A CRC Press FreeBook

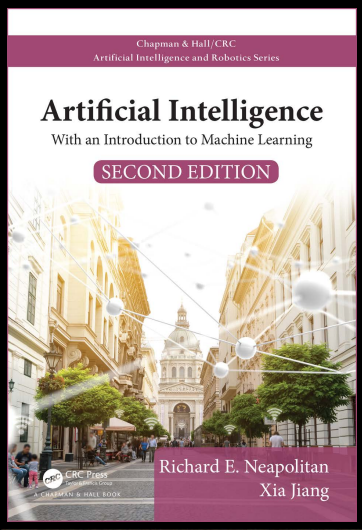
Explorations in Artificial Intelligence and Machine Learning

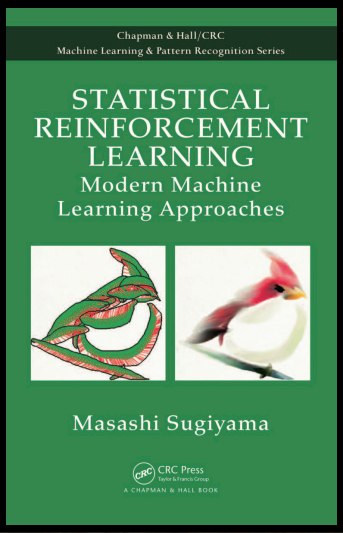
CRC Press
Taylor & Francis Group

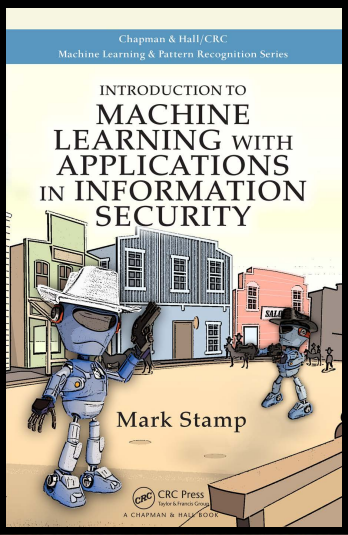
TABLE OF CONTENTS

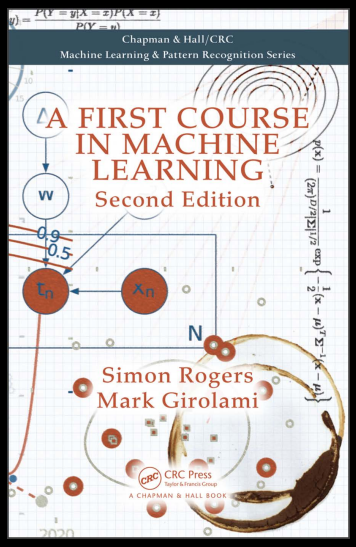
- Introduction (Prof. Roberto V. Zicari)
- 1 • Introduction to Machine Learning
- 2 • The Bayesian Approach to Machine Learning
- 3 • A Revealing Introduction to Hidden Markov Models
- 4 • Introduction to Reinforcement Learning
- 5 • Deep Learning for Feature Representation
- 6 • Neural Networks and Deep Learning
- 7 • AI-Completeness: The Problem Domain of Super-intelligent Machines

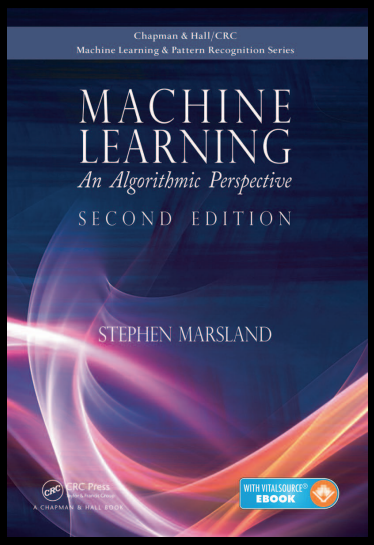
READ THE LATEST ON ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING WITH THESE KEY TITLES

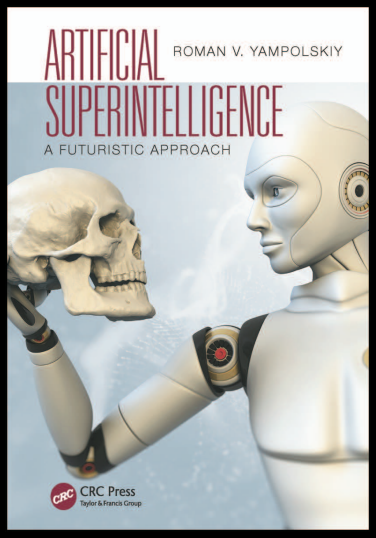












VISIT

WWW.CRCPRESS.COM/COMPUTER-SCIENCE-ENGINEERING

TO BROWSE OUR FULL RANGE OF TITLES

SAVE 20% AND GET **FREE SHIPPING** WITH DISCOUNT CODE **ODB18**

Introduction by Prof. Roberto V. Zicari

Frankfurt Big Data Lab, Goethe University Frankfurt

Editor of ODBMS.org

Artificial Intelligence (AI) seems the defining technology of our time.

Google has just re-branded its Google Research division to Google AI as the company pursues developments in the field of artificial intelligence.

John McCarthy defines AI, back in 1956 like this: "AI involves machines that can perform tasks that are characteristic of human intelligence".

This Free Book gives you a brief introduction to Artificial Intelligence, Machine Learning, and Deep Learning.

But, what are the main differences between Artificial Intelligence, Machine Learning, and Deep Learning?

To put it simply, Machine Learning is a way of achieving AI.

Arthur Samuel's definition of Machine Learning (ML) is from 1959: "Machine Learning: Field of study that gives computers the ability to learn without being explicitly programmed".

Typical problems solved by Machine Learning are:

- Regression.
- Classification.
- Segmentation.
- Network analysis.

What has changed dramatically since those pioneering days is the rise of Big Data and of computing power, making it possible to analyze massive amounts of data at scale!

AI needs Big Data and Machine Learning to scale.

Machine learning is a way of “training” an algorithm so that it can learn.

Huge amounts of data are used to train algorithms and allowing algorithms to "learn" and improve.

Deep Learning is a subset of Machine Learning and was inspired by the structure and function of the brain.

As an example, Artificial Neural Networks (ANNs), are algorithms that resemble the biological structure of the brain, namely the interconnecting of many neurons.

This Free Book gives a gentle introduction to Machine Learning, lists various ML approaches such as decision tree learning, Hidden Markov Models, reinforcement learning, Bayesian networks, as well as covering some aspects of Deep Learning and how this relates to AI.

It should help you achieve an understanding of some of the advances in the field of AI and Machine Learning while giving you an idea of the specific skills you'll need to get started if you wish to work as a Machine Learning Engineer.

About the Editor:

Prof. Dott-Ing. Roberto V. Zicari is Full Professor of Database and Information Systems at Frankfurt University. He was for more than 15 years a representative of the OMG in Europe. Previously, Zicari served as associate professor at Politecnico di Milano, Italy; Visiting scientist at IBM Almaden Research Center, USA, and the University of California at Berkeley, USA; Visiting professor at EPFL in Lausanne, Switzerland, at the National University of Mexico City, Mexico and at the Copenhagen Business School.

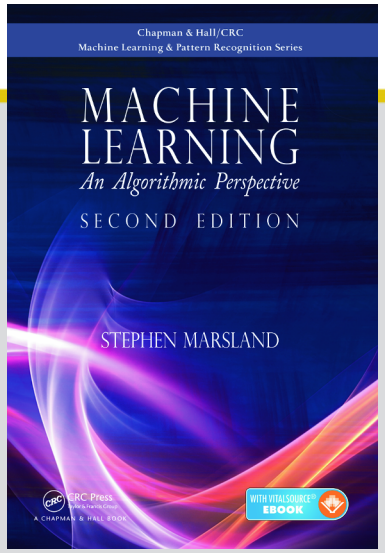
About the ODBMS.org Portal:

Launched in 2005, ODBMS.ORG was created to serve faculty and students at educational and research institutions as well as software developers in the open source community or at commercial companies.

It is designed to meet the fast-growing need for resources focusing on Big Data, Analytical data platforms, Cloud platforms, Graphs Databases, In-Memory Databases, NewSQL Databases, NoSQL databases, Object databases, Object-relational bindings, RDF Stores, Service platforms, and new approaches to concurrency control

CHAPTER 1

INTRODUCTION TO MACHINE LEARNING



This chapter is excerpted from

Machine Learning: An Algorithmic Perspective, 2nd Ed.

by Stephen Marsland.

© 2018 Taylor & Francis Group. All rights reserved.

Learn more

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

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

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

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

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

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

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

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

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

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

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

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

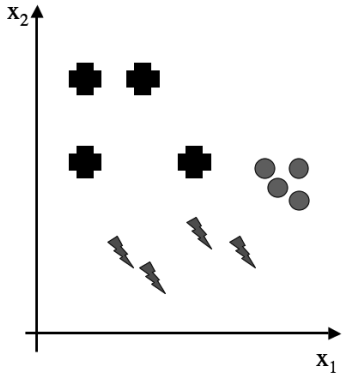


FIGURE 1.1 A set of datapoints as numerical values and as points plotted on a graph. It is easier for us to visualise data than to see it in a table, but if the data has more than three dimensions, we can't view it all at once.

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

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

1.2 LEARNING

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

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

1.2.1 Machine Learning

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

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

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

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

1.3 TYPES OF MACHINE LEARNING

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

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

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

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

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

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

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

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

1.4 SUPERVISED LEARNING

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

1.4.1 Regression

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

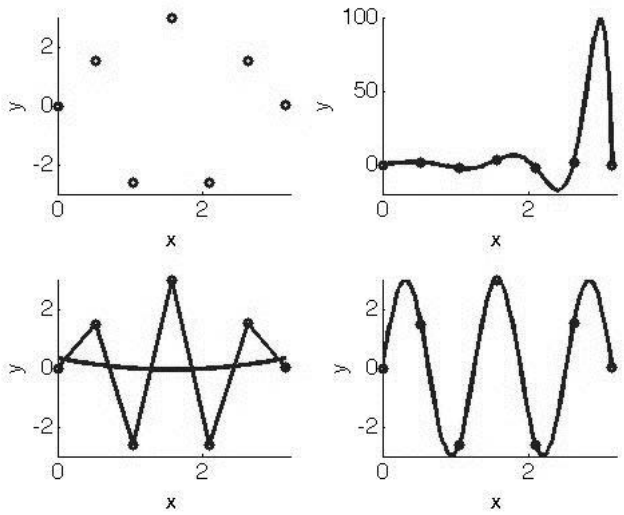


FIGURE 1.3 *Top left:* A few datapoints from a sample problem. *Bottom left:* Two possible ways to predict the values between the known datapoints: connecting the points with straight lines, or using a cubic approximation (which in this case misses all of the points). *Top and bottom right:* Two more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

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

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

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

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

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

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

1.4.2 Classification

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

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

FIGURE 1.4 The New Zealand coins.

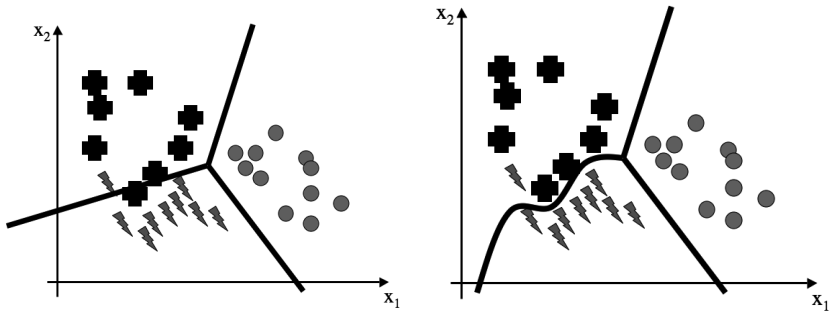


FIGURE 1.5 *Left:* A set of straight line decision boundaries for a classification problem. *Right:* An alternative set of decision boundaries that separate the plusses from the lightening strikes better, but requires a line that isn't straight.

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

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

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

1.5 THE MACHINE LEARNING PROCESS

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

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

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

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

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

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

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

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

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

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

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

1.6 A NOTE ON PROGRAMMING

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

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

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

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

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

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

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

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

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

1.7 A ROADMAP TO THE BOOK

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

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

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

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

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

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

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

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

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

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

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

FURTHER READING

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

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

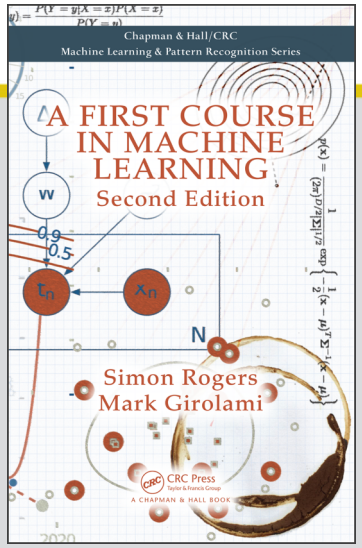
Other texts that provide alternative views of similar material include:

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

CHAPTER

2

THE BAYESIAN APPROACH TO MACHINE LEARNING



This chapter is excerpted from

A First Course in Machine Learning, Second Edition

by Simon Rogers and Mark Girolami

© 2018 Taylor & Francis Group. All rights reserved.

Learn more

In the previous chapter, we saw how explicitly adding noise to our model allowed us to obtain more than just point predictions. In particular, we were able to quantify the uncertainty present in our parameter estimates and our subsequent predictions. Once content with the idea that there will be uncertainty in our parameter estimates, it is a small step towards considering our parameters themselves as random variables. Bayesian methods are becoming increasingly important within Machine Learning and we will devote the next two chapters to providing an introduction to an area that many people find challenging. In this chapter, we will cover some of the fundamental ideas of Bayesian statistics through two examples. Unfortunately, the calculations required to perform Bayesian inference are often not analytically tractable. In Chapter 4 we will introduce three approximation methods that are popular in the Machine Learning community.

3.1 A COIN GAME

Imagine you are walking around a fairground and come across a stall where customers are taking part in a coin tossing game. The stall owner tosses a coin ten times for each customer. If the coin lands heads on six or fewer occasions, the customer wins back their £1 stake plus an additional £1. Seven or more and the stall owner keeps their money. The binomial distribution (described in Section 2.3.2) describes the probability of a certain number of successes (heads) in N binary events. The probability of y heads from N tosses where each toss lands heads with probability r is given by

$$P(Y = y) = \begin{pmatrix} N \\ y \end{pmatrix} r^y (1-r)^{N-y}. \tag{3.1}$$

You assume that the coin is fair and therefore set $r = 0.5$. For $N = 10$ tosses, the probability distribution function can be seen in Figure 3.1, where the bars corresponding to $y \leq 6$ have been shaded. Using Equation 3.1, it is possible to calculate the probability of winning the game, i.e. the probability that Y is less than or equal

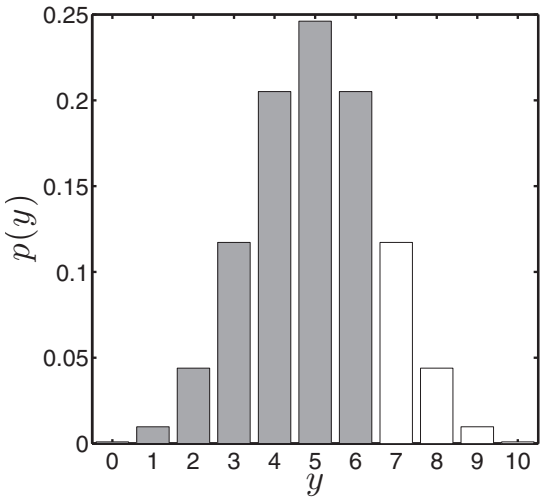


FIGURE 3.1 The binomial density function (Equation 3.1) when $N = 10$ and $r = 0.5$.

to 6, $P(Y \leq 6)$:

$$\begin{aligned} P(Y \leq 6) = 1 - P(Y > 6) &= 1 - [P(Y = 7) + P(Y = 8) \\ &\quad + P(Y = 9) + P(Y = 10)] \\ &= 1 - [0.1172 + 0.0439 + 0.0098 + 0.0010] \\ &= 0.8281. \end{aligned}$$

This seems like a pretty good game – you'll double your money with probability 0.8281. It is also possible to compute the expected return from playing the game. The expected value of a function $f(X)$ of a random variable X is computed as (introduced in Section 2.2.8)

$$\mathbf{E}_{P(x)}\{f(X)\} = \sum_x f(x)P(x),$$

where the summation is over all possible values that the random variable can take. Let X be the random variable that takes a value of 1 if we win and a value of 0 if we lose: $P(X = 1) = P(Y \leq 6)$. If we win, $(X = 1)$, we get a return of £2 (our original stake plus an extra £1) so $f(1) = 2$. If we lose, we get a return of nothing so $f(0) = 0$. Hence our expected return is

$$f(1)P(X = 1) + f(0)P(X = 0) = 2 \times P(Y \leq 6) + 0 \times P(Y > 6) = 1.6562.$$

Given that it costs £1 to play, you win, on average, $1.6562 - 1$ or approximately $66p$ per game. If you played 100 times, you'd expect to walk away with a profit of £65.62.

Given these odds of success, it seems sensible to play. However, whilst waiting you notice that the stall owner looks reasonably wealthy and very few customers seem to

be winning. Perhaps the assumptions underlying the calculations are wrong. These assumptions are

1. The number of heads can be modelled as a random variable with a binomial distribution, and the probability of a head on any particular toss is r.
2. The coin is fair – the probability of heads is the same as the probability of tails, $r = 0.5$.

It seems hard to reject the binomial distribution – events are taking place with only two possible outcomes and the tosses do seem to be independent. This leaves r, the probability that the coin lands heads. Our assumption was that the coin was fair – the probability of heads was equal to the probability of tails. Maybe this is not the case? To investigate this, we can treat r as a parameter (like $\mathbf{w}$ and σ^2 in the previous chapter) and fit it to some data.

3.1.1 Counting heads

There are three people in the queue to play. The first one plays and gets the following sequence of heads and tails:

H,T,H,H,H,H,H,H,H,H,

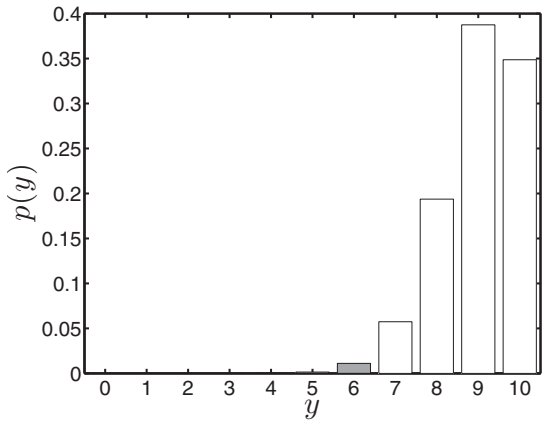


FIGURE 3.2 The binomial density function (Equation 3.1) when $N = 10$ and $r = 0.9$.

nine heads and one tail. It is possible to compute the maximum likelihood value of r as follows. The likelihood is given by the binomial distribution:

$$P(Y = y|r, N) = \left(\begin{array}{c} N \\ 9 \end{array} \right) r^y (1 - r)^{N-y}. \tag{3.2}$$

Taking the natural logarithm gives

$$L = \log P(Y = y|r, N) = \log \left(\begin{array}{c} N \\ 9 \end{array} \right) + y \log r + (N - y) \log(1 - r).$$

As in Chapter 2, we can differentiate this expression, equate to zero and solve for the maximum likelihood estimate of the parameter:

$$\begin{aligned} \frac{\partial L}{\partial r} &= \frac{y}{r} - \frac{N-y}{1-r} = 0 \\ y(1-r) &= r(N-y) \\ y &= rN \\ r &= \frac{y}{N}. \end{aligned}$$

Substituting $y = 9$ and $N = 10$ gives $r = 0.9$. The corresponding distribution function is shown in Figure 3.2 and the recalculated probability of winning is $P(Y \leq 6) = 0.0128$. This is much lower than that for $r = 0.5$. The expected return is now

$$2 \times P(Y \leq 6) + 0 \times P(Y > 6) = 0.0256.$$

Given that it costs £1 to play, we expect to make $0.0256 - 1 = -0.9744$ per game – a loss of approximately $97p$. $P(Y \leq 6) = 0.0128$ suggests that only about 1 person in every 100 should win, but this does not seem to be reflected in the number of people who *are* winning. Although the evidence from this run of coin tosses suggests $r = 0.9$, it seems too biased given that several people *have* won.

3.1.2 The Bayesian way

The value of r computed in the previous section was based on just ten tosses. Given the random nature of the coin toss, if we observed several sequences of tosses it is likely that we would get a different r each time. Thought about this way, r feels a bit like a random variable, R. Maybe we can learn something about the distribution of R rather than try and find a particular value. We saw in the previous section that obtaining an exact value by counting is heavily influenced by the particular tosses in the short sequence. No matter how many such sequences we observe there will always be some uncertainty in r – considering it as a random variable with an associated distribution will help us measure and understand this uncertainty.

In particular, defining the random variable Y_N to be the number of heads obtained in N tosses, we would like the distribution of r conditioned on the value of Y_N:

$$p(r|y_N).$$

Given this distribution, it would be possible to compute the expected probability of winning by taking the expectation of $P(Y_{\mathsf{new}} \leq 6|r)$ with respect to $p(r|y_N)$:

$$P(Y_{\mathsf{new}} \leq 6|y_N) = \int P(Y_{\mathsf{new}} \leq 6|r)p(r|y_N)dr,$$

where Y_{new} is a random variable describing the number of heads in a future set of ten tosses.

In Section 2.2.7 we gave a brief introduction to Bayes' rule. Bayes' rule allows us to reverse the conditioning of two (or more) random variables, e.g. compute $p(a|b)$ from $p(b|a)$. Here we're interested in $p(r|y_N)$, which, if we reverse the conditioning, is $p(y_N|r)$ – the probability distribution function over the number of heads in N

independent tosses where the probability of a head in a single toss is r. This is the binomial distribution function that we can easily compute for any y_N and r. In our context, Bayes' rule is (see also Equation 2.11)

$$p(r|y_N) = \frac{P(y_N|r)p(r)}{P(y_N)}. \tag{3.3}$$

This equation is going to be very important for us in the following chapters so it is worth spending some time looking at each term in detail.

The likelihood, $P(y_N|r)$ We came across likelihood in Chapter 2. Here it has exactly the same meaning: how likely is it that we would observe our data (in this case, the data is y_N) for a particular value of r (our model)? For our example, this is the binomial distribution. This value will be high if r could have feasibly produced the result y_N and low if the result is very unlikely. For example, Figure 3.3 shows the likelihood $P(y_N|r)$ as a function of r for two different scenarios. In the first, the data consists of ten tosses ($N = 10$) of which six were heads. In the second, there were $N = 100$ tosses, of which 70 were heads.

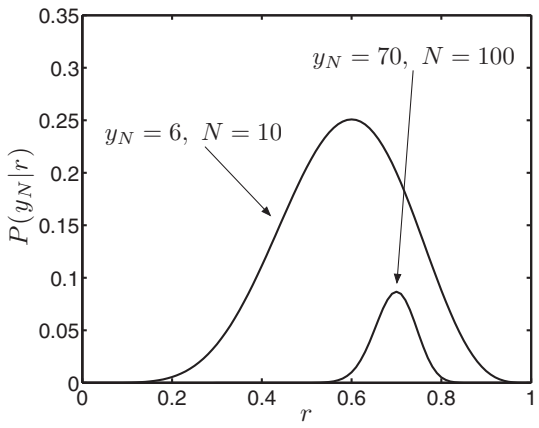


FIGURE 3.3 Examples of the likelihood $p(y_N|r)$ as a function of r for two scenarios.

This plot reveals two important properties of the likelihood. Firstly, it is not a probability density. If it were, the area under both curves would have to equal 1. We can see that this is not the case without working out the area because the two areas are completely different. Secondly, the two examples differ in how much they appear to tell us about r. In the first example, the likelihood has a non-zero value for a large range of possible r values (approximately $0.2 \leq r \leq 0.9$). In the second, this range is greatly reduced (approximately $0.6 \leq r \leq 0.8$). This is very intuitive: in the second example, we have much more data (the results of 100 tosses rather than 10) and so we *should* know more about r.

The prior distribution, $p(r)$ The prior distribution allows us to express any belief we have in the value of r *before* we see any data. To illustrate this, we shall consider the following three examples:

1. We do not know anything about tossing coins or the stall owner.
2. We think the coin (and hence the stall owner) is fair.
3. We think the coin (and hence the stall owner) is biased to give more heads than tails.

We can encode each of these beliefs as different prior distributions. r can take any value between 0 and 1 and therefore it must be modelled as a continuous random variable. Figure 3.4 shows three density functions that might be used to encode our three different prior beliefs.

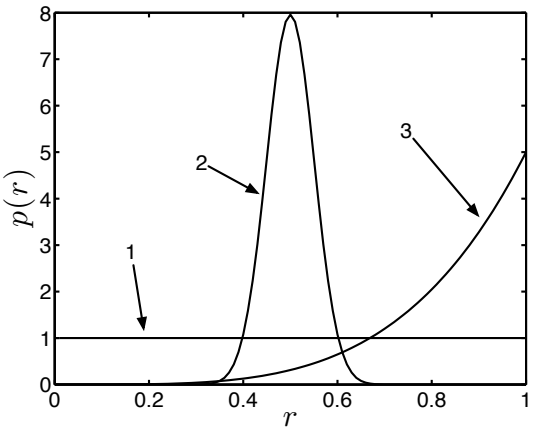


FIGURE 3.4 Examples of prior densities, $p(r)$, for r for three different scenarios.

Belief number 1 is represented as a uniform density between 0 and 1 and as such shows no preference for any particular r value. Number 2 is given a density function that is concentrated around $r = 0.5$, the value we would expect for a fair coin. The density suggests that we do not expect much variance in r: it's almost certainly going to lie between 0.4 and 0.6. Most coins that any of us have tossed agree with this. Finally, number 3 encapsulates our belief that the coin (and therefore the stall owner) is biased. This density suggests that $r > 0.5$ and that there is a high level of variance. This is fine because our belief is just that the coin is biased: we don't really have any idea how biased at this stage.

We will not choose between our three scenarios at this stage, as it is interesting to see the effect these different beliefs will have on $p(r|y_N)$.

The three functions shown in Figure 3.4 have not been plucked from thin air. They are all examples of beta probability density functions (see Section 2.5.2). The beta density function is used for continuous random variables constrained to lie between 0 and 1 – perfect for our example. For a random variable R with parameters α and β, it is defined as

$$p(r) = \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)} r^{\alpha-1}(1-r)^{\beta-1}. \tag{3.4}$$

$\Gamma(a)$ is known as the gamma function (see Section 2.5.2). In Equation 3.4 the gamma functions ensure that the density is normalised (that is, it integrates to 1 and is therefore a probability density function). In particular

$$\frac{\Gamma(\alpha)\Gamma(\beta)}{\Gamma(\alpha+\beta)} = \int_{r=0}^{r=1} r^{\alpha-1}(1-r)^{\beta-1}\, dr,$$

ensuring that

$$\int_{r=0}^{r=1} \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)} r^{\alpha-1}(1-r)^{\beta-1}\, dr = 1.$$

The two parameters α and β control the shape of the resulting density function and must both be positive. Our three beliefs as plotted in Figure 3.4 correspond to the following pairs of parameter values:

1. Know nothing: $\alpha = 1,\ \beta = 1$.
2. Fair coin: $\alpha = 50,\ \beta = 50$.
3. Biased: $\alpha = 5,\ \beta = 1$.

The problem of choosing these values is a big one. For example, why should we choose $\alpha = 5,\ \beta = 1$ for a biased coin? There is no easy answer to this. We shall see later that, for the beta distribution, they can be interpreted as a number of previous, hypothetical coin tosses. For other distributions no such analogy is possible and we will also introduce the idea that maybe these too should be treated as random variables. In the mean time, we will assume that these values are sensible and move on.

The marginal distribution of y_N – $P(y_N)$ The third quantity in our equation, $P(y_N)$, acts as a normalising constant to ensure that $p(r|y_N)$ is a properly defined density. It is known as the marginal distribution of y_N because it is computed by integrating r out of the joint density $p(y_N, r)$:

$$P(y_N) = \int_{r=0}^{r=1} p(y_N, r)\, dr.$$

This joint density can be factorised to give

$$P(y_N) = \int_{r=0}^{r=1} P(y_N|r)p(r)\, dr,$$

which is the product of the prior and likelihood integrated over the range of values that r may take.

