a C in front of the first term. This is equivalent up to scaling. The picture to think of now is almost the opposite of Figure 5.3: we want the predictions to be inside the tube of radius ϵ that surrounds the correct line. To allow for errors, we again introduce slack variables for each datapoint (ϵ_i for datapoint i) with their constraints and follow the same procedure of

introducing Lagrange multipliers, transferring to the dual problem, using a kernel function and solving the problem with a quadratic solver.

There is a lot of advanced work on kernel methods and SVMs. This includes lots of work on the optimisation, including Sequential Minimal Optimisation, and extensions to compute posterior probabilities instead of hard decisions, such as the Relevance Vector Machine. There are some references in the Further Reading section.

There are SVM implementations available via the Internet. They are mostly written in C, but some include wrappers to be called from other languages, including Python. An Internet search will find you some possibilities to try.

Further Reading

The treatment of SVMs here has only skimmed the surface of the topic. There is a useful tutorial paper on SVMs at:

- C.J. Burges. A tutorial on support vector machines for pattern recognition. *Data mining and knowledge discovery*, 2(2): 121–167, 1998.

If you want more information, then any of the following books will provide it (the first is by the creator of SVMs):

- V. Vapnik. *The Nature of Statistical Learning Theory*. Springer, Berlin, Germany, 1995.
- B. Schölkopf, C.J.C. Burges, and A.J. Smola. *Advances in Kernel Methods: Support Vector Learning*. MIT Press, Cambridge, MA, USA, 1999.
- J. Shawe-Taylor and N. Cristianini. *Kernel Methods for Pattern Analysis*. Cambridge University Press, Cambridge, UK, 2004.

If you want to know more about quadratic programming, then a good reference is:

- S. Boyd and L. Vandenberghe. *Convex Optimization*. Cambridge University Press, Cambridge, UK, 2004.

Other machine learning books that give useful coverage of this area are:

- Chapter 12 of T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*. Springer, Berlin, Germany, 2001.
- Chapter 7 of C.M. Bishop. *Pattern Recognition and Machine Learning*. Springer, Berlin, Germany, 2006.

Practice Questions

Problem 5.1 Suppose that the following are a set of points in two classes:

$$\text{class } 1 : \begin{pmatrix} 1 \\ 1 \end{pmatrix} \begin{pmatrix} 1 \\ 2 \end{pmatrix} \begin{pmatrix} 2 \\ 1 \end{pmatrix} \tag{5.14}$$

$$\text{class } 2 : \begin{pmatrix} 0 \\ 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} \tag{5.15}$$

Plot them and find the optimal separating line. What are the support vectors, and what is the margin?

Problem 5.2 Suppose that the points are now:

$$\text{class } 1 : \begin{pmatrix} 0 \\ 0 \end{pmatrix} \begin{pmatrix} 1 \\ 2 \end{pmatrix} \begin{pmatrix} 2 \\ 1 \end{pmatrix} \tag{5.16}$$

$$\text{class } 2 : \begin{pmatrix} 1 \\ 1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} \tag{5.17}$$

Try out the different basis functions that were given in the chapter to see which separate this data and which do not.

Problem 5.3 Download one of the SVM packages from the Internet and practice using it. In particular, apply it to the `wine` dataset. Compare the results to using an MLP. Do the same for the `yeast` dataset.

Problem 5.4 Use an SVM on the MNIST dataset.

Chapter 6

Learning with Trees

We are now going to move away from neural networks and take a rather different approach, starting with one of the most common and powerful data structures in the whole of computer science: the binary tree. The computational cost of making the tree is fairly low, but the cost of using it is even lower: $\mathcal{O}(\log N)$, where N is the number of data points. This is important for machine learning, since querying the trained algorithm should be as fast as possible since it happens more often, and the result is often wanted immediately. This is sufficient to make trees seem attractive for machine learning. However, they do have other benefits, such as the fact that they are easy to understand (following a tree to get a classification answer is transparent, which makes people trust it more than getting an answer from a 'black box' neural network). For these reasons, classification by decision trees has grown in popularity over recent years. You are very likely to have been subjected to decision trees if you've ever phoned a helpline, for example for computer faults. The phone operators are guided through the decision tree by your answers to their questions.

The idea of a decision tree is that we break classification down into a set of choices about each feature in turn, starting at the root (base) of the tree and progressing down to the leaves, where we receive the classification decision. The trees are very easy to understand, and can even be turned into a set of if-then rules, suitable for use in a rule induction system.

6.1 Using Decision Trees

As a student it can be difficult to decide what to do in the evening. There are four things that you actually quite enjoy doing, or have to do: going to the pub, watching TV, going to a party, or even (gasp) studying. The choice is sometimes made for you—if you have an assignment due the next day, then you need to study, if you are feeling lazy then the pub isn't for you, and if there isn't a party then you can't go to it. You are looking for a nice algorithm that will let you decide what to do each evening without having to think about it every night. Figure 6.1 provides just such an algorithm.

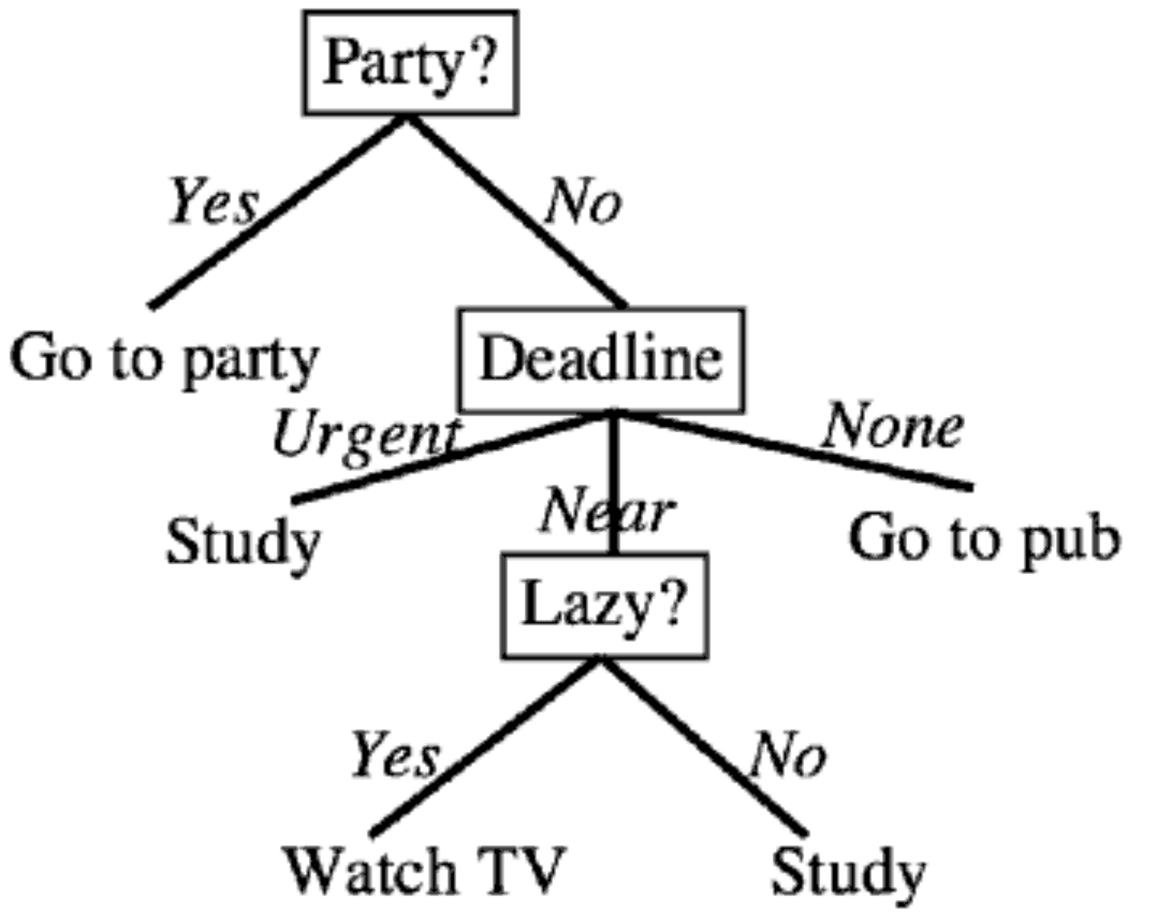


FIGURE 6.1: A simple decision tree to decide how you will spend the evening.

Each evening you start at the top (root) of the tree and check whether any of your friends know about a party that night. If there is one, then you need to go, regardless. Only if there is not a party do you worry about whether or not you have an assignment deadline coming up. If there is a crucial deadline, then you have to study, but if there is nothing that is urgent for the next few days, you think about how you feel. A sudden burst of energy might make you study, but otherwise you'll be slumped in front of the TV indulging your secret love of Shortland Street (or other soap opera of your choice) rather than studying. Of course, near the start of the semester when there are no assignments to do, and you are feeling rich, you'll be in the pub.

One of the reasons that decision trees are popular is that we can turn them into a set of logical disjunctions (`if ... then` rules) that then go into program code very simply—the first part of the tree above can be turned into:

- `if` *there is a party* `then` *go to it*
- `if` *there is not a party* `and` *you have an urgent deadline* `then` *study*
- etc.

That's all that there is to using the decision tree. The far more interesting part is how to construct the tree from data, and that is the focus of the next section.

6.2 Constructing Decision Trees

In the example above, the three features that we need for the algorithm are the state of your energy level, the date of your nearest deadline, and whether

or not there is a party tonight. The question we need to ask is how, based on those features, we can construct the tree. There are a few different decision tree algorithms, but they are almost all variants of the same principle: the algorithms build the tree in a greedy manner starting at the root, choosing the most informative feature at each step. We are going to start by focusing on the most common: Quinlan's ID3, although we'll also mention its extension, known as C4.5, and another known as CART.

There was an important word hidden in the sentence above about how the trees work, which was informative. Choosing which feature to use next in the decision tree can be thought of like playing the game '20 Questions,' where you try to elicit the item your opponent is thinking about by asking questions about it. At each stage, you choose a question that gives you the most information given what you know already. Thus, you would ask 'Is it an animal?' before you asked 'Is it a cat?'. The idea is to quantify this question of how much information is provided to you by knowing certain facts. Encoding this mathematically is the task of information theory.

6.2.1 Quick Aside: Entropy in Information Theory

Information theory was 'born' in 1948 when Claude Shannon published a paper called "A Mathematical Theory of Communication." In that paper, he proposed the measure of information entropy, which describes the amount of impurity in a set of features. The entropy H of a set of probabilities p_i is (for those who know some physics, the relation to physical entropy should be clear):

$$H(p) = -\sum_i p_i \log_2 p_i, \tag{6.1}$$

where the logarithm is base 2 because we are imagining that we encode everything using binary digits (bits), and we define $0 \log 0 = 0$. A graph of the entropy is given in Figure 6.2. Suppose that we have a set of positive and negative examples of some feature (where the feature can only take 2 values: positive and negative). If all of the examples are positive, then we don't get any extra information from knowing the value of the feature for any particular example, since whatever the value of the feature, the example will be positive. Thus, the entropy of that feature is 0. However, if the feature separates the examples into 50% positive and 50% negative, then the amount of entropy is at a maximum, and knowing about that feature is very useful to us. The basic concept is that it tells us how much *extra* information we would get from knowing the value of that feature. A function for computing the entropy is very simple, as here:

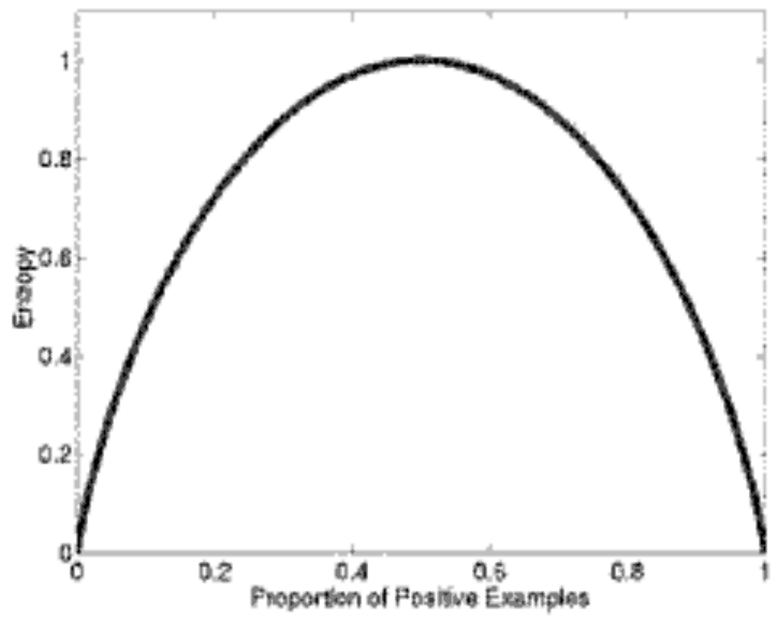


FIGURE 6.2: A graph of entropy, detailing how much information is available from finding out another piece of information given what you already know.

```
def calc_entropy(p):

    if p!=0:
        return -p * log2(p)
    else:
        return 0
```

For our decision tree, the best feature to pick as the one to classify on now is the one that gives you the most information, i.e., the one with the highest entropy. After using that feature, we re-evaluate the entropy of each feature and again pick the one with the highest entropy.

Information theory is a very interesting subject. It is possible to download Shannon's 1948 paper from the Internet, and also to find many resources showing where it has been applied. There are now whole journals devoted to information theory because it is relevant to so many areas such as computer and telecommunication networks, machine learning, and data storage. Some further readings in the area are given at the end of the chapter.

6.2.2 ID3

Now that we have a suitable measure for choosing which feature to choose next, entropy, we just have to work out how to apply it. The important idea is to work out how much the entropy of the whole training set would decrease if we choose each particular feature for the next classification step. This is known as the information gain, and it is defined as the entropy of the whole set minus the entropy when a particular feature is chosen. This is defined by (where S is the set of examples, F is a possible feature out of the set of all possible ones, and $|S_f|$ is a count of the number of members of S that have value f for feature F):

$$\text{Gain}(S, F) = \text{Entropy}(S) - \sum_{f \in values(F)} \frac{|S_f|}{|S|} \text{Entropy}(S_f). \tag{6.2}$$

As an example, suppose that we have data (with outcomes) $S = \{s_1 = \text{true}, s_2 = \text{false}, s_3 = \text{false}, s_4 = \text{false}\}$ and one feature F that can have values $\{f_1, f_2, f_3\}$. In the example, the feature value for s_1 could be f_2, for s_2 it could be f_2, for s_3, f_3 and for s_4, f_1 then we can calculate the entropy of S as (where $\oplus$ means true, of which we have one example, and $\ominus$ means false, of which we have three examples):

$$\begin{aligned}
\text{Entropy}(S) &= -p_\oplus \log_2 p_\oplus - p_\ominus \log_2 p_\ominus \\
&= -\frac{1}{4} \log_2 \frac{1}{4} - \frac{3}{4} \log_2 \frac{3}{4} \\
&= 0.5 + 0.311 = 0.811.
\end{aligned} \tag{6.3}$$

If you were trying to follow that calculation on a calculator, you might be wondering how to compute $\log_2 p$. The answer is to use the identity $\log_2 p = \ln p / \ln(2)$, where ln is the natural logarithm, which your calculator can produce. NumPy has the `log2()` function.

We now want to compute the information gain of F, so we now need to compute each of the values inside the summation in Equation (6.2), $\frac{|S_f|}{|S|}\text{Entropy}(S)$:

$$\begin{aligned}
\frac{|S_{f_1}|}{|S|}\text{Entropy}(S_{f_1}) &= \frac{1}{4} \times \left(-\frac{0}{1} \log_2 \frac{0}{1} - \frac{1}{1} \log_2 \frac{1}{1} \right) \\
&= 0
\end{aligned} \tag{6.4}$$

$$\begin{aligned}
\frac{|S_{f_2}|}{|S|}\text{Entropy}(S_{f_2}) &= \frac{2}{4} \times \left(-\frac{1}{2} \log_2 \frac{1}{2} - \frac{1}{2} \log_2 \frac{1}{2} \right) \\
&= \frac{1}{2}
\end{aligned} \tag{6.5}$$

$$\begin{aligned}
\frac{|S_{f_3}|}{|S|}\text{Entropy}(S_{f_3}) &= \frac{1}{4} \times \left(-\frac{0}{1} \log_2 \frac{0}{1} - \frac{1}{1} \log_2 \frac{1}{1} \right) \\
&= 0
\end{aligned} \tag{6.6}$$

The information gain from adding this feature is the entropy of S minus the sum of the three values above:

$$\text{Gain}(S, F) = 0.811 - (0 + 0.5 + 0) = 0.311. \tag{6.7}$$

This can be computed in an algorithm using the following function (where lots of the code is to get the relevant data):

```
def calc_info_gain(data,classes,feature):
    gain = 0
    nData = len(data)
    # List the values that feature can take
    values = []
    for datapoint in data:
        if values.count(datapoint[feature])==0:
            values.append(datapoint[feature])
    featureCounts = zeros(len(values))
    entropy = zeros(len(values))
    valueIndex = 0
    # Find where those values appear in data[feature] and the
    corresponding class
    for value in values:
        dataIndex = 0
        newClasses = []
        for datapoint in data:
            if datapoint[feature]==value:
                featureCounts[valueIndex]+=1
                newClasses.append(classes[dataIndex])
            dataIndex += 1
        # Get the values in newClasses
        classValues = []
        for aclass in newClasses:
            if classValues.count(aclass)==0:
                classValues.append(aclass)

        classCounts = zeros(len(classValues))
        classIndex = 0
        for classValue in classValues:
            for aclass in newClasses:
                if aclass == classValue:
                    classCounts[classIndex]+=1
            classIndex += 1

        for classIndex in range(len(classValues)):
            entropy[valueIndex] += calc_entropy(float(classCounts[
            classIndex])/sum(classCounts))
        gain += float(featureCounts[valueIndex])/nData * entropy[
        valueIndex]
        valueIndex += 1
    return gain
```

The ID3 algorithm computes this information gain for each feature and chooses the one that produces the highest value. In essence, that is all there is to the algorithm. It searches the space of possible trees in a greedy way by choosing the feature with the highest information gain at each stage. The output of the algorithm is the tree, i.e., a list of nodes, edges, and leaves. As with any tree in computer science, it can be constructed recursively. At each stage the best feature is selected and then removed from the dataset, and the algorithm is recursively called on the rest. The recursion stops when either there is only one class remaining in the data, or there are no features left. In the first case a leaf is added with that class as its label, while in the second the most common label in the remaining data are used.

The ID3 Algorithm

- if all examples have the same label:
  - return a leaf with that label
- else if there are no features left to test:
  - return a leaf with the most common label
- else:
  - choose the feature F that maximises the information gain of S to be the next node using Equation (6.2)
  - add a branch from the node for each possible value f in $\hat{F}$
  - for each branch:
    - calculate S_f by removing $\hat{F}$ from the set of features
    - recursively call the algorithm with S_f, to compute the gain relative to the current set of examples

---

Owing to the focus on classification for real world examples, trees are often used with text features rather than numeric values. This makes it rather difficult to use NumPy, and so the sample implementation is pretty well pure Python. It uses a feature of Python that is uncommon in other languages, which is the dictionary in order to hold the tree, which uses the braces {, }, and which is described next before we look at the decision tree implementation.

6.2.3 Implementing Trees and Graphs in Python

Trees are really just a restricted version of graphs, since they both consist of nodes and edges between the nodes. Graphs are a very useful data structure in many different areas of computer science. There are two reasonable ways to represent a graph computationally. One is as an $N \times N$ matrix, where N

is the number of nodes in the network. Each element of the matrix is a 1 if there is a link between the two nodes, and a 0 otherwise. The benefit of this approach is that it is easy to give weights to the links by changing the 1s to the values of the weights. The alternative is to store a list of nodes, following each by a list of nodes that it is linked to. Both are fairly natural in Python, with the second making use of the dictionary, a basic data structure that we have not used much, except for very simply in the decision tree (Chapter 6) that consists of a set of keys and values. For a graph, the key to each dictionary entry is the name of the node, and its value is a list of the nodes that it is connected to, as in this example:

```
graph = {'A': ['B', 'C'],'B': ['C', 'D'],'C': ['D'],'D': ['C'],
'E': ['F'],'F': ['C']}
```

That is all there is to it for creating the dictionary, and using it is not very different, since there are built-in methods to get a list of keys (`keys()`) and check if a key is in a dictionary (`in`). Code to find a path through the graph can then be written as a simple recursive function:

```
def findPath(graph, start, end, pathSoFar):
    pathSoFar = pathSoFar + [start]
    if start == end:
        return pathSoFar
    if start not in graph:
        return None
    for node in graph[start]:
        if node not in pathSoFar:
            newpath = findPath(graph, node, end, pathSoFar)
            return newpath
    return None
```

Using those methods we can now look at a Python implementation of the decision tree, which also has a recursive function call as its basis.

6.2.4 Implementation of the Decision Tree

The `make_tree()` function (which uses the `calc_entropy()` and `calc_info_gain()` functions that were described previously) looks like:

```
def make_tree(data,classes,featureNames):
    # Various initialisations
    default = classes[argmax(frequency)]
    if nData==0 or nFeatures == 0:
        # Have reached an empty branch
        return default
    elif classes.count(classes[0]) == nData:
        # Only 1 class remains
        return classes[0]
    else:
        # Choose which feature is best
        gain = zeros(nFeatures)
        for feature in range(nFeatures):
            g = calc_info_gain(data,classes,feature)
            gain[feature] = totalEntropy - g

        bestFeature = argmax(gain)
        tree = {featureNames[bestFeature]:{}}
        # Find the possible feature values
        for value in values:
            # Find the datapoints with each feature value
            for datapoint in data:
                if datapoint[bestFeature]==value:
                    if bestFeature==0:
                        datapoint = datapoint[1:]
                        newNames = featureNames[1:]
                    elif bestFeature==nFeatures:
                        datapoint = datapoint[:-1]
                        newNames = featureNames[:-1]
                    else:
                        datapoint = datapoint[:bestFeature]
                        datapoint.extend(datapoint[bestFeature+1:])
                        newNames = featureNames[:bestFeature]
                        newNames.extend(featureNames[bestFeature+1:
                        ])
                    newData.append(datapoint)
                    newClasses.append(classes[index])
                index += 1
            # Now recurse to the next level
            subtree = make_tree(newData,newClasses,newNames)
            # And on returning, add the subtree on to the tree
            tree[featureNames[bestFeature]][value] = subtree
    return tree
```

It is worth considering how ID3 generalises from training examples to the set of all possible inputs. It uses a method known as the inductive bias. The choice of the next feature to add into the tree is the one with the highest information gain, which biases the algorithm towards smaller trees, since it tries to minimise the amount of information that is left. This is consistent with a well-known principle that short solutions are usually better than longer ones (not necessarily true, but simpler explanations are usually easier to remember and understand). You might have heard of this principle as 'Occam's Razor,' although I prefer it as an acronym: KISS (Keep It Simple, Stupid). In fact, there is a sound information-theoretic way to write down this principle. It is known as the Minimum Description Length (MDL) and was proposed by Rissanen in 1989. In essence it says that the shortest description of something, i.e., the most compressed one, is the best description.

Note that the algorithm can deal with noise in the dataset, because the labels are assigned to the most common value of the target attribute. Another benefit of decision trees is that they can deal with missing data. Think what would happen if an example has a missing feature. In that case, we can skip that node of the tree and carry on without it, summing over all the possible values that that feature could have taken. This is virtually impossible to do with neural networks: how do you represent missing data when the computation is based on whether or not a neuron is firing? In the case of neural networks it is common to either throw away any datapoints that have missing data, or guess (more technically impute any missing values, either by identifying similar datapoints and using their value or by using the mean or median of the data values for that feature). This assumes that the data that is missing is randomly distributed within the dataset, not missing because of some unknown process.

Saying that ID3 is biased towards short trees is only partly true. The algorithm uses all of the features that are given to it, even if some of them are not necessary. This obviously runs the risk of overfitting, indeed it makes it very likely. There are a few things that you can do to avoid overfitting, the simplest one being to limit the size of the tree. You can also use a variant of early stopping by using a validation set and measuring the performance of the tree so far against it. However, the approach that is used in more advanced algorithms (most notably C4.5, which Quinlan invented to improve on ID3) is pruning.

There are a few versions of pruning, all of which are based on computing the full tree and reducing it, evaluating the error on a validation set. The most naïve version runs the decision tree algorithm until all of the features are used, so that it is probably overfitted, and then produces smaller trees by running over the tree, picking each node in turn, and replacing the subtree beneath every node with a leaf labelled with the most common classification of the sub-tree. The error of the pruned tree is evaluated on the validation set, and the pruned tree is kept if the error is the same as or less than the original tree, and rejected otherwise.

C4.5 uses a different method called **rule post-pruning**. This consists of taking the tree generated by ID3, converting it to a set of if-then rules, and then pruning each rule by removing preconditions if the accuracy of the rule increases without it. The rules are then sorted according to their accuracy on the training set and applied in order. The advantages of dealing with rules are that they are easier to read and their order in the tree does not matter, just their accuracy in the classification.

6.2.5 Dealing with Continuous Variables

One thing that we have not yet discussed is how to deal with continuous variables, we have only considered those with discrete sets of feature values. The simplest solution is to discretise the continuous variable. However, it is also possible to leave it continuous and modify the algorithm. For a continuous variable there is not just one place to split it: the variable can be broken between any pair of datapoints, as shown in Figure 6.3. It can, of course, be split in any of the infinite locations along the line as well, but they are no different to this smaller set of locations. Even this smaller set makes the algorithm more expensive for continuous variables than it is for discrete ones, since as well as calculating the information gain of each variable to pick the best one, the information gain of many points within each variable has to be computed. In general, only one split is made to a continuous variable, rather than allowing for threeway or higher splits, although these can be done if necessary.

The trees that these algorithms make are all **univariate** trees, because they pick one feature (dimension) at a time and split according to that one. There are also algorithms that make **multivariate** trees by picking combinations of features. This can make for considerably smaller trees if it is possible to find straight lines that separate the data well, but are not parallel to any axis. However, univariate trees are simpler and tend to get good results, so we won't consider multivariate trees any further. This fact that one feature is chosen at a time provides another useful way to visualise what the decision tree is doing. Figure 6.4 shows the idea. Given a dataset that contains three classes, the algorithm picks a feature and value for that feature to split the remaining data into two. The final tree that results from this is shown in Figure 6.5.

6.2.6 Computational Complexity

The computational cost of constructing binary trees is well known for the general case, being $\mathcal{O}(N \log N)$ for construction and $\mathcal{O}(\log N)$ for returning a particular leaf, where N is the number of nodes. However, these results are for **balanced** binary trees, and decision trees are often not balanced; while the information measures attempt to keep the tree balanced by finding splits that separate the data into two even parts (since that will have the largest entropy),

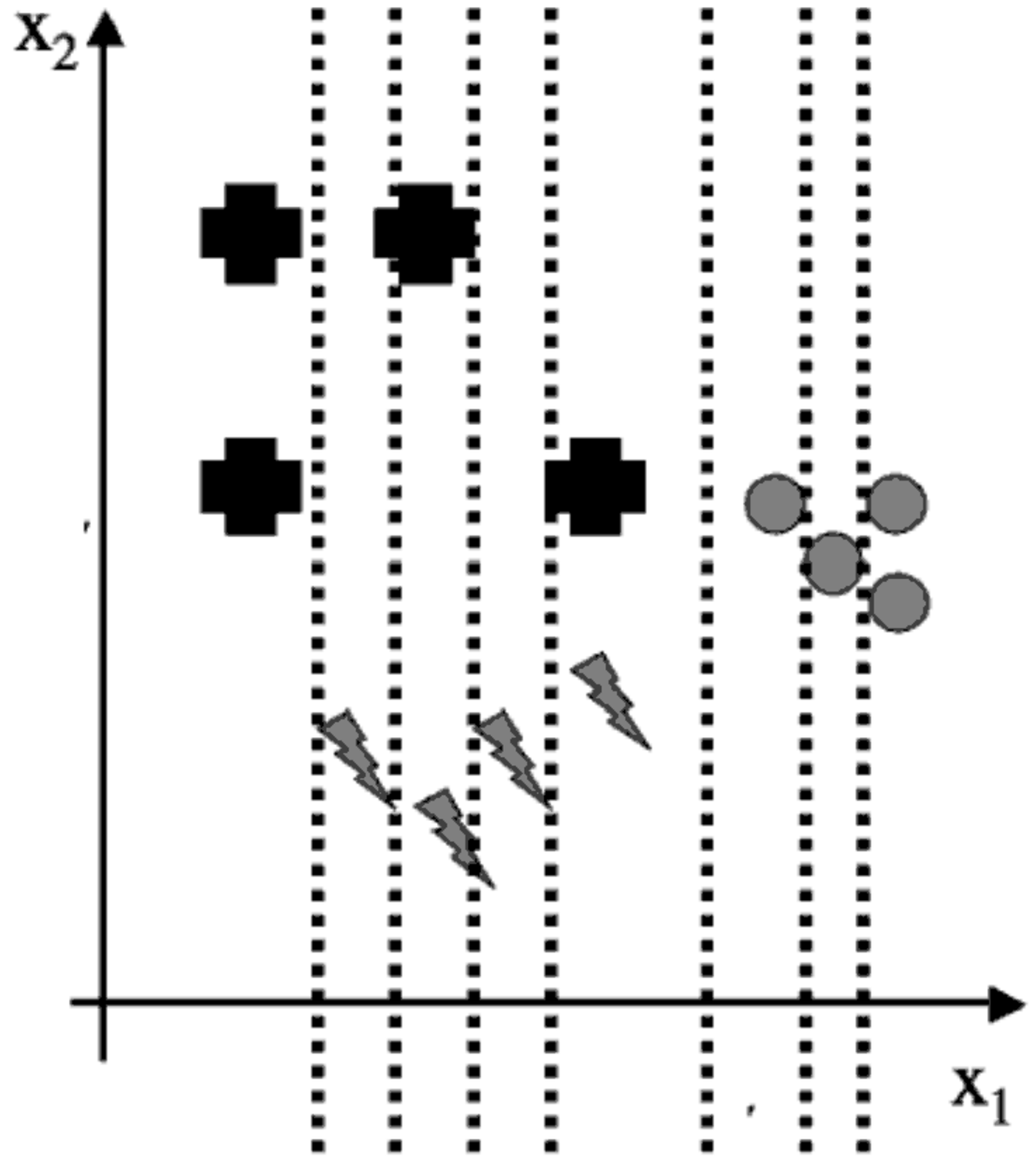


FIGURE 6.3: Possible places to split the variable x_1, between each of the datapoints as the feature value increases.

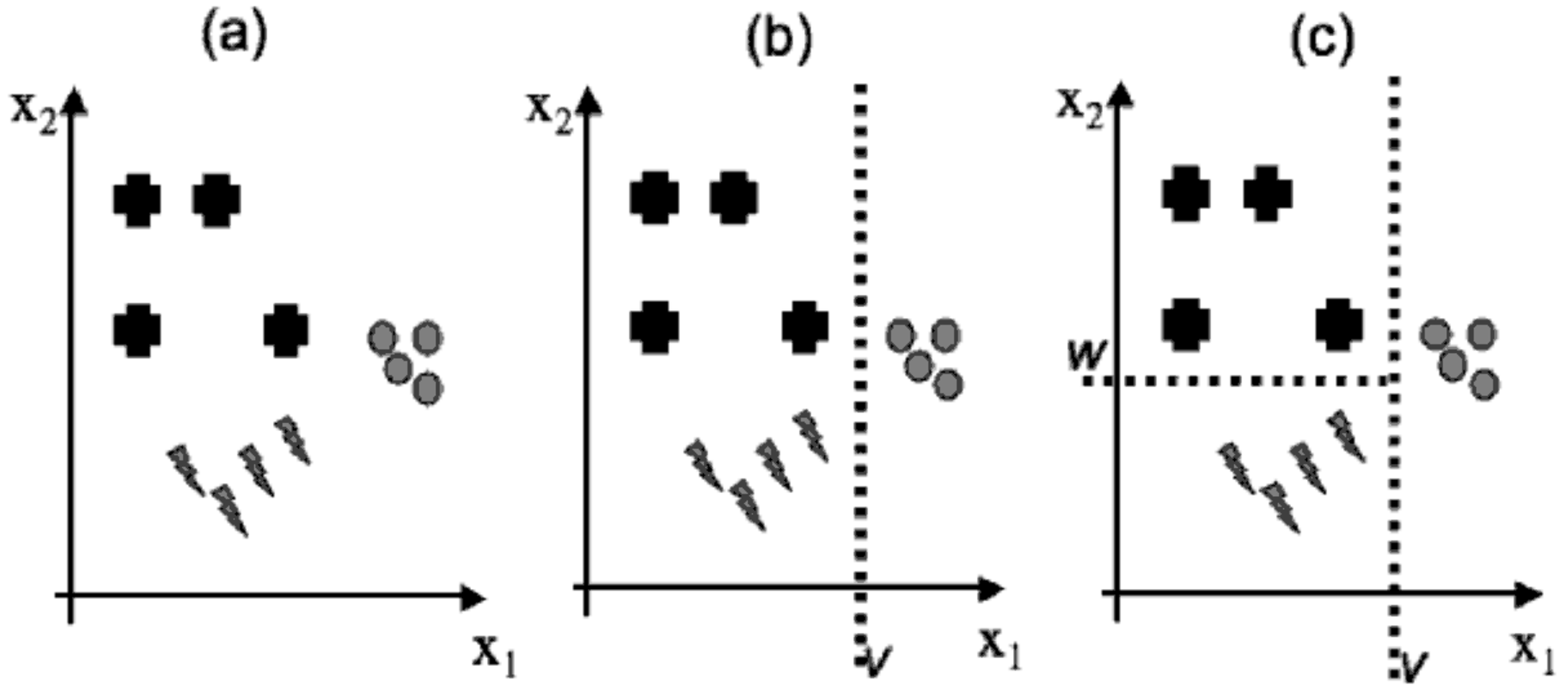


FIGURE 6.4: The effect of decision tree choices. The two-dimensional dataset shown in(a) is split first by choosing feature x_1 (b) and then x_2, (c) which separates out the three classes. The final tree is shown in Figure 6.5.

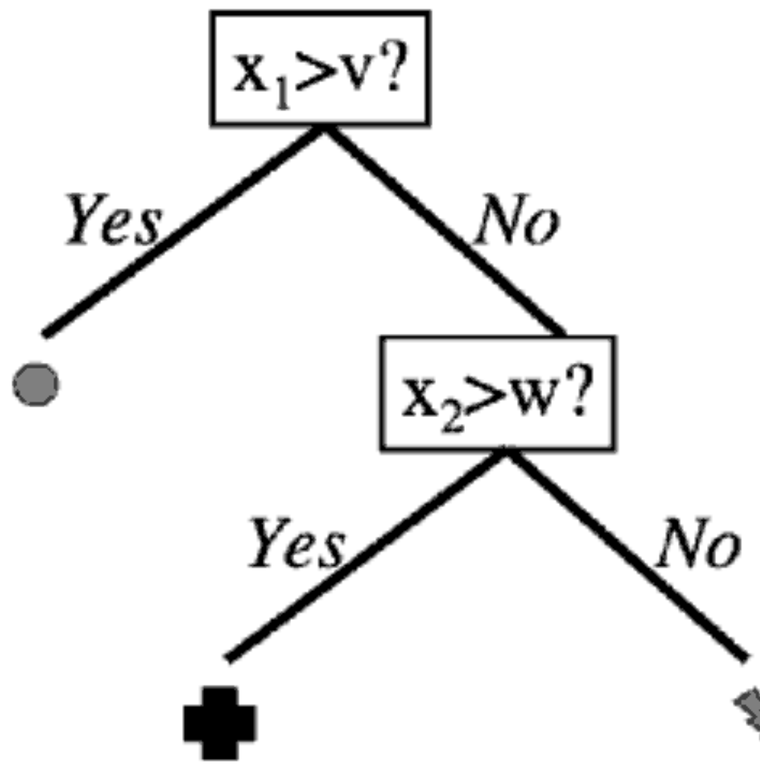


FIGURE 6.5: The final tree created by the splits in Figure 6.4.

there is no guarantee of this. Nor are they necessarily binary, especially for ID3 and C4.5, as our example shows.

If we assume that the tree is approximately balanced, then the cost at each node consists of searching through the d possible features (although this decreases by 1 at each level, that doesn't affect the complexity in the $\mathcal{O}(\cdot)$ notation) and then computing the information gain for the dataset for each split. This has cost $\mathcal{O}(dn \log n)$, where n is the size of the dataset at that node. For the root, $n = N$, and if the tree is balanced then n is divided by 2 at each stage down the tree. Summing this over the approximately $\log N$ levels in the tree gives computational cost $\mathcal{O}(dN^2 \log N)$.

6.3 Classification and Regression Trees (CART)

There is another well-known tree-based algorithm, CART, whose name indicates that it can be used for both classification and regression. Classification is not wildly different in CART, although it is usually constrained to construct binary trees. This might seem odd at first, but there are sound computer science reasons why binary trees are good, as suggested in the computational cost discussion above, and it is not a real limation. Even in the example that we started the chapter with, we can always turn questions into binary decisions by splitting the question up a little. Thus, a question that has three answers (say the question about when your nearest assignment deadline is, which is either 'urgent', 'near', or 'none') can be split into two questions: first, 'is the deadline urgent?', and then if the answer to that is 'no', second 'is the deadline near?' The only real difference with classification in CART is that a different information measure is commonly used. This is discussed next, before we look briefly at regression with trees.

6.3.1 Gini Impurity

The entropy that was used in ID3 as the information measure is not the only way to pick features. Another possibility is something known as the Gini impurity. The 'impurity' in the name suggests that the aim of the decision tree is to have each leaf node represent a set of datapoints that are in the same class, so that there are no mismatches. This is known as purity. If a leaf is pure then all of the training data within it have just one class. In which case, if we count the number of datapoints at the node (or better, the fraction of the number of datapoints) that belong to a class i (call it $N(i)$), then it should be 0 for all except one value of i. So suppose that you want to decide on which feature to choose for a split. The algorithm loops over the different features and checks how many points belong to each class. If the node is pure, then $N(i) = 0$ for all values of i except one particular one. So for any particular feature k you can compute:

$$G_k = \sum_{i=1}^{c} \sum_{j \neq i} N(i)N(j), \tag{6.8}$$

where c is the number of classes. In fact, you can reduce the algorithmic effort required by noticing that $\sum_i N(i) = 1$ (since there has to be some output class) and so $\sum_{j \neq i} N(j) = 1 - N(i)$. Then Equation (6.8) is equivalent to:

$$G_k = 1 - \sum_{i=1}^{c} N(i)^2. \tag{6.9}$$

Either way, the Gini impurity is equivalent to computing the expected error rate if the classification was picked according to the class distribution. The information gain can then be measured in the same way, subtracting each value G_i from the total Gini impurity.

The information measure can be changed in another way, which is to add a weight to the misclassifications. The idea is to consider the cost of misclassifying an instance of class i as class j (which we will call the risk in Section 8.1.1) and add a weight that says how important each datapoint is. It is typically labelled as λ_{ij} and is presented as a matrix, with element λ_{ij} representing the cost of misclassifying i as j. Using it is simple, modifying the Gini impurity (Equation (6.8)) to be:

$$G_i = \sum_{j \neq i} \lambda_{ij} N(i)N(j). \tag{6.10}$$

We will see in Section 7.1 that there is another benefit to using these weights, which is to successively improve the classification ability by putting higher weight on datapoints that the algorithm is getting wrong.

6.3.2 Regression in Trees

The new part about CART is its application in regression. While it might seem strange to use trees for regression, it turns out to require only a simple modification to the algorithm. Suppose that the outputs are continuous, so that a regression model is appropriate. None of the node impurity measures that we have considered so far will work. Instead, we'll go back to our old favourite—the sum-of-squares error. To evaluate the choice of which feature to use next, we also need to find the value at which to split the dataset according to that feature. Remember that the output is a value at each leaf. In general, this is just a constant value for the output, computed as the mean average of all the data points that are situated in that leaf. This is the optimal choice in order to minimise the sum-of-squares error, but it also means that we can choose the split point quickly for a given feature, by choosing it to minimise the sum-of-squares error. We can then pick the feature that has the split point that provides the best sum-of-squares error, and continue to use the algorithm as for classification.

6.4 Classification Example

We'll work through an example using ID3 in this section. The data that we'll use will be a continuation of the one we started the chapter with, about what to do in the evening. When we want to construct the decision tree to decide what to do in the evening, we start by listing everything that we've done for the past few days to get a suitable dataset (here, the last ten 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

To produce a decision tree for this problem, the first thing that we need to do is work out which feature to use as the root node. We start by computing the entropy of S:

$$\begin{aligned}\text{Entropy}(S) &= -p_{\text{party}}\log_2 p_{\text{party}} - p_{\text{study}}\log_2 p_{\text{study}} \\ &\quad - p_{\text{pub}}\log_2 p_{\text{pub}} - p_{\text{TV}}\log_2 p_{\text{TV}} \\ &= -\frac{5}{10}\log_2\frac{5}{10} - \frac{3}{10}\log_2\frac{3}{10} - \frac{1}{10}\log_2\frac{1}{10} - \frac{1}{10}\log_2\frac{1}{10} \\ &= 0.5 + 0.5211 + 0.3322 + 0.3322 = 1.6855 \end{aligned} \tag{6.11}$$

and then find which feature has the maximal information gain:

$$\begin{aligned}\text{Gain}(S, \text{Deadline}) &= 1.6855 - \frac{|S_{\text{urgent}}|}{10}\text{Entropy}(S_{\text{urgent}}) \\ &\quad - \frac{|S_{\text{near}}|}{10}\text{Entropy}(S_{\text{near}}) - \frac{|S_{\text{none}}|}{10}\text{Entropy}(S_{\text{none}}) \\ &= 1.6855 - \frac{3}{10}\left(-\frac{2}{3}\log_2\frac{2}{3} - \frac{1}{3}\log_2\frac{1}{3}\right) \\ &\quad - \frac{4}{10}\left(-\frac{2}{4}\log_2\frac{2}{4} - \frac{1}{4}\log_2\frac{1}{4} - \frac{1}{4}\log_2\frac{1}{4}\right) \\ &\quad - \frac{3}{10}\left(-\frac{1}{3}\log_2\frac{1}{3} - \frac{2}{3}\log_2\frac{2}{3}\right) \\ &= 1.6855 - 0.2755 - 0.6 - 0.2755 \\ &= 0.5345 \end{aligned} \tag{6.12}$$

$$\begin{aligned}\text{Gain}(S, \text{Party}) &= 1.6855 - \frac{5}{10}\left(-\frac{5}{5}\log_2\frac{5}{5}\right) \\ &\quad - \frac{5}{10}\left(-\frac{3}{5}\log_2\frac{3}{5} - \frac{1}{5}\log_2\frac{1}{5} - \frac{1}{5}\log_2\frac{1}{5}\right) \\ &= 1.6855 - 0 - 0.6855 \\ &= 1.0 \end{aligned} \tag{6.13}$$

$$\begin{aligned}\text{Gain}(S, \text{Lazy}) &= 1.6855 - \frac{6}{10}\left(-\frac{3}{6}\log_2\frac{3}{6} - \frac{1}{6}\log_2\frac{1}{6} - \frac{1}{6}\log_2\frac{1}{6} - \frac{1}{6}\log_2\frac{1}{6}\right) \\ &\quad - \frac{4}{10}\left(-\frac{2}{4}\log_2\frac{2}{4} - \frac{2}{4}\log_2\frac{2}{4}\right) \\ &= 1.6855 - 1.0755 - 0.4 \\ &= 0.21 \end{aligned} \tag{6.14}$$

Therefore, the root node will be the party feature, which has two feature values ('yes' and 'no'), so it will have two branches coming out of it (see Figure 6.6). When we look at the 'yes' branch, we see that in all five cases where there was a party we went to it, so we just put a leaf node there, saying

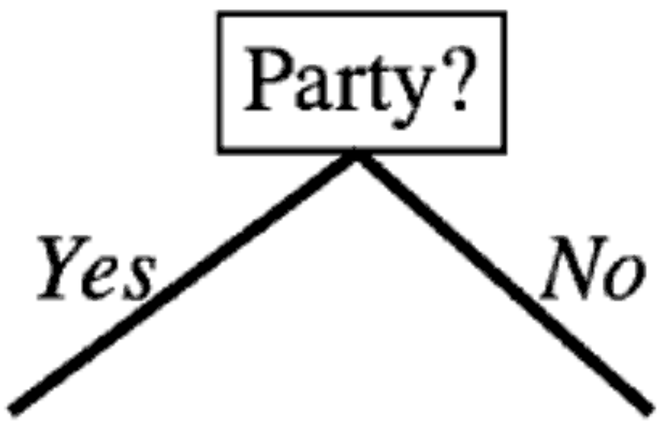


FIGURE 6.6: The decision tree after one step of the algorithm.

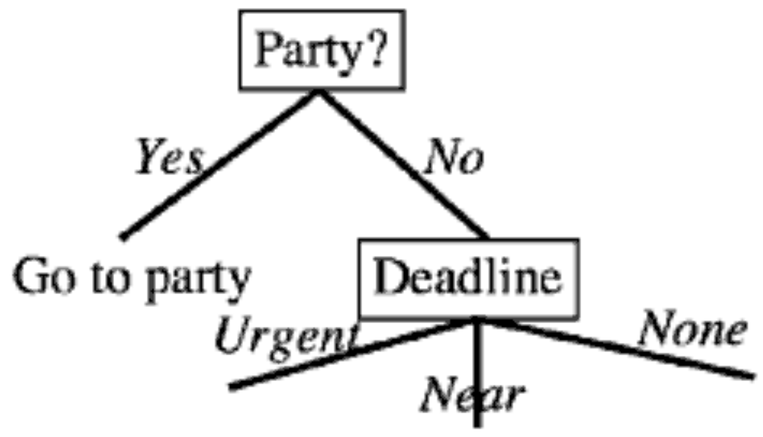


FIGURE 6.7: The tree after another step.

'party'. For the 'no' branch, out of the five cases there are three different outcomes, so now we need to choose another feature. The five cases we are looking at are:

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

We've used the party feature, so we just need to calculate the information gain of the other two over these five examples:

$$\begin{aligned}\text{Gain}(S, \text{Deadline}) &= 1.371 - \frac{2}{5}\left(-\frac{2}{2}\log_2\frac{2}{2}\right) \\ &\quad - \frac{2}{5}\left(-\frac{1}{2}\log_2\frac{1}{2} - \frac{1}{2}\log_2\frac{1}{2}\right) - \frac{1}{5}\left(-\frac{1}{1}\log_2\frac{1}{1}\right) \\ &= 1.371 - 0 - 0.4 - 0 \\ &= 0.971 \end{aligned} \tag{6.15}$$

$$\begin{aligned}\text{Gain}(S, \text{Lazy}) &= 1.371 - \frac{4}{5}\left(-\frac{2}{4}\log_2\frac{2}{4} - \frac{1}{4}\log_2\frac{1}{4} - \frac{1}{4}\log_2\frac{1}{4}\right) \\ &\quad - \frac{1}{5}\left(-\frac{1}{1}\log_2\frac{1}{1}\right) \\ &= 1.371 - 1.2 - 0 \\ &= 0.1710 \end{aligned} \tag{6.16}$$

This leads to the tree shown in Figure 6.7. From this point it is relatively simple to complete the tree, leading to the one that was shown in Figure 6.1.

Further Reading

For more information about decision trees, the following two books are of interest:

- J.R. Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kaufmann, San Francisco, CA, USA, 1993.
- L. Breiman, J.H. Friedman, R.A. Olshen, and C.J. Stone. *Classification and Regression Trees.* Chapman & Hall, New York, USA, 1993.

If you want to know more about information theory, then there are lots of books on the topic, including:

- T.M. Cover and J.A. Thomas. *Elements of Information Theory.* Wiley-Interscience, New York, USA, 1991.
- F.M. Reza. *An Introduction to Information Theory.* McGraw-Hill, New York, USA, 1961.

The original paper that started the field is:

- C.E. Shannon. A mathematical theory of information. *The Bell System Technical Journal*, 27(3):379–423 and 623–656, 1948.

A book that covers information theory and machine learning is:

- D.J.C. MacKay. *Information Thoery, Inference and Learning Algorithms.* Cambridge University Press, Cambridge, UK, 2003.

Other machine learning textbooks that cover decision trees include:

- Sections 8.2–8.4 of R.O. Duda, P.E. Hart, and D.G. Stork. *Pattern Classification.* Wiley-Interscience, New York, USA, 2nd edition, 2001.
- Chapter 7 of B.D. Ripley. *Pattern Recognition and Neural Networks.* Cambridge University Press, Cambridge, UK, 1996.
- Chapter 3 of T. Mitchell. *Machine Learning.* McGraw-Hill, New York, USA, 1997.

Practice Questions

Problem 6.1 Suppose that the probability of five events are P(first) = 0.5, and P(second) = P(third) = P(fourth) = P(fifth) = 0.125. Calculate the entropy. Write down in words what this means.

Problem 6.2 Make a decision tree that computes the logical AND function. How does it compare to the Perceptron solution?

Problem 6.3 Turn this politically incorrect data from Quinlan into a decision tree to classify which attributes make a person attractive, and then extract the rules.

Height	Hair	Eyes	Attractive?
Small	Blonde	Brown	No
Tall	Dark	Brown	No
Tall	Blonde	Blue	Yes
Tall	Dark	Blue	No
Small	Dark	Blue	No
Tall	Red	Blue	Yes
Tall	Blonde	Brown	No
Small	Blonde	Blue	Yes

Problem 6.4 When you arrive at the pub, your five friends already have their drinks on the table. Jim has a job and buys the round half of the time. Jane buys the round a quarter of the time, and Sarah and Simon buy a round one eighth of the time. John hasn't got his wallet out since you met him three years ago.

Compute the entropy of each of them buying the round and work out how many questions you need to ask (on average) to find out who bought the round.

Two more friends now arrive and everybody spontaneously decides that it is your turn to buy a round (for all eight of you). Your friends set you the challenge of deciding who is drinking beer and who is drinking vodka according to their gender, whether or not they are students, and whether they went to the pub last night. Use ID3 to work it out, and then see if you can prune the tree.

Drink	Gender	Student	Pub last night
Beer	T	T	T
Beer	T	F	T
Vodka	T	F	F
Vodka	T	F	F
Vodka	F	T	T
Vodka	F	F	F
Vodka	F	T	T
Vodka	F	T	T

Problem 6.5 The CPU dataset in the UCI repository is a very good regression problem for a decision tree. You will need to modify the decision tree code so that it does regression, as discussed in Section 6.3.2. You will also have to work out the Gini impurity for multiple classes.

Problem 6.6 Modify the implementation to deal with continuous variables, as discussed in Section 6.2.5.

Problem 6.7 The misclassification impurity is:

$$N(i) = 1 - \max_j P(w_j). \tag{6.17}$$

Add this into the code and test the new version on some of the datasets above.

Chapter 7

Decision by Committee: Ensemble Learning

The old saying has it that two heads are better than one. Which naturally leads to the idea that even more heads are better than that, and ends up with decision by committee, which is famously useless for human activities (as in the old joke that a camel is a horse designed by a committee). For machine learning methods the results are rather more impressive, as we'll see in this chapter.

The basic idea is that by having lots of learners that each get slightly different results on a dataset—some learning certain things well and some learning others—and putting them together, the results that are generated will be significantly better than any one of them on its own (provided that you put them together well... otherwise the results could be significantly worse). One analogy that might prove useful is to think about how your doctor goes about performing a diagnosis of some complaint that you visit her with. If she cannot find the problem directly, then she will ask for a variety of tests to be performed, e.g., scans, blood tests, consultations with experts. She will then aggregate all of these opinions in order to perform a diagnosis. Each of the individual tests will suggest a diagnosis, but only by putting them together can an informed decision be reached.

Figure 7.1 shows the basic idea of ensemble learning, as these methods are collectively called. Given a relatively simple binary classification problem and some learner that puts an ellipse around a subset of the data, combining the ellipses can provide a considerably more complex decision boundary.

There are then only a couple of questions to ask: which learners should we use, how should we ensure that they learn different things, and how should we combine their results? The methods that we are investigating in this chapter can use any classifier at all. Although in general they only use one type of classifier at a time, they do not have to. A common choice of classifier is the decision tree (see Chapter 6).

Ensuring that the learners see different things can be performed in different ways, and it is the primary difference between the algorithms that we shall see. However, it can also come about naturally depending upon the application area. Suppose that you have lots and lots of data. In that case you could simply randomly partition the data and give different sets of data to different classifiers. Even here there are choices: do you make the partitions separate,

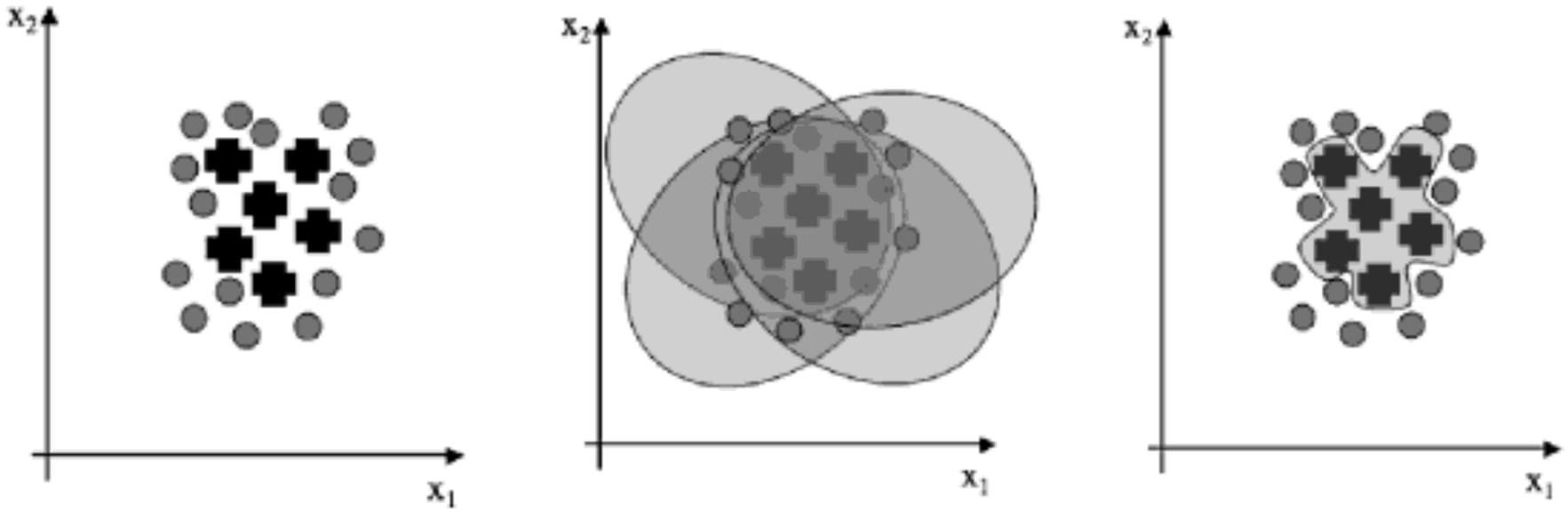


FIGURE 7.1: By combining lots of simple classifiers (here that simply put an elliptical decision boundary onto the data), the decision boundary can be made much more complicated, enabling the difficult separation of the pluses from the circles.

or include overlaps? If there is no overlap, then it could be difficult to work out how to combine the classifiers, or it might be very simple: if your doctor always asks for opinions from two colleagues, one specialising in heart problems and one in sports injuries, then upon discovering that your leg started hurting after you went for a run she would likely accord more weight to the diagnosis of the sports injury expert.

Interestingly, ensemble methods do very well when there is very little data as well as when there is too much. To see why, think cross-validation (Section 3.3.5). We used cross-validation when there was not enough data to go around, and trained lots of neural networks on different subsets of the data. Then we threw away most of them. With an ensemble method we keep them all, and combine their results in some way. One very simple way to combine the results is to use majority voting — if it's good enough for electing governments in elections, it's good enough for machine learning. Majority voting has the interesting property that for binary classification, the combined classifier will only get the answer wrong if more than half of the classifiers were wrong. Hopefully, this isn't going to happen too often (although you might be able to think of government elections where this has been the case in your view). There are alternative ways to combine the results, as we'll discuss. These things will become clearer as we look at the algorithms, so let's get started.

7.1 Boosting

At first sight the claim of the most popular ensemble method, boosting, seems amazing. If we take a collection of very poor (weak in the jargon) learners, each performing only just better than chance, then by putting them together it is possible to make an ensemble learner that can perform arbitrarily

well. So we just need lots of low quality learners, and a way to put them together usefully, and we can make a learner that will do very well.

The principal algorithm of boosting is named AdaBoost, and is described in Section 7.1.1. The algorithm was first described in the mid-1990s by Freund and Shapiro, and while it has had many variations derived from it, the principal algorithm is still one of the most widely used. The algorithm was proposed as an improvement on the original 1990 boosting algorithm, which was rather data hungry. In that algorithm, the training set was split into three. A classifier was trained on the first third, and then tested on the second third. All of the data that was misclassified during that testing was used to form a new dataset, along with an equally sized random selection of the data that was correctly classified. A second classifier was trained on this new dataset, and then both of the classifiers were tested on the final third of the dataset. If they both produced the same output, then that datapoint was ignored, otherwise the datapoint was added to yet another new dataset, which formed the training set for a third classifer. Rather than looking further at this version, we will look at the more common algorithm.

7.1.1 AdaBoost

The innovation that AdaBoost (which stands for adaptive boosting) uses is to give weights to each datapoint according to how difficult previous classifiers have found to get it correct. These weights are given to the classifier as part of the input when it is trained.

The AdaBoost algorithm is conceptually very simple. At each iteration a new classifier is trained on the training set, with the weights that are applied to the training set for each datapoint being modified at each iteration according to how successfully that datapoint has been classified in the past. The weights are initially all set to the same value, $1/N$, where N is the number of datapoints in the training set. Then, at each iteration, the error (ϵ) is computed as the sum of the weights of the misclassified points, and the weights for incorrect examples are updated by being multiplied by $\alpha = \epsilon/(1-\epsilon)$. Weights for correct examples are left alone, and then the whole set is normalised so that it sums to 1 (which is effectively a reduction in the importance of the correctly classified datapoints). Training terminates after a set number of iterations, or when either all of the datapoints are classified correctly, or one point contains more than half of the available weight.

Figure 7.2 shows the effect of weighting incorrectly classified examples as training proceeds, with the size of each datapoint being a measure of its importance.

As an algorithm this looks like (where $I(y_n \neq h_t(x_n))$ is an indicator function that returns 1 if the target and output are not equal, and 0 if they are):

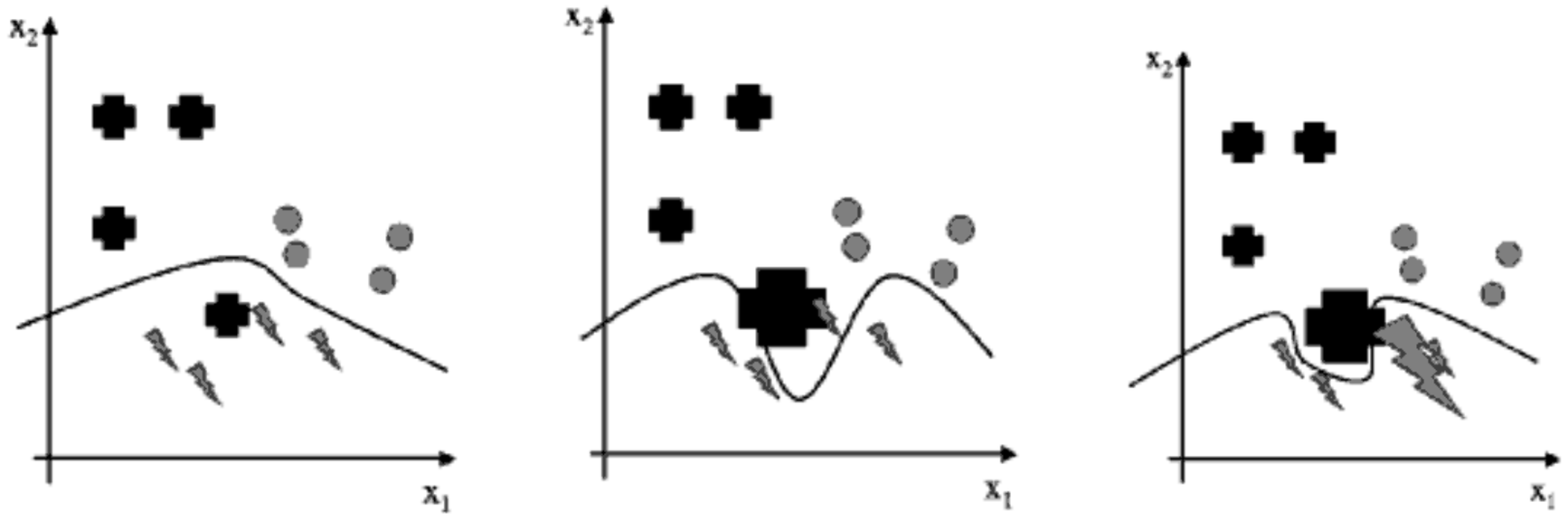


FIGURE 7.2: As points are misclassified, so their weights increase in boosting (shown by the datapoint getting larger), which makes the importance of those datapoints increase, making the classifiers pay more attention to them.