$p(y_N)$ is also known as the **marginal likelihood**, as it is the likelihood of the data, y_N, averaged over all parameter values. We shall see in Section 3.4.1 that it can be a useful quantity in model selection, but, unfortunately, in all but a small minority of cases, it is very difficult to calculate.

The posterior distribution – $p(r|y_N)$ This **posterior** is the distribution in which we are interested. It is the result of updating our prior belief $p(r)$ in light of new evidence y_N. The shape of the density is interesting – it tells us something about how much information we have about r after combining what we knew beforehand (the prior) and what we've seen (the likelihood). Three hypothetical examples are provided in Figure 3.5 (these are purely illustrative and do not correspond to the particular likelihood and prior examples shown in Figures 3.3 and 3.4). (a) is uniform – combining the likelihood and the prior together has left all values of r equally likely. (b) suggests that r is most likely to be low but could be high. This might be the result of starting with a uniform prior and then observing more tails than heads. Finally, (c) suggests the coin is biased to land heads more often. As it is a density, the posterior tells us not just which values are likely but also provides an indication of the level of uncertainty we still have in r having observed some data.

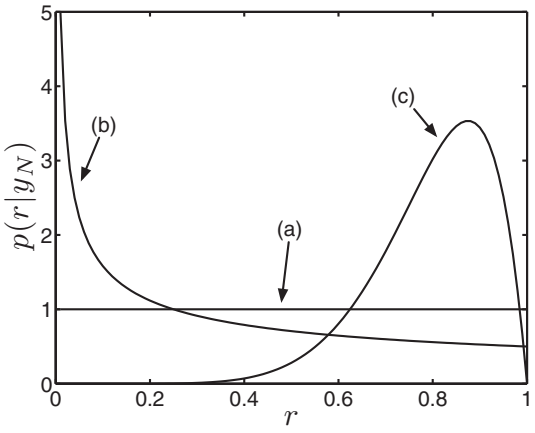


FIGURE 3.5 Examples of three possible posterior distributions $p(r|y_N)$.

As already mentioned, we can use the posterior density to compute expectations. For example, we could compute

$$\mathbf{E}_{p(r|y_N)}\{P(Y_{10} \leq 6)\} = \int_{r=0}^{r=1} P(Y_{10} \leq 6|r)p(r|y_N)\ dr,$$

the expected value of the probability that we will win. This takes into account the data we have observed, our prior beliefs and the uncertainty that remains. It will be useful in helping to decide whether or not to play the game. We will return to this later, but first we will look at the kind of posterior densities we obtain in our coin example.

Comment 3.1 – Conjugate priors: A likelihood-prior pair is said to be conjugate if they result in a posterior which is of the same form as the prior. This enables us to compute the posterior density analytically without having to worry about computing the denominator in Bayes' rule, the marginal likelihood. Some common conjugate pairs are listed in the table to the right.

Prior	Likelihood
Gaussian	Gaussian
Beta	Binomial
Gamma	Gaussian
Dirichlet	Multinomial

3.2 THE EXACT POSTERIOR

The beta distribution is a common choice of prior when the likelihood is a binomial distribution. This is because we can use some algebra to compute the posterior density exactly. In fact, the beta distribution is known as the **conjugate** prior to the binomial likelihood (see Comment 3.1). If the prior and likelihood are conjugate, the posterior will be of the same form as the prior. Specifically, $p(r|y_N)$ will give a beta distribution with parameters δ and γ, whose values will be computed from the prior and y_N. The beta and binomial are not the only conjugate pair of distributions and we will see an example of another conjugate prior and likelihood pair when we return to the Olympic data later in this chapter.

Using a conjugate prior makes things much easier from a mathematical point of view. However, as we mentioned in both our discussion on loss functions in Chapter 1 and noise distributions in Chapter 2, it is more important to base our choices on modelling assumptions than mathematical convenience. In the next chapter we will see some techniques we can use in the common scenario that the pair are non-conjugate.

Returning to our example, we can omit $p(y_N)$ from Equation 3.3, leaving

$$p(r|y_N) \propto P(y_N|r)p(r).$$

Replacing the terms on the right hand side with a binomial and beta distribution gives

$$p(r|y_N) \propto \left[\begin{pmatrix} N \\ y_N \end{pmatrix} r^{y_N}(1-r)^{N-y_N} \right] \times \left[\frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)} r^{\alpha-1}(1-r)^{\beta-1} \right]. \quad (3.5)$$

Because the prior and likelihood are conjugate, we know that $p(r|y_N)$ has to be a beta density. The beta density, with parameters δ and γ, has the following general form:

$$p(r) = Kr^{\delta-1}(1-r)^{\gamma-1},$$

where K is a constant. If we can arrange all of the terms, including r, on the right hand side of Equation 3.5 into something that looks like $r^{\delta-1}(1-r)^{\gamma-1}$, we can be sure that the constant must also be correct (it has to be $\Gamma(\delta+\gamma)/(\Gamma(\delta)\Gamma(\gamma))$ because we know that the posterior density is a beta density). In other words, we know what the normalising constant for a beta density is so we do not need to compute $p(y_N)$.

Rearranging Equation 3.5 gives us

$$\begin{aligned} p(r|y_N) &\propto \left[\binom{N}{y_N}\frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)}\right] \times \left[r^{y_N} r^{\alpha-1}(1-r)^{N-y_N}(1-r)^{\beta-1}\right] \\ &\propto r^{y_N+\alpha-1}(1-r)^{N-y_N+\beta-1} \\ &\propto r^{\delta-1}(1-r)^{\gamma-1} \\ \text{where } \delta &= y_N+\alpha \text{ and } \gamma = N-y_N+\beta. \end{aligned}$$

Therefore

$$p(r|y_N) = \frac{\Gamma(\alpha+\beta+N)}{\Gamma(\alpha+y_N)\Gamma(\beta+N-y_N)} r^{\alpha+y_N-1}(1-r)^{\beta+N-y_N-1} \tag{3.6}$$

(note that when adding γ and δ, the y_N terms cancel). This is the posterior density of r based on the prior $p(r)$ and the data y_N. Notice how the posterior parameters are computed by adding the number of heads (y_n) to the first prior parameter (α) and the number of tails ($N - y_N$) to the second (β). This allows us to gain some intuition about the prior parameters α and β – they can be thought of as the number of heads and tails in $\alpha + \beta$ previous tosses. For example, consider the second two scenarios discussed in the previous section. For the fair coin scenario, $\alpha = \beta = 50$. This is equivalent to tossing a coin 100 times and obtaining 50 heads and 50 tails. For the biased scenario, $\alpha = 5, \beta = 1$, corresponding to six tosses and five heads. Looking at Figure 3.4, this helps us explain the differing levels of variability suggested by the two densities: the fair coin density has much lower variability than the biased one because it is the result of many more hypothetical tosses. The more tosses, the more we should know about r.

The analogy is not perfect. For example, α and β don't have to be integers and can be less than 1 (0.3 heads doesn't make much sense). The analogy also breaks down when $\alpha = \beta = 1$. Observing one head and one tail means that values of $r = 0$ and $r = 1$ are impossible. However, density 1 in Figure 3.4), suggests that all values of r are equally likely. Despite these flaws, the analogy will be a useful one to bear in mind as we progress through our analysis (see Exercises 3.1, 3.2, 3.3 and 3.4)

3.3 THE THREE SCENARIOS

We will now investigate the posterior distribution $p(r|y_N)$ for the three different prior scenarios shown in Figure 3.4 – no prior knowledge, a fair coin and a biased coin.

3.3.1 No prior knowledge

In this scenario (MATLAB script: `coin_scenario1.m`), we assume that we know nothing of coin tossing or the stall holder. Our prior parameters are $\alpha = 1,\ \beta = 1$, shown in Figure 3.6(a).

To compare different scenarios we will use the expected value and variance of r under the prior. The expected value of a random variable from a beta distribution with parameters α and β (the density function of which we will henceforth denote

as $\mathcal{B}(\alpha, \beta)$) is given as (see Exercise 3.5)

$$\begin{aligned} p(r) &= \mathcal{B}(\alpha, \beta) \\ \mathbf{E}_{p(r)}\{R\} &= \frac{\alpha}{\alpha + \beta}. \end{aligned}$$

For scenario 1:

$$\mathbf{E}_{p(r)}\{R\} = \frac{\alpha}{\alpha + \beta} = \frac{1}{2}.$$

The variance of a beta distributed random variable is given by (see Exercise 3.6)

$$\mathsf{var}\{R\} = \frac{\alpha\beta}{(\alpha + \beta)^2(\alpha + \beta + 1)}, \tag{3.7}$$

which for $\alpha = \beta = 1$ is

$$\mathsf{var}\{R\} = \frac{1}{12}.$$

Note that in our formulation of the posterior (Equation 3.6) we are not restricted to updating our distribution in blocks of ten – we can incorporate the results of any number of coin tosses. To illustrate the evolution of the posterior, we will look at how it changes toss by toss.

A new customer hands over £1 and the stall owner starts tossing the coin. The first toss results in a head. The posterior distribution after one toss is a beta distribution with parameters $\delta = \alpha + y_N$ and $\gamma = \beta + N - y_N$:

$$p(r|y_N) = \mathcal{B}(\delta, \gamma).$$

In this scenario, $\alpha = \beta = 1$, and as we have had $N = 1$ tosses and seen $y_N = 1$ heads,

$$\begin{aligned} \delta &= 1 + 1 = 2 \\ \gamma &= 1 + 1 - 1 = 1. \end{aligned}$$

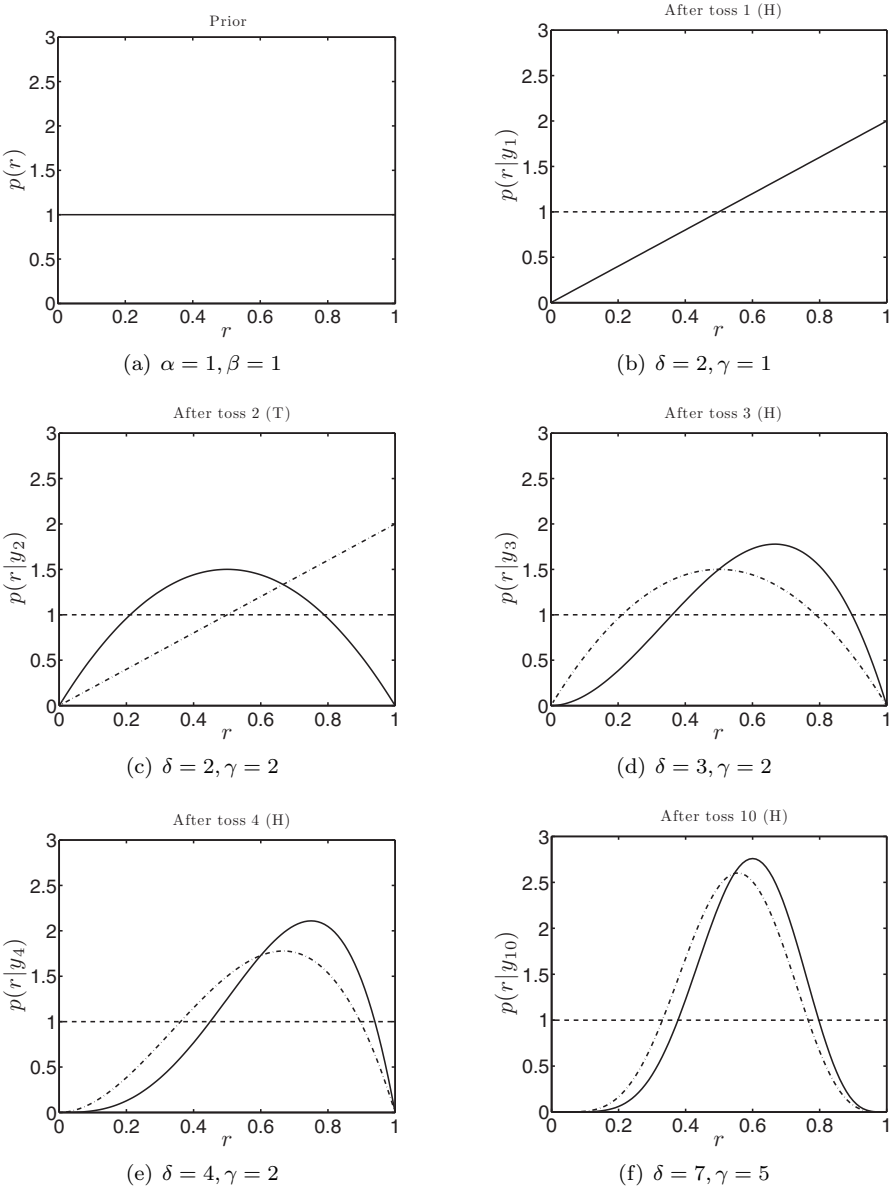
This posterior distribution is shown as the solid line in Figure 3.6(b) (the prior is also shown as a dashed line). This single observation has had quite a large effect – the posterior is very different from the prior. In the prior, all values of r were equally likely. This has now changed – higher values are more likely than lower values with zero density at $r = 0$. This is consistent with the evidence – observing one head makes high values of r slightly more likely and low values slightly less likely. The density is still very broad, as we have observed only one toss. The expected value of r under the posterior is

$$\mathbf{E}_{p(r|y_N)}\{R\} = \frac{2}{3}$$

and we can see that observing a solitary head has increased the expected value of r from 1/2 to 2/3. The variance of the posterior is (using Equation 3.7)

$$\mathsf{var}\{R\} = \frac{1}{18}$$

which is lower than the prior variance (1/12). So, the reduction in variance tells us that we have less uncertainty about the value of r than we did (we have learnt



(a) $\alpha = 1, \beta = 1$

(b) $\delta = 2, \gamma = 1$

(c) $\delta = 2, \gamma = 2$

(d) $\delta = 3, \gamma = 2$

(e) $\delta = 4, \gamma = 2$

(f) $\delta = 7, \gamma = 5$

FIGURE 3.6 Evolution of $p(r|y_N)$ as the number of observed coin tosses increases.

something) and the increase in expected value tells us that what we've learnt is that heads are slightly more likely than tails.

The stall owner tosses the second coin and it lands tails. We have now seen one head and one tail and so $N = 2$, $y_N = 1$, resulting in

$$\begin{aligned} \delta &= 1 + 1 = 2 \\ \gamma &= 1 + 2 - 1 = 2. \end{aligned}$$

The posterior distribution is shown as the solid dark line in Figure 3.6(c). The lighter dash-dot line is the posterior we saw after one toss and the dashed line is the prior. The density has changed again to reflect the new evidence. As we have now observed a tail, the density at $r = 1$ should be zero and is ($r = 1$ would suggest that the coin always lands heads). The density is now curved rather than straight (as we have already mentioned, the beta density function is very flexible) and observing a tail has made lower values more likely. The expected value and variance are now

$$\mathbf{E}_{p(r|y_N)}\{R\} = \frac{1}{2},\ \mathsf{var}\{R\} = \frac{1}{20}.$$

The expected value has decreased back to 1/2. Given that the expected value under the prior was also 1/2, you might conclude that we haven't learnt anything. However, the variance has decreased again (from 1/18 to 1/20) so we have less uncertainty in r and have learnt something. In fact, we've learnt that r is closer to 1/2 than we assumed under the prior.

The third toss results in another head. We now have $N = 3$ tosses, $y_N = 2$ heads and $N - y_N = 1$ tail. Our updated posterior parameters are

$$\begin{aligned} \delta &= \alpha + y_N = 1 + 2 = 3 \\ \gamma &= \beta + N - y_N = 1 + 3 - 2 = 2. \end{aligned}$$

This posterior is plotted in Figure 3.6(d). Once again, the posterior is the solid dark line, the previous posterior is the solid light line and the dashed line is the prior. We notice that the effect of observing this second head is to skew the density to the right, suggesting that heads are more likely than tails. Again, this is entirely consistent with the evidence – we have seen more heads than tails. We have only seen three coins though, so there is still a high level of uncertainty – the density suggests that r could potentially still be pretty much any value between 0 and 1. The new expected value and variance are

$$\mathbf{E}_{p(r|y_N)}\{R\} = \frac{3}{5},\ \mathsf{var}\{R\} = \frac{1}{25}.$$

The variance has decreased again reflecting the decrease in uncertainty that we would expect as we see more data.

Toss 4 also comes up heads ($y_N = 3, N = 4$), resulting in $\delta = 1 + 3 = 4$ and $\gamma = 1 + 4 - 3 = 2$. Figure 3.6(e) shows the current and previous posteriors and prior in the now familiar format. The density has once again been skewed to the right – we've now seen three heads and only one tail so it seems likely that r is greater than 1/2. Also notice the difference between the $N = 3$ posterior and the $N = 4$ posterior for very low values of r – the extra head has left us pretty convinced that r is not 0.1 or lower. The expected value and variance are given by

$$\mathbf{E}_{p(r|y_N)}\{R\} = \frac{2}{3},\ \mathsf{var}\{R\} = \frac{2}{63} = 0.0317,$$

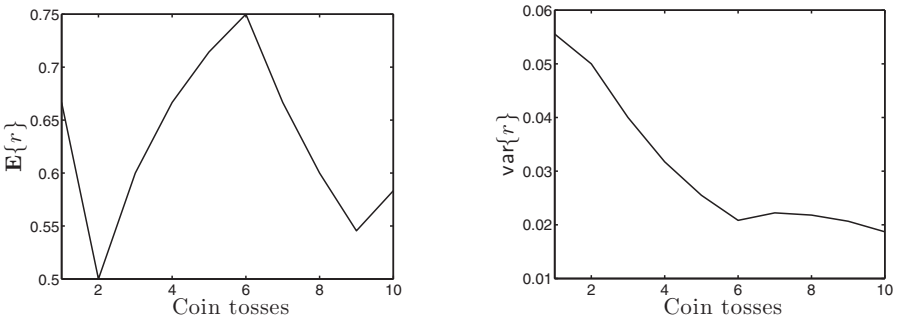
where the expected value has increased and the variance has once again decreased. The remaining six tosses are made so that the complete sequence is

H,T,H,H,H,H,T,T,T,H,

a total of six heads and four tails. The posterior distribution after $N = 10$ tosses ($y_N = 6$) has parameters $\delta = 1 + 6 = 7$ and $\gamma = 1 + 10 - 6 = 5$. This (along with the posterior for $N = 9$) is shown in Figure 3.6(f). The expected value and variance are

$$\mathbf{E}_{p(r|y_N)}\{R\} = \frac{7}{12} = 0.5833, \ \mathsf{var}\{R\} = 0.0187. \tag{3.8}$$

Our ten observations have increased the expected value from 0.5 to 0.5833 and decreased our variance from $1/12 = 0.0833$ to 0.0187. However, this is not the full story. Examining Figure 3.6(f), we see that we can also be pretty sure that $r > 0.2$ and $r < 0.9$. The uncertainty in the value of r is still quite high because we have only observed ten tosses.



(a) Expected value (b) Variance

FIGURE 3.7 Evolution of expected value (a) and variance (b) of r as coin toss data is added to the posterior.

Figure 3.7 summarises how the expected value and variance change as the 10 observations are included. The expected value jumps around a bit, whereas the variance steadily decreases as more information becomes available. At the seventh toss, the variance increases. The first seven tosses are

H,T,H,H,H,H,T.

The evidence up to and including toss 6 is that heads is much more likely than tails (5 out of 6). Tails on the seventh toss is therefore slightly unexpected. Figure 3.8 shows the posterior before and after the seventh toss. The arrival of the tail has forced the density to increase the likelihood of low values of r and, in doing so, increased the uncertainty.

The posterior density encapsulates all of the information we have about r. Shortly, we will use this to compute the expected probability of winning the game. Before we do so, we will revisit the idea of using point estimates by extracting a

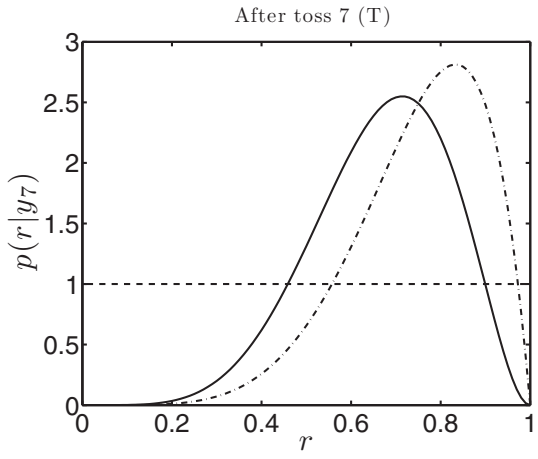


FIGURE 3.8 The posterior after six (light) and seven (dark) tosses.

single value $\widehat{r}$ of r from this density. We will then be able to compare the expected probability of winning with the probability of winning computed from a single value of r. A sensible choice would be to use $\mathbf{E}_{p(r|y_N)}\{R\}$. With this value, we can compute the probability of winning – $P(Y_{\mathsf{new}} \leq 6|\widehat{r})$. This quantity could be used to decide whether or not to play. Note that, to make the distinction between observed tosses and future tosses, we will use Y_{new} as a random variable that describes ten future tosses.

After ten tosses, the posterior density is beta with parameters $\delta = 7, \gamma = 5$. $\widehat{r}$ is therefore

$$\widehat{r} = \frac{\delta}{\delta + \gamma} = \frac{7}{12}.$$

The probability of winning the game follows as

$$\begin{aligned} P(Y_{\mathsf{new}} \leq 6|\widehat{r}) &= 1 - \sum_{y_{\mathsf{new}}=7}^{10} P(Y_{\mathsf{new}} = y_{\mathsf{new}}|\widehat{r}) \\ &= 1 - 0.3414 \\ &= 0.6586, \end{aligned}$$

suggesting that we will win more often than lose.

Using all of the posterior information requires computing

$$\mathbf{E}_{p(r|y_N)}\left\{P(Y_{\mathsf{new}} \leq 6|r)\right\}.$$

Rearranging and manipulating the expectation provides us with the following ex-

pression:

$$\begin{aligned}
\mathbf{E}_{p(r|y_N)}\left\{P(Y_{\mathsf{new}} \leq 6|r)\right\} &= \mathbf{E}_{p(r|y_N)}\left\{1 - P(Y_{\mathsf{new}} \geq 7|r)\right\} \\
&= 1 - \mathbf{E}_{p(r|y_N)}\left\{P(Y_{\mathsf{new}} \geq 7|r)\right\} \\
&= 1 - \mathbf{E}_{p(r|y_N)}\left\{\sum_{y_{\mathsf{new}}=7}^{y_{\mathsf{new}}=10} P(Y_{\mathsf{new}} = y_{\mathsf{new}}|r)\right\} \\
&= 1 - \sum_{y_{\mathsf{new}}=7}^{y_{\mathsf{new}}=10} \mathbf{E}_{p(r|y_N)}\left\{P(Y_{\mathsf{new}} = y_{\mathsf{new}}|r)\right\}.
\end{aligned} \tag{3.9}$$

To evaluate this, we need to be able to compute $\mathbf{E}_{p(r|y_N)}\left\{P(Y_{\mathsf{new}} = y_{\mathsf{new}}|r)\right\}$. From the definition of expectations, this is given by

$$\begin{aligned}
&\mathbf{E}_{p(r|y_N)}\left\{P(Y_{\mathsf{new}} = y_{\mathsf{new}}|r)\right\} = \int_{r=0}^{r=1} P(Y_{\mathsf{new}} = y_{\mathsf{new}}|r)p(r|y_N)\, dr \\
= &\int_{r=0}^{r=1} \left[\begin{pmatrix} N_{\mathsf{new}} \\ y_{\mathsf{new}} \end{pmatrix} r^{y_{\mathsf{new}}}(1-r)^{N_{\mathsf{new}}-y_{\mathsf{new}}}\right]\left[\frac{\Gamma(\delta+\gamma)}{\Gamma(\delta)\Gamma(\gamma)} r^{\delta-1}(1-r)^{\gamma-1}\right] dr \\
= &\begin{pmatrix} N_{\mathsf{new}} \\ y_{\mathsf{new}} \end{pmatrix} \frac{\Gamma(\delta+\gamma)}{\Gamma(\delta)\Gamma(\gamma)} \int_{r=0}^{r=1} r^{y_{\mathsf{new}}+\delta-1}(1-r)^{N_{\mathsf{new}}-y_{\mathsf{new}}+\gamma-1}\, dr.
\end{aligned} \tag{3.10}$$

This integral looks a bit daunting. However, on closer inspection, the argument inside the integral is an unnormalised beta density with parameters $\delta + y_{\mathsf{new}}$ and $\gamma + N_{\mathsf{new}} - y_{\mathsf{new}}$. In general, for a beta density with parameters α and β, the following *must* be true:

$$\int_{r=0}^{r=1} \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)} r^{\alpha-1}(1-r)^{\beta-1} dr = 1,$$

and therefore

$$\int_{r=0}^{r=1} r^{\alpha-1}(1-r)^{\beta-1} dr = \frac{\Gamma(\alpha)\Gamma(\beta)}{\Gamma(\alpha+\beta)}.$$

Our desired expectation becomes

$$\mathbf{E}_{p(r|y_N)}\left\{P(Y_{\mathsf{new}} = y_{\mathsf{new}}|r)\right\} = \begin{pmatrix} N_{\mathsf{new}} \\ y_{\mathsf{new}} \end{pmatrix} \frac{\Gamma(\delta+\gamma)}{\Gamma(\delta)\Gamma(\gamma)} \frac{\Gamma(\delta+y_{\mathsf{new}})\Gamma(\gamma+N_{\mathsf{new}}-y_{\mathsf{new}})}{\Gamma(\delta+\gamma+N_{\mathsf{new}})}$$

which we can easily compute for a particular posterior (i.e. values of γ and δ) and values of N_{new} and y_{new}.

After ten tosses, we have $\delta = 7$, $\gamma = 5$. Plugging these values in, we can compute the expected probability of success:

$$\begin{aligned}
\mathbf{E}_{p(r|y_N)}\left\{P(Y_{\mathsf{new}} \leq 6|r)\right\} &= 1 - \sum_{y_{\mathsf{new}}=7}^{y_{\mathsf{new}}=10} \mathbf{E}_{p(r|y_N)}\left\{P(Y_{\mathsf{new}} = y_{\mathsf{new}}|r)\right\} \\
&= 1 - 0.3945 \\
&= 0.6055.
\end{aligned}$$

Comparing this with the value obtained using the point estimate, we can see that both predict we will win more often than not. This is in agreement with the evidence – the one person we have fully observed got six heads and four tails and hence won £2. The point estimate gives a higher probability – ignoring the posterior uncertainty makes it more likely that we will win.

Another customer plays the game. The sequence of tosses is

$$\text{H,H,T,T,H,H,H,H,H,H,}$$

eight heads and two tails – the stall owner has won. Combining all 20 tosses that we have observed, we have $N = 20$, $y_N = 6 + 8 = 14$ heads and $N - y_N = 20 - 14 = 6$ tails. This gives $\delta = 15$ and $\gamma = 7$. The posterior density is shown in Figure 3.9 where the light line shows the posterior we had after ten and the dashed line the prior. The expected value and variance are

$$\mathbf{E}_{p(r|y_N)}\{R\} = 0.6818, \mathsf{var}\{R\} = 0.0094.$$

The expected value has increased and the variance has decreased (c.f. Equation 3.8). Both behaviours are what we would expect – eight heads and two tails should increase the expected value of r and the increased data should decrease the variance. We can now recompute $\mathbf{E}_{p(r|y_N)}\{P(Y_{\mathsf{new}} \leq 6|r)\}$ in light of the new evidence. Plug-

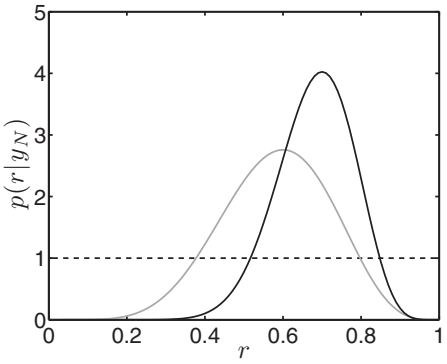


FIGURE 3.9 Posterior distribution after observing 10 tosses (light curve) and 20 tosses (dark curve). The dashed line corresponds to the prior density.

ging in the appropriate values, this is

$$\mathbf{E}_{p(r|y_N)}\{P(Y_{\mathsf{new}} \leq 6|r)\} = 0.4045.$$

The new evidence has pushed the density to the right, made high values of r (and hence the coin landing heads) more likely and reduced the probability of winning. For completeness, we can also compute $P(Y_{\mathsf{new}} \leq 6|\widehat{r}) = 0.3994$.

This corresponds to an expected return of:

$$2 \times 0.4045 - 1 = -0.1910,$$

equivalent to a loss of about 20p per go.

In this example we have now touched upon all of the important components of Bayesian Machine Learning – choosing priors, choosing likelihoods, computing posteriors and using expectations to make predictions. We will now repeat this process for the other two prior scenarios.

3.3.2 The fair coin scenario

For the fair coin scenario (MATLAB script: `coin_scenario2.m`), we assumed that $\alpha = \beta = 50$, which is analogous to assuming that we have already witnessed 100 tosses, half of which resulted in heads. The first thing to notice here is that 100 tosses corresponds to much more data than we are going to observe here (20 tosses). Should we expect our data to have the same effect as it did in the previous scenario?

Figure 3.10(a) shows the prior density and Figures 3.10(b), 3.10(c), 3.10(d), 3.10(e) and 3.10(f) show the posterior after 1, 5, 10, 15 and 20 tosses, respectively. For this scenario, we have not shown the previous posterior at each stage – it is too close to the current one. However, in most cases, the change in posterior is so small that the lines almost lie right on top of one another. In fact, it is only after about ten tosses that the posterior has moved significantly from the prior. Recalling our analogy for the beta prior, this prior includes the evidential equivalent of 100 tosses and so it is not surprising that adding another ten makes much difference.

The evolution of $\mathbf{E}_{p(r|y_N)}\{R\}$ and $\mathsf{var}\{R\}$ as the 20 tosses are observed can be seen in Figure 3.11. We see very little change in either as the data appear compared to the changes we observed in Figure 3.6. Such small changes are indicative of a very *strong* prior density. The prior will dominate over the data until we've observed many more tosses – i.e., $p(r)$ dominates $p(y_N|r)$ in Equation 3.3. We have created a model that is stuck in its ways and will require a lot of persuasion to believe otherwise.

Just as in the previous section, we can work out $\mathbf{E}_{p(r|y_N)}\{P(Y_{\mathsf{new}} \leq 6|r)\}$. After all 20 tosses have been observed, we have $\delta = \alpha + y_N = 50 + 14 = 64$ and $\gamma = \beta + N - y_N = 50 + 20 - 14 = 56$. The expectation works out as

$$\mathbf{E}_{p(r|y_N)}\{P(Y_{\mathsf{new}} \leq 6|r)\} = 0.7579. \tag{3.11}$$

As before, we can also see how much difference there is between this value and the value obtained using the point estimate $\widehat{r}$, $P(Y_{\mathsf{new}} \leq 6|\widehat{r})$ (in this case, $\widehat{r} = 64/(64+56) = 0.5333$):

$$P(Y_{\mathsf{new}} \leq 6|\widehat{r}) = 0.7680.$$

Both quantities predict that we will win more often than not. In light of what we've seen about the posterior, this should come as no surprise. The data has done little to overcome the prior assumption that the coin is fair, and we already know that, if the coin is fair, we will tend to win (a fair coin will result in us winning, on average, 66p per game – see the start of Section 3.1).

As an aside, consider how accurate our approximation $P(Y_{\mathsf{new}} \leq 6|\widehat{r})$ is to the proper expectation in this scenario and the previous one. In the previous one, the difference between the two values was

$$|\mathbf{E}_{p(r|y_N)}\{P(Y_{\mathsf{new}} \leq 6|r)\} - P(Y_{\mathsf{new}} \leq 6|\widehat{r})| = 0.0531.$$

In this example, the values are closer:

$$|\mathbf{E}_{p(r|y_N)}\{P(Y_{\mathsf{new}} \leq 6|r)\} - P(Y_{\mathsf{new}} \leq 6|\widehat{r})| = 0.0101.$$

There is a good reason why this is the case – as the variance in the posterior decreases (the variance in scenario 2 is much lower than in scenario 1), the probability density becomes more and more condensed around one particular point. Imagine the variance

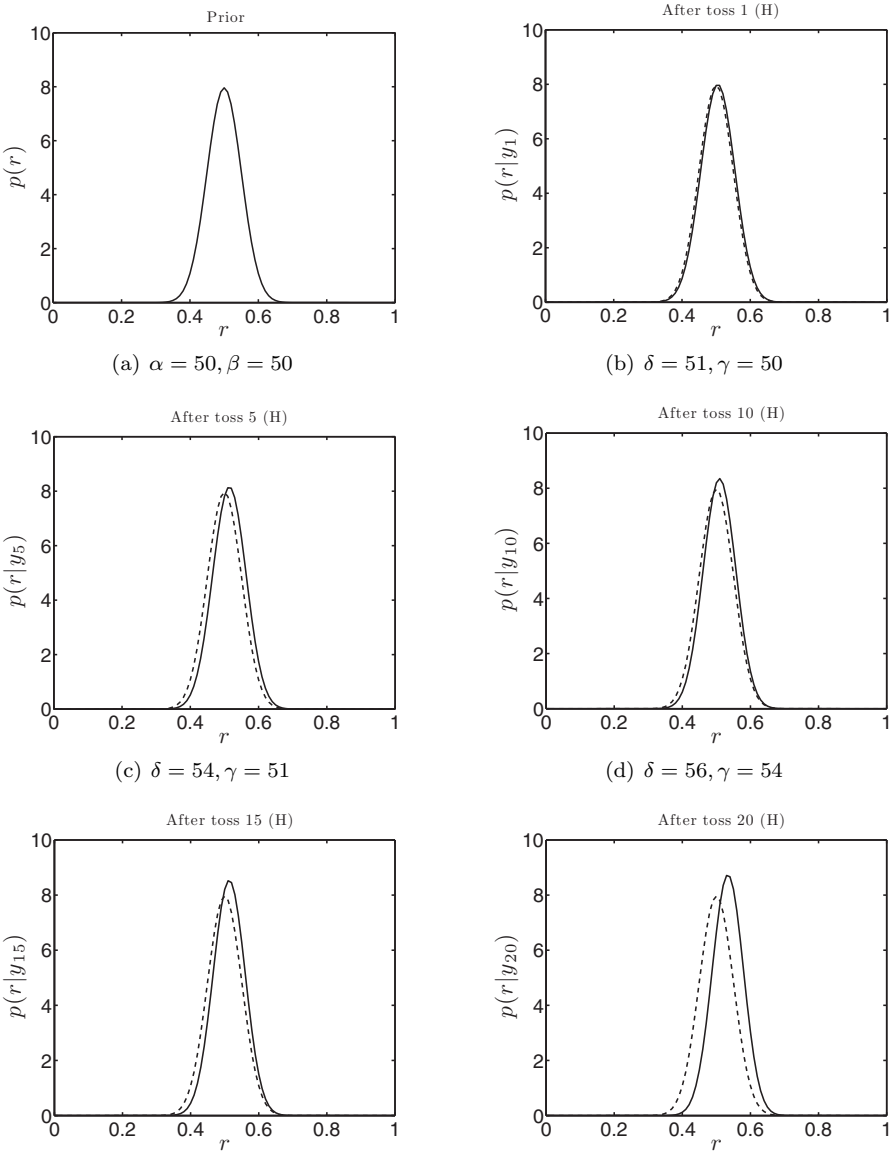
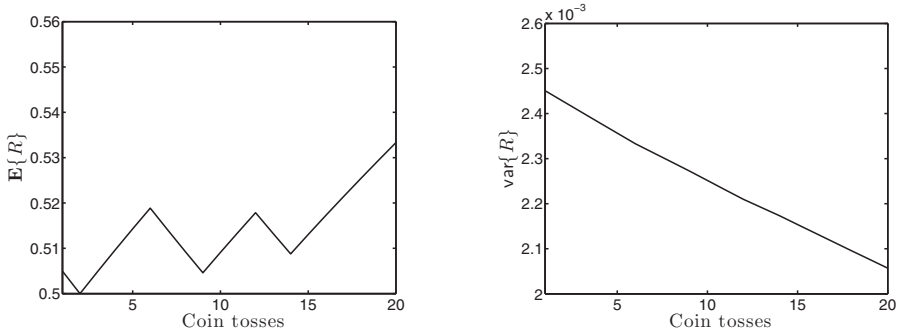


FIGURE 3.10 Evolution of the posterior $p(r|y_N)$ as more coin tosses are observed for the fair coin scenario. The dashed line shows the prior density.



(a) Expected value (b) Variance

FIGURE 3.11 Evolution of $\mathbf{E}_{p(r|y_N)}\{R\}$ (a) and $\mathsf{var}\{R\}$ (b) as the 20 coin tosses are observed for the fair coin scenario.

decreasing to such an extent that there was a single value of r that had probability 1 of occurring with $p(r|y_N)$ being zero everywhere else. The expectation we are calculating is

$$\mathbf{E}_{p(r|y_N)}\{P(Y_{\mathsf{new}} \leq 6|r)\} = \int_{r=0}^{r=1} P(Y_{\mathsf{new}} \leq 6|r)p(r|y_N)\ dr.$$

If $p(r|y_N)$ is zero everywhere except at one specific value (say $\widehat{r}$), this becomes

$$\mathbf{E}_{p(r|y_N)}\{P(Y_{\mathsf{new}} \leq 6|r)\} = P(Y_{\mathsf{new}} \leq 6|\widehat{r}).$$

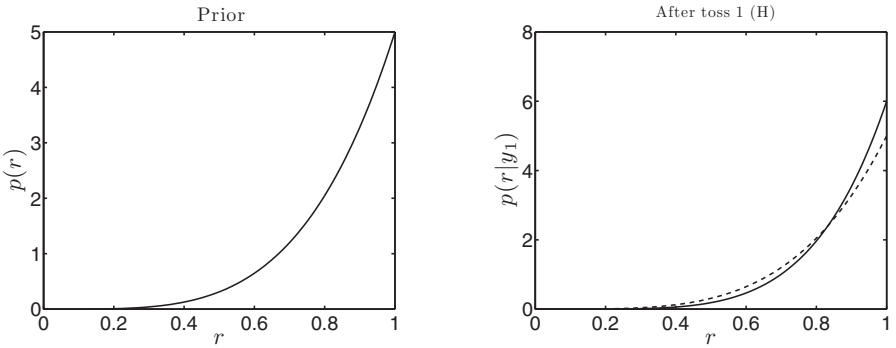
In other words, as the variance decreases, $P(Y_{\mathsf{new}} \leq 6|\widehat{r})$ becomes a better and better approximation to the true expectation. This is not specific to this example – as the quantity of data increases (and uncertainty about parameters subsequently decreases), point approximations become more reliable.

3.3.3 A biased coin

In the final scenario we assume that the coin (and therefore the stall owner) is biased to generate more heads than tails (MATLAB script: `coin_scenario3.m`). This is encoded through a beta prior with parameters $\alpha = 5$, $\beta = 1$. The expected value is

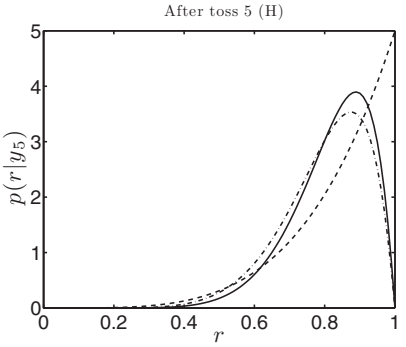
$$\mathbf{E}_{p(r)}\{r\} = 5/6,$$

five coins out of every six will come up heads. Just as for scenario 2, Figure 3.12(a) shows the prior density and Figures 3.12(b), 3.12(c), 3.12(d), 3.12(e) and 3.12(f) show the posterior after 1, 5, 10, 15 and 20 tosses, respectively. Given what we've already seen, there is nothing unusual here. The posterior moves quite rapidly away from the prior (the prior effectively has only the influence of $\alpha + \beta = 6$ data points). Figure 3.13 shows the evolution of expected value and variance. The variance curve has several bumps corresponding to tosses resulting in tails. This is because of the strong prior bias towards a high r value. We don't expect to see many tails under

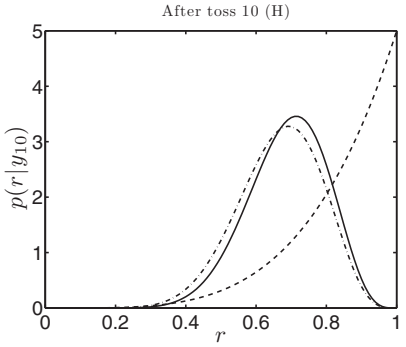


(a) $\alpha = 5, \beta = 1$

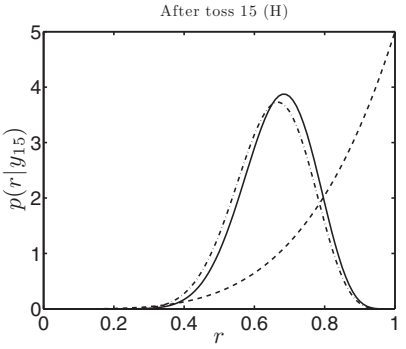
(b) $\delta = 6, \gamma = 1$



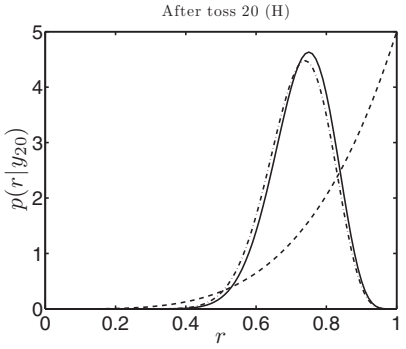
(c) $\delta = 9, \gamma = 2$



(d) $\delta = 11, \gamma = 5$

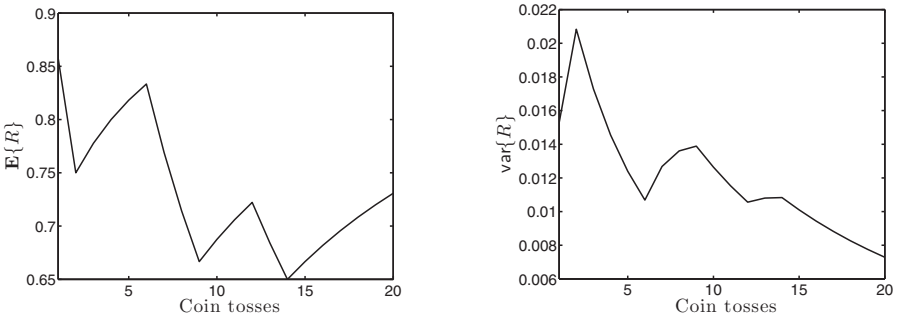


(e) $\delta = 14, \gamma = 7$



(f) $\delta = 19, \gamma = 7$

FIGURE 3.12 Evolution of the posterior $p(r|y_N)$ as more coin tosses are observed for the biased coin scenario. The dashed line shows the prior density and in the last four plots, the dash-dot line shows the previous posterior (i.e. the posterior after 4, 9, 14 and 19 tosses).



(a) Expected value (b) Variance

FIGURE 3.13 Evolution of $\mathbf{E}_{p(r|y_N)}\{R\}$ (a) and $\mathsf{var}\{R\}$ (b) as the 20 coin tosses are observed for the biased coin scenario.

this assumption and so when we do, the model becomes less certain. Once again, we calculate the true quantity of interest, $\mathbf{E}_{p(r|y_N)}\{P(Y_{\mathsf{new}} \leq 6|r)\}$. The final posterior parameter values are $\delta = \alpha + y_N = 5 + 14 = 19$, $\gamma = 1 + N - y_N = 1 + 20 - 14 = 7$. Plugging these in,

$$\mathbf{E}_{p(r|y_N)}\{P(Y_{\mathsf{new}} \leq 6|r)\} = 0.2915.$$

The approximation, noting that $\widehat{r} = 19/(19+7) = 0.7308$ is

$$P(Y_{\mathsf{new}} \leq 6|\widehat{r}) = 0.2707.$$

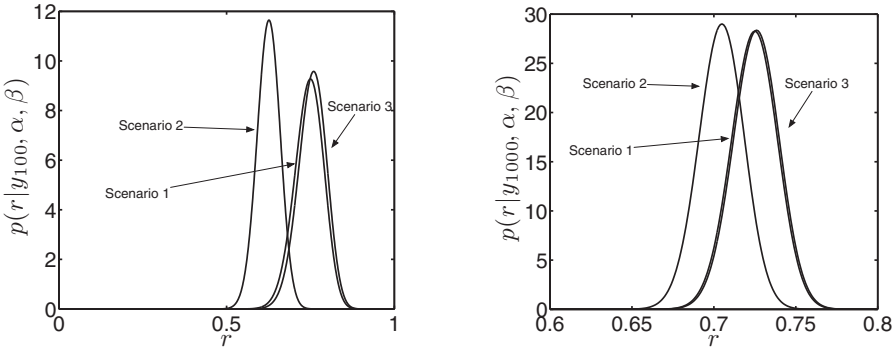
Both values suggest we will lose money on average.

3.3.4 The three scenarios – a summary

Our three different scenarios have given us different values for the expected probability of winning:

1. No prior knowledge: $\mathbf{E}_{p(r|y_N)}\{P(Y_{\mathsf{new}} \leq 6|r)\} = 0.4045$.
2. Fair coin: $\mathbf{E}_{p(r|y_N)}\{P(Y_{\mathsf{new}} \leq 6|r)\} = 0.7579$.
3. Biased coin: $\mathbf{E}_{p(r|y_N)}\{P(Y_{\mathsf{new}} \leq 6|r)\} = 0.2915$.

Which one should we choose? We could choose based on which of the prior beliefs seems most plausible. Given that the stall holder doesn't look like he is about to go out of business, scenario 3 might be sensible. We might decide that we really do not know anything about the stall holder and coin and look to scenario 1. We might believe that an upstanding stall holder would never stoop to cheating and go for scenario 2. It is possible to justify any of them. What we have seen is that the Bayesian technique allows you to combine the data observed (20 coin tosses) with some prior knowledge (one of the scenarios) in a principled way. The posterior density explicitly models the uncertainty that remains in r at each stage and can be used to make predictions (see Exercises 3.7 and 3.8).



(a) The three posteriors after 100 tosses

(b) The three posteriors after 1000 tosses

FIGURE 3.14 The posterior densities for the three scenarios after 100 coin tosses (left) and 1000 coin tosses (right).

3.3.5 Adding more data

Before we move on, it is worth examining the effect of adding more and more data. We have seen in each of our scenarios that the addition of more data results in the posterior diverging from the prior – usually through a decrease in variance. In fact, if we continue adding more data, we will find that the posteriors for all three scenarios start to look very similar. In Figure 3.14 we see the posteriors for the three scenarios after 100 and 1000 tosses. Compared with the posteriors for the three scenarios after small numbers of tosses have been observed (Figures 3.6(f), 3.10(d) and 3.12(d)), we notice that the posteriors are becoming more and more similar. This is particularly noticeable for scenarios 1 and 3 – by 1000 tosses they are indistinguishable. The difference between these two and the posteriors for scenario 2 is due to the high strength (low variance) of the prior for scenario 2 – the prior corresponds to a very strong belief and it will take a lot of contradictory data to remove that influence.

The diminishing effect of the prior as the quantity of data increases is easily explained if we look at the expression used to compute the posterior. Ignoring the normalising marginal likelihood term, the posterior is proportional to the likelihood multiplied by the prior. As we add more data, the prior is unchanged but the likelihood becomes a product (if the normal independence assumptions are made) of individual likelihood for more and more observations. This increase will gradually swamp the single contribution from the prior. It is also very intuitive – as we observe more and more data, beliefs we had before seeing any become less and less important.

3.4 MARGINAL LIKELIHOODS

Fortunately, subjective beliefs are not the only option for determining which of our three scenarios is best. Earlier in this chapter, when discussing the terms in Equation 3.3, we showed how the denominator $p(y_N)$ could be considered to be

related to r as follows:

$$\begin{aligned} p(y_N) &= \int_{r=0}^{r=1} p(r, y_N)\ dr \\ &= \int_{r=0}^{r=1} p(y_N|r)p(r)\ dr. \end{aligned} \tag{3.12}$$

Now when considering different choices of $p(r)$, we need to be more strict about our conditioning. $p(r)$ should actually be written as $p(r|\alpha, \beta)$ as the density is conditioned on a particular pair of α and β values. Extending this conditioning through Equation 3.12 gives

$$p(y_N|\alpha, \beta) = \int_{r=0}^{r=1} p(y_N|r)p(r|\alpha, \beta)\ dr. \tag{3.13}$$

The marginal likelihood (so called because r has been marginalised), $p(y_N|\alpha, \beta)$, is a very useful and important quantity. It tells us how likely the data (y_N) is given our choice of prior parameters α and β. The higher $p(y_N|\alpha, \beta)$, the better our evidence agrees with the prior specification. Hence, for our dataset, we could use $p(y_N|\alpha, \beta)$ to help choose the best scenario: select the scenario for which $p(y_N|\alpha, \beta)$ is highest.

To compute this quantity, we need to evaluate the following integral:

$$\begin{aligned} p(y_N|\alpha, \beta) &= \int_{r=0}^{r=1} p(y_N|r)p(r|\alpha, \beta)\ dr \\ &= \int_{r=0}^{r=1} \left(\begin{array}{c} N \\ y_N \end{array} \right) r^{y_N}(1-r)^{N-y_N} \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)} r^{\alpha-1}(1-r)^{\beta-1}\ dr \\ &= \left(\begin{array}{c} N \\ y_N \end{array} \right) \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)} \int_{r=0}^{r=1} r^{\alpha+y_N-1}(1-r)^{\beta+N-y_N-1}\ dr. \end{aligned}$$

This is of exactly the same form as Equation 3.10. The argument inside the integral is an unnormalised beta density and so we know that by integrating it we will get the inverse of the normal beta normalising constant. Therefore,

$$p(y_N|\alpha, \beta) = \left(\begin{array}{c} N \\ y_N \end{array} \right) \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)} \frac{\Gamma(\alpha+y_N)\Gamma(\beta+N-y_N)}{\Gamma(\alpha+\beta+N)}. \tag{3.14}$$

In our example, $N = 20$ and $y_N = 14$ (there were a total of 14 heads in the 2 sets of 10 tosses). We have three different possible pairs of α and β values. Plugging these values into Equation 3.14 gives

1. No prior knowledge, $\alpha = \beta = 1$, $p(y_N|\alpha, \beta) = 0.0476$.
2. Fair coin, $\alpha = \beta = 50$, $p(y_N|\alpha, \beta) = 0.0441$.
3. Biased coin, $\alpha = 5, \beta = 1$, $p(y_N|\alpha, \beta) = 0.0576$.

The prior corresponding to the biased coin has the highest marginal likelihood and the fair coin prior has the lowest. In the previous section we saw that the probability of winning under that scenario was $\mathbf{E}_{p(r|y_N,\alpha,\beta)}\{P(Y_{\mathsf{new}} \leq 6|r)\} = 0.2915$ (note that we're now conditioning the posterior on the prior parameters – $p(r|y_N, \alpha, \beta)$).

A word of caution is required here. Choosing priors in this way is essentially choosing the prior that best agrees with the data. The prior no longer corresponds

to our beliefs *before* we observe any data. In some applications this may be unacceptable. What it does give us is a single value that tells us how much the data backs up the prior beliefs. In the above example, the data suggests that the biased coin prior is best supported by the evidence.

3.4.1 Model comparison with the marginal likelihood

It is possible to extend the prior comparison in the previous section to using the marginal likelihood to optimise α and β. Assuming that α and β can take any value in the ranges

$$\begin{aligned} 0 \leq \quad \alpha \quad &\leq 50 \\ 0 \leq \quad \beta \quad &\leq 30, \end{aligned}$$

we can search for the values of α and β that maximise $p(y_N|\alpha, \beta)$.

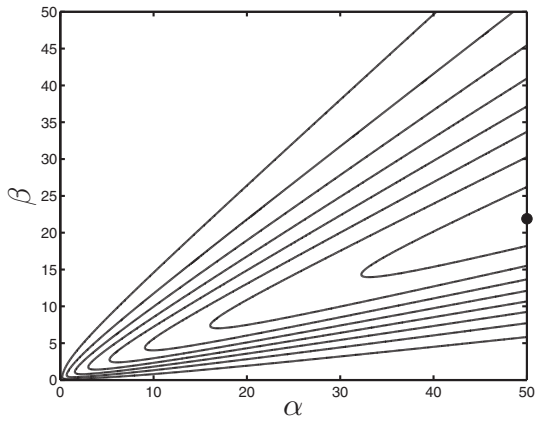


FIGURE 3.15 Marginal likelihood contours (as a function of the prior parameters, α and β) for the coin example. The circle towards the top right shows the optimum.

Figure 3.15 shows the marginal likelihood as α and β are varied in their respective ranges. The optimum value is $\alpha = 50, \beta = 22$, resulting in a marginal likelihood of 0.1694. Choosing parameters in this way is known as Type II Maximum Likelihood (to distinguish it from standard (i.e. Type I) Maximum Likelihood, introduced in Chapter 2).

3.5 HYPERPARAMETERS

The Bayesian analysis presented thus far has all been based on the idea that we can represent any quantities of interest as random variables (e.g. r, the probability of a coin landing heads). r is not the only parameter of interest in our example. α and β are also parameters – could we do the same thing with them? In some cases we can be directed towards particular values based on our knowledge of the problem (we

might know that the coin is biased). Often we will not know the exact value that they should take and should therefore treat them as random variables. To do so, we need to define a prior density over all random variables – $p(r, \alpha, \beta)$. This factorises as (see Section 2.2.5)

$$p(r, \alpha, \beta) = p(r|\alpha, \beta)p(\alpha, \beta).$$

In addition, it will often be useful to assume that α and β are independent: $p(\alpha, \beta) = p(\alpha)p(\beta)$. The quantity in which we are interested is the posterior over all parameters in the model:

$$p(r, \alpha, \beta|y_N).$$

Applying Bayes' rule, we have

$$\begin{aligned} p(r, \alpha, \beta|y_N) &= \frac{p(y_N|r, \alpha, \beta)p(r, \alpha, \beta)}{p(y_N)} \\ &= \frac{p(y_N|r)p(r, \alpha, \beta)}{p(y_N)} \\ &= \frac{p(y_N|r)p(r|\alpha, \beta)p(\alpha, \beta)}{p(y_N)}. \end{aligned}$$

Note that, in the second step, we removed α and β from the likelihood $p(y_N|r)$. This is another example of conditional independence (see Section 2.8.1). The distribution over y_N depends on α and β but only through their influence on r. Conditioned on a particular value of r, this dependence is broken.

$p(\alpha, \beta)$ will normally require some additional parameters – i.e. $p(\alpha, \beta|\kappa)$ where κ controls the density in the same way that α and β control the density for r. κ is known as a **hyper-parameter** because it is a parameter controlling the prior on the parameters controlling the prior on r. When computing the marginal likelihood, we integrate over all random variables and are just left with the data conditioned on the hyperparameters:

$$p(y_N|\kappa) = \iiint p(y_N|r)p(r|\alpha, \beta)p(\alpha, \beta|\kappa)\ dr\ d\alpha\ d\beta.$$

Unfortunately, adding this extra complexity to the model often means that computation of the quantities of interest – the posterior $p(r, \alpha, \beta|y_N, \kappa)$ (and any predictive expectations) and the marginal likelihood $p(y_N|\kappa)$ – is analytically intractable and requires one of the approximation methods that we will introduce in Chapter 4.

At this point, one could imagine indefinitely adding layers to the model. For example, κ could be thought of as a random variable that comes from a density parameterised by other random variables. The number of levels in the hierarchy (how far we go before we fix one or more parameters) will be dictated by the data we are trying to model (perhaps we can specify exact values at some level) or how much computation we can tolerate. In general, the more layers we add the more complex it will be to compute posteriors and predictions.