AdaBoost Algorithm

- initialise all weights to $1/N$, where N is the number of datapoints
- while $0 < \epsilon_t < \frac{1}{2}$ (and $t < T$, some maximum number of iterations):
  - train classifier on $\{S, w^{(t)}\}$, getting hypotheses $h_t(x_n)$ for datapoints x_n
  - compute training error $\epsilon_t = \sum_{n=1}^{N} w_n^{(t)} I(y_n \neq h_t(x_n))$
  - set $\alpha_t = \log\left(\frac{1-\epsilon_t}{\epsilon_t}\right)$
  - update weights using:

$$w_n^{(t+1)} = w_n^{(t)} \exp(\alpha_t I(y_n \neq h_t(x_n))/Z_t, \tag{7.1}$$

    where Z_t is a normalisation constant
- output $f(x) = \text{sign}\left(\sum_{t=1}^{T} \alpha_t h_t(x)\right)$

There is nothing too difficult to the implementation, either, as can be seen from the main loop here:

```
for t in range(T):
    classifiers[:,t] = train(data,classes,w[:,t])
    outputs,errors = classify(data,classifiers[0,t],
    classifiers[1,t])

    index[:,t] = errors
```

```
print "index: ", index[:,t]
e[t] = sum(w[:,t]*index[:,t])/sum(w[:,t])

if t>0 and (e[t]==0 or e[t]>=0.5):
    T=t
    alpha = alpha[:t]
    index = index[:,:t]
    w = w[:,:t]
    break

alpha[t] = log((1-e[t])/e[t])
w[:,t+1] = w[:,t]* exp(alpha[t]*index[:,t])
w[:,t+1] = w[:,t+1]/sum(w[:,t+1])
```

Most of the work of the algorithm is done by the classification algorithm, which is given new weights at each iteration. In this respect, boosting is not quite a stand-alone algorithm: the classifiers need to consider the weights when they perform their classifications. It is not always obvious how to do this for a particular classifier, but we have seen methods of doing it for a few classifiers. For the decision tree we saw a method in Section 6.3.1, when we looked at the Gini impurity. There, we allowed for a $\boldsymbol{\lambda}$ matrix that encoded the risks associated with misclassification, and these are a perfect place in which to introduce weights. Modification of the decision tree algorithm to deal with these weights is suggested as an exercise for this chapter. A similar argument can be used for the Bayes' classifier; this will be discussed in Section 8.1.1.

As a very simple example showing how boosting works, a very simple classifier was created that can only separate data by fitting one either horizontal or vertical line, with it choosing which to fit at the current iteration at random. A two-dimensional dataset was created with data in the top right-hand corner being in one class, and the rest in another, plus a couple of the datapoints were randomly mislabelled to simulate noise. Clearly, this dataset cannot be separated by a single horizontal or vertical decision boundary. However, Figure 7.3 shows the output of the classifier on an independent test set, where the algorithm gets only one datapoint wrong, and that is one that is coincidentally close to one of the 'noisy' datapoints in the training data. Figure 7.4 shows the training data, the error curve on both the training and testing sets and the first few iterations of the classifier, which can only put in one horizontal or linear classification line.

Clearly, such impressive results require some explanation and understanding. The key to this understanding is to compute the loss function, which is simply the measure of the error that is applied (we have been using a sum-of-squares loss function for many algorithms in the book). The loss function for AdaBoost has the form

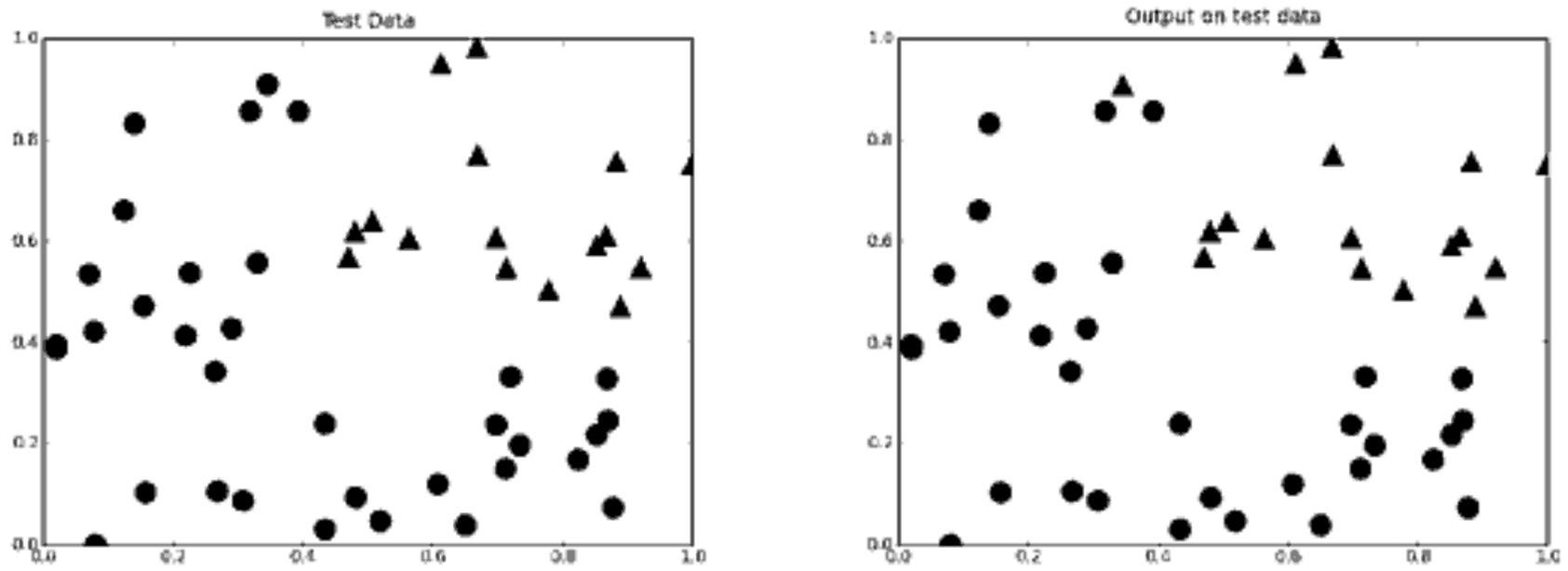


FIGURE 7.3: Boosting learns this simple dataset very successfully, producing an ensemble classifier that is rather more complicated than the simple horizontal or vertical line classifier that the algorithm boosts. On the independent test set shown here, the algorithm gets only 1 datapoint wrong, and that is one that is coincidentally close to one that was misclassified to simulate noise in the training data.

$$G_t(\alpha) = \sum_{n=1}^{N} \exp\left(-y_n(\alpha h_t(x_n) + f_{t-1}(x_n))\right), \tag{7.2}$$

where $f_{t-1}(x_n)$ is the sum of the hypotheses of that datapoint from the previous iterations:

$$f_{t-1}(x_n) = \sum_{\tau=0}^{t-1} \alpha_\tau h_\tau(x_n). \tag{7.3}$$

Exponential loss functions are well behaved and robust to outliers. The weights $w^{(t)}$ in the algorithm are nothing more than the second term in Equation (7.2), which can therefore be rewritten as:

$$G_t(\alpha) = \sum_{n=1}^{N} w^{(t)} \exp\left(-y_n \alpha h_t(x_n)\right). \tag{7.4}$$

Deriving the rest of the algorithm from here requires substituting in for the hypotheses h and then solving for α, which produces the full algorithm. Interestingly, this is not the way that AdaBoost was created; this understanding of why it works so well came later. It is possible to choose other loss functions, and providing that they are differentiable they will provide useful boosting-like algorithms, which are collectively known as arcing algorithms (for adaptive reweighting and combining).

Adaboost can be modified to perform regression rather than classification (known as real adaboost, or sometime adaboost.R). There is another variant on boosting (also called AdaBoost, confusingly) that uses the weights to sample from the full dataset, training on a sample of the data rather than the full

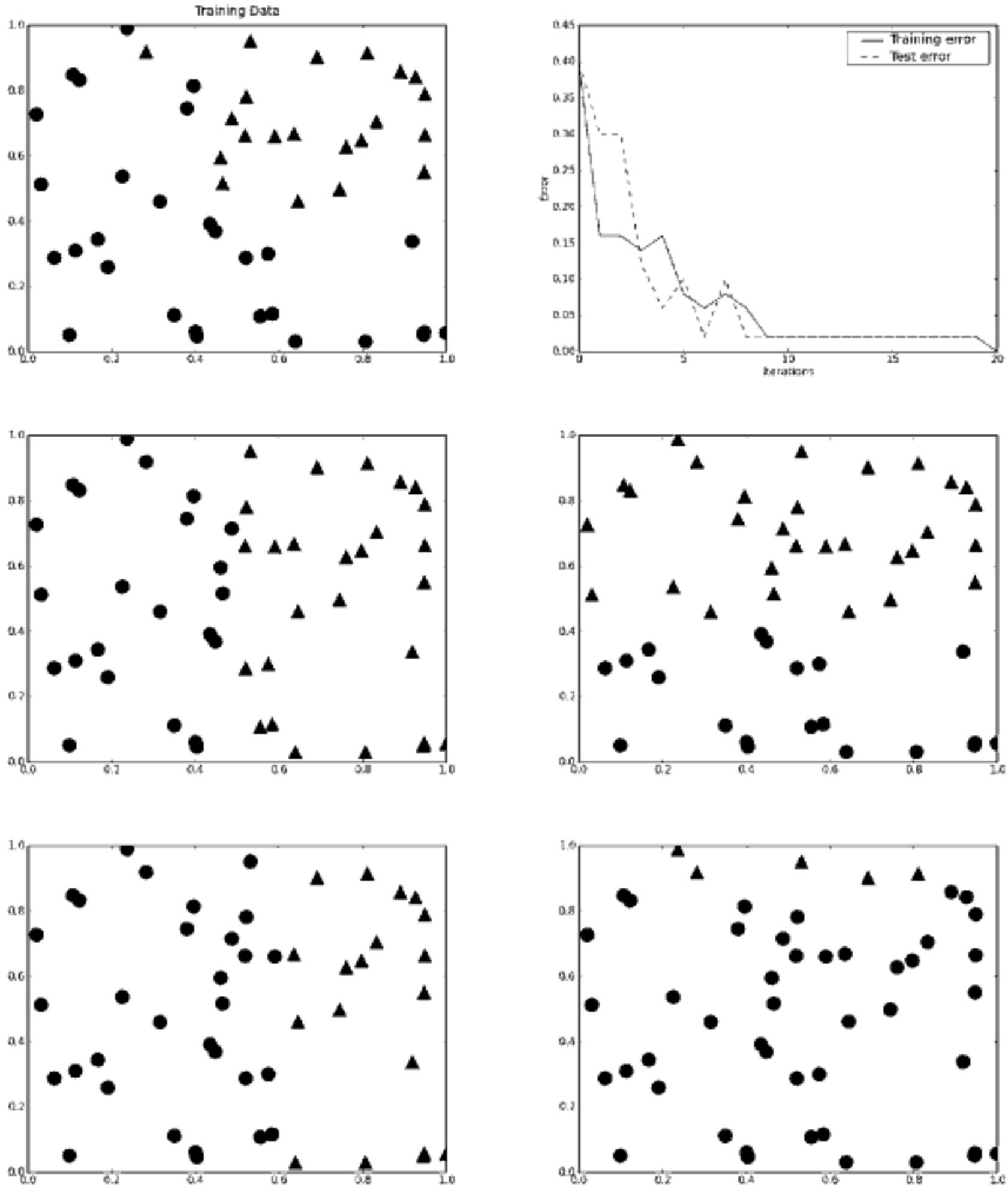


FIGURE 7.4: *Top:* the training data and the error curve. *Middle and bottom:* The first few iterations of the classifier; each plot shows the output of one of the weak classifiers that are boosted by the algorithm.

weighted set, with more difficult examples more likely to be in the training sample. This is more in line with the original boosting algorithm, and is obviously faster, since each training run has fewer data to learn about.

7.1.2 Stumping

There is a very extreme form of boosting that is applied to trees. It goes by the descriptive name of **stumping**. The stump of a tree is the tiny piece that is left over when you chop off the rest, and the same is true here: stumping consists of simply taking the root of the tree and using that as the decision maker. So for each classifier you use the very first question that makes up the root of the tree, and that is it. Often, this is worse than chance on the whole dataset, but by using the weights to sort out when that classifier should be used, and to what extent, as opposed to the other ones, the overall output of stumping can be very successful. In fact, it is pretty much exactly what the simple example that we saw consisted of.

7.2 Bagging

The simplest method of combining classifiers is known as **bagging**, which stands for **bootstrap aggregating**, the statistical description of the method. This is fine if you know what a **bootstrap** is, but fairly useless if you don't. A bootstrap sample is a sample taken from the original dataset **with replacement**, so that we may get some data several times and others not at all. The bootstrap sample is the same size as the original, and lots and lots of these samples are taken: B of them, where B is at least 50, and could even be in the thousands. The name bootstrap is more popular in computer science than anywhere else, since there is also a bootstrap loader, which is the first program to run when a computer is turned on. It comes from the nonsensical idea of 'picking yourself up by your bootstraps,' which means lifting yourself up by your shoelaces, and is meant to imply starting from nothing.

Bootstrap sampling seems like a very strange thing to do. We've taken a perfectly good dataset, mucked it up by sampling from it, which might be good if we had made a smaller dataset (since it would be faster), but we still ended up with a dataset the same size. Worse, we've done it lots of times. Surely this is just a way to burn up computer time without gaining anything. The benefit of it is that we will get lots of learners that perform slightly differently, which is exactly what we want for an ensemble method. Another benefit is that estimates of the accuracy of the classification function can be made without complicated analytic work, by throwing computer resources at the problem (technically, bagging is a **variance reducing algorithm**; the meaning of this will

become clearer when we talk about bias and variance in Section 8.2.4). This is a standard technique in modern statistics; we'll see another example in Chapter 14 when we look at Markov Chain Monte Carlo methods. It is sufficiently common to have inspired the comment that "statistics is defined as the discipline where those that think don't count and those that count don't think."

Having taken a set of bootstrap samples, the bagging method simply requires that we fit a model to each dataset, and then combine them by taking the output to be the majority vote of all the classifiers. A NumPy implementation is shown next, and then we will look at a simple example.

```
# Compute boostrap samples
samplePoints = random.randint(0,nPoints,(nPoints,nSamples))
classifiers = []

for i in range(nSamples):
 sample = []
 sampleTarget = []
 for j in range(nPoints):
  sample.append(data[samplePoints[j,i]])
  sampleTarget.append(targets[samplePoints[j,i]])
 # Train classifiers
 classifiers.append(self.tree.make_tree(sample,sampleTarget,
 features))
```

The example consists of taking the party data that was used in Section 6.4 to demonstrate the decision tree, and restricting the trees to stumps, so that they can make a classification based on just one variable. The output of a decision tree that uses the whole dataset for this is not surprising: it takes the two largest classes, and separates them. However, using just stumps of trees and 20 samples, bagging can separate the data perfectly, as this output shows:

```
Tree Stump Prediction
['Party', 'Party', 'Party', 'Party', 'Pub', 'Party', 'Study',
'Study', 'Party', 'Study']
Correct Classes
['Party', 'Study', 'Party', 'Party', 'Pub', 'Party', 'Study',
'TV', 'Party', 'Study']
Bagged Results
['Party', 'Study', 'Party', 'Party', 'Pub', 'Party', 'Study',
'TV', 'Party', 'Study']
```

7.2.1 Subagging

For some reason, ensemble methods often have good names, such as boosting and bagging (and we will see my choice for best-named, bragging, in Section 7.3). However, the method of subagging wins the prize for the oddest sounding word. It is a combination of 'subsample' and 'bagging,' and it is the fairly obvious idea that you don't need to produce samples that are the same size as the original data. If you make smaller datasets, then it makes sense to sample without replacement, but otherwise the implementation is only very slightly different from the bagging one, except that in NumPy you use `shuffle()` to produce the samples. It is common to use a dataset size that is half that of the original data, and the results of this can often be comparable to a full bagging simulation.

7.3 Different Ways to Combine Classifiers

Bagging puts most of its effort into ensuring that the different classifiers see different data, since they see different samples of the data. This is different than boosting, where the data stays the same, but the importance of each datapoint changes for the different classifiers, since they each get different weights according to how well the previous classifiers have performed. Just as important for an ensemble method, though, is how it combines the outputs of the different classifiers. Both boosting and bagging take a vote from amongst the classifiers, although they do it in different ways: boosting takes a weighted vote, while bagging simple takes the majority vote. There are other alternatives to these methods, as well.

In fact, even majority voting is not necessarily simple. Some classification systems will only produce an output where all the classifiers agree, or more than half of them agree, whereas others simply take the most common output, which is what we usually mean by majority voting. The idea of not always producing an output is to ensure that the ensemble does not produce outputs that are contentious, because they are probably difficult datapoints. If the number of classifiers is odd and the classifiers are each independent of each other, then majority voting will return the correct label if more than half of the classifiers agree. Assuming that each individual classifier has a success rate of p, the probability of the ensemble getting the correct answer is a binomial distribution of the form:

$$\sum_{k=T/2+1}^{T} \binom{T}{k} p^k (1-p)^{T-k}, \tag{7.5}$$

where T is the number of classifiers. If $p > 0.5$ then this sum approaches 1

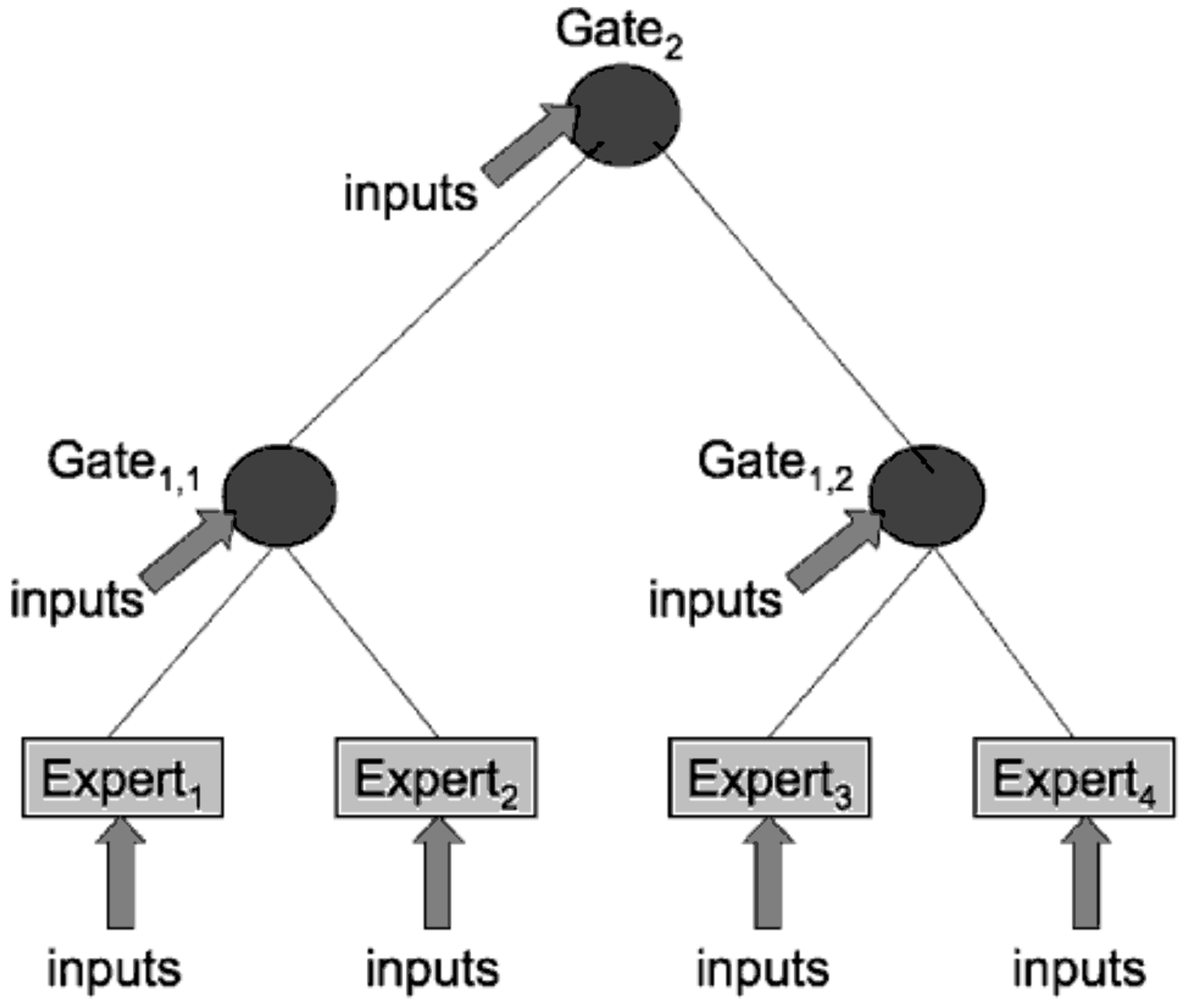


FIGURE 7.5: The Hierarchical Mixture of Networks network, consisting of a set of classifiers (experts) with gating systems that also use the inputs to decide which classifiers to trust.

as $T \to \infty$. This is a lot of the power behind ensemble methods: even if each classifier only gets about half the answers right, if we use a decent number of classifiers (maybe 100) then the probability of the ensemble being correct gets close to 1. In fact, even with less than 50% chance of success for each individual classifier, the ensemble can often do very well indeed.

For regression problems, rather than taking the majority vote, it is common to take the mean of the outputs. However, the mean is heavily affected by outliers, with the result that the median is a more common average to use. It is the use of the median that produces the **bragging** algorithm, which is meant to imply 'robust bagging.'

There is one more thing that can be done to combine classifiers, and that is to learn how to do it. There is an algorithm that does precisely this, known as the **mixture of experts**.

Inputs are presented to the network, and each individual classifier makes an assessment. These outputs from the classifiers are then weighted by the relevant **gate**, which produces a weight w using the current inputs, and this is propagated further up the hierarchy. The most common version of the mixture of experts works as follows:

The Mixture of Experts Algorithm

- for each expert:
  - calculate the probability of the input belonging to each possible class by computing (where the $\mathbf{w}_i$ are the weights for that classifier):

$$o_i(\mathbf{x}, \mathbf{w}_i) = \frac{1}{1 + \exp(-\mathbf{w}_i \cdot \mathbf{x})}. \tag{7.6}$$

- for each gating network up the tree:
  - compute:

$$g_i(\mathbf{x}, \mathbf{v}_i) = \frac{\exp(\mathbf{v}_i \mathbf{x})}{\sum_l \exp(\mathbf{v}_l \mathbf{x})}. \tag{7.7}$$

- pass as input to the next level gates (where the sum is over the relevant inputs to that gate):

$$\sum_k o_j g_j. \tag{7.8}$$

The most common way to train this network is using an EM algorithm. This is a general statistical approximation algorithm that will be discussed in Section 8.3.1. It is also possible to use gradient descent on the parameters.

There are a couple of other ways to view these mixture of experts methods. One is to regard them as trees, except that the splits are not the hard splits that we performed in Chapter 6, but rather soft, because they are based on probability. The other is to compare them with radial basis function (RBF) networks (see Section 4.2). Each RBF gave a constant output within its receptive field. If, instead, each node were to give a linear approximation to the data, then the result would be the mixture of experts network.

Further Reading

Three papers that cover the three main ensemble methods described in this section are:

- R.E. Schapire. The boosting approach to machine learning: An overview. In D. D. Denison, M. H. Hansen, C. Holmes, B. Mallick, and B. Yu, editors, *Nonlinear Estimation and Classification*, Springer, Berlin, Germany, 2003.

- L. Breiman. Bagging predictors. *Machine Learning*, 26(2):123–140, 1996.

- M.I. Jordan and R.A. Jacobs. Hierarchical mixtures of experts and the EM algorithm. *Neural Computation*, 6(2):181–214, 1994.

An overview of the whole area is provided by:

- L. Kuncheva. *Combining Pattern Classifiers: Methods and Algorithms.* Wiley-Interscience, New York, USA, 2004.

For an alternative viewpoint, see:

- Sections 15.3–15.6 of E. Alpaydin. *Introduction to Machine Learning.* MIT Press, Cambridge, MA, USA, 2004.

- Section 9.5 of R.O. Duda, P.E. Hart, and D.G. Stork. *Pattern Classification.* Wiley-Interscience, New York, USA, 2nd edition, 2001.

Practice Questions

Problem 7.1 Modify the decision tree implementation to use weights in the computation of the Gini impurity. This is not trivial, since you have to modify the total value of the Gini impurity, too. Once you have done it, use stump trees on the party data.

Problem 7.2 Implement the alternative form of boosting that uses the weights to sample the dataset. Does this make any difference to the outputs?

Problem 7.3 Stumping picks out the single most informative feature in the dataset and uses this. For a binary classification problem this will typically get at least half of the dataset correct. Why? How does this statement generalise to multiple classes?

Problem 7.4 Compare and contrast bagging and cross-validation.

Problem 7.5 The `Breastcancer` dataset in the UCI Machine Learning repository gives ten features and asks for a classification of breast tumours into benign and malignant. It is a difficult dataset, and provides a good comparison of the standard decision tree with boosted and bagged versions. Use all of the methods, using stumping and more advanced trees and see which work better.

Problem 7.6 The Mixture of Experts algorithm works with any kind of expert. Suppose that the experts were each MLPs. Implement this algorithm and see how well it does on the `Breastcancer` dataset above.

Chapter 8

Probability and 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 6. 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. The penalty that we pay for this is that there are going to be a whole lot of statistical ideas that we need to understand.

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.

8.1 Turning Data into Probabilities

Take a look at the plot in Figure 8.1. 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. 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.' 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

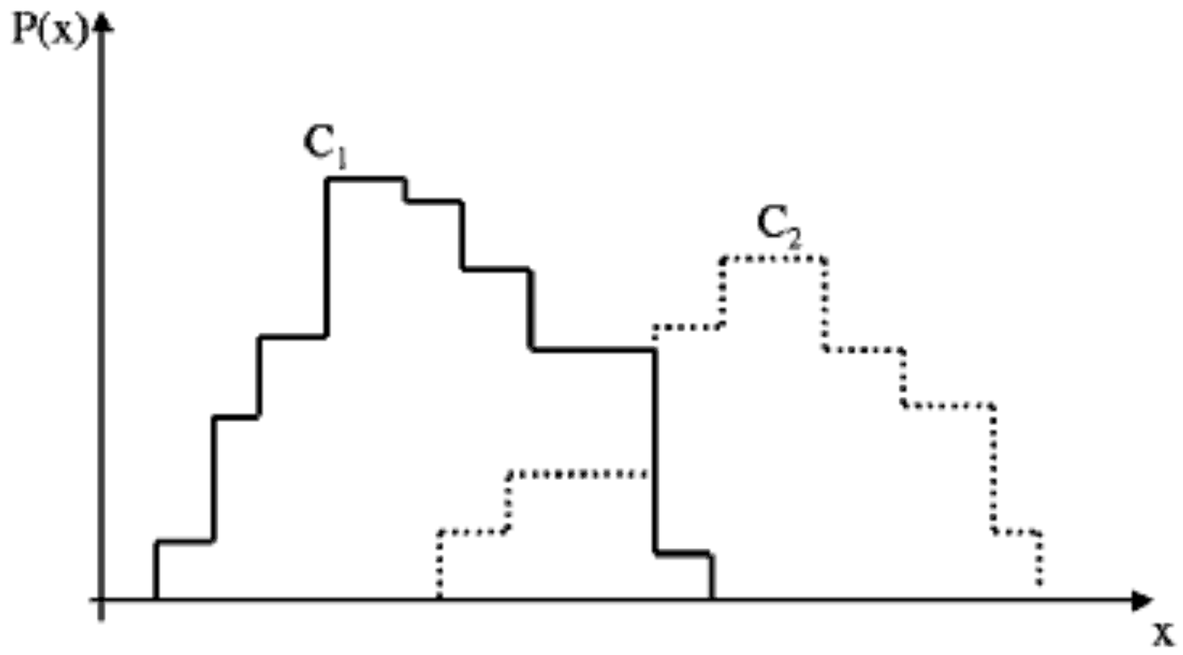


FIGURE 8.1: A histogram of feature values (x) against their probability for two classes.

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 8.1 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 8.1. 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 class C_i.

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

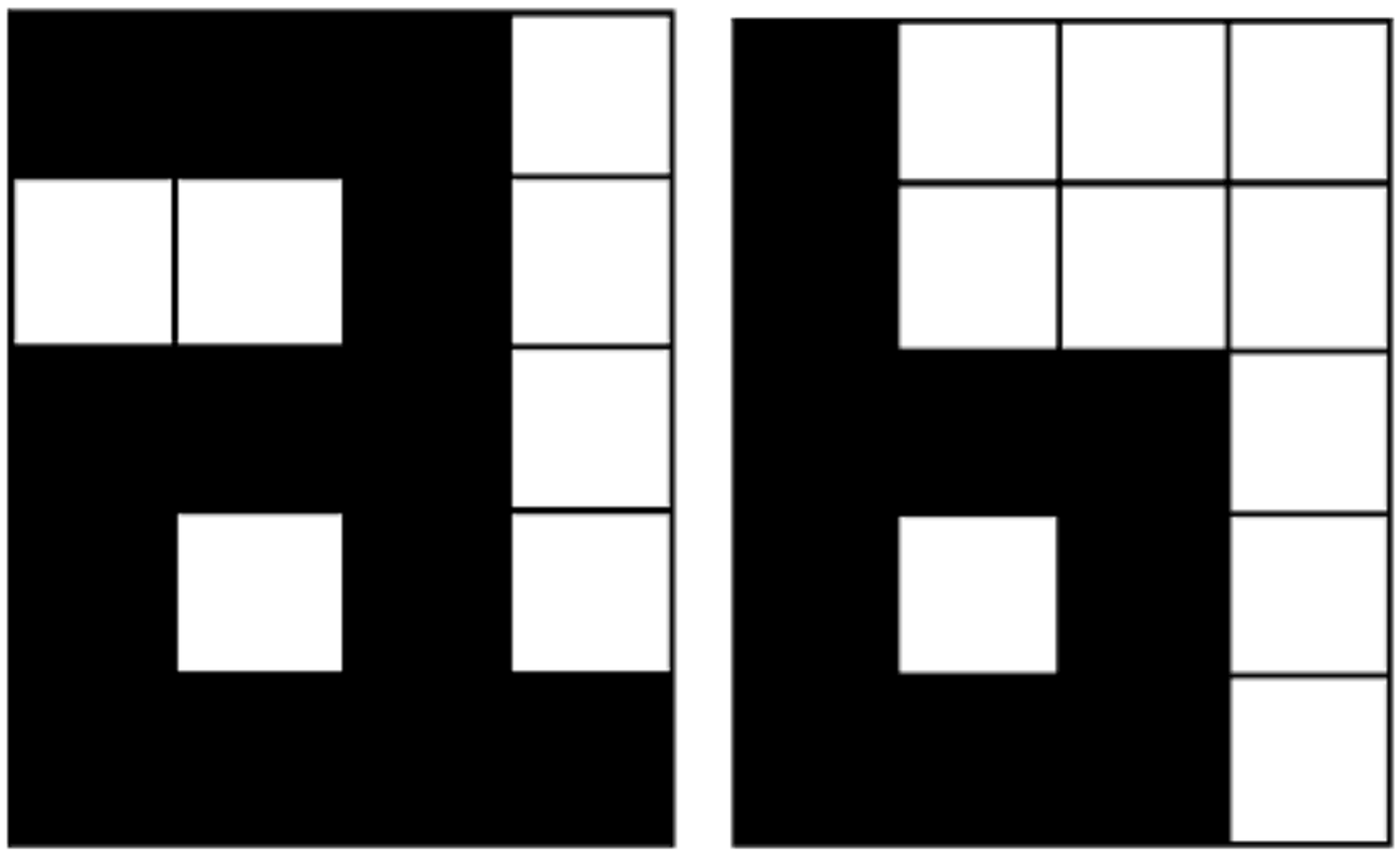


FIGURE 8.2: The letters 'a' and 'b' in pixel form.

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:

$$P(C_i, X_j) = P(X_j|C_i)P(C_i), \tag{8.1}$$

or equivalently:

$$P(C_i, X_j) = P(C_i|X_j)P(X_j). \tag{8.2}$$

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:

$$P(C_i|X_j) = \frac{P(X_j|C_i)P(C_i)}{P(X_j)}. \tag{8.3}$$

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:

$$P(X_k) = \sum_i P(X_k|C_i)P(C_i). \tag{8.4}$$

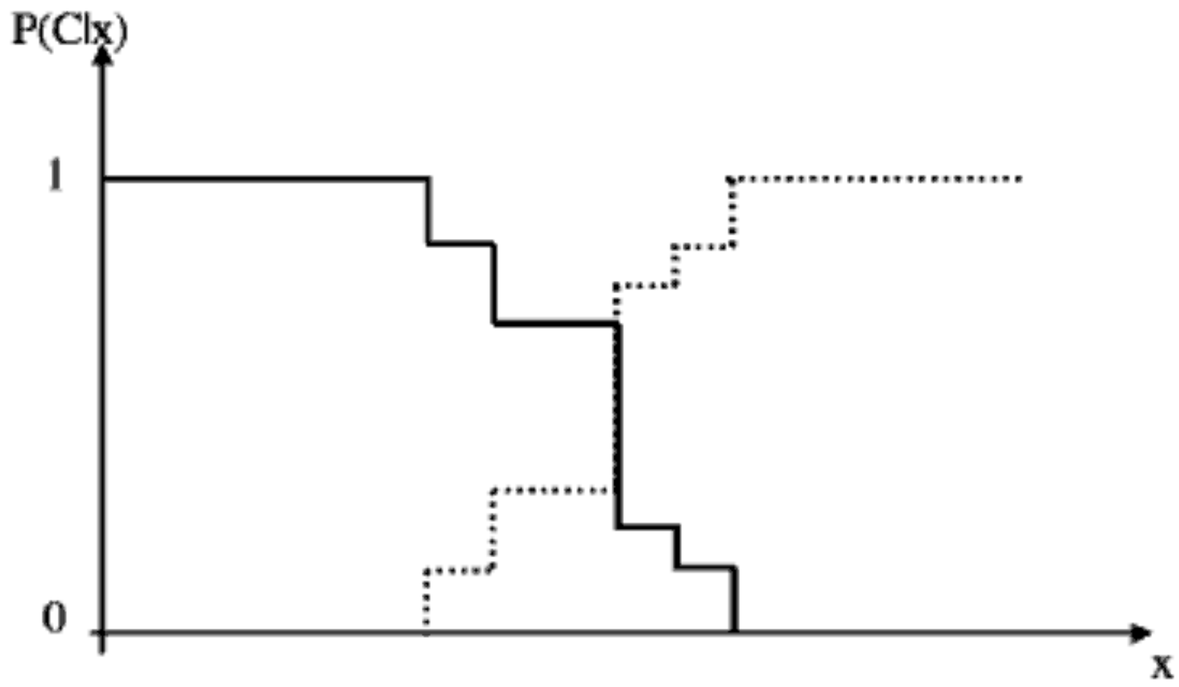


FIGURE 8.3: The posterior probabilities of the two classes C_1 and C_2 for feature x.

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 8.3) to assign each new observation to one of the classes by picking the class C_i where:

$$P(C_i|\mathbf{x}) > P(C_j|\mathbf{x}) \quad \forall \; i \neq j, \tag{8.5}$$

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. It is what we did in Section 3.4.2 for the MLP, choosing the class that the MLP gave the highest output activation to. 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.

8.1.1 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 when we looked at classification, for example in Section 2.2.6, 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.

8.1.2 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 (8.5). 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 4.3), 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:

$$P(X_j^1 = a_1|C_i) \times P(X_j^2 = a_2|C_i) \times \ldots \times P(X_j^n = a_n|C_i) = \prod_k P(X_j^k = a_k|C_i), \tag{8.6}$$

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:

$$P(C_i) \prod_k P(X_j^k = a_k|C_i). \tag{8.7}$$

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.

We will look again at the example that was used in Chapter 6, particularly Section 6.4, of deciding what to do in the evening. The way that we solve this problem using the naïve Bayes' classifier is different. We feed in the current values for our three feature variables and ask the classifier to compute the probabilities of each of the four possible classes 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 urgent, that there is no party on, and that you are currently poor. 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)

These evaluate to:

$$P(\text{Party}) = \frac{5}{10} \times \frac{2}{5} \times \frac{0}{5} \times \frac{2}{5} = 0 \tag{8.8}$$

$$P(\text{Study}) = \frac{3}{10} \times \frac{1}{3} \times \frac{3}{3} \times \frac{1}{3} = \frac{1}{30} \tag{8.9}$$

$$P(\text{Pub}) = \frac{1}{10} \times \frac{0}{1} \times \frac{1}{1} \times \frac{1}{1} = 0 \tag{8.10}$$

$$P(\text{TV}) = \frac{1}{10} \times \frac{1}{1} \times \frac{1}{1} \times \frac{0}{1} = 0 \tag{8.11}$$

So based on this you will be studying tonight. And quite right, too.

8.2 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.

8.2.1 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.

8.2.2 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

$$E = -1 \times \frac{199,999}{200,000} + 99,999 \times \frac{1}{200,000} = -0.5, \tag{8.12}$$

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, $\boldsymbol{\mu}$):

$$\text{var}(\{\mathbf{x}_i\}) = \sigma^2(\{\mathbf{x}_i\}) = E((\{\mathbf{x}_i\} - \boldsymbol{\mu})^2) = \sum_{i=1}^{N} (\mathbf{x}_i - \boldsymbol{\mu})^2. \tag{8.13}$$

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:

$$\text{cov}(\{\mathbf{x}_i\}, \{\mathbf{y}_i\}) = E((\{\mathbf{x}_i\} - \boldsymbol{\mu})^2) E((\{\mathbf{y}_i\} - \boldsymbol{\nu})^2), \tag{8.14}$$

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:

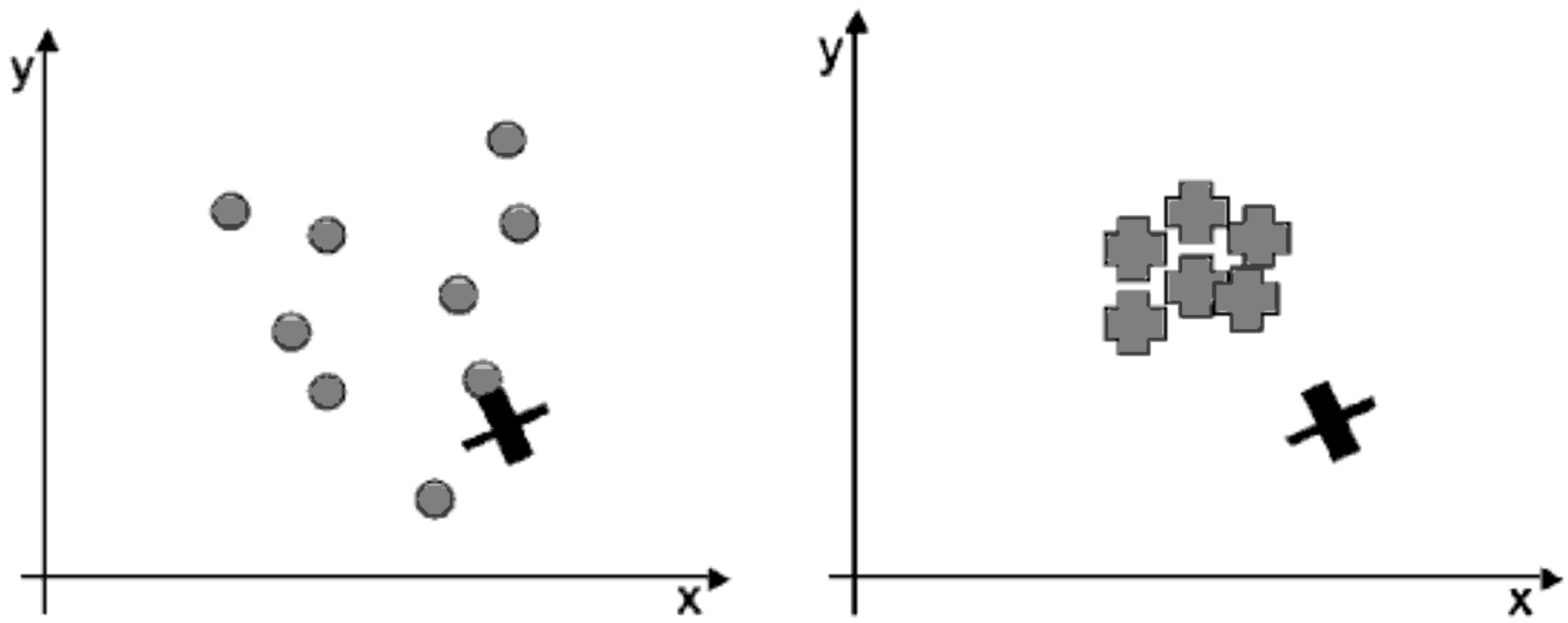


FIGURE 8.4: Two different datasets and a testpoint.

$$\boldsymbol{\Sigma} = \begin{pmatrix} E[(\mathbf{x}_1 - \boldsymbol{\mu}_1)(\mathbf{x}_1 - \boldsymbol{\mu}_1)] & E[(\mathbf{x}_1 - \boldsymbol{\mu}_1)(\mathbf{x}_2 - \boldsymbol{\mu}_2)] & \dots & E[(\mathbf{x}_1 - \boldsymbol{\mu}_1)(\mathbf{x}_n - \boldsymbol{\mu}_n)] \\ E[(\mathbf{x}_2 - \boldsymbol{\mu}_2)(\mathbf{x}_1 - \boldsymbol{\mu}_1)] & E[(\mathbf{x}_2 - \boldsymbol{\mu}_2)(\mathbf{x}_2 - \boldsymbol{\mu}_2)] & \dots & E[(\mathbf{x}_2 - \boldsymbol{\mu}_2)(\mathbf{x}_n - \boldsymbol{\mu}_n)] \\ \dots & \dots & \dots & \dots \\ E[(\mathbf{x}_n - \boldsymbol{\mu}_n)(\mathbf{x}_1 - \boldsymbol{\mu}_1)] & E[(\mathbf{x}_n - \boldsymbol{\mu}_n)(\mathbf{x}_2 - \boldsymbol{\mu}_2)] & \dots & E[(\mathbf{x}_n - \boldsymbol{\mu}_n)(\mathbf{x}_n - \boldsymbol{\mu}_n)] \end{pmatrix} \tag{8.15}$$

where $\mathbf{x}_i$ describes the elements of the ith variable, and $\boldsymbol{\mu}_i$ is their mean. Note that the matrix is square, that the elements on the trace (the leading diagonal) of the matrix are equal to one (since each variable must be exactly dependent upon itself), and that it is symmetric since $\text{cov}(\mathbf{x}_i, \mathbf{x}_j) = \text{cov}(\mathbf{x}_j, \mathbf{x}_i)$. Equation (8.15) 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 10 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 8.4 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:

$$D_M(\mathbf{x}) = \sqrt{(\mathbf{x} - \boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1} (\mathbf{x} - \boldsymbol{\mu})}, \tag{8.16}$$

where $\mathbf{x}$ is the usual column vector of data with mean vector $\boldsymbol{\mu}$, and $\boldsymbol{\Sigma}^{-1}$ is the inverse of the covariance matrix. If we set the covariance matrix to

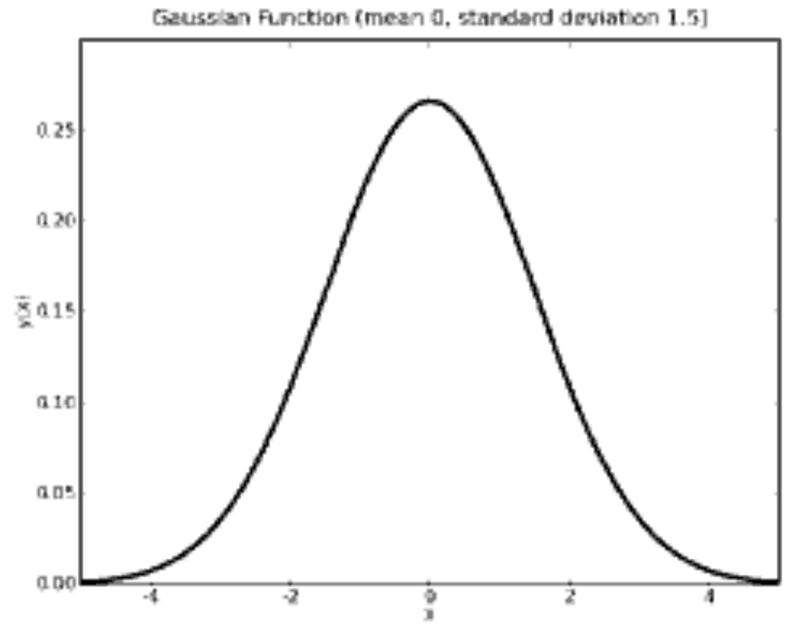


FIGURE 8.5: Plot of the one-dimensional Gaussian curve.

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 (`cov(x)` for data matrix `x`) and the inverse is called `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.

8.2.3 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 8.5, and its equation in one dimension is:

$$p(x) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(\frac{-(x-\mu)^2}{2\sigma^2}\right), \tag{8.17}$$

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:

$$p(\mathbf{x}) = \frac{1}{(2\pi)^{d/2}|\mathbf{\Sigma}|^{1/2}} \exp\left(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^T\mathbf{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})\right), \tag{8.18}$$

where $\mathbf{\Sigma}$ is the $n \times n$ covariance matrix (with $|\mathbf{\Sigma}|$ being its determinant and $\mathbf{\Sigma}^{-1}$ being its inverse). Figure 8.6 shows the appearance in two dimensions of

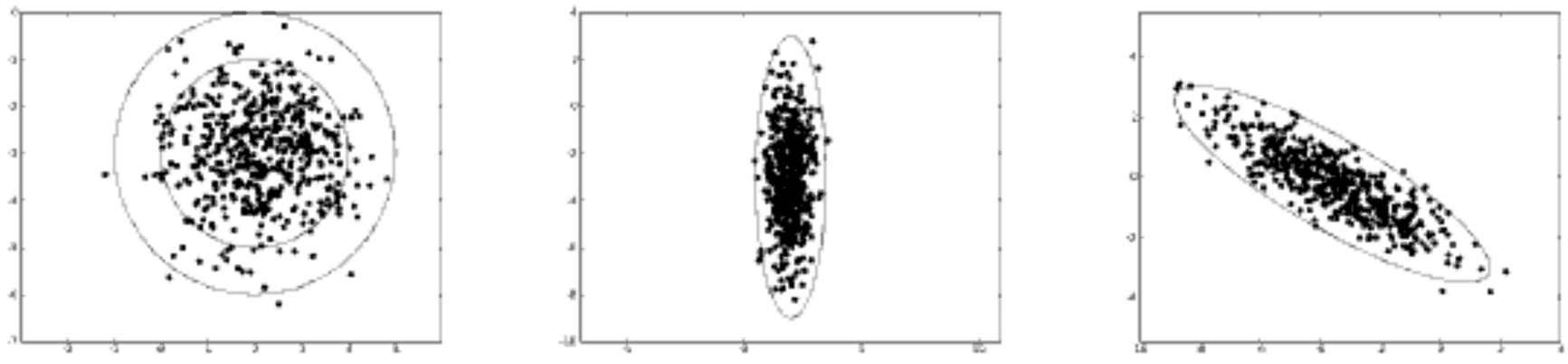


FIGURE 8.6: The two-dimensional Gaussian when (*left*) the covariance matrix is the identity, (*centre*) the covariance matrix has elements on the leading diagonal only, and (*right*) the general case.

three different cases: when the covariance matrix is the identity, when there are only numbers on the trace (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.

8.2.4 The Bias-Variance Tradeoff

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 trade-off, 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 spline. 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.

When looking at the usual sum-of-squares error function we can split it up into separate pieces that represent the bias and the variance:

$$E((\mathbf{y} - \hat{f}(\mathbf{x}))^2) = \sigma^2 + \left[f(\mathbf{x}) - \frac{1}{k} \sum_{i=0}^{n} f(\mathbf{x}_n) \right]^2 + \frac{\sigma^2}{k}. \tag{8.19}$$

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 the square of the bias, while the third is the variance. 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.

8.3 Gaussian Mixture Models

For the Bayes' classifier that we saw in Section 8.1.2 the data was labelled, so we could use the labels to perform the classification. However, suppose that the data is not labelled (unsupervised learning). We will see lots of ways to deal with this in Chapters 9 and 10, 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 9.1. 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:

$$f(\mathbf{x}) = \sum_{m=1}^{M} \alpha_m \phi(\mathbf{x}; \boldsymbol{\mu}_m, \boldsymbol{\Sigma}_m), \tag{8.20}$$

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 $\sum_{m=1}^{M} \alpha_m = 1$.

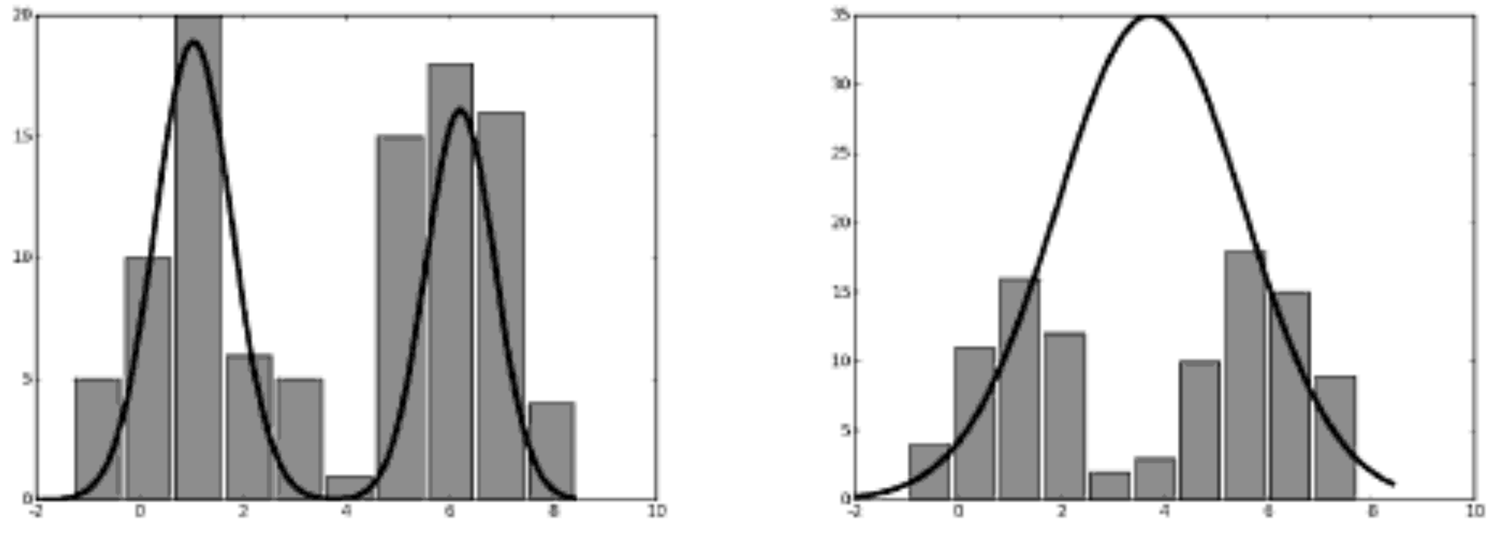


FIGURE 8.7: Histograms of training data from a mixture of two Gaussians and two fitted models, shown as the line plot. The model shown on the left fits well, but the one on the right produces two Gaussians right on top of each other that do not fit the data well.

Figure 8.7 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:

$$p(\mathbf{x}_i \in c_m) = \frac{\hat{\alpha}_m \phi(\mathbf{x}_i; \hat{\boldsymbol{\mu}}_m; \hat{\boldsymbol{\Sigma}}_m)}{\sum_{k=1}^{M} \hat{\alpha}_m \phi(\mathbf{x}_i; \hat{\boldsymbol{\mu}}_k; \hat{\boldsymbol{\Sigma}}_k)}. \tag{8.21}$$

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 minimise 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 15.3.3, 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.

8.3.1 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 $\boldsymbol{\sigma}$):

$$\begin{aligned} G_1 &= \mathcal{N}(\boldsymbol{\mu}_1, \boldsymbol{\sigma}_1^2) \\ G_2 &= \mathcal{N}(\boldsymbol{\mu}_2, \boldsymbol{\sigma}_2^2) \\ y &= pG_1 + (1-p)G_2. \end{aligned} \tag{8.22}$$

If the probability distribution of p is written as $\boldsymbol{\pi}$, then the probability density is:

$$P(\mathbf{y}) = \boldsymbol{\pi}\phi(\mathbf{y}; \boldsymbol{\mu}_1, \boldsymbol{\sigma}_1) + (1-\boldsymbol{\pi})\phi(\mathbf{y}; \boldsymbol{\mu}_2, \boldsymbol{\sigma}_2). \tag{8.23}$$

Finding the maximum likelihood solution (actually the minimum log likelihood) to this problem is then a case of computing the sum of the logarithm of Equation (8.23) 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 data point 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 the expectation of it from the data:

$$\begin{aligned} \boldsymbol{\gamma}_i(\hat{\boldsymbol{\mu}}_1, \hat{\boldsymbol{\mu}}_2, \hat{\boldsymbol{\sigma}}_1, \hat{\boldsymbol{\sigma}}_2, \hat{\boldsymbol{\pi}}) &= E(f|\hat{\boldsymbol{\mu}}_1, \hat{\boldsymbol{\mu}}_2, \hat{\boldsymbol{\sigma}}_1, \hat{\boldsymbol{\sigma}}_2, \hat{\boldsymbol{\pi}}, D) \\ &= P(f = 1|\hat{\boldsymbol{\mu}}_1, \hat{\boldsymbol{\mu}}_2, \hat{\boldsymbol{\sigma}}_1, \hat{\boldsymbol{\sigma}}_2, \hat{\boldsymbol{\pi}}, D), \end{aligned} \tag{8.24}$$

where D denotes the data and a hat ($\hat{\cdot}$) means an estimate of a variable.

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 $\hat{\mu}_1$, $\hat{\mu}_2$ 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) $\hat{\gamma}_i = \frac{\hat{\pi}\phi(y_i;\hat{\mu}_i,\hat{\sigma}_i)}{(1-\hat{\pi})\phi(y_i;\hat{\mu}_1,\hat{\sigma}_1)+\hat{\pi}\phi(y_i;\hat{\mu}_2,\hat{\sigma}_2)}$ 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) $\hat{\sigma}_1 = \frac{\sum_{i=1}^{N}(1-\hat{\gamma}_i)(y_i-\hat{\mu}_i)^2}{\sum_{i=1}^{N}(1-\hat{\gamma}_i)}$
  - (M-step 4) $\hat{\sigma}_2 = \frac{\sum_{i=1}^{N}\hat{\gamma}_i(y_i-\hat{\mu}_i)^2}{\sum_{i=1}^{N}\hat{\gamma}_i}$
  - (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*exp(-(y[i]-mu2)**2/s2)/((1-pi) * exp(-(y[
```

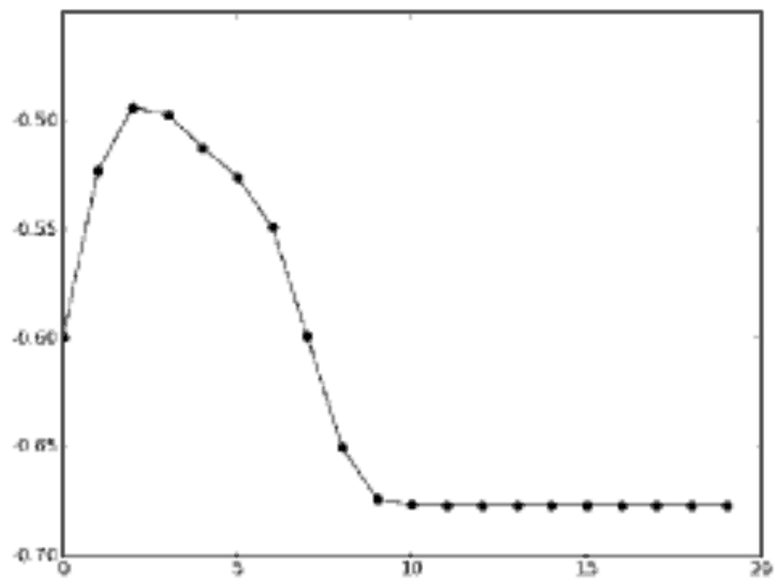


FIGURE 8.8: Plot of the log likelihood changing as the Gaussian Mixture Model EM algorithm learns to fit the two Gaussians shown on the left of Figure 8.7.