3.6 GRAPHICAL MODELS

When adding extra layers to our model (hyperparameters, etc.), they can quickly become unwieldy. It is popular to describe them graphically. A **graphical model** is a network where nodes correspond to random variables and edges to dependencies

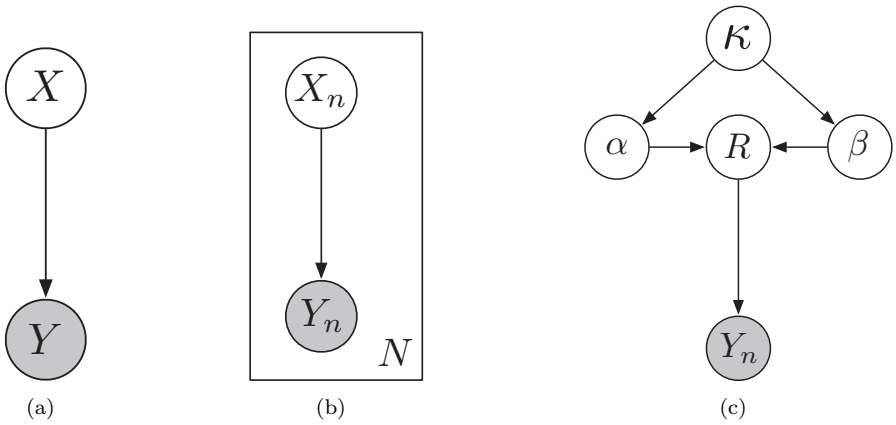


FIGURE 3.16 Graphical model examples. Nodes correspond to random variables, with the shaded nodes corresponding to things that we observe. Arrows describe the dependencies between variables and the plates describe multiple instances. For example, in (b), there are N random variables Y_n $(n = 1, \ldots, N)$ and each is dependent on a random variable X_n. (c) is a graphical representation of the model used in the coin example with the addition of a prior on α and β parameterised by κ.

between random variables. For example, in Section 2.2.4 we introduced various properties of random variables through a model that consisted of two random variables – one representing the toss of a coin (X) and one representing how I say the coin landed (Y). The model is defined through the conditional distribution $P(Y = y|X = x)$ and is represented graphically in Figure 3.16(a). The two nodes are joined by an aarrow to show that Y is defined as being conditioned on X. Note also that the node for Y is shaded. This is because, as far as the listener is concerned, this variable is *observed*. The listener does not see the coin actually landing and so doesn't observe X. Imagine that the procedure was repeated N times; we now have $2N$ random variables, $X_1, \ldots, X_N$ and $Y_1, \ldots, Y_N$. Drawing all of these would be messy. Instead we can embed the nodes within a **plate**. Plates are rectangles that tell us that whatever is embedded within them is repeated a number of times. The number of times is given in the bottom right corner, as shown in Figure 3.16(b).

Figure 3.16(c) shows a graphical representation of our coin toss model. It has a single (observed) random variable that represents the number of heads in N tosses, y_N. This is conditioned on a random variable R, which depends on random variables α and β. Finally, α and β are dependent on the hyper-parameter κ.

More information on graphical models can be found in the suggested reading at the end of the chapter.

3.7 SUMMARY

In the previous sections we have introduced many new concepts. Perhaps the most important is the idea of treating all quantities of interest as random variables. To do this we must define a prior distribution over the possible values of these quantities and then use Bayes' rule (Equation 3.3) to see how the density changes as we incorporate evidence from observed data. The resulting posterior density can be examined and used to compute interesting expectations. In addition, we have shown how the marginal likelihood (the normalisation constant in Bayes' rule) can be used to compare different models – for example, choosing the most likely prior in our coin tossing example – and discussed the possible pitfalls and objections to such an approach. Finally, we have shown how the Bayesian method can be extended by treating parameters that define the priors over other parameters as random variables. Additions to the hierarchy such as this often make analytical computations intractable and we have to resort to sampling and approximation based techniques, which are the subject of the next chapter.

3.8 A BAYESIAN TREATMENT OF THE OLYMPIC 100 m DATA

We now return to the Olympic 100 m data. In the previous chapters we fitted a linear (in the parameters) model by minimising the squared loss and then incorporated an explicit noise model and found optimal parameter values by maximising the likelihood. In this section, we will give the data a Bayesian treatment with the aim of making a prediction for the 2012 Olympics in London. This will involve several steps. Firstly, we will need to define the prior and likelihood (as we did in the coin example) and use these to compute the posterior density over the parameters of our model, just as we computed the posterior over r in the coin example. Once we've computed the posterior, we can use it to make predictions for new Olympic years.

3.8.1 The model

We will use the kth order polynomial model that was introduced in Chapter 1 with the Gaussian noise model introduced in Chapter 2:

$$t_n = w_0 + w_1 x_n + w_2 x_n^2 + \cdots + w_K x_n^K + \epsilon_n,$$

where $\epsilon_n \sim \mathcal{N}(0, \sigma^2)$. In vector form, this corresponds to

$$t_n = \mathbf{w}^\mathsf{T} \mathbf{x}_n + \epsilon_n$$

where $\mathbf{w} = [w_0, \ldots, w_K]^\mathsf{T}$ and $\mathbf{x}_n = [1, x_n, x_n^2, \ldots, x_n^K]^\mathsf{T}$. Stacking all of the responses into one vector $\mathbf{t} = [t_1, \ldots, t_N]^\mathsf{T}$ and all of the inputs into a single matrix, $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N]^\mathsf{T}$ (just as in Equation 1.18), we get the following expression for the whole dataset:

$$\mathbf{t} = \mathbf{X}\mathbf{w} + \boldsymbol{\epsilon},$$

where $\boldsymbol{\epsilon} = [\epsilon_1, \ldots, \epsilon_N]^\mathsf{T}$.

In this example, we are going to slightly simplify matters by assuming that we know the true value of σ^2. We could use all of the methods introduced in this chapter to treat σ^2 as a random variable and we could get analytical results for the posterior

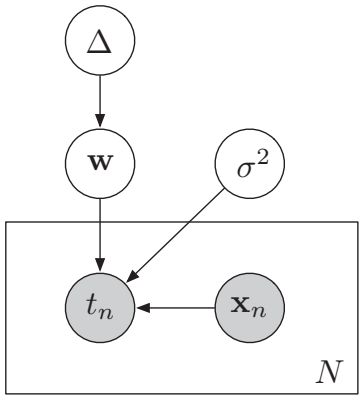


FIGURE 3.17 Graphical model for the Bayesian model of the Olympic men's 100 m data.

distribution but the maths is messier, which could detract from the main message. Substituting these various symbols into Bayes' rule gives

$$\begin{aligned} p(\mathbf{w}|\mathbf{t}, \mathbf{X}, \sigma^2, \Delta) &= \frac{p(\mathbf{t}|\mathbf{w}, \mathbf{X}, \sigma^2, \Delta)p(\mathbf{w}|\Delta)}{p(\mathbf{t}|\mathbf{X}, \sigma^2, \Delta)} \\ &= \frac{p(\mathbf{t}|\mathbf{w}, \mathbf{X}, \sigma^2)p(\mathbf{w}|\Delta)}{p(\mathbf{t}|\mathbf{X}, \sigma^2, \Delta)} \end{aligned}$$

where Δ corresponds to some set of parameters required to define the prior over $\mathbf{w}$ that will be defined more precisely below. The graphical model can be seen in Figure 3.17. Expanding the marginal likelihood we have

$$p(\mathbf{w}|\mathbf{t}, \mathbf{X}, \sigma^2, \Delta) = \frac{p(\mathbf{t}|\mathbf{w}, \mathbf{X}, \sigma^2)p(\mathbf{w}|\Delta)}{\int p(\mathbf{t}|\mathbf{w}, \mathbf{X}, \sigma^2)p(\mathbf{w}|\Delta)\ d\mathbf{w}}. \tag{3.15}$$

We are interested in making predictions which will involve taking an expectation with respect to this posterior density. In particular, for a set of attributes $\mathbf{x}_{\text{new}}$ corresponding to a new Olympic year, the density over the associated winning time t_{new} is given by

$$p(t_{\text{new}}|\mathbf{x}_{\text{new}}, \mathbf{X}, \mathbf{t}, \sigma^2, \Delta) = \int p(t_{\text{new}}|\mathbf{x}_{\text{new}}, \mathbf{w}, \sigma^2)p(\mathbf{w}|\mathbf{t}, \mathbf{X}, \sigma^2, \Delta)\ d\mathbf{w}. \tag{3.16}$$

Notice again the conditioning on the right hand side. The posterior density of $\mathbf{w}$ does not depend on $\mathbf{x}_{\text{new}}$ and so it does not appear in the conditioning. Similarly, when we make predictions, we will not be using Δ and so it doesn't appear in $p(t_{\text{new}}|\mathbf{x}_{\text{new}}, \mathbf{w}, \sigma^2)$. Predictions could also take the form of probabilities. For example, we could compute the probability that the winning time will be under 9.5 seconds:

$$P(t_{\text{new}} < 9.5|\mathbf{x}_{\text{new}}, \mathbf{X}, \mathbf{t}, \sigma^2, \Delta) = \int P(t_{\text{new}} < 9.5|\mathbf{x}_{\text{new}}, \mathbf{w}, \sigma^2)p(\mathbf{w}|\mathbf{t}, \mathbf{X}, \sigma^2, \Delta)\ d\mathbf{w}. \tag{3.17}$$

3.8.2 The likelihood

The likelihood $p(\mathbf{t}|\mathbf{w}, \mathbf{X}, \sigma^2)$ is exactly the quantity that we maximised in the previous chapter. Our model tells us that

$$\mathbf{t} = \mathbf{X}\mathbf{w} + \boldsymbol{\epsilon}$$

where $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \sigma^2\mathbf{I}_N)$. This is a Gaussian random variable ($\boldsymbol{\epsilon}$) plus a constant. We showed in Section 2.8 that this is equivalent to the Gaussian random variable with the constant added to the mean. This gives us our likelihood

$$p(\mathbf{t}|\mathbf{w}, \mathbf{X}, \sigma^2) = \mathcal{N}(\mathbf{X}\mathbf{w}, \sigma^2\mathbf{I}_N),$$

an N-dimensional Gaussian density with mean $\mathbf{X}\mathbf{w}$ and variance $\sigma^2\mathbf{I}_N$. The analogous expression in the coin example is the binomial likelihood given in Equation 3.2.

3.8.3 The prior

Because we are interested in being able to produce an exact expression for our posterior, we need to choose a prior, $p(\mathbf{w}|\Delta)$, that is conjugate to the Gaussian likelihood. Conveniently, a Gaussian prior is conjugate to a Gaussian likelihood. Therefore, we will use a Gaussian prior for $\mathbf{w}$. In particular,

$$p(\mathbf{w}|\boldsymbol{\mu}_0, \boldsymbol{\Sigma}_0) = \mathcal{N}(\boldsymbol{\mu}_0, \boldsymbol{\Sigma}_0),$$

where we will choose the parameters $\boldsymbol{\mu}_0$ and $\boldsymbol{\Sigma}_0$ later. This is analogous to Equation 3.4 in the coin example. From now on we will not always explicitly condition on $\boldsymbol{\mu}_0$ and $\boldsymbol{\Sigma}_0$ in our expressions. For example, for brevity, instead of writing $p(\mathbf{w}|\mathbf{t}, \mathbf{X}, \sigma^2, \boldsymbol{\mu}_0, \boldsymbol{\Sigma}_0)$ we will use $p(\mathbf{w}|\mathbf{t}, \mathbf{X}, \sigma^2)$ (see Exercise 3.10).

3.8.4 The posterior

We now turn our attention to computing the posterior. As in the coin example, we will use the fact that we *know* that the posterior will be Gaussian. This allows us to ignore the marginal likelihood in Equation 3.15 and just manipulate the likelihood and prior until we find something that is proportional to a Gaussian. As a first step, we can collect the terms in $\mathbf{w}$ together and ignore any term that does not include $\mathbf{w}$:

$$\begin{aligned}
p(\mathbf{w}|\mathbf{t}, \mathbf{X}, \sigma^2) &\propto p(\mathbf{t}|\mathbf{w}, \mathbf{X}, \sigma^2)p(\mathbf{w}|\boldsymbol{\mu}_0, \boldsymbol{\Sigma}_0) \\
&= \frac{1}{(2\pi)^{N/2}|\sigma^2\mathbf{I}|^{1/2}} \exp\left(-\frac{1}{2}(\mathbf{t} - \mathbf{X}\mathbf{w})^\mathsf{T}(\sigma^2\mathbf{I})^{-1}(\mathbf{t} - \mathbf{X}\mathbf{w})\right) \\
&\quad \times \frac{1}{(2\pi)^{N/2}|\boldsymbol{\Sigma}_0|^{1/2}} \exp\left(-\frac{1}{2}(\mathbf{w} - \boldsymbol{\mu}_0)^\mathsf{T}\boldsymbol{\Sigma}_0^{-1}(\mathbf{w} - \boldsymbol{\mu}_0)\right) \\
&\propto \exp\left(-\frac{1}{2\sigma^2}(\mathbf{t} - \mathbf{X}\mathbf{w})^\mathsf{T}(\mathbf{t} - \mathbf{X}\mathbf{w})\right) \\
&\quad \times \exp\left(-\frac{1}{2}(\mathbf{w} - \boldsymbol{\mu}_0)^\mathsf{T}\boldsymbol{\Sigma}_0^{-1}(\mathbf{w} - \boldsymbol{\mu}_0)\right) \\
&= \exp\left\{-\frac{1}{2}\left(\frac{1}{\sigma^2}(\mathbf{t} - \mathbf{X}\mathbf{w})^\mathsf{T}(\mathbf{t} - \mathbf{X}\mathbf{w}) + (\mathbf{w} - \boldsymbol{\mu}_0)^\mathsf{T}\boldsymbol{\Sigma}_0^{-1}(\mathbf{w} - \boldsymbol{\mu}_0)\right)\right\}.
\end{aligned}$$

Multiplying the terms in the bracket out and once again removing any that don't involve $\mathbf{w}$ gives

$$p(\mathbf{w}|\mathbf{t}, \mathbf{X}, \sigma^2) \propto \exp\left\{-\frac{1}{2}\left(-\frac{2}{\sigma^2}\mathbf{t}^\mathsf{T}\mathbf{X}\mathbf{w} + \frac{1}{\sigma^2}\mathbf{w}^\mathsf{T}\mathbf{X}^\mathsf{T}\mathbf{X}\mathbf{w} + \mathbf{w}^\mathsf{T}\boldsymbol{\Sigma}_0^{-1}\mathbf{w} - 2\boldsymbol{\mu}_0^\mathsf{T}\boldsymbol{\Sigma}_0^{-1}\mathbf{w}\right)\right\}.$$

We know that the posterior will be Gaussian. Therefore we can remove the constants (i.e. terms not involving $\mathbf{w}$) and rearrange an expression for a multivariate Gaussian to make it look something like the expression we have above:

$$\begin{aligned} p(\mathbf{w}|\mathbf{t}, \mathbf{X}, \sigma^2) &= \mathcal{N}(\boldsymbol{\mu}_\mathbf{w}, \boldsymbol{\Sigma}_\mathbf{w}) \\ &\propto \exp\left(-\frac{1}{2}(\mathbf{w} - \boldsymbol{\mu}_\mathbf{w})^\mathsf{T}\boldsymbol{\Sigma}_\mathbf{w}^{-1}(\mathbf{w} - \boldsymbol{\mu}_\mathbf{w})\right) \\ &\propto \exp\left\{-\frac{1}{2}\left(\mathbf{w}^\mathsf{T}\boldsymbol{\Sigma}_\mathbf{w}^{-1}\mathbf{w} - 2\boldsymbol{\mu}_\mathbf{w}^\mathsf{T}\boldsymbol{\Sigma}_\mathbf{w}^{-1}\mathbf{w}\right)\right\}. \end{aligned} \tag{3.18}$$

The terms linear and quadratic in $\mathbf{w}$ in Equation 3.8.4 must be equal to those in Equation 3.18. Taking the quadratic terms, we can solve for $\boldsymbol{\Sigma}_\mathbf{w}$:

$$\begin{aligned} \mathbf{w}^\mathsf{T}\boldsymbol{\Sigma}_\mathbf{w}^{-1}\mathbf{w} &= \frac{1}{\sigma^2}\mathbf{w}^\mathsf{T}\mathbf{X}^\mathsf{T}\mathbf{X}\mathbf{w} + \mathbf{w}^\mathsf{T}\boldsymbol{\Sigma}_0^{-1}\mathbf{w} \\ &= \mathbf{w}^\mathsf{T}\left(\frac{1}{\sigma^2}\mathbf{X}^\mathsf{T}\mathbf{X} + \boldsymbol{\Sigma}_0^{-1}\right)\mathbf{w} \end{aligned}$$

$$\boldsymbol{\Sigma}_\mathbf{w} = \left(\frac{1}{\sigma^2}\mathbf{X}^\mathsf{T}\mathbf{X} + \boldsymbol{\Sigma}_0^{-1}\right)^{-1}.$$

Similarly, equating the linear terms from Equations 3.8.4 and 3.18 (and using our new expression for $\boldsymbol{\Sigma}_\mathbf{w}$) we can get an expression for $\boldsymbol{\mu}_\mathbf{w}$:

$$\begin{aligned} -2\boldsymbol{\mu}_\mathbf{w}^\mathsf{T}\boldsymbol{\Sigma}_\mathbf{w}^{-1}\mathbf{w} &= -\frac{2}{\sigma^2}\mathbf{t}^\mathsf{T}\mathbf{X}\mathbf{w} - 2\boldsymbol{\mu}_0^\mathsf{T}\boldsymbol{\Sigma}_0^{-1}\mathbf{w} \\ \boldsymbol{\mu}_\mathbf{w}^\mathsf{T}\boldsymbol{\Sigma}_\mathbf{w}^{-1}\mathbf{w} &= \frac{1}{\sigma^2}\mathbf{t}^\mathsf{T}\mathbf{X}\mathbf{w} + \boldsymbol{\mu}_0^\mathsf{T}\boldsymbol{\Sigma}_0^{-1}\mathbf{w} \\ \boldsymbol{\mu}_\mathbf{w}^\mathsf{T}\boldsymbol{\Sigma}_\mathbf{w}^{-1} &= \frac{1}{\sigma^2}\mathbf{t}^\mathsf{T}\mathbf{X} + \boldsymbol{\mu}_0^\mathsf{T}\boldsymbol{\Sigma}_0^{-1} \\ \boldsymbol{\mu}_\mathbf{w}^\mathsf{T}\boldsymbol{\Sigma}_\mathbf{w}^{-1}\boldsymbol{\Sigma}_\mathbf{w} &= \left(\frac{1}{\sigma^2}\mathbf{t}^\mathsf{T}\mathbf{X} + \boldsymbol{\mu}_0^\mathsf{T}\boldsymbol{\Sigma}_0^{-1}\right)\boldsymbol{\Sigma}_\mathbf{w} \\ \boldsymbol{\mu}_\mathbf{w}^\mathsf{T} &= \left(\frac{1}{\sigma^2}\mathbf{t}^\mathsf{T}\mathbf{X} + \boldsymbol{\mu}_0^\mathsf{T}\boldsymbol{\Sigma}_0^{-1}\right)\boldsymbol{\Sigma}_\mathbf{w} \end{aligned}$$

$$\boldsymbol{\mu}_\mathbf{w} = \boldsymbol{\Sigma}_\mathbf{w}\left(\frac{1}{\sigma^2}\mathbf{X}^\mathsf{T}\mathbf{t} + \boldsymbol{\Sigma}_0^{-1}\boldsymbol{\mu}_0\right), \tag{3.19}$$

because $\boldsymbol{\Sigma}_\mathbf{w}^\mathsf{T} = \boldsymbol{\Sigma}_\mathbf{w}$ due to the fact that it must be symmetric. Therefore,

$$p(\mathbf{w}|\mathbf{t}, \mathbf{X}, \sigma^2) = \mathcal{N}(\boldsymbol{\mu}_\mathbf{w}, \boldsymbol{\Sigma}_\mathbf{w}) \tag{3.20}$$

where

$$\boldsymbol{\Sigma}_\mathbf{w} = \left(\frac{1}{\sigma^2}\mathbf{X}^\mathsf{T}\mathbf{X} + \boldsymbol{\Sigma}_0^{-1}\right)^{-1} \tag{3.21}$$

$$\boldsymbol{\mu}_\mathbf{w} = \boldsymbol{\Sigma}_\mathbf{w}\left(\frac{1}{\sigma^2}\mathbf{X}^\mathsf{T}\mathbf{t} + \boldsymbol{\Sigma}_0^{-1}\boldsymbol{\mu}_0\right) \tag{3.22}$$

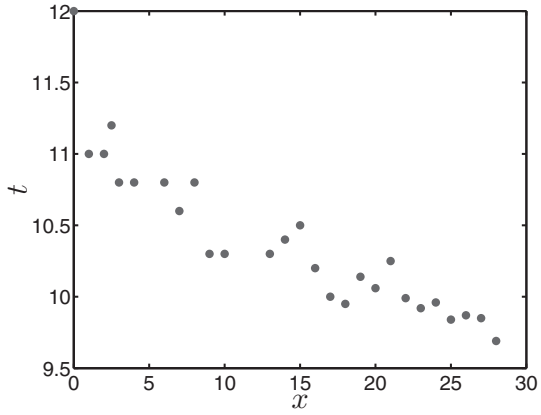


FIGURE 3.18 Olympic data with rescaled x values.

(see Exercise 3.12). These expressions do not look too far away from things we have seen before. In particular, compare Equation 3.22 with the regularised least squares solution given in Equation 1.21. In fact, if $\boldsymbol{\mu}_0 = [0, 0, \ldots, 0]^\mathsf{T}$, the expressions are almost identical. Given that the posterior is a Gaussian, the single most likely value of $\mathbf{w}$ is the mean of the posterior, $\boldsymbol{\mu}_\mathbf{w}$. This is known as the **maximum a posteriori** (MAP) estimate of $\mathbf{w}$ and can also be thought of as the maximum value of the joint density $p(\mathbf{w}, \mathbf{t}|\mathbf{X}, \sigma^2, \Delta)$ (the likelihood multiplied by the prior). We have already seen that the squared loss considered in Chapter 1 is very similar to a Gaussian likelihood and it follows from this that computing the most likely posterior value (when the likelihood is Gaussian) is equivalent to using regularised least squares (see Exercise 3.9). This comparison can often help to provide intuition regarding the effect of the prior.

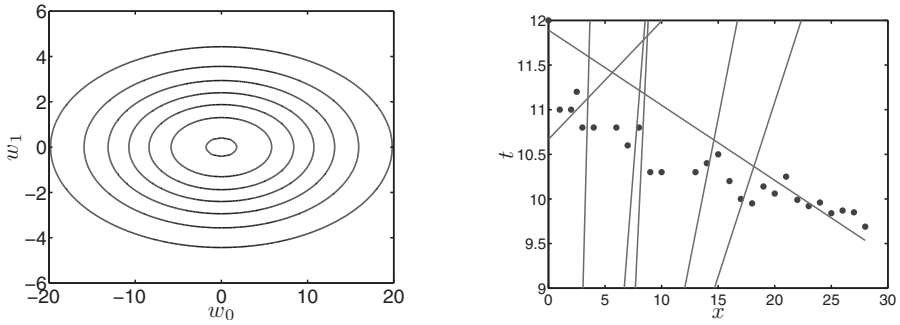
3.8.5 A first-order polynomial

We will illustrate the prior and posterior with a first-order polynomial, as it is possible to visualise densities in the two-dimensional parameter space. The input vectors also have two elements, $\mathbf{x}_n = [1,\ x_n]^\mathsf{T}$. To aid visualisation, we will rescale the Olympic year by subtracting the year of the first Olympics (1896) from each year and then dividing each number by 4. This means that x_1 is now 0, x_2 is 1, etc. The data with this new x scaling is plotted in Figure 3.18.

Returning to the fairground, the first step in our analysis is the choice of prior parameters $\boldsymbol{\mu}_0$ and $\boldsymbol{\Sigma}_0$. For $\boldsymbol{\mu}_0$, we will assume that we don't really know anything about what the parameters should be and choose $\boldsymbol{\mu}_0 = [0, 0]^\mathsf{T}$. For the covariance, we will use

$$\boldsymbol{\Sigma}_0 = \begin{bmatrix} 100 & 0 \\ 0 & 5 \end{bmatrix}.$$

The larger value for the variance of w_0 is due to the fact that we saw in the maximum likelihood estimate that the optimal value of w_0 was much higher than that for w_1. We have also assumed that the two variables are independent in the prior by setting



(a) Prior density.

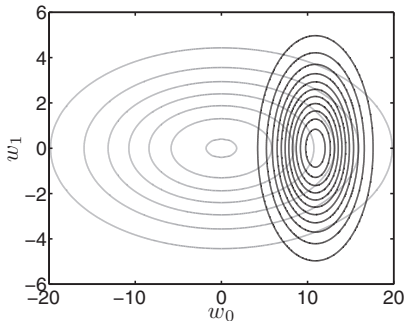
(b) Functions created from parameters drawn from prior.

FIGURE 3.19 Gaussian prior used for the Olympic 100 m data (a) and some functions created with samples drawn from the prior (b).

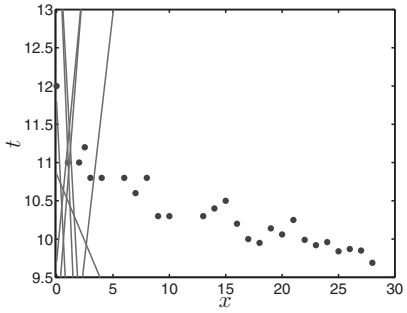
the off-diagonal elements in the covariance matrix to zero. This does not preclude them from being dependent in the posterior. The contours of this prior density can be seen in Figure 3.19(a). It's hard to visualise what this means in terms of the model. To help, in Figure 3.19(b) we have shown functions corresponding to several sets of parameters drawn from this prior. To create these, we sampled $\mathbf{w}$ from the Gaussian defined by $\boldsymbol{\mu}_0$ and $\boldsymbol{\Sigma}_0$ and then substituted these into our linear model – $t_n = w_0 + w_1 x_n$. The examples show that the prior admits the possibility of many very different models.

Using $\sigma^2 = 10$ for illustrative purposes (MATLAB script: `olympbayes.m`), we can now compute the posterior distribution when we observe one data point. Using the data point corresponding to the first Olympics, our data is summarised as $\mathbf{x} = [1, 0]^\mathsf{T}$, $\mathbf{X} = [1, 0]$, $\mathbf{t} = [12]$. Plugging these values along with our prior parameters and $\sigma^2 = 10$ into Equations 3.20–3.22, we obtain the posterior distribution shown in Figure 3.20(a). The posterior now has much more certainty regarding w_0 but still knows very little about w_1. This makes sense – we've provided a data point at $x = 0$ so this should be highly informative in determining the intercept but tells us very little about the gradient (one data point alone could never tell us much about the gradient). Some functions created with samples from this posterior are shown in Figure 3.20(b). They look quite different from those from the prior – in particular, they all pass quite close to our first data point.

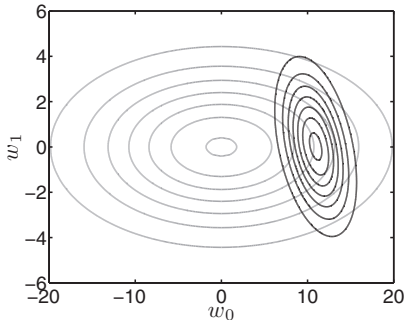
Figures 3.20(c), 3.20(d) and 3.20(e) show the evolution of the posterior after 2, 5 and 10 data points, respectively. Just as in the coin example, we notice that the posterior becomes more condensed (we are becoming more certain about the value of $\mathbf{w}$). Also, as it evolves, the posterior begins to tilt. This is indicative of a dependence developing between the two parameters – if we increase the intercept w_0, we must decrease the gradient. Recall that, in the prior, we assumed that the two parameters were independent ($\boldsymbol{\Sigma}_0$ only had non-zero values on the diagonal) so this dependence is coming entirely from the evidence within the data. To help visualise what the posterior means at this stage, Figure 3.20(f) shows a set of functions made from parameters drawn from the posterior. When compared with Figure 3.20(b),



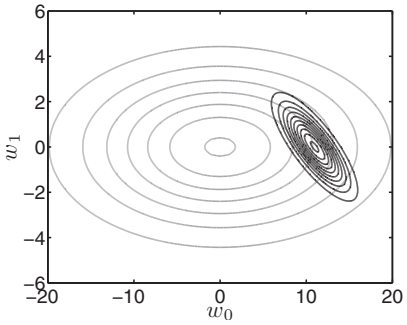
(a) Posterior density (dark contours) after the first data point has been observed. The lighter contours show the prior density.



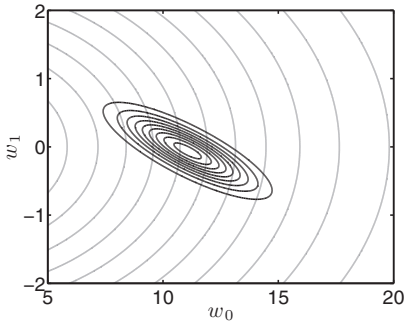
(b) Functions created from parameters drawn from the posterior after observing the first data point.