```
    i]-mu1)**2/s1) + pi* exp(-(y[i]-mu2)**2/s2))

# M-step
mu1 = sum((1-gamma)*y)/sum(1-gamma)
mu2 = sum(gamma*y)/sum(gamma)
s1 = sum((1-gamma)*(y-mu1)**2)/sum(1-gamma)
s2 = sum(gamma*(y-mu2)**2)/sum(gamma)
pi = sum(gamma)/N

ll[count-1] = sum(log((1-pi)*exp(-(y[i]-mu1)**2/s1) +
pi*exp(-(y[i]-mu2)**2/s2)))
```

Figure 8.8 shows the log likelihood dropping as the algorithm learns for the example on the left of Figure 8.7. 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.

8.4 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 8.9.

This method suffers from the curse of dimensionality (Section 4.3). First, as shown above, the computational costs get higher as the number of dimensions

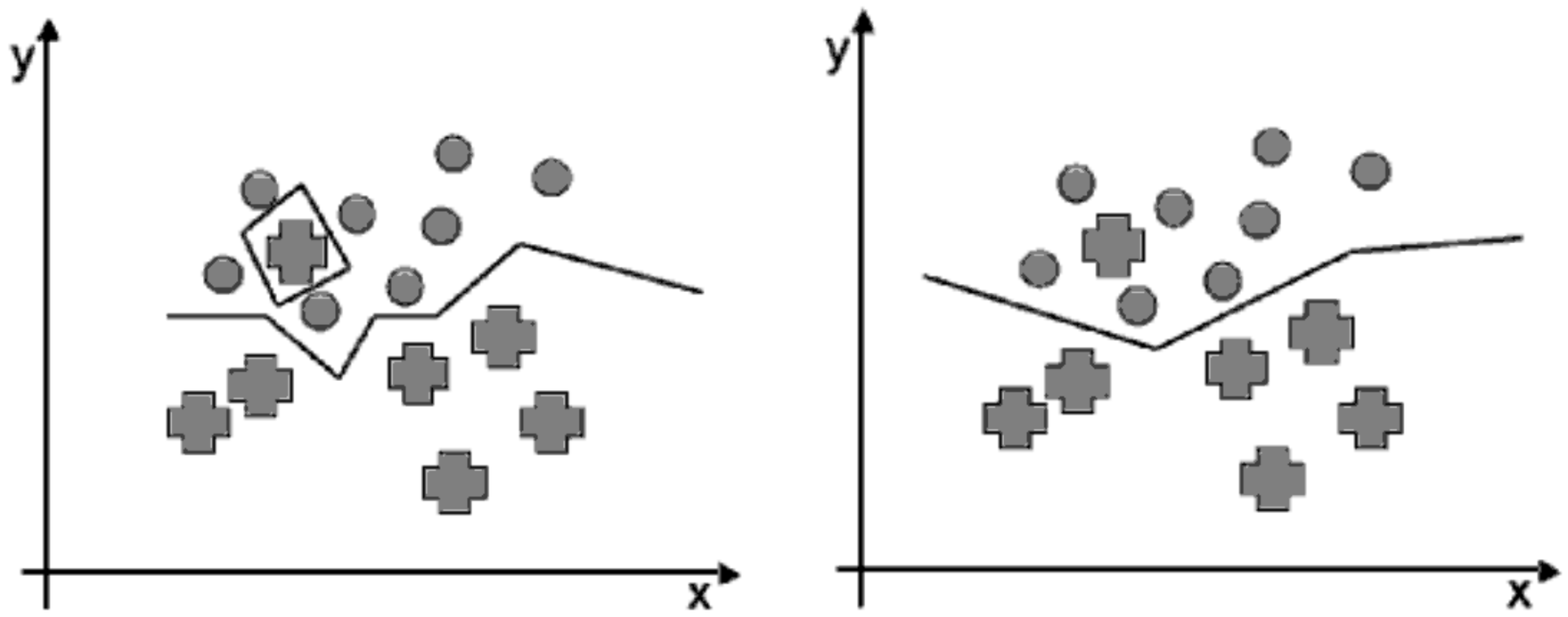


FIGURE 8.9: The nearest neighbours decision boundary with *left:* one neighbour and *right:* two neighbours.

grows. This is not as bad as it might appear at first: there are a set of methods such as KD-Trees (see Section 8.4.2 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 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 = shape(inputs)[0]
    closest = zeros(nInputs)

    for n in range(nInputs):
        # Compute distances
        distances = sum((data-inputs[n,:])**2,axis=1)

        # Identify the nearest neighbours
        indices = argsort(distances,axis=0)

        classes = unique(dataClass[indices[:k]])
        if len(classes)==1:
            closest[n] = unique(classes)
        else:
            counts = zeros(max(classes)+1)
```

```
    for i in range(k):
        counts[dataClass[indices[i]]] += 1
        closest[n] = 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.

8.4.1 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 much more about kernels in a different context in Section 5.2, 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:

$$K_{E,\lambda}(x_0, x) = \begin{cases} 0.75\left(1 - (x_0 - x)^2/\lambda^2\right) & \text{if } |x - x_0| < \lambda \\ 0 & \text{otherwise} \end{cases}, \tag{8.25}$$

and the tricube kernel:

$$K_{T,\lambda}(x_0, x) = \begin{cases} \left(1 - \left|\frac{x_0 - x}{\lambda}\right|^3\right)^3 & \text{if } |x - x_0| < \lambda \\ 0 & \text{otherwise} \end{cases}. \tag{8.26}$$

The results of using these kernels are shown in Figure 8.10 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.

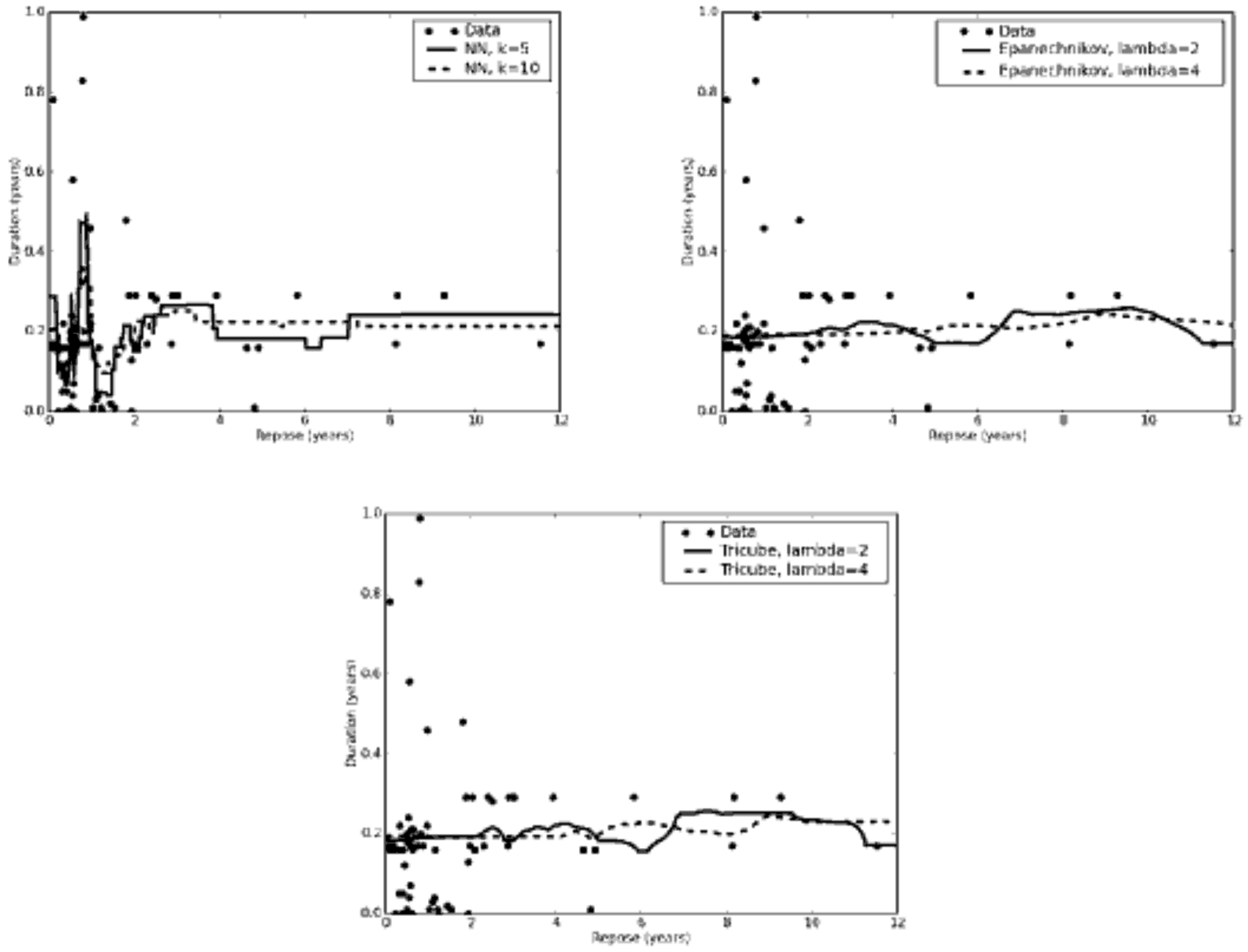


FIGURE 8.10: Output of the nearest neighbour method and two kernel smoothers on the data of duration and repose of eruptions of Mount Ruapehu 1860–2006.

8.4.2 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 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 6), 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 = mod(depth,shape(points)[1])

# Find the median point
indices = argsort(points[:,whichAxis])
points = points[indices,:]
median = ceil(float(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 8.11). 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 8.12 and the tree shown in Figure 8.13.

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 leaf node that is in the region containing the test point. Using the tree shown in Figure 8.13 we introduce the testpoint $(3,5)$, which finds $(2,2)$ as the leaf for the box that $(3,5)$ is in. However, looking at Figure 8.14 we see that this is not the closest point at

Hidden page

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 testpoint 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 8.14 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 testpoint. 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 sibling 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. This leads to the following Python program:

```
def returnNearest(tree,point,depth):
    if tree.left is None:
        # Have reached a leaf
        distance = sum((tree.point-point)**2)
        return tree.point,distance,0
    else:
        # Pick next axis to split on
        whichAxis = mod(depth,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 = returnNearest(tree.
                right,point,depth+1)
            else:
                bestGuess2,distance2,height2 = returnNearest(tree.
                left,point,depth+1)

        # Check this node
        distance3 = sum((tree.point-point)**2)
        if (distance3<distance2):
            distance2 = distance3
            bestGuess2 = tree.point
```

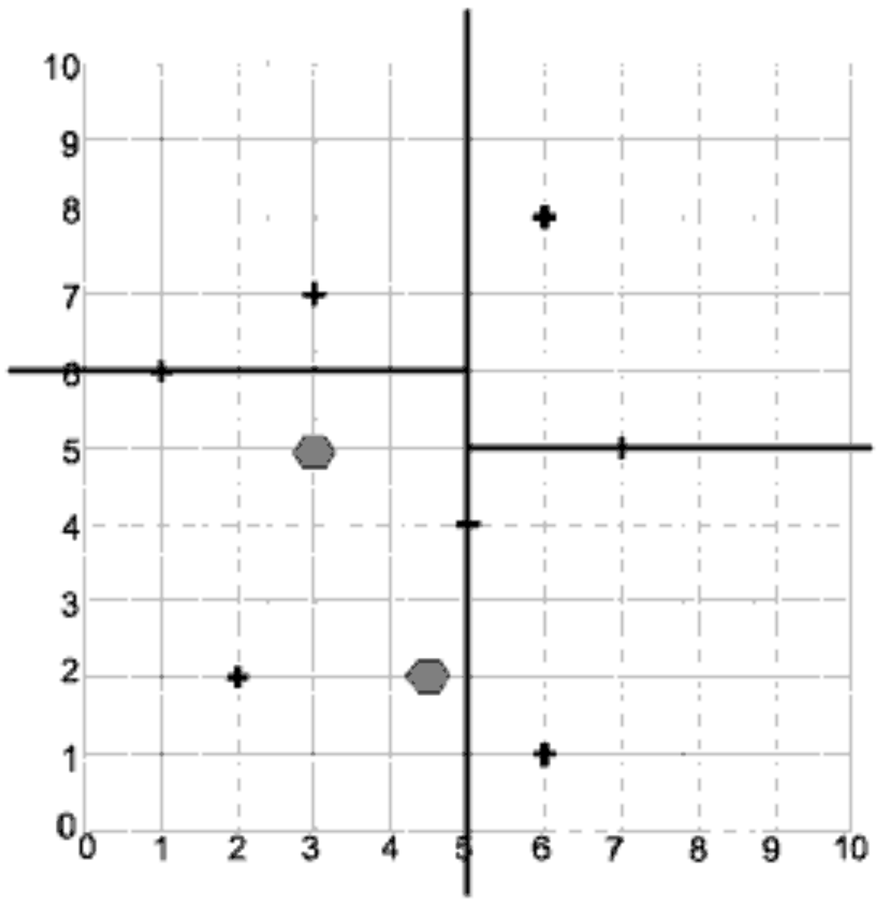


FIGURE 8.14: Two testpoints for the example KD-tree.

```
if (distance2<distance):
    distance = distance2
    bestGuess = bestGuess2
return bestGuess,distance,height+1
```

8.4.3 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:

Hidden page

If we put $k = 1$ then we get the city-block distance (Equation (8.28)), and $k = 2$ gives the Euclidean distance (Equation (8.27)). 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 data point, 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 in the chapter: the Mahalanobis distance in Section 8.2.2.

There are plenty of other possible metrics to choose, depending upon the data space. 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,

Hidden page

Problem 8.3 Use the naïve Bayes classifier on the datasets that you used for the decision tree (this will involve some effort in turning the textual data into probabilities) and compare the results.

Problem 8.4 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 8.5 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 8.6 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.

Chapter 9

Unsupervised Learning

The learning algorithms that we have seen to date have made use of a training set that consists of a collection of labelled **target** data. This is obviously useful, since it enables us to show the algorithm the correct answer to sample data, but in many circumstances it is difficult to obtain—it could, for instance, involve somebody labelling each instance by hand. In addition, it doesn't seem to be very biologically plausible: most of the time when we are learning, we don't get told exactly what the right answer should be. In this chapter we will consider exactly the opposite case, where there is no information about the correct outputs available at all, and the algorithm is left to spot some similarity between different inputs for itself. We will see some more methods of doing this in Chapter 10, while in Chapter 13 we will consider the intermediate possibility, where the system receives information about whether or not it is correct, but is not told how to correct errors.

Unsupervised learning is a conceptually different problem. Obviously, we can't hope to perform regression—we don't know the outputs for any points, so we can't guess what the function is. Can we hope to do classification then? The aim of classification is to identify similarities between inputs that belong to the same class. There isn't any information about the correct classes, but if the algorithm can exploit similarities between inputs in order to **cluster** inputs that are similar together, this might perform classification automatically. So the aim of unsupervised learning is to find clusters of similar inputs in the data without being explicitly told that these datapoints belong to one class and those to a different class. Instead, the algorithm has to discover the similarities for itself.

The supervised learning algorithms that we have discussed so far have aimed to minimise some external error criterion—mostly the sum-of-squares error—based on the difference between the targets and the outputs. Calculating and minimising this error was possible because we had target data to calculate it from, which is not true for unsupervised learning. This means that we need to find something else to drive the learning. The problem is more general than sum-of-squares error: we can't use any error criterion that relies on targets or other outside information (an **external** error criterion), we need to find something internal to the algorithm. This means that the measure has to be independent of the task, because we can't keep on changing the whole algorithm every time a new task is introduced. In supervised learning the

error criterion was task-specific, because it was based on the target data that we provided.

To see how to work out a general error criterion that we can use, we need to go back to some of the important concepts that were discussed in Section 4.1.1: input space and weight space. If two inputs are close together then it means that their vectors are similar, and so the distance between them is small (distance measures were discussed in Section 8.4.3, but here we will stick to Euclidean distance). Then inputs that are close together are identified as being similar, so that they can be clustered, while inputs that are far apart are not clustered together. We can extend this to the nodes of a network by aligning weight space with input space. Now if the weight values of a node are similar to the elements of an input vector then that node should be a good match for the input, and any other inputs that are similar. In order to start to see these ideas in practice we'll look at a simple clustering algorithm, the k-means algorithm, which has been around in statistics for a long time.

9.1 The k-Means Algorithm

If you have ever watched a group of tourists with a couple of tour guides who hold umbrellas up so that everybody can see them and follow them, then you have seen a dynamic version of the k-means algorithm. Our version is simpler, because the data (playing the part of the tourists) does not move, only the tour guides.

Suppose that we want to divide our input data into k categories, where we know the value of k (for example, we have a set of medical test results from lots of people for three diseases, and we want to see how well the tests identify the three diseases). We allocate k cluster centres to our input space, and we would like to position these centres so that there is one cluster centre in the middle of each cluster. However, we don't know where the clusters are, let alone where their 'middle' is, so we need an algorithm that will find them. Learning algorithms generally try to minimise some sort of error, so we need to think of an error criterion that describes this aim. The idea of the 'middle' is the first thing that we need to think about. How do we define the middle of a set of points? There are actually two things that we need to define:

A distance measure In order to talk about distances between points, we need some way to measure distances. It is often the normal Euclidean distance, but there are other alternatives; we've covered some other alternatives in Section 8.4.3.

The mean average Once we have a distance measure, we can compute the central point of a set of datapoints, which is the mean average (if you

aren't convinced, think what the mean of two numbers is, it is the point halfway along the line between them). Actually, this is only true in Euclidean space, which is the one you are used to, where everything is nice and flat. Everything becomes a lot trickier if we have to think about curved spaces; when we have to worry about curvature, the Euclidean distance metric isn't the right one, and there are at least two different definitions of the mean. So we aren't going to worry about any of these things, and we'll assume that space is flat. This is what statisticians do all the time.

We can now think about a suitable way of positioning the cluster centres: we compute the mean point of each cluster, $\boldsymbol{\mu}_{c(i)}$, and put the cluster centre there. This is equivalent to minimising the Euclidean distance (which is the sum-of-squares error again) from each datapoint to its cluster centre.

How do we decide which points belong to which clusters? It is important to decide, since we will use that to position the cluster centres. The obvious thing is to associate each point with the cluster centre that it is closest too. This might change as the algorithm iterates, but that's fine.

We start by positioning the cluster centres randomly though the input space, since we don't know where to put them, and then we update their positions according to the data. We decide which cluster each datapoint belongs to by computing the distance between each datapoint and all of the cluster centres, and assigning it to the cluster that is the closest. Note that we can reduce the computational cost of this procedure by using the KD-Tree algorithm that was described in Section 8.4.2. For all of the points that are assigned to a cluster, we then compute the mean of them, and move the cluster centre to that place. We iterate the algorithm until the cluster centres stop moving. Here is the algorithmic description:

The k-Means Algorithm

- **Initialisation**
  - choose a value for k
  - choose k random positions in the input space
  - assign the cluster centres $\boldsymbol{\mu}_j$ to those positions
- **Learning**
  - repeat
    - for each datapoint $\mathbf{x}_i$:
      - compute the distance to each cluster centre
      - assign the datapoint to the nearest cluster centre with distance

$$d_i = \min_j d(\mathbf{x}_i, \boldsymbol{\mu}_j). \tag{9.1}$$

        * for each cluster centre:
            - move the position of the centre to the mean of the points in that cluster (N_j is the number of points in cluster j):

$$\boldsymbol{\mu}_j = \frac{1}{N_j} \sum_{i=1}^{N_j} \mathbf{x}_i \tag{9.2}$$

    - until the cluster centres stop moving

- **Usage**

    - for each test point:
        * compute the distance to each cluster centre
        * assign the datapoint to the nearest cluster centre with distance

$$d_i = \min_j d(\mathbf{x}_i, \boldsymbol{\mu}_j). \tag{9.3}$$

The NumPy implementation follows these steps almost exactly, and we can take advantage of the `argmin()` function, which returns the index of the minimum value, to find the closest cluster. The code that computes the distances, finds the nearest cluster centre, and updates them can then be written as:

```
# Compute distances
distances = ones((1,self.nData))*sum((data-self.centres[0,:])
**2,axis=1)
for j in range(self.k-1):
    distances = append(distances,ones((1,self.nData))*sum((data-
    self.centres[j+1,:])**2,axis=1),axis=0)

# Identify the closest cluster
cluster = distances.argmin(axis=0)
cluster = transpose(cluster*ones((1,self.nData)))

# Update the cluster centres
for j in range(self.k):
    thisCluster = where(cluster==j,1,0)
    if sum(thisCluster)>0:
        self.centres[j,:] = sum(data*thisCluster,axis=0)/sum(
        thisCluster)
```

To see how this works in practice, Figures 9.1 and 9.2 show some data and some different ways to cluster that data computed by the k-means algorithm.

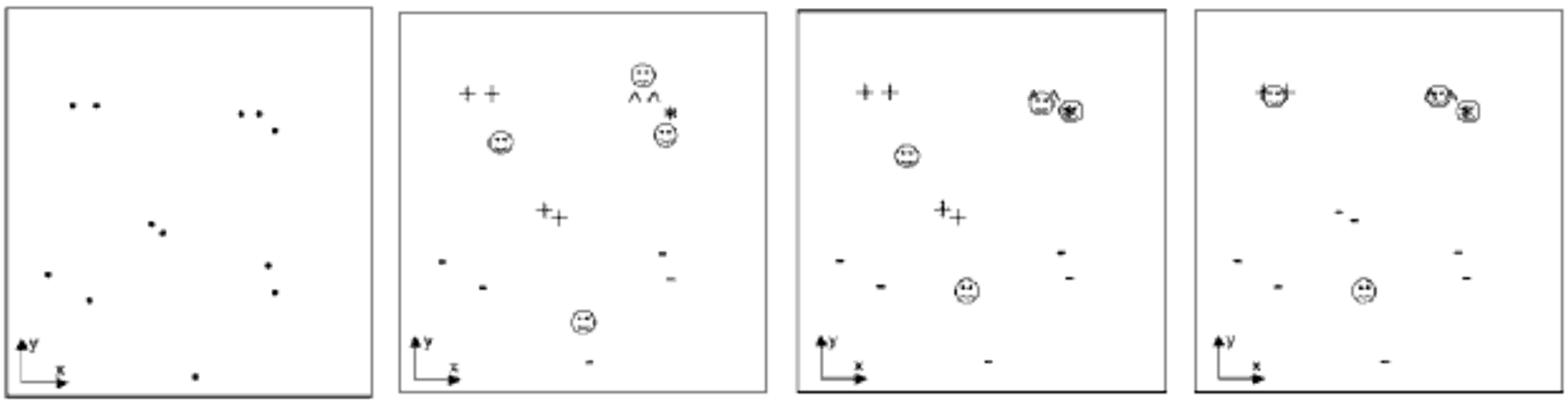


FIGURE 9.1: *Left:* A two-dimensional dataset. *Right:* Three possible ways to position 4 centres (drawn as faces) using the k-means algorithm, which is clearly susceptible to local minima.

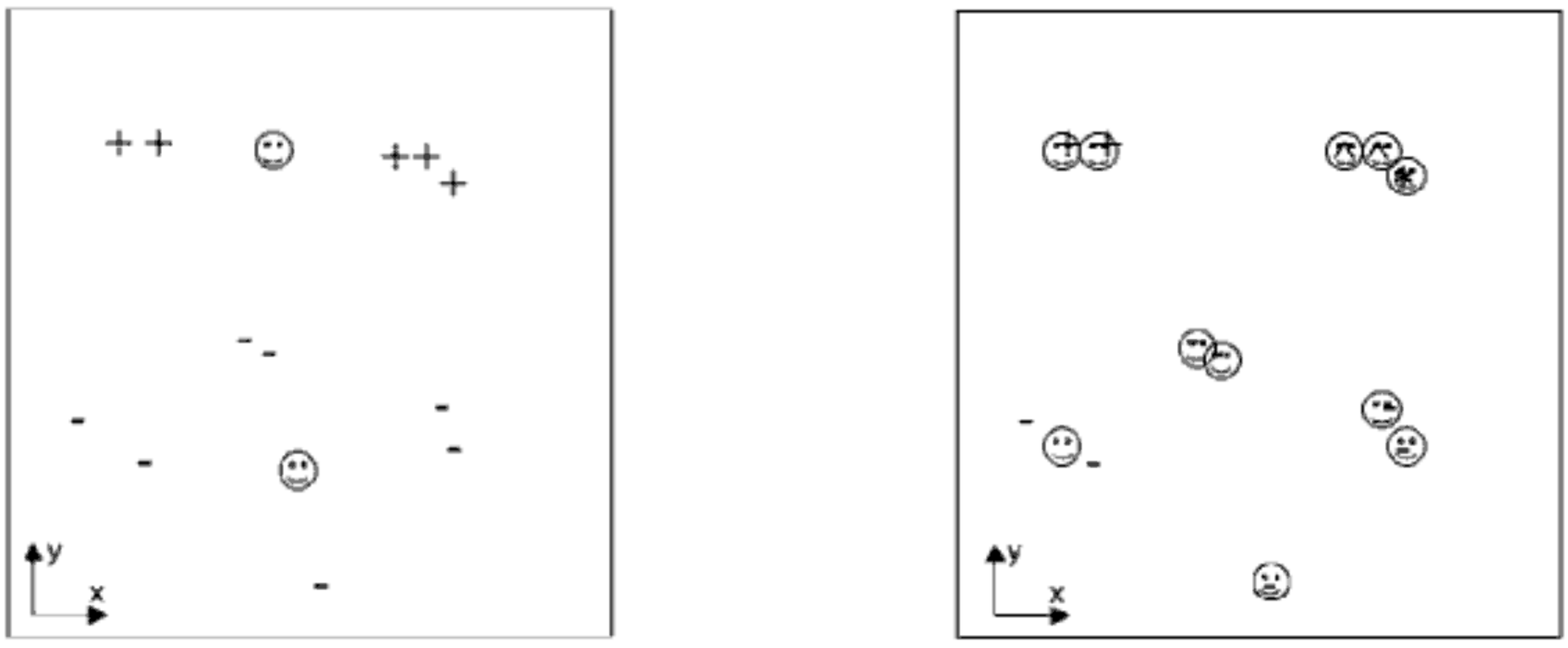


FIGURE 9.2: *Left:* A solution with only 2 classes, which does not match the data well. *Right:* A solution with 11 classes, showing severe overfitting.

It should be clear that the algorithm is susceptible to local minima: depending upon where the centres are initially positioned in the space, you can get very different solutions, and many of them look very unlikely to our eyes. Figure 9.2 shows examples of what happens when you choose the number of centres wrongly. There are certainly cases where we don't know in advance how many clusters we will see in the data, but the k-means algorithm doesn't deal with this at all well.

At the cost of significant extra computational expense, we can get around both of these problems by running the algorithm many different times. To find a good local optimum (or even the global one) we use many different initial centre locations, and the solution that minimises the overall sum-of-squares error is likely to be the best one.

By running the algorithm with lots of different values of k, we can see which values give us the best solution. Of course, we need to be careful with this. If we still just measure the sum-of-squares error, then when we set k to be equal to the number of datapoints, we can position one centre on every datapoint, and the sum-of-squares error will be zero (in fact, this won't happen, since the random initialisation will mean that several clusters will end up coinciding).

However, there is no generalisation in this solution (it is a case of serious overfitting). As usual, the answer is to use a validation set and monitor the error there.

9.1.1 Dealing with Noise

There are lots of reasons for performing clustering, but one of the more common ones is to deal with noisy data readings. These might be slightly corrupted, or occasionally just plain wrong. If we can choose the clusters correctly, then we have effectively removed the noise, because we replace each noisy datapoint by the cluster centre (we will use this way of representing datapoints for other purposes in Section 9.2). Unfortunately, the mean average, which is central to the k-means algorithm, is very susceptible to outliers, i.e., very noisy measurements. One way to avoid the problem is to replace the mean average with the median, which is what is known as a robust statistic, meaning that it is not affected by outliers (the mean of $(1, 2, 1, 2, 100)$ is 21.2, while the median is 2). The only change that is needed to the algorithm is to replace the computation of the mean with the computation of the median. This is computationally more expensive, as we've discussed previously, but it does remove noise effectively.

9.1.2 The k-Means Neural Network

The k-means algorithm clearly works, despite its problems with noise and the difficulty with choosing the number of clusters. Interestingly, while it might seem a long way from neural networks, it isn't. If we think about the cluster centres that we optimise the positions of as locations in weight space, then we could position neurons in those places and use neural network training. The computation that happened in the k-means algorithm was that each input decided which cluster centre it was closest to by calculating the distance to all of the centres. We could do this inside a neural network, too: the location of each neuron is its position in weight space, which matches the values of its weights. So for each input, we just make the activation of a node be the distance between that node in weight space and the current input, as we did for Radial Basis Functions in Chapter 4. Then training is just moving the position of the node, which means adjusting the weights.

So, we can implement the k-means algorithm using a set of neurons. We will use just one layer of neurons, together with some input nodes, and no bias node. The first layer will be the inputs, which don't do any computation, as usual, and the second layer will be a layer of competitive neurons, that is, neurons that 'compete' to fire, with only one of them actually succeeding. Only one cluster centre can represent a particular input vector, and so we will choose the neuron with the highest activation h to be the one that fires. This is known as winner-takes-all activation, and it is an example of competitive learning, since the set of neurons compete with each other to fire,

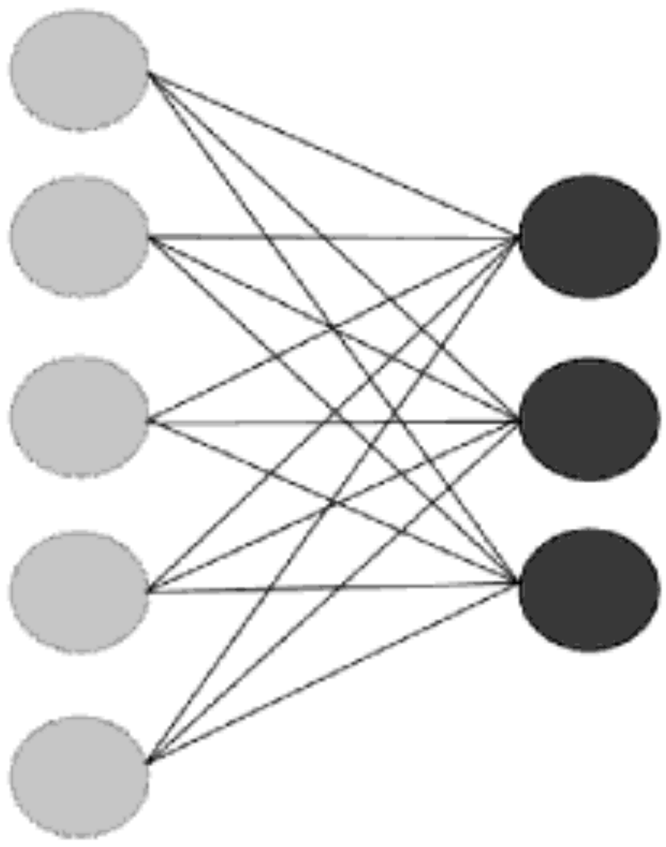


FIGURE 9.3: A single-layer neural network can implement the k-means solution.

with the winner being the one that best matches (i.e., is closest to) the input. Competitive learning is sometimes said to lead to grandmother cells, because each neuron in the network will learn to recognise one particular feature, and will fire only when that input is seen. You would then have a specific neuron that was trained to recognise your grandmother (and others for anybody else/anything else that you see often).

We will choose k neurons (for hopefully obvious reasons) and fully connect the inputs to the neurons, as usual. There is a picture of this network in Figure 9.3. We will use neurons with a linear transfer function, computing the activation of the neurons as simply the product of the weights and inputs:

$$h_i = \sum_j w_{ij} x_j. \tag{9.4}$$

Providing that the inputs are normalised so that their absolute size is the same (a point that we'll come back to in Section 9.1.3), this effectively measures the distance between the input vector and the cluster centre represented by that neuron, with larger numbers (higher activations) meaning that the two points are closer together.

So the winning neuron is the one that is closest to the current input. The question is how can we then change the position of that neuron in weight space, that is, how do we update its weights? In the k-means algorithm that was described earlier it was easy: we just set the cluster centre to be the mean of all the datapoints that were assigned to that centre. However, when we do neural network training, we often just feed in one input vector at a time and change the weights (that is, we use the algorithm on-line, rather than batch), we do not know the mean because we don't know about all the datapoints, just the current one. So we approximate it by moving the winning neuron closer to the current input, making that centre even more likely to be the best

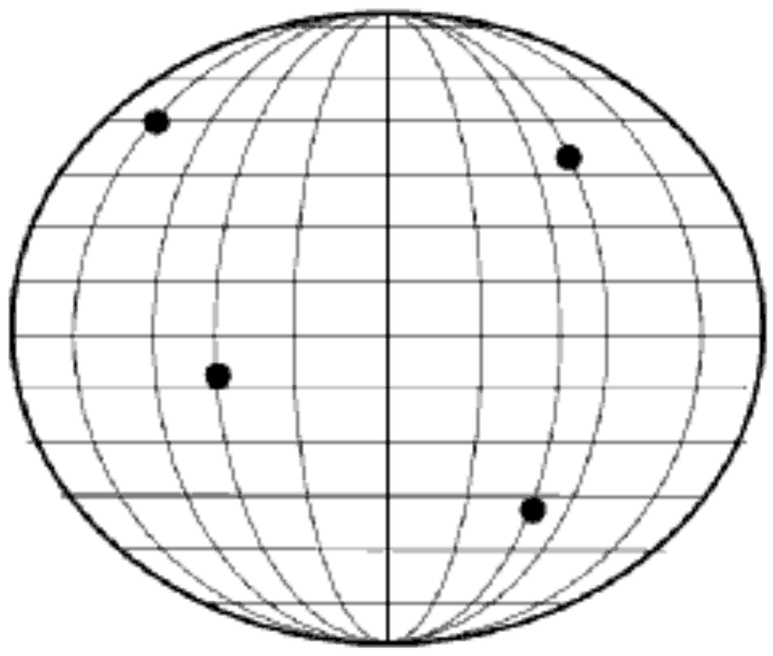


FIGURE 9.4: A set of neurons positioned on the unit sphere in 3D.

match next time that input is seen. This corresponds to:

$$\Delta w_{ij} = \eta x_j. \tag{9.5}$$

However, this is not good enough. To see why not, let's get back to that question of normalisation. This is important enough to need its own subsection.

9.1.3 Normalisation

Suppose that the weights of all the neurons are small (maybe less than 1) except for those to one particular neuron. We'll make those weights be 10 for the example. If an input vector with values $(0.2, 0.2, -0.1)$ is presented, and it happens to be an exact match for one of the neurons, then the activation of that neuron will be $0.2 \times 0.2 + 0.2 \times 0.2 + -0.1 \times -0.1 = 0.09$. The other neurons are not perfect matches, so their activations should all be less. However, consider the neuron with large weights. Its activation will be $10 \times 0.2 + 10 \times 0.2 + 10 \times -0.1 = 3$, and so it will be the winner. Thus, we can only compare activations if we know that the weights for all of the neurons are the same size. We do this by insisting that the weight vector is normalised so that the distance between the vector and the origin (the point $(0, 0, \ldots 0)$) is one. This means that all of the neurons are positioned on the unit hypersphere, which we described in Section 4.3 when we talked about the curse of dimensionality: it is the set of all points that are distance one from the origin, so it is a circle in 2D, a sphere in 3D (as shown in Figure 9.4), and a hypersphere in higher dimensions.

Computing this normalisation in NumPy takes a little bit of care because we are normalising the total Euclidean distance from the origin, and the sum and division are row-wise rather than column-wise, which means that the matrix has to be transposed before and after the division:

```
normalisers=sqrt(sum(data**2,axis=1))*ones((1,shape(data)[0]))
data = transpose(transpose(data)/normalisers)
```

The neuronal activation (Equation (9.4)) can be written as:

$$h_i = \mathbf{W}_i^T \cdot \mathbf{x}, \tag{9.6}$$

where, as usual, $\cdot$ refers to the inner product or scalar product between the two vectors, and $\mathbf{W}_i^T$ is the transpose of the ith row of W. The inner product computes $\|\mathbf{W}_i\|\|\mathbf{x}\|\cos\theta$, where θ is the angle between the two vectors and $\|\cdot\|$ is the magnitude of the vector. So if the magnitude of all the vectors is one, then only the angle θ affects the size of the dot product, and this tells us about the difference between the vector directions, since the more they point in the same direction, the larger the activation will be.

9.1.4 A Better Weight Update Rule

The weight update rule given in Equation (9.5) lets the weights grow without any bound, so that they do not lie on the unit hypersphere any more. If we normalise the inputs as well, which certainly seems reasonable, then we can use the following weight update rule:

$$\Delta w_{ij} = \eta(x_j - w_{ij}), \tag{9.7}$$

which has the effect of moving the weight w_{ij} directly towards the current input. Remember that the only weights that we are updating are those of the winning unit:

```
for i in range(self.nEpochs):
    for j in range(self.nData):
        activation = sum(self.weights*transpose(data[j:j+1,:]),
        axis=0)
        winner = argmax(activation)
        self.weights[:,winner] += self.eta * data[j,:] - self.
        weights[:,winner]
```

For many of our supervised learning algorithms we minimised the sum-of-squares difference between the output and the target. This was a global error criterion that affected all of the weights together. Now we are minimising a function that is effectively independent in each weight. So the minimisation that we are doing is actually more complicated, even though it doesn't look it. This makes it very difficult to analyse the behaviour of the algorithm, which is a general problem for competitive learning algorithms. However, they do tend to work well.

Now that we have a weight update rule that works, we can consider the entire algorithm for the on-line k-means network.

The On-Line k-Means Algorithm

- **Initialisation**
  - choose a value for k, which corresponds to the number of output nodes
  - initialise the weights to have small random values
- **Learning**
  - normalise the data so that all the points lie on the unit sphere
  - repeat:
    - for each datapoint:
      - compute the activations of all the nodes
      - pick the winner as the node with the highest activation
      - update the weights using Equation (9.7)
    - until number of iterations is above a threshold
  - **Usage**
    - for each test point:
      - compute the activations of all the nodes
      - pick the winner as the node with the highest activation

9.1.5 Example: The Iris Dataset Again

Now that we have a method of training the k-means algorithm we can use it to learn about data. Except we need to think about how to understand the results. If there aren't any labels in the data, then we can't really do much to analyse the results, since we don't have anything to compare them with. However, we might use unsupervised learning methods to cluster data where we know at least some of the labels. For example, we can use the algorithm on the iris dataset that we looked at in Section 3.4.3, where we classified three types of iris flowers using the MLP. All we need to do is to give some of the data to the algorithm and train it, and then use some more to test the output. However, the output of the algorithm isn't as clear now, because we don't use the labels that come with the data, since we aren't doing supervised learning anymore. To get around that, we need to work out some way of turning the results from the algorithm, which is the index of the cluster that best matches it, into a classification output that we can compare with the labels. This is relatively easy if we used three clusters in the algorithm, since there should

hopefully be a one-to-one correspondence between them, but it might turn out that using more clusters gets better results, although this will make the analysis more difficult. You can do this by hand if there are relatively small numbers of datapoints, or you could use a supervised learning algorithm to do it for you, as is discussed next.

To see how the k-means algorithm is used, we can see how it is used on the iris dataset:

```
import kmeansnet
net = kmeansnet.kmeans(3,train)
net.kmeanstrain(train)
cluster = net.kmeansfwd(test)
print cluster
print iris[3::4,4]
```

The output that is produced by this in an example run is (where the top line is the output of the algorithm and the bottom line is the classes from the dataset):

```
[ 0. 0. 0. 0. 0. 1. 1. 1. 1. 2. 1. 2. 2. 2. 0. 1. 2. 1. 0.
  1. 2. 2. 2. 1. 1. 2. 0. 0. 1. 0. 0. 0. 0. 2. 0. 2. 1.]
[ 1. 1. 1. 1. 1. 2. 2. 2. 1. 0. 2. 0. 0. 0. 1. 1. 0. 2. 2.
  2. 0. 0. 0. 2. 2. 0. 1. 2. 1. 1. 1. 1. 1. 0. 1. 0. 2.]
```

and then we can see that cluster 0 corresponds to label 1 and cluster 1 to label 2, in which case the algorithm gets 1 of cluster 0 wrong, 2 of cluster 1, and none of cluster 2.

9.1.6 Using Competitive Learning for Clustering

Deciding which cluster any datapoint belongs to is now an easy task: we present the datapoint to the trained algorithm and it will tell us. If we don't have any target data, then the problem is finished. However, for many problems we might want to interpret the best-matching cluster as a class label (alternatively, a set of cluster centres could all correspond to one class). This is fine, since if we have target data we can match the output classes to the targets, provided that we are a bit careful: there is no reason why the order of the nodes in the network should match the order in the data, since the algorithm knows nothing about that order. For that reason, when assigning class labels to the outputs, you need to check which numbers match up carefully, or the results will look a lot worse than they actually are.

There is an alternative solution to this problem of assigning labels, and it is one that we have seen before. In Chapter 4 we considered using the k-means

network in order to train the positions of the RBF nodes. It is now possible to see how this works. The k-means part positions the RBFs in the input space, so that they represent the input data well. A Perceptron is then used on top of this in order to provide the match to the outputs in the supervised learning part of the network. Since this is now supervised learning, it ensures that the output categories match the target data classes. It also means that you can use lots of clusters in the k-means network without having to work out which datapoints belong to which cluster, since the Perceptron will do this for you.

We are now going to look at another major algorithm in competitive learning, the Self-Organising Feature Map. As motivation for it, we are going to consider a sample problem for competitive learning, which is a problem in data compression called vector quantisation.

9.2 Vector Quantisation

We've already discussed using competitive learning for removing noise. There is a related application, data compression, which is used both for storing data and for the transmission of speech and image data. The reason that the applications are related is that both replace the current input by the cluster centre that it belongs to. For noise reduction we do this to replace the noisy input with a cleaner one, while for data compression we do it to reduce the number of datapoints that we send.

Both of these things can be understood by considering them as examples of data communication. Suppose that I want to send data to you, but that I have to pay for each data bit I transmit, so I want to keep the amount of data that I send to a minimum. I notice that there are lots of repeated datapoints, so I decide to encode my data before I send it, so that instead of sending the entire set, we agree a codebook of prototype vectors together. Now, instead of transmitting the actual data, I can transmit the index of that datapoint in the codebook, which is shorter. All you have to do is take the indices I send you and look them up, and you have the data. We can actually make the code even more efficient by using shorter indices for the datapoints that are more common. This is an important problem in information theory, and every kind of sound and image compression algorithm has a different method of solving it.

There is one problem with the scenario so far, which is that the codebook won't contain every possible datapoint. What happens when I want to send a datapoint and it isn't in the codebook? In that case we need to accept that our data will not look exactly the same, and I send you the index of the prototype vector that is closest to it (this is known as vector quantisation, and is the way that lossy compression works).

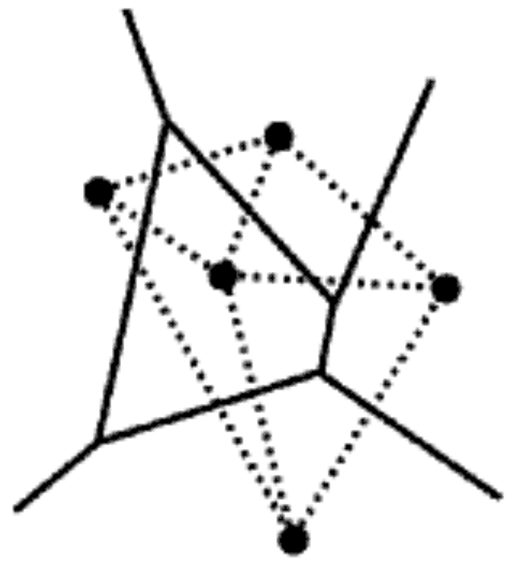


FIGURE 9.5: The Voronoi tesselation of space that performs vector quantisation. Any datapoint is represented by the dot within its cell, which is the prototype vector.

Figure 9.5 shows an interpretation of prototype vectors in two dimensions. The dots at the centre of each cell are the prototype vectors, and any datapoint that lies within a cell is represented by the dot. The name for each cell is the Voronoi set of a particular prototype. Together, they produce the Voronoi tesselation of the space. If you connect together every pair of points that share an edge, as is shown by the dotted lines, then you get the Delaunay triangulation, which is the optimal way to organise the space to perform function approximation.

The question is how to choose the prototype vectors, and this is where competitive learning comes in. We need to choose prototype vectors that are as close as possible to all of the possible inputs that we might see. This application is called learning vector quantisation because we are learning an efficient vector quantisation. The k-means algorithm can be used to solve the problem if we know how large we want our codebook to be. However, another algorithm turns out to be more useful, the Self-Organising Feature Map, which is described next.

9.3 The Self-Organising Feature Map

By far the most commonly used competitive learning algorithm is the Self-Organising Feature Map (often abbreviated to SOM), which was proposed by Teuvo Kohonen in 1988. Kohonen was considering the question of how sensory signals get mapped into the cerebral cortex of the brain with an order. For example, in the auditory cortex, which deals with the sounds that we hear, neurons that are excited (i.e., that are caused to fire) by similar sounds are positioned closely together, whereas two neurons that are excited by very different sounds will be far apart.

There are two novel departures in this for us: firstly, the relative locations

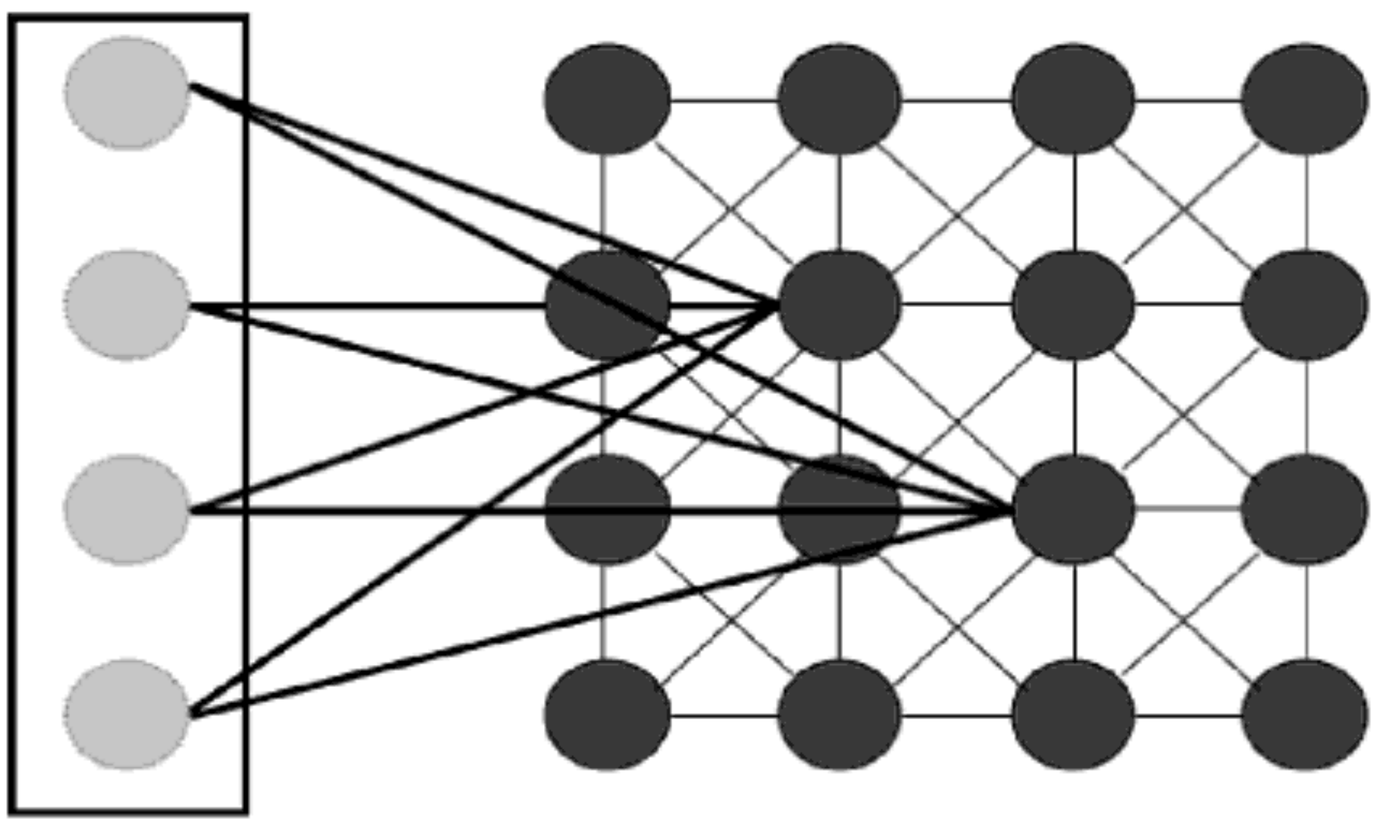


FIGURE 9.6: The Self-Organising Map network. As usual, input nodes (on the left) do no computation, and the weights are modified to change the activations of the neurons (weights are only shown to two nodes for clarity). However, the nodes within the SOM affect each other in that the winning node also changes the weights of neurons that are close to it. Connections are shown in the figure to the eight closest nodes, but this is a parameter of the network.

of the neurons in the network matters (this property is known as feature mapping—nearby neurons correspond to similar input patterns), and secondly, the neurons are arranged in a grid with connections between the neurons, rather than in layers with connections only between the different layers. In the auditory cortex there appears to be sheets of neurons arranged in 2D, and that is the typical arrangement of neurons for the SOM: a grid of neurons arranged in 2D, as can be seen in Figure 9.6. A 1D line of neurons is also sometimes used. In mathematical terms, the SOM demonstrates relative ordering preservation, which is sometimes known as topology preservation. The relative ordering of the inputs should be preserved by the ordering in the neurons, so that neurons that are close together represent inputs that are close together, while neurons that are far apart represent inputs that are far apart.

This topology preservation is not necessarily possible, because the SOM typically uses a 1D or 2D array of neurons, and most of our input spaces are of much higher dimensionality than that. This means that the ordering cannot be preserved. We have seen this in Figure 1.2, where one view of some wind turbines made it look like they are on top of each other, when they clearly are not, because we used a two-dimensional representation of three-dimensional reality. You've probably seen the same thing in other photos, where trees appear to be growing out of somebody's head. A different way to see the same thing is given in Figure 9.7, where mismatches between the topology of the input space and map lead to changes in the relative ordering. The best that can be said is that SOM is perfectly topology-preserving, which means that if the dimensionality of the input and the map correspond, then

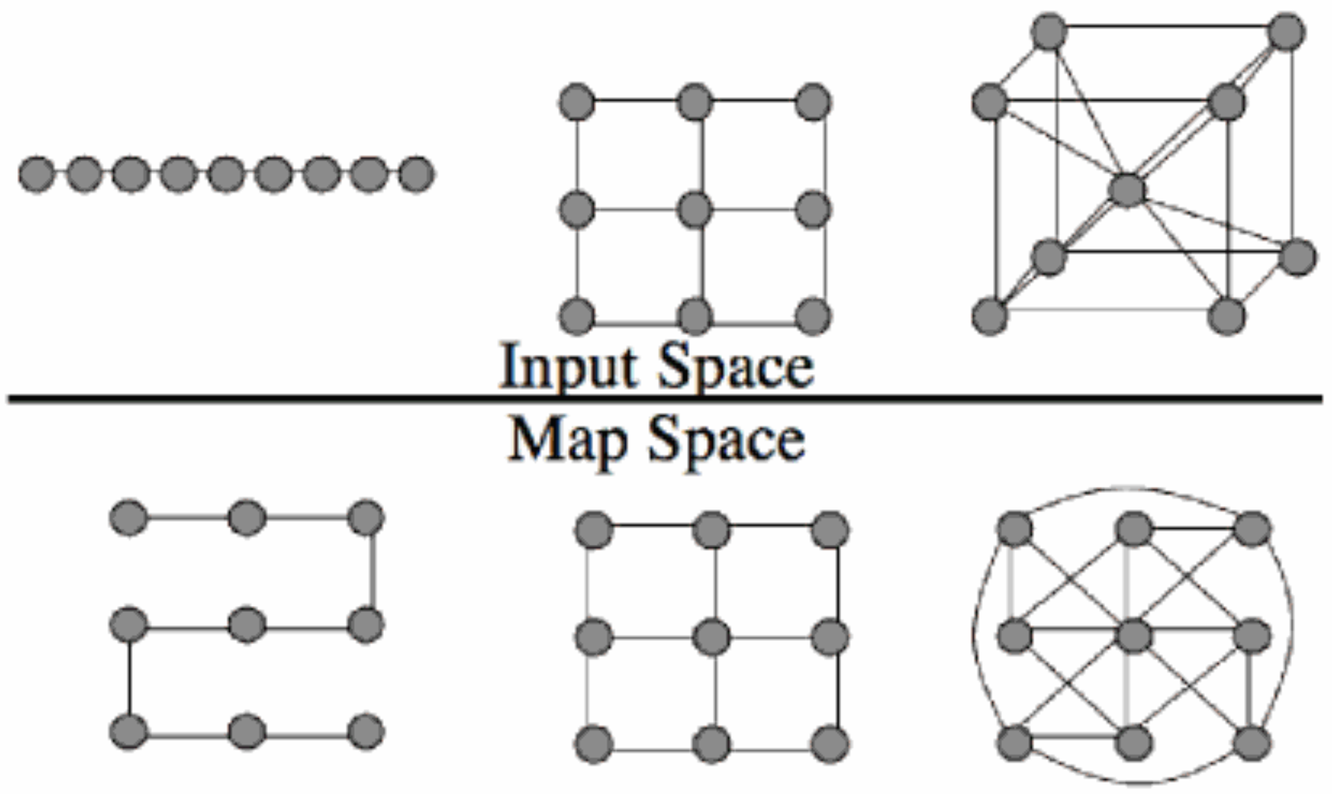


FIGURE 9.7: When inputs in 1D (a straight line), a 2D grid, and a 3D cube are represented by a 2D grid of neurons, the relative ordering is not perfectly preserved. The 1D line is bent, which means that points that used to be a long way apart (such as the first and sixth on the line) are now close together, while the cube becomes very complicated. The lines in the bottom part of the figure represent connections that are meant to be close.

the topology of the input space will be preserved. We are going to look at other methods of performing dimensionality reduction in Chapter 10.

The question, then, is how we can implement feature mapping in an unsupervised learning algorithm. The first thing to recognise is that we need some interaction between the neurons in the network, so that when one neuron fires, it affects what happens to those around it. We have seen something like this before, for example, between different layers of the MLP, but now we are thinking about neurons that are within a layer. These are known as lateral connections (i.e., within the layer of the network). How should this interaction work? We are trying to introduce feature mapping, so neurons that are close together in the map should represent similar features. This means that the winning neuron should pull other neurons that are close to it in the network closer to itself in weight space, which means that we need positive connections. Likewise, neurons that are further away should represent different features, and so should be a long way off in weight space, so the winning neuron 'repels' them, by using negative connections to push them away. Neurons that are very far away in the network should already represent different features, so we just ignore them. This is known as the 'Mexican Hat' form of lateral connections, for reasons that should be clear from the picture in Figure 9.8. We can then just use ordinary competitive learning, just like we did for the k-means network in Section 9.1.2. The Self-Organising Map does pretty much exactly this.

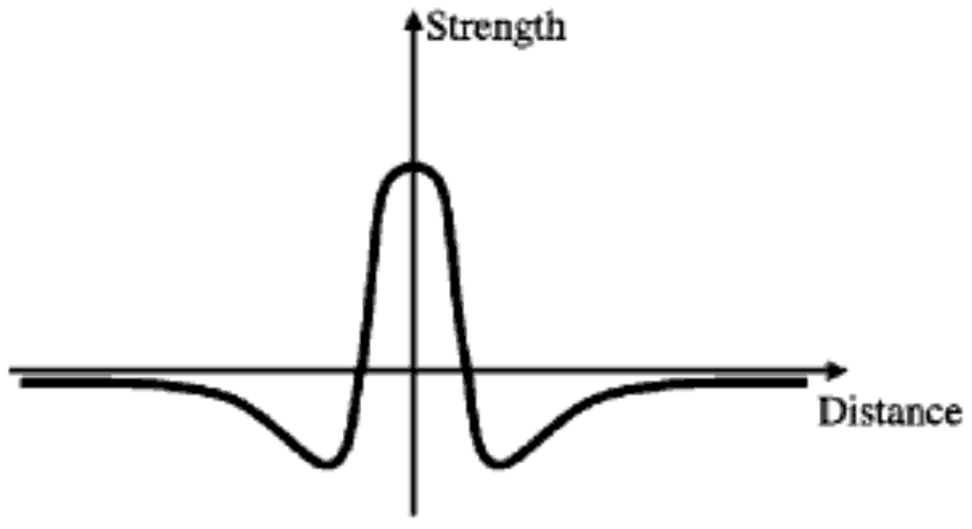


FIGURE 9.8: Graph of the strength of lateral connections for a feature mapping algorithm known as the 'Mexican Hat.'

9.3.1 The SOM Algorithm

Using the full Mexican hat lateral interactions between neurons is fine, but it isn't essential. In Kohonen's SOM algorithm, the weight update rule is modified instead, so that information about neighbouring neurons is included in the learning rule, which makes the algorithm simpler. The algorithm is a competitive learning algorithm, so that one neuron is chosen as the winner, but when its weights are updated, so are those of its neighbours, although to a lesser extent. Neurons that are not within the neighbourhood are ignored, not repelled.

We will now look at the SOM algorithm before examining some of the details further.

The Self-Organising Feature Map Algorithm

- **Initialisation**
  - choose a size (number of neurons) and number of dimensions d for the map
  - Either:
    - choose random values for the weight vectors so that they are all different OR
    - set the weight values to increase in the direction of the first d principal components of the dataset
- **Learning**
  - repeat:
    - for each datapoint:
      - select the best-matching neuron n_b using the minimum Euclidean distance between the weights and the input,

$$n_b = \min_j \|\mathbf{x} - \mathbf{w}_j^T\|. \tag{9.8}$$

* update the weight vector of the best-matching node using:

$$\mathbf{w}_j^T \leftarrow \mathbf{w}_j^T + \eta(t)(\mathbf{x} - \mathbf{w}_j^T), \tag{9.9}$$

where $\eta(t)$ is the learning rate.

* update the weight vector of all other neurons using:

$$\mathbf{w}_j^T \leftarrow \mathbf{w}_j^T + \eta_n(t)h(n_b, t)(\mathbf{x} - \mathbf{w}_j^T), \tag{9.10}$$

where $\eta_n(t)$ is the learning rate for neighbourhood nodes, and $h(n_b, t)$ is the neighbourhood function, which decides whether each neuron should be included in the neighbourhood of the winning neuron (so $h = 1$ for neighbours and $h = 0$ for non-neighbours)

* reduce the learning rates and adjust the neighbourhood function, typically by $\eta(t+1) = \alpha\eta(t)^{k/k_{\max}}$ where $0 \leq \alpha \leq 1$ decides how fast the size decreases, k is the number of iterations the algorithm has been running for, and $k_{\max}$ is when you want the learning to stop. The same equation is used for both learning rates (η, η_n) and the neighbourhood function $h(n_b, t)$.

– until the map stops changing or some maximum number of iterations is exceeded

• **Usage**

– for each test point:

* select the best-matching neuron n_b using the minimum Euclidean distance between the weights and the input:

$$n_b = \min_j \|\mathbf{x} - \mathbf{w}_j^T\| \tag{9.11}$$

9.3.2 Neighbourhood Connections

The size of the neighbourhood is thus another parameter that we need to control. How large should the neighbourhood of a neuron be? If we start our network off with random weights, as we did for the MLP, then at the beginning of learning, the network is pretty well unordered (as the weights are random, two nodes that are very close in weight space could be on opposite sides of the map, and vice versa) and so it makes sense that the neighbourhoods should be large, so that we get the rough ordering of the network correct. However, once the network has been learning for a while, the rough ordering has already been created, and the algorithm starts to fine-tune the individual local regions of the network. At this stage, the neighbourhoods should be small, as is shown in

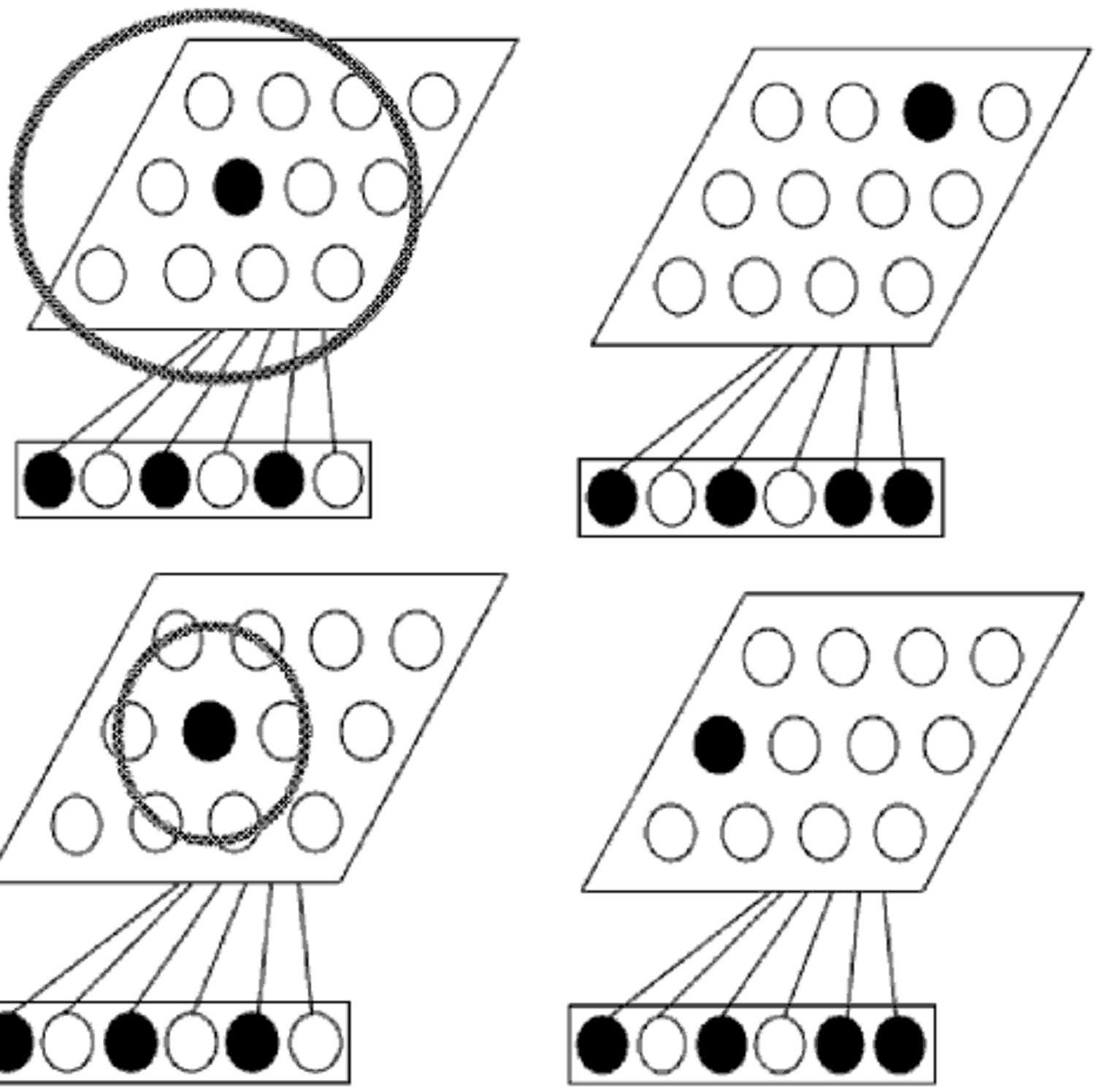


FIGURE 9.9: *Top:* Initially, similar input vectors excite neurons that are far apart, so that the neighbourhood (shown as a circle) needs to be large. *Bottom:* Later on during training the neighbourhood can be smaller, because similar input vectors excite neurons that are close together.

Figure 9.9. It therefore makes sense to reduce the size of the neighbourhood as the network adapts. These two phases of learning are also known as ordering and convergence. Typically, we reduce the neighbourhood size by a small amount at each iteration of the algorithm. We control the learning rate η in exactly the same way, so that it starts off large and decreases over time, as is shown in the algorithm below.

The fact that the size of the neighbourhood changes as the algorithm runs has consequences for an implementation. There is no point using actual connections between nodes, since the number of these will change as the algorithm runs. We therefore set up a matrix that measures the distances between nodes in the network and choose the nodes in the neighbourhood of a particular node as those within a neighbourhood radius that shrinks as the algorithm runs.

```
# Set up the map distance matrix
mapDist = zeros((self.x*self.y,self.x*self.y))
for i in range(self.x*self.y):
    for j in range(i+1,self.x*self.y):
```

Hidden page

9.3.3 Self-Organisation

You might be wondering what the self-organisation in the name of the SOM is. A particularly interesting aspect of feature mapping is that we get a global ordering of the neurons in the network, despite the fact that the interactions are all local, since neurons that are very far apart do not interact with each other. We thus get a global ordering of the space using only a set of local interactions, which is amazing. This is known as self-organisation, and it appears everywhere. It is part of the growing science of complexity. To see how common self-organisation is, consider a flock of birds flying in formation. The birds cannot possibly know exactly where each other are, so how do they keep in formation? In fact, simulations have shown that if each bird just tries to stay diagonally behind the bird to its right, and fly at the same speed, then they form perfect flocks, no matter how they start off and what objects are placed in their way. So the global ordering of the whole flock can arise from the local interactions of each bird looking to the one on its right (or left).

9.3.4 Network Dimensionality and Boundary Conditions

We typically think about applying the SOM algorithm to a 2D rectangular array of neurons (as shown in Figure 9.6), but there is nothing in the algorithm to force this. There are cases where a line of neurons (1D) works better, or where three dimensions are needed. It depends on the dimensionality of the inputs (actually on the intrinsic dimensionality, the number of dimensions that you actually need to represent the data), not the number that it is embedded in. As an example, consider a set of inputs spread through the room you are in, but all on the plane that connects the bottom of the wall to your left with the top of the wall to your right. These points have intrinsic dimensionality two since they are all on the plane, but they are embedded in your three-dimensional room. Noise and other inaccuracies in data often leads to it being represented in more dimensions than are actually required, and so finding the intrinsic dimensionality can help to reduce the noise.

We also need to consider the boundaries of the network. In some cases, it makes sense that the edges of the map of neurons is strictly defined — for example, if we are arranging sounds from low pitch to high pitch, then the lowest and highest pitches we can hear are obvious end points. However, it is not always the case that such boundaries are clearly defined. In this case we might want to remove the boundary conditions. We can do this by removing the boundary by tying the ends together. In 1D this means that we turn a line into a circle, while in 2D we turn a rectangle into a torus. To see this, try taking a piece of paper and bend it so that the top and bottom edges line up. You've now got a tube. If you bend the tube round so that the two open ends meet up you have a circle of tube known as a torus. Pictures of these effects are shown in Figure 9.10. In effect, it means that there are no neurons on the edge of the feature map. The choice of the number of dimensions and the

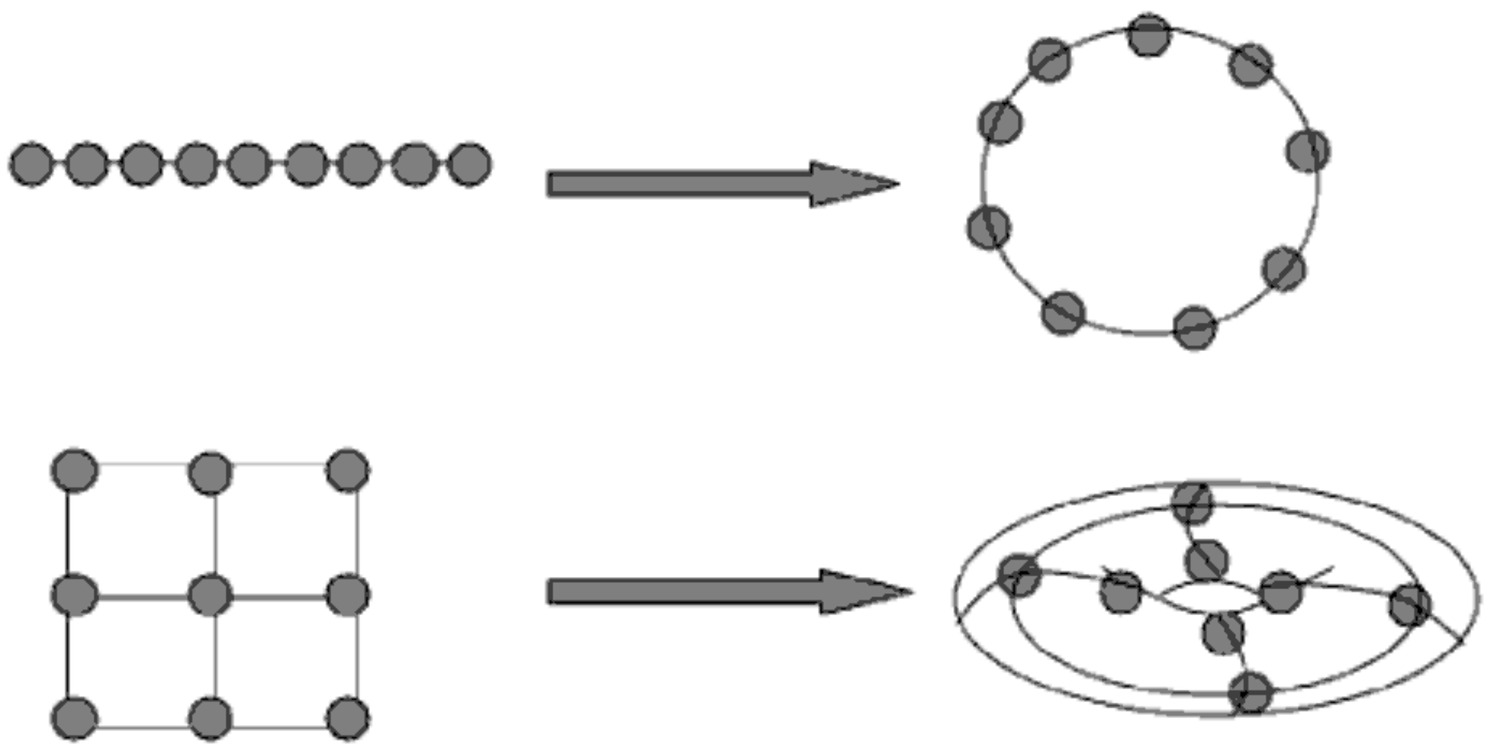


FIGURE 9.10: Using circular boundary conditions in 1D turns a line into a circle, while in 2D it turns a rectangle into a torus.

boundary conditions depends on the problem that we are considering, but it is usually the case that the torus works better than the rectangle, although it is not always clear why.

The one cost that this has is that the map distances get more complicated to calculate, since we now need to calculate the distances allowing for the wrap around. This can be done using modulo arithmetic, but it is easier to think about taking copies of the map and putting them around the map, so that the original map has copies of itself all around: one above, one below, to the right and left, and also diagonally above and below, as is shown in Figure 9.11. Now we keep one of the points in the original map, and the distance to the second node is the smallest of the distances between the first node and the copies of the second node in the different maps (including the original). By treating the distances in x and y separately, the number of distances that has to be computed can be reduced.

As with the competitive learning algorithm that we considered earlier, the size of the SOM is defined before we start learning. The size of the network (that is, the number of neurons that we put into it) decides how fine-grained the learning is. If there are very few neurons, then the best that the network can do is to find gross generalisations that link the data. However, if there are very large numbers of neurons, then the network can represent every input without ever needing to generalise at all. This is yet another example of overfitting. Clearly, then, choosing the correct size of network is important. The common approach is to test out several different sizes of network, such as 5×5 and 10×10 and see how well the network learns.

9.3.5 Examples of Using the SOM

As a first example of using the SOM, and one that shows the topological ordering of the network, consider training the network on a set of two-

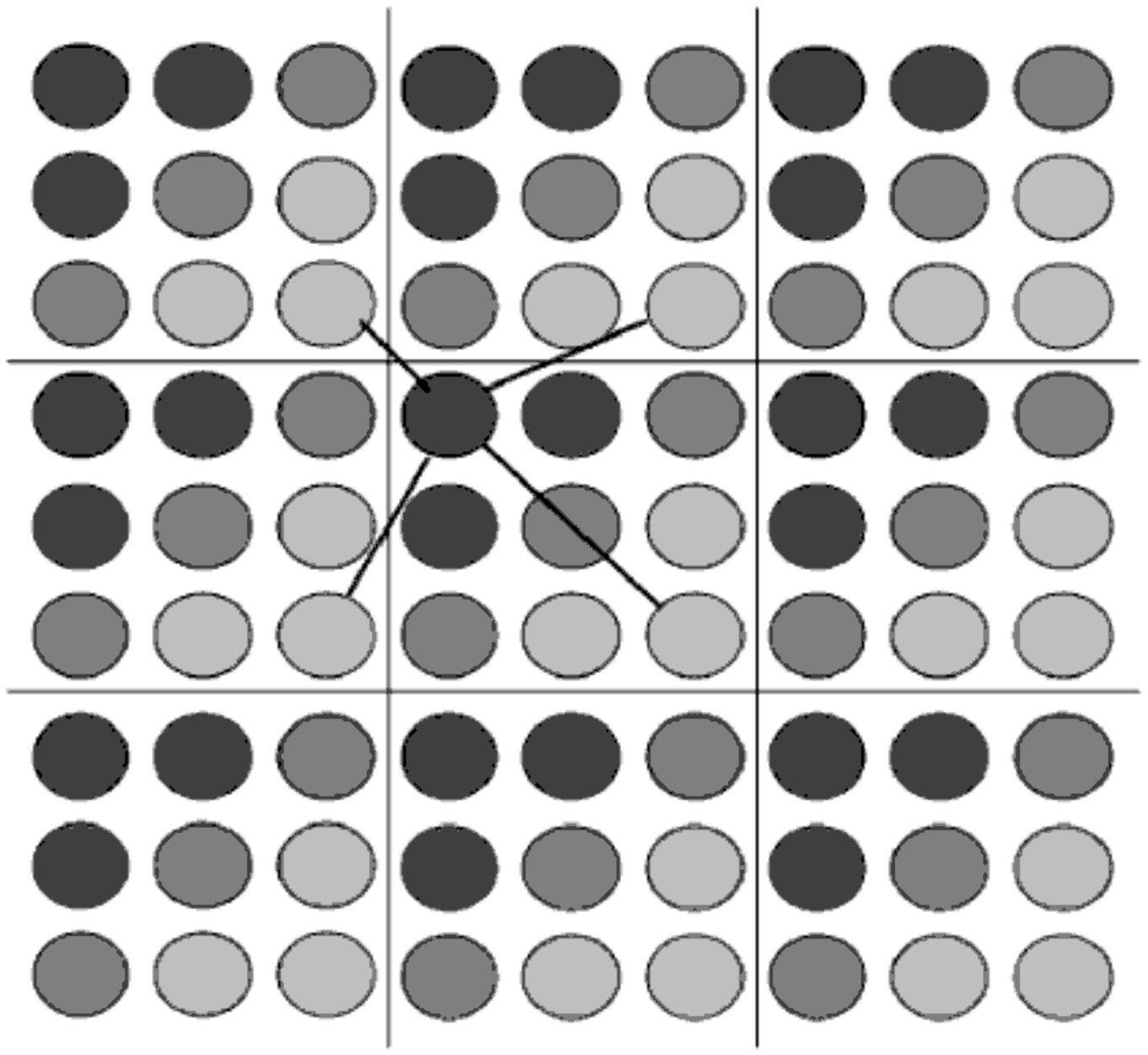


FIGURE 9.11: One way to compute distances between points without any boundary on the map is to imagine copies of the entire map being placed around the original, and picking the shortest of the distances between a node and any of the copies of the other node.

dimensional data drawn at random from a uniform distribution in $[-1, 1]$ in both directions. If the network weights are started off randomly, then initially the network is completely disordered (as shown in the top-left picture in Figure 9.12), but after 10 iterations of training the network is ordered so that neighbouring nodes map to data that is close together (bottom-left). Using PCA to initialise the map is not especially useful for this dataset, but it does speed things up: only five iterations through the dataset produce the output shown on the bottom-right of the figure, where it started from the version on the top-right.

For two examples of using the SOM on non-random data, where we can expect to see some actual learning, we will first look at the iris data that we used with the k-means algorithm earlier in this chapter. Figure 9.13 shows a plot of which node of a 5×5 Self-Organising Map was the best match on a set of test data after training for 100 iterations. The three different classes are shown as different shapes (squares, plus triangles pointing up and down), but remember that the network did not receive any information about these target classes. It can be seen that the examples in each of the three classes form different clusters in the map. Looking at the figure, you might be wondering if it is possible to use the plot to identify the different classes by assuming that they are separated in the map. This has been investigated—often by using methods similar to those of Linear Discriminant Analysis that are described

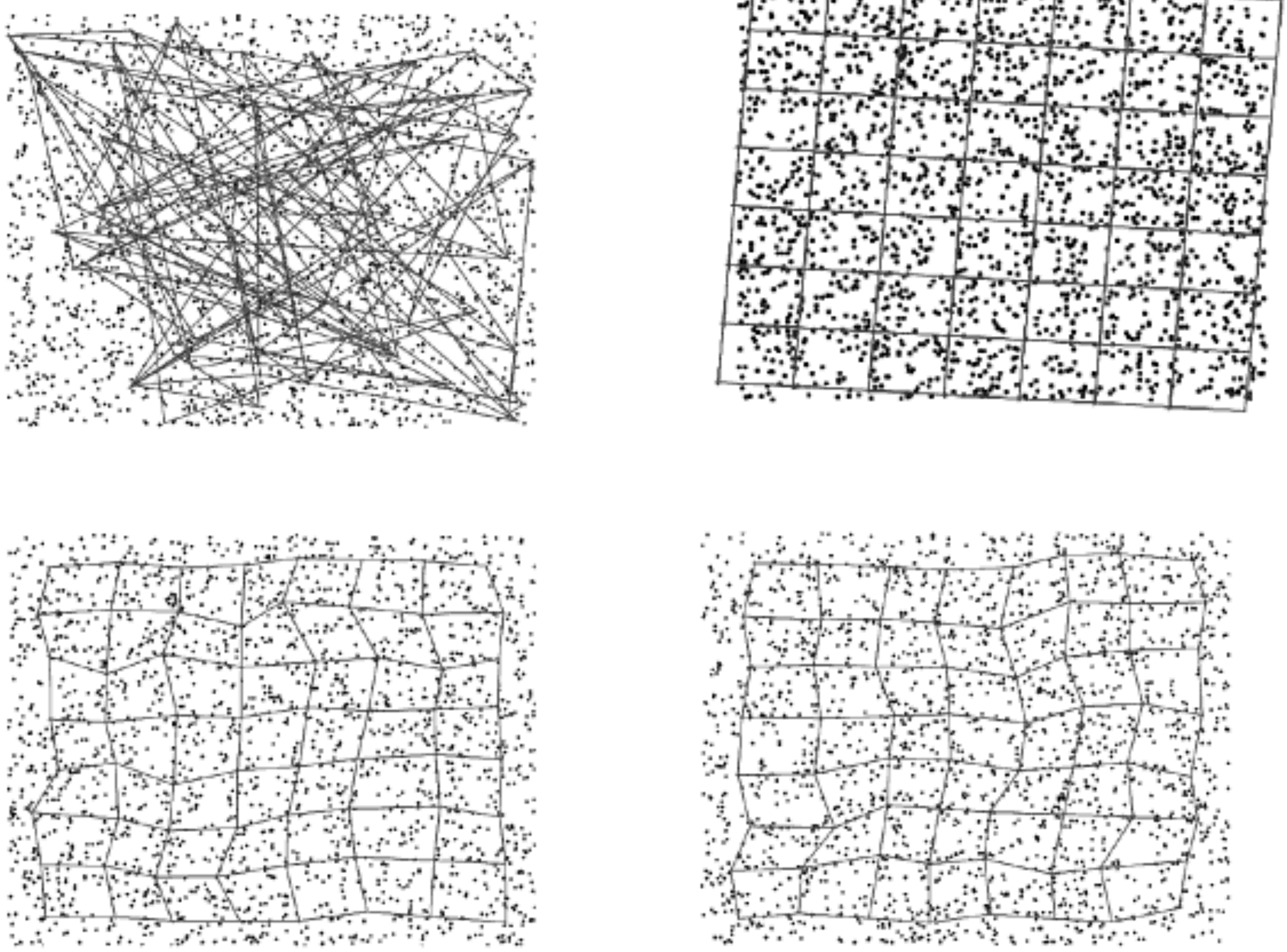


FIGURE 9.12: Training the SOM on a set of uniformly randomly sampled two-dimensional data in the range $[-1, 1]$ in both dimensions. *Top:* Initialisation of the map using *left:* random weights and *right:* PCA (the randomness in the data means that the directions of variation are not necessarily along the obvious directions). *Bottom:* The output after just 10 iterations of training on the left, and 5 on the right, both with typical parameter values.

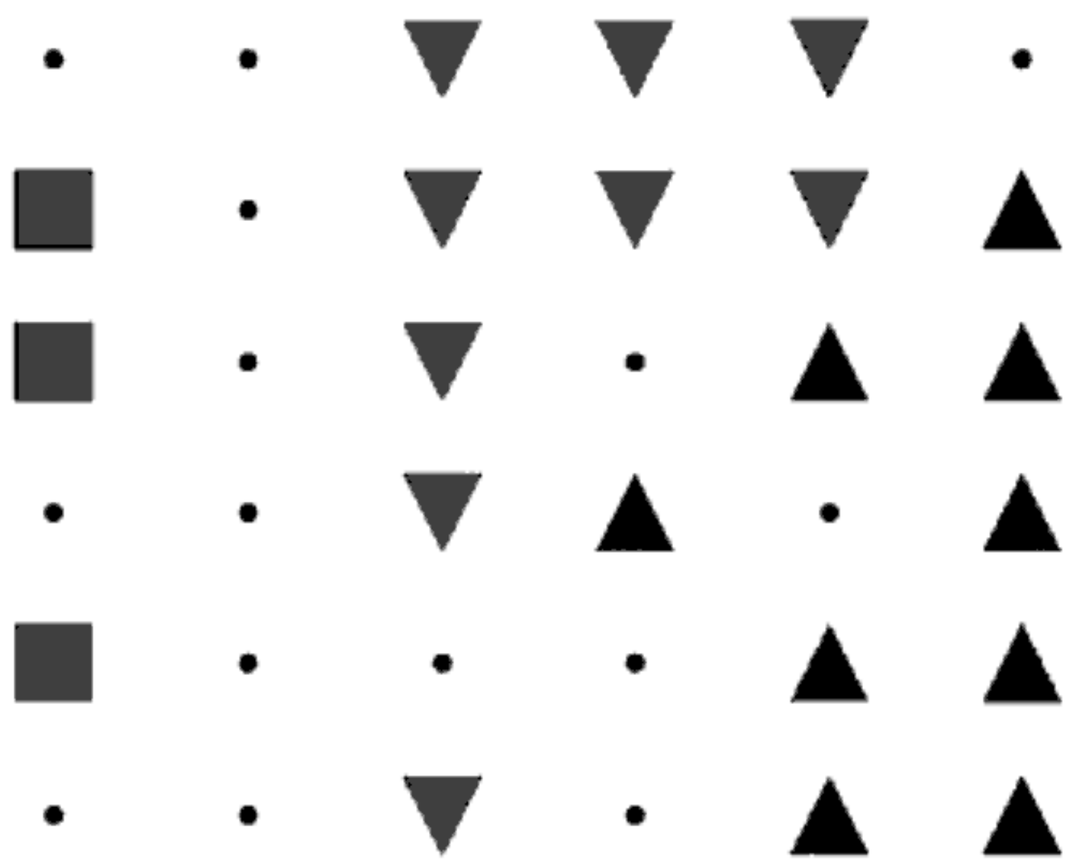


FIGURE 9.13: Plot showing which nodes are the best match according to class, with the three shapes corresponding to three different classes in the iris dataset. The small dots represent nodes that did not fire.

in Section 10.1—with some success, and a reference is provided at the end of the chapter.

A more difficult problem is shown in Figure 9.14. The data are the `ecoli` dataset from the UCI Machine Learning repository, and the class is the localisation site of the protein, based on a set of protein measurements. The results with this dataset when testing are not as clearly impressive (but note that the MLP gets about 50% accuracy on this dataset, and that has the target data, which the SOM doesn't). However, the clusters can still be seen to some extent, and they are very clear in the training data. Note that the boundary conditions can make things a little more complicated, since the cluster does not necessarily respect the edges of the map.

Further Reading

There is a book by Kohonen, the inventor of the SOM, that provides a very good overview of the area:

- T. Kohonen. *Self-Organisation and Associative Memory.* Springer, Berlin, Germany, 3rd edition, 1989.

The two on-line self-organising networks that were mentioned in the chapter were:

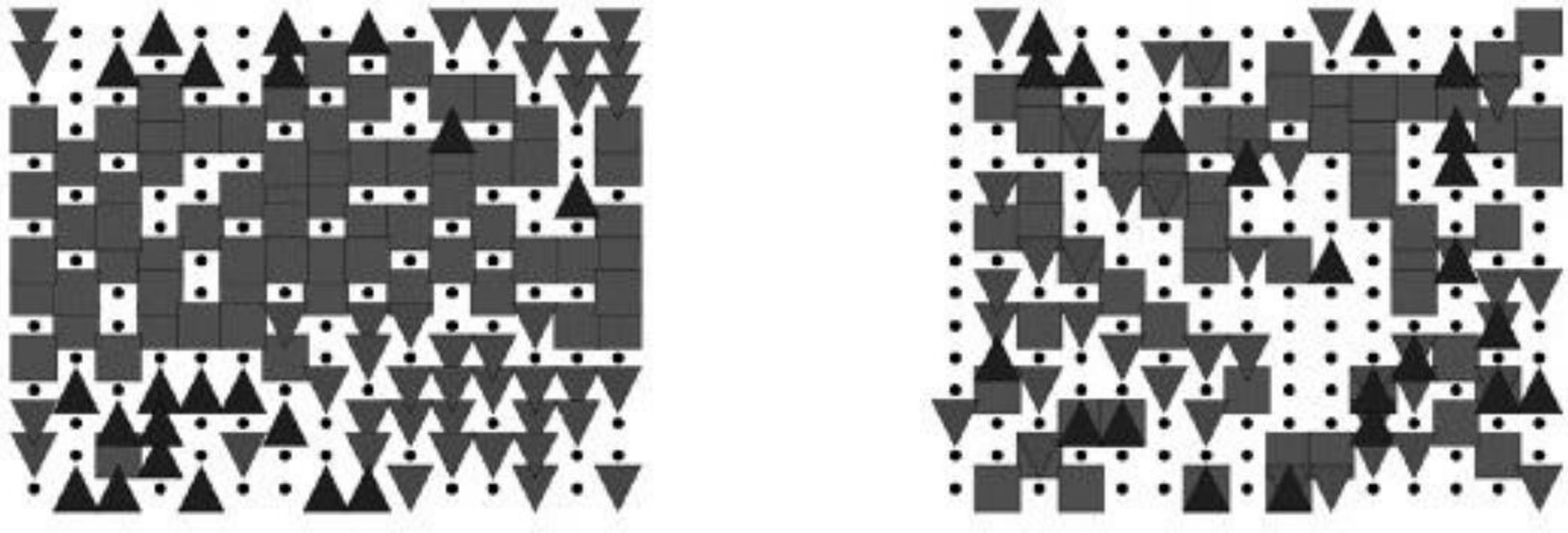


FIGURE 9.14: Plots showing which nodes are the best match according to class, with the three shapes corresponding to three different classes in the e-coli dataset, tested on *left:* the training set and *right:* a separate test set. The small dots represent nodes that did not fire.

- B. Fritzke. A growing neural gas network learns topologies. In Gerald Tesauro, David S. Touretzky, and Todd K. Leen, editors, *Advances in Neural Information Processing Systems*, volume 7, MIT Press, Cambridge, MA, USA, 1995.

- S. Marsland, J.S. Shapiro, and U. Nehmzow. A self-organising network that grows when required. *Neural Networks*, 15(8-9):1041–1058, 2002.

A possible reference on processing the data in the map in order to identify clusters is:

- S. Wu and T.W.S. Chow. Self-organizing-map based clustering using a local clustering validity index. *Neural Processing Letters*, 17(3):253–271, 2003.

Books that cover the area include:

- Section 10.14 of R.O. Duda, P.E. Hart, and D.G. Stork. *Pattern Classification.* Wiley-Interscience, New York, USA, 2nd edition, 2001.

- Chapter 9 of S. Haykin. *Neural Networks: A Comprehensive Foundation.* Prentice-Hall, New Jersey, USA, 2nd edition, 1999.

- Section 9.3 of B.D. Ripley. *Pattern Recognition and Neural Networks.* Cambridge University Press, Cambridge, UK, 1996.

Practice Questions

Problem 9.1 What is the purpose of the neighbourhood function in the SOM? How does it change the learning?

Problem 9.2 A simplistic intruder detection system for a computer network consists of an attempt to categorise users according to (i) the time of day they log in, (ii) the length of time they log in for, (iii) the types of programs they run while logged in, (iv) the number of programs they run while logged in. Suggest how you would train a SOM and the naïve Bayes' classifier to perform the categorisation. What preprocessing of the data would you do, how much data would you need, and how large would you make the SOM? Do you think that such a system would work for intruder detection?

Problem 9.3 The Music Genome Project (`http://www.pandora.com`) does not work by using a SOM. But it could. Describe how you would implement it.

Problem 9.4 A bank wants to detect fraudulent credit card transactions. They have data for lots and lots of transactions (each transaction is an amount of money, a shop, and the time and date) and some information about when credit cards were stolen, and the transactions that were performed on the stolen card. Describe how you could use a competitive learning method to cluster people's transactions together to identify patterns, so that stolen cards can be detected as changes in pattern. How well do you think this would work? There is much more data of transactions when cards are not stolen, compared to stolen transactions. How does this affect the learning, and what can you do about it?

Problem 9.5 It is possible to use any competitive learning method to position the basis functions of a Radial Basis Function network. The example code used k-means. Modify it to use the SOM instead and compare the results on the `wine` and `yeast` datasets.

Problem 9.6 For the `wine` dataset, experiment with different sizes of map, and boundary conditions. How much difference does it make? Can you use the principal components in order to set the size automatically?

Chapter 10

Dimensionality Reduction

In Chapter 9 we saw that the Self-Organising Map (SOM) reduced the number of dimensions in the data to the two dimensions of the map. The choice of two dimensions was imposed arbitrarily by the fact that the neurons were arranged in a two-dimensional grid, and we saw in Section 9.3 that this can cause problems, since projecting the data into two dimensions usually changes the relative ordering of the datapoints. However, there are many reasons why this dimensionality reduction is useful. The most obvious justification is that it reduces the curse of dimensionality, and also the computational cost of many of the algorithms, since the dimensionality is usually an explicit factor. 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, where the number of dimensions becomes three or less, 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 why 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

x	y
2.00	-1.43
2.37	-2.80
1.00	-3.17
0.63	-1.80

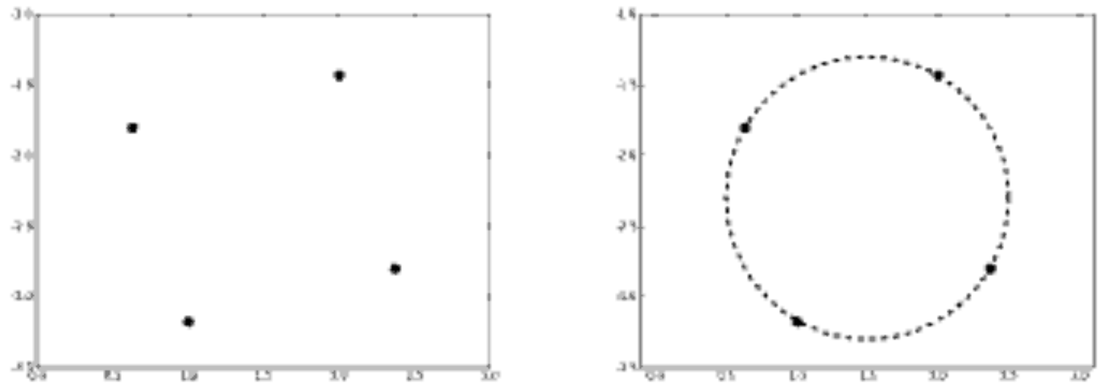


FIGURE 10.1: Three views of the same four points. *Left:* As numbers, where the links are unclear. *Centre:* As four plotted points. *Right:* As four points that lie on a circle.

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 10.1. 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.

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 have already seen a method of doing feature selection, since it is inherent to the way that the decision tree (Chapter 6) 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 the pruning that was applied to 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 11.4) are employed, although backtracking can also be employed to check whether the search gets stuck.

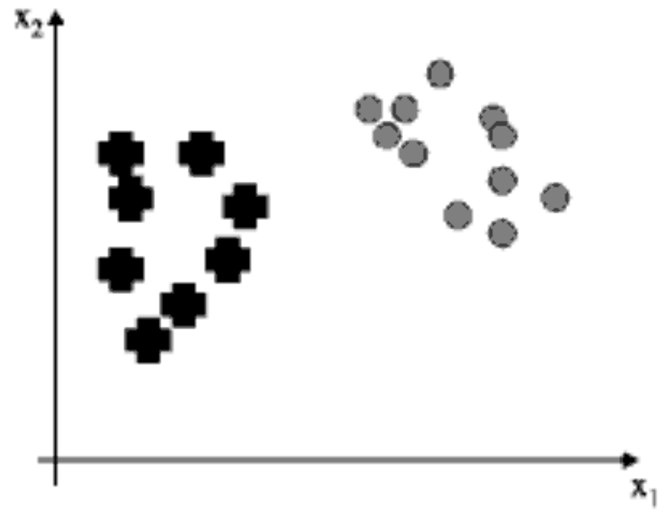


FIGURE 10.2: A set of datapoints in two dimensions, with two classes.

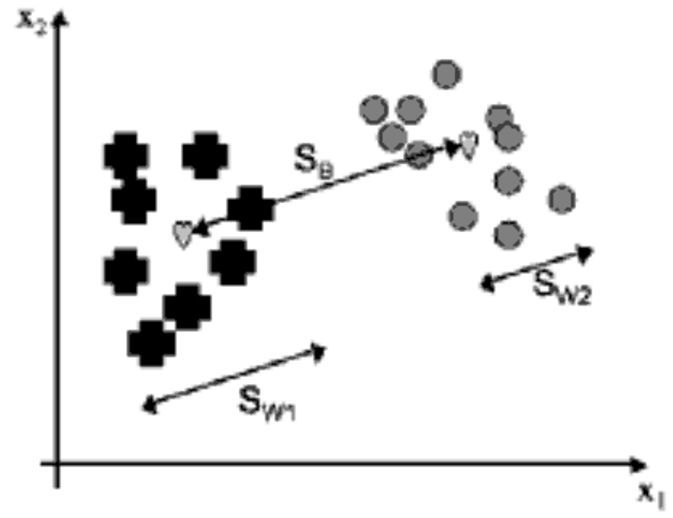


FIGURE 10.3: The mean-ing of the between-class and within-class scatter. The hearts mark the means of the two classes.

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.

10.1 Linear Discriminant Analysis (LDA)

Figure 10.2 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 8.2.2 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:

$$S_W = \sum_{\text{classes } c} \sum_{j \in c} p_c (\mathbf{x}_j - \boldsymbol{\mu}_c)(\mathbf{x}_j - \boldsymbol{\mu}_c)^T. \tag{10.1}$$

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:

$$S_B = \sum_{\text{classes } c} (\boldsymbol{\mu}_c - \boldsymbol{\mu})(\boldsymbol{\mu}_c - \boldsymbol{\mu})^T. \tag{10.2}$$

The meanings of these two measurements is shown in Figure 10.3. 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 10.4 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 2 that any line can be written as a vector $\mathbf{w}$ (which we used as our weight vector in Section 2.3; 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 10.4. 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. Replacing $\mathbf{x}_j$ with $\mathbf{w}^T \cdot \mathbf{x}_j$ in Equations (10.1) and (10.2) 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:

$$\sum_{\text{classes } c} \sum_{j \in c} p_c(\mathbf{w}^T \cdot (\mathbf{x}_j - \boldsymbol{\mu}_c))(\mathbf{w}^T \cdot (\mathbf{x}_j - \boldsymbol{\mu}_c))^T = \mathbf{w}^T S_W \mathbf{w} \tag{10.3}$$

$$\sum_{\text{classes } c} \mathbf{w}^T(\boldsymbol{\mu}_c - \boldsymbol{\mu})(\boldsymbol{\mu}_c - \boldsymbol{\mu})^T\mathbf{w} = \mathbf{w}^T S_B \mathbf{w}. \tag{10.4}$$

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:

Hidden page

Hidden page

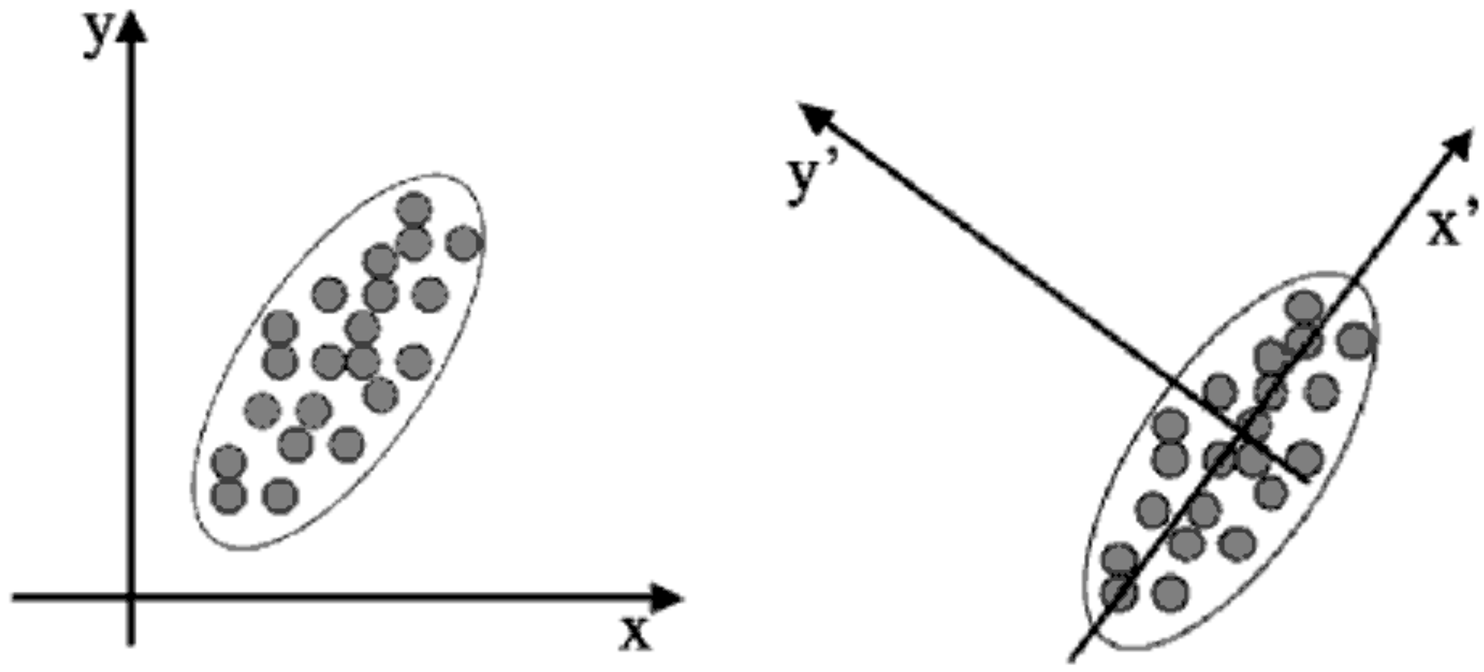


FIGURE 10.6: Two different sets of coordinate axes. The second consists of a rotation and translation of the first and was found using Principal Components Analysis.

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 10.6, 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 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 it 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.,

$$\text{cov}(\mathbf{Y}) = \text{cov}(\mathbf{P}^T\mathbf{X}) = \begin{pmatrix} \lambda_1 & 0 & 0 & \dots & 0 \\ 0 & \lambda_2 & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & \dots & \lambda_N \end{pmatrix} . \tag{10.7}$$

We can get a different handle on this by using some linear algebra and the definition of covariance to see that:

$$\begin{aligned} \text{cov}(\mathbf{Y}) &= E[\mathbf{Y}\mathbf{Y}^T] && (10.8)\\ &= E[(\mathbf{P}^T\mathbf{X})(\mathbf{P}^T\mathbf{X})^T] && (10.9)\\ &= E[(\mathbf{P}^T\mathbf{X})(\mathbf{X}^T\mathbf{P})] && (10.10)\\ &= \mathbf{P}^T E(\mathbf{X}\mathbf{X}^T)\mathbf{P} && (10.11)\\ &= \mathbf{P}^T \text{cov}(\mathbf{X})\mathbf{P}. && (10.12) \end{aligned}$$

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:

$$\mathbf{P}\text{cov}(\mathbf{Y}) = \mathbf{P}\mathbf{P}^T\text{cov}(\mathbf{X})\mathbf{P} = \text{cov}(\mathbf{X})\mathbf{P}, \tag{10.13}$$

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:

$$\mathbf{P}\text{cov}(\mathbf{Y}) = [\lambda_1\mathbf{p}_1, \lambda_2\mathbf{p}_2, \dots, \lambda_N\mathbf{p}_N], \tag{10.14}$$

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:

$$\boldsymbol{\lambda}\mathbf{P} = \mathbf{Z}\mathbf{P}. \tag{10.15}$$

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. Since $\boldsymbol{\lambda}$ is only a column vector, all it does is rescale the components of $\mathbf{P}$; 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 unit length and are orthogonal to each other. This tells us that the eigenvectors define a space. If we make a matrix $\mathbf{E}$ that contains the 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 $\mathbf{E}^T$. 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 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

---

Hidden page

Hidden page

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 we saw in Section 5.2.

10.2.2 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 we saw in Section 5.2) to get around this problem, just as the SVM got around it for the Perceptron. Just as we did 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:

$$\mathbf{C} = \frac{1}{N}\sum_{n=1}^{N}\Phi(\mathbf{x}_n)\Phi(\mathbf{x}_N)^T, \tag{10.16}$$

which produces the eigenvector equation:

$$\lambda\left(\Phi(\mathbf{x}_i)\mathbf{V}\right) = \left(\Phi(\mathbf{x}_i)\mathbf{C}\mathbf{V}\right) \;\; i = 1\ldots N, \tag{10.17}$$

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:

$$\mathbf{K}_{(i,j)} = \left(\Phi(\mathbf{x}_i)\cdot\Phi(\mathbf{x}_j)\right). \tag{10.18}$$

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:

$$\left(\mathbf{V}^k\cdot\Phi(\mathbf{x})\right) = \sum_{i=1}^{N}\boldsymbol{\alpha}_i^k\left(\Phi(\mathbf{x}_i)\cdot\Phi(\mathbf{x}_j)\right). \tag{10.19}$$

This is all there is to the algorithm.

Hidden page

FIGURE 10.9: Plot of the first two non-linear principal components of the iris data, showing that the three classes are clearly distinguishable.

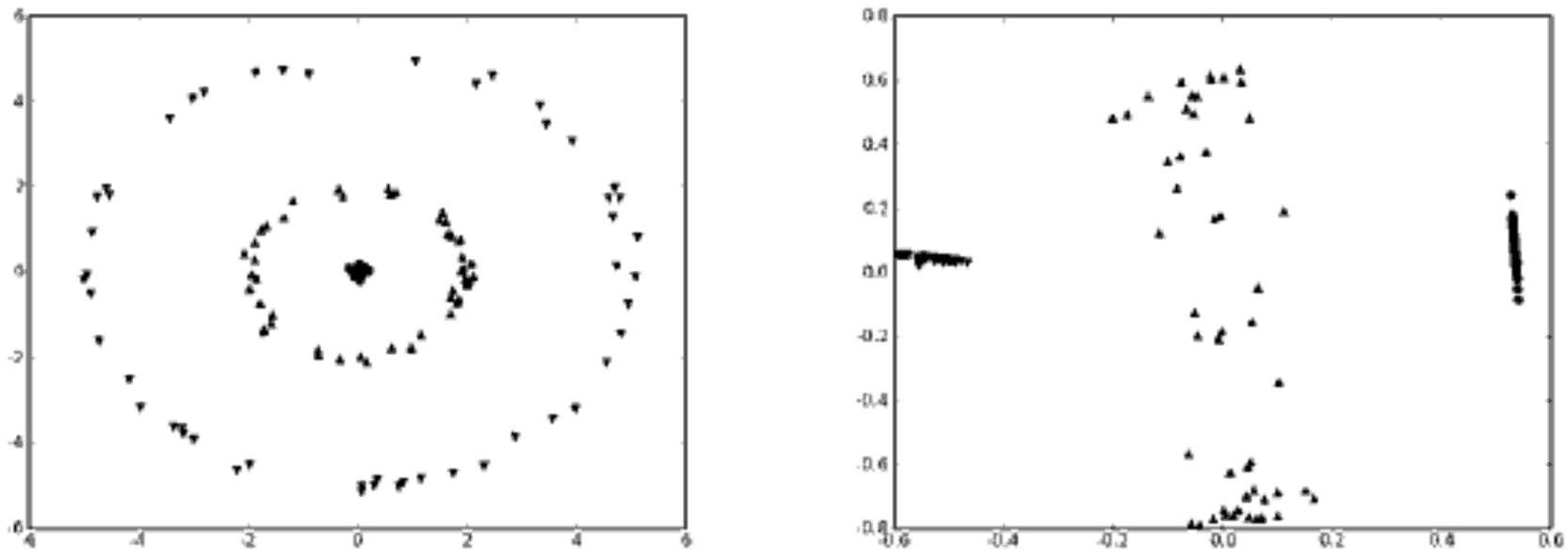


FIGURE 10.10: A very definitely non-linear dataset consisting of three concentric circles, and the kernel PCA mapping of the data, which requires only one component to separate the data.

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.

10.3 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

Hidden page

$$\mathbf{W}_{new} = \left(\mathbf{y}E(\mathbf{x}|\mathbf{y})^T\right)\left(E(\mathbf{x}\mathbf{x}^T|\mathbf{y})\right)^{-1}, \tag{10.23}$$

$$\Psi_{new} = \frac{1}{N}\text{diagonal}\left(\mathbf{x}\mathbf{x}^T - WE(\mathbf{x}|\mathbf{y})\mathbf{y}^T\right), \tag{10.24}$$

where diagonal() ensures that the matrix retains values only on the diagonal and the expectations are:

$$E(\mathbf{x}|\mathbf{y}) = \mathbf{W}^T(\mathbf{W}\mathbf{W}^T + \Psi)^{-1}\mathbf{b} \tag{10.25}$$

$$E(\mathbf{x}\mathbf{x}^T|\mathbf{x}) - E(\mathbf{x}|\mathbf{y})E(\mathbf{x}|\mathbf{y})^T = \mathbf{I} - \mathbf{W}^T(\mathbf{W}\mathbf{W}^T + \Psi)^{-1}W. \tag{10.26}$$

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 = dot(W,transpose(W)) + diag(Psi)
logA = log(abs(linalg.det(A)))
A = linalg.inv(A)

WA = dot(transpose(W),A)
WAC = dot(WA,C)
Exx = eye(nRedDim) - dot(WA,W) + dot(WAC,transpose(WA))

# M-step
W = dot(transpose(WAC),linalg.inv(Exx))
Psi = Cd - (dot(W,WAC)).diagonal()

tAC = (A*transpose(C)).sum()

L = -N/2*log(2.*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 10.11.

Hidden page

Hidden page

Hidden page

Hidden page

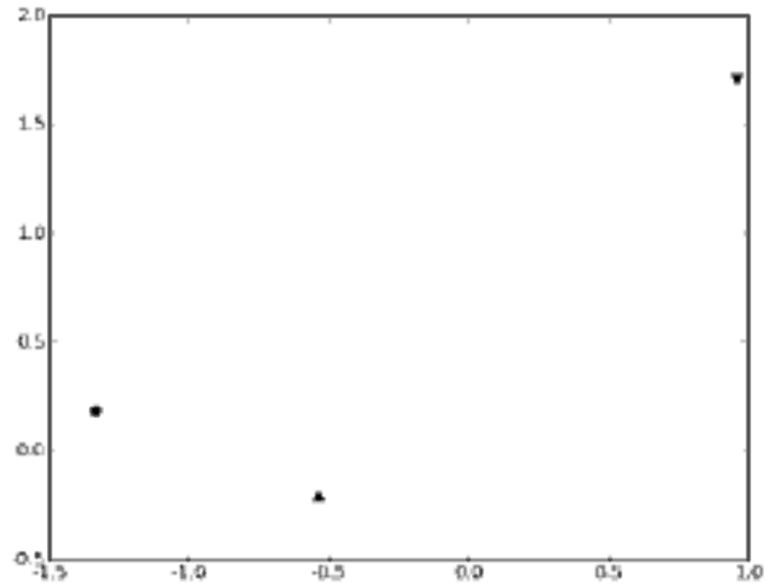


FIGURE 10.12: The Locally Linear Embedding algorithm with $k = 12$ neighbours transforms the iris dataset into three points, separating the data perfectly.

```
for i in range(ndata):
    Z  = data[neighbours[i,:],:] - kron(ones((K,1)),data[i,:])
    C = dot(Z,transpose(Z))
    C = C+identity(K)*1e-3*trace(C)
    W[:,i] = transpose(linalg.solve(C,ones((K,1))))
    W[:,i] = W[:,i]/sum(W[:,i])

M = eye(ndata,dtype=float)
for i in range(ndata):
    w = transpose(ones((1,shape(W)[0]))*transpose(W[:,i]))
    j = neighbours[i,:]
    ww = dot(w,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 = linalg.eig(M)
ind = argsort(evals)
y = evecs[:,ind[1:nRedDim+1]]*sqrt(ndata)
```

The LLE algorithm produces a very interesting result on the iris dataset: it separates the three groups into three points (Figure 10.12). 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 10.13 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 10.13 shows that LLE can successfully unroll it.

Hidden page

Hidden page

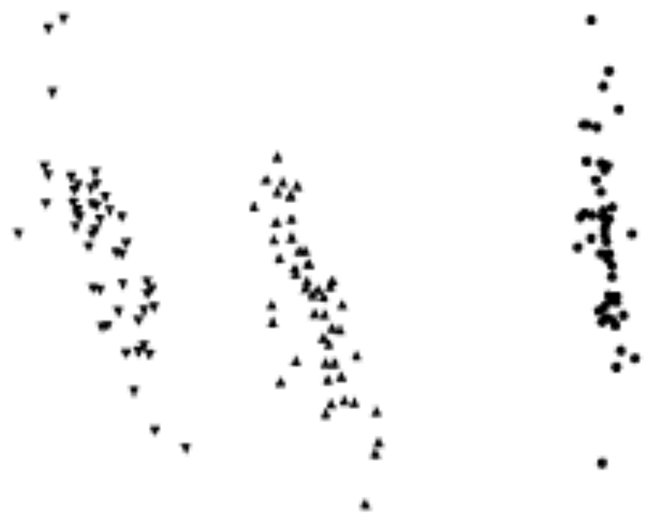


FIGURE 10.14: Isomap transforms the iris data in a similar way to factor analysis, provided that the neighbourhood size is large enough to avoid points becoming disconnected.

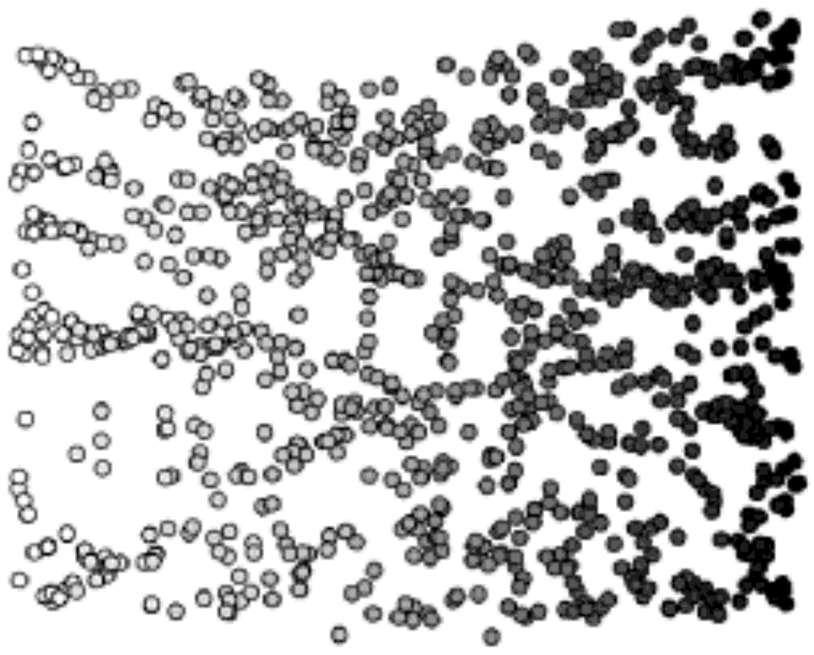


FIGURE 10.15: Isomap also produces a good remapping of the swissroll dataset.

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 10.14 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 10.13, Isomap produces qualitatively similar results to LLE, as can be seen in Figure 10.15.

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*. Springer, Berlin, Germany, 2001.
- (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 10.1 Use LDA on the Iris dataset (which is what Fisher originally tested LDA on).

Problem 10.2 Compare the results with using PCA, which is not supervised and will not therefore be able to find the same space.

Problem 10.3 Compute the eigenvalues and eigenvectors of:

$$\begin{pmatrix} 5 & 7 \\ -2 & -4 \end{pmatrix} \quad \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \quad \begin{pmatrix} 1 & 2 & 1 \\ 6 & -1 & 0 \\ -1 & -2 & -1 \end{pmatrix} \tag{10.33}$$

Problem 10.4 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 10.5 Modify the Isomap code to use Dijkstra's algorithm rather than Floyd's algorithm.

Problem 10.6 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.

Hidden page

Hidden page

Hidden page

Hidden page

Hidden page

value of a function at a point in terms of its derivatives. So, for a function $f(\mathbf{x})$:

$$f(\mathbf{x}) = f(\mathbf{x}_0) + \nabla f(\mathbf{x})|_{\mathbf{x}_0}(\mathbf{x}-\mathbf{x}_0) + \frac{1}{2}(\mathbf{x}-\mathbf{x}_0)^T \nabla^2 f(\mathbf{x})|_{\mathbf{x}_0}(\mathbf{x}-\mathbf{x}_0) + \ldots, \quad (11.2)$$

where $\mathbf{x}_0$ is a common, but potentially slightly confusing notation for the initial guess $\mathbf{x}(0)$, the $|_{\mathbf{x}_0}$ notation means that the function is evaluated at that point, and $\mathbf{J}(\cdot)$ is the Jacobian matrix of first derivatives:

$$\mathbf{J}(\mathbf{x}) = \frac{\partial f(\mathbf{x})}{\partial \mathbf{x}} = \begin{pmatrix} \frac{\partial f_1(\mathbf{x})}{\partial x_1} & \cdots & \frac{\partial f_1(\mathbf{x})}{\partial x_n} \\ & \cdots & \\ \frac{\partial f_n(\mathbf{x})}{\partial x_1} & \cdots & \frac{\partial f_n(\mathbf{x})}{\partial x_n} \end{pmatrix} \quad (11.3)$$

and $\mathbf{H}(\mathbf{x})$ is the Hessian matrix of second derivatives defined in a similar way to $\mathbf{J}(\mathbf{x})$, but using second derivatives. If $f(\mathbf{x})$ is a scalar function (so that it returns just 1 number) then $\mathbf{J}(\mathbf{x}) = \nabla f(\mathbf{x})$ and is a vector and $\mathbf{H}(\mathbf{x}) = \nabla^2 f(\mathbf{x})$ is a two-dimensional matrix. For a vector $\mathbf{f}(\mathbf{x})$, $\mathbf{J}(\mathbf{x})$ is a two-dimensional matrix and $\mathbf{H}(\mathbf{x})$ is three-dimensional.

If we choose to minimise Equation (11.2) exactly as it is written (i.e., ignoring third derivatives and higher), then we find the Newton direction at the kth iteration to be: $\mathbf{p}_k = -(\nabla^2 f(\mathbf{x}_k))^{-1} \nabla f(\mathbf{x}_k)$. There is something important to notice about this equation, which is that we actually use the inverse of the Hessian. Computing this is generally of order $\mathcal{O}(N^3)$ (where N is the number of elements in the matrix) which makes this a computationally expensive method. The compensation for this cost is that we don't really have to worry about the stepsize at all; it is always set to 1.

Implementing this requires only 1 line of change to our basic steepest descent algorithm, plus the addition of a function that computes the Hessian. The line to change is the one that computes $\mathbf{p}_k$, which becomes:

```
p = -dot(linalg.inv(Hessian(x)),Jacobian(x))
```

For this simple example, this algorithm goes straight to the correct answer in one step, which is much better than the steepest descent method that we saw earlier. However, for more complicated functions it won't work as well, and we can do better, as we shall see.

11.2.2 The Levenberg-Marquardt Algorithm

For least-squares problems, the objective function that we are optimising is:

$$f(\mathbf{x}) = \frac{1}{2}\sum_{j=1}^{m} r_j^2(\mathbf{x}) = \frac{1}{2}\|\mathbf{r}(\mathbf{x})\|_2^2, \quad (11.4)$$

where the $\frac{1}{2}$ makes the derivative nicer, and $\mathbf{r}(\mathbf{x}) = (r_1(\mathbf{x}), r_2(\mathbf{x}), \ldots, r_m(\mathbf{x}))^T$. In this last version, we can write the Jacobian of $\mathbf{r}$ as:

$$\mathbf{J}(\mathbf{x}) = \left\{ \begin{array}{cccc} \frac{\partial r_1}{\partial x_1} & \frac{\partial r_2}{\partial x_1} & \cdots & \frac{\partial r_m}{\partial x_1} \\ \frac{\partial r_1}{\partial x_2} & \frac{\partial r_2}{\partial x_2} & \cdots & \frac{\partial r_m}{\partial x_2} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial r_1}{\partial x_n} & \frac{\partial r_2}{\partial x_n} & \cdots & \frac{\partial r_m}{\partial x_n} \end{array} \right\} = \left[\frac{\partial r_j}{\partial x_i} \right]_{j=1,\ldots,m,\ i=1,\ldots,n}. \tag{11.5}$$

This is useful because the function gradients that we want can mostly be computed directly:

$$\nabla f(\mathbf{x}) = \mathbf{J}(\mathbf{x})^T \mathbf{r}(\mathbf{x}) \tag{11.6}$$

$$\nabla^2 f(\mathbf{x}) = \mathbf{J}(\mathbf{x})^T \mathbf{J}(\mathbf{x}) + \sum_{j=1}^{m} r_j(\mathbf{x}) \nabla^2 r_j(\mathbf{x}). \tag{11.7}$$

The upshot of this is that knowing the Jacobian gives you the first (and usually, most important) part of the Hessian effectively without any additional computational cost, and it is this that special algorithms can exploit to solve least-squares problems efficiently. To see this, remember that, as in all of the other gradient-descent algorithms that we have looked at, we are approximating the function by the Taylor series (Equation (11.2)) up to second-order (Hessian) terms.

If $\|\mathbf{r}(\mathbf{x})\|$ is linear (which means that $f(\mathbf{x})$ is quadratic), then the Jacobian is constant and $\nabla^2 r_j(\mathbf{x}) = 0$ for all j. In this case, $f(\mathbf{x}) = \frac{1}{2}\|\mathbf{J}\mathbf{x} + \mathbf{r}\|^2$, and at a solution:

$$\nabla f(\mathbf{x}) = \mathbf{J}^T(\mathbf{J}\mathbf{x} + \mathbf{r}) = 0, \tag{11.8}$$

and so:

$$\mathbf{J}^T\mathbf{J}\mathbf{x} = \mathbf{J}^T\mathbf{r}(\mathbf{x}). \tag{11.9}$$

We can use this along with some linear algebra to find $\mathbf{x}$ in a variety of different ways, such as Cholesky factorisation, QR factorisation, and using the Singular Value Decomposition. We will look at the last of these methods, since it uses eigenvectors, which we have already seen in Chapter 10. The Singular Value Decomposition (SVD) is the decomposition of a matrix $\mathbf{A}$ of size $m \times n$ into:

$$\mathbf{A} = \mathbf{U}\mathbf{S}\mathbf{V}^T, \tag{11.10}$$

where $\mathbf{U}$ and $\mathbf{V}$ are orthogonal matrices (i.e., the inverse of the matrix is its transpose, so $\mathbf{U}^T\mathbf{U} = \mathbf{U}\mathbf{U}^T = \mathbf{I}$, where $\mathbf{I}$ is the identity matrix). $\mathbf{U}$ is of size $m \times m$ and $\mathbf{V}$ is of size $n \times n$. $\mathbf{S}$ is a diagonal matrix of size $m \times n$, with the elements of this matrix, σ_i, being known as singular values.

Hidden page

The Levenberg-Marquardt Algorithm

- given start point $\mathbf{x}_0$
- while $\mathbf{J}^T\mathbf{r}(\mathbf{x})$ >tolerance and maximum number of iterations not exceeded:
  - repeat
    - solve $(\mathbf{J}^T\mathbf{J} + \nu\mathbf{I})\mathbf{dx} = -\mathbf{J}^T\mathbf{r}$ for $\mathbf{dx}$ using linear least-squares
    - set $\mathbf{x}_{\text{new}} = \mathbf{x} + \mathbf{dx}$
    - compute the ratio of the actual and prediction reductions:
      - actual $= \|f(\mathbf{x}) - f(\mathbf{x}_{new})\|$
      - predicted $= \nabla f^T(\mathbf{x}) \times \mathbf{x}_{new} - \mathbf{x}$
      - ρ = actual/predicted
    - if $0 < \rho < 0.25$:
      - accept step: $\mathbf{x} = \mathbf{x}_{\text{new}}$
    - else if $\rho > 0.25$:
      - accept step: $\mathbf{x} = \mathbf{x}_{\text{new}}$
      - increase trust region size (reduce ν)
    - else:
      - reject step
      - reduce trust region (increase ν)
  - until $\mathbf{x}$ is updated or maximum number of iterations is exceeded

We will look at two examples of using non-linear least-squares. One is a simple case of finding the minimum of a function that consists of two quadratic terms added together, i.e., a sum-of-squares problem, while the second is to minimise the fitting of a function to data.

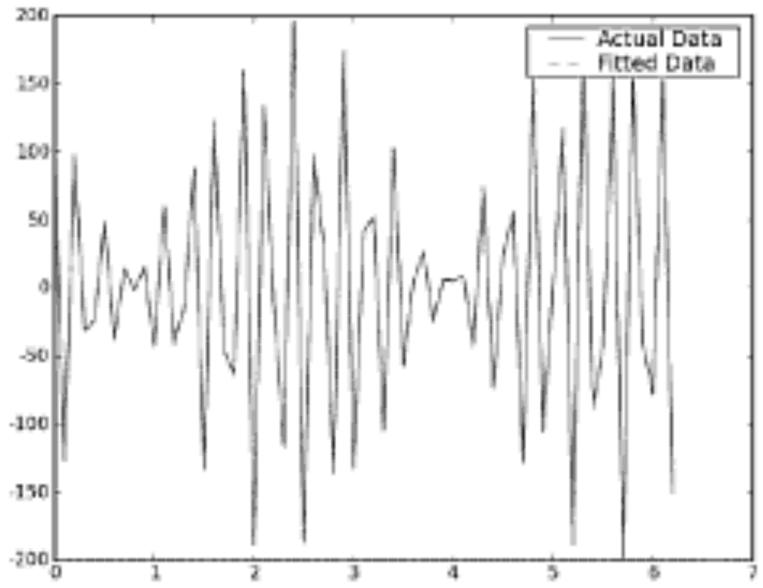
The function that we will attempt to minimise is Rosenbrock's function:

$$f(x_1, x_2) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2. \tag{11.15}$$

This is a common problem to try since it has a long narrow valley, so finding the optimal solution is not especially easy (except by hand: if you look at the problem, then guessing that $x_1 = 1, x_2 = 1$ is the minimum is fairly obvious). You need to work out how to encode this in the form required for a sum-of-squares problem, which is basically to write:

$$\mathbf{r} = (10(x_2 - x_1^2), 1 - x_1)^T. \tag{11.16}$$

Hidden page



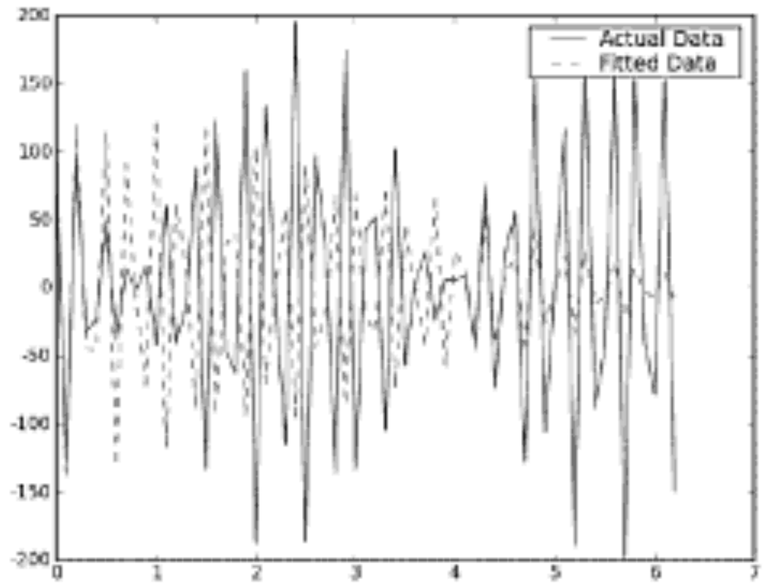


FIGURE 11.4: Using Levenberg-Marquardt for least-squares data fitting of data from Equation (11.18). The example on the left converges to the correct solution, while the one on the right, which still starts from a point close to the correct solution, fails to find it, resulting in significantly different output.

starting point is $(100.5, 102.5)$, while on the right it is $(101, 101)$. It can be seen that on this problem, Levenberg-Marquardt is very susceptible to local minima, since while the example on the left works (converging after only 8 iterations), the example on the right, which still starts with parameter values very close to the correct ones, gets stuck and fails, with final parameter values $(100.89, 101.13)$.

11.3 Conjugate Gradients

Not every problem that we want to solve is a least-squares problem. The good news is that we can do rather better than steepest descent even when we want to minimise an arbitrary objective function. The key to this is to look again at Figure 11.3, where you can see that there are several of the steepest gradient lines that are in pretty much the same direction. We would only need to go in that direction once if we knew how far to go the first time. And then we would go in a direction orthogonal (at right angles) to that one and, in two dimensions, we would be finished, as is shown on the right of Figure 11.5, where one step in the x direction and one in the y direction are enough to complete the minimisation. In n dimensions we would have to take n steps, and then we would have finished. This amazing scenario is the aim of the method of conjugate gradients. It manages to achieve it in the linear case, but in most non-linear cases, which are the kind we are usually interested in, it usually requires a few more iterations that it theoretically should, although still many less steps than most other methods for real problems.

It turns out that making the lines be orthogonal is generally impossible, since you don't have enough information about the solution space. However,

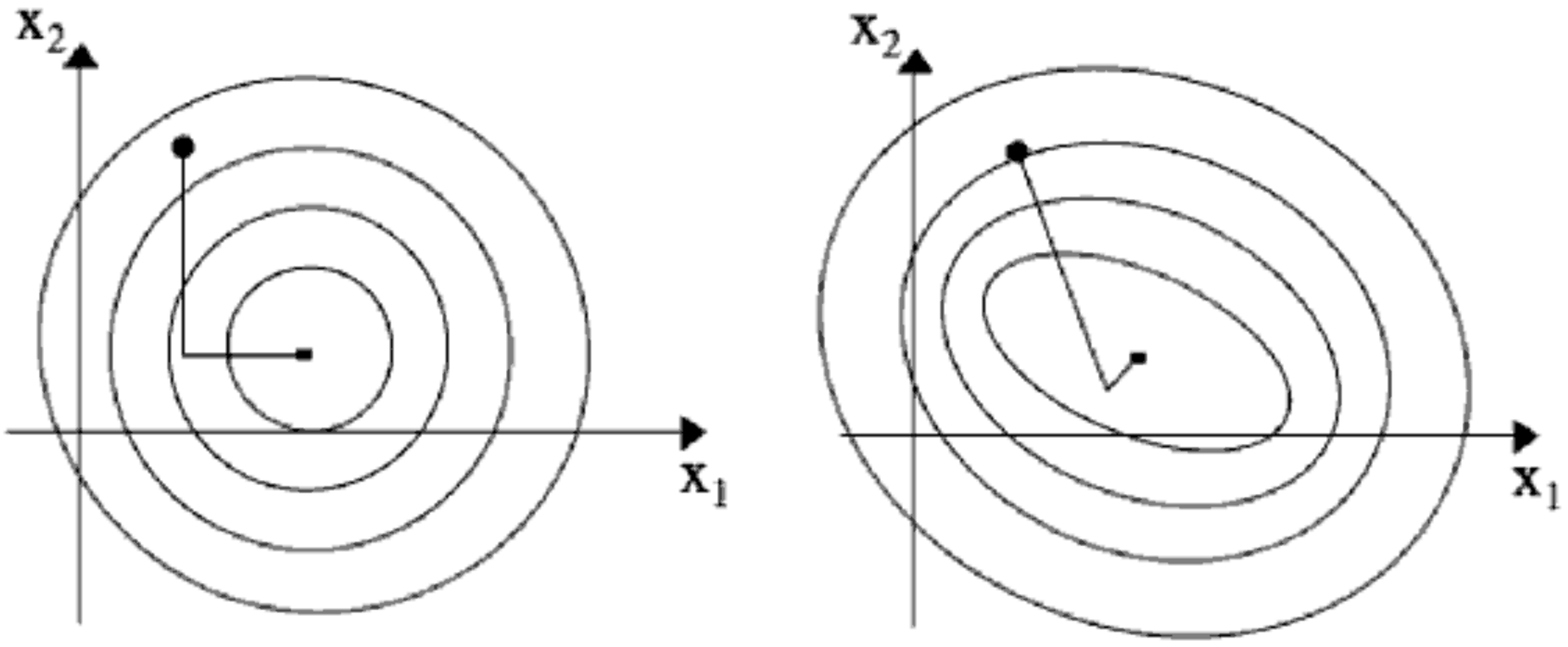


FIGURE 11.5: *Left:* If the directions are orthogonal to each other and the stepsize is correct, then only one step is needed for each dimension in the data, here two. *Right:* The conjugate directions are not orthogonal to each other on the ellipse.

it is possible to make them conjugate or A-orthogonal. Two vectors $\mathbf{p}_i, \mathbf{p}_j$ are conjugate if $\mathbf{p}_i^T \mathbf{A} \mathbf{p}_j = 0$ for some matrix $\mathbf{A}$. Conjugate lines for the ellipse contours in Figure 11.2 are shown on the right of Figure 11.5. Amazingly, the line search that we wrote down in Equation (11.1) is soluble along these directions, since they do not interfere with each other, with solution:

$$\alpha_i = \frac{\mathbf{p}_i^T(-\nabla f(\mathbf{x}_{i-1})}{\mathbf{p}_i^T \mathbf{A} \mathbf{p}_i}). \tag{11.20}$$

We then need to use a function to find the zeros of this. The Newton-Raphson iteration, which is one method that will do it, is described below. So if we can find conjugate directions, then the line search is much better. The only question that remains is how to find them. This requires a Gram-Schmidt process, which constructs each new direction by taking a candidate solution and then subtracting off any part that lies along any of the directions that have already been used. We start by picking a set of mutually orthogonal vectors $\mathbf{u}_i$ (the basic coordinate axes will do; there are better options, but they are beyond the scope of this book) and then using:

$$\mathbf{p}_k = \mathbf{u}_k + \sum_{i=0}^{k-1} \beta_{ki} \mathbf{p}_i. \tag{11.21}$$

There are two possible β terms that can be used. They are both based on the ratios between the squared Jacobian before and after an update. The Fletcher-Reeves formula is:

$$\beta_{i+1} = \frac{\nabla f(\mathbf{x}_{x+1})^T \nabla f(\mathbf{x}_{i+1})}{\nabla f(\mathbf{x}_i)^T \nabla f(\mathbf{x}_i)}, \tag{11.22}$$

Hidden page

The Conjugate Gradients Algorithm

- given start point $\mathbf{x}_0$, and stopping parameter ϵ, set $\mathbf{p}_0 = -\nabla f(\mathbf{x})$
- set $\mathbf{p}_{\text{new}} = \mathbf{p}_0$
- while $\mathbf{p}_{\text{new}} > \epsilon^2 \mathbf{p}_0$:
  - compute α_k and $\mathbf{x}_{\text{new}} = \mathbf{x} + \alpha_k \mathbf{p}$ using the Newton-Raphson iteration:
    - while $\alpha^2 dp > \epsilon^2$:
      - $\alpha = -(\nabla f(\mathbf{x})^T \mathbf{p})/(\mathbf{p}^T \mathbf{H}(\mathbf{x})\mathbf{p})$
      - $\mathbf{x} = \mathbf{x} + \alpha \mathbf{p}$
      - $dp = \mathbf{p}^T \mathbf{p}$
  - evaluate $\nabla f(\mathbf{x}_{\text{new}})$
  - compute β_{k+1} using Equation (11.22) or (11.23)
  - update $\mathbf{p}_{\text{new}} = \nabla f(\mathbf{x}_{\text{new}}) + \beta_{k+1}\mathbf{p}$
  - check for restarts

11.3.1 Conjugate Gradients Example

Computing the conjugate gradients solution to the same function as above: $f(\mathbf{x}) = (0.5x_1^2 + 0.2x_2^2 + 0.6x_3^2)$ makes use of the Jacobian and Hessian again. The first Newton-Raphson step yields an α value of 0.931, so that the next step is:

$$\mathbf{x}(1) = \begin{pmatrix} -2 \\ 2 \\ 0 \end{pmatrix} + 0.931 \times \begin{pmatrix} 2 \\ -0.8 \\ 2.4 \end{pmatrix} = \begin{pmatrix} -0.138 \\ 1.255 \\ 0.235 \end{pmatrix} \tag{11.27}$$

Then $\beta = 0.0337$, so that the direction is:

$$\mathbf{p}(1) = \begin{pmatrix} 0.138 \\ -0.502 \\ -0.282 \end{pmatrix} + 0.0337 \times \begin{pmatrix} 2 \\ -0.8 \\ 2.4 \end{pmatrix} = \begin{pmatrix} 0.205 \\ -0.529 \\ -0.201 \end{pmatrix} \tag{11.28}$$

In the second step, $\alpha = 1.731$,

$$\mathbf{x}(2) = \begin{pmatrix} -0.138 \\ 1.255 \\ 0.235 \end{pmatrix} + 1.731 \times \begin{pmatrix} 0.205 \\ -0.529 \\ -0.201 \end{pmatrix} = \begin{pmatrix} -0.217 \\ -0.136 \\ 0.136 \end{pmatrix} \tag{11.29}$$

and the update is:

$$p(2) = \begin{pmatrix} -0.217 \\ -0.136 \\ 0.136 \end{pmatrix} + 0.240 \times \begin{pmatrix} 0.205 \\ -0.529 \\ -0.201 \end{pmatrix} = \begin{pmatrix} -0.168 \\ -0.263 \\ 0.088 \end{pmatrix} \tag{11.30}$$

A third step then gives the final answer as $(0, 0, 0)$.

There are other methods of doing gradient descent, some of which are more effective on certain problems (but note that the No Free Lunch theorem tells us that no one solver will be the most effective for every problem). However, they are not necessary for an understanding of machine learning. The scientific Python libraries SciPy include a set of optimisation algorithms in `scipy.optimize` that might be worth a look if you want to find more complete algorithms in this area.

We will next consider what happens when the problems that we wish to solve are discrete, and so there is no gradient to find.

11.4 Search: Three Basic Approaches

We are going to discuss three different ways to attempt optimisation without gradients. For each one, we will see how it works on the Travelling Salesman Problem (TSP), which is a classic discrete optimisation problem that consists of trying to find the shortest route through a set of cities that visits each city exactly once and returns to the start. For the first (starting) city we can choose any of the N that are available. For the next, there are $N - 1$ choices, and for the next $N - 2$. Using a brute force search in this way provides a $\mathcal{O}(N!)$ solution, which is obviously infeasible. In fact, the TSP is an NP-hard problem. The best known solution that is guaranteed to find the global maximum is using dynamic programming and its computational cost is $\mathcal{O}(n^2 2^n)$, but we won't be considering that here—the TSP is an example, not a problem we really want to solve here. The basic search methods are described next.

11.4.1 Exhaustive Search

Try out every solution and pick the best one. While this is obviously guaranteed to find the global optimum, because it checks every single solution, it is impractical for any reasonable size problem. For the TSP it would involve testing out every single possible way of ordering the cities, and calculating the distance for each ordering, so the computational complexity is $\mathcal{O}(N!)$, which is worse than exponential. It is computationally infeasible to do the computations for more than about $N = 10$ cities. The basic part of the algorithm

Hidden page

Hidden page

FIGURE 11.6: A one-armed bandit machine. It has one arm, and it steals your money.

it will take a very long time to reach the optimal solution. The second is on a plateau, where no changes that the algorithm makes affect the solution. In this case the solution will just change randomly, if at all, and the maximum will probably not be found. The third case is when there is a very gently sloping ridge in the data. Most directions that the algorithm looks in will be downhill, and so it may decide that it has already reached the maximum.

11.5 Exploitation and Exploration

The search methods above can be separated into methods that perform exploration of the search space, always trying out new solutions, like exhaustive search, and those performing exploitation of the current best solution, by trying out local variations of that current best solution, like hill climbing. Ideally, we would like some combination of the two—we should be trying to improve on the current best solution by local search, and also looking around in case there is an even better solution hiding elsewhere in the search space.

One way to think about this is known as the n-armed bandit problem. Suppose that we have a room full of one-armed bandit machines in some tacky Las Vegas casino (for those who don't know, a one-armed bandit is a slot machine with a lever that you pull, as in Figure 11.6). You don't know anything about the machines in advance, such as what the payouts are, and how likely you are to get the payout. You enter the room with a fistful of 50 cent coins from your student loan, aiming to generate enough beer money to get through the year. How do you choose which machine to use?

At first, you have no information at all, so you choose randomly. However, as you explore, you pick up information about which machines are good (here,

Hidden page

Hidden page

Hidden page

Problem 11.2 Experiment with the Fletcher-Reeves and Polak-Ribiere formulas (Equations (11.22) and (11.23)) when solving Rosenbrock's function using conjugate gradients. Can you find places where one works better than the other?

Problem 11.3 Generate data from the equation $a(1 - \exp(-b(x - c)))$ for choice of parameters a, b, c and x in the range -5 to 5 (with noise). Use Levenberg-Marquardt to fit the parameters.

Problem 11.4 It is possible to use the exact gradient descent methods described in this chapter for an MLP. The first thing you have to do is to put the various sets of weights into one data structure (which is easy: simply put the two sets into one array). You then need to work out what the derivatives are for these different functions, and then compute the Hessian. The structure can then be passed to the optimisation code. Implement this and compare the results to using back-propagation.

Problem 11.5 By incorporating back-tracking into hill climbing, it is possible to escape from some poor local maxima. Add this into the code and test the results on the Travelling Salesman problem.

Problem 11.6 The logical satisfiability problem is an NP-complete problem that consists of finding truth assignments to sets of logical statements (e.g., $(a_1 \wedge a_2) \vee (\neg a_1 \vee a_3)$) so that they are true. It is an NP-complete problem to find truth assignments. Devise a way to use hill climbing and simulated annealing on the problem.

Chapter 12

Evolutionary Learning

In this chapter we are going to start by treating evolution the same way that we treated neuroscience earlier in the book—by cherry-picking a few useful concepts, and then filling in the gaps with computer science in order to make an effective learning method. To see why this might be interesting, you need to view evolution as a search problem. We don't generally think of it in this way, but animals are competing with each other in all kinds of ways—for example, eating each other—which encourages them to try to find camouflage colours, become toxic to certain predators, etc.

Evolution works on a population through an imaginary fitness landscape, which has an implicit bias towards animals that are 'fitter,' i.e., those animals that live long enough to reproduce, are more attractive, and so get more mates, and generate more and healthier offspring. You can find out more from hundreds of books, such as Charles Darwin's "The Origin of Species" (the original book on the topic, still in print and very interesting) and Richard Dawkin's "The Blind Watchmaker."

The genetic algorithm models the genetic process that gives rise to evolution. In particular, it models sexual reproduction, where both parents give some genetic information to their offspring. As is sketched in Figure 12.1, in biological organisms, each parent passes on one chromosome out of their two, and so there is a 50% chance of any gene making it into the offspring. Of the two versions of each gene (one from each parent) one allele (variation) is selected. Hence, children have similarities with their parents, and there is lots of genetic inheritance. However, there are also random mutations, caused by copying errors when the chromosome material is reproduced, which mean that some things do change over time. Real genetics is obviously a lot more complicated than this, but we are taking only the things that we want for our model.

The genetic algorithm shows many of the things that are best and worst about machine learning: it is often, but not always, very effective, it has an array of parameters that are crucial, but hard to set, and it is impossible to guarantee that it will find a result that is any good at all. Having said all that, it often works very well, and it has become a very popular algorithm for people to use when they have no idea of any other way to find a reasonable solution.

In the terms that we saw at the end of the previous chapter, genetic al-

Hidden page

- a way to calculate the fitness of a solution
- a selection method to choose parents
- a way to generate offspring by breeding the parents

These items are all described in the following sections, and the basic algorithm is described. We are going to use an example to describe the methods, which is an NP-complete problem (if you are not familiar with the term NP-complete, its practical implication is that the problem runs in exponential time in the number of inputs) known as the knapsack problem (a knapsack is a rather old name for a rucksack or bag). Sections 12.3.1 and 12.3.4 provide other examples. The knapsack problem is easy to describe, but difficult to solve in general. Here is the version of it that we will use:

Suppose that you are packing for your holidays. You've bought the biggest and best rucksack that was for sale, but there is still no way that you are going to fit in everything you want to take (camera, money, addresses of friends, etc.) and the things that your mum is insisting you take (spare underwear, phrasebook, stamps to write home with, etc.). As a good computer scientist you decide to assign a value to each item, and measure how much space it takes up. Then you want to maximise the value of the items you will take with you, with the constraint that everything has to fit into the bag.

This problem, and variations of it, appear in various disguises in cryptography, combinatorics, applied mathematics, logistics, and business, so it is an important problem. Unfortunately, it is also NP-complete, so finding the optimal solution for interesting cases is computationally impossible. We are going to find approximations to the correct solution using a Genetic Algorithm.

12.1.1 String Representation

The first thing that we need is some way to represent the individual solutions, in analogy to the chromosome. GAs use a string, with each element of the string (equivalent to the gene) being chosen from some alphabet. The different values in the alphabet, which is often just binary, are analogous to the alleles. For the problem we are trying to solve we have to work out a way of encoding the description of a solution as a string. We then create a set of random strings to be our initial population.

It is possible to modify the GA so that the alphabet it uses runs over the real numbers. While purists don't think that this is a GA at all, it is quite popular, because of the number of applications, but it is not as elegant as using a discrete alphabet. It also makes the mutation operator that we will see later less useful.

For the knapsack problem the alphabet is very simple, since we can make it binary. We make the string L units long, where L is the total number of things we would like to take with us, and make each unit a binary digit. We then encode a solution using 0 for the things we will not take and 1 for the

things we will. So if there were four things we wanted to take, then $(0, 1, 1, 0)$ would mean that we take the middle two, but not the first or last.

Note that this does not tell us whether or not this string is possible (that is, whether it will fit into the knapsack), nor whether it is a good string (whether it fills the knapsack). To work these out we need some way to decide how well each string fulfills the problem criteria. This is known as the fitness of the string.

12.1.2 Evaluating Fitness

The fitness function can be seen as an oracle that takes a string as an argument and returns a value for that string. It is the only problem-specific part of the algorithm. It is worth thinking about what we want from our fitness function. Clearly, the best string should have the highest fitness, and the fitness should decrease as the strings do less well on the problem. In general, fitness should always be a positive function—even the least fit strings should have fitness of at least zero. In real evolution, the fitness landscape is not static: there is competition between different species, such as predators and prey, or medical cures for certain diseases, and so the measure of fitness changes over time. We'll ignore that in the genetic algorithm.

For the knapsack problem, we could decide that we want to make the bag as full as possible. So we would need to know the volume of each item that we want to put into the knapsack, and then for a given string that says which things should be taken, and which should not, we can compute the total volume. This is then a possible fitness function. However, it does not tell us anything about whether they will fit into the bag—with this fitness function the optimal solution is to take everything. So we need to check that they will fit, and if they will not, reduce the fitness of that solution. One option would be to set the fitness to 0 if it will not fit. However, suppose that the solution is almost perfect, it is just that there is one thing too many in the knapsack. By setting the fitness to 0 we are reducing the chance of this solution being allowed to evolve and improve during later iterations. For this reason we will make the fitness function be the sum of the values of the items to be taken if they fit into the knapsack, but if they do not we will subtract twice the amount by which they are too big for the knapsack from the size of the knapsack. This allows solutions that are only just over to be considered for improvement, but tries to ensure that they are not the fittest solutions around.

There is an obvious greedy algorithm that finds solutions to the knapsack problem. At each stage it takes the largest thing that hasn't been packed yet and that will still fit into the bag, and iterates that rule. This will not necessarily return the optimal solution (unless each thing is larger than the sum of all the ones smaller than it, in which case it will), but it is very quick and simple. So a GA should be getting a much better solution than the greedy rule in general to be worth all the effort involved in writing and running it.

12.1.3 Population

We can now measure the fitness of any string. The GA works on a population of strings, with the first generation usually being created randomly. The fitness of each string is then evaluated, and that first generation is bred together to make a second generation, which is then used to generate a third, and so on. After the initial population is chosen randomly, the algorithm evolves in such a way that the fitness of individuals in the population increases over the generations.

So for the knapsack problem, we will now create a set of random binary strings of length L by using the random number generator. We'll arbitrarily decide to make 100 strings. We now need to choose parents out of this population, and start breeding them. At every iteration the population stays the same size, something else that is unlike real evolution. Creating the initial population is very easy in NumPy using the uniform random number generator and the `where()` function:

```
pop = random.rand(popSize,stringLength)
pop = where(pop<0.5,0,1)
```

12.1.4 Generating Offspring: Parent Selection

For the current generation we need to select those strings that will be used to generate new offspring. The idea here is that fitness will improve if we select strings that are already relatively fit compared to the other members of the population (following natural selection). This is exploitation of our current population. However, it is also good to allow some exploration in there, which means that we have to allow some possibility of weak strings being considered. The basic idea is that we choose strings proportionally to their fitness, so that fitter strings are more likely to be chosen to enter the 'mating pool.' There are two commonly employed ways to do this, although the second one is better:

Truncation Selection A simple method is just to pick some fraction f of the best strings and ignore the rest. For example, $f = 0.5$ is often used, so the best 50% of the strings are put into the mating pool, and chosen with equal probability. This is obviously very easy to implement, but it does limit the amount of exploration that is done, biasing the GA towards exploitation.

Fitness Proportional Selection The better option is to select strings probabilistically, with the probability of a string being selected being proportional to its fitness. The function that is generally used is (for string α):

$$p^\alpha = \frac{F^\alpha}{\sum_{\alpha'} F^{\alpha'}}, \tag{12.1}$$

where F^α is the fitness. This probabilistic interpretation is the reason why fitness should be positive. If they aren't guaranteed positive, then Boltzmann selection can be used to make them so (where s is the selection strength, a parameter, and you might recognise the equation as the softmax activation from Chapter 3):

$$p^\alpha = \frac{\exp(sF^\alpha)}{\sum_{\alpha'} \exp(sF^{\alpha'})}. \tag{12.2}$$

There is an implementation issue here. We want to pick each string with probability proportional to its fitness, but if we only have one copy of each string, then the probability of picking each string is the same. One way around this is to add more copies of the fitter strings, so that they are more likely to get chosen. This is sometimes called 'roulette selection,' because if you imagine that each string gets an area on a roulette wheel, then the larger the area associated to one number, the more likely it is that the ball will land there. You can then just randomly pick strings from this larger set. A method of doing this is shown in the following code snippet, which uses the `kron()` function. We've seen this before (in Section 10.5); it is a NumPy function that multiples each element of its first array argument by every element of the second, putting all of the results together into one multi-dimensional output array. It is useful here in order to populate the new and much larger `newPopulation` array, which contains multiple copies of each string.

```
# Put in repeated copies of each string according to fitness
# Deal with strings with very low fitness
j=0
while round(fitness[j])<1:
 j = j+1

newPop = kron(ones((round(fitness[j]),1)),pop[j,:])

# Add multiple copies of strings into the newPop
for i in range(j+1,self.popSize):
 if round(fitness[i])>=1:
  newPop = concatenate((newPop,kron(ones((round(fitness[i]),1))
  ,pop[i,:])),axis=0)

# Shuffle the order (note that there are still too many)
indices = range(shape(newPop)[0])
```

```
random.shuffle(indices)
newPop = newPop[indices[:popSize],:]
return newPop
```

However we select the strings to put into the mating pool, the next operation is to put them into pairs. Since the order that they are in is random, we can simply pair up the strings so that each even-indexed string takes the following odd-indexed one as its mate.

12.2 Generating Offspring: Genetic Operators

Having selected our breeding pairs, we now need to decide how to combine their two strings to generate the offspring, which is the genetics part of the algorithm. There are two genetic operators that are generally used, and they are discussed now. There are others, but these were the original choices, and are far and away the most common.

12.2.1 Crossover

In biology, organisms have two chromosomes, and each parent donates one of them. Members of our GA population only have one chromosome-equivalent, the string. Thus, we generate the new string as part of the first parent and part of the second. The most common way of doing this is to pick one point at random in the string, and to use parent 1 for the first part of the string, up to the crossover point and parent 2 for the rest. We actually generate two offspring, with the second one consisting of the first part of parent 2 and the second part of parent 1. This scheme is known as single point crossover, and the extension to multi-point crossover is hopefully obvious. The most extreme version is known as uniform crossover and consists of independently selecting each element of the string at random from the two parents. The three types of crossover are shown in Figure 12.2.

Crossover is the operator that performs global exploration, since the strings that are produced are radically different to both parents. The hope is that sometimes we will take good parts of both solutions and put them together to make an even better solution. A nice picture example is to imagine a bird that has webbed feet for good swimming, but that cannot fly, breeding with a bird that can fly, but not swim. The offspring? A duck! Obviously, this is not biologically plausible, but it is a good picture of how crossover works. One interesting feature of the GA that obviously isn't true in real genetics is that in addition to the duck the algorithm would produce the bird that can't fly or

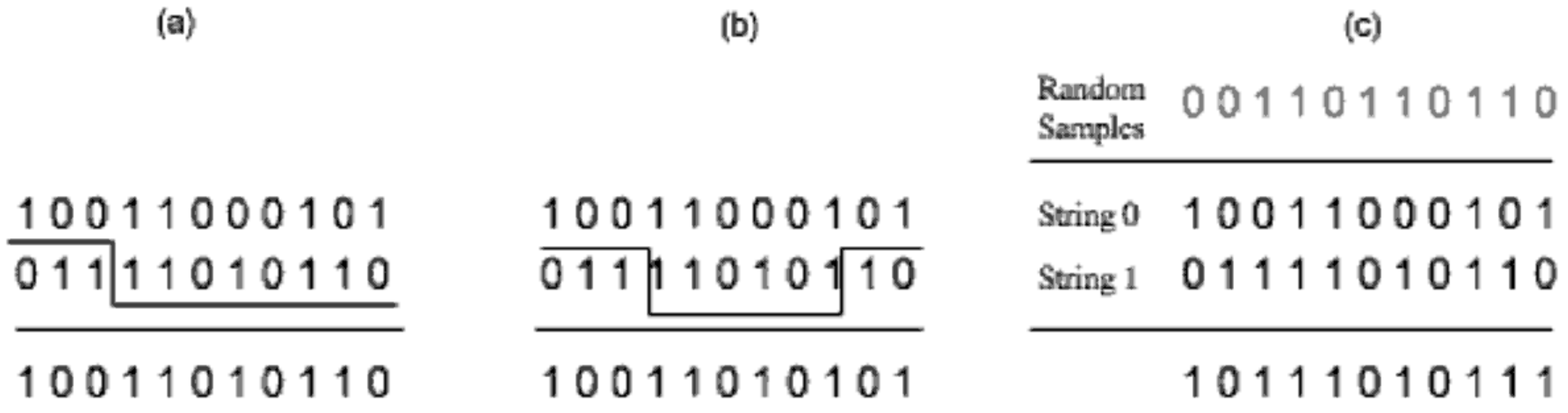


FIGURE 12.2: The different forms of the crossover operator. (a) Single point crossover. A position in the string is chosen at random, and the offspring is made up of the first part of parent 1 and the second part of parent 2.(b) Multi-point crossover. Multiple points are chosen, with the offspring being made in the same way. (c) Uniform crossover. Random numbers are used to select which parent to take each element from.

swim, although it is unlikely to last long since its fitness will presumably not be high. In fact, there are exceptions to this, such as the great New Zealand Kiwi, which can neither swim nor fly, but is happily not extinct.

The following code snippet shows a NumPy implementation of single point crossover. The extension to multi-point and uniform crossover is not particularly difficult.

```
def spCrossover(pop):
    newPop = zeros(shape(pop))
    crossoverPoint = random.randint(0,stringLength,popSize)
    for i in range(0,self.popSize,2):
 newPop[i,:crossoverPoint[i]] = pop[i,:crossoverPoint[i]]
 newPop[i+1,:crossoverPoint[i]] = pop[i+1,:crossoverPoint[i]]
 newPop[i,crossoverPoint[i]:] = pop[i+1,crossoverPoint[i]:]
 newPop[i+1,crossoverPoint[i]:] = pop[i,crossoverPoint[i]:]
    return newPop
```

Crossover is not always useful, depending upon the problem; for example, in the Travelling Salesman Problem that we talked about in Chapter 11, the strings that are generated by crossover might not even be valid tours. However, when it is useful, it is often the more powerful of the genetic operators, and has led to the building block hypothesis of how GAs work. The idea is that GAs work well on problems where the solution comes from putting together lots of little solutions, so that different strings assemble each separate building block, and then crossover puts those substrings together to make the final solution.

Hidden page

Hidden page

The Basic Genetic Algorithm

- **Initialisation**
  - generate N random strings of length L with our chosen alphabet
- **Learning**
  - repeat:
    - create an (initially empty) new population
    - repeat:
      - select two strings from current population by fitness
      - recombine them to produce two new strings
      - mutate the offspring
      - either add the two offspring to the population or use elitism or tournaments
      - keep track of the best string in the population
    - until N strings for the new population are generated
    - replace the current population with the new population
  - until stopping criteria met

12.3 Using Genetic Algorithms

12.3.1 Map Colouring

Graph colouring is a typical discrete optimisation problem. We want to colour a graph using only k colours, and choose them in such a way that adjacent regions have different colours. It has been mathematically proven that any two-dimensional planar graph can be coloured with four colours, which was the first ever proof that used a computer program to check the cases. Even though it might be impossible, we are going to try to solve the three-colour problem using a genetic algorithm, we just won't be upset if the solution isn't perfect (this is a good idea with a GA anyway, of course). With all problems where you want to apply a genetic algorithm, there are three basic tasks that need to be performed:

Encode possible solutions as strings For this problem, we'll choose our alphabet to consist of the three possible shades (black (b), dark (d), and light (l), say). So for a six-region map, a possible string is $\alpha = \{bdblbb\}$. This says that the first region is black, the second dark grey, etc. We

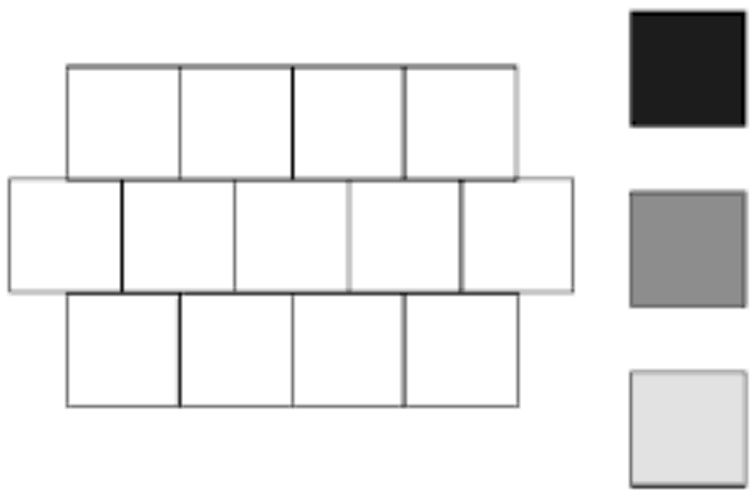


FIGURE 12.4: A sample map that we wish to colour using the three colours shown, without any two adjacent squares having the same colour.

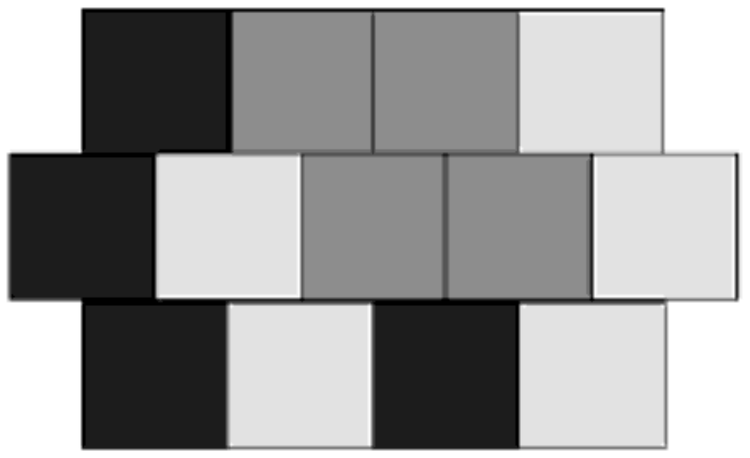


FIGURE 12.5: A possible colouring with several adjacent squares having the same colour.

choose an order to record the regions in and stick to it for all the strings, and now we can encode any way of colouring in those six regions. An example problem and a colouring are given in Figures 12.4 and 12.5.

Choose a suitable fitness function The thing that we want to minimise (a cost function) is the number of times that two adjacent regions have the same colour. We could count these up fairly simply, but it is not a fitness function, because the best solution has the lowest number, not the highest. One easy way to turn it into a fitness function would be to use the Boltzmann selection described earlier (Equation (12.2)), or to count the total number of lines between regions and subtract off the number where the two regions on either side of the line have the same colour. However, we could also just count the number of correct edges. The example in Figure 12.5 has 16 out of the 26 boundaries correct (where a boundary is the intersection between any two squares), so its fitness is 16.

Choose suitable genetic operators We'll use the standard genetic operators for this, since this example makes the operations of crossover and mutation clear. The way that they are used is shown in Figures 12.6 and 12.7. In general, people just use the standard operators for most problems, but if they don't work well, it can be worth putting some effort into thinking of new ones.

Having made those choices, we can let the GA run on the problem, with a possible population and their offspring shown in Figure 12.8, and look at the best solutions after some preset number of iterations. The GA produces good solutions to this problem, and implementing it for yourself is one of the suggested exercises for this chapter.

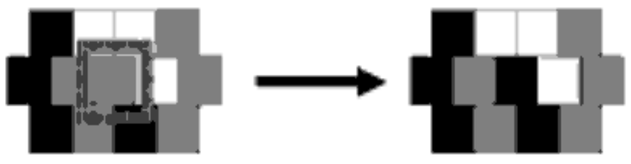


FIGURE 12.6: The way that mutation is performed on a colour, changing it into one of the other colours.

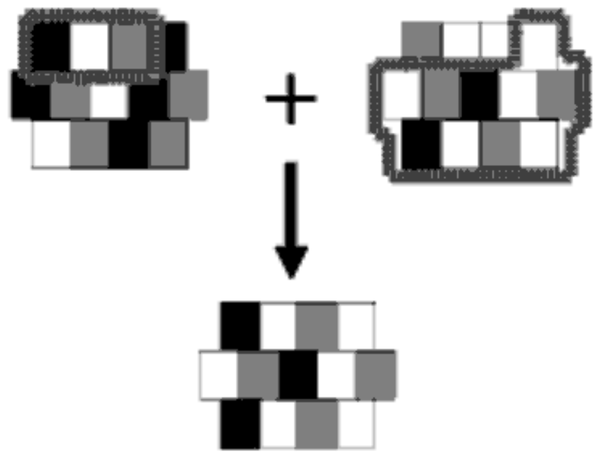


FIGURE 12.7: The effects of crossover on a map.

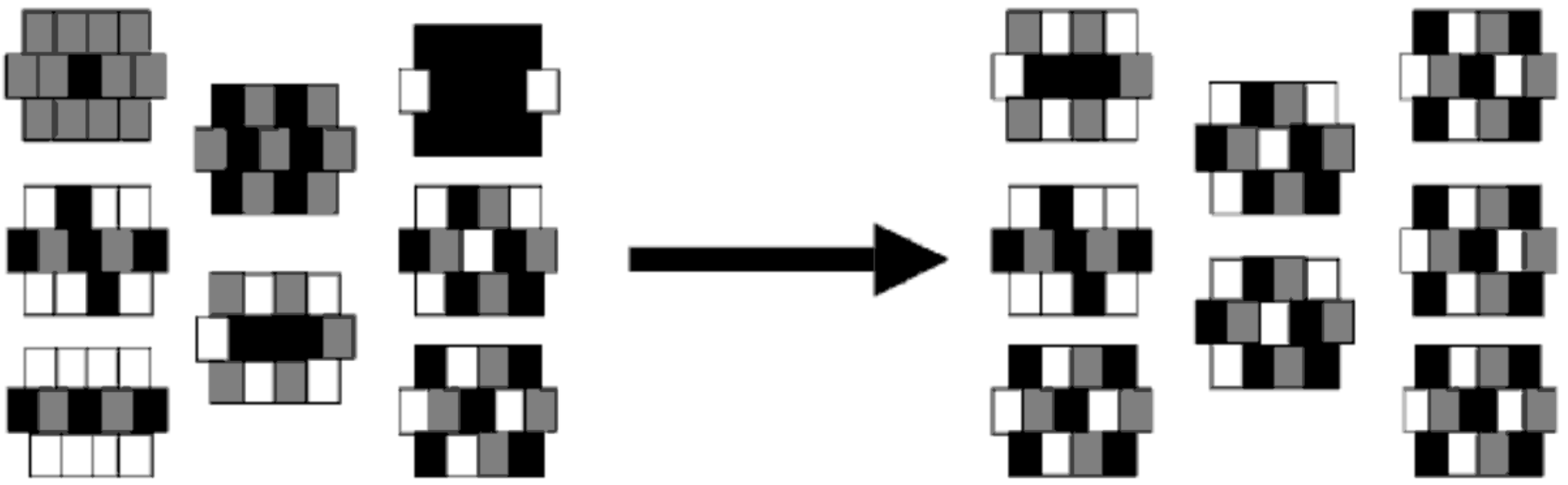


FIGURE 12.8: One generation of the GA working on the map colouring problem.

12.3.2 Punctuated Equilibrium

For a long time, one thing that creationists and others who did not believe in evolution used as an argument against it was the problem of the lack of intermediate animals in the fossil record. The argument runs that if humans evolved from apes, then there should be some evidence of a whole set of intermediary species that existed during the transition phase, and there aren't. Interestingly, GAs demonstrate one of the explanations why this is not correct, which is that the way that evolution actually seems to work is known as punctuated equilibrium. There is basically a steady population of some species for a long time, and then something changes and over a very short (in evolutionary terms... still hundreds or thousands of years) period, there is a big change, and then everything settles down again. So the chance of finding fossils from the intermediary stage is quite small. There is a graph showing this effect in Figure 12.9.

12.3.3 Example: The Knapsack Problem

We used the knapsack problem as an example while we were looking at components of the GA. It is now time to see it being solved. Before we do that, we can use some of the methods from Section 11.4 to solve it. We've

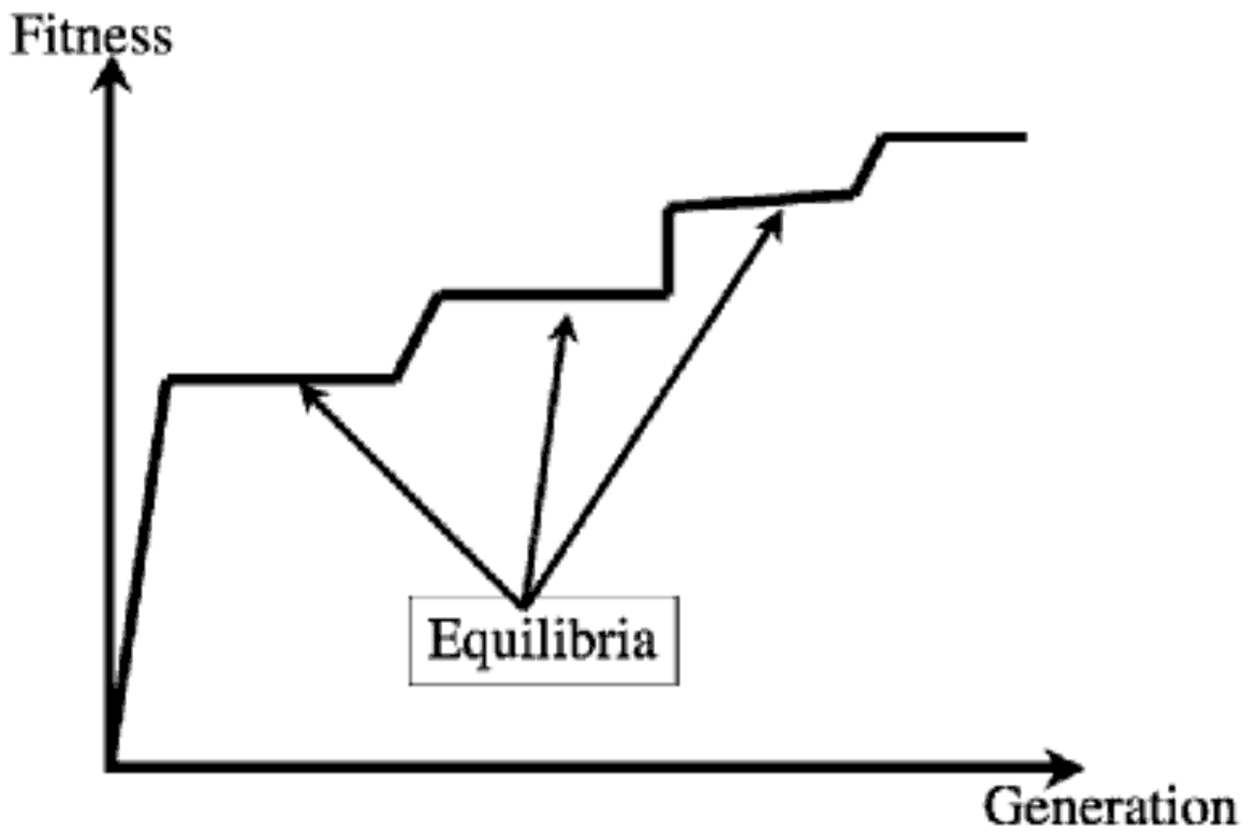


FIGURE 12.9: A graph showing punctuated equilibrium in a genetic algorithm. There is an effectively steady state where fitness does not improve, followed by rapid improvements in fitness until another steady state is reached.

already mentioned the greedy algorithm solution, and we can of course use exhaustive search, as well, or any of the other methods we discussed in the last chapter, such as simulated annealing or hillclimbing.

The website has a simple example with 20 different packages, which have a total size of 2436.77 and a maximum knapsack size of 500. The greedy algorithm finds a solution of 487.47, while the optimal solution is eventually found by the exhaustive search as 499.98. The question is how well the GA does on the same problem. We will use the fitness function that was described in Section 12.1.2, where solutions that are too large are penalised by having twice the amount they are over subtracted from the maximum size. Figure 12.10 shows a graph of the output when the GA is run on this problem for 100 iterations. The GA rapidly finds a near-optimal solution (of 499.94) to this relatively simple problem, although in this run it did not find the global optimum.

12.3.4 Example: The Four Peaks Problem

The four peaks is a toy problem (that is, simple problem that isn't useful itself, but is good for testing algorithms) that is quite often used to test out GAs and various developments of them. It is an invented fitness function that rewards strings with lots of consecutive 0s at the start of the string, and lots of consecutive 1s at the end. The fitness consists of counting the number of 0s at the start, and the number of 1s at the end and returning the maximum of them as the fitness. However, if both the number of 0s and the number of 1s are above some threshold value T then the fitness function gets a bonus of 100 added to it. This is where the name 'four peaks' comes from: there are two small peaks where there are lots of 0s, or lots of 1s, and then there

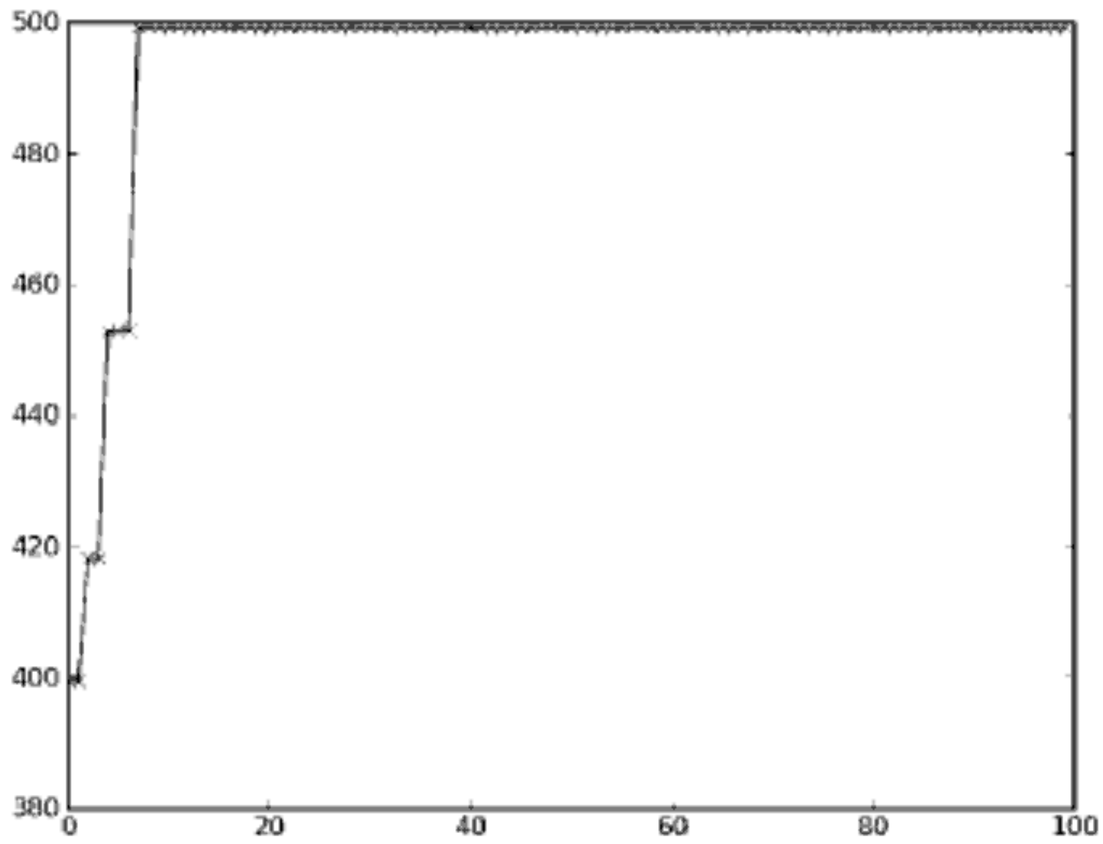


FIGURE 12.10: Evolution of the solution to the knapsack problem. The GA finds a very good solution to this simple problem within a few iterations, but never finds the optimal solution.

are two larger peaks, where the bonus is included. The GA should find these larger peaks for a successful run.

In NumPy the four peaks fitness function can be written as:

```
def fourpeaks(pop,T=10):

    start = zeros((shape(pop)[0],1))
    finish = zeros((shape(pop)[0],1))

    fitness = zeros((shape(pop)[0],1))

    for i in range(shape(pop)[0]):
        s = where(pop[i,:]==1)
        f = where(pop[i,:]==0)
        if size(s)>0:
            start = s[0][0]
        else:
            start = 0

    if size(f)>0:
        finish = shape(pop)[1] - f[-1][-1] -1
    else:
        finish = 0

    if start>T and finish>T:
        fitness[i] = maximum(start,finish)+100
```

Hidden page

guarantee that the algorithm will converge at all, and certainly not to the optimal solution. This bothers a lot of researchers. That said, genetic algorithms are widely used when other methods do not work, and they are usually treated as a black box—strings are pushed in one end, and eventually an answer emerges. This is risky, because without knowledge of how the algorithm works it is not possible to improve it, nor do you know how cautiously you should treat the results.

12.3.6 Training Neural Networks with Genetic Algorithms

We trained our neural networks, most notably the MLP, using gradient descent. However, we could encode the problem of finding the correct weights as a set of strings, with the fitness function measuring the sum-of-squares error. This has been done, and with good reported results. However, there are some problems with this approach. The first is that we turn all the local information from the targets about the error at each output node of the network into just one number, the fitness, which is throwing away useful information, and the second is that we are ignoring the gradient information, which is also throwing away useful information.

A more sensible use for GAs with neural networks is to use the GA to choose the topology of the network. Previously, we chose the structure in a completely ad hoc way by trying out different structures and choosing the one that worked best. We can use a GA for this problem, although the crossover operator doesn't make a great deal of sense, so we just consider mutation. However, we allow for four different types of mutation: delete a neuron, delete a weight connection, add a neuron, add a connection. The deletion operators bias the learning towards simple networks. Making the GA more complicated by adding extra mutation operators might make you wonder if you can make it more complicated again. And you can; one example of where this can lead is discussed next.

12.4 Genetic Programming

One extension of genetic algorithms that has had a lot of attention is the idea of **genetic programming**. This was introduced by John Koza, and the basic idea is to represent a computer program as a tree (imagine a flow chart of the code). For certain programming languages, notably LISP, this is actually a very natural way to represent a program, but it doesn't work very well in Python, so we will have a quick look at the idea, but not get into writing any explicit algorithms for the method.

Tree-based variants on mutation and crossover are defined (replace sub-

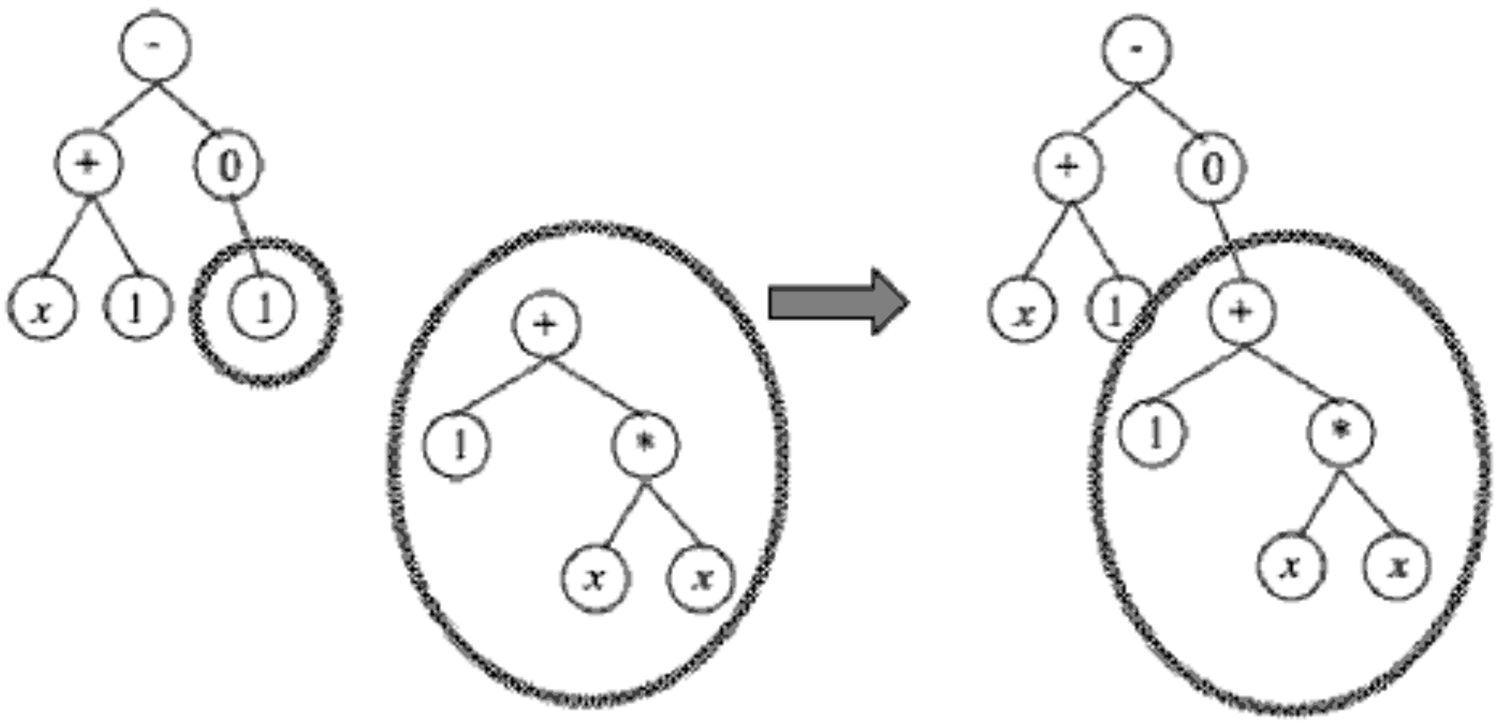


FIGURE 12.13: Example of a mutation in genetic programming.

trees by other subtrees, either randomly generated (mutation, Figure 12.13) or swapped from another tree (crossover, Figure 12.14)), and then the genetic program runs just like a normal genetic algorithm, but acting on these program trees rather than strings. Figure 12.15 shows a set of simple trees that perform arithmetic operations, and some possible developments of them, made using these operators.

Genetic programming has been used for many different tasks, from recognising skin melanomas to circuit design, and lots of very impressive results have been claimed for it. However, the search space is unbelievably large, and the mutation operator not especially useful, and so a lot depends upon the initial population. A set of possibly useful subtrees are usually chosen by the system developer first in order to give the system a head start. There are a couple of places where you can find more information on genetic programming in the Further Reading section.

12.5 Combining Sampling with Evolutionary Learning

The last machine learning method in this chapter is an interesting variation on the theme of evolutionary learning, combined with probabilistic models of the type that are described in Chapter 15, namely Bayesian networks. They are often known as estimation of distribution algorithms (EDA).

The most basic version is known as Population-Based Incremental Learning (PBIL), and it is amazingly simple. It works on a binary alphabet, just like the basic GA, but instead of maintaining a population, instead it keeps a probability vector p that gives the probability of each element being a 0 or 1. Initially, each value of this vector is 0.5, so that each element has equal

Hidden page

chance of being 0 or 1. A population is then constructed by sampling from the distribution specified vector, and the fitness of each member of the population is computed. A subset of this population (typically just the two fittest vectors) is chosen to update the probability vector, using a learning rate η, which is often set to 0.005 (where `best` and `second` represent the best and second-best elements of the population):

$$p = p \times (1 - \eta) + \eta(\text{best} + \text{second})/2. \tag{12.3}$$

The population is then thrown away, and a new one sampled from the updated probability vector. The results of using this simple algorithm on the four-peaks problem with $T = 11$ are shown in Figure 12.16 using strings of length 100 with 200 strings in each population. This is directly comparable with Figure 12.12.

The centre of the algorithm is simply the code to find the strings with the two highest fitnesses and use them to update the vector. Everything else is directly equivalent to the genetic algorithm.