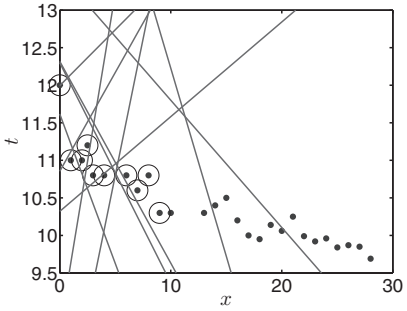
(c) Posterior density (dark contours) after the first two data points have been observed.



(d) Posterior density (dark contours) after the first five data points have been observed.

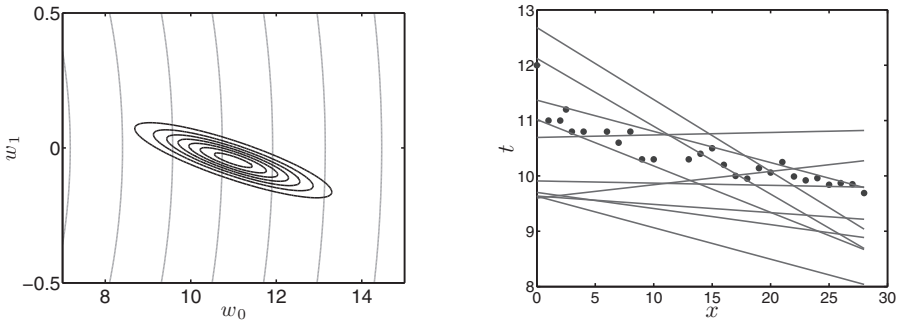


(e) Posterior density (dark contours) after the first ten data points have been observed. (Note that we have zoomed in.)



(f) Functions created from parameters drawn from the posterior after observing the first ten (highlighted) data points.

FIGURE 3.20 Evolution of the posterior density and example functions drawn from the posterior for the Olympic data as observations are added.



(a) Posterior density (dark contours) after all datapoints have been observed. The lighter contours show the prior density. (Note that we have zoomed in.)

(b) Functions created from parameters drawn from the posterior after observing all data points.

FIGURE 3.21 Posterior density (a) and sampled functions (b) for the Olympic data when all 27 data points have been added.

we see that the posterior density is beginning to favour parameters that correspond to models suited to our data. Finally, in Figure 3.21(a) we see the posterior after all 27 data points have been included and in Figure 3.21(b) we see functions drawn from this posterior. The functions are really now beginning to follow the trend in our data. There is still a lot of variability though. This is due to the relatively high value of $\sigma^2 = 10$ that we chose to help visualise the prior and posteriors. For making predictions, we might want to use a more realistic value. In Figure 3.22(a) we show the posterior after all data has been observed for $\sigma^2 = 0.05$ (this is roughly the maximum likelihood value we obtained in Section 2.8.2). The posterior is now far more condensed – very little variability remains in $\mathbf{w}$, as can be seen by the homogeneity of the set of functions drawn in Figure 3.22(b). We will now turn our attention to making predictions.

3.8.6 Making predictions

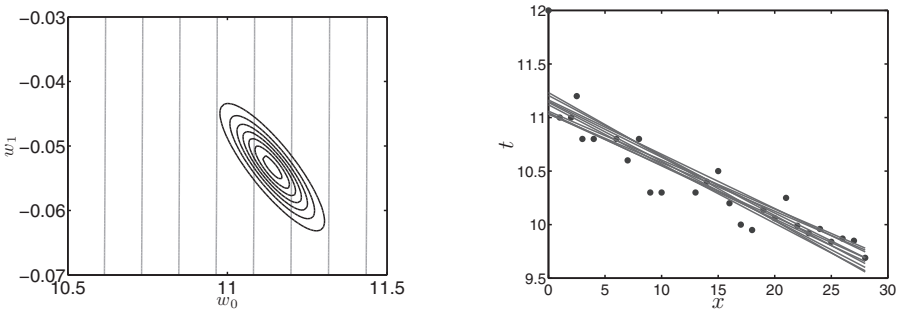
Given a new observation $\mathbf{x}_{\text{new}}$, we are interested in the density

$$p(t_{\text{new}}|\mathbf{x}_{\text{new}}, \mathbf{X}, \mathbf{t}, \sigma^2).$$

Notice that this is not conditioned on $\mathbf{w}$ – just as in the coin example, we are going to integrate out $\mathbf{w}$ by taking an expectation with respect to the posterior, $p(\mathbf{w}|\mathbf{t}, \mathbf{X}, \sigma^2)$. In particular, we need to compute

$$\begin{aligned} p(t_{\text{new}}|\mathbf{x}_{\text{new}}, \mathbf{X}, \mathbf{t}, \sigma^2) &= \mathbf{E}_{p(\mathbf{w}|\mathbf{t},\mathbf{X},\sigma^2)}\left\{p(t_{\text{new}}|\mathbf{x}_{\text{new}}, \mathbf{w}, \sigma^2)\right\} \\ &= \int p(t_{\text{new}}|\mathbf{x}_{\text{new}}, \mathbf{w}, \sigma^2) p(\mathbf{w}|\mathbf{t}, \mathbf{X}, \sigma^2)\, d\mathbf{w}. \end{aligned}$$

This is analogous to Equation 3.9 in the coin example.



(a) Posterior density (dark contours) after all data points have been observed. The lighter contours show the prior density. (Note that we have zoomed in.)

(b) Functions created from parameters drawn from the posterior after observing all data points.

FIGURE 3.22 Posterior density (a) and sampled functions (b) for the Olympic data when all 27 data points have been added with more realistic noise variance, $\sigma^2 = 0.05$.

$p(t_{\mathsf{new}}|\mathbf{x}_{\mathsf{new}}, \mathbf{w}, \sigma^2)$ is defined by our model as the product of $\mathbf{x}_{\mathsf{new}}$ and $\mathbf{w}$ with some additive Gaussian noise:

$$p(t_{\mathsf{new}}|\mathbf{x}_{\mathsf{new}}, \mathbf{w}, \sigma^2) = \mathcal{N}(\mathbf{x}_{\mathsf{new}}^{\mathsf{T}}\mathbf{w}, \sigma^2).$$

Because this expression and the posterior are both Gaussian, the result of the expectation is another Gaussian. In general, if $p(\mathbf{w}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) = \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$, then the expectation of another Gaussian density ($\mathcal{N}(\mathbf{x}_{\mathsf{new}}^{\mathsf{T}}\mathbf{w}, \sigma^2)$) is given by

$$p(t_{\mathsf{new}}|\mathbf{x}_{\mathsf{new}}, \mathbf{X}, \mathbf{t}, \sigma^2) = \mathcal{N}(\mathbf{x}_{\mathsf{new}}^{\mathsf{T}}\boldsymbol{\mu}_{\mathbf{w}}, \sigma^2 + \mathbf{x}_{\mathsf{new}}^{\mathsf{T}}\boldsymbol{\Sigma}_{\mathbf{w}}\mathbf{x}_{\mathsf{new}}).$$

For the posterior shown in Figure 3.22(a), this is

$$p(t_{\mathsf{new}}|\mathbf{x}_{\mathsf{new}}, \mathbf{X}, \mathbf{t}, \sigma^2) = \mathcal{N}(9.5951, 0.0572)$$

and is plotted in Figure 3.23.

This density looks rather like the predictive densities we obtained from the maximum likelihood solution in Chapter 2. However, there is one crucial difference. With the maximum likelihood we chose one particular model: the one corresponding to the highest likelihood. To generate the density shown in Figure 3.23, we have averaged over all models that are consistent with our data and prior (we averaged over our posterior). Hence this density takes into account all uncertainty that remains in $\mathbf{w}$ given a particular prior and the data.

3.9 MARGINAL LIKELIHOOD FOR POLYNOMIAL MODEL ORDER SELECTION

In Section 1.5 we used a cross-validation procedure to select the order of polynomial to be used. The cross-validation procedure correctly identified that the dataset was

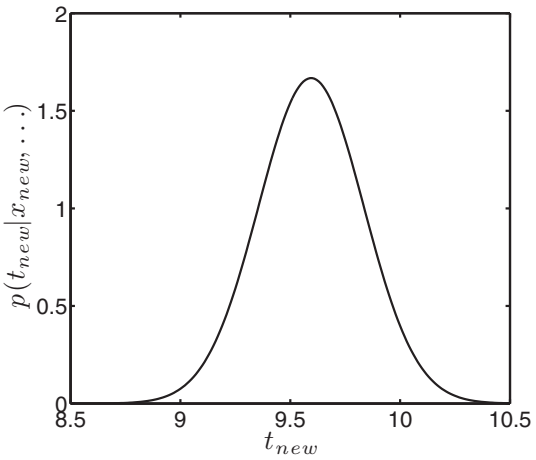


FIGURE 3.23 Predictive distribution for the winning time in the men's 100 m sprint at the 2012 London Olympics.

generated from a third-order polynomial. In Section 3.4 we saw how the marginal likelihood could be used to choose prior densities. We will now see that it can also be used to choose models. In particular, we will use it to determine which order polynomial function to use for some synthetic data.

The marginal likelihood for our Gaussian model is defined as

$$p(\mathbf{t}|\mathbf{X}, \boldsymbol{\mu}_0, \boldsymbol{\Sigma}_0) = \int p(\mathbf{t}|\mathbf{X}, \mathbf{w}, \sigma^2) p(\mathbf{w}|\boldsymbol{\mu}_0, \boldsymbol{\Sigma}_0)\ d\mathbf{w}.$$

This is analogous to Equation 3.14 in the coin example. It is of the same form as the predictive density discussed in the previous section and is another Gaussian,

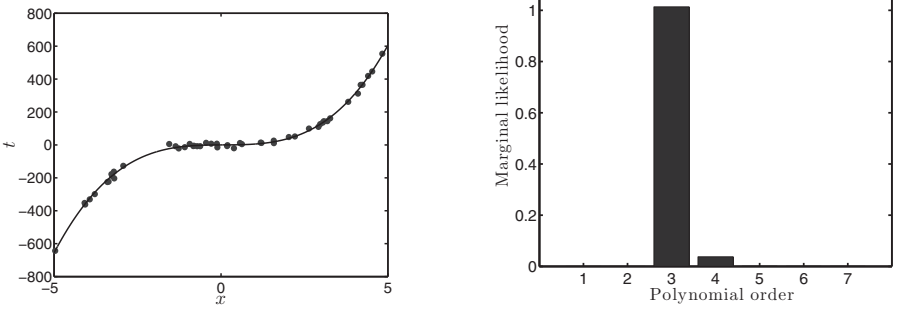
$$p(\mathbf{t}|\mathbf{X}, \boldsymbol{\mu}_0, \boldsymbol{\Sigma}_0) = \mathcal{N}(\mathbf{X}\boldsymbol{\mu}_0, \sigma^2\mathbf{I}_N + \mathbf{X}\boldsymbol{\Sigma}_0\mathbf{X}^\mathsf{T}), \tag{3.23}$$

which we evaluate at $\mathbf{t}$ – the responses in the training set. Just as in Section 1.5, we will generate data from a noisy third-order polynomial and then compute the marginal likelihood for models from first to seventh-order. For each possible model, we will use a Gaussian prior on $\mathbf{w}$ with zero mean and an identity covariance matrix. For example, for the first-order model,

$$\boldsymbol{\mu}_0 = [0, 0]^\mathsf{T},\ \boldsymbol{\Sigma}_0 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

and for the fourth-order model

$$\boldsymbol{\mu}_0 = [0, 0, 0, 0, 0]^\mathsf{T},\ \boldsymbol{\Sigma}_0 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}.$$



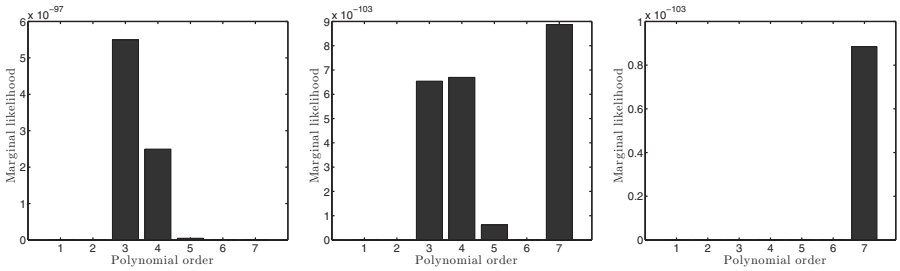
(a) Noisy data from a third-order polynomial.

(b) Marginal likelihood for models of different order.

FIGURE 3.24 Dataset sampled from the function $t = 5x^3 - x^2 + x$ (a) and marginal likelihoods for polynomials of increasing order (b).

The data and true polynomial are shown in Figure 3.24(a) (MATLAB script: `margpoly.m`). The true polynomial is $t = 5x^3 - x^2 + x$ and Gaussian noise has been added with mean zero and variance 150. The marginal likelihood for models from first to seventh order is calculated by plugging the relevant prior into Equation 3.23 and then evaluating this density at $\mathbf{t}$, the observed responses. The values are shown in Figure 3.24(b). We can see that the marginal likelihood value is very sharply peaked at the true third-order model. The advantage of this over the cross-validation method is that, for this model, it is computationally undemanding (we don't have to fit several different datasets). We can also use all the data. However, as we have already mentioned, calculating the marginal likelihood is, in general, very difficult and we will often find it easier to resort to cross-validation techniques.

The marginal likelihood is conditioned on the prior parameters and so changing them will have an effect on the marginal likelihood values and possibly the highest scoring model. To show the effect of this, we can define $\mathbf{\Sigma}_0 = \sigma_0^2\mathbf{I}$ and vary σ_0^2. We have already seen the result for $\sigma_0^2 = 1$. If we decrease σ_0^2, we see higher-order models performing better. This can be seen in Figure 3.25. Decreasing σ_0^2 from 1 to 0.3 results in the seventh-order polynomial becoming the most likely model. By decreasing σ_0^2, we are saying that the parameters have to take smaller and smaller values. For a third order polynomial model to fit well, one of the parameters needs to be 5 (recall that $t = 5x^3 - x^2 + x$). As we decrease σ_0^2, this becomes less and less likely, and higher-order models with lower parameter values become more likely. This emphasises the importance of understanding what we mean by a model. In this example, the model consists of the order of polynomial *and* the prior specification and we must be careful to choose the prior sensibly (see Exercise 3.11).



(a) $\sigma_0^2 = 0.7$ (b) $\sigma_0^2 = 0.4$ (c) $\sigma_0^2 = 0.3$

FIGURE 3.25 Marginal likelihoods for the third-order polynomial example with $\mathbf{\Sigma}_0 = \sigma_0^2\mathbf{I}$ as σ_0^2 is decreased.

3.10 CHAPTER SUMMARY

This chapter has provided an introduction to the Bayesian way of performing Machine Learning tasks – treating all parameters as random variables. We have performed a Baysesian analysis for a coin tossing model and the linear regression model introduced in Chapters 1 and 2. In both cases, we defined prior densities over parameters, defined likelihoods and computed posterior densities. In both examples, the prior and likelihood were chosen such that the posterior could be computed analytically. In addition, we computed predictions by taking expectations with respect to the posterior and introduced marginal likelihood as a possible model selection criterion.

Unfortunately, these expressions are not often analytically tractable and we must resort to sampling and approximation techniques. These techniques are the foundations of modern Bayesian inference and form an important area of Machine Learning research and development. The next chapter will describe three popular techniques – point estimates, Laplace approximations and Markov chain Monte Carlo.

3.11 EXERCISES

3.1 For $\alpha, \beta = 1$, the beta distribution becomes uniform between 0 and 1. In particular, if the probability of a coin landing heads is given by r and a beta prior is placed over r, with parameters $\alpha = 1, \beta = 1$, this prior can be written as

$$p(r) = 1 \quad (0 \leq r \leq 1).$$

Using this prior, compute the posterior density for r if y heads are observed in N tosses (i.e. multiply this prior by the binomial likelihood and manipulate the result to obtain something that looks like a beta density).

3.2 Repeat the previous exercise for the following prior, also a particular form of the beta density:

$$p(r) = \begin{cases} 2r & 0 \leq r \leq 1 \\ 0 & \text{otherwise} \end{cases}$$

What are the values of the prior parameters α and β that result in $p(r) = 2r$?

3.3 Repeat the previous exercise for the following prior (again, a form of beta density):

$$p(r) = \begin{cases} 3r^2 & 0 \leq r \leq 1 \\ 0 & \text{otherwise.} \end{cases}$$

What are the prior parameters here?

3.4 What are the effective prior sample sizes (α and β) for the previous three exercises (i.e. how many heads and tails are they equivalent to)?

3.5 If a random variable R has a beta density

$$p(r) = \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)} r^{\alpha-1}(1-r)^{\beta-1},$$

derive an expression for the expected value of r, $\mathbf{E}_{p(r)}\{r\}$. You will need the following identity for the gamma function:

$$\Gamma(n+1) = n\Gamma(n).$$

Hint: Use the fact that

$$\int_{r=0}^{r=1} r^{a-1}(1-r)^{b-1}\, dr = \frac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)}.$$

3.6 Using the setup in the previous exercise, and the identity

$$\mathsf{var}\{r\} = \mathbf{E}_{p(r)}\left\{r^2\right\} - \left(\mathbf{E}_{p(r)}\{r\}\right)^2,$$

derive an expression for $\mathsf{var}\{r\}$. You will need the gamma identity given in the previous exercise.

3.7 At a different stall, you observe 20 tosses of which 9 were heads. Compute the posteriors for the three scenarios, the probability of winning in each case and the marginal likelihoods.

3.8 Use MATLAB to generate coin tosses where the probability of heads is 0.7. Generate 100 tosses and compute the posteriors for the three scenarios, the probabilities of winning and the marginal likelihoods.

3.9 In Section 3.8.4 we derived an expression for the Gaussian posterior for a linear model within the context of the Olympic 100 m data. Substituting $\boldsymbol{\mu}_0 = [0, 0, \ldots, 0]^\mathsf{T}$, we saw the similarity between the posterior mean

$$\boldsymbol{\mu}_\mathbf{w} = \frac{1}{\sigma^2}\left(\frac{1}{\sigma^2}\mathbf{X}^\mathsf{T}\mathbf{X} + \boldsymbol{\Sigma}_0^{-1}\right)^{-1}\mathbf{X}^\mathsf{T}\mathbf{t}$$

and the regularised least squares solution

$$\widehat{\mathbf{w}} = \left(\mathbf{X}^\mathsf{T}\mathbf{X} + N\lambda\mathbf{I}\right)^{-1}\mathbf{X}^\mathsf{T}\mathbf{t}.$$

For this particular example, find the prior covariance matrix $\boldsymbol{\Sigma}_0$ that makes the two identical. In other words, find $\boldsymbol{\Sigma}_0$ in terms of λ.

3.10 Redraw the graphical representation of the Olympic 100 m model to reflect the fact that the prior over $\mathbf{w}$ is actually conditioned on $\boldsymbol{\mu}_0$ and $\boldsymbol{\Sigma}_0$.

3.11 In Figure 3.25 we studied the effect of reducing σ_0^2 on the marginal likelihood. Using MATLAB, investigate the effect of increasing σ_0^2.

3.12 When performing a Bayesian analysis on the Olympics data, we assumed that the prior was known. If a Gaussian prior is placed on $\mathbf{w}$ and an inverse gamma prior on the variance σ^2

$$p(\sigma^2|\alpha,\beta) = \frac{\beta^\alpha}{\Gamma(\alpha)}(\sigma^2)^{-\alpha-1}\exp\left\{-\frac{\beta}{\sigma^2}\right\},$$

the posterior will also be the product of a Gaussian and an inverse gamma. Compute the posterior parameters.

3.12 FURTHER READING

[1] Ben Calderhead and Mark Girolami. Estimating Bayes factors via thermodynamic integration and population MCMC. *Comput. Stat. Data Anal.*, 53:4028–4045, October 2009.

An article by the authors describing a novel approach for calculating the marginal likelihoods (Bayes factors) in models where it is not analytically tractable.

[2] Andrew Gelman, John B. Carlin, Hal S. Stern, and Donald B. Rubin. *Bayesian Data Analysis*. Chapman and Hall/CRC, second edition, 2004.

One of the most popular textbooks on Bayesian inference. Provides a detailed and practical description of Bayesian Inference.

[3] Michael Isard and Andrew Blake. Contour tracking by stochastic propagation of conditional density. In *European Conference on Computer Vision*, pages 343–356, 1996.

An interesting example of the use of Bayesian methods in the field of human computer interaction. The authors use a sampling technique to infer posterior probabilities over gestures being performed by users.

[4] Michael Jordan, editor. *Learning in Graphical Models*. MIT Press, 1999.

An introduction to the field of graphical models and how to use them for learning tasks.

[5] Christian Robert. *The Bayesian Choice: From Decision-Theoretic Foundations to Computational Implementation*. Springer, second edition edition, 2007.

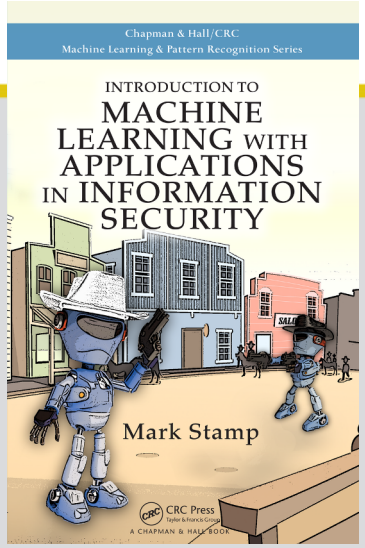
[6] Tian-Rui Xu et al. Inferring signaling pathway topologies from multiple perturbation measurement of specific biochemical species. *Science Signalling*, 3(113), 2010.

A paper showing how Bayesian model selection via the marginal likelihood can be used to answer interesting scientific questions in the field of biology. It is also an interesting example of large-scale Bayesian sampling.

CHAPTER

3

A REVEALING INTRODUCTION TO HIDDEN MARKOV MODELS



This chapter is excerpted from

Introduction to Machine Learning with Applications in Information Security

by Mark Stamp.

© 2018 Taylor & Francis Group. All rights reserved.

Learn more

The cause is hidden. The effect is visible to all.
— Ovid

Introduction and Background

Not surprisingly, a hidden Markov model (HMM) includes a Markov process that is "hidden," in the sense that we cannot directly observe the state of the process. But we do have access to a series of observations that are probabilistically related to the underlying Markov model.

While the formulation of HMMs might initially seem somewhat contrived, there exist a virtually unlimited number of problems where the technique can be applied. Best of all, there are efficient algorithms, making HMMs extremely practical. Another very nice property of an HMM is that structure within the data can often be deduced from the model itself.

In this chapter, we first consider a simple example to motivate the HMM formulation. Then we dive into a detailed discussion of the HMM algorithms. Realistic applications—mostly from the information security domain—can be found in Chapter 9.

This is one of the most detailed chapters in the book. A reason for going into so much depth is that once we have a solid understanding of this particular machine learning technique, we can then compare and contrast it to the other techniques that we'll consider. In addition, HMMs are relatively easy to understand—although the notation can seem intimidating, once you have the intuition, the process is actually fairly straightforward.[1]

[1]To be more accurate, your dictatorial author wants to start with HMMs, and that's all that really matters.

The bottom line is that this chapter is the linchpin for much of the remainder of the book. Consequently, if you learn the material in this chapter well, it will pay large dividends in most subsequent chapters. On the other hand, if you fail to fully grasp the details of HMMs, then much of the remaining material will almost certainly be more difficult than is necessary.

HMMs are based on discrete probability. In particular, we'll need some basic facts about conditional probability, so in the remainder of this section, we provide a quick overview of this crucial topic.

The notation "|" denotes "given" information, so that $P(B \mid A)$ is read as "the probability of B, given A." For any two events A and B, we have

$$P(A \text{ and } B) = P(A)\,P(B \mid A). \tag{2.1}$$

For example, suppose that we draw two cards without replacement from a standard 52-card deck. Let $A = \{1^{\text{st}} \text{ card is ace}\}$ and $B = \{2^{\text{nd}} \text{ card is ace}\}$. Then

$$P(A \text{ and } B) = P(A)\,P(B \mid A) = 4/52 \cdot 3/51 = 1/221.$$

In this example, $P(B)$ depends on what happens in the first event A, so we say that A and B are *dependent* events. On the other hand, suppose we flip a fair coin twice. Then the probability that the second flip comes up heads is $1/2$, regardless of the outcome of the first coin flip, so these events are *independent*. For dependent events, the "given" information is relevant when determining the sample space. Consequently, in such cases we can view the information to the right of the "given" sign as defining the space over which probabilities will be computed.

We can rewrite equation (2.1) as

$$P(B \mid A) = \frac{P(A \text{ and } B)}{P(A)}.$$

This expression can be viewed as the definition of conditional probability. For an important application of conditional probability, see the discussion of naïve Bayes in Section 7.8 of Chapter 7.

We'll often use the shorthand "A, B" for the joint probability which, in reality is the same as "A and B." Also, in discrete probability, "A and B" is equivalent to the intersection of the sets A and B and sometimes we'll want to emphasize this set intersection. Consequently, throughout this section

$$P(A \text{ and } B) = P(A, B) = P(A \cap B).$$

Finally, matrix notation is used frequently in this chapter. A review of matrices and basic linear algebra can be found in Section 4.2.1 of Chapter 4, although no linear algebra is required in this chapter.

A Simple Example

Suppose we want to determine the average annual temperature at a particular location on earth over a series of years. To make it more interesting, suppose the years we are focused on lie in the distant past, before thermometers were invented. Since we can't go back in time, we instead look for indirect evidence of the temperature.

To simplify the problem, we only consider "hot" and "cold" for the average annual temperature. Suppose that modern evidence indicates that the probability of a hot year followed by another hot year is 0.7 and the probability that a cold year is followed by another cold year is 0.6. We'll assume that these probabilities also held in the distant past. This information can be summarized as

$$\begin{array}{c} \\ H \\ C \end{array} \begin{array}{c} \begin{array}{cc} H & C \end{array} \\ \begin{pmatrix} 0.7 & 0.3 \\ 0.4 & 0.6 \end{pmatrix} \end{array} \tag{2.2}$$

where H is "hot" and C is "cold."

Next, suppose that current research indicates a correlation between the size of tree growth rings and temperature. For simplicity, we only consider three different tree ring sizes, small, medium, and large, denoted S, M, and L, respectively. Furthermore, suppose that based on currently available evidence, the probabilistic relationship between annual temperature and tree ring sizes is given by

$$\begin{array}{c} \\ H \\ C \end{array} \begin{array}{c} \begin{array}{ccc} S & M & L \end{array} \\ \begin{pmatrix} 0.1 & 0.4 & 0.5 \\ 0.7 & 0.2 & 0.1 \end{pmatrix} \end{array}. \tag{2.3}$$

For this system, we'll say that the *state* is the average annual temperature, either H or C. The transition from one state to the next is a *Markov process*,[2] since the next state depends only on the current state and the fixed probabilities in (2.2). However, the actual states are "hidden" since we can't directly observe the temperature in the past.

Although we can't observe the state (temperature) in the past, we can observe the size of tree rings. From (2.3), tree rings provide us with probabilistic information regarding the temperature. Since the underlying states are hidden, this type of system is known as a *hidden Markov model* (HMM). Our goal is to make effective and efficient use of the observable information, so as to gain insight into various aspects of the Markov process.

[2] A Markov process where the current state only depends on the previous state is said to be of order one. In a Markov process of order n, the current state depends on the n consecutive preceding states. In any case, the "memory" is finite—much like your absent-minded author's memory, which seems to become more and more finite all the time. Let's see, now where was I?

For this HMM example, the state transition matrix is

$$A = \begin{pmatrix} 0.7 & 0.3 \\ 0.4 & 0.6 \end{pmatrix}, \tag{2.4}$$

which comes from (2.2), and the observation matrix is

$$B = \begin{pmatrix} 0.1 & 0.4 & 0.5 \\ 0.7 & 0.2 & 0.1 \end{pmatrix}, \tag{2.5}$$

which comes from (2.3). For this example, suppose that the initial state distribution, denoted by π, is

$$\pi = \begin{pmatrix} 0.6 & 0.4 \end{pmatrix}, \tag{2.6}$$

that is, the chance that we start in the H state is 0.6 and the chance that we start in the C state is 0.4. The matrices π, A, and B are *row stochastic*, which is just a fancy way of saying that each row satisfies the requirements of a discrete probability distribution (i.e., each element is between 0 and 1, and the elements of each row sum to 1).

Now, suppose that we consider a particular four-year period of interest from the distant past. For this particular four-year period, we observe the series of tree ring sizes S, M, S, L. Letting 0 represent S, 1 represent M, and 2 represent L, this observation sequence is denoted as

$$\mathcal{O} = \big(0, 1, 0, 2\big). \tag{2.7}$$

We might want to determine the most likely state sequence of the Markov process given the observations (2.7). That is, we might want to know the most likely average annual temperatures over this four-year period of interest. This is not quite as clear-cut as it seems, since there are different possible interpretations of "most likely." On the one hand, we could define "most likely" as the state sequence with the highest probability from among all possible state sequences of length four. Dynamic programming (DP) can be used to efficiently solve this problem. On the other hand, we might reasonably define "most likely" as the state sequence that maximizes the expected number of correct states. An HMM can be used to find the most likely hidden state sequence in this latter sense.

It's important to realize that the DP and HMM solutions to this problem are not necessarily the same. For example, the DP solution must, by definition, include valid state transitions, while this is not the case for the HMM. And even if all state transitions are valid, the HMM solution can still differ from the DP solution, as we'll illustrate in an example below.

Before going into more detail, we need to deal with the most challenging aspect of HMMs—the notation. Once we have the notation, we'll discuss the

three fundamental problems that HMMs enable us to solve, and we'll give detailed algorithms for the efficient solution of each. We also consider critical computational issues that must be addressed when writing any HMM computer program. Rabiner [113] is a standard reference for further introductory information on HMMs.

Notation

The notation used in an HMM is summarized in Table 2.1. Note that the observations are assumed to come from the set $\{0, 1, \ldots, M-1\}$, which simplifies the notation with no loss of generality. That is, we simply associate each of the M distinct observations with one of the elements $0, 1, \ldots, M-1$, so that $\mathcal{O}_i \in V = \{0, 1, \ldots, M-1\}$ for $i = 0, 1, \ldots, T-1$.

Table 2.1: HMM notation

Notation	Explanation
T	Length of the observation sequence
N	Number of states in the model
M	Number of observation symbols
Q	Distinct states of the Markov process, $q_0, q_1, \ldots, q_{N-1}$
V	Possible observations, assumed to be $0, 1, \ldots, M-1$
A	State transition probabilities
B	Observation probability matrix
π	Initial state distribution
$\mathcal{O}$	Observation sequence, $\mathcal{O}_0, \mathcal{O}_1, \ldots, \mathcal{O}_{T-1}$

A generic hidden Markov model is illustrated in Figure 2.1, where the X_i represent the hidden states and all other notation is as in Table 2.1. The state of the Markov process, which we can view as being hidden behind a "curtain" (the dashed line in Figure 2.1), is determined by the current state and the A matrix. We are only able to observe the observations $\mathcal{O}_i$, which are related to the (hidden) states of the Markov process by the matrix B.

For the temperature example in the previous section, the observations sequence is given in (2.7), and we have $T = 4$, $N = 2$, $M = 3$, $Q = \{H, C\}$, and $V = \{0, 1, 2\}$. Note that we let $0, 1, 2$ represent small, medium, and large tree rings, respectively. For this example, the matrices A, B, and π are given by (2.4), (2.5), and (2.6), respectively.

In general, the matrix $A = \{a_{ij}\}$ is $N \times N$ with

$$a_{ij} = P(\text{state } q_j \text{ at } t+1 \,|\, \text{state } q_i \text{ at } t).$$

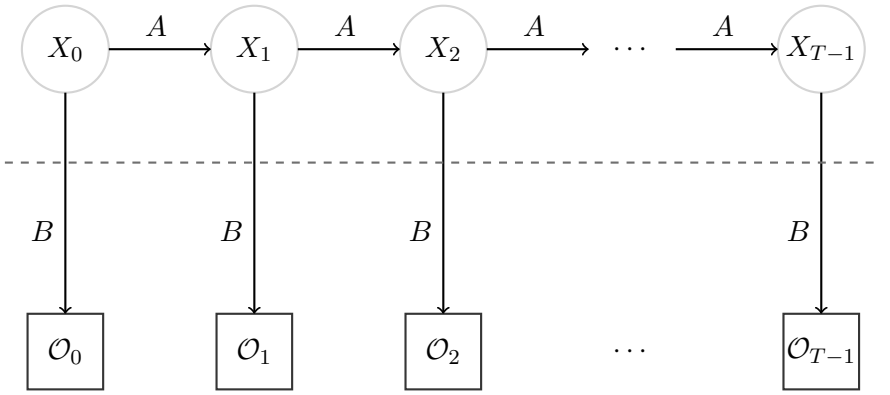


Figure 2.1: Hidden Markov model

The matrix A is always row stochastic. Also, the probabilities a_{ij} are independent of t, so that the A matrix does not change. The matrix $B = \{b_j(k)\}$ is of size $N \times M$, with

$$b_j(k) = P(\text{observation } k \text{ at } t \,|\, \text{state } q_j \text{ at } t).$$

As with the A matrix, B is row stochastic, and the probabilities $b_j(k)$ are independent of t. The somewhat unusual notation $b_j(k)$ is convenient when specifying the HMM algorithms.

An HMM is defined by A, B, and π (and, implicitly, by the dimensions N and M). Thus, we'll denote an HMM as $\lambda = (A, B, \pi)$.

Suppose that we are given an observation sequence of length four, which is denoted as

$$\mathcal{O} = \big(\mathcal{O}_0, \mathcal{O}_1, \mathcal{O}_2, \mathcal{O}_3\big).$$

The corresponding (hidden) state sequence is

$$X = \big(X_0, X_1, X_2, X_3\big).$$

We'll let π_{X_0} denote the probability of starting in state X_0, and $b_{X_0}(\mathcal{O}_0)$ denotes the probability of initially observing $\mathcal{O}_0$, while a_{X_0,X_1} is the probability of transiting from state X_0 to state X_1. Continuing, we see that the probability of a given state sequence X of length four is

$$P(X, \mathcal{O}) = \pi_{X_0} b_{X_0}(\mathcal{O}_0) a_{X_0,X_1} b_{X_1}(\mathcal{O}_1) a_{X_1,X_2} b_{X_2}(\mathcal{O}_2) a_{X_2,X_3} b_{X_3}(\mathcal{O}_3). \quad (2.8)$$

Note that in this expression, the X_i represent indices in the A and B matrices, not the names of the corresponding states.[3]

[3]Your kindly author regrets this abuse of notation.

Consider again the temperature example in Section 2.2, where the observation sequence is $\mathcal{O} = (0, 1, 0, 2)$. Using (2.8) we can compute, say,

$$P(HHCC) = 0.6(0.1)(0.7)(0.4)(0.3)(0.7)(0.6)(0.1) = 0.000212.$$

Similarly, we can directly compute the probability of each possible state sequence of length four, for the given observation sequence in (2.7). We have listed these results in Table 2.2, where the probabilities in the last column have been normalized so that they sum to 1.

Table 2.2: State sequence probabilities

State	Probability	Normalized probability
$HHHH$	0.000412	0.042787
$HHHC$	0.000035	0.003635
$HHCH$	0.000706	0.073320
$HHCC$	0.000212	0.022017
$HCHH$	0.000050	0.005193
$HCHC$	0.000004	0.000415
$HCCH$	0.000302	0.031364
$HCCC$	0.000091	0.009451
$CHHH$	0.001098	0.114031
$CHHC$	0.000094	0.009762
$CHCH$	0.001882	0.195451
$CHCC$	0.000564	0.058573
$CCHH$	0.000470	0.048811
$CCHC$	0.000040	0.004154
$CCCH$	0.002822	0.293073
$CCCC$	0.000847	0.087963

To find the optimal state sequence in the dynamic programming (DP) sense, we simply choose the sequence with the highest probability, which in this example is $CCCH$. To find the optimal state sequence in the HMM sense, we choose the most probable symbol at each position. To this end we sum the probabilities in Table 2.2 that have an H in the first position. Doing so, we find the (normalized) probability of H in the first position is 0.18817 and the probability of C in the first position is 0.81183. Therefore, the first element of the optimal sequence (in the HMM sense) is C. Repeating this for each element of the sequence, we obtain the probabilities in Table 2.3.

From Table 2.3, we find that the optimal sequence—in the HMM sense—is $CHCH$. Note that in this example, the optimal DP sequence differs from the optimal HMM sequence.

Table 2.3: HMM probabilities

	Position in state sequence			
	0	1	2	3
$P(H)$	0.188182	0.519576	0.228788	0.804029
$P(C)$	0.811818	0.480424	0.771212	0.195971

The Three Problems

There are three fundamental problems that we can solve using HMMs. Here, we briefly describe each of these problems, then in the next section we discuss efficient algorithms for their solution.

2.4.1 HMM Problem 1

Given the model $\lambda = (A, B, \pi)$ and a sequence of observations $\mathcal{O}$, determine $P(\mathcal{O} \mid \lambda)$. That is, we want to compute a score for the observed sequence $\mathcal{O}$ with respect to the given model λ.

2.4.2 HMM Problem 2

Given $\lambda = (A, B, \pi)$ and an observation sequence $\mathcal{O}$, find an optimal state sequence for the underlying Markov process. In other words, we want to uncover the hidden part of the hidden Markov model. This is the problem that was discussed in some detail above.

2.4.3 HMM Problem 3

Given an observation sequence $\mathcal{O}$ and the parameter N, determine a model of the form $\lambda = (A, B, \pi)$ that maximizes the probability of $\mathcal{O}$. This can be viewed as training a model to best fit the observed data. We'll solve this problem using a discrete hill climb on the parameter space represented by A, B, and π. Note that the dimension M is determined from the training sequence $\mathcal{O}$.

2.4.4 Discussion

Consider, for example, the problem of speech recognition—which happens to be one of the earliest and best-known applications of HMMs. We can use the solution to HMM Problem 3 to train an HMM λ to, for example, recognize the spoken word "yes." Then, given an unknown spoken word, we can use the solution to HMM Problem 1 to score this word against this

model λ and determine the likelihood that the word is "yes." In this case, we don't need to solve HMM Problem 2, but it is possible that such a solution—which uncovers the hidden states—might provide additional insight into the underlying speech model.

The Three Solutions

2.5.1 Solution to HMM Problem 1

Let $\lambda = (A, B, \pi)$ be a given HMM and let $\mathcal{O} = (O_0, \mathcal{O}_1, \ldots, \mathcal{O}_{T-1})$ be a series of observations. We want to find $P(\mathcal{O} \mid \lambda)$.

Let $X = (X_0, X_1, \ldots, X_{T-1})$ be a state sequence. Then by the definition of B we have

$$P(\mathcal{O} \mid X, \lambda) = b_{X_0}(\mathcal{O}_0) b_{X_1}(\mathcal{O}_1) \cdots b_{X_{T-1}}(\mathcal{O}_{T-1})$$

and by the definition of π and A it follows that

$$P(X \mid \lambda) = \pi_{X_0} a_{X_0,X_1} a_{X_1,X_2} \cdots a_{X_{T-2},X_{T-1}}.$$

Since

$$P(\mathcal{O}, X \mid \lambda) = \frac{P(\mathcal{O} \cap X \cap \lambda)}{P(\lambda)}$$

and

$$P(\mathcal{O} \mid X, \lambda) P(X \mid \lambda) = \frac{P(\mathcal{O} \cap X \cap \lambda)}{P(X \cap \lambda)} \cdot \frac{P(X \cap \lambda)}{P(\lambda)} = \frac{P(\mathcal{O} \cap X \cap \lambda)}{P(\lambda)}$$

we have

$$P(\mathcal{O}, X \mid \lambda) = P(\mathcal{O} \mid X, \lambda) P(X \mid \lambda).$$

Summing over all possible state sequences yields

$$\begin{aligned} P(\mathcal{O} \mid \lambda) &= \sum_X P(\mathcal{O}, X \mid \lambda) \\ &= \sum_X P(\mathcal{O} \mid X, \lambda) P(X \mid \lambda) \\ &= \sum_X \pi_{X_0} b_{X_0}(\mathcal{O}_0) a_{X_0,X_1} b_{X_1}(\mathcal{O}_1) \cdots a_{X_{T-2},X_{T-1}} b_{X_{T-1}}(\mathcal{O}_{T-1}). \end{aligned} \tag{2.9}$$

The direct computation in (2.9) is generally infeasible, since the number of multiplications is about $2TN^T$, where T is typically large and $N \geq 2$. One of the major strengths of HMMs is that there exists an efficient algorithm to achieve this same result.

To determine $P(\mathcal{O} \mid \lambda)$ in an efficient manner, we can use the following approach. For $t = 0, 1, \ldots, T-1$ and $i = 0, 1, \ldots, N-1$, define

$$\alpha_t(i) = P(\mathcal{O}_0, \mathcal{O}_1, \ldots, \mathcal{O}_t, X_t = q_i \mid \lambda). \tag{2.10}$$

Then $\alpha_t(i)$ is the probability of the partial observation sequence up to time t, where the underlying Markov process is in state q_i at time t.

The crucial insight is that the $\alpha_t(i)$ can be computed recursively—and efficiently. This recursive approach is known as the *forward algorithm*, or α-pass, and is given in Algorithm 2.1.

Algorithm 2.1 Forward algorithm

Given:
  Model $\lambda = (A, B, \pi)$
  Observations $\mathcal{O} = (O_0, \mathcal{O}_1, \ldots, \mathcal{O}_{T-1})$
for $i = 0, 1, \ldots, N-1$ **do**
  $\alpha_0(i) = \pi_i b_i(\mathcal{O}_0)$
end for
for $t = 1, 2, \ldots, T-1$ **do**
  for $i = 0, 1, \ldots, N-1$ **do**
    $$\alpha_t(i) = \left(\sum_{j=0}^{N-1} \alpha_{t-1}(j) a_{ji} \right) b_i(\mathcal{O}_t)$$
  end for
end for

The forward algorithm only requires about N^2T multiplications. This is in stark contrast to the naïve approach, which has a work factor of more than $2TN^T$. Since T is typically large and N is relatively small, the forward algorithm is highly efficient.

It follows from the definition in (2.10) that

$$P(\mathcal{O} \mid \lambda) = \sum_{i=0}^{N-1} \alpha_{T-1}(i).$$

Hence, the forward algorithm gives us an efficient way to compute a score for a given sequence $\mathcal{O}$, relative to a given model λ.

2.5.2 Solution to HMM Problem 2

Given the model $\lambda = (A, B, \pi)$ and a sequence of observations $\mathcal{O}$, our goal here is to find the most likely state sequence. As mentioned above, there are different possible interpretations of "most likely"—for an HMM, we maximize the expected number of correct states. In contrast, a dynamic program finds

the highest-scoring overall path. As we have seen, these solutions are not necessarily the same.

First, we define

$$\beta_t(i) = P(\mathcal{O}_{t+1}, \mathcal{O}_{t+2}, \ldots, \mathcal{O}_{T-1} \,|\, X_t = q_i, \lambda)$$

for $t = 0, 1, \ldots, T-1$, and $i = 0, 1, \ldots, N-1$. The $\beta_t(i)$ can be computed recursively (and efficiently) using the *backward algorithm*, or β*-pass*, which is given here in Algorithm 2.2. This is analogous to the α-pass discussed above, except that we start at the end and work back toward the beginning.

Algorithm 2.2 Backward algorithm

Given:
  Model $\lambda = (A, B, \pi)$
  Observations $\mathcal{O} = (O_0, \mathcal{O}_1, \ldots, \mathcal{O}_{T-1})$
for $i = 0, 1, \ldots, N-1$ **do**
  $\beta_{T-1}(i) = 1$
end for
for $t = T-2, T-3, \ldots, 0$ **do**
  for $i = 0, 1, \ldots, N-1$ **do**
    $$\beta_t(i) = \sum_{j=0}^{N-1} a_{ij} b_j(\mathcal{O}_{t+1}) \beta_{t+1}(j)$$
  end for
end for

Now, for $t = 0, 1, \ldots, T-1$ and $i = 0, 1, \ldots, N-1$, define

$$\gamma_t(i) = P(X_t = q_i \,|\, \mathcal{O}, \lambda).$$

Since $\alpha_t(i)$ measures the relevant probability up to time t and $\beta_t(i)$ measures the relevant probability after time t, we have

$$\gamma_t(i) = \frac{\alpha_t(i)\beta_t(i)}{P(\mathcal{O} \,|\, \lambda)}.$$

Recall that the denominator $P(\mathcal{O} \,|\, \lambda)$ is obtained by summing $\alpha_{T-1}(i)$ over i. From the definition of $\gamma_t(i)$ it follows that the most likely state at time t is the state q_i for which $\gamma_t(i)$ is maximum, where the maximum is taken over the index i. Then the most likely state at time t is given by

$$\widetilde{X}_t = \max_i \gamma_t(i).$$

2.5.3 Solution to HMM Problem 3

Here we want to adjust the model parameters to best fit the given observations. The sizes of the matrices (N and M) are known, while the elements

of A, B, and π are to be determined, subject to row stochastic conditions. The fact that we can efficiently re-estimate the model itself is perhaps the more impressive aspect of HMMs.

For $t = 0, 1, \ldots, T-2$ and $i, j \in \{0, 1, \ldots, N-1\}$, define the "di-gammas" as

$$\gamma_t(i, j) = P(X_t = q_i, X_{t+1} = q_j \,|\, \mathcal{O}, \lambda).$$

Then $\gamma_t(i, j)$ is the probability of being in state q_i at time t and transiting to state q_j at time $t + 1$. The di-gammas can be written in terms of α, β, A, and B as

$$\gamma_t(i, j) = \frac{\alpha_t(i) a_{ij} b_j(\mathcal{O}_{t+1}) \beta_{t+1}(j)}{P(\mathcal{O} \,|\, \lambda)}.$$

For $t = 0, 1, \ldots, T-2$, we see that $\gamma_t(i)$ and $\gamma_t(i, j)$ are related by

$$\gamma_t(i) = \sum_{j=0}^{N-1} \gamma_t(i, j).$$

Once the $\gamma_t(i, j)$ have been computed, the model $\lambda = (A, B, \pi)$ is re-estimated using Algorithm 2.3. The HMM training algorithm is known as Baum-Welch re-estimation, and is named after Leonard E. Baum and Lloyd R. Welch, who developed the technique in the late 1960s while working at the Center for Communications Research (CCR),[4] which is part of the Institute for Defense Analyses (IDA), located in Princeton, New Jersey.

The numerator of the re-estimated a_{ij} in Algorithm 2.3 can be seen to give the expected number of transitions from state q_i to state q_j, while the denominator is the expected number of transitions from q_i to any state.[5] Hence, the ratio is the probability of transiting from state q_i to state q_j, which is the desired value of a_{ij}.

The numerator of the re-estimated $b_j(k)$ in Algorithm 2.3 is the expected number of times the model is in state q_j with observation k, while the denominator is the expected number of times the model is in state q_j. Therefore, the ratio is the probability of observing symbol k, given that the model is in state q_j, and this is the desired value for $b_j(k)$.

Re-estimation is an iterative process. First, we initialize $\lambda = (A, B, \pi)$ with a reasonable guess, or, if no reasonable guess is available, we choose

[4]Not to be confused with Creedence Clearwater Revival [153].

[5]When re-estimating the A matrix, we are dealing with expectations. However, it might make things clearer to think in terms of frequency counts. For frequency counts, it would be easy to compute the probability of transitioning from state i to state j. That is, we would simply count the number of transitions from state i to state j, and divide this count by the total number of times we could be in state i. This is the intuition behind the re-estimation formula for the A matrix, and a similar statement holds when re-estimating the B matrix. In other words, don't let all of the fancy notation obscure the relatively simple ideas that are at the core of the re-estimation process.

Algorithm 2.3 Baum-Welch re-estimation

Given:
  $\gamma_t(i)$, for $t = 0, 1, \ldots, T-1$ and $i = 0, 1, \ldots, N-1$
  $\gamma_t(i, j)$, for $t = 0, 1, \ldots, T-2$ and $i, j \in \{0, 1, \ldots, N-1\}$
for $i = 0, 1, \ldots, N-1$ **do**
  $\pi_i = \gamma_0(i)$
end for
for $i = 0, 1, \ldots, N-1$ **do**
  for $j = 0, 1, \ldots, N-1$ **do**
    $$a_{ij} = \sum_{t=0}^{T-2} \gamma_t(i, j) \Big/ \sum_{t=0}^{T-2} \gamma_t(i)$$
  end for
end for
for $j = 0, 1, \ldots, N-1$ **do**
  for $k = 0, 1, \ldots, M-1$ **do**
    $$b_j(k) = \sum_{\substack{t \in \{0,1,\ldots,T-1\} \\ \mathcal{O}_t = k}} \gamma_t(j) \Big/ \sum_{t=0}^{T-1} \gamma_t(j)$$
  end for
end for

random values such that $\pi_i \approx 1/N$ and $a_{ij} \approx 1/N$ and $b_j(k) \approx 1/M$. It's critical that A, B, and π be randomized, since exactly uniform values will result in a local maximum from which the model cannot climb. And, as always, π, A and B must be row stochastic.

The complete solution to HMM Problem 3 can be summarized as follows.

1. Initialize, $\lambda = (A, B, \pi)$.
2. Compute $\alpha_t(i)$, $\beta_t(i)$, $\gamma_t(i, j)$ and $\gamma_t(i)$.
3. Re-estimate the model $\lambda = (A, B, \pi)$ using Algorithm 2.3.
4. If $P(\mathcal{O} \mid \lambda)$ increases, goto 2.

In practice, we would want to stop when $P(\mathcal{O} \mid \lambda)$ does not increase by some predetermined threshold, say, ε. We could also (or alternatively) set a maximum number of iterations. In any case, it's important to verify that the model has converged, which can usually be determined by perusing the B matrix.[6]

[6]While it might seem obvious to stop iterating when the change in $P(\mathcal{O} \mid \lambda)$ is small, this requires some care in practice. Typically, the change in $P(\mathcal{O} \mid \lambda)$ is very small over

Dynamic Programming

Before completing our discussion of the elementary aspects of HMMs, we make a brief detour to show the close relationship between dynamic programming (DP) and HMMs. The executive summary is that a DP can be viewed as an α-pass where "sum" is replaced by "max." More precisely, for π, A, and B as above, the dynamic programming algorithm, which is also known as the Viterbi algorithm, is given in Algorithm 2.4.

Algorithm 2.4 Dynamic programming

Given:
  Model $\lambda = (A, B, \pi)$
  Observations $\mathcal{O} = (O_0, \mathcal{O}_1, \ldots, \mathcal{O}_{T-1})$
for $i = 0, 1, \ldots, N-1$ **do**
  $\delta_0(i) = \pi_i b_i(\mathcal{O}_0)$
end for
for $t = 1, 2, \ldots, T-1$ **do**
  for $i = 0, 1, \ldots, N-1$ **do**
    $\delta_t(i) = \max_{j \in \{0,1,\ldots,N-1\}} \big(\delta_{t-1}(j) a_{ji} b_i(\mathcal{O}_t)\big)$
  end for
end for

At each successive t, a dynamic program determines the probability of the best path ending at each of the states $i = 0, 1, \ldots, N-1$. Consequently, the probability of the best overall path is

$$\max_{j \in \{0,1,\ldots,N-1\}} \delta_{T-1}(j). \tag{2.11}$$

It is important to realize that (2.11) only gives the optimal probability, not the corresponding path. By keeping track of each preceding state, the DP procedure given here can be augmented so that we can recover the optimal path by tracing back from the highest-scoring final state.

Consider again the example in Section 2.2. The initial probabilities are

$$P(H) = \pi_0 b_0(0) = 0.6(0.1) = 0.06 \text{ and } P(C) = \pi_1 b_1(0) = 0.4(0.7) = 0.28.$$

The probabilities of the paths of length two are given by

$$P(HH) = 0.06(0.7)(0.4) = 0.0168$$

the first several iterations. The model then goes through a period of rapid improvement—at which point the model has converged—after which the the change in $P(\mathcal{O} \mid \lambda)$ is again small. Consequently, if we simply set a threshold, the re-estimation process might stop immediately, or it might continue indefinitely. Perhaps the optimal approach is to combine a threshold with a minimum number of iterations—the pseudo-code in Section 2.8 uses this approach.

$$P(HC) = 0.06(0.3)(0.2) = 0.0036$$
$$P(CH) = 0.28(0.4)(0.4) = 0.0448$$
$$P(CC) = 0.28(0.6)(0.2) = 0.0336$$

and hence the best (most probable) path of length two ending with H is CH while the best path of length two ending with C is CC. Continuing, we construct the diagram in Figure 2.2 one level or stage at a time, where each arrow points to the next element in the optimal path ending at a given state. Note that at each stage, the dynamic programming algorithm only needs to maintain the highest-scoring path ending at each state—not a list of all possible paths. This is the key to the efficiency of the algorithm.

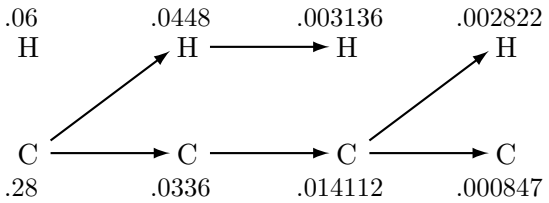


Figure 2.2: Dynamic programming

In Figure 2.2, the maximum final probability is 0.002822, which occurs at the final state H. We can use the arrows to trace back from H to find that the optimal path is $CCCH$. Note that this agrees with the brute force calculation in Table 2.2.

Underflow is a concern with a dynamic programming problem of this form—since we compute products of probabilities, the result will tend to 0. Fortunately, underflow is easily avoided by simply taking logarithms. An underflow-resistant version of DP is given in Algorithm 2.5.

Algorithm 2.5 Dynamic programming without underflow

Given:
  Model $\lambda = (A, B, \pi)$
  Observations $\mathcal{O} = (O_0, \mathcal{O}_1, \ldots, \mathcal{O}_{T-1})$
for $i = 0, 1, \ldots, N-1$ **do**
  $\widehat{\delta}_0(i) = \log\big(\pi_i b_i(\mathcal{O}_0)\big)$
end for
for $t = 1, 2, \ldots, T-1$ **do**
  for $i = 0, 1, \ldots, N-1$ **do**
    $\widehat{\delta}_t(i) = \max_{j \in \{0,1,\ldots,N-1\}} \Big(\widehat{\delta}_{t-1}(j) + \log(a_{ji}) + \log\big(b_i(\mathcal{O}_t)\big)\Big)$
  end for
end for

Not surprisingly, for the underflow-resistant version in Algorithm 2.5, the optimal score is given by

$$\max_{j\in\{0,1,\ldots,N-1\}} \widehat{\delta}_{T-1}(j).$$

Again, additional bookkeeping is required to determine the optimal path.

Scaling

The three HMM solutions in Section 2.5 all require computations involving products of probabilities. It's very easy to see, for example, that $\alpha_t(i)$ tends to 0 exponentially as T increases. Therefore, any attempt to implement the HMM algorithms as given in Section 2.5 will inevitably result in underflow. The solution to this underflow problem is to scale the numbers. However, care must be taken to ensure that the algorithms remain valid.

First, consider the computation of $\alpha_t(i)$. The basic recurrence is

$$\alpha_t(i) = \sum_{j=0}^{N-1} \alpha_{t-1}(j)a_{ji}b_i(\mathcal{O}_t).$$

It seems sensible to normalize each $\alpha_t(i)$ by dividing by

$$\sum_{j=0}^{N-1} \alpha_t(j).$$

Following this approach, we compute scaling factors c_t and the scaled $\alpha_t(i)$, which we denote as $\widehat{\alpha}_t(i)$, as in Algorithm 2.6.

To verify Algorithm 2.6 we first note that $\widehat{\alpha}_0(i) = c_0\alpha_0(i)$. Now suppose that for some t, we have

$$\widehat{\alpha}_t(i) = c_0c_1\cdots c_t\alpha_t(i). \tag{2.12}$$

Then

$$\begin{aligned}
\widehat{\alpha}_{t+1}(i) &= c_{t+1}\widetilde{\alpha}_{t+1}(i) \\
&= c_{t+1}\sum_{j=0}^{N-1} \widehat{\alpha}_t(j)a_{ji}b_i(\mathcal{O}_{t+1}) \\
&= c_0c_1\cdots c_tc_{t+1}\sum_{j=0}^{N-1} \alpha_t(j)a_{ji}b_i(\mathcal{O}_{t+1}) \\
&= c_0c_1\cdots c_{t+1}\alpha_{t+1}(i)
\end{aligned}$$

and hence (2.12) holds, by induction, for all t.