```
# Pick best
best[count] = max(fitness)
bestplace = argmax(fitness)
fitness[bestplace] = 0
secondplace = argmax(fitness)

# Update vector
p  = p*(1-eta) + eta*((pop[bestplace,:]+pop[secondplace,:])/2)
```

The probabilistic model that is used in PBIL is very simple: it is assumed that each element of the probability vector is independent, so that there is no interaction. However, there is no reason why more complicated interactions between variables cannot be considered, and several methods have been developed that do exactly this. The first option is to construct a chain, so that each variable depends only on the one to its left. This might involve sorting the order of the probability vector first, but then the algorithm simply needs to measure the mutual information (see Section 6.2.1) between each pair of neighbouring variables. This use of mutual information gives the algorithm its name: MIMIC. There are also more complicated variants using full Bayesian networks, such as the Bayesian Optimisation Algorithm (BOA) and Factorised Distribution Algorithm (FDA).

The power of these developments of the GA is that they use probabilistic models and are therefore more amenable to analysis than normal GAs, which have steadfastly withstood many attempts to better understand their behaviour. They also enable the algorithm to discover correlations between input variables, which can be useful if you want to understand the solution rather than just apply it.

Hidden page

- D.E. Goldberg. *Genetic Algorithms in Search, Optimisation, and Machine Learning.* Addison-Wesley, Reading, MA, USA, 1999.

There are also entire books on genetic programming, including:

- J.R. Koza. *Genetic Programming: On the Programming of Computers by the Means of Natural Selection.* MIT Press, Cambridge, MA, USA, 1992.

- Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs.* Springer, Berlin, Germany, third edition, 1999.

For more on Estimation of Distribution algorithms, look at:

- S. Baluja and R. Caruana. Removing the genetics from the standard genetic algorithm. In A. Prieditis and S. Russel, editors, *The International Conference on Machine Learning*, pages 38–46, San Mateo, CA, USA, 1995. Morgan Kaufmann Publishers.

- M. Pelikan, D.E. Goldberg, and F. Lobo. A survey of optimization by building and using probabilistic models. *Computational Optimization and Applications*, 21(1):5–20, 2002. Also IlliGAL Report No. 99018.

Details of the two books mentioned about real evolution are:

- C. Darwin. *On the Origin of Species By Means of Natural Selection.* Wordsworth, London, UK, 6th edition, 1872.

- R. Dawkins. *The Blind Watchmaker: Why the Evidence of Evolution Reveals a Universe Without Design.* Penguin, London, UK, 1996.

Practice Questions

Problem 12.1 Suppose that you want to archive your data files, but you have only got one CD, and more data files than will fit on it. You decide to choose the files you will save so as to try to maximise the amount of space you fill on the disk, so that the most data is backed up, but you can't split a data file. Write a greedy algorithm and a hill-climbing algorithm to solve this problem. What guarantees can you make about efficiency of the solutions?

Problem 12.2 (from Jon Shapiro)
In video poker, you are dealt five cards face up. You have one chance to replace any of the cards (or all or none) with cards drawn from the

Hidden page

is the sum of the fitness for each block. Implement this fitness function and test it on strings of length 16, with blocks of lengths 1, 2, 4, 8. Run your GAs for 10,000 iterations. Compare the results to using PBIL.

Hidden page

> those which are accompanied or closely followed by discomfort to the animal will, other things being equal, have their connections with that situation weakened, so that, when it recurs, they will be less likely to occur. The greater the satisfaction or discomfort, the greater the strengthening or weakening of the bond. (E. L. Thorndike, "Animal Intelligence," page 244.)

This is where the name 'reinforcement learning' comes from, since you repeat actions that are reinforced by a feeling of satisfaction. To see how it can be applied to machine learning, we will need some more notation.

13.1 Overview

Reinforcement learning maps states or situations to actions in order to maximise some numerical reward. That is, the algorithm knows about the current input (the state), and the possible things it can do (the actions), and its aim is to maximise the reward. There is a clear distinction drawn between the agent that is doing the learning and the environment, which is where the agent acts, and which produces the state and the rewards. The most common way to think about reinforcement learning is on a robot. The current sensor readings of the robot, or processed versions of them, could define the state. They are a representation of the environment around the robot in some way. Note that the state doesn't necessarily tell us everything that it would be useful to know (the robot's sensors don't tell it its location, only what it can see about it), and there can be noise and inaccuracies in the state data. The possible ways that the robot can drive its motors are the actions, which move the robot in the environment, and the reward could be how well it does its task without crashing into things. Figure 13.1 shows the idea of state, actions, and environment to a robot, while Figure 13.2 shows how they are linked, together with the reward.

In reinforcement learning the algorithm gets feedback in the form of the reward about how well it is doing. In contrast to supervised learning, where the algorithm is 'taught' the correct answer, the reward function evaluates the current solution, but does not suggest how to improve it. Just to make the situation a little more difficult, we need to think about the possibility that the reward can be delayed, which means that you don't actually get the reward until a long time in the future (for example, think about a robot that is learning to traverse a maze. It doesn't know whether it has found the centre of the maze until it gets there, and it doesn't get the reward until it reaches the centre of the maze.) We therefore need to allow for rewards that don't appear until long after the relevant actions have been taken. Sometimes we think of the immediate reward and the total expected reward into the future.

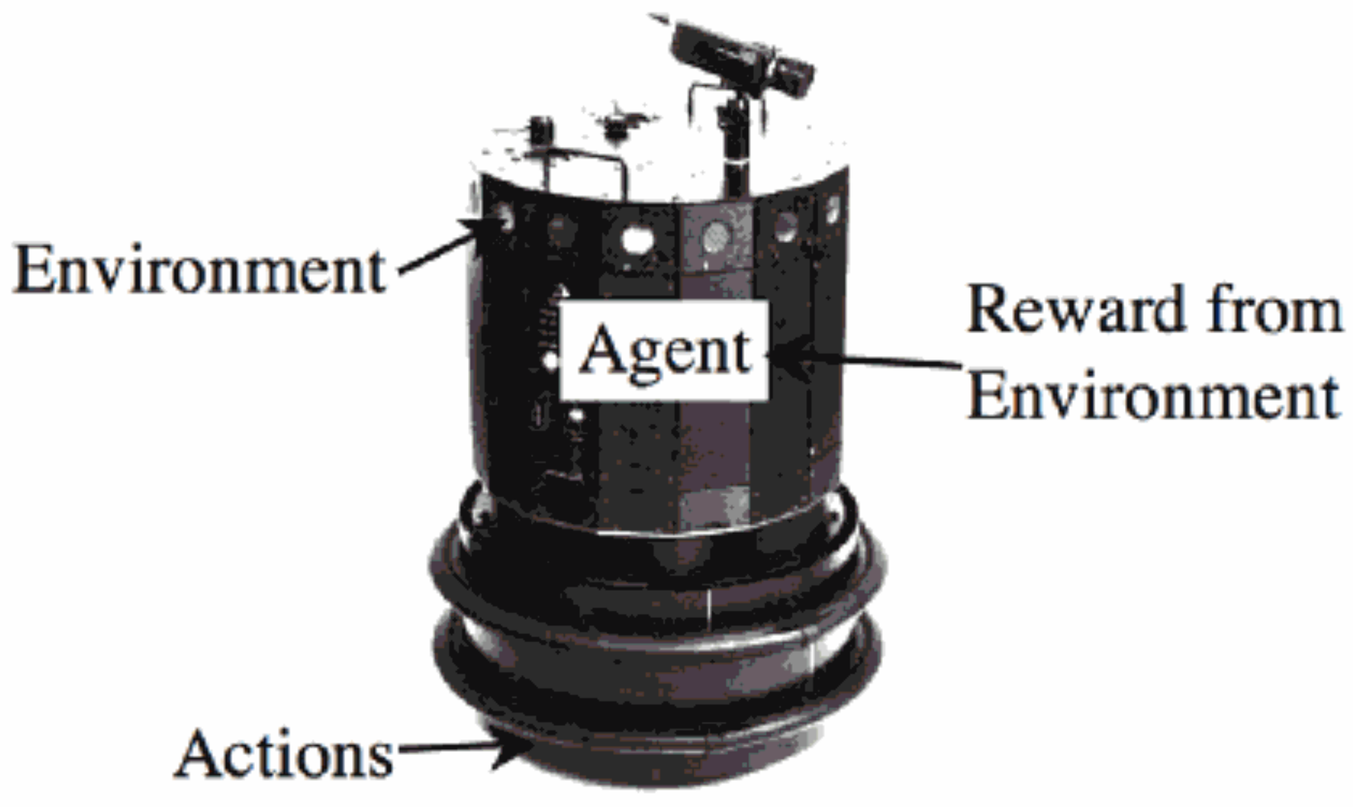


FIGURE 13.1: A robot perceives the current state of its environment through its sensors, and performs actions by moving its motors. The reinforcement learner (agent) within the robot tries to predict the next state and reward.

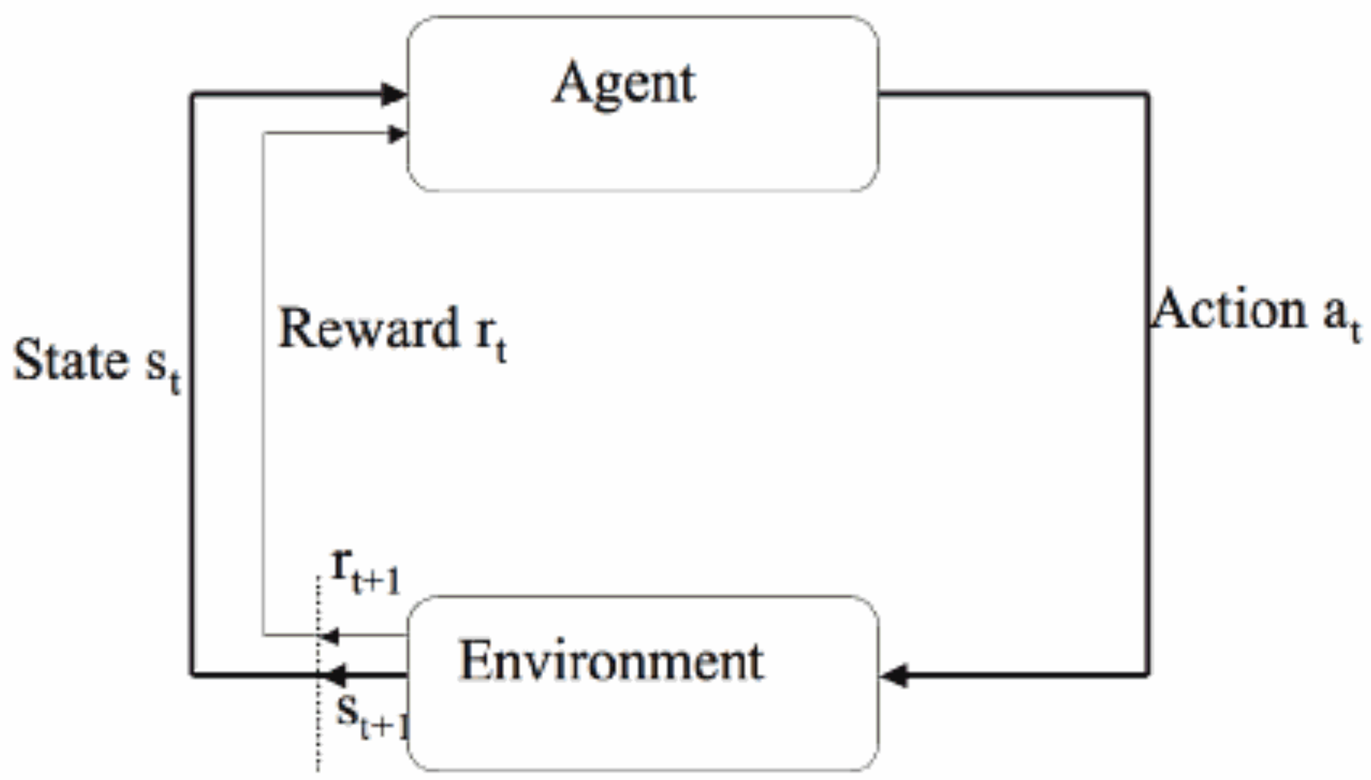


FIGURE 13.2: The reinforcement learning cycle: the learning agent performs action a_t in state s_t and receives reward r_{t+1} from the environment, ending up in state s_{t+1}.

Hidden page

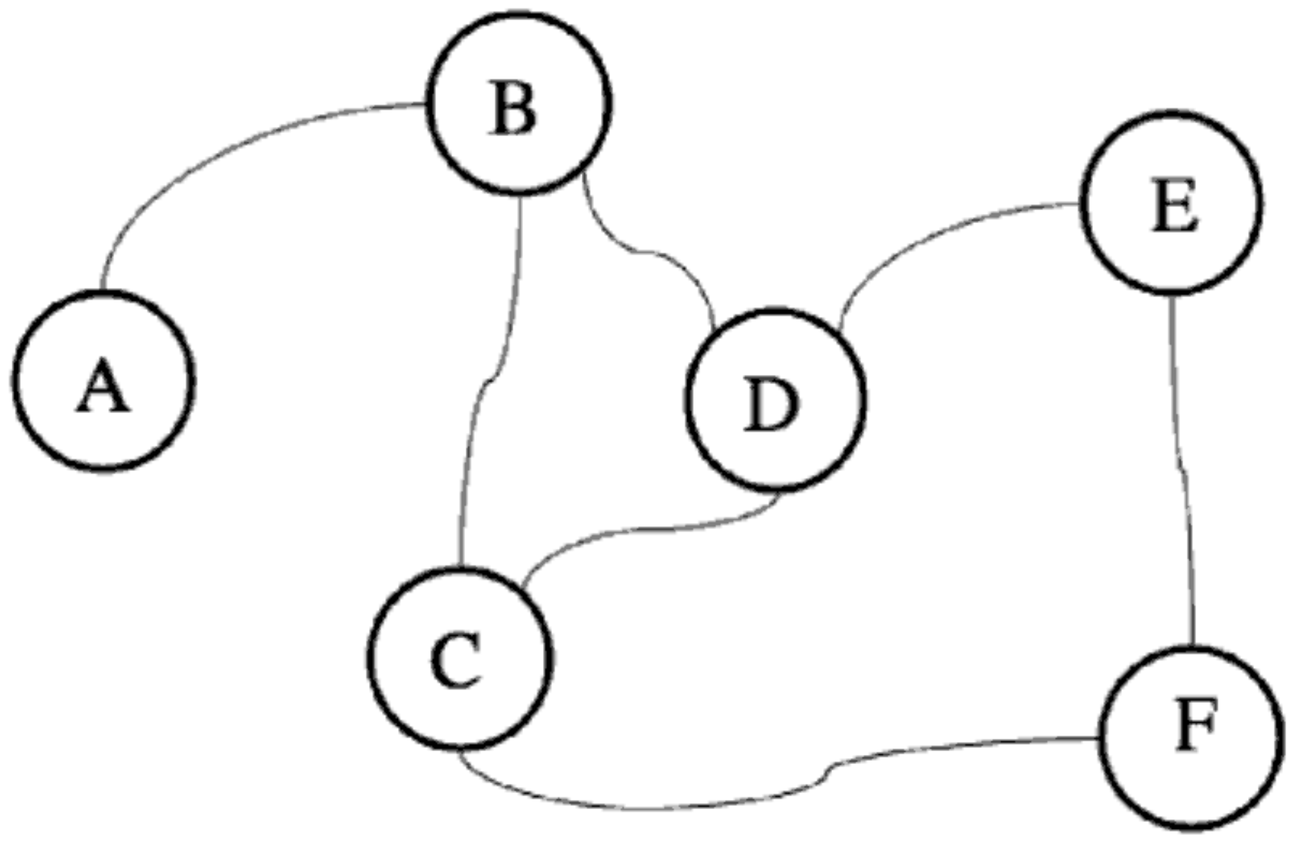


FIGURE 13.3: The old town that you find yourself lost in.

ment can be viewed as punishment, it doesn't necessarily correspond clearly, but you might want to imagine it as pinching yourself so that you stay awake). Of course, once you reach state F you are in the backpacker's and will therefore stay there. This is known as an absorbing state, and is the end of the problem, when you get the reward of eating all the chips you bought. Now moving between two squares could be good, because it might take you closer to F. But without looking at the map you won't know that, so you decide to just apply a reward when you actually reach F, and leave everything else as neutral. Where there is no direct road between two squares (so that no action takes you from one to the other) there is no reward because it is not a viable action. This results in the reward matrix R shown below (where '-' shows that there is no link) and also in Figure 13.4.

	Next State					
Current State	A	B	C	D	E	F
A	-5	0	-	-	-	-
B	0	-5	0	0	-	-
C	-	0	-5	0	-	100
D	-	0	0	-5	0	-
E	-	-	-	0	-5	100
F	-	-	0	-	0	-

Of course, as a reinforcement learner you don't actually know the reward matrix. That's pretty much what you are trying to discover, but it doesn't make for a very good example. We'll assume that you have now reached a stage of tiredness where you can't even read what is on your paper properly. Having got this set up we've reached the stage where we need to do some learning, but for now we'll leave you stranded in that foreign city and flesh out a few of the things that we've talked about so far.

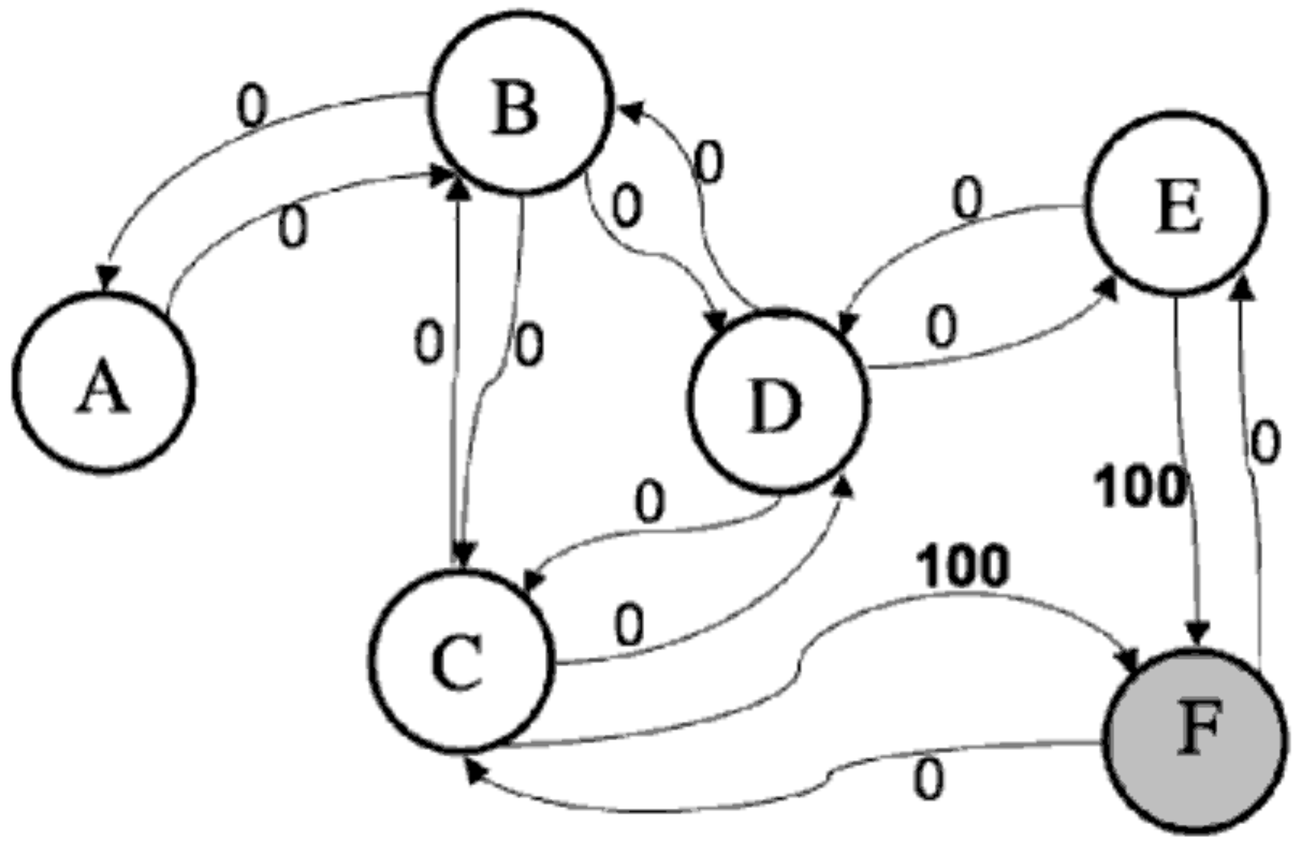


FIGURE 13.4: The *state diagram* if you are correct and the backpacker's is in square (state) F. The connections from each state back into itself (meaning that you don't move) are not shown, to avoid the figure getting too complicated. They are each worth -5 (except for staying in state F, which means that you are in the backpacker's).

13.2.1 State and Action Spaces

Our reinforcement learner is basically a search algorithm, and obviously the larger the number of states that the algorithm has to search through, the longer it will take to find a good solution. The set of all states that are possible for the learner to experience is known as the **state space**. There is a corresponding **action space** that contains all of the possible actions. If we can reduce the size of the state space and action space, then it is almost always a good idea, providing that it does not oversimplify the problem. In the example there are only six states, but still, look at Figure 13.4 and imagine wandering through all of the squares over and over again while we search: it seems like this learning is going to take a long time. And it generally does.

Computing the size of the state space (and the corresponding action space) is relatively simple. For example, suppose that there are five inputs, each an integer between 0 and 100. The size of the state space is then $100 \times 100 \times 100 \times 100 \times 100 = 100^5$, which is incredibly large, so the curse of dimensionality is really kicking in here. However, if we decide that we can **quantize** the data so that instead of 100 numbers there are only two for each input (for example, by assigning every number less than 50 to class 1, and every number 50 and above to class 2), then the size of the state space is a more manageable $2^5 = 32$. Choosing the state space and action space carefully is therefore a crucial part of making a successful reinforcement learner. You want them to be as small as possible without losing accuracy in the results—by reducing the scale of each input from 100 to 2, we have obviously thrown away a lot of information that might have made the quality of the answer better. As is usually the case, there is some element of compromise between the two.

13.2.2 Carrots and Sticks: the Reward Function

The basic idea of the learner is that it will choose the action that gets the maximum expected reward. In the example, we worked out what the rewards would be in a fairly ad hoc way, by saying what we wanted and then thinking about how to get it. That's pretty much the way that it works in practice, too: in Chapter 12 where we looked at genetic algorithms, we had to carefully craft the fitness function to solve the problem that we wanted, and the same thing is true of the reward function. In fact, they can be seen as the same thing.

The reward function takes the current state and the chosen action and produces a numerical reward based on them. So in the example, if we are in state A, and choose the action of doing nothing, so that we remain in state A, we get a reward of -5. Note that the reward can be positive or negative, with the latter corresponding to 'punishment,' showing that particular actions should be avoided. The reward is generated by the environment around the learner, it is not internal to the learner itself (this is what makes it difficult to describe in our example: the environment doesn't give you rewards in the real world, only when there is a computer (or brain) as part of the environment to help out). In effect, the reward function makes the goal of the learner explicit—the learner is trying to maximise the reward, which means behaving in exactly the way that the reward function expects. The reward tells the learner what the goal is, not how the goal should be achieved, which would be supervised learning. It is therefore usually a bad idea to include sub-goals (extra things that the learner should achieve along the way, which are meant to speed up learning), because the learner can find methods of achieving the sub-goals without actually achieving the real goal.

The choice of a suitable reward function is absolutely crucial, with very different behaviours resulting from varying the reward function. For example, consider the difference between these two reward functions for a maze-traversing robot (try to work out the difference before reading the paragraph that follows them):

- receive a reward of 50 when you find the centre of the maze
- receive a reward of -1 for each move and a reward of +50 when you find the centre of the maze

In the first version, the robot will learn to get to the centre of the maze, just as it will in the second version, but the second reward function is biased towards shorter routes through the maze, which is probably a good thing. The maze problem is episodic: learning is split into episodes that have a definite endpoint when the robot reaches the centre of the maze. This means that the rewards can be given at the end and then propagated back through all the actions that were performed to update the learner. However, there are plenty of other examples that are not episodic (continual tasks), and there is no cut

off when the task stops. An example is the child learning to walk that was mentioned at the start of the chapter. A child can walk successfully when it doesn't fall over at all, not when it doesn't fall over for 10 minutes.

Now that the reward has been broken into two parts—an immediate part and a pay-off in the end—we need to think about the learning algorithm a bit more. The thing that is driving the learning is the total reward, which is the expected reward from now until the end of the task (when the learner reaches the **terminal state** or **accepting state**—the backpacker's in our example). At that point there is generally a large pay-off that signals the successful completion of the task. However, the same thing does not work for continual tasks, because there is no terminal state, so we want to predict the reward forever into the infinite future, which is clearly impossible.

13.2.3 Discounting

The solution to this problem is known as **discounting**, and means that we take into account how certain we can be about things that happen in the future: there is lots of uncertainty in the learning anyway, so we should discount our predictions of rewards in the future according to how much chance there is that they are wrong. The rewards that we expect to get very soon are probably going to be more accurate predictions than those a long time in the future, because lots of other things might change. So we add an additional parameter $0 \leq \gamma \leq 1$, and then discount future rewards by multiplying them by γ^t, where t is the number of timesteps in the future this reward is from. As γ is less than 1, so γ^2 is smaller again, and $\gamma^k \to 0$ as $k \to \infty$ (i.e., γ gets smaller and smaller as k gets larger and larger), so that we can ignore most of the future predictions. This means that our prediction of the total future reward is:

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots + \gamma^{k-1} r_k + \ldots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}. \quad (13.1)$$

Obviously, the closer γ is to zero, the less distance we look into the future, while with $\gamma = 1$ there is no discounting, as in the episodic case above (in fact, discounting is sometimes used for episodic learning as well, since the eventual reward could be a very long way off and we have to deal with that uncertainty in learning somehow). We can apply discounting to the example of learning to walk. When you fall over you give yourself a reward of -1, and otherwise there are no rewards. The -1 reward is discounted into the future, so that a reward k steps into the future has reward $-\gamma^k$. The learner will therefore try to make k as large as possible, resulting in proper walking.

The point of the reward function is that it gives us a way to choose what to do next—our **predictions** of the reward let us exploit our current knowledge and try to maximise the reward we get. Alternatively, we can carry on exploring

Hidden page

13.2.5 Policy

We have just considered different action selection methods, such as ϵ-greedy and soft-max. The aim of the action selection is to trade off exploration and exploitation in such a way as to maximise the expected reward into the future. Instead, we can make an explicit decision that we are going to always take the optimal choice at each stage, and not do exploration any more. This choice of which action to take in each state in order to get optimal results is known as the policy, π. The hope is that we can learn a better policy that is specific to the current state s_t. This is the crux of the learning part of reinforcement learning—learn a policy π from states to actions. There is at least one optimal policy that gives the maximum reward, and that is what we want to find. In order to find a policy, there are a few things that we need to worry about. The first is how much information we need to know regarding how we got to the current state, and the second is how we ascribe a value to the current state. The first one is important enough both for here and for Chapter 15 that we are going to go into some detail now.

13.3 Markov Decision Processes

13.3.1 The Markov Property

Let's go back to the example. Standing in the square labelled D you need to make a choice of what action to take next. There are four possible options (see Figure 13.4): standing still, or moving to one of B, C, or E. The question is whether or not that is enough information for you to predict the reward accurately and so to choose the best possible action. Or do you also need to know where you have been in the past? Let's say that you know that you came to D from B. In that case, maybe it does not make sense to move back to B, since your reward won't change. However, if you came to D from E then it does actually make sense to go back there, since it moves you closer to F. So in this case, it appears that knowing your previous action doesn't actually help very much, because you don't have enough information to work out what was useful.

Another example where this is usually true is in a game of chess, where the current situation of all the pieces on the board (the state) is enough to predict whether or not the next move is a good one—it does not depend on precisely how each piece got to the current location. Thus, the current state provides enough information. A state that has this property, which is that the current state provides enough information for the reward to be computed without looking at previous states, is known as a Markov state. The importance of this can be seen from the following two equations, the first of which is what

Hidden page

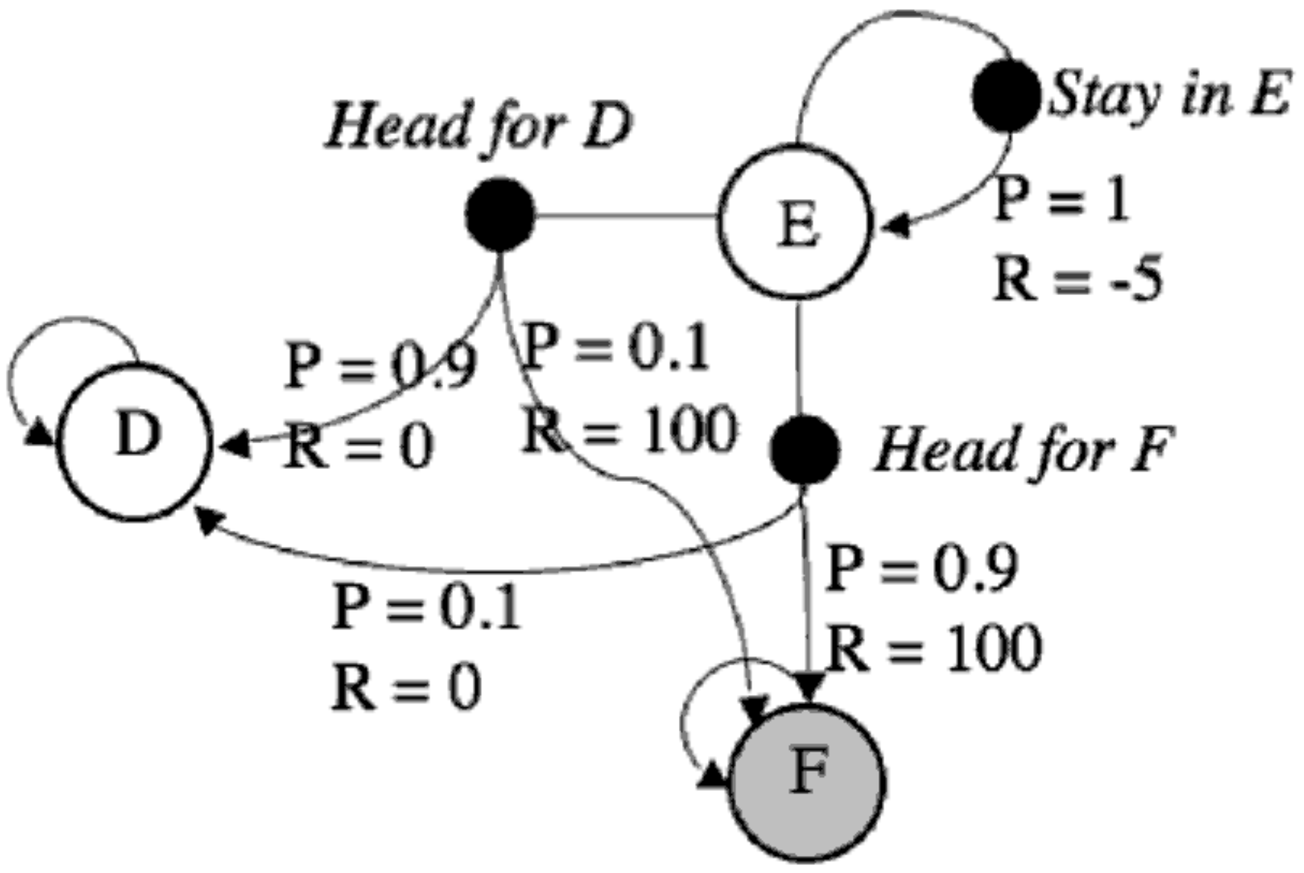


FIGURE 13.6: A small part of the transition diagram for the example. From state E there are three possible actions, and the states in which they end up, together with the rewards, are shown here.

of a finite Markov Decision Process and usually includes information about the rewards.

We can make a transition diagram for our example, based on Figure 13.4. We'll make the situation a little bit more complicated now by adding in the assumption that you are so tired that even though you are in state B and trying to get to state A, there is a small probability that you will actually take the wrong street and end up in either C or D. We'll make those probabilities be 0.1 for each extra exit that there is from each state, and we'll assume that you can stand in one place without fear of ending up elsewhere. A very tiny bit of the transition diagram, centred on state E, is shown in Figure 13.6. There are three actions that can be taken in state E (shown by the black circles), with associated probabilities and expected rewards. Learning and using this transition diagram can be seen as the aim of any reinforcement learner.

The Markov Decision Process formalism is a powerful one that can deal with additional uncertainties. For example, it can be extended to deal with the case where the true states are not known, only an observation of the state can be made, which is probabilistically related to the state, and possibly the action. These are known as partially observable Markov Decision Processes (POMDPs), and they are related to the Hidden Markov Models that we will see in Section 15.3. POMDPs are commonly used for robotics, where the sensors of the robots are usually far too inexact and noisy for places the robot visits to be identified with any degree of success. Methods to deal with these problems maintain an estimate of belief of their current state and use that in the reinforcement learning calculations. It is now time to get back to the reinforcement learner and the concept of values.

13.4 Values

The reinforcement learner is trying to decide on what action to take in order to maximise the expected reward into the future. This expected reward is known as the value. There are two ways that we can compute a value. We can consider the current state, and average across all of the actions that can be taken, leaving the policy to sort this out for itself (the state-value function, $V(s)$), or we can consider the current state and each possible action that can be taken separately, the action-value function, $Q(s, a)$. In either case we are thinking about what the expected reward would be if we started in state s (where $E(\cdot)$ is the statistical expectation):

$$V(s) = E(r_t|s_t = s) = E\left\{\sum_{i=0}^{\infty} \gamma^i r_{t+i+1}|s_t = s\right\}, \tag{13.5}$$

$$Q(s, a) = E(r_t|s_t = s, a_t = a) = E\left\{\sum_{i=0}^{\infty} \gamma^i r_{t+i+1}|s_t = s, a_t = a\right\}. \tag{13.6}$$

It should be fairly obvious that the second estimate is more accurate in the long run, because we have more information: we know which action we are going to take. However, because of that we need to collect a lot more data, and so it will take a long time to learn. In other words, the action-value function is even more susceptible to the curse of dimensionality than the state-value function. In situations where there are lots of states it will not be possible to store either, and some other method, such as using a parameterised solution space (i.e., having a set of parameters that are controlled by the learner, rather than explicit solutions), will be needed. This is more complicated than we will consider here.

There are now two problems that we need to solve, predicting the value function, and then selecting the optimal policy. We'll think about the second one first. The optimal policy is the one in which the value function is the greatest over all possible states. We label this (not necessarily unique) policy with a star: π^*. The optimal state-value function is then $V^*(s) = \max_\pi V^\pi(s)$ for all possible states s, and the optimal action-value function is $Q^*(s, a) = \max_\pi Q^\pi(s, a)$ for all possible states s and actions a. We can link these two value functions, because the first considers taking the optimal action in each case (since the policy π^* is optimal), while the second considers taking action a this time, and then following the optimal policy from then on. Hence we only need to worry about the current reward and the (discounted) estimate of the future rewards:

$$Q^*(s,a) = E(r_{t+1}) + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1})$$
$$= E(r_{t+1}) + \gamma V^*(s_{t+1}|s_t = s, a_t = a). \quad (13.7)$$

Of course, there is no guarantee that we will ever manage to learn the optimal policy. There is bound to be noise and other inaccuracies in the system, and the curse of dimensionality is liable to limit the amount of exploration that we do. However, it will be enough to learn good approximations to the optimal policy. One thing that will work to our advantage is that reinforcement learning operates on-line, in that the learner is exploring the different states as it learns, which means that it is getting more chances to learn about the states that are seen more often, and so will have a better chance of finding the optimal policy for those states.

The question is how you actually update the value function ($V(s)$ or $Q(s,a)$). The idea is to make a look-up table of all the possible states or state-action pairs, and set them all to zero to start with. Then we will use experience to fill them in. Returning to your foreign trip, you wander around until eventually you stumble upon the backpacker's. Gorging yourself on the chips you remember that at the last timestep you were in square E (this remembering is known as a backup). Now you can update the value for E (the reward is $\gamma \times 100$). That is all we do, since your memory is so shot that you can't remember anything else. And there we stop until the next night, when you wake up again and the same thing happens. Except now, you have information about E, although not about any other state. However, when you reach E now, say from D, then you can update the value for D to have reward $\gamma^2 \times 100$. And so it continues, until all of the states (and possibly actions) have values attached.

The obvious problem with this approach is that we have to wait until we reach the goal before we can update the values. Instead, we can use the same trick that we used in Equation (13.7) and use the current reward and the discounted prediction instead, so that the update equation looks like (where μ is the learning rate as usual):

$$V(s_t) \leftarrow V(s_t) + \mu(r_{t+1} + \gamma V(s_{t+1}) - V(s_t)). \quad (13.8)$$

The $Q(s,a)$ version looks very similar, except that we have to include the action information. In both cases we are using the difference between the current and previous estimates, which is why these methods have the name of temporal difference (TD) methods. Suppose that we knew rather more about where we had been. In that case, we could have updated more states when we got to the backpacker's, which would have been rather more efficient. The trouble is that we don't know if those states were useful or not—it might have been chance that we visited them. The answer to this is similar to discounting: we introduce another parameter $0 \leq \lambda \leq 1$ that we apply to

Hidden page

Hidden page

Hidden page

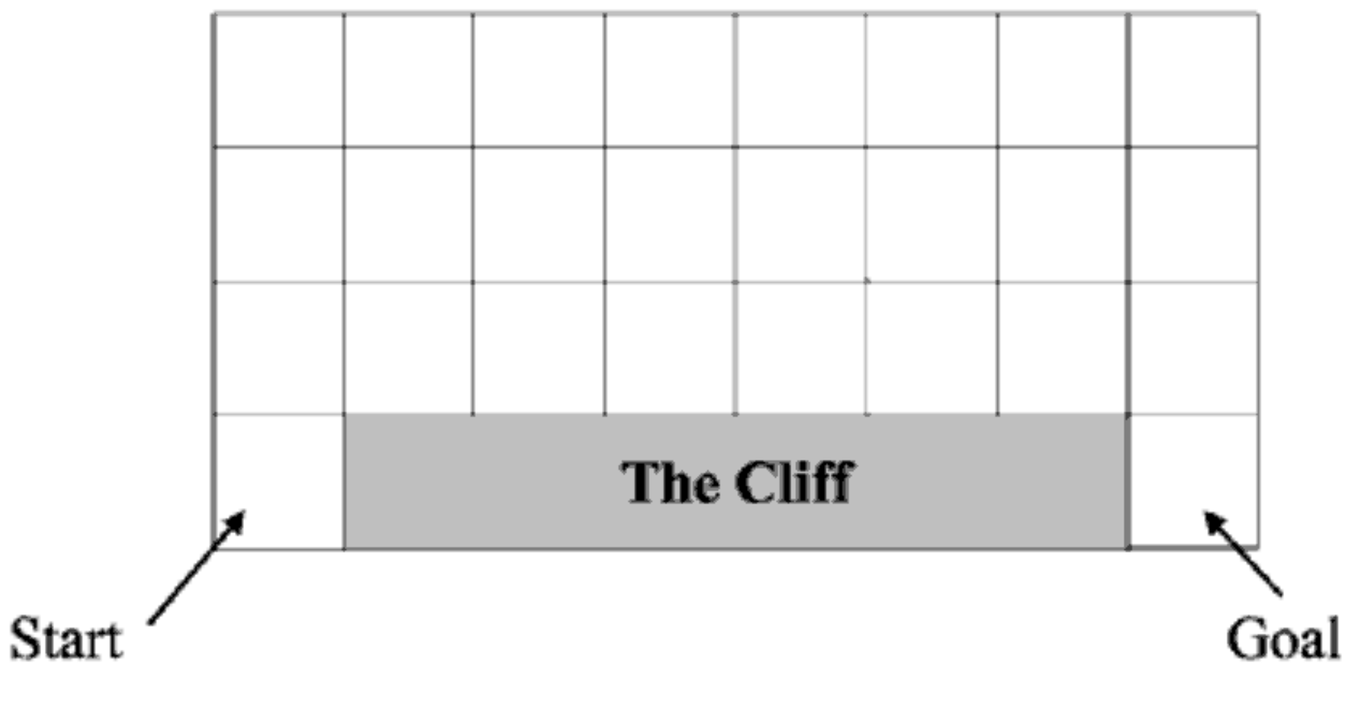


FIGURE 13.7: The example environment.

13.6 The Difference between Sarsa and Q-Learning

It might not be clear what the difference is between the two algorithms in practice. We're going to consider the little environment that is shown in Figure 13.7, where the agent has to learn a route from the start location on the left to the final location on the right (the example comes from Section 6.5 of Sutton and Barto's book, which is in the readings at the end of the chapter). The reward structure is that every move gets a reward of -1, except for moves that end up on the cliff. These get a reward of -100, and the agent gets put back at the start location. This is clearly an episodic problem, since there is a clear end state.

Both algorithms will start out with no information about the environment, and will therefore explore randomly, using the ϵ-greedy policy. However, over time, the strategies that the two algorithms produce are quite different. The main reason for the difference is that Q-learning always attempts to follow the optimal path, which is the shortest one. This takes it close to the cliff, and the ϵ-greedy part means that inevitably it will sometimes fall over. By way of contrast, the sarsa algorithm will converge to a much safer route that keeps it well away from the cliff, even though it takes longer. The two solutions are shown in Figures 13.8 and 13.9. The sarsa algorithm produces the safe route because it includes information about action selection in its estimates of Q, while Q-learning produces the riskier, but shorter, route. The choice of which is better is up to you, and it depends on how serious the effects of falling off the cliff are.

The reason for the difference between the algorithms is because Q-learning always assumes that the policy will pick the optimal action, and while this is true most of the time, the ϵ-greedy policy does occasionally choose a different action, which can cause problems here. However, the algorithm ignores these dangers because it only focuses on the optimal solution. Sarsa does not take this maximum, and so it will be biased against solutions that take it close to

Hidden page

A famous example of reinforcement learning was TD-Gammon, which was produced by Gerald Tesauro. His idea was that reinforcement learning should be very good at learning to play games, because games were clearly episodic—you played until somebody won—and there was a clear reward structure, with a positive reward for winning. There was another benefit, which was that you could set the learner to play against itself. This is actually very important, since the version of TD-Gammon that was actually bundled with the IBM operating system OS/2 Warp had played 1,500,000 games against itself before it stopped improving.

Further Reading

A detailed book on reinforcement learning is:

- R.S. Sutton and A.G. Barto. *Reinforcement Learning: An Introduction.* MIT Press, Cambridge, MA, USA, 1998.

An interesting article concerning the use of reinforcement learning is:

- G. Tesauro. Temporal difference learning and TD-gammon. *Communications of the ACM*, 38(3):58–68, 1995.

Alternative treatments are:

- Chapter 13 of T. Mitchell. *Machine Learning.* McGraw-Hill, New York, USA, 1997.
- Chapter 16 of E. Alpaydin. *Introduction to Machine Learning.* MIT Press, Cambridge, MA, USA, 2004.

Practice Questions

Problem 13.1 Work through the first few steps of the hill by hand for both sarsa and Q-learning. Then modify the code to run on this example and ensure that they match.

Problem 13.2 Design a Q-learner for playing noughts-and-crosses (also known as Tic-Tac-Toe). Run the algorithm by hand, describing the states, transitions, rewards, and Q-values. Assume that the opponent picks a random (but valid) square for each move. How would your learner change if the opponent played optimally? Would a TD learner behave differently?

Hidden page

in each row, and 7 again for 2 counters being on the board. However, from there the number of states mushrooms. In the case where the game is a draw, so that all of the squares are full, there is something less than $2^{7\times6} = 2^{42}$ states. I say something less because this counts all the cases that include a line of 4, and also ignores the fact that there are only 21 counters of each colour. The fact remains that the state space is immense, so it is probably going to take a long time to learn.

However, programming the game is relatively simple. There are two absorbing states: when the board is full, and when somebody wins. In either of these cases a reward is given. So you will have to decide on rewards, and write some code that detects when one or the other state has happened. The choice that is made at each turn is simply which column to add the new counter in, so there are only seven possible actions. You need to represent the board, for which I'd recommend a 2D array with 0 meaning empty, 1 meaning contains a red counter, and 2 meaning contains a yellow counter. This should make it easy to detect the absorbing states.

Having set up that lot, you need to make a number of modifications to the Q-learning code. Firstly, you are not going to pass in transition and reward matrices, since making them would be crazy. You are probably going to give a reward of 0 to every move except a win and a loss, so change the code to present those rewards. You then need to change the ϵ-greedy search strategy to simply pick a random (but not full) column, rather than look at the transition matrix. Then that's it, set it running (and be prepared to wait for a very long time. I trained the algorithm for 20,000 games against a purely random player, and at the end of that the Q-learner was winning about 80% of the games).

Chapter 14

Markov Chain Monte Carlo (MCMC) Methods

In this chapter we are going to look at a method that has revolutionised statistical computing and statistical physics over the past 20 years. The principal algorithm has been around since 1953, but only when computers became fast enough to be able to perform the computations on real world examples in hours instead of weeks did the methods become really well known. However, this algorithm has now been cited as one of the most influential ever created.

There are two basic problems that can be solved using these methods, and they are the two that we have been wrestling with for pretty much the entire book: we may want to compute the optimum solution to some objective function, or compute the posterior distribution of a statistical learning problem. In either case the state space may well be very large, and we are only interested in finding the best possible answer—the steps that we go through along the way are not important. We've seen several methods of solving these types of problems during the book, and here we are going to look at one more. We will see a place where MCMC methods are very useful in Section 15.1.

The idea behind everything that we are going to talk about in this chapter is that as we explore the state space, we can also construct samples as we go along in such a way that the samples are likely to come from the most probable parts of the state space. In order to see what this means, we will discuss what Monte Carlo sampling is, and look at Markov chains.

14.1 Sampling

We have produced samples from probability distributions in almost all of the algorithms we have looked at, for example, for initialisation of weights. In many cases, the probability distribution we have used has been the uniform one on $[0, 1)$, and we have done it using the `random.rand()` function in NumPy, although we have also seen sampling from Gaussian distributions using `random.normal()`.

14.1.1 Random Numbers

The basis of all of these sampling methods is in the generation of random numbers, and this is something that computers are not really capable of doing. However, there are plenty of algorithms that produce pseudo-random numbers, the simplest of which is the linear congruential generator. This is a very simple function that is defined by a recurrence relation (i.e., you put one number in to get the second number, and then feed that back in to get the third, and then repeat the cycle):

$$x_{n+1} = (ax_n + c) \mod m, \tag{14.1}$$

where a, c, and m are parameters that have to be chosen carefully. All of them, and the initial input x_0 (which is known as the seed), are integers, and so are all of the outputs. The modulus function means that the largest number that can be produced is m, and so there are at most m numbers that can be produced by the algorithm. Once one number appears a second time, the whole pattern will repeat again since the equation only uses the current output as input. The length of the sequence between repeats is the period, and it should obviously be as long as possible, since it is the most obvious non-randomness in the algorithm. There has been a lot of investigation of choices of the parameters so that the period is length m, so that every integer between 0 and m is produced before the pattern cycles. There are various choices of the parameters that have been selected to work well, including $m = 2^{32}$; $a = 1,664,525$; and $c = 1,013,904,223$. Clearly, just picking numbers at random isn't going to be that useful.

There has been a lot of effort put into different random number generators, since they are important not just for statistical computing, but also cryptography and security. The industry-standard algorithm for generating random samples is the Mersenne Twister, which is based on Mersenne prime numbers. It is the random number generator used in NumPy. No matter what algorithm generates the numbers, though, it is important to remember that they are not genuinely random, and as John von Neumann, one of the fathers of modern computing, stated:

> Anyone who considers arithmetic methods of producing random digits is, of course, in a state of sin.

The other troublesome thing about random numbers is that it is not actually possible to prove that a sequence of numbers are truly random. There are several tests that can be made of a sequence of numbers to see if they seem to be random. Examples include calculating the entropy of the sequence (entropy was described in Section 6.2.1), using a compression algorithm on the sequence (since compression algorithms exploit redundancy, i.e., predictability, in the input, if the compression algorithms fail to make the input smaller, then it might be because they are random), and just checking how many numbers

are odd compared to even. However, you can never guarantee that a sequence is random, just that it hasn't failed most of the tests yet (but just because it fails one or two of them at some point in the sequence doesn't mean that the sequence isn't random; truly random numbers can look deterministic for a long time... this is part of the joy of randomness!). I'll leave the last word on this to von Neumann again:

> In my experience it was more trouble to test random sequences than to manufacture them.

14.1.2 Gaussian Random Numbers

The Mersenne twistor produces uniform random numbers. However, often we might want to produce samples from other distributions, e.g., Gaussian. The usual method of doing this is the Box-Muller scheme, which uses a pair of uniformly randomly distributed numbers in order to make two independent Gaussian-distributed numbers with zero mean and unit variance. There are a few ways to implement this; we will look at the more efficient polar form of the method, which works as follows:

The Box-Muller Scheme

- Pick two uniformly distributed random numbers between -1 and 1 (x_1, x_2) (for example, use `random.rand(2)*2-1`)
- If $x_1^2 + x_2^2 > 1$, discard them and pick two more (this means that $p(x_1, x_2) = \frac{1}{\pi}$, since they are inside the unit circle)
- Compute $w = x_1^2 + x_2^2$
- Compute $y_1 = x_1 \left(\frac{-2 \ln w}{w}\right)^{\frac{1}{2}}$ and similarly for y_2
- These y_i have probability:

$$p(y_1, y_2) = p(x_1, x_2) \left(\frac{\partial(x_1, x_2)}{\partial(y_1, y_2)}\right) = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{1}{2} y_1^2\right) \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{1}{2} y_2^2\right) \tag{14.2}$$

  which describes two independent variables with zero mean, unit variance Gaussian distribution

---

A plot of 1,000 samples created by the Box-Muller scheme along with the zero mean, unit variance Gaussian line is shown in Figure 14.1. There is a more efficient algorithm for computing Gaussian-distributed random numbers known as the Ziggurat algorithm.

There may well be many other distributions that we want to sample from. For common statistical distributions people have worked out schemes like the

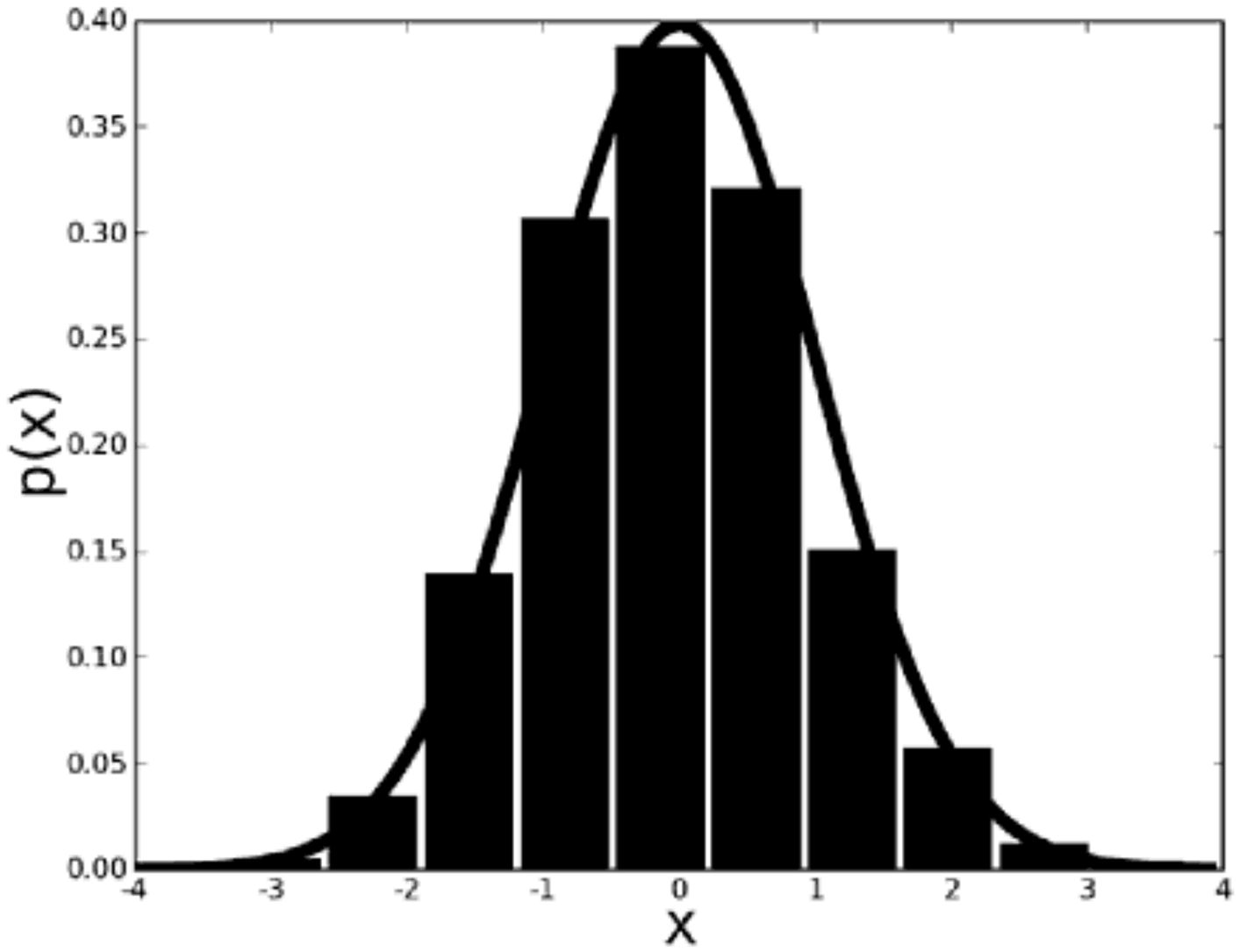


FIGURE 14.1: Histogram of 1,000 Gaussian samples created by the Box-Muller scheme. The line gives the Gaussian distribution with zero mean and unit variance.

Box-Muller scheme, but we might want to sample from distributions that we can't describe in those terms. We will see examples of this in Chapter 15. We would like a method of sampling from a distribution that doesn't have to be tailored to the distribution. There is one important concept that can be seen in the Box-Muller scheme, and that is the idea of rejection. When the original samples were not inside the unit circle they were rejected and another one computed to replace them. This is a bit like simulated annealing as we saw it in Section 11.6: we constructed a possible solution and then decided whether or not to use it. Rejection adds computational cost to the procedure, since if we were unlucky this algorithm could run for a long time before it found a pair of numbers that satisfied the criteria. However, it also means that we find samples that satisfy our requirements without having to design any tricky code, and it is generally faster as well, since the computational cost of generating some random numbers is rather less than the cost of doing the complicated transform. We are going to see rejection used a lot more in this chapter, but before we get there, we should set the idea of sampling on to a proper theoretical footing.

Hidden page

$$\hat{\mathbf{x}} = \arg\max_{\mathbf{x}^{(i)}} p\left(\mathbf{x}^{(i)}\right). \tag{14.5}$$

Allegedly, the idea of Monte Carlo sampling (and the reason that it got its name) first came about when Stan Ulam was considering the probabilities of particular hands of cards. In fact, the whole of probability theory was originally developed by some of the great French mathematicians, such as Fermat, in order to reason about games of chance, so Monte Carlo sampling is in pretty good company. Suppose that you want to do something relatively simple, such as to predict how many times you should expect to win at the patience game that came free with your computer. All you need to do is work out the rules for when you win based on the initial setup, and then look at how many of these setups there are. In a standard deck there are 52 cards, so there are 52! ($\approx 8 \times 10^{67}$) different ways in which the cards can be distributed. So even before we start thinking about the specific rules for the game, we know that the number of different layouts is so large it is basically impossible to think about. Despairing of working it out, you might decide to play a couple of hands of patience and see how well you do. In fact, the Monte Carlo principle tells you that that is exactly what you should be doing. Suppose that you play ten games of patience and six of them come out. You might be able to argue that approximately 60% of the patience games will do well. To believe this, you will have to play far more than ten games with the same success rate, or course; and even then it assumes that you are a good patience player, and don't cheat.

14.3 The Proposal Distribution

We now have everything that we need if the distribution $p(\mathbf{x})$ that we are sampling from is easy (that is, not computationally expensive) to sample from. Unfortunately, this is very rarely the case, but fortunately there is a way to get around this problem, which is to cheat by inventing a simpler distribution $q(\mathbf{x})$ that we can sample from easily, and picking samples from there. Obviously we can't just pick any distribution $q(\mathbf{x})$, there has to be some relation between them. So we assume that even though we don't know $p(\mathbf{x})$, we can evaluate $\tilde{p}(\mathbf{x})$, for a given $\mathbf{x}$ where:

$$p(\mathbf{x}) = \frac{1}{Z_p}\tilde{p}(\mathbf{x}), \tag{14.6}$$

where Z_p is some normalisation constant that we don't know. This is not usually an unreasonable assumption; we are not saying that we do not know $p(\mathbf{x})$, just that we can't sample from it easily. Now we can pick a number M so that $\tilde{p}(\mathbf{x}) \leq Mq(\mathbf{x})$ for all values of $\mathbf{x}$. We generate a random number $\mathbf{x}^*$

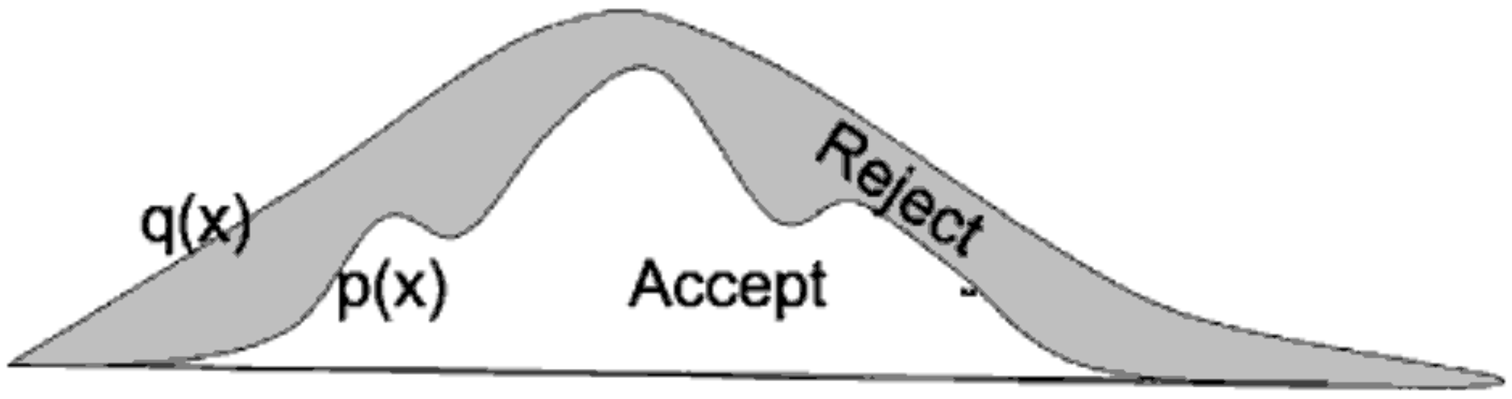


FIGURE 14.2: The proposal distribution method.

from $q(\mathbf{x})$, and we want this to look like a sample from $p(\mathbf{x})$. We therefore turn to the idea of rejection again, looking at how likely it is that the sample comes from $p(\mathbf{x})$, and discarding it if it turns out to be unlikely.

We make the decision of whether or not to accept the sample by picking a uniformly distributed random number u between 0 and $Mq(\mathbf{x}^*)$. If this random number is less than $\tilde{p}(\mathbf{x}^*)$, then we reject $\mathbf{x}^*$, otherwise we keep it. The reason why this works is known as the envelope principle: the pair $(\mathbf{x}^*, u)$ is uniformly distributed under $Mq(\mathbf{x}^*)$, and the rejection part throws away samples that don't match the uniform distribution on $p(\mathbf{x}^*)$, so $Mq(\mathbf{x})$ forms an envelope on $p(\mathbf{x})$. Figure 14.2 shows the idea. We sample from $Mq(\mathbf{x})$ and reject any sample that lies in the grey area. The smaller M is, the more samples we get to keep, but we need to ensure that $\tilde{p}(\mathbf{x}) \leq Mq(\mathbf{x})$. This method is known as rejection sampling, and the algorithm can be written as:

The Rejection Sampling Algorithm

- Sample $\mathbf{x}^*$ from $q(\mathbf{x})$ (e.g., using the Box-Muller scheme if $q(\mathbf{x})$ is Gaussian)
- Sample u from uniform$(0, \mathbf{x}^*)$
- if $u < p(\mathbf{x}^*)/Mq(\mathbf{x}^*)$:
  - add $\mathbf{x}^*$ to the set of samples
- else:
  - reject $\mathbf{x}$ and pick another sample

As an example of using rejection sampling, Figure 14.3 shows the results of using it to sample from the mixture of two Gaussians by using the uniform distribution shown by the dotted line. Using $M = 0.8$, as shown in the figure, the algorithm rejects about half of the samples. Using $M = 2$ the algorithm rejects about 85% of samples. So with rejection sampling, you have to throw away samples, and if you don't pick M properly, you will have to reject a lot of them. The curse of dimensionality makes the problem even worse. There are

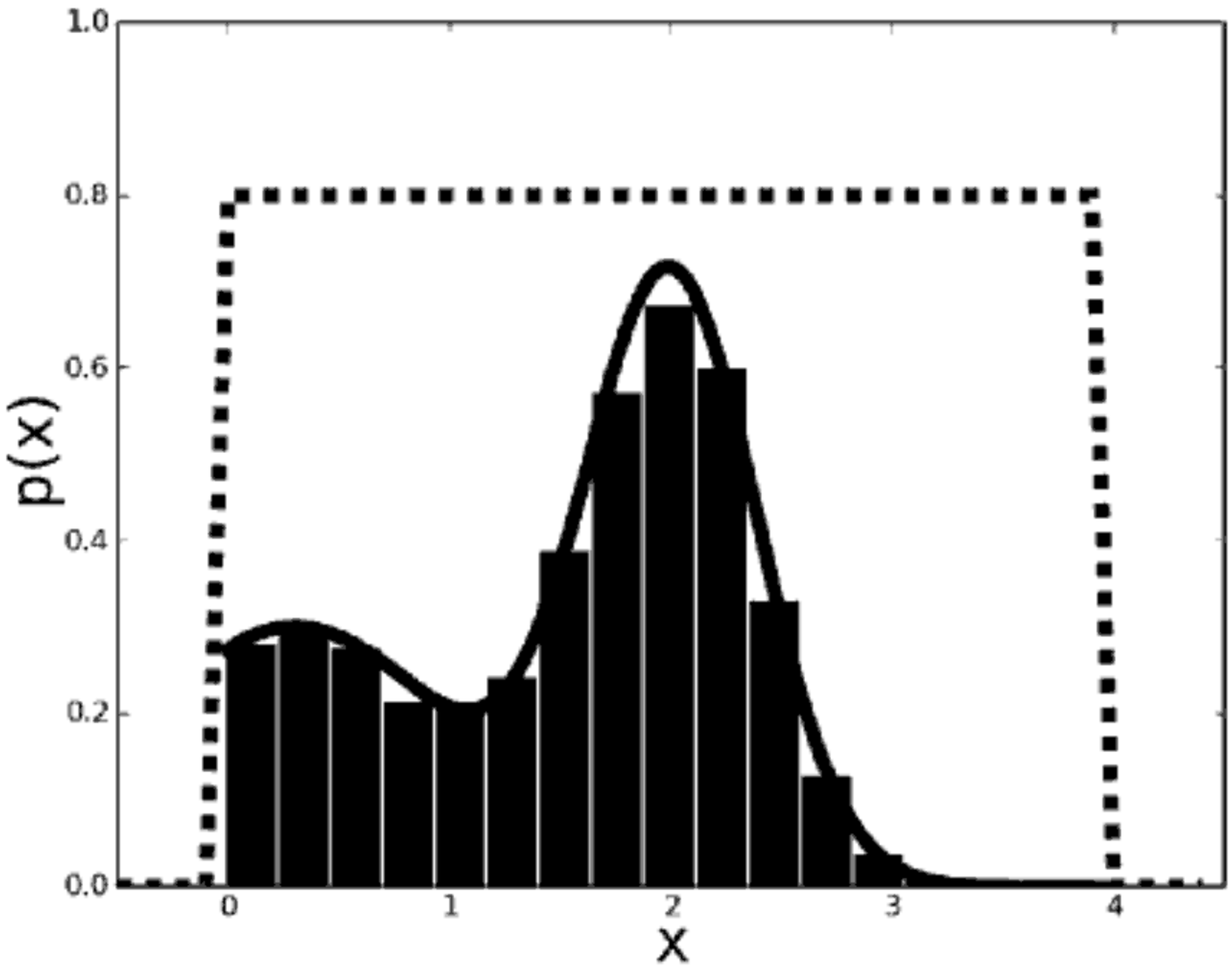


FIGURE 14.3: The histogram shows samples of a mixture of two Gaussians (given by the solid line) as sampled from the uniform box shown as a dotted line by using rejection sampling.

two things that we can do to get over this problem. One is to develop some more sophisticated methods of understanding the space that we are sampling, and the other is to try to ensure that the samples are taken from areas of the space that have high probability.

The reason why we are using these methods at all is that we can't sample from the actual distribution we want, since that is too difficult and/or expensive, but it might be possible to understand it in other ways. In Section 14.4.1 we will look at methods that allow us to travel around within the space by using simple local moves. Before we get to that we will look at a method that tries to ensure that the samples come from regions of high probability. The method is known as importance sampling, because it attaches a weight that says how important each sample is.

Suppose that we want to compute the expectation of a function $f(\mathbf{x})$ for a continuous random variable $\mathbf{x}$ distributed according to unknown distribution $p(\mathbf{x})$. Starting from the expression of the expectation that we wrote out earlier, we can introduce another distribution $q(\mathbf{x})$:

$$
\begin{aligned}
E(f) &= \int p\left(\mathbf{x}\right) f\left(x\right) d\mathbf{x} \\
&= \int p(\mathbf{x}) f(\mathbf{x}) \frac{q(\mathbf{x})}{q(\mathbf{x})} d\mathbf{x} \\
&\approx \frac{1}{N} \sum_{i=1}^{N} \frac{p\left(\mathbf{x}^{(i)}\right)}{q\left(\mathbf{x}^{(i)}\right)} f\left(x^{(i)}\right), \qquad (14.7)
\end{aligned}
$$

where we have used the fact that $q(\mathbf{x})$ is the density of a random variable, and so if we perform $\int q(\mathbf{x}) d\mathbf{x}$ over all values of $\mathbf{x}$ then it must equal 1. The ratio $w(\mathbf{x}^{(i)}) = p(\mathbf{x}^{(\mathbf{i})})/q(\mathbf{x}^{(\mathbf{i})})$ is called the importance weight, and it corrects for sampling from the grey region in Figure 14.2 without having to reject samples. While this can be used to estimate the expectation directly, the real benefit of computing the importance weights is that they can be used in order to resample the data. This leads to an algorithm known descriptively as Sampling-Importance-Resampling. In the words of the advert, it 'does exactly what it says on the tin':

The Sampling-Importance-Resampling Algorithm

- Produce N samples $\mathbf{x}^{(i)}$, $i = 1 \ldots N$ from $q(\mathbf{x})$
- Compute normalised importance weights

$$
w^{(i)} = \frac{p(\mathbf{x}^{(i)})/q(\mathbf{x}^{(i)})}{\sum_j p(\mathbf{x}^{(j)})/q(\mathbf{x}^{(j)})} \qquad (14.8)
$$

- Resample from the distribution $\{\mathbf{x}^{(i)}\}$ with probabilities given by the weights $w^{(i)}$

An implementation of this in Python is shown next, and the results of using sampling-importance-resampling on the example in Figure 14.3 is given in Figure 14.4. Note that this method does not reject any samples, but it does involve two separate sampling steps and a relatively expensive loop. Like the other algorithms we have seen, it is sensitive to the quality of the match between the proposal distribution $q(\mathbf{x})$ and the actual distribution $p(\mathbf{x})$.