Algorithm 2.6 Scaling factors

Given:
  $\alpha_t(i)$, for $t = 0, 1, \ldots, T-1$ and $i = 0, 1, \ldots, N-1$
for $i = 0, 1, \ldots, N-1$ **do**
  $\widetilde{\alpha}_0(i) = \alpha_0(i)$
end for
$c_0 = 1/\sum_{j=0}^{N-1} \widetilde{\alpha}_0(j)$
for $i = 0, 1, \ldots, N-1$ **do**
  $\widehat{\alpha}_0(i) = c_0 \widetilde{\alpha}_0(i)$
end for
for $t = 1, 2, \ldots, T-1$ **do**
  for $i = 0, 1, \ldots, N-1$ **do**
    $\widetilde{\alpha}_t(i) = \sum_{j=0}^{N-1} \widehat{\alpha}_{t-1}(j) a_{ji} b_i(\mathcal{O}_t)$
  end for
  $c_t = 1 \Big/ \sum_{j=0}^{N-1} \widetilde{\alpha}_t(j)$
  for $i = 0, 1, \ldots, N-1$ **do**
    $\widehat{\alpha}_t(i) = c_t \widetilde{\alpha}_t(i)$
  end for
end for

From (2.12) and the definitions of $\widetilde{\alpha}$ and $\widehat{\alpha}$ it follows that

$$\widehat{\alpha}_t(i) = \alpha_t(i) \Big/ \sum_{j=0}^{N-1} \alpha_t(j). \tag{2.13}$$

From equation (2.13) we see that for all t and i, the desired scaled value of $\alpha_t(i)$ is indeed given by $\widehat{\alpha}_t(i)$.

From (2.13) it follows that

$$\sum_{j=0}^{N-1} \widehat{\alpha}_{T-1}(j) = 1.$$

Also, from (2.12) we have

$$\begin{aligned} \sum_{j=0}^{N-1} \widehat{\alpha}_{T-1}(j) &= c_0 c_1 \cdots c_{T-1} \sum_{j=0}^{N-1} \alpha_{T-1}(j) \\ &= c_0 c_1 \cdots c_{T-1} P(\mathcal{O} \,|\, \lambda). \end{aligned}$$

Combining these results gives us

$$P(\mathcal{O} \,|\, \lambda) = 1 \Big/ \prod_{j=0}^{T-1} c_j.$$

It follows that we can compute the log of $P(\mathcal{O} \,|\, \lambda)$ directly from the scaling factors c_t as

$$\log\big(P(\mathcal{O} \,|\, \lambda)\big) = -\sum_{j=0}^{T-1} \log c_j. \tag{2.14}$$

It is fairly easy to show that the same scale factors c_t can be used in the backward algorithm by simply computing $\widehat{\beta}_t(i) = c_t \beta_t(i)$. We then determine $\gamma_t(i,j)$ and $\gamma_t(i)$ using the same formulae as in Section 2.5, but with $\widehat{\alpha}_t(i)$ and $\widehat{\beta}_t(i)$ in place of $\alpha_t(i)$ and $\beta_t(i)$, respectively. The resulting gammas and di-gammas are then used to re-estimate π, A, and B.

By writing the original re-estimation formulae (as given in lines 3, 7, and 12 of Algorithm 2.3) directly in terms of $\alpha_t(i)$ and $\beta_t(i)$, it is a straightforward exercise to show that the re-estimated π and A and B are exact when $\widehat{\alpha}_t(i)$ and $\widehat{\beta}_t(i)$ are used in place of $\alpha_t(i)$ and $\beta_t(i)$. Furthermore, $P(\mathcal{O} \,|\, \lambda)$ isn't required in the re-estimation formulae, since in each case it cancels in the numerator and denominator. Therefore, (2.14) determines a score for the model, which can be used, for example, to decide whether the model is improving sufficiently to continue to the next iteration of the training algorithm.

All Together Now

Here, we give complete pseudo-code for solving HMM Problem 3, including scaling. This pseudo-code also provides virtually everything needed to solve HMM Problems 1 and 2.

1. Given

   Observation sequence $\mathcal{O} = (\mathcal{O}_0, \mathcal{O}_1, \ldots, \mathcal{O}_{T-1})$.

2. Initialize

   (a) Select N and determine M from $\mathcal{O}$. Recall that the model is denoted $\lambda = (A, B, \pi)$, where $A = \{a_{ij}\}$ is $N \times N$, $B = \{b_j(k)\}$ is $N \times M$, and $\pi = \{\pi_i\}$ is $1 \times N$.

   (b) Initialize the three matrices A, B, and π. You can use knowledge of the problem when generating initial values, but if no such

information is available (as is often the case), let $\pi_i \approx 1/N$ and let $a_{ij} \approx 1/N$ and $b_j(k) \approx 1/M$. Always be sure that your initial values satisfy the row stochastic conditions (i.e., the elements of each row sum to 1, and each element is between 0 and 1). Also, make sure that the elements of each row are *not* exactly uniform.

(c) Initialize each of the following.

`minIters` = minimum number of re-estimation iterations
ε = threshold representing negligible improvement in model
`iters` $= 0$
`oldLogProb` $= -\infty$

3. Forward algorithm or α-pass

```
// compute α0(i)
c0 = 0
for i = 0 to N − 1
    α0(i) = πi bi(O0)
    c0 = c0 + α0(i)
next i
// scale the α0(i)
c0 = 1/c0
for i = 0 to N − 1
    α0(i) = c0 α0(i)
next i
// compute αt(i)
for t = 1 to T − 1
    ct = 0
    for i = 0 to N − 1
        αt(i) = 0
        for j = 0 to N − 1
            αt(i) = αt(i) + αt−1(j) aji
        next j
        αt(i) = αt(i) bi(Ot)
        ct = ct + αt(i)
    next i
    // scale αt(i)
    ct = 1/ct
    for i = 0 to N − 1
        αt(i) = ct αt(i)
    next i
next t
```

4. Backward algorithm or β-pass

// Let $\beta_{T-1}(i) = 1$ scaled by c_{T-1}
for $i = 0$ to $N - 1$
    $\beta_{T-1}(i) = c_{T-1}$
next i
// β-pass
for $t = T - 2$ to 0 by -1
    for $i = 0$ to $N - 1$
        $\beta_t(i) = 0$
        for $j = 0$ to $N - 1$
            $\beta_t(i) = \beta_t(i) + a_{ij} b_j(\mathcal{O}_{t+1}) \beta_{t+1}(j)$
        next j
        // scale $\beta_t(i)$ with same scale factor as $\alpha_t(i)$
        $\beta_t(i) = c_t \beta_t(i)$
    next i
next t

5. Compute the gammas and di-gammas

for $t = 0$ to $T - 2$
    $\texttt{denom} = 0$
    for $i = 0$ to $N - 1$
        for $j = 0$ to $N - 1$
            $\texttt{denom} = \texttt{denom} + \alpha_t(i) a_{ij} b_j(\mathcal{O}_{t+1}) \beta_{t+1}(j)$
        next j
    next i
    for $i = 0$ to $N - 1$
        $\gamma_t(i) = 0$
        for $j = 0$ to $N - 1$
            $\gamma_t(i, j) = \big(\alpha_t(i) a_{ij} b_j(\mathcal{O}_{t+1}) \beta_{t+1}(j)\big) / \texttt{denom}$
            $\gamma_t(i) = \gamma_t(i) + \gamma_t(i, j)$
        next j
    next i
next t
// Special case for $\gamma_{T-1}(i)$
$\texttt{denom} = 0$
for $i = 0$ to $N - 1$
    $\texttt{denom} = \texttt{denom} + \alpha_{T-1}(i)$
next i
for $i = 0$ to $N - 1$
    $\gamma_{T-1}(i) = \alpha_{T-1}(i) / \texttt{denom}$
next i

6. Re-estimate the model $\lambda = (A, B, \pi)$

```
// re-estimate π
for i = 0 to N − 1
    π_i = γ_0(i)
next i
// re-estimate A
for i = 0 to N − 1
    for j = 0 to N − 1
        numer = 0
        denom = 0
        for t = 0 to T − 2
            numer = numer + γ_t(i, j)
            denom = denom + γ_t(i)
        next t
        a_ij = numer/denom
    next j
next i
// re-estimate B
for i = 0 to N − 1
    for j = 0 to M − 1
        numer = 0
        denom = 0
        for t = 0 to T − 1
            if(O_t == j) then
                numer = numer + γ_t(i)
            end if
            denom = denom + γ_t(i)
        next t
        b_i(j) = numer/denom
    next j
next i
```

7. Compute $\log\big(P(\mathcal{O} \mid \lambda)\big)$

```
logProb = 0
for i = 0 to T − 1
    logProb = logProb + log(c_i)
next i
logProb = −logProb
```

8. To iterate or not to iterate, that is the question.

```
iters = iters + 1
δ = |logProb − oldLogProb|
if(iters < minIters or δ > ε) then
    oldLogProb = logProb
    goto 3.
else
    return λ = (A, B, π)
end if
```

The Bottom Line

Hidden Markov models are powerful, efficient, and extremely useful in practice. Virtually no assumptions need to be made, yet the HMM process can extract significant statistical information from data. Thanks to efficient training and scoring algorithms, HMMs are practical, and they have proven useful in a wide range of applications. Even in cases where the underlying assumption of a (hidden) Markov process is questionable, HMMs are often applied with success. In Chapter 9 we consider selected applications of HMMs. Most of these applications are in the field of information security.

In subsequent chapters, we often compare and contrast other machine learning techniques to HMMs. Consequently, a clear understanding of the material in this chapter is crucial before proceeding with the remainder of the book. The homework problem should help the dedicated reader to clarify any remaining issues. And the applications in Chapter 9 are highly recommended, with the English text example in Section 9.2 being especially highly recommended.

Problems

When faced with a problem you do not understand,
do any part of it you do understand, then look at it again.
— Robert Heinlein

1. Suppose that we train an HMM and obtain the model $\lambda = (A, B, \pi)$ where

$$A = \begin{pmatrix} 0.7 & 0.3 \\ 0.4 & 0.6 \end{pmatrix}, \quad B = \begin{pmatrix} 0.1 & 0.4 & 0.5 \\ 0.7 & 0.2 & 0.1 \end{pmatrix}, \quad \pi = \begin{pmatrix} 0.0 & 1.0 \end{pmatrix}.$$

Furthermore, suppose the hidden states correspond to H and C, respectively, while the observations are S, M, and L, which are mapped to 0, 1, and 2, respectively. In this problem, we consider the observation sequence $\mathcal{O} = (\mathcal{O}_0, \mathcal{O}_1, \mathcal{O}_2) = (M, S, L) = (1, 0, 2)$.

a) Directly compute $P(\mathcal{O} \mid \lambda)$. That is, compute

$$P(\mathcal{O} \mid \lambda) = \sum_X P(\mathcal{O}, X \mid \lambda)$$

using the probabilities in $\lambda = (A, B, \pi)$ for each of the following cases, based on the given observation sequence $\mathcal{O}$.

$$P(\mathcal{O}, X = HHH) = ___ \cdot ___ \cdot ___ \cdot ___ \cdot ___ \cdot ___ = _____$$
$$P(\mathcal{O}, X = HHC) = ___ \cdot ___ \cdot ___ \cdot ___ \cdot ___ \cdot ___ = _____$$
$$P(\mathcal{O}, X = HCH) = ___ \cdot ___ \cdot ___ \cdot ___ \cdot ___ \cdot ___ = _____$$
$$P(\mathcal{O}, X = HCC) = ___ \cdot ___ \cdot ___ \cdot ___ \cdot ___ \cdot ___ = _____$$
$$P(\mathcal{O}, X = CHH) = ___ \cdot ___ \cdot ___ \cdot ___ \cdot ___ \cdot ___ = _____$$
$$P(\mathcal{O}, X = CHC) = ___ \cdot ___ \cdot ___ \cdot ___ \cdot ___ \cdot ___ = _____$$
$$P(\mathcal{O}, X = CCH) = 1.0 \cdot 0.2 \cdot 0.6 \cdot 0.7 \cdot 0.4 \cdot 0.5 = _____$$
$$P(\mathcal{O}, X = CCC) = ___ \cdot ___ \cdot ___ \cdot ___ \cdot ___ \cdot ___ = _____$$

The desired probability is the sum of these eight probabilities.

b) Compute $P(\mathcal{O} \mid \lambda)$ using the α pass. That is, compute

$$\alpha_0(0) = ___ \cdot ___ = _____$$
$$\alpha_0(1) = 1.0 \cdot 0.2 = _____$$
$$\alpha_1(0) = (___ \cdot ___ + ___ \cdot ___) \cdot ___ = _____$$
$$\alpha_1(1) = (___ \cdot ___ + ___ \cdot ___) \cdot ___ = _____$$
$$\alpha_2(0) = (___ \cdot ___ + ___ \cdot ___) \cdot ___ = _____$$
$$\alpha_2(1) = (___ \cdot ___ + ___ \cdot ___) \cdot ___ = _____$$

where we initialize

$$\alpha_0(i) = \pi_i b_i(\mathcal{O}_0), \text{ for } i = 0, 1, \ldots, N-1$$

and the recurrence is

$$\alpha_t(i) = \left(\sum_{j=0}^{N-1} \alpha_{t-1}(j) a_{ji} \right) b_i(\mathcal{O}_t)$$

for $t = 1, 2, \ldots, T-1$ and $i = 0, 1, \ldots, N-1$. The desired probability is given by

$$P(\mathcal{O} \mid \lambda) = \sum_{i=0}^{N-1} \alpha_{T-1}(i).$$

c) In terms of N and T, and counting only multiplications, what is the work factor for the method in part a)? What is the work factor for the method in part b)?

2. For this problem, use the same model λ and observation sequence $\mathcal{O}$ given in Problem 1.

   a) Determine the best hidden state sequence (X_0, X_1, X_2) in the dynamic programming sense.

   b) Determine the best hidden state sequence (X_0, X_1, X_2) in the HMM sense.

3. Summing the numbers in the "probability" column of Table 2.2, we find $P(\mathcal{O} \mid \lambda) = 0.009629$ for $\mathcal{O} = (0, 1, 0, 2)$.

   a) By a similar direct calculation, compute $P(\mathcal{O} \mid \lambda)$ for each observation sequence of the form $\mathcal{O} = (\mathcal{O}_0, \mathcal{O}_1, \mathcal{O}_2, \mathcal{O}_3)$, where $\mathcal{O}_i \in \{0, 1, 2\}$. Verify that $\sum P(\mathcal{O} \mid \lambda) = 1$, where the sum is over the observation sequences of length four. Note that you will need to use the probabilities for A, B, and π given in equations (2.4), (2.5), and (2.6) in Section 2.2, respectively.

   b) Use the forward algorithm to compute $P(\mathcal{O} \mid \lambda)$ for the same observation sequences and model as in part a). Verify that you obtain the same results as in part a).

4. From equation (2.9) and the definition of $\alpha_t(i)$ in equation (2.10), it follows that

$$\alpha_t(i) = \sum_X \pi_{X_0} b_{X_0}(\mathcal{O}_0) a_{X_0,X_1} b_{X_1}(\mathcal{O}_1) \cdots a_{X_{t-2},X_{t-1}} b_{X_{t-1}}(\mathcal{O}_{t-1}) a_{X_{t-1},i} b_i(\mathcal{O}_t)$$

   where $X = (X_0, X_1, \ldots, X_{t-1})$. Use this expression for $\alpha_t(i)$ to directly verify the forward algorithm recurrence

$$\alpha_t(i) = \left(\sum_{j=0}^{N-1} \alpha_{t-1}(j) a_{ji} \right) b_i(\mathcal{O}_t).$$

5. As discussed in this chapter, the forward algorithm is used solve HMM Problem 1, while the forward algorithm and backward algorithm together are used to compute the gammas, which are then used to solve HMM Problem 2.

   a) Explain how you can solve HMM Problem 1 using the backward algorithm instead of the forward algorithm.

b) Using the model $\lambda = (A, B, \pi)$ and the observation sequence $\mathcal{O}$ in Problem 1, compute $P(\mathcal{O} \mid \lambda)$ using the backward algorithm, and verify that you obtain the same result as when using the forward algorithm.

6. This problem deals with the Baum-Welch re-estimation algorithm.

   a) Write the re-estimation formulae, as given in lines 3, 7, and 12 of Algorithm 2.3, directly in terms of the $\alpha_t(i)$ and $\beta_t(i)$.

   b) Using the re-estimation formulae obtained in part a), substitute the scaled values $\widehat{\alpha}_t(i)$ and $\widehat{\beta}_t(i)$ for $\alpha_t(i)$ and $\beta_t(i)$, respectively, and show that the resulting re-estimation formulae are exact.

7. Instead of using c_t to scale the $\beta_t(i)$, we can scale each $\beta_t(i)$ by

$$d_t = 1 \Big/ \sum_{j=0}^{N-1} \widetilde{\beta}_t(j)$$

   where the definition of $\widetilde{\beta}_t(i)$ is analogous to that of $\widetilde{\alpha}_t(i)$ as given in Algorithm 2.6.

   a) Using the scaling factors c_t and d_t show that the Baum-Welch re-estimation formulae in Algorithm 2.3 are exact with $\widehat{\alpha}$ and $\widehat{\beta}$ in place of α and β.

   b) Write $\log\big(P(\mathcal{O} \mid \lambda)\big)$ in terms of c_t and d_t.

8. When training, the elements of λ can be initialized to approximately uniform. That is, we let $\pi_i \approx 1/N$ and $a_{ij} \approx 1/N$ and $b_j(k) \approx 1/M$, subject to the row stochastic conditions. In Section 2.5.3, it is stated that it is a bad idea to initialize the values to exactly uniform, since the HMM would be stuck at a local maximum and hence it could not climb to an improved solution. Suppose that $\pi_i = 1/N$ and $a_{ij} = 1/N$ and $b_j(k) = 1/M$. Verify that the re-estimation process leaves all of these values unchanged.

9. In this problem, we consider generalizations of the HMM formulation discussed in this chapter.

   a) Consider an HMM where the state transition matrix is time dependent. Then for each t, there is an $N \times N$ row-stochastic $A_t = \{a_{ij}^t\}$ that is used in place of A in the HMM computations. For such an HMM, provide pseudo-code to solve HMM Problem 1.

   b) Consider an HMM of order two, that is, an HMM where the underlying Markov process is of order two. Then the state at time t depends on the states at time $t-1$ and $t-2$. For such an HMM, provide pseudo-code to solve HMM Problem 1.

10. Write an HMM program for the English text problem in Section 9.2 of Chapter 9. Test your program on each of the following cases.

    a) There are $N = 2$ hidden states. Explain your results.

    b) There are $N = 3$ hidden states. Explain your results.

    c) There are $N = 4$ hidden states. Explain your results.

    d) There are $N = 26$ hidden states. Explain your results.

11. In this problem, you will use an HMM to break a simple substitution ciphertext message. For each HMM, train using 200 iterations of the Baum-Welch re-estimation algorithm.

    a) Obtain an English plaintext message of 50,000 plaintext characters, where the characters consist only of lower case `a` through `z` (i.e., remove all punctuation, special characters, and spaces, and convert all upper case to lower case). Encrypt this plaintext using a randomly generated shift of the alphabet. Remember the key.

    b) Train an HMM with $N = 2$ and $M = 26$ on your ciphetext from part a). From the final B matrix, determine the ciphertext letters that correspond to consonants and vowels.

    c) Generate a digraph frequency matrix A for English text, where a_{ij} is the count of the number of times that letter i is followed by letter j. Here, we assume that `a` is letter 0, `b` is letter 1, `c` is letter 2, and so on. This matrix must be based on 1,000,000 characters where, as above, only the 26 letters of the alphabet are used. Next, add five to each element in your 26×26 matrix A. Finally, normalize your matrix A by dividing each element by its row sum. The resulting matrix A will be row stochastic, and it will not contain any 0 probabilities.

    d) Train an HMM with $N = M = 26$, using the first 1000 characters of ciphertext you generated in part a), where the A matrix is initialized with your A matrix from part c). Also, in your HMM, do not re-estimate A. Use the final B matrix to determine a putative key and give the fraction of putative key elements that match the actual key (as a decimal, to four places). For example, if 22 of the 26 key positions are correct, then your answer would be $22/26 = 0.8462$.

12. Write an HMM program to solve the problem discussed in Section 9.2, replacing English text with the following.

    a) French text.

    b) Russian text.

    c) Chinese text.

13. Perform an HMM analysis similar to that discussed in Section 9.2, replacing English with "Hamptonese," the mysterious writing system developed by James Hampton. For information on Hamptonese, see

    `http://www.cs.sjsu.edu/faculty/stamp/Hampton/hampton.html`

14. Since HMM training is a hill climb, we are only assured of reaching a local maximum. And, as with any hill climb, the specific local maximum that we find will depend on our choice of initial values. Therefore, by training a hidden Markov model multiple times with different initial values, we would expect to obtain better results than when training only once.

    In the paper [16], the authors use an expectation maximization (EM) approach with multiple random restarts as a means of attacking homophonic substitution ciphers. An analogous HMM-based technique is analyzed in the report [158], where the effectiveness of multiple random restarts on simple substitution cryptanalysis is explored in detail. Multiple random restarts are especially helpful in the most challenging cases, that is, when little data (i.e., ciphertext) is available. However, the tradeoff is that the work factor can be high, since the number of restarts required may be very large (millions of random restarts are required in some cases).

    a) Obtain an English plaintext message consisting of 1000 plaintext characters, consisting only of lower case `a` through `z` (i.e., remove all punctuation, special characters, and spaces, and convert all upper case letters to lower case). Encrypt this plaintext using a randomly selected shift of the alphabet. Remember the key. Also generate a digraph frequency matrix A, as discussed in part c) of Problem 11.

    b) Train n HMMs, for each of $n = 1$, $n = 10$, $n = 100$, and $n = 1000$, following the same process as in Problem 11, part d), but using the $T = 1000$ observations generated in part a) of this problem. For a given n select the best result based on the model scores and give the fraction of the putative key that is correct, calculated as in Problem 11, part d).

    c) Repeat part b), but only use the first $T = 400$ observations.

    d) Repeat part c), but only use the first $T = 300$ observations.

15. The Zodiac Killer murdered at least five people in the San Francisco Bay Area in the late 1960s and early 1970s. Although police had a prime suspect, no arrest was ever made and the murders remain officially unsolved. The killer sent several messages to the police and to local newspapers, taunting police for their failure to catch him. One of these

messages contained a homophonic substitution consisting of 408 strange symbols.[7] Not surprisingly, this cipher is known as the Zodiac 408. Within days of its release, the Zodiac 408 was broken by Donald and Bettye Harden, who were schoolteachers from Salinas, California. The Zodiac 408 ciphertext is given below on the left, while the corresponding plaintext appears on the right.

I L I K E K I L L I N G P E O P L
E B E C A U S E I T I S S O M U C
H F U N I T I S M O R E F U N T H
A N K I L L I N G W I L D G A M E
I N T H E F O R R E S T B E C A U
S E M A N I S T H E M O S T D A N
G E R O U E A N A M A L O F A L L
T O K I L L S O M E T H I N G G I
V E S M E T H E M O S T T H R I L
L I N G E X P E R E N C E I T I S
E V E N B E T T E R T H A N G E T
T I N G Y O U R R O C K S O F F W
I T H A G I R L T H E B E S T P A
R T O F I T I S T H A E W H E N I
D I E I W I L L B E R E B O R N I
N P A R A D I C E A N D A L L T H
E I H A V E K I L L E D W I L L B
E C O M E M Y S L A V E S I W I L
L N O T G I V E Y O U M Y N A M E
B E C A U S E Y O U W I L L T R Y
T O S L O I D O W N O R A T O P M
Y C O L L E C T I O G O F S L A V
E S F O R M Y A F T E R L I F E E
B E O R I E T E M E T H H P I T I

Note the (apparently intentional) misspellings in the plaintext, including "FORREST", "ANAMAL", and so on. Also, the final 18 characters (underlined in the plaintext above) appear to be random filler.

a) Solve the Zodiac 408 cipher using the HMM approach discussed in Section 9.4. Initialize the A matrix as in part c) of Problem 11, and do not re-estimate A. Use 1000 random restarts of the HMM, and 200 iterations of Baum-Welch re-estimation in each case. Give your answer as the percentage of characters of the actual plaintext that are recovered correctly.

b) Repeat part a), but use 10,000 random restarts.

c) Repeat part b), but use 100,000 random restarts.

d) Repeat part c), but use 1,000,000 random restarts.

[7]The Zodiac 408 ciphertext was actually sent in three parts to local newspapers. Here, we give the complete message, where the three parts have been combined into one. Also, a homophonic substitution is like a simple substitution, except that the mapping is many-to-one, that is, multiple ciphertext symbols can map to one plaintext symbol.

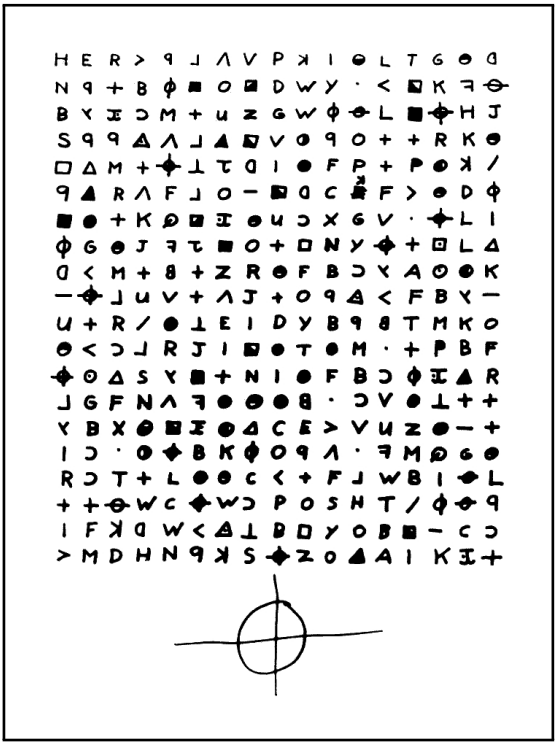
e) Repeat part a), except also re-estimate the A matrix.

f) Repeat part b), except also re-estimate the A matrix.

g) Repeat part c), except also re-estimate the A matrix.

h) Repeat part d), except also re-estimate the A matrix.

16. In addition to the Zodiac 408 cipher, the Zodiac Killer (see Problem 15) released a similar-looking cipher with 340 symbols. This cipher is known as the Zodiac 340 and remains unsolved to this day.[8] The ciphertext is given below.



a) Repeat Problem 15, parts a) through d), using the Zodiac 340 in place of the Zodiac 408. Since the plaintext is unknown, in each case, simply print the decryption obtained from your highest scoring model.

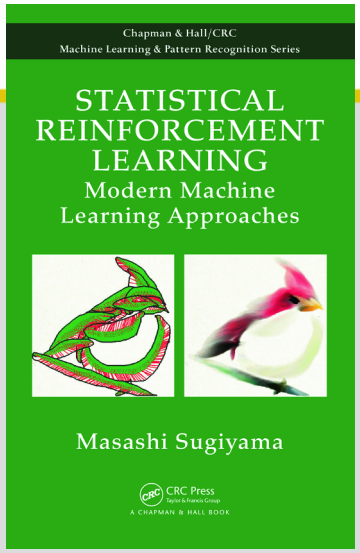
b) Repeat part a) of this problem, except use parts e) through h) of Problem 15.

[8]It is possible that the Zodiac 340 is not a cipher at all, but instead just a random collection of symbols designed to frustrate would-be cryptanalysts. If that's the case, your easily frustrated author can confirm that the "cipher" has been wildly successful.

CHAPTER

4

INTRODUCTION TO REINFORCEMENT LEARNING



This chapter is excerpted from

Statistical Reinforcement Learning

by Masashi Sugiyama.

© 2018 Taylor & Francis Group. All rights reserved.

Learn more

Introduction to Reinforcement Learning

Reinforcement learning is aimed at controlling a computer agent so that a target task is achieved in an unknown environment.

In this chapter, we first give an informal overview of reinforcement learning in Section 1.1. Then we provide a more formal formulation of reinforcement learning in Section 1.2. Finally, the book is summarized in Section 1.3.