```
# Sample from q
sample1 = random.rand(n)*4

# Compute weights
w = p(sample1)/q(sample1)
w /= sum(w)
```

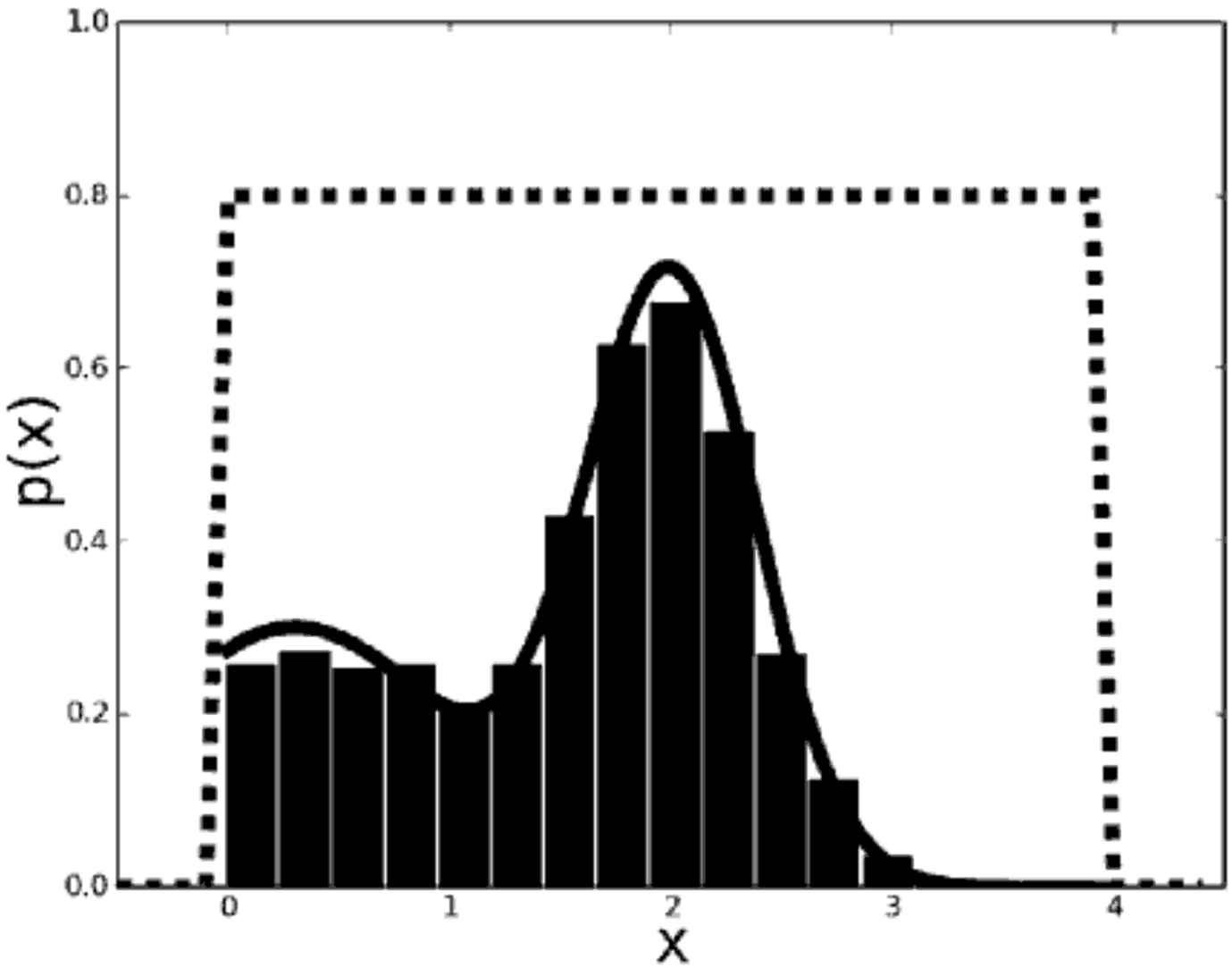


FIGURE 14.4: The histogram shows samples created using sampling-importance-resampling from a mixture of two Gaussians (given by the solid line) as sampled from the uniform box shown as a dotted line.

```
# Sample from sample1 according to w
cumw = zeros(n)
cumw[0] = w[0]
for i in range(1,n):
    cumw[i] = cumw[i-1]+w[i]

u = random.rand(n)

index = 0
for i in range(n):
    indices = where(u<cumw[i])
    sample2[index:index+size(indices)] = sample1[i]
    index += size(indices)
    u[indices]=2
```

In Section 15.4.2 we will see a method that uses sampling-importance-resampling in an on-line application, known as a particle filter or sequential Monte Carlo method. However, we will first turn our attention to how we can find out more about the sample space. The basic idea is to keep track of the sequence of samples and modify the proposal distribution to take advantage of this, for whic we will have to use some more complicated machinery.

14.4 Markov Chain Monte Carlo

14.4.1 Markov Chains

In probabilistic terms a chain is a sequence of possible states, where the probability of being in state s at time t is a function of the previous state. A Markov chain is a chain with the Markov property, i.e., the probability at time t depends only on the state at $t-1$, as discussed in Section 13.3. The set of possible states are linked together by transition probabilities that say how likely it is that you move from the current state to each of the others, and they are generally written as a matrix T. They might be constant, or functions of some other variables, but here we will assume that they are constant. Note that, unlike the Markov Decision Processes that we saw in Section 13.3, there is no action here that affects the probability of moving into a particular state.

Given a chain, we can perform a random walk on the chain by choosing a start state and randomly choosing each successive state according to the transition probabilities. The link to sampling that we need is that if the transition probabilities reflect the distribution that we wish to sample from, then a random walk will explore that distribution. One problem with this is that random walks are very inefficient at exploring space, since they move back towards the start as often as they move away, which means the distance they move from the start scales as $\sqrt{t}$, where t is the number of samples. We therefore want to explore more efficiently than just using a random walk.

We do this by setting up our Markov chain so that it reflects the distribution we wish to sample from, and we want the distribution $p(\mathbf{x}^{(i)})$ to converge to the actual distribution $p(\mathbf{x})$ no matter what state we start from. Since we can start from any state, this tells us that every state is reachable from every other state (so that the transition matrix can't be cut up into smaller matrices); this means that the chain is irreducible, a property that is known as ergodicity, and aperiodic so that there are no cycles in the chain.

We also want the distribution $p(\mathbf{x})$ to be invariant to the Markov chain, which means that the transition probabilities don't change the distribution:

$$p(\mathbf{x}) = \sum_{\mathbf{y}} T(\mathbf{y}, \mathbf{x}) p(\mathbf{x}). \tag{14.9}$$

Finding the transition probabilities to make this true requires that we can move backwards and forwards along the chain with equal probability, so that the chain is reversible. This says that the probability of being in an unlikely state s (sampling datapoint $\mathbf{x}$), but heading for a likely state s' (datapoint $\mathbf{x}'$) should be the same as being in the likely state s' and heading for the unlikely state s, so that:

$$p(\mathbf{x})T(\mathbf{x}, \mathbf{x}') = p(\mathbf{x}')T(\mathbf{x}', \mathbf{x}). \tag{14.10}$$

This is known as the detailed balance condition and the fact that it leaves the distribution $p(\mathbf{x})$ alone is fairly obvious with a little calculation. If the chain satisfies the detailed balance condition, then it must be ergodic, since $\sum_{\mathbf{y}} T(\mathbf{x}, \mathbf{y}) = 1$, since you must have come from some state, and so:

$$\sum_{\mathbf{y}} p(\mathbf{y}) T(\mathbf{y}, \mathbf{x}) = p(\mathbf{x}), \tag{14.11}$$

which means that $p(\mathbf{x})$ must be an invariant distribution of T. So if we can work out how to construct a Markov chain with detailed balance we can sample from it in order to sample from our distribution. This is known as Markov Chain Monte Carlo (MCMC) sampling, and the most popular algorithm that is used for MCMC is the Metropolis-Hastings algorithm after the two people who were directly involved in its creation.

14.4.2 The Metropolis-Hastings Algorithm

We assume that we have a proposal distribution of the form $q(\mathbf{x}^{(i)}|\mathbf{x}^{(i-1)})$ that we can sample from. The idea of Metropolis-Hastings is similar to that of rejection sampling: we take a sample $\mathbf{x}^*$ and choose whether or not to keep it. Except, unlike rejection sampling, rather than picking another sample if we reject the current one, instead we add another copy of the previous accepted sample. Here, the probability of keeping the sample is $u(\mathbf{x}^*|\mathbf{x}^{(i-1)})$:

$$u(\mathbf{x}^*|\mathbf{x}^{(i-1)}) = \min\left(1, \frac{\tilde{p}(\mathbf{x}^*) q(\mathbf{x}^{(i)}|\mathbf{x}^*)}{\tilde{p}(\mathbf{x}^{(i)}) q(\mathbf{x}^*|\mathbf{x}^{(i)})}\right). \tag{14.12}$$

The Metropolis-Hastings Algorithm

- Given an initial value x_0
- repeat
  - sample $\mathbf{x}^*$ from $q(\mathbf{x}_i|\mathbf{x}_{i-1})$
  - sample u from the uniform distribution
  - if $u <$ Equation (14.12):
    - set $\mathbf{x}[i+1] = \mathbf{x}^*$
  - otherwise:
    - set $\mathbf{x}[i+1] = \mathbf{x}[i]$
- until you have enough samples

So why does this algorithm work? Each step involves using the current value to sample from the proposal distribution. These values are accepted if they move the Markov chain towards more likely states, and because the

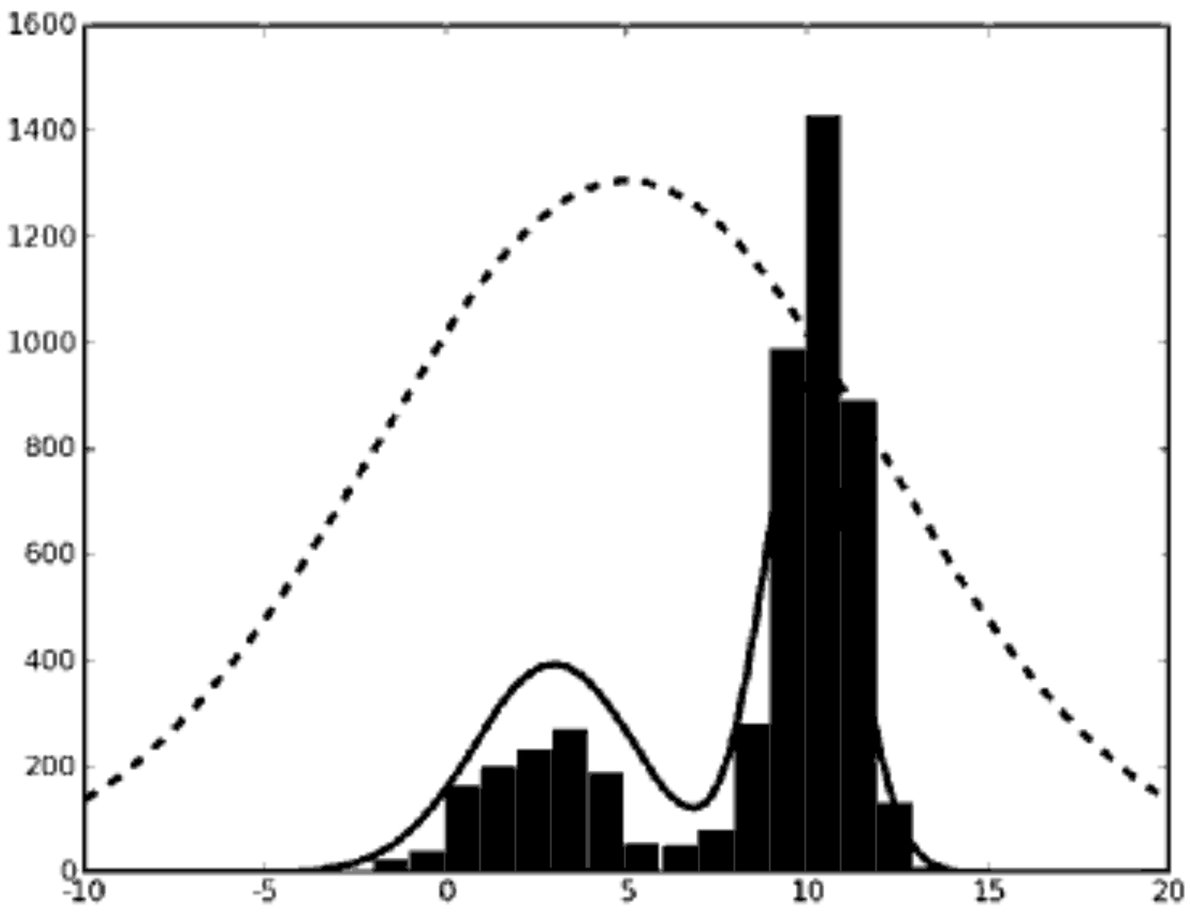


FIGURE 14.5: The results of the Metropolis-Hastings algorithm when the true distribution is a mixture of two Gaussians (shown by the solid line) and the proposal distribution is a single Gaussian (the dotted line).

Markov chain is reversible (since it satisfies the detailed balance condition) the algorithm explores states that are proportional to the difficult distribution $p(x)$.

The Metropolis-Hastings (and variants of it) are by far the most commonly used MCMC methods, and it is also the most general. It requires that you choose the proposal distribution $q(x^*|x)$ carefully, but it is a very simple algorithm to use. Figure 14.5 shows 5,000 samples computed using the algorithm on a mixture of two Gaussians based on a proposal distribution that is a single Gaussian.

Note that if the proposal distribution is symmetric, then it drops out of the test in Equation (14.12). This is the original Metropolis algorithm, and it is much closer to the pure random walk. The results of using this algorithm on the same data can be seen in Figure 14.6.

There are other choices of proposal distribution, and they lead to variants on the Metropolis-Hastings algorithm. We will consider the two most common choices next.

14.4.3 Simulated Annealing (Again)

There are lots of times when we might just want to find the maximum of a distribution rather than approximate the distribution itself. We can do this in calculating $\arg\max_{\mathbf{x}^{(i)}} p(\mathbf{x}^{(i)})$ (that is, the $\mathbf{x}^{(i)}$ with the largest probability), but while doing this we will have computed samples from many parts of the space, not just around the maximal region. A possible solution is to use simulated annealing as we did in Section 11.6. This changes the Markov chain

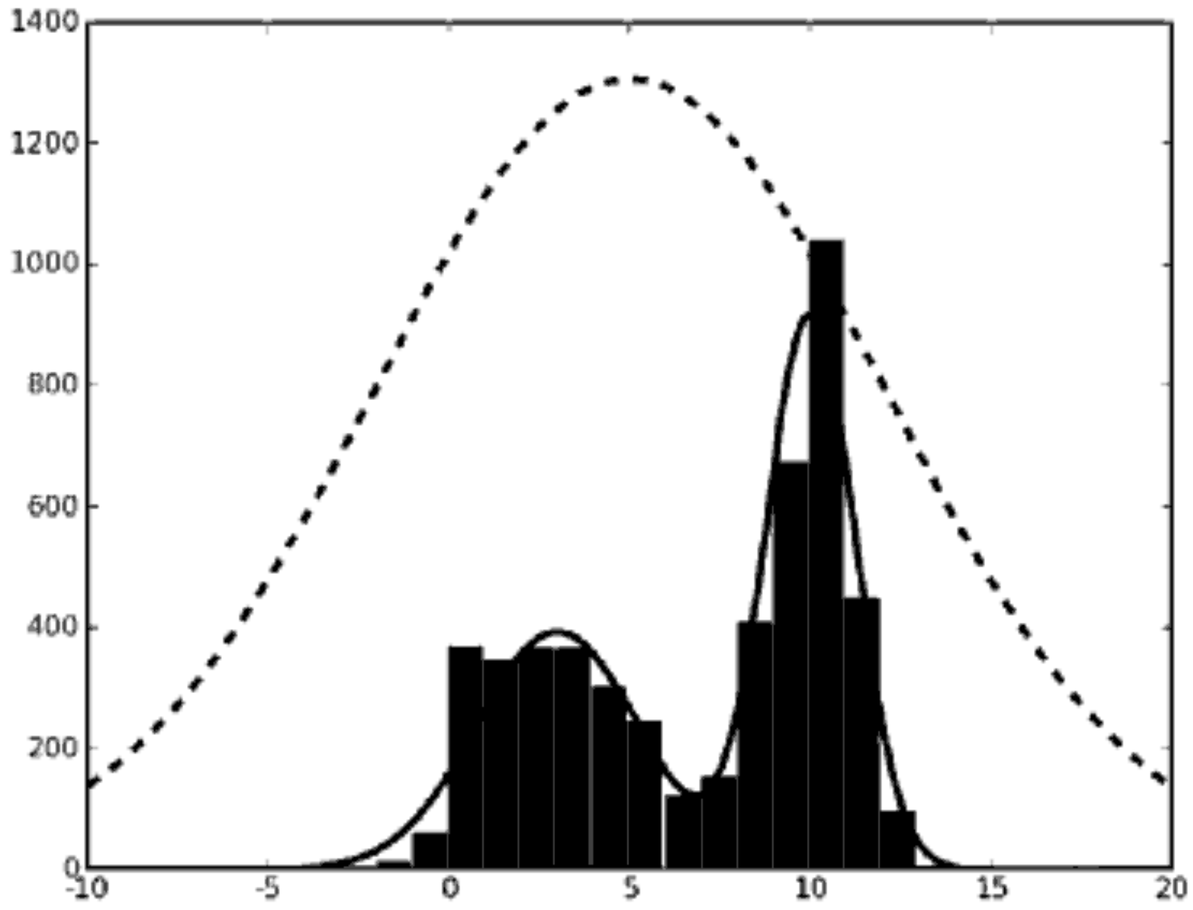


FIGURE 14.6: The results of the Metropolis algorithm when the true distribution is a mixture of two Gaussians (shown by the solid line) and the proposal distribution is a single Gaussian (the dotted line).

so that its invariant distribution is not $p(\mathbf{x})$, but rather $p^{1/T_i}(\mathbf{x})$, where $T_i \to 0$ as $i \to \infty$. We need an annealing schedule that cools the system down over time so that we are progressively less likely to accept solutions that are worse over time.

There are only two modifications needed to the Metropolis-Hastings algorithm, and both are trivial: we extend the acceptance criterion to include the temperature and add a line into the loop to include the annealing schedule. The results of using simulated annealing on the example where the true distribution is a mixture of two Gaussians and the proposal distribution is just one is shown in Figure 14.7.

14.4.4 Gibbs Sampling

Another variation on the Metropolis-Hastings algorithm comes when we already know the full conditional probability $p(x_j|x_1, \ldots x_{j-1}, x_{j+1}, \ldots x_n)$ (which is often written as $p(x_j|x_{-j})$ for convenience). We are going to see some examples of this in the next chapter: Bayesian networks. In Section 15.1.2 we will deal with a set of probabilities from a network that looks like:

$$p(\mathbf{x}) = \prod_j p(_j|x_{\alpha j}). \tag{14.13}$$

Given that we know $p(x_j|x_{\alpha j}) \prod_{k \in \beta(j)} p(x_k|x_{\alpha(k)})$ (which is $p(x_j|x_{-j})$), maybe we should try using it as the proposal distribution, giving:

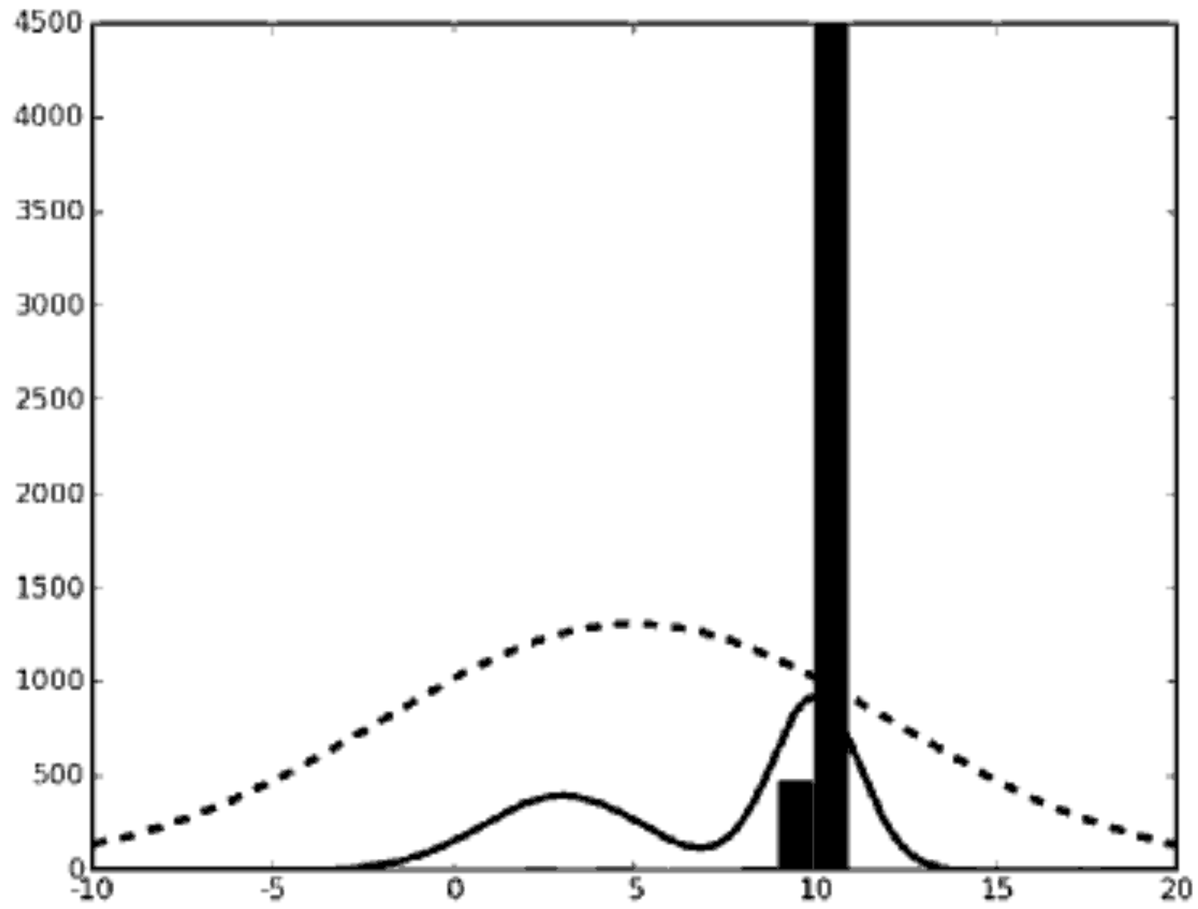


FIGURE 14.7: Using simulated annealing gives the maximum rather than an approximation to the distribution, as is shown here for the same example as in Figures 14.5 and 14.6.

$$q(x^*|x^{(i)}) = \begin{cases} p\left(x_j^*, x_{-j}^{(i)}\right) & \text{if } x^*_{-j} = x^{(i)}_{-j} \\ 0 & \text{otherwise.} \end{cases} \tag{14.14}$$

If we then use Metropolis-Hastings, we find that the acceptance probability P_a is:

$$P_a = \min\left\{1, \frac{p(x^*)p(x_j^{(i)}|x^{(i)}_{-j})}{p(x^{(i)})p(x_j^*|x^*_{-j})}\right\}, \tag{14.15}$$

and looking carefully at this and expanding out the conditional probabilities we get:

$$P_a = \min\left\{1, \frac{p(x^*)p(x_j^{(i)}, x^{(i)}_{-j})p(x^{(i)}_{-j})}{p(x^{(i)})p(x_j^*, x^*_{-j})p(x^*_{-j})}\right\}. \tag{14.16}$$

Since $p(x_j^*, x^*_{-j}) = p(x^*)$, and similarily for $p^{(i)}$, we only have to worry about $\frac{p(x^{(i)})}{p(x^*_{-j})}$. From the definition of the proposal distribution we know that $x*_{-j} = x^{(i)}_{-j}$, and so the computation is actually $\min 1, 1 = 1$. So we always accept the proposal, which makes things much simpler.

The total algorithm is given by choosing each variable and sampling from its conditional distribution. That's it! The only option that you have is whether to go through the variables in order, or whether to update them in a random order. Rather than running up to some maximum value N, it is not

uncommon to run until the joint distribution stops changing. This algorithm is known as the Gibbs sampler and it forms the basis of the software package BUGS (Bayesian Updating With Gibbs Sampling) that is commonly used in statistics. It is also a very useful algorithm for Bayesian networks, as we shall see in the next chapter.

The Gibbs Sampler

- for each variable x_j:
  - initialise $x_j^{(0)}$
- repeat
  - for each variable x_j:
    - sample $x_1^{(i+1)}$ from $p(x_1|x_2^{(i)}, \ldots x_n^{(i)})$
    - sample $x_2^{(i+1)}$ from $p(x_2|x_1^{(i+1)}, x_3^{(i)}, \ldots x_n^{(i)})$
    - ...
    - sample $x_n^{(i+1)}$ from $p(x_n|x_1^{(i+1)}, \ldots x_{n-1}^{(i+1)})$
- until you have enough samples

As an example, suppose that we have a distribution that is made up of two different distributions, a binomial one in x and a beta in y. If you don't know what these distributions are, the combined distribution can be written as:

$$p(x, y, n) = \left(\frac{n!}{x!(n-x)!}\right) y^{x+\alpha-1} + (1-y)^{n-x+\beta-1}. \tag{14.17}$$

The important point is that the overall distribution is a product of two separate ones that can be sampled from separately. Figure 14.8 shows the output of the sampling using the Gibbs sampler, with the line being the correct distribution as usual. There is another example of Gibbs sampling in Section 15.1.2.

Hidden page

Hidden page

Chapter 15

Graphical Models

Throughout this book we have seen that machine learning brings together computer science and statistics. Nowhere is this more clearly shown than in one of the most popular areas of current research in machine learning: graphical models (or more completely, probabilistic graphical models), which use graph theory with all its underlying computational and mathematical machinery in order to explain probabilistic models.

The graphs used in graphical models are the exact ones that are taught in basic algorithms classes: a set of nodes, together with links between them, which can be either directed (i.e., have arrows on them so that you can only go one way along them) or not. There are two basic types of graphical models, depending upon whether or not the edges are directed. We will focus primarily on directed graphs, but the undirected kind (known as Markov Random Fields) are described in Section 15.2. For such a simple data structure, graphs have turned out to be incredibly powerful in many different parts of computer science, from constructing compilers to managing computer networks. For this reason, there are lots of readily available algorithms for finding shortest paths (Floyd's and Djiksta's algorithms, which we've already discussed briefly in Section 10.6), determining cycles, etc. Any good book on algorithms will give details of these and many other graph algorithms.

For our part, we are interested in using graphs to encode probability distributions and so we need to decide what nodes and links are in this context. The nodes are fairly obvious. We generate a node for each random variable, and label it accordingly. In this book, we will only consider discrete variables, so that there is a finite number of possible values that the random variable can take. Given a continuous variable we will discretise it into a finite set. While this loses information, it makes the problem much simpler. The alternative is to specify the variable by a probability density function, which can be done, but makes the whole thing harder to describe and understand.

The question is what to make the links represent. Perhaps the best way to think about this is to ask what it means if two nodes are not linked. In this case we are saying that there is no connection between those two variables, which is the same as saying that they are independent. Except it isn't quite as simple as that, because two nodes could be linked through a third node. Have a look at the right of Figure 15.1, where C is not directly linked to B, but there is a link through A. For this reason we have to be careful and talk

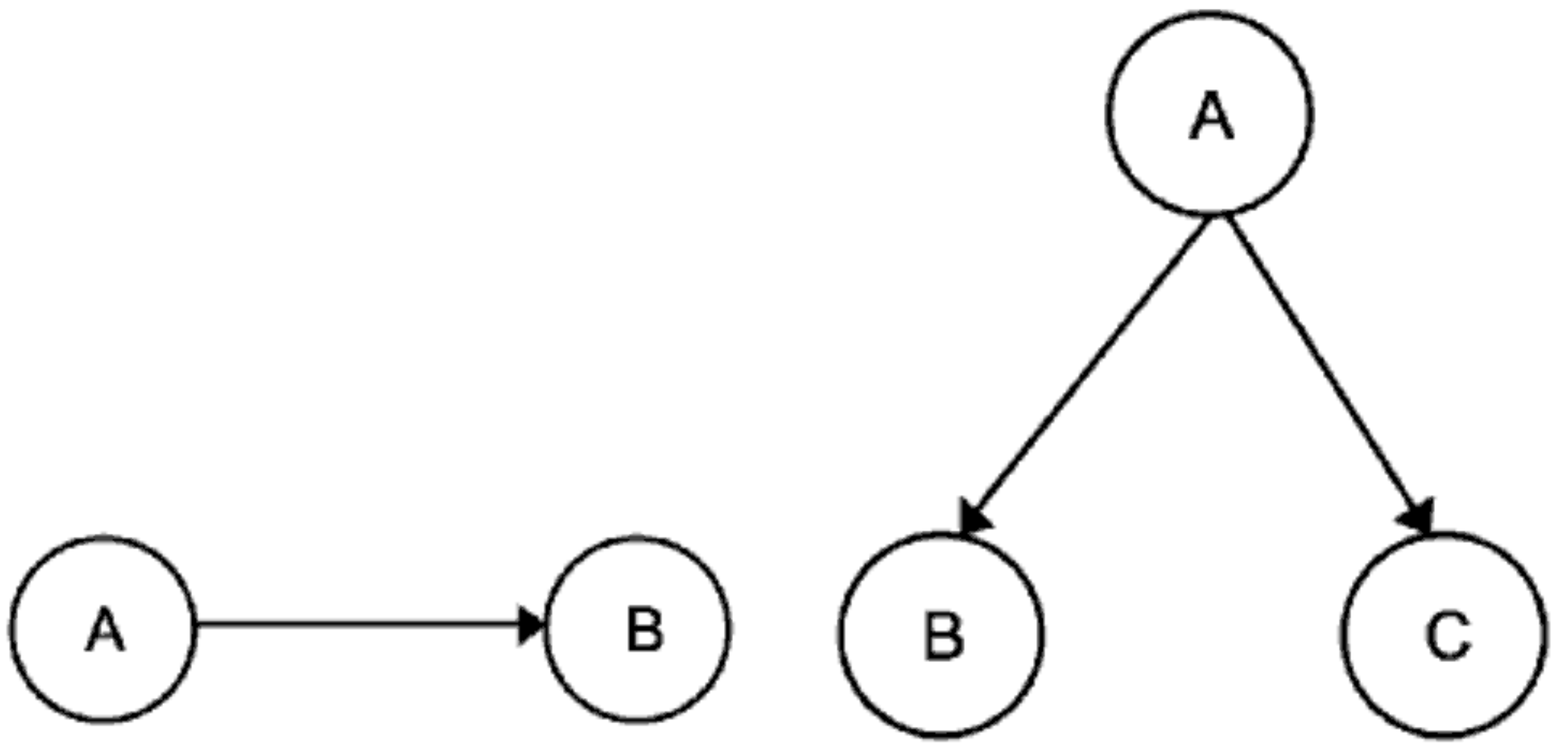


FIGURE 15.1: Two simple graphical models. The arrows denote causal relationships between nodes that represent features.

about conditional independence: C is conditionally independent of B, given A.

We use directed links because these relationships are not symmetrical (unless the variables are independent, in which case there is no link). What does the simplest connected graph that we can make, the one on the left of Figure 15.1, mean? There is a rather loose interpretation of the link, which is to say that 'A' causes 'B' (but note that this isn't quite the same semantic usage that we normally have for 'causes,' since there may be several variables that are all involved in causing B). This is a useful intuition to have, but it is not really correct. More properly, the graph tells us that the probability of A and B is the same as the probability of A times the probability of B conditioned on A: $P(a, b) = P(b|a)P(a)$.

There is a third thing that we need in order to specify the problem properly, which is the conditional probability table for each variable. This specifies what the probabilities are for each of the nodes, conditioned on any nodes that are its parents. If we wanted to work out a value for $P(a, b)$, then we would need a distribution table for $P(a)$ and one for $P(b|a)$. The nodes are separated into these where we can see their values directly—observed nodes—and hidden or latent nodes, whose values we hope to infer, and which may not have clear meanings in all cases.

The basic concept of the graphical model is very simple, which makes it all the more amazing that it produces a powerful set of tools for understanding and creating machine learning algorithms. We will start by looking at the most general model, the Bayesian Belief network or more simply, Bayesian network, and see how they are represented, and the difficulties involved in dealing with them. Following this, we will identify a few places where these difficulties can be overcome, resulting in some very important algorithms that solve a variety of different tasks. In particular, we will look at Markov Random Fields (MRFs), Hidden Markov Models (HMMs), the Kalman Filter, and particle filter.

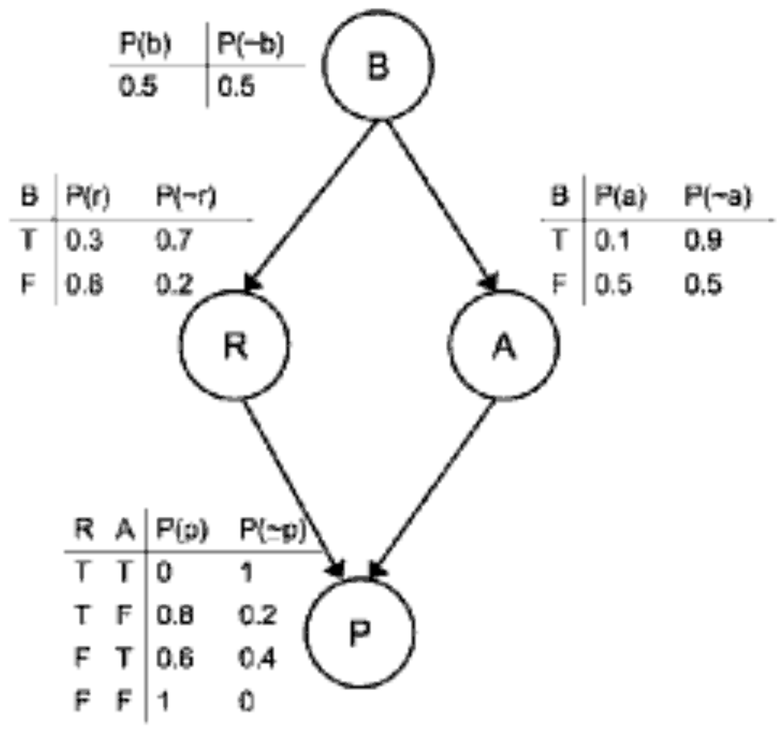


FIGURE 15.2: The sample graphical model. 'B' denotes a node stating whether the exam was boring, 'R' whether or not you revised, 'A' whether or not you attended lectures, and 'P' whether or not you will panic before the exam.

15.1 Bayesian Networks

To start with, we will consider directed graphs, and make one restriction to them, namely that they must not contain cycles, that is, there cannot be any loops in the graphs. These graphs go by the rather unlovely name of DAGs: directed, acyclic graphs, but for graphical models, when they are paired with the conditional probability tables, they are called Bayesian networks. In order to see what we can do with such a network, we need an example.

15.1.1 Example: Exam Panic

Figure 15.2 shows a graph with a full set of distribution tables specified. It is a handy guide to whether or not you will panic before an exam based on whether or not the course was boring ('B'), which was the key factor you used to decide whether or not to attend lectures ('A') and revise ('R'). We can use it to perform inference in order to decide the likelihood of you panicking ('P'). There are two kinds of inferences, depending on whether the observations that are made come from the top of the graph or the bottom. If we have a set of observations that can be used to predict an unknown outcome, then we are doing top-down inference or prediction, whereas if the outcome is known, but the causes are hidden, then we are doing bottom-up inference or diagnosis. Either way, we are working out the values of the hidden (unknown) nodes given information about the observed nodes. For the example in Figure 15.2 we will start by predicting whether or not you will panic before the exam, so it is the outcome that is hidden.

In order to compute the probability of panicking, we need to compute $P(b, r, a, p)$, where the lower-case letters indicate particular values that the upper-case variables can take. The wonderful thing about the graphical model

Hidden page

$$
\begin{aligned}
P(r|p) &= \frac{P(p|r)P(r)}{P(p)} \\
&= \frac{\sum_{b,a} P(b,a,r,p)}{P(p)} \\
&= \frac{0.5 \cdot (0.3 \cdot 0.1 \cdot 0 + 0.3 \cdot 0.9 \cdot 0.8) + 0.5 \cdot (0.8 \cdot 0.5 \cdot 0 + 0.8 \cdot 0.5 \cdot 0.8)}{P(p)} \\
&= \frac{0.268}{0.684} = 0.3918. \qquad (15.3)
\end{aligned}
$$

$$
\begin{aligned}
P(r|a) &= \frac{P(p|a)P(a)}{P(p)} \\
&= \frac{0.144}{0.684} = 0.2105. \qquad (15.4)
\end{aligned}
$$

This use of Bayes' rule is the reason why this type of graphical model is known as a Bayesian network. Even in this very simple example, the inference was not trivial, since there were a lot of calculations to do. However, the problem is actually rather worse than that. The computational cost of the simple algorithm we used (start at the root, and follow each link through the graph to perform the computation) is $\mathcal{O}(2^N)$ for a graph with N nodes where each node can be either true or false. In general the problem of exact inference on Bayesian networks is NP-hard (technically, it is actually #P-hard, which is even worse). However, for so-called polytrees where there is at most one path between any two nodes, the computational cost is much smaller—linear in the size of the network.

Unfortunately, it is rare to find such polytrees in real examples, so we can either try to turn other networks into polytrees, or consider only approximate inference, which is the most common solution to the problem, and the method that we'll consider next. We can speed things up a little by getting things into the form of Equation (15.1), where the summations were carefully placed as far to the right as possible, so that program loops can be minimised. By doing this the algorithm is as efficient as possible, but it is still NP-hard. This is sometimes known as the variable elimination algorithm, which is a variation on the bucket elimination algorithm. The idea is to convert the conditional probability tables into what are called λ tables, which simply list all of the possible values for all variables, and which initially contain the conditional probabilities. For example, the λ table for the 'P' variable in Figure 15.2 is:

R	A	P	λ
T	T	T	0
T	T	F	1
T	F	T	0.8
T	F	F	0.2
F	T	T	0.6
F	T	F	0.4
F	F	T	1
F	F	F	0

If I see you panicking outside the exam (so that P is true), then I can eliminate it from the graph by removing from each table all rows that have P false in them, and deleting the P column. This simplifies things a little, but I have to do rather more in order to compute the probability of you having attended lectures. I don't know whether you revised or not, and I don't know if you found the lectures boring, so I have to marginalise over these variables. The order in which we marginalise doesn't change the correctness (although more advanced algorithms can improve the speed by taking advantage of conditional independence) so we'll pick R first. To eliminate it from the graph, we have to find all of the λ tables that contain it (there will be two of them containing R: the one for R itself and the one that we have just modified to remove P). To remove R, we have to add together the products of the λ values that correspond to places where the other values match. So to complete the entry where B is true and A is false, we have to multiply together the values where B, A, R are respectively true, false, true in the two tables and then add to that the product of where B, A, R are respectively true, false, false. In other words:

$$\begin{pmatrix} B & R & \lambda \\ T & T & 0.3 \\ T & F & 0.7 \\ F & T & 0.8 \\ F & F & 0.2 \end{pmatrix} \times \begin{pmatrix} R & A & \lambda \\ T & T & 0 \\ T & F & 0.8 \\ F & T & 0.6 \\ F & F & 1 \end{pmatrix} \Rightarrow \begin{pmatrix} B & A & \lambda \\ T & T & 0.3 \cdot 0 + 0.7 \cdot 0.6 = 0.42 \\ T & F & 0.3 \cdot 0.8 + 0.7 \cdot 1 = 0.94 \\ F & T & 0.8 \cdot 0 + 0.2 \cdot 0.6 = 0.12 \\ F & F & 0.8 \cdot 0.8 + 0.2 \cdot 1 = 0.84 \end{pmatrix} \quad (15.5)$$

We can do the same thing in order to eliminate B, which involves all three of the tables, and this will enable the computation of the conditional probability of you attending lectures given that I saw you panicking before the exam. The benefit of doing things this way is that the whole thing can be written as a general algorithm:

The Variable Elimination Algorithm

- create the λ tables:
  - for each variable v:
    - make a new table

        * for all possible true assignments x of the parent variables:
            · add rows for $P(v|x)$ and $1 - P(v|x)$ to the table
        * add this table to the set of tables

- eliminate known variables v:
    - for each table:
        * remove rows where v is incorrect
        * remove columm for v from table

- eliminate other variables (where x is the variable to keep):
    - for each variable v to be eliminated:
        * create a new table t'
        * for each table t containing v:
            · $v_{\text{true},t} = v_{\text{true},t} \times P(v|x)$
            · $v_{\text{false},t} = v_{\text{false},t} \times P(\neg v|x)$
        * $v_{\text{true},t'} = \sum_t (v_{\text{true},t})$
        * $v_{\text{false},t'} = \sum_t (v_{\text{false},t})$
    - replace tables t with the new one t'

- calculate conditional probability:
    - for each table:
        * $x_{\text{true}} = x_{\text{true}} \times P(x)$
        * $x_{\text{false}} = x_{\text{false}} \times P(\neg x)$
        * probability is $x_{\text{true}}/(x_{\text{true}} + x_{\text{false}})$

To see that these algorithms do not scale well, consider Figure 15.3, which shows a very simple development of the example in Figure 15.2 by adding just one extra node to the network: whether or not this is your final year ('F'). This makes the network significantly more complicated, since we need another table and extra entries in two of the other tables, and therefore the variable elimination algorithm will take rather longer to run.

15.1.2 Approximate Inference

Since the variable elimination algorithm will only take you so far, for reasonably sized Bayesian networks there is no choice but to perform approximate inference. Fortunately, we have already seen a set of algorithms that are ideally suited to the problem: the Markov Chain Monte Carlo methods that we saw in Chapter 14. There are two other methods of doing approximate inference (loopy belief propagation and mean field approximation) as well, but

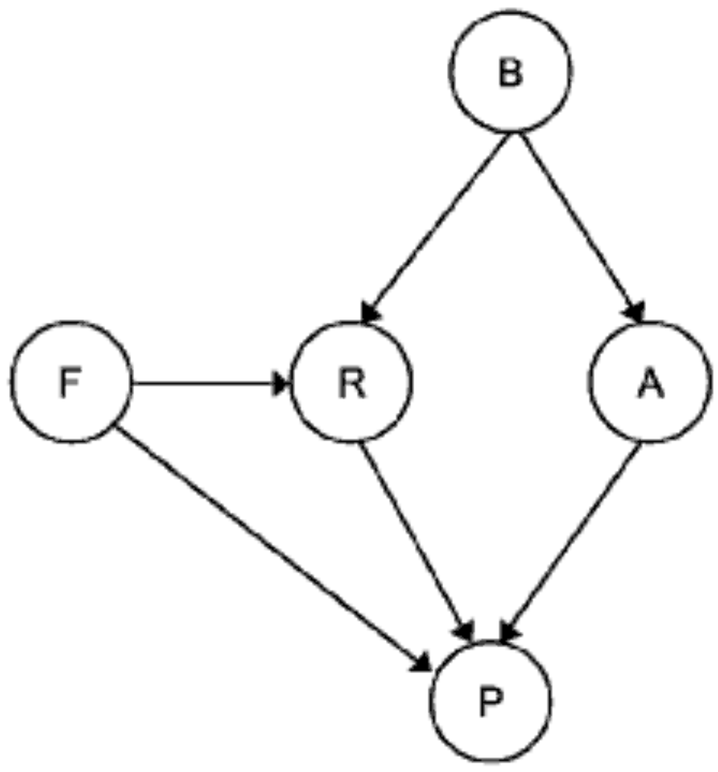


FIGURE 15.3: Adding just one extra node ('F', information about whether or not this is your final year) makes the conditional probability tables significantly more complicated.

we will not consider them further; there are references to descriptions of these methods at the end of the chapter.

The basic idea of using MCMC methods in Bayesian networks is to sample from the hidden variables, and then (depending upon the MCMC algorithm employed) weight the samples by their likelihoods. Creating the samples is very easy: for prediction, we start at the top of the graph and sample from each of the known probability distributions. Using Figure 15.2 again, we generate a sample from $P(b)$, and then use that value in the conditional probability tables for 'R' and 'A' to compute $P(r|b = \text{sample value})$ and $P(a|b = \text{sample value})$. These three values are then used to sample from $P(p|b, a, r)$. We can take as many samples as we like in this way, and expect that as the number of samples gets large, so the frequency of specific samples will converge to their expected values.

In this sampling method, we have to work through the graph from top to bottom and select rows from the conditional probability table that match the previous case. This is not what we would do if we were constructing the table by hand. Suppose that you wanted to know how many courses you did not attend the lectures for because the course was boring. You would simply look back through your courses and count the number of boring courses where you didn't go to lectures, ignoring all the interesting courses. We can use exactly this idea if we use rejection sampling (see Section 14.3). The method samples from the unconditional distribution and simply rejects any samples that don't have the correct prior probability. It means that we can sample from each distribution independently, and then throw away any samples that don't match the other variables. This is obviously computationally easier, but we might have to reject a lot of samples.

The solution to this problem is to work out what evidence we already have and use this evidence to assign likelihoods to the other variables that are

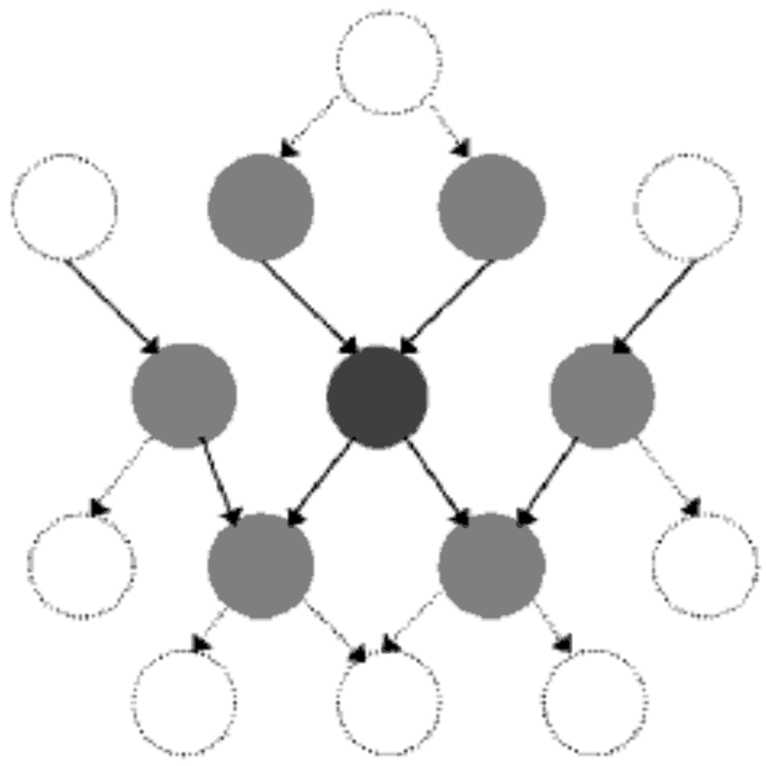


FIGURE 15.4: The Markov blanket of a node is the set of nodes (shaded light grey) that are either parents or children of the node, or other parents of its children (shaded dark grey).

sampled. Suppose that we sample from $P(b)$ and get value 'true'. If we already know that we did revise, then we weight the observation $P(r|b)$ by the appropriate probability, which is 0.3. We continue through the other variables, sampling where there is no evidence and using the tables to find the probability if we do have evidence. However, we can do rather better than this by using the full MCMC framework. We start by setting values for all of the possible probabilities, based on either evidence or random choices. This gives us an initial state for a Markov chain. Now Gibbs sampling (Section 14.4.4) will find us the maxima of our probability distribution given enough samples.

The probabilities in the network are:

$$p(x) = \prod_j p(x_j|x_{\alpha j}), \tag{15.6}$$

where $x_{\alpha j}$ are the parent nodes of x_j. In a Bayesian network, any given variable is independent of any node that is not their child, given their parents. So we can write:

$$p(x_j|x_{-j}) = p(x_j|x_{\alpha j}) \prod_{k \in \beta(j)} p(x_k|x_{\alpha(k)}), \tag{15.7}$$

where $\beta(j)$ is the set of children of node x_j and x_{-j} signifies all values of x_i except x_j. For any node we only need to consider its parents, its children, and the other parents of the children, as shown in Figure 15.4. This set is known as the Markov blanket of a node.

Given these calculations, computing the inference on any real Bayesian network generally consists of using Gibbs sampling in order to approximate the inference. For the exam panic example, the algorithm to perform the Gibbs sampling consists of computing the probability distributions (possibly

by using parts of the variable elimination algorithm) and then sampling from it:

```
for i in range(nsamples):
    # values contain current samples of b, r, a, p
    values = where(random.rand(4)<0.5,0,1)
    for j in range(nsteps):
        values=pb_rap(values)
        values=pr_bap(values)
        values=pa_brp(values)
        values=pp_bra(values)
    distribution[values[0]+2*values[1]+4*values[2]+8*values[3]]
    += 1
distribution /= nsamples
```

For the example, a sample distribution (based on 500 samples, with 10 iterations of each chain) is:

```
b r a p:   dist
1 1 1 1    0.0
1 1 1 0    0.086
1 1 0 1    0.038
1 1 0 0    0.052
1 0 1 1    0.048
1 0 1 0    0.116
1 0 0 1    0.274
1 0 0 0    0.0
0 1 1 1    0.0
0 1 1 0    0.088
0 1 0 1    0.068
0 1 0 0    0.114
0 0 1 1    0.03
0 0 1 0    0.076
0 0 0 1    0.01
0 0 0 0    0.0
```

15.1.3 Making Bayesian Networks

If we are given the structure and conditional probability tables of the Bayesian network, then we can perform inference on it by using Gibbs sampling or, if the network is simply enough, exactly. However, this raises the important question about where the Bayesian network itself comes from. Unfortunately, the news in this area isn't particularly good: the computational

Hidden page

and then you use data in order to compute the conditional probability tables. However, it is still difficult. The idea is to choose the probability distributions to maximise the likelihood of the training data. If there are no hidden nodes, then it is possible to compute the likelihood directly:

$$\begin{aligned} L &= \frac{1}{M} \log \prod_{m=1}^{N} P(D_m|G) \\ &= \frac{1}{M} \sum_{n=1}^{N} \sum_{m=1}^{M} \log P(X_n|\text{parents}(X_n), D_m), \end{aligned} \tag{15.8}$$

where M is the number of training data examples D_m, and X_n is one of the N nodes in graph G. Equation (15.8) has broken everything into sums over each node individually, which means that we can compute each separate conditional probability table. To compute the values of the table, you just need to count how often you have panicked in an exam given each of the possible values for having revised and attended lectures, and normalise it to make it into a probability. The danger with this is that with small amounts of data there could be examples that have not happened in training, and that will therefore have probability 0, although this can be dealt with by including prior probabilities and using Bayes' rule to update the estimates using the real data.

Obviously, this doesn't work if there are hidden nodes, since we don't know values for them in the data. Surprisingly, getting around this problem isn't as difficult as might be expected. The key is to see that if we did have values for them, then Equation (15.8) could be used. We can estimate values for them by inference, and then we can iterate these two steps: an estimation step using inference followed by a maximisation step, making this an EM algorithm (Section 8.3.1).

There is lots more work on Bayesian networks, and the references at the end of the chapter include entire books on the topic for anybody wishing to explore more in this area. We will now turn our attention to some other types of graphical models, starting with the variation where the edges are undirected.

15.2 Markov Random Fields

Bayesian networks are inherently asymmetric, since each edge had an arrow on it. If we remove this constraint, then there is no longer any idea of children and parent nodes. It also makes the idea of conditional independence that we saw for the Bayesian network easier: two nodes in a Markov Random Field

(MRF) are conditionally independent of each other, given a third node, if there is no path between the two nodes that doesn't pass through the third node. This is actually a variation on the Markov property, which is how the networks got their name: the state of a particular node is a function only of the states of its immediate neighbours, since all other nodes are conditionally independent given its neighbours. You might think that this fact would make inference on MRFs simpler, but unfortunately it doesn't; in general it is still a #P-hard problem. However, there are particular applications where MRF methods have turned out to be particularly useful, often for images.

The most well-known example is image denoising, something that we have already seen in Section 3.4.5 when we talked about auto-associative learning in the MLP. Suppose that we have a binary image I with pixel values $I_{x_i,x_j} \in \{-1, 1\}$. This image is a representation of an 'ideal' image I'_{x_i,x_j} that has no noise in it, which is what we want to recover. If we assume that the amount of noise is small, then there should be a good correlation between the values of each pixel in the two images, so I_{x_i,x_j} and I'_{x_i,x_j} should be correlated. We also assume that within a small 'patch' or region in an image, there is good correlation between pixels (so I_{x_i,x_j} should correlate well with I_{x_i+1,x_j} and its other neighbouring pixels (I_{x_i,x_j-1}, etc.). This assumption says that there are lots of places in the image where all of the pixels are of the same value, and this is (at least approximately) true for most images, and says that the pixels are correlated (and that other pixels in the image are conditionally independent of I_{x_i,x_j} given the neighbours of that pixel, which is the MRF bit).

The original theory of MRFs was worked out by physicists, initially by looking at the **Ising model**, which is a statistic description of a set of atoms connected in a chain, where each can spin up (+1) or down (-1) and whose spin affects those connected to it in the chain. Physicists tend to think of the energy of such systems, and argue that stable states are those with the lowest energy, since the system needs to get extra energy if it wants to move out of this state. For this reason, the jargon of MRFs is in terms of energies, and we therefore want the energy of our pair of images to be low when the pixels match, and higher when they do not. So we write the energy of the same pixel in two images as $-\eta I_{x_i,x_j} I'_{x_i,x_j}$, where η is a positive constant. Note that if the two pixels have the same sign then the energy is negative, while if they have opposite signs then the energy is positive and therefore larger. The energy of two neighbouring pixels is $-\zeta I_{x_i,x_j} I_{x_i+1,x_j}$, and we can just add these components together to get the total energy:

$$E(I, I') = -\zeta \sum_{i,j}^{N} I_{x_i,x_j} I_{x_i \pm 1, x_j \pm 1} - \eta \sum_{i,j=1}^{N} I_{x_i,x_j} I'_{x_i,x_j}, \qquad (15.9)$$

where the index of the pixels is assumed to run from 1 to N in both the x and y directions in both images and we are only interested in locally flat patches of the image we are changing, which is I.

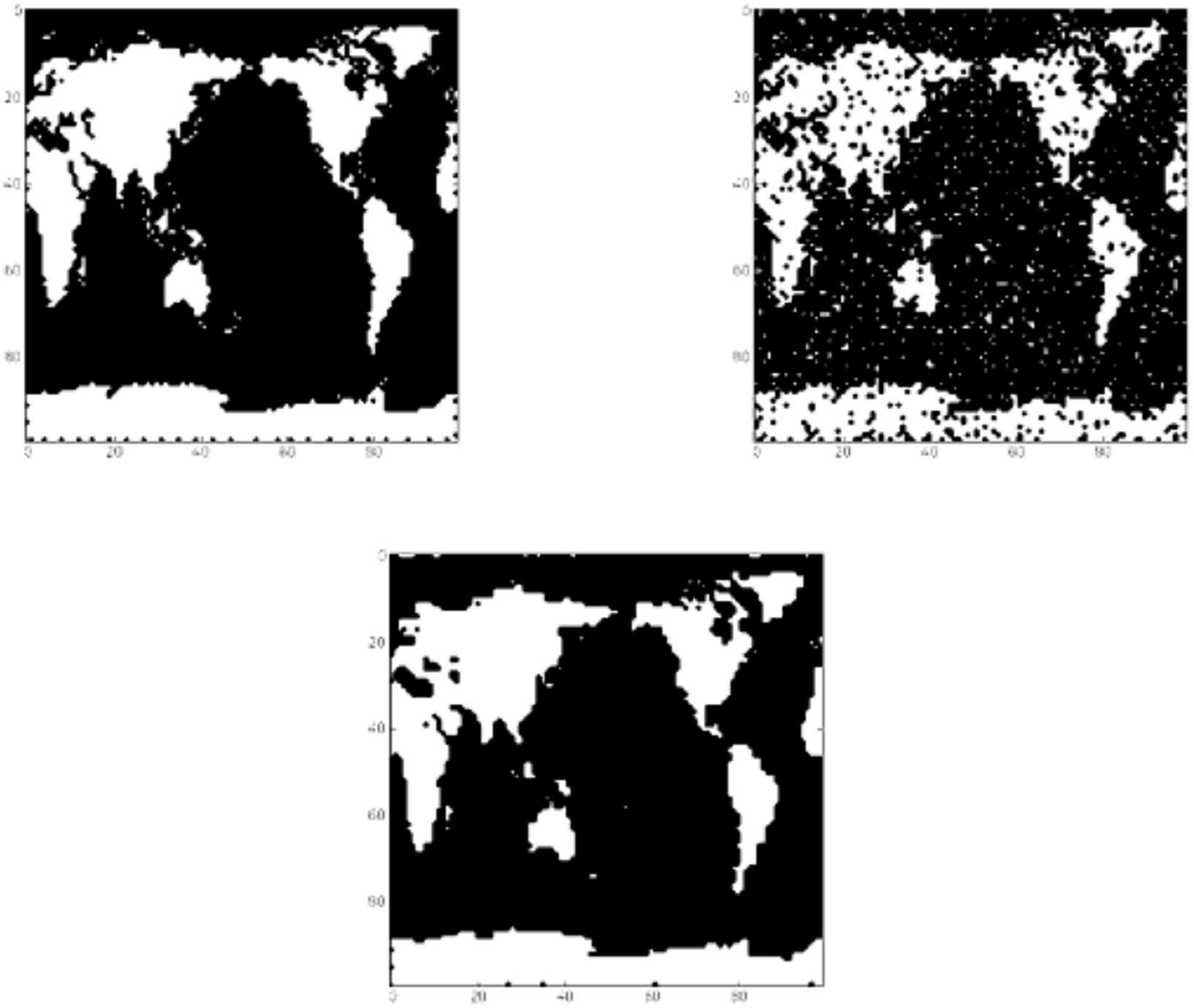


FIGURE 15.5: Using the MRF image denoising algorithm with $\eta = 2.1, \zeta = 1.5$ on a map of the world (*top left*) corrupted by 10% uniformly distributed random noise (*top right*) gives the image below which has about 1% error, although it has smoothed out the edges of all the continents.

There is now a simple iterative update algorithm, which is to start with noisy image I and ideal I', and update I so that at each step the energy calculation is lower. So you pick one pixel I_{x_i,x_j} for some values of x_i, x_j at a time, and compute the energies with this pixel being set to -1 and 1, picking the lower one. In probabilistic terms, we are making the probability $p(I, I')$ higher. The algorithm then moves on to another pixel, either choosing a random pixel at each step or moving through them in some pre-determined order, running through the set of pixels until their values stop changing. Figure 15.5 shows an original black and white image, a version corrupted with 10% noise, and the MRF-reconstructed version using parameters $\eta = 2.0, \zeta = 1.5$. This reduces the error from 10% to less than 1%, although it also removes my home country of New Zealand from the map!

The Markov Random Field Image Denoising Algorithm

- given a noisy image I and an original image I', together with parameters η, ζ:
- loop over the pixels of image I:

Hidden page

scared, or fine. I want to use these observations to try to work out what you did last night. The problem is that I don't know why you look the way you do, but I can guess by assigning probabilities to those things. So if you look hungover, then I might give probability 0.5 to the guess that you went to the pub last night, 0.25 to the guess that you went to a party, 0.2 to watching TV, and 0.05 to studying. In fact, we will use these the other way round, using the probability that you look hungover given what you did last night. These are known as observation or emission probabilities.

I don't have access to the other information that was used in Chapter 6, such as what parties are on and what other assignments you have (one of the worst things about stopping being a student is that the number of parties you get invited to drops off), but based on my own experience of being a student I can guess how likely parties are, etc., and knowing what student finances are, I can guess things like the probability of you going to the pub tonight if you went to the pub last night. So now it is just a question of putting these things into a form where I can work with them, and I can prepare my lectures according to how well you are working.

Each day that I see you in lectures I make an observation of your appearance, $o(t)$, and I want to use that observation to guess the state $\omega(t)$. This requires me to build up some kind of probabilities $P(o_k(t)|\omega_j(t))$, which is the probability that I see observation o_k (e.g., you are tired) given that you were in state ω_j (e.g., you went to a party) last night. These are usually labelled as b_{jk}. The other information that I have, or think I have, is the transition probability, which tells me how likely you are to be in state ω_j tonight given that you were in state ω_i last night. So if I think you were at the pub last night I will probably guess that the probability of you being there again tonight is small because your student loan won't be able to handle it. This is written as $P(\omega_j(t+1)|\omega_i(t))$ and is usually labelled as a_{ij}.

I can add one more constraint to each of the probability distributions a_{ij} and b_{ij}. I know that you did something last night, so $\sum_j a_{ij} = 1$ and I know that I will make some observation (since if you aren't in the lecture I'll assume you were too tired), so $\sum_k b_{jk} = 1$. There is one other thing that is generally assumed, which is that the Markov chain is ergodic, something that we saw in Section 14.4.1: it means that there is a non-zero probability of reaching every state eventually, no matter what the starting state.

After a couple of weeks of the course I have made observations about you, and I am ready to sort out my HMM. There are three things that I might want to do with the data:

- see how well the sequence of observations that I've made match my current HMM (Section 15.3.1)
- work out the most probable sequence of states that you've been in based on my observations (Section 15.3.2)

- given several sets of observations (for example, by watching several students) generate a good HMM for the data (Section 15.3.3)

We will start by assuming that I invent a model and want to see how good it is. So I use my own knowledge of being a student to work out the probability distributions and then I can test the observations I make of you against my model. At this point I will probably find out that my student life was different to yours, or things have changed since I was a student, and I will have to generate a new model to match current data. I can then use this improved model to work out what you've been doing each evening. These problems are dealt with in the next three sections.

The HMM itself is made up of the transition probabilities a_{ij} and the observation probabilities b_{jk}. So these are the things that I need to specify for myself, starting with the transition probabilities (which are also shown in Figure 15.7):

	Previous night			
	TV	Pub	Party	Study
TV	0.4	0.6	0.7	0.3
Pub	0.3	0.05	0.05	0.4
Party	0.1	0.1	0.05	0.25
Study	0.2	0.25	0.2	0.05

and then the observation probabilities:

	TV	Pub	Party	Study
Tired	0.2	0.4	0.3	0.3
Hungover	0.1	0.2	0.4	0.05
Scared	0.2	0.1	0.2	0.3
Fine	0.5	0.3	0.1	0.35

15.3.1 The Forward Algorithm

Suppose that I see the following observations: O = (tired, tired, fine, hungover, hungover, scared, hungover, fine) and I want to work out the likely run of states that generated it. The probability that my observations $O = \{o(1), \ldots, o(T)\}$ come from the model can be computed using simple conditional probability. I know you were doing something last night, so for an observation $o(t)$ =tired (say) I just need to compute the probability that I made that observation given you were in a particular state (say watching TV) and multiply it by the probability that you were in that state given the state I thought you were in the night before (say partying). So for the example, I compute the probability that you were tired given that you were watching TV, which is 0.2, and then multiply it by the probability that you spent last night watching TV given that I thought you were partying the night before, which is 0.1. So this yields probability 0.02 for this particular state change.

Hidden page

$$P(O) = \sum_{r=1}^{R} \prod_{t=1}^{T} P(o_k(t)|\omega_j(t)) P(\omega_j(t)|\omega_i(t-1))$$
$$= \sum_{r=1}^{R} \prod_{t=1}^{T} b_{jk} a_{ij}. \tag{15.13}$$

This looks fairly easy now. The only problem is in that sum over r, which runs over all possible sequences of hidden states. If there are N hidden states then there are N^T possible sequences, and for each one we have to compute a product of T probabilities. Not only will these probabilities be incredibly small, but the computational cost of getting them will be astronomical: $\mathcal{O}(N^T T)$.

Fortunately, the Markov property comes to our rescue again. Since the probability of each state only depends on the data at the current and previous timestep $(o(t), \omega(t), \omega(t-1))$ we can build up our computation of $P(O)$ one timestep at a time. This is known as the forward trellis by some people, since it looks like a garden trellis in Figure 15.8. To construct the trellis we introduce a new variable $\alpha_i(t)$ that describes the probability that at time t the state is ω_i and that the first $(t-1)$ steps all matched the observations $o(t)$:

$$\alpha_j(t) = \begin{cases} 0 & t = 0, j \neq \text{initial state} \\ 1 & t = 0, j = \text{initial state} \\ \sum_i \alpha_i(t-1) a_{ij} b_{j(o_t)} & \text{otherwise.} \end{cases} \tag{15.14}$$

where $b_{j(o_t)}$ means the particular transition probability for output o_t. This ensures that only the observation probability that has the index that matches the observation o_t contributes to the sum. Computing $P(O)$ now requires only $\mathcal{O}(N^2 T)$, which is a substantial improvement, in a very simple algorithm:

The HMM Forward Algorithm

- for each observation in order o_t, $t = 1, \ldots, T$
  - for each possible state s
    - $\alpha_{s,t} = b_{s,o_t} * \sum_x (\alpha_{x,t} * a_{x,s})$

Let's look at the first two states of our example HMM. In both, the observation is that you were tired, so we need to compute $\alpha_i(t)$ and so construct the trellis. Figure 15.8 shows the idea, with the initial $\alpha_1(\cdot)$ coming from my guesses about how likely each state is, and we just need to run through the set of computations to compute $\alpha_2(\cdot)$ and so on, getting the numbers that are shown in the figure. We then repeat this for the next step to get the $\alpha_3(\cdot)$ and so on until we reach the final state. At this stage we can sum up all

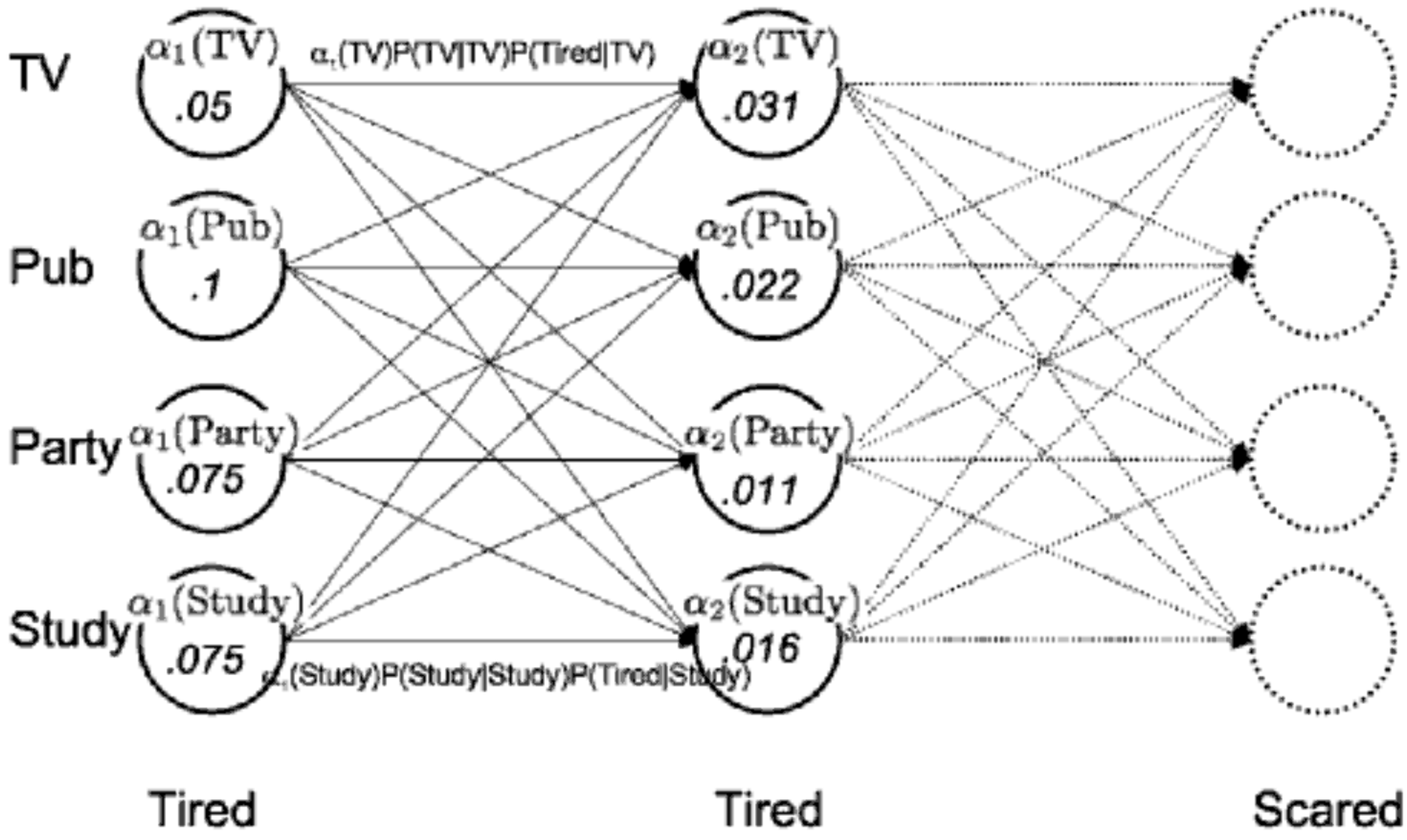


FIGURE 15.8: The forward trellis for the first two observations of the example HMM.

of the possible probabilities, which tells me in this case that you were most likely watching TV last night. We will also need to be able to go backwards through the trellis, which is a very similar algorithm that works backwards to compute β values, also based on the transmission probability and observation probability matrices.

15.3.2 The Viterbi Algorithm

The next problem that we want to solve is the decoding problem: I can use my model of how students are expected to behave and match them with my observations to guess what you have been doing each evening. The algorithm is known as the Viterbi algorithm after its creator, although he actually derived it for error correction, a completely different application! It works on very similar principles to the forward algorithm, except that we act in a greedy way, since for each timestep we pick the state that is most likely as the next step in the path, rather than maintaining probabilities of all possible paths.

The HMM Viterbi Algorithm

- for each observation in order o_t, $t = 1 \ldots T$
  - for each possible state s
    - $v_{s,t} = \max_x(v_{x,t-1} * a_{x,s} * b_{s,o_t})$
  - $\text{path}_t = \arg\max_t(v_{x,t})$

Figure 15.9 shows the path for the first three states of the example.

Hidden page

ples. We can then use these forwards and backwards estimates to compute transition probabilities. It works like this. Suppose we want to compute the probability of a transition between state ω_i at time t and ω_j at time $t+1$. We first use our current model to run forwards via α to get to state ω_i at time t, and run backwards to get to state ω_j at time $t+1$ via β. Then we use the current estimates of a_{ij} and b_{jk}. The only other thing that we need is to realise that we might have got there by some other path through the states, so we need to normalise this calculation by how likely this particular training sequence is according to the current model, which is $P(O|a_{ij}, b_{jk})$. This value is usually called γ_{ij} and written out in all its glory looks like:

$$\gamma_{ij}(t) = \frac{\alpha_i(t-1)a_{ij}b_{jk}\beta_j(t)}{P(O|a_{ij}, b_{jk})}. \tag{15.16}$$

What is the meaning of $\sum_{t=1}^{T} \gamma_{ij}(t)$? It tells us how many times we can expect to transition from state ω_i to state ω_j at any time in the sequence. This is the expectation that we are using within this EM algorithm. We just need to work out how to do the maximisation bit. We know how many times we can expect to do the ω_i to ω_j transition, but a_{ij} is concerned with the probability of this event. So we need to divide this number by the number of times we expect to transition out of state ω_i, regardless of where we end up. This is $\sum_{t=1}^{T} \sum_m \gamma_{im}(t)$. The update rule for a_{ij} is simply the ratio of these:

$$a_{ij} = \frac{\sum_{t=1}^{T} \gamma_{ij}(t)}{\sum_{t=1}^{T} \sum_m \gamma_{im}(t)}. \tag{15.17}$$

The update rule for b_{jk} is similar, except that we need to think about the frequency that an observation o_k is made in state j compared to any other symbol:

$$b_{jk} = \frac{\sum_{t=1, o(t)=o_k}^{T} \sum_m \gamma_{km}(t)}{\sum_{t=1}^{T} \sum_m \gamma_{jm}(t)}. \tag{15.18}$$

This leads to the complete Baum-Welch algorithm:

The HMM Baum-Welch (Forward-Backward) Algorithm

- while updates have not converged:
  - **E-step:**
  - Compute forwards and backwards steps (α and β)
  - for each observation in order $o_t, t = 1 \ldots T$

    * for each possible pair of states s and σ:
        · $\gamma_{\sigma,s,t} = \alpha_{\sigma,t} * a_{\sigma,s} * \beta_{s,t+1} * b_{s,o(t+1)} / \max_x(\alpha_{x,T-1})$
- **M-step:**
- for each possible pair of states s and σ:
    * $a_{s,\sigma} = \sum_x \gamma_{s,\sigma,x} / \sum_y \sum_z \gamma_{s,y,z}$
- for each observation o:
    * for each state s:
        · tally $= \sum_x \sum_y \gamma_{s,x,y}$
        · $b_{s,o}$ =sum(tally where observation o was seen)/total tally

---

Since this is the most important algorithm, here is a Python implementation as well:

```
def BaumWelch(obs,nStates):

    T = shape(obs)[0]
    a = random.rand(nStates,nStates)
    b = random.rand(nStates,T)
    olda = zeros((nStates,nStates))
    oldb = zeros((nStates,T))
    maxCount = 50
    tolerance = 1e-5

    count = 0
    while (abs(a-olda)).max() > tolerance and
     (abs(b-oldb)).max() > tolerance and count < maxCount:
        # E-step

# Compute the forward and backward steps
        alpha = HMMfwd(a,b,obs)
        beta = HMMbwd(a,b,obs)
        gamma = zeros((nStates,nStates,T))

        for t in range(T-1):
          for s in range(nStates):
            gamma[:,s,t] = alpha[:,t] * a[:,s]
             ... * b[s,obs[t+1]] * beta[s,t+1]
             / max(alpha[:,T-1])

    # M-step
    olda = a.copy()
    oldb = b.copy()
```

```
            for i in range(nStates):
                for j in range(nStates):
                    a[i,j]=sum(gamma[i,j,:])/sum(sum(gamma[i,:,:]))

            for o in range(max(obs)):
                for j in range(nStates):
                    places = (obs==o).nonzero()
                    tally = sum(gamma[j,:,:],axis=0)
                    b[j,o]=sum(tally[places])/sum(sum(gamma[j,:,:
                    ]))

                count += 1
        return a,b
```

We can't really use this algorithm very well on the simple example, since we would need rather more data to do justice to the training. However, if we do apply it and then compute the Viterbi path, then it gives the same answer as with the invented data. That pretty much sums it up for the HMM. It is worth mentioning two limitations of it, which are that the probability distributions are not time dependent, and that the probabilities can get very small. The second of these problems is an implementation detail that needs careful monitoring, while the first can be dealt with by using more general graphical models, although with the additional computational costs that come with that.

15.4 Tracking Methods

We will now look at two methods of performing tracking. You perform tracking fairly easily, keeping tabs on where something is and how it is moving. This has an obvious evolutionary benefit, since keeping track of where predators were and whether they were coming towards you could keep you alive. It is also useful for a machine to be able to do this, both for similar reasons to a human or animal (watching something moving and predicting what path it will follow, for example in radar or other imaging method) and to keep track of a changing probability distribution. We will look at two methods of doing it, the Kalman filter and the particle filter.

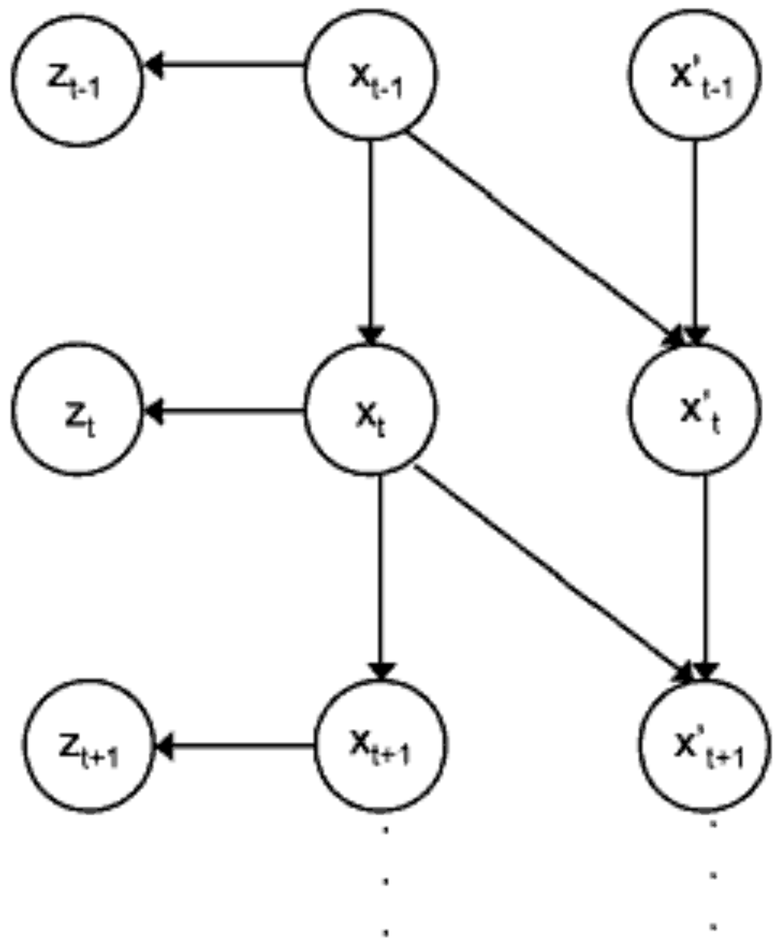


FIGURE 15.10: A representation of the Kalman filter as a graphical model.

15.4.1 The Kalman Filter

The Kalman filter (named for E. Kalman; although he was not the original inventor he did do quite a lot of work on it) is a recursive estimator. It makes an estimate of the next step, then computes an error term based on the value that was actually produced in the next step, and tries to correct it. It then uses both of those to make the next prediction, and iterates this procedure. It can be seen as a simple cycle of predict-correct behaviour, where the error at each step is used to improve the estimate at the next iteration. The Kalman filter can be represented by the graphical model shown in Figure 15.10.

Much of the jargon that is associated with the Kalman filter is familiar to us: the state, which is hidden, consists of the variables that we want to know, which we see through noisy observations over time. There is a transition model that tells us how states change from one to another, and an observation model (also called the sensor model here) that tells us how states lead to observations.

The principal simplifying assumptions of the Kalman filter are that the process is linear and that all of the distributions are Gaussian with constant covariance. Since the convolution of Gaussians is also Gaussian, this means that we can put them together to form new Gaussians, and so the model stays well behaved. This was a significant advantage over previous methods of tracking, which tended to stop working fairly quickly, since the estimates broke down because the probability distribution stopped being well-defined. We assume that both the transition model and the observation model are Gaussians with means based on the previous observations, and fixed covariances $\mathbf{Q}$ and $\mathbf{R}$:

$$P(\mathbf{x}_{t+1}|\mathbf{x}_t) = \mathcal{N}(\mathbf{x}_{t+1}|\mathbf{A}\mathbf{x}_t, \mathbf{Q}) \tag{15.19}$$

$$P(\mathbf{z}_{t+1}|\mathbf{x}_{t+1}) = \mathcal{N}(\mathbf{z}_{t+1}|\mathbf{H}\mathbf{x}_{t+1}, \mathbf{R}), \tag{15.20}$$

where $\mathbf{A}$ and $\mathbf{H}$ are matrices that provide a (constant) represention of the probability models, just as we had for the HMM above.

The idea of the Kalman filter is to make a prediction and then correct it when the next observation is available, i.e., at the next timestep. The predicted observation is $\hat{\mathbf{z}}_{t+1} = \mathbf{HAx}_{t+1}$ (by following the equations above) and so the error is $\mathbf{z}_{t+1} - \mathbf{HAx}_{t+1}$. The predicted covariance matrix that goes with it is $\hat{\mathbf{\Sigma}}_{t+1} = \mathbf{A}\mathbf{\Sigma}_t\mathbf{A}^T + \mathbf{Q}$. The Kalman filter weights these error computations by how much trust the filter currently has in its predictions; these weights are known as the Kalman gain and are computed by:

$$\mathbf{K}_{t+1} = \hat{\mathbf{\Sigma}}_{t+1}\mathbf{H}^T \left(\mathbf{H}\hat{\mathbf{\Sigma}}_{t+1}\mathbf{H}^T + \mathbf{R}\right)^{-1}. \tag{15.21}$$

So the update for the estimate is:

$$\mathbf{x}_{t+1} = \hat{\mathbf{x}}_{t+1} + \mathbf{K}_{t+1}\left(\mathbf{z}_{t+1} - \mathbf{H}\hat{\mathbf{x}}_{t+1}\right). \tag{15.22}$$

All that is then required is to update the covariance estimate:

$$\mathbf{\Sigma}_{t+1} = (\mathbf{I} - \mathbf{K}_{t+1}\mathbf{H})\hat{\mathbf{\Sigma}}_{t+1}, \tag{15.23}$$

where $\mathbf{I}$ is the identity matrix of the relevant size. Putting these equations together leads to a simple algorithm.

The Kalman Filter Algorithm

- given an initial estimate $\mathbf{x}(0)$
- for each timestep:
  - **predict the next step**
    - predict state as $\hat{\mathbf{x}}_{t+1} = \mathbf{Ax}_t$
    - predict covariance as $\hat{\mathbf{\Sigma}}_{t+1} = \mathbf{A}\mathbf{\Sigma}_t\mathbf{A}^T + \mathbf{Q}$
  - **update the estimate**
    - compute the error in the estimate, $\boldsymbol{\epsilon} = \mathbf{z}_{t+1} - \mathbf{HAx}_{t+1}$
    - compute the Kalman gain using Equation (15.21)
    - update the state using Equation (15.22)
    - update the covariance using Equation (15.23)

Figure 15.11 shows a simple 1D example of using the Kalman filter. The dots are noisy estimates of a process, and the line is the Kalman filter estimate, with the dashed lines being the possible error. It can be seen that the initial estimate was not very good, but the algorithm quickly converges to a good estimate of the mean of the data, and the error drops accordingly.

Now that we have seen the Kalman filter in action, we need to work out how to use it for tracking. If we write the position of the object at time t

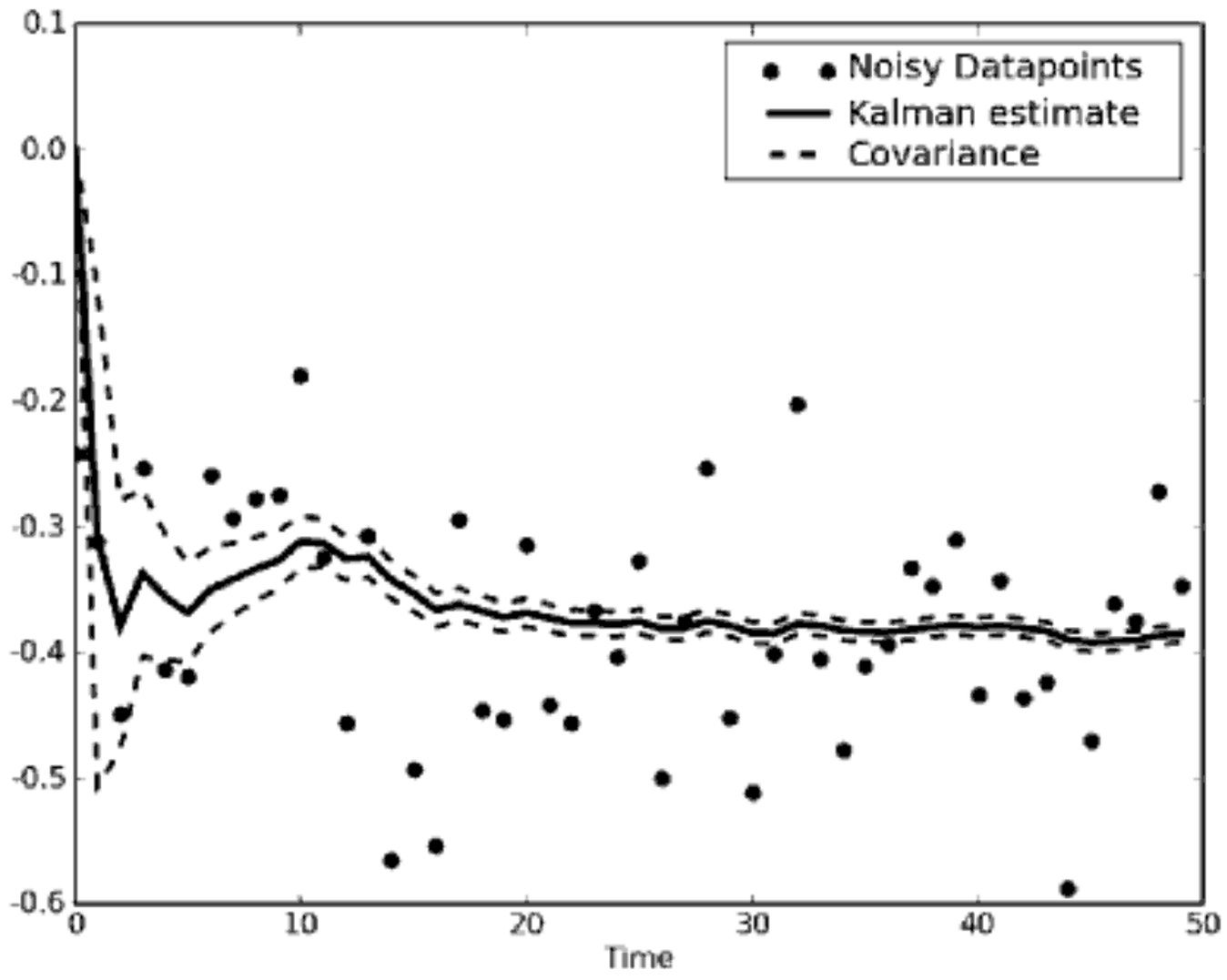


FIGURE 15.11: Estimates of a 1D noisy process using a Kalman filter.

as $\mathbf{y}_t$, then its velocity is the time derivative of its position, $\dot{\mathbf{y}}_t$. Now, the position at time $t + \Delta t$ is $\mathbf{y}_t + \dot{\mathbf{y}}\Delta t$. We are using the Kalman filter to keep track of both the position and the velocity of the object, so we write the state as:

$$\mathbf{x}_t = \begin{pmatrix} \mathbf{y} \\ \dot{\mathbf{y}} \end{pmatrix}. \tag{15.24}$$

The update equations will therefore be:

$$\mathbf{x}_{t+1} = \mathbf{A}\mathbf{x}_t + \mathbf{B}\mathbf{a}_{t+1}, \tag{15.25}$$

where the acceleration $\mathbf{a}_t$ is a Gaussian random variable with mean $\mathbf{0}$ and variance $\boldsymbol{\sigma}$, and:

$$\mathbf{A} = \begin{pmatrix} 1 & \Delta t \\ 0 & 1 \end{pmatrix} \quad \mathbf{B} = \begin{pmatrix} \frac{1}{2}\Delta t^2 \\ \Delta t \end{pmatrix}. \tag{15.26}$$

The same Kalman filter algorithm can then be used to perform the tracking. A 1D example of such tracking is shown in Figure 15.12, where an object is moving in one spatial dimension and time. The Kalman filter does a good job of following the object.

One of the main assumptions of the Kalman filter was that the process was linear. Where this is not true it is possible to linearise about the current estimate $(\mathbf{x}_t, \boldsymbol{\Sigma}_t)$ and this leads to the Extended Kalman Filter, where the main change is that you have to compute the Jacobian of the covariance matrix and use that in the update rule. This is not an optimal estimator, and there have

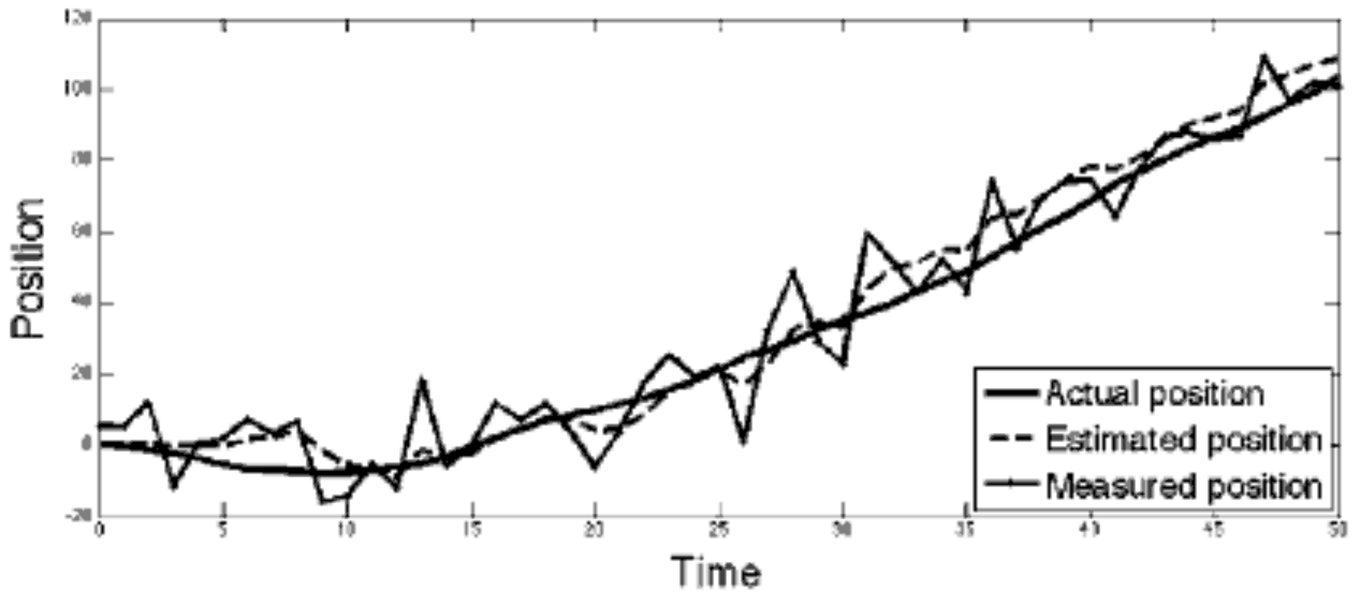


FIGURE 15.12: Demonstration of the Kalman filter tracking an object moving in one spatial dimension.

been various attempts to improve upon it; the most successful have been to use sampling, and the leads to the Unscented Kalman Filter, which uses a particular sampling technique known as the unscented transform. For more information on this, see the Further Reading section; instead we will look at a common MCMC algorithm for performing tracking, the particle filter.

15.4.2 The Particle Filter

There was one assumption that the Kalman filter made that we did not challenge at all, which was that the distributions were Gaussian. If this is not the case then we cannot use an algorithm like the Kalman filter, since the distributions would stop behaving as probabilities very quickly. In order to get around this problem, we return to the methods that have underpinned many of the algorithms in this chapter: sampling. The particular sampling technique that we will use is the sampling-importance-resampling algorithm of Section 14.3, which forms the basis of the particle filter, or condensation method. This is a relatively recent development, and has been finding many successful applications in tracking, including in image and signal analysis. The idea is to use sampling to keep track of the state of the probability distribution. This is actually known as sequential sampling, since we are using a set of samples for time t to estimate the process at time $t + 1$, and then resampling from there.

One benefit of sampling methods is that we don't have to hold on to the Markov assumption. In tracking, prior history can be useful, which means that the Markov assumption is a bad one. The proposal distribution is generally written as $q(\mathbf{x}_{t+1}|\mathbf{x}_{0:t}, \mathbf{z}_{0:t})$ to make this dependence clear, and the proposal distribution that is generally used in the estimated transition probabilities $p(\hat{\mathbf{x}}_{t+1}|\mathbf{x}_{0:t}, \mathbf{z}_{01:t})$, since it is a simple distribution that is related to the process. With this decided, there is nothing else to the particle filter. Figure 15.13 shows it keeping track of a distribution based on a noisy sinusoidal function. The plot on the right of the figure shows the particles that were created at each iteration.

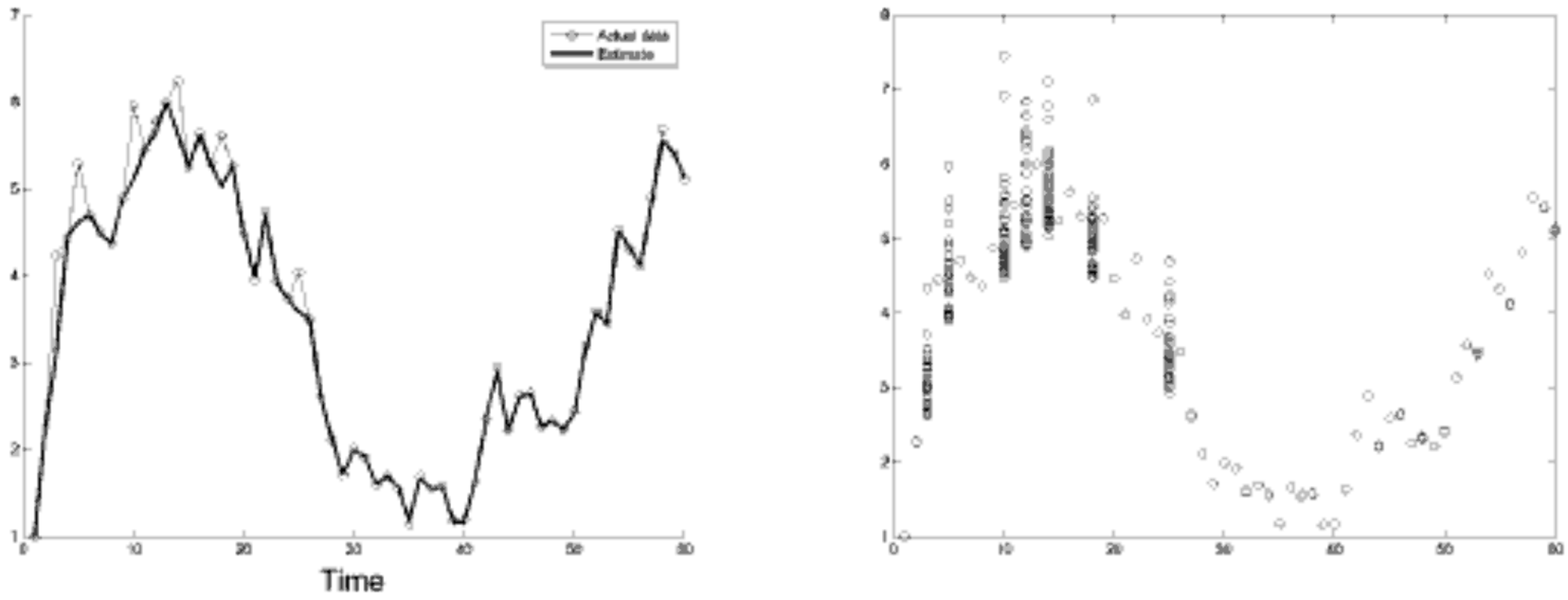


FIGURE 15.13: *Left:* A set of (noisy) data from a noisy distribution based on a sinusoid, and the results of a particle filter tracking the distribution. *Right:* The particles that were sampled at each iteration.

Further Reading

Graphical models are a growth area at the moment, with lots of interesting research being done in the area. The original work in the area, and the motivations for it, are described in:

- J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference.* Morgan Kaufmann, San Mateo, CA, USA, 1988.

Alternative overviews of Bayesian networks can be found in the following papers and books, the last of which is a collection of papers that provides a good overview of the area:

- W.L. Buntine. Operations for learning with graphical models. *Journal of Artificial Intelligence Research*, 2: 159–225, 1994.

- D. Husmeier. Introduction to learning Bayesian networks from data. In D. Husmeier, R. Dybowski, and S. Roberts, editors, *Probabilistic Modelling in Bioinformatics and Medical Informatics*, Springer, Berlin, Germany, 2005.

- Chapters 8 and 13 of C.M. Bishop. *Pattern Recognition and Machine Learning.* Springer, Berlin, Germany, 2006.

- M.I. Jordan, editor. *Learning in Graphical Models.* MIT Press, Cambridge, MA, USA, 1999.

In the area of Markov Random Fields, the image denoising example comes from:

- S. Geman and D. Geman. Stochastic relaxation, Gibbs distributions and the Bayesian restoration of images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 6:721–741, 1984.

Markov Random Fields are most commonly used in imaging. There are good overviews in:

- P. Pèrez. Markov random fields and images. *CWI Quarterly*, 11(4): 413–437, 1998.
- R. Kindermann and J.L. Snell. *Markov Random Fields and Their Applications.* American Mathematical Society, Providence, RI, USA, 1980.

For more details on the Hidden Markov Model, the Kalman filter and the particle filter, you might want to look at:

- L.R. Rabiner. A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–268, 1989.
- Z. Ghahramani. An introduction to Hidden Markov Models and Bayesian networks. *International Journal of Pattern Recognition and Artificial Intelligence*, 15:9–42, 2001.
- G. Welch and G. Bishop. An introduction to the Kalman filter, 1995. URL `http://www.cs.unc.edu/~welch/kalman/`. Technical Report TR 95-041, Department of Computer Science, University of North Carolina at Chapel Hill, USA.
- M.S. Arulampalam, S. Maskell, N. Gordon, and T. Clapp. A tutorial on particle filters for online nonlinear/non-Gaussian Bayesian tracking. *IEEE Transactions on Signal Processing*, 50(2):174–188, 2002.

A more rigorous treatment is given in:

- Chapters 8 and 13 of C.M. Bishop. *Pattern Recognition and Machine Learning.* Springer, Berlin, Germany, 2006.

Practice Questions

Problem 15.1 Compute the probability of taking notes (N) in the Bayesian network shown in Figure 15.14. The problem describes the chance of you taking notes in the lecture or sleeping (S) according to whether or not the course was boring (B) based on whether or not the professor is

Hidden page

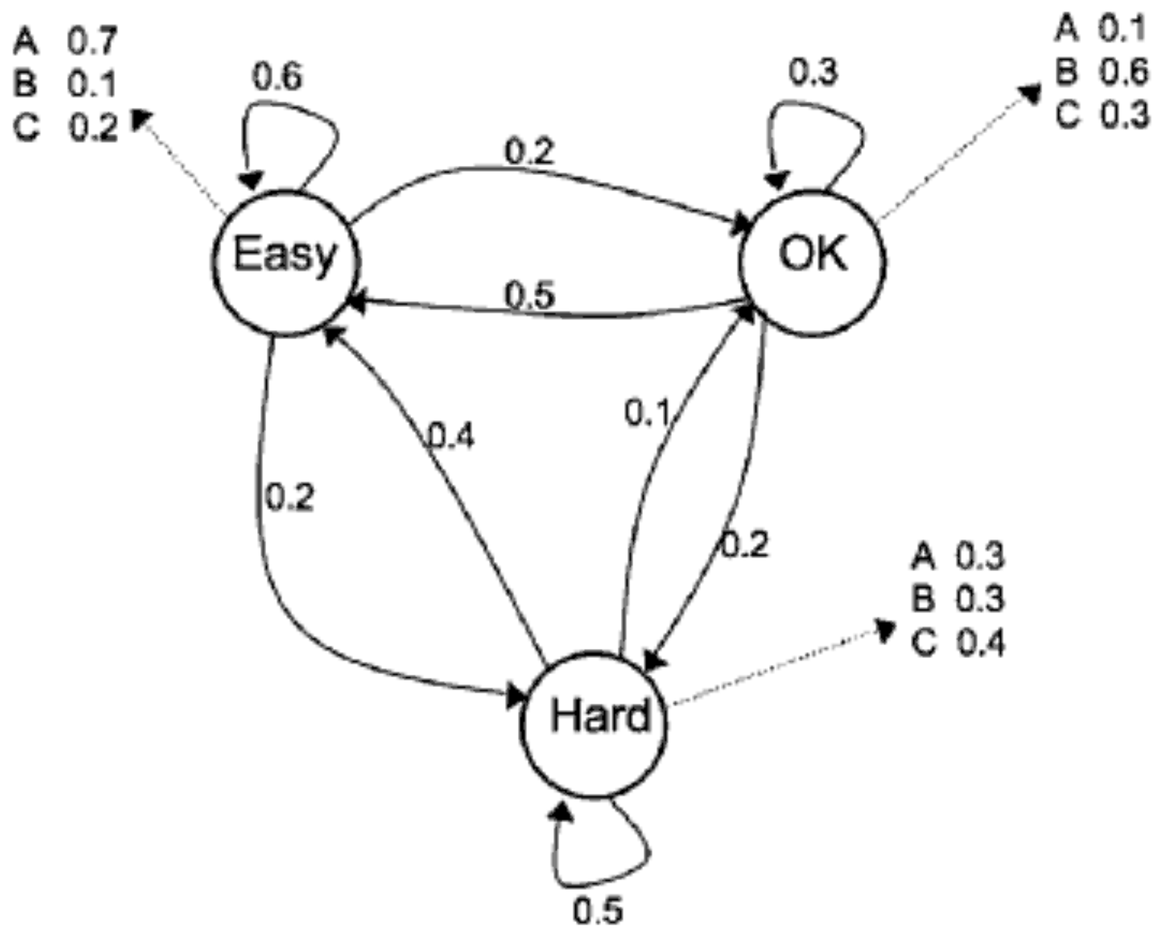


FIGURE 15.15: The Hidden Markov Model example for Problem 15.3.

Hidden page

Hidden page

same object. This might not seem important, but Python works by reference, which means that the command `>>> a = b` does not put a copy of the value of `b` into `a`, but rather assigns to `a` a reference to the variable `b`. This can be a trap for the unwary, as will be discussed more shortly. The normal logical operators are slightly unusual in Python, with the normal logical operators using the words `and`, `or` and `not`; the symbols `&`, `|` perform bit-wise and/or. These bit-wise operators are actually quite useful, as we'll see later.

In addition to integer and floating point representations of numbers, Python also deals with strings, which are described by using single or double quotes (' or ") to surround them: `>>> b = 'hello'`. For strings, the `+` operator is overloaded (given a new meaning), which is concatenation: merging the strings. So `>>> 'a' + 'd'` returns the new string `'ad'`.

Having made the basic data types, Python then allows you to combine them into three different basic data structures:

Lists A list is a combination of basic data types, surrounded by square brackets. So `>>> mylist = [0, 3, 2, 'hi']` is a perfectly good list that contains integers and a string. This ability to store different types inside a list gives you a hint that Python handles lists differently to the way other languages handle arrays. This comes about because Python is inherently object-oriented, so that every variable that you make is simply an object, and so a list is just a collection of objects. This is why the type of the object does not matter. It also means that you can have lists of lists without a problem: `>>> newlist = [3, 2, [5, 4, 3], [2, 3, 2]]`.

Accessing particular elements of a list simply requires giving it an index. Like C, but unlike MATLAB, Python indices start at 0, so `>>> newlist[0]` returns the first element (3). You can also index from the end using a minus sign, so `>>> newlist[-1]` returns the last element, `>>> newlist[-2]` the last-but-one, etc. The length of a list is given by `len`, so `>>> len(newlist)` returns 4. Note that `>>> newlist[3]` returns the list in the 4th location of `newlist` (i.e., `[2, 3, 2]`). To access an element of that list you need an extra index: `>>> newlist[3][1]` returns 3.

A useful feature of Python is the slice operator. This is written as a colon (`:`) and enables you to access sections of a list easily, such as `>>> newlist[2:4]` which returns the elements of `newlist` in positions 2 and 3 (the arguments you use in a slice are inclusive at the start and exclusive at the end, so the second parameter is the first index that is excluded). In fact, the slice can take three operators, which are [start:stop:step], the third element saying what stepsize to use. So `>>> newlist[0:4:2]` returns the elements in locations 0 and 2, and you can use this to reverse the order of a list: `>>> newlist[::-1]`. This last example shows a couple of other refinements of the slice operator: if

you don't put a value in for the first number (so it looks like `[:3]`) then the value is taken as 0, and if you don't put a value for the second operator (`[1:]`) then it is taken as running to the end of the list. These can be very useful, especially the second one, since it avoids having to calculate the length of the list every time you want to run through it. `>>> newlist[:]` returns the whole string.

This last use of the slice operator, returning the whole string, might seem useless. However, because Python is object-oriented, all variable names are simply **references** to objects. This means that copying a variable of type `list` isn't as obvious as it could be. Consider the following command: `>>> alist = mylist`. You might expect that this has made a copy of `mylist`, but it hasn't. To see this, use the following command `>>> alist[3] = 100` and then have a look at the contents of `mylist`. You will see that the 3rd element is now 100. So if you want to copy things you need to be careful. The slice operator lets you make actual copies using: `>>> alist = mylist[:]`. Unfortunately, there is an extra wrinkle in this if you have lists of lists. Remember that lists work as references to objects. We've just used the slice operator to return the values of the objects, but this only works for one level. In location 2 of `newlist` is another list, and the slice operator just copied the reference to that embedded list. To see this, perform `>>> blist = newlist[:]` and then `>>> blist[2][2] = 100` and have a look at `newlist` again. What we've done is called a **shallow copy**, to copy everything (known as a **deep copy**) requires a bit more effort. There is a `deepcopy` command, but to get to it we need to `import` the `copy` module using `>>> import copy` (we will see more about importing in Section 16.3.1). Now we can call `>>> clist = copy.deepcopy(newlist)` and we finally have a copy of a complete list.

There are a variety of functions that can be applied to lists, but there is another interesting feature of the fact that they are objects. The functions (methods) that can be used are part of the object class, so they modify the list itself and do not return a new list (this is known as working **in place**). To see this, make a new list `>>> list = [3, 2, 4, 1]` and suppose that you want to print out a list of the numbers sorted into order. There is a function `sort()` for this, but the obvious `>>> print list.sort()` produces the output `None`, meaning that no value was returned. However, the two commands `>>> list.sort()` followed by `>>> print list` do exactly what is required. So functions on lists modify the list, and any future operations will be applied to this modified list.

Some other functions that are available to operate on lists are:

`append(x)` adds `x` to the end of the list

`count(x)` counts how many times `x` appears in the list

`extend(L)` adds the elements in list `L` to the end of the original list

`index(x)` returns the index of the first element of the list to match `x`

`insert(i, x)` inserts element `x` at location `i` in the list, moving everything else along

`pop(i)` removes the item at index `i`

`remove(x)` deletes the first element that matches `x`

`reverse()` reverses the order of the list

`sort()` we've already seen

You can compare lists using `>>> a==b`, which works elementwise through the list, comparing each element against the matching one in the second list, returning True if the test is true for each pair (and the two lists are the same length), and False otherwise.

Tuples A tuple is an immutable list, meaning that it is read-only and doesn't change. Tuples are defined using round brackets, e.g., `>>> mytuple = (0, 3, 2, 'h')`. It might seem odd to have them in the language, but they are useful if you want to create lists that cannot be modified, especially by mistake.

Dictionaries In the list that we saw above we indexed elements by their position within the list. In a dictionary you assign a key to each entry that you can use to access it. So suppose you want to make a list of the number of days in each month. You could use a dictionary (shown by the curly braces): `>>> months = {'Jan': 31, 'Feb': 28, 'Mar': 31}` and then you access elements of the dictionary using their key, so `>>> months['Jan']` returns 31. Giving an incorrect key results in an exception error.

The function `months.keys()` returns a list of all the keys in the dictionary, which is useful for looping over all elements in a dictionary. The `months.values()` function returns a list of values instead, while `months.items()` gives a list of tuples containing everything. There are lots of other things you can do with dictionaries, and we shall see some of them when we use the dictionary in Chapter 6.

There is one more data type that is built directly into Python, and that is the `file`. This makes reading from and writing to files very simple in Python: files are opened using `>>> input = open('filename')`, closed using `>>> input.close()` and reading and writing are performed using `readlines()` (and `read()`, and `writelines()` and `write()`). There are also `readline()` and `writeline()` functions, that read and write one line at a time.

Hidden page

Hidden page

The most common loop is the `for` loop, which differs slightly from other languages in that it iterates over a list of values:

```
for var in set:
    commands
else:
    commands
```

There is one very useful command that goes with this `for` loop, which is the `range` command, which produces a list output. Its most basic form is simply `>>> range(4)`, which produces the list `[0, 1, 2, 3]`. However, it can also take 2 or 3 arguments, and works in the same way as in the slice command, but with commas between them instead of colons: `>>> range(start,stop,step)`. This can include going down instead of up a list, so `>>> range(5,-3,-2)` produces `[5, 3, 1, -1]` as output.

Finally, there is a `while` loop:

```
while condition:
    commands
else:
    commands
```

16.3.3 Functions

Functions are defined by:

```
def name(args):
    commands
    return value
```

The `return value` line is optional, but enables you to return values from the function (otherwise it returns `None`). You can list several things to return in the line with commas between them, and they will all be returned. Once you have defined a function you can call it from the command line and from within other functions. Python is case sensitive, so with both function names and variable names, `Name` is different to `name`.

As an example, here is a function that computes the hypotenuse of a triangle given the other two distances (`x` and `y`). Note the use of '#' to denote a comment:

```
def pythagorus(x,y):
    """ Computes the hypotenuse of two arguments"""
    h = pow(x**2+y**2,0.5)
    # pow(x,0.5) is the square root
    return h
```

Now calling `pythagorus(3,4)` gets the expected answer of `5.0`. You can also call the function with the parameters in any order provided that you specify which is which, so `pythagorus(y=4,x=3)` is perfectly valid. When you make functions you can allow for default values, which means that if fewer arguments are presented the default values are given. To do this, modify the function definition line: `def pythagorus(x=3,y=4):`

16.3.4 The `doc` String

The help facilities within Python are accessed by using `help()`. For help on a particular module, use `help('modulename')`. (So using `help(pythagorus)` in the previous example would return the description of the function that is given there). A useful resource for most code is the `doc` string, which is the first thing defined within the function, and is a text string enclosed in three sets of double quotes (`"""`). It is intended to act as the documentation for the function or class. It can be accessed using `>>> print functionname.__doc__`. The Python documentation generator `pydoc` uses these strings to automatically generate documentation for functions, in the same way that `javadoc` does.

16.3.5 `map` and `lambda`

Python has a special way of performing repeated function calls. If you want to apply the same function to every element of a list you don't need to loop over the elements of the list, but can instead use the `map` command, which looks like `map(function,list)`. This applies the function to every element of the list. There is one extra tweak, which is the fact that the function can be anonymous (created just for this job without needing a name) by using the `lambda` command, which looks like `lambda args : command`. A `lambda` function can only execute one command, but it enables you to write very short code to do relatively complicated things. As an example, the following instruction takes a list and cubes each number in it and adds 7:

```
map(lambda x:pow(x,3)+7,list)
```

Another way that `lambda` can be used is in conjunction with the `filter` command. This returns elements of a list that evaluate to `True`, so:

```
filter(lambda x:x>=2,list)
```

returns those elements of the list that are greater than or equal to 2. NumPy provides simpler ways to do these things for arrays of numbers, as we shall see.

16.3.6 Exceptions

Like other modern languages, Python allows for the trapping of exceptions. This is done through the `try ... except ... else` and `try... finally` constructions. This example shows the use of the most common version. For more details, including the types of exceptions that are defined, see a Python programming book.

```
try:
    x/y
except ZeroDivisonError:
    print "Divisor must not be 0"
except TypeError:
    print "They must be numbers"
except:
    print "Something unspecified went wrong"
else:
    print "Everything worked"
```

16.3.7 Classes

For those that wish to use it in this way, Python is fully object-oriented, and classes are defined (with their constructor) by:

```
class myclass(superclass):

    def __init__(self,args):

    def functionname(self,args):
```

If a superclass is not specified then the class does not inherit from elsewhere. The `__init__(self,args)` function is the constructor for the class. There can also be a destructor `__del__(self)`, although they are rarely used. Accessing methods from the class uses the `classname.functionname()` syntax. The `self` argument can be ignored in all function calls, since Python fills it in for you, but it does need to be specified in the function definition. Many of the

examples in the book are based on classes provided on the book website. You need to be aware that you have to create an instance of the class before you can run it. So to import and run the class you need to use:

```
import myclass
var = myclass.myclass()
var.function()
```

16.4 Using NumPy and Matplotlib

Most of the commands that are used in this book actually come from the NumPy and Matplotlib packages, rather than the basic Python language. More specialised commands are described thoughout the book in the places where they become relevant. There are lots of examples of performing tasks using the various functions within NumPy on its website. Getting information about functions within NumPy is easy, because there is a special command: `info()`; for example, to find out about the sum command, use `info(sum)`.

NumPy has a base collection of functions and then additional packages that have to be imported as well if you want to use them. To import the NumPy base library and get started you use:

```
>>> from numpy import *
```

16.4.1 Arrays

The basic data structure that is used for numerical work, and by far the most important one for the programming in this book, is the array. This is exactly like multi-dimensional arrays (or matrices) in any other language; it consists of one or more dimensions of numbers or chars. Unlike Python lists, the elements of the array all have the same type, which can be Boolean, integer, real, or complex numbers.

Arrays are made using a function call, and the values are passed in as a list, or set of lists for higher dimensions. Here are one-dimensional and two-dimensional arrays (which are effectively arrays of arrays) being made. Arrays can have as many dimensions as you like up to a language limit of 40 dimensions, which is more than enough for this book.

```
>>> myarray = array([4,3,2])
>>> mybigarray = array([[3, 2, 4], [3, 3, 2], [4, 5, 2]])
```

Hidden page

`r_[]` **and** `c_[]` Perform row and column concatenation, including the use of the slice operator: `r_[1:4,0,4] = array([1, 2, 3, 0, 4])`. There is also a variation on `linspace()` using a `j` in the last entry: `r_[2,1:7:3j] = array([ 2.  , 1.  , 4.  , 7.])`. This is another nice feature of NumPy that can be used with `arange()` and `mgrid()` as well. The `j` on the last value specifies that you want 3 equally spaced points starting at 0 and running up to (and including) 7, and the function works out the locations of these points for you. The column version is similar.

The array `a` used in the next set of examples was made using `>>> a = arange(6).reshape(3,2)`, which produces:

```
array([[0, 1],
       [2, 3],
       [4, 5]])
```

Indexing elements of an array is performed using square brackets '[' and ']', remembering that indices start from 0. So `a[2,1]` returns `5` and `a[:,1]` returns `array([1, 3, 5])`. We can also get various pieces of information about an array and change it in a variety of different ways, as follows.

Getting information about arrays, changing their shape, copying them

---

`ndim(a)` Returns the number of dimensions (here `2`).

`size(a)` Returns the number of elements (here `6`).

`shape(a)` Returns the size of the array in each dimension (here `(3, 2)`). You can access the first element of the result using `shape(a)[0]`.

`reshape(a,(2,3))` Reshapes the array as specified. Note that the new dimensions are in brackets. One nice thing about `reshape()` is that you can use '-1' for 1 dimension within the reshape command to mean 'as many as is required.' This saves you doing the multiplication yourself. For this example, you could use `reshape(a,(2,-1))` or `reshape(a,(-1,2))`.

`ravel(a)` Makes the array one-dimensional (here `array([0, 1, 2, 3, 4, 5])`).

`transpose(a)` Compute the matrix transpose. For the example:

```
[[0 2 4]
 [1 3 5]]
```

`a[::-1]` Reverse the elements of each dimension.

`a.min()`, `a.max(a)`, `a.sum(a)` Returns the smallest or largest element of the matrix, or the sum of the elements. Often used to sum the rows or columns using the `axis` option: `a.sum(axis=0)` for columns and `a.sum(axis=1)` for rows.

Hidden page

Hidden page

`linalg.pinv(a)` Compute the pseudo-inverse, which is defined even if `a` is not square

`linalg.det(a)` Compute the determinant of `a`

`linalg.eig(a)` Compute the eigenvalues and eigenvectors of `a`

16.4.4 Plotting

The plotting functions that we will be using are in the Matplotlib package. These are designed to look exactly like the MATLAB plotting functions. The entire set of functions, with examples, are given on the Matplotlib webpage, but the two most important ones that we will need are `plot` and `hist`. In order to use Matplotlib, you have to import it. For some reason it is called `pylab`, so the relevant command is `>>> from pylab import *`, which gives you access to the plotting commands. Matplotlib also provides some MATLAB functionality, and sometimes this can overwrite NumPy functions. The best way to avoid this is to ensure that you always import PyLab before NumPy. When producing plots they sometimes do not appear. This is usually because you need to specify the command `>>> ion()` which turns interactive plotting on. If you are using Matplotlib within Eclipse it has a nasty habit of closing all of the display windows when the program finishes. To get around this, issue a `show()` command at the end of your function.

The basic plotting commands of Matplotlib are demonstrated here, for more advanced plotting facilities see the package webpage.

The following code (best typed into a file and executed as a script) computes a Gaussian function for values -2 to 2.5 in steps of 0.01 and plots it, then labels the axes and gives the figure a title:

```
from pylab import *
from numpy import *

gaussian = lambda x: exp(-(0.5-x)**2/1.5)
x = arange(-2,2.5,0.01)
y = gaussian(x)
plot(x,y)
xlabel('x values')
ylabel('exp(-(0.5-x)**2/1.5')
title('Gaussian Function')
show()
```

The output of running this piece of code is shown in Figure 16.1.

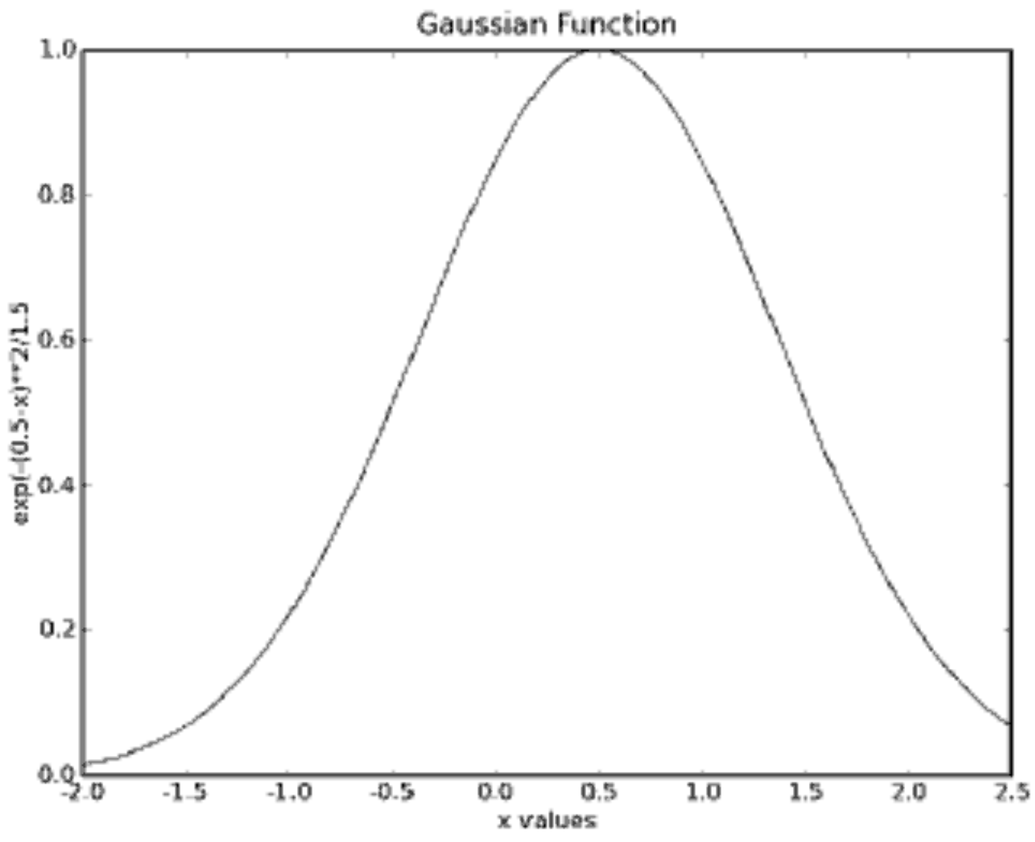


FIGURE 16.1: The Matplotlib package produces useful graphical output, such as this plot of the Gaussian function.

Further Reading

Python has become incredibly popular for both general computing and scientific computing. Because writing extension packages for Python is simple (it does not require any special programming commands: any Python module can be imported as a package, as can packages written in C), many people have done so, and made their code available on the Internet. Any search engine will find many of these, but a good place to start is the Python Cookbook website.

If you are looking for more complete introductions to Python, some of the following may be useful:

- M.L. Hetland. *Beginning Python: From Novice to Professional.* Apress Inc., Berkeley, CA, USA, 2nd edition, 2008.
- G. van Rossum and F.L. Drake Jr., editors. *An Introduction to Python.* Network Theory Ltd, Bristol, UK, 2006.
- W.J. Chun. *Core Python Programming.* Prentice-Hall, New Jersey, USA, 2006.
- B. Eckel. *Thinking in Python.* Mindview, La Mesa, CA, USA, 2001.
- T. Oliphant. Guide to NumPy, e-book, 2006. The official guide to NumPy by its creator.

Practice Questions

- Make an array `a` of size 6×4 where every element is a 2.
- Make an array `b` of size 6×4 that has 3 on the leading diagonal and 1 everywhere else. (You can do this without loops.)
- Can you multiply these two matrices together? Why does `a * b` work, but not `dot(a,b)`?
- Compute `dot(a.transpose(),b)` and `dot(a,b.transpose())`. Why are the results different shapes?
- Write a function that prints some output on the screen and make sure you can run it in Eclipse or whichever environment you are using.
- Now write one that makes some random arrays and prints out their sums, the mean value, etc.
- Write a function that consists of a set of loops that run through an array and count the number of ones in it. Do the same thing using the `where()` function (use `info(where)` to find out how to use it).

Index

Computer Science/Computer Engineering

Chapman & Hall/CRC
Machine Learning & Pattern Recognition Series

Drawing from computer science, statistics, mathematics, and engineering, the multidisciplinary nature of machine learning is underscored by its applicability to areas ranging from finance to biology and medicine to physics and chemistry. **Machine Learning: An Algorithmic Perspective** provides an introduction to the field of machine learning covering all of the major areas, including neural networks, graphical models, reinforcement learning, evolutionary algorithms, dimensionality reduction methods, and the important area of optimization.

The book treads the fine line between adequate academic rigor and overwhelming readers with equations and mathematical concepts. The author addresses the topics in a practical way while providing complete information and references where other expositions can be found. He describes algorithms with code examples backed up by a website that provides working implementations in Python. The book also contains data from a variety of applications to demonstrate the methods.

Features

- Provides a clear introduction to the basic concepts of machine learning
- Focuses on algorithms and applications and uses explanation rather than equations and mathematical concepts
- Presents real-world problems through structured exercises and programming examples in Python
- Contains a chapter that introduces the use of Python

Written in an easily accessible style, this book bridges the gaps between disciplines, providing the ideal blend of theory and practical, applicable knowledge.

CRC Press
Taylor & Francis Group
an **informa** business
www.crcpress.com

6000 Broken Sound Parkway, NW
Suite 300, Boca Raton, FL 33487
270 Madison Avenue
New York, NY 10016
2 Park Square, Milton Park
Abingdon, Oxon OX14 4RN, UK

C6718

ISBN: 978-1-4200-6718-7
90000
9 781420 067187