1.1 Reinforcement Learning

A schematic of reinforcement learning is given in Figure 1.1. In an unknown environment (e.g., in a maze), a computer agent (e.g., a robot) takes an action (e.g., to walk) based on its own control policy. Then its state is updated (e.g., by moving forward) and evaluation of that action is given as a "reward" (e.g., praise, neutral, or scolding). Through such interaction with the environment, the agent is trained to achieve a certain task (e.g., getting out of the maze) without explicit guidance. A crucial advantage of reinforcement learning is its non-greedy nature. That is, the agent is trained not to improve performance in a short term (e.g., greedily approaching an exit of the maze), but to optimize the long-term achievement (e.g., successfully getting out of the maze).

A reinforcement learning problem contains various technical components such as states, actions, transitions, rewards, policies, and values. Before going into mathematical details (which will be provided in Section 1.2), we intuitively explain these concepts through illustrative reinforcement learning problems here.

Let us consider a *maze problem* (Figure 1.2), where a robot agent is located in a maze and we want to guide him to the goal without explicit supervision about which direction to go. *States* are positions in the maze which the robot agent can visit. In the example illustrated in Figure 1.3, there are 21 states in the maze. *Actions* are possible directions along which the robot agent can move. In the example illustrated in Figure 1.4, there are 4 actions which correspond to movement toward the north, south, east, and west directions. States

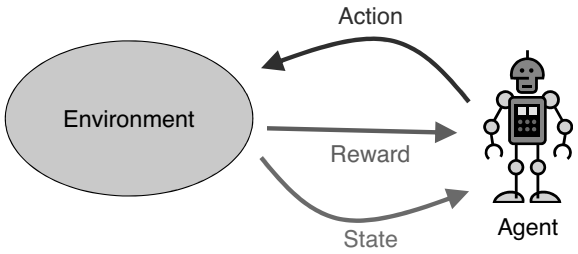


FIGURE 1.1: Reinforcement learning.

and actions are fundamental elements that define a reinforcement learning problem.

Transitions specify how states are connected to each other through actions (Figure 1.5). Thus, knowing the transitions intuitively means knowing the map of the maze. *Rewards* specify the incomes/costs that the robot agent receives when making a transition from one state to another by a certain action. In the case of the maze example, the robot agent receives a positive reward when it reaches the goal. More specifically, a positive reward is provided when making a transition from state 12 to state 17 by action "east" or from state 18 to state 17 by action "north" (Figure 1.6). Thus, knowing the rewards intuitively means knowing the location of the goal state. To emphasize the fact that a reward is given to the robot agent right after taking an action and making a transition to the next state, it is also referred to as an *immediate reward*.

Under the above setup, the goal of reinforcement learning to find the *policy* for controlling the robot agent that allows it to receive the maximum amount of rewards in the long run. Here, a policy specifies an action the robot agent takes at each state (Figure 1.7). Through a policy, a series of states and actions that the robot agent takes from a start state to an end state is specified. Such a series is called a *trajectory* (see Figure 1.7 again). The sum of immediate rewards along a trajectory is called the *return*. In practice, rewards that can be obtained in the distant future are often discounted because receiving rewards earlier is regarded as more preferable. In the maze task, such a discounting strategy urges the robot agent to reach the goal as quickly as possible.

To find the optimal policy efficiently, it is useful to view the return as a function of the initial state. This is called the (state-)*value*. The values can be efficiently obtained via *dynamic programming*, which is a general method for solving a complex optimization problem by breaking it down into simpler subproblems recursively. With the hope that many subproblems are actually the same, dynamic programming solves such overlapped subproblems only once and reuses the solutions to reduce the computation costs.

In the maze problem, the value of a state can be computed from the values of neighboring states. For example, let us compute the value of state 7 (see

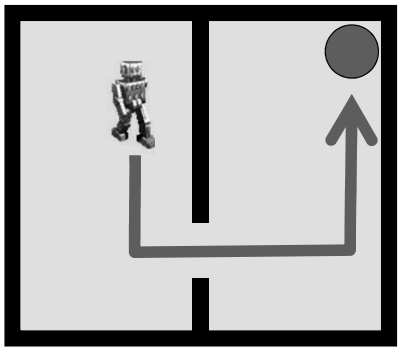


FIGURE 1.2: A maze problem. We want to guide the robot agent to the goal.

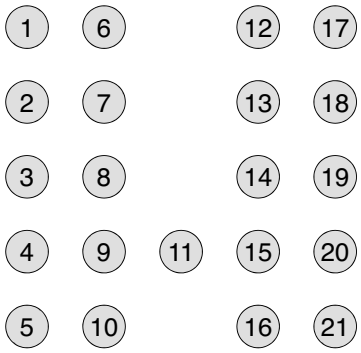


FIGURE 1.3: States are visitable positions in the maze.

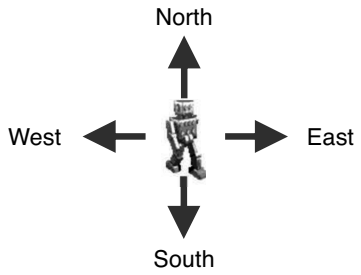


FIGURE 1.4: Actions are possible movements of the robot agent.

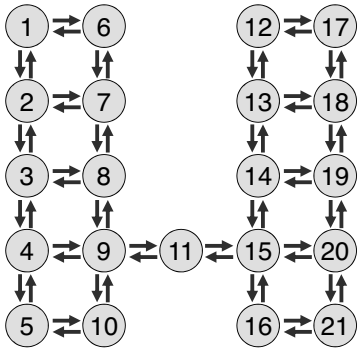


FIGURE 1.5: Transitions specify connections between states via actions. Thus, knowing the transitions means knowing the map of the maze.

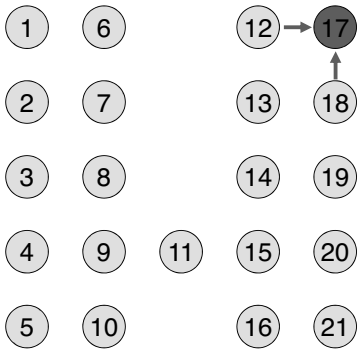


FIGURE 1.6: A positive reward is given when the robot agent reaches the goal. Thus, the reward specifies the goal location.

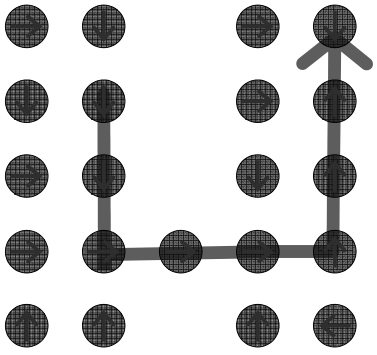


FIGURE 1.7: A policy specifies an action the robot agent takes at each state. Thus, a policy also specifies a trajectory, which is a series of states and actions that the robot agent takes from a start state to an end state.

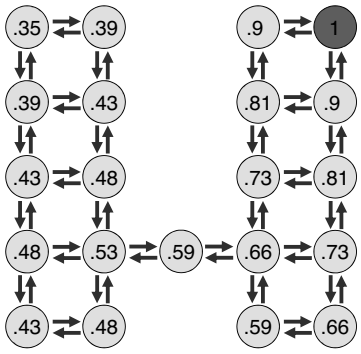


FIGURE 1.8: Values of each state when reward +1 is given at the goal state and the reward is discounted at the rate of 0.9 according to the number of steps.

Figure 1.5 again). From state 7, the robot agent can reach state 2, state 6, and state 8 by a single step. If the robot agent knows the values of these neighboring states, the best action the robot agent should take is to visit the neighboring state with the largest value, because this allows the robot agent to earn the largest amount of rewards in the long run. However, the values of neighboring states are unknown in practice and thus they should also be computed.

Now, we need to solve 3 subproblems of computing the values of state 2, state 6, and state 8. Then, in the same way, these subproblems are further decomposed as follows:

- The problem of computing the value of state 2 is decomposed into 3 subproblems of computing the values of state 1, state 3, and state 7.
- The problem of computing the value of state 6 is decomposed into 2 subproblems of computing the values of state 1 and state 7.
- The problem of computing the value of state 8 is decomposed into 3 subproblems of computing the values of state 3, state 7, and state 9.

Thus, by removing overlaps, the original problem of computing the value of state 7 has been decomposed into 6 unique subproblems: computing the values of state 1, state 2, state 3, state 6, state 8, and state 9.

If we further continue this problem decomposition, we encounter the problem of computing the values of state 17, where the robot agent can receive reward +1. Then the values of state 12 and state 18 can be explicitly computed. Indeed, if a discounting factor (a multiplicative penalty for delayed rewards) is 0.9, the values of state 12 and state 18 are $(0.9)^1 = 0.9$. Then we can further know that the values of state 13 and state 19 are $(0.9)^2 = 0.81$. By repeating this procedure, we can compute the values of all states (as illustrated in Figure 1.8). Based on these values, we can know the optimal action

the robot agent should take, i.e., an action that leads the robot agent to the neighboring state with the largest value.

Note that, in real-world reinforcement learning tasks, transitions are often not deterministic but stochastic, because of some external disturbance; in the case of the above maze example, the floor may be slippery and thus the robot agent cannot move as perfectly as it desires. Also, stochastic policies in which mapping from a state to an action is not deterministic are often employed in many reinforcement learning formulations. In these cases, the formulation becomes slightly more complicated, but essentially the same idea can still be used for solving the problem.

To further highlight the notable advantage of reinforcement learning that not the immediate rewards but the long-term accumulation of rewards is maximized, let us consider a *mountain-car problem* (Figure 1.9). There are two mountains and a car is located in a valley between the mountains. The goal is to guide the car to the top of the right-hand hill. However, the engine of the car is not powerful enough to directly run up the right-hand hill and reach the goal. The optimal policy in this problem is to first climb the left-hand hill and then go down the slope to the right with full acceleration to get to the goal (Figure 1.10).

Suppose we define the immediate reward such that moving the car to the right gives a positive reward $+1$ and moving the car to the left gives a negative reward -1. Then, a greedy solution that maximizes the immediate reward moves the car to the right, which does not allow the car to get to the goal due to lack of engine power. On the other hand, reinforcement learning seeks a solution that maximizes the return, i.e., the discounted sum of immediate rewards that the agent can collect over the entire trajectory. This means that the reinforcement learning solution will first move the car to the left even though negative rewards are given for a while, to receive more positive rewards in the future. Thus, the notion of "prior investment" can be naturally incorporated in the reinforcement learning framework.

1.2 Mathematical Formulation

In this section, the reinforcement learning problem is mathematically formulated as the problem of controlling a computer agent under a Markov decision process.

We consider the problem of controlling a computer agent under a discrete-time *Markov decision process* (MDP). That is, at each discrete time-step t, the agent observes a state $\boldsymbol{s}_t \in \mathcal{S}$, selects an action $a_t \in \mathcal{A}$, makes a transition $\boldsymbol{s}_{t+1} \in \mathcal{S}$, and receives an immediate reward,

$$r_t = r(\boldsymbol{s}_t, a_t, \boldsymbol{s}_{t+1}) \in \mathbb{R}.$$

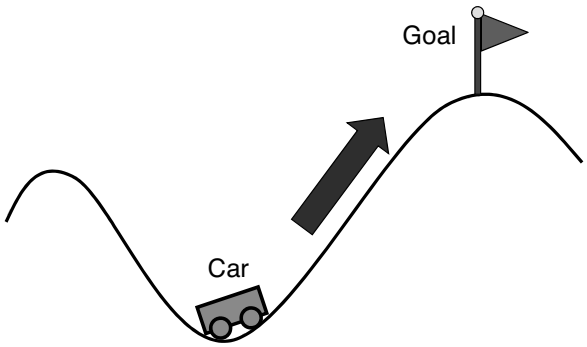


FIGURE 1.9: A mountain-car problem. We want to guide the car to the goal. However, the engine of the car is not powerful enough to directly run up the right-hand hill.

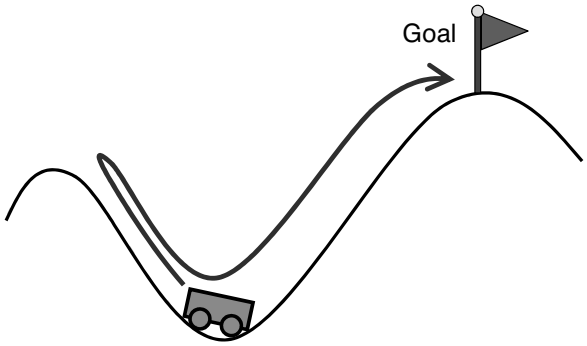


FIGURE 1.10: The optimal policy to reach the goal is to first climb the left-hand hill and then head for the right-hand hill with full acceleration.

$\mathcal{S}$ and $\mathcal{A}$ are called the *state space* and the *action space*, respectively. $r(\boldsymbol{s}, a, \boldsymbol{s}')$ is called the *immediate reward function*.

The initial position of the agent, $\boldsymbol{s}_1$, is drawn from the initial probability distribution. If the state space $\mathcal{S}$ is discrete, the initial probability distribution is specified by the *probability mass function* $P(\boldsymbol{s})$ such that

$$0 \leq P(\boldsymbol{s}) \leq 1, \ \forall \boldsymbol{s} \in \mathcal{S},$$
$$\sum_{\boldsymbol{s} \in \mathcal{S}} P(\boldsymbol{s}) = 1.$$

If the state space $\mathcal{S}$ is continuous, the initial probability distribution is specified by the *probability density function* $p(\boldsymbol{s})$ such that

$$p(\boldsymbol{s}) \geq 0, \ \forall \boldsymbol{s} \in \mathcal{S},$$

$$\int_{\boldsymbol{s}\in\mathcal{S}} p(\boldsymbol{s})\mathrm{d}\boldsymbol{s} = 1.$$

Because the probability mass function $P(\boldsymbol{s})$ can be expressed as a probability density function $p(\boldsymbol{s})$ by using the *Dirac delta function*[1] $\delta(\boldsymbol{s})$ as

$$p(\boldsymbol{s}) = \sum_{\boldsymbol{s}'\in\mathcal{S}} \delta(\boldsymbol{s}' - \boldsymbol{s})P(\boldsymbol{s}'),$$

we focus only on the continuous state space below.

The dynamics of the environment, which represent the transition probability from state $\boldsymbol{s}$ to state $\boldsymbol{s}'$ when action a is taken, are characterized by the *transition probability distribution* with conditional probability density $p(\boldsymbol{s}'|\boldsymbol{s}, a)$:

$$p(\boldsymbol{s}'|\boldsymbol{s}, a) \geq 0,\ \forall \boldsymbol{s}, \boldsymbol{s}' \in \mathcal{S},\ \forall a \in \mathcal{A},$$
$$\int_{\boldsymbol{s}'\in\mathcal{S}} p(\boldsymbol{s}'|\boldsymbol{s}, a)\mathrm{d}\boldsymbol{s}' = 1,\ \forall \boldsymbol{s} \in \mathcal{S},\ \forall a \in \mathcal{A}.$$

The agent's decision is determined by a *policy* π. When we consider a *deterministic* policy where the action to take at each state is uniquely determined, we regard the policy as a function of states:

$$\pi(\boldsymbol{s}) \in \mathcal{A},\ \forall \boldsymbol{s} \in \mathcal{S}.$$

Action a can be either discrete or continuous. On the other hand, when developing more sophisticated reinforcement learning algorithms, it is often more convenient to consider a *stochastic* policy, where an action to take at a state is probabilistically determined. Mathematically, a stochastic policy is a conditional probability density of taking action a at state $\boldsymbol{s}$:

$$\pi(a|\boldsymbol{s}) \geq 0,\ \forall \boldsymbol{s} \in \mathcal{S},\ \forall a \in \mathcal{A},$$
$$\int_{a\in\mathcal{A}} \pi(a|\boldsymbol{s})\mathrm{d}a = 1,\ \forall \boldsymbol{s} \in \mathcal{S}.$$

By introducing stochasticity in action selection, we can more actively *explore* the entire state space. Note that when action a is discrete, the stochastic policy is expressed using Dirac's delta function, as in the case of the state densities.

A sequence of states and actions obtained by the procedure described in Figure 1.11 is called a *trajectory*.

[1] The Dirac delta function $\delta(\cdot)$ allows us to obtain the value of a function f at a point τ via the *convolution* with f:

$$\int_{-\infty}^{\infty} f(s)\delta(s - \tau)\mathrm{d}s = f(\tau).$$

Dirac's delta function $\delta(\cdot)$ can be expressed as the Gaussian density with standard deviation $\sigma \to 0$:

$$\delta(a) = \lim_{\sigma\to 0} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{a^2}{2\sigma^2}\right).$$

1. The initial state $\boldsymbol{s}_1$ is chosen following the initial probability $p(\boldsymbol{s})$.
2. For $t = 1, \ldots, T$,
   (a) The action a_t is chosen following the policy $\pi(a_t|\boldsymbol{s}_t)$.
   (b) The next state $\boldsymbol{s}_{t+1}$ is determined according to the transition probability $p(\boldsymbol{s}_{t+1}|\boldsymbol{s}_t, a_t)$.

FIGURE 1.11: Generation of a trajectory sample.

When the number of steps, T, is finite or infinite, the situation is called the *finite horizon* or *infinite horizon*, respectively. Below, we focus on the finite-horizon case because the trajectory length is always finite in practice. We denote a trajectory by h (which stands for a "*history*"):

$$h = [\boldsymbol{s}_1, a_1, \ldots, \boldsymbol{s}_T, a_T, \boldsymbol{s}_{T+1}].$$

The discounted sum of immediate rewards along the trajectory h is called the *return*:

$$R(h) = \sum_{t=1}^{T} \gamma^{t-1} r(\boldsymbol{s}_t, a_t, \boldsymbol{s}_{t+1}),$$

where $\gamma \in [0, 1)$ is called the *discount factor* for future rewards.

The goal of reinforcement learning is to learn the optimal policy π^* that maximizes the *expected return*:

$$\pi^* = \operatorname*{argmax}_{\pi} \mathbb{E}_{p^\pi(h)}\Big[R(h)\Big],$$

where $\mathbb{E}_{p^\pi(h)}$ denotes the expectation over trajectory h drawn from $p^\pi(h)$, and $p^\pi(h)$ denotes the probability density of observing trajectory h under policy π:

$$p^\pi(h) = p(\boldsymbol{s}_1) \prod_{t=1}^{T} p(\boldsymbol{s}_{t+1}|\boldsymbol{s}_t, a_t)\pi(a_t|\boldsymbol{s}_t).$$

"argmax" gives the maximizer of a function (Figure 1.12).

For policy learning, various methods have been developed so far. These methods can be classified into *model-based reinforcement learning* and *model-free reinforcement learning*. The term "model" indicates a model of the transition probability $p(\boldsymbol{s}'|\boldsymbol{s}, a)$. In the model-based reinforcement learning approach, the transition probability is learned in advance and the learned transition model is explicitly used for policy learning. On the other hand, in the model-free reinforcement learning approach, policies are learned without explicitly estimating the transition probability. If strong prior knowledge of the

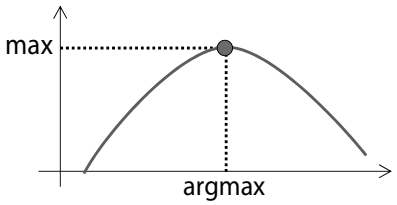


FIGURE 1.12: "argmax" gives the maximizer of a function, while "max" gives the maximum value of a function.

transition model is available, the model-based approach would be more favorable. On the other hand, learning the transition model without prior knowledge itself is a hard statistical estimation problem. Thus, if good prior knowledge of the transition model is not available, the model-free approach would be more promising.

1.3 Structure of the Book

In this section, we explain the structure of this book, which covers major reinforcement learning approaches.

1.3.1 Model-Free Policy Iteration

Policy iteration is a popular and well-studied approach to reinforcement learning. The key idea of policy iteration is to determine policies based on the *value function*.

Let us first introduce the *state-action value function* $Q^\pi(\boldsymbol{s}, a) \in \mathbb{R}$ for policy π, which is defined as the expected return the agent will receive when taking action a at state $\boldsymbol{s}$ and following policy π thereafter:

$$Q^\pi(\boldsymbol{s}, a) = \mathbb{E}_{p^\pi(h)}\Big[R(h)\Big|\boldsymbol{s}_1 = \boldsymbol{s}, a_1 = a\Big],$$

where "$|\boldsymbol{s}_1 = \boldsymbol{s}, a_1 = a$" means that the initial state $\boldsymbol{s}_1$ and the first action a_1 are fixed at $\boldsymbol{s}_1 = \boldsymbol{s}$ and $a_1 = a$, respectively. That is, the right-hand side of the above equation denotes the conditional expectation of $R(h)$ given $\boldsymbol{s}_1 = \boldsymbol{s}$ and $a_1 = a$.

Let $Q^*(\boldsymbol{s}, a)$ be the optimal state-action value at state $\boldsymbol{s}$ for action a defined as

$$Q^*(\boldsymbol{s}, a) = \max_\pi Q^\pi(\boldsymbol{s}, a).$$

Based on the optimal state-action value function, the optimal action the agent should take at state $\boldsymbol{s}$ is deterministically given as the maximizer of $Q^*(\boldsymbol{s}, a)$

1. Initialize policy $\pi(a|\boldsymbol{s})$.
2. Repeat the following two steps until the policy $\pi(a|\boldsymbol{s})$ converges.
   (a) Policy evaluation: Compute the state-action value function $Q^\pi(\boldsymbol{s}, a)$ for the current policy $\pi(a|\boldsymbol{s})$.
   (b) Policy improvement: Update the policy as

$$\pi(a|\boldsymbol{s}) \longleftarrow \delta\left(a - \operatorname*{argmax}_{a'} Q^\pi(\boldsymbol{s}, a')\right).$$

FIGURE 1.13: Algorithm of policy iteration.

with respect to a. Thus, the optimal policy $\pi^*(a|\boldsymbol{s})$ is given by

$$\pi^*(a|\boldsymbol{s}) = \delta\left(a - \operatorname*{argmax}_{a'} Q^*(\boldsymbol{s}, a')\right),$$

where $\delta(\cdot)$ denotes Dirac's delta function.

Because the optimal state-action value Q^* is unknown in practice, the policy iteration algorithm alternately evaluates the value Q^π for the current policy π and updates the policy π based on the current value Q^π (Figure 1.13).

The performance of the above policy iteration algorithm depends on the quality of policy evaluation; i.e., how to learn the state-action value function from data is the key issue. Value function approximation corresponds to a *regression* problem in statistics and machine learning. Thus, various statistical machine learning techniques can be utilized for better value function approximation. Part II of this book addresses this issue, including least-squares estimation and model selection (Chapter 2), basis function design (Chapter 3), efficient sample reuse (Chapter 4), active learning (Chapter 5), and robust learning (Chapter 6).

1.3.2 Model-Free Policy Search

One of the potential weaknesses of policy iteration is that policies are learned via value functions. Thus, improving the quality of value function approximation does not necessarily contribute to improving the quality of resulting policies. Furthermore, a small change in value functions can cause a big difference in policies, which is problematic in, e.g., robot control because such instability can damage the robot's physical system. Another weakness of policy iteration is that policy improvement, i.e., finding the maximizer of $Q^\pi(\boldsymbol{s}, a)$ with respect to a, is computationally expensive or difficult when the action space $\mathcal{A}$ is continuous.

Policy search, which directly learns policy functions without estimating value functions, can overcome the above limitations. The basic idea of policy search is to find the policy that maximizes the expected return:

$$\pi^* = \operatorname*{argmax}_{\pi} \mathbb{E}_{p^\pi(h)}\Big[R(h)\Big].$$

In policy search, how to find a good policy function in a vast function space is the key issue to be addressed. Part III of this book focuses on policy search and introduces gradient-based methods and the expectation-maximization method in Chapter 7 and Chapter 8, respectively. However, a potential weakness of these direct policy search methods is their instability due to the stochasticity of policies. To overcome the instability problem, an alternative approach called *policy-prior search*, which learns the policy-prior distribution for deterministic policies, is introduced in Chapter 9. Efficient sample reuse in policy-prior search is also discussed there.

1.3.3 Model-Based Reinforcement Learning

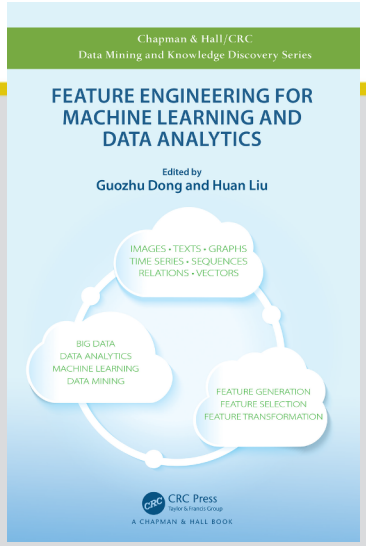
In the above model-free approaches, policies are learned without explicitly modeling the unknown environment (i.e., the transition probability of the agent in the environment, $p(\boldsymbol{s}'|\boldsymbol{s}, a)$). On the other hand, the model-based approach explicitly learns the environment in advance and uses the learned environment model for policy learning.

No additional sampling cost is necessary to generate artificial samples from the learned environment model. Thus, the model-based approach is particularly useful when data collection is expensive (e.g., robot control). However, accurately estimating the transition model from a limited amount of trajectory data in multi-dimensional continuous state and action spaces is highly challenging. Part IV of this book focuses on model-based reinforcement learning. In Chapter 10, a non-parametric transition model estimator that possesses the optimal convergence rate with high computational efficiency is introduced. However, even with the optimal convergence rate, estimating the transition model in high-dimensional state and action spaces is still challenging. In Chapter 11, a *dimensionality reduction* method that can be efficiently embedded into the transition model estimation procedure is introduced and its usefulness is demonstrated through experiments.

CHAPTER

5

DEEP LEARNING FOR FEATURE REPRESENTATION



This chapter is excerpted from

Feature Engineering for Machine Learning and Data Analytics

by Guozhu Dong and Huan Liu.

© 2018 Taylor & Francis Group. All rights reserved.

Learn more

Deep Learning for Feature Representation

Suhang Wang

Arizona State University

Huan Liu

Arizona State University

11.1 Introduction 279
11.2 Restricted Boltzmann Machine 280
11.2.1 Deep Belief Networks and Deep Boltzmann Machine ... 281
11.2.2 RBM for Real-Valued Data 283
11.3 AutoEncoder 284
11.3.1 Sparse Autoencoder 286
11.3.2 Denoising Autoencoder 287
11.3.3 Stacked Autoencoder 287
11.4 Convolutional Neural Networks 288
11.4.1 Transfer Feature Learning of CNN 290
11.5 Word Embedding and Recurrent Neural Networks 291
11.5.1 Word Embedding 291
11.5.2 Recurrent Neural Networks 294
11.5.3 Gated Recurrent Unit 295
11.5.4 Long Short-Term Memory 296
11.6 Generative Adversarial Networks and Variational Autoencoder 296
11.6.1 Generative Adversarial Networks 297
11.6.2 Variational Autoencoder 298
11.7 Discussion and Further Readings 299
Bibliography 301

11.1 Introduction

Deep learning methods have become increasingly popular in recent years because of their tremendous success in image classification [19], speech recognition [20] and natural language processing tasks [60]. In fact, deep learning methods have regularly won many recent challenges in these domains [19]. The

great success of deep learning mainly comes from specially designed structures of deep nets, which are able to learn discriminative non-linear features that can facilitate the task at hand. For example, the specially designed convolutional layers of CNN allow it to extract translation-invariant features from images while the max pooling layers of CNN help to reduce the parameters to be learned. In essence, the majority of existing deep learning algorithms can be used as powerful feature learning/extraction tools, i.e., the *latent features* extracted by deep learning algorithms are the new learned representations. In this chapter, we will review classical and popular deep learning algorithms and explain how they can be used for feature representation learning. We will also discuss how they are used for hierarchical and disentangled representation learning, and how they can be applied to various domains.

11.2 Restricted Boltzmann Machine

A restricted Boltzmann machine (RBM) is an undirected graphical model that defines a probability distribution over a vector of observed, or visible, variables $\mathbf{v} \in \{0,1\}^m$ and a vector of latent, or hidden, variables $\mathbf{h} \in \{0,1\}^d$, where m is the dimension of input features and d is the dimension of the latent features. It is widely used for unsupervised representation learning. For example, $\mathbf{v}$ can be the bag-of-words representation of documents or the vectorized binary images and $\mathbf{h}$ *is the learned representation for the input data*. A typical choice is that $d < m$, i.e., learning compact representation. Figure 11.1(a) gives a toy example of an RBM. In the figure, each node of the hidden layer is connected to each node in the visible layer, while there are no connections between hidden nodes or visible nodes. Figure 11.1(b) is a simplified representation of RBM, where the connection details between hidden layers and visible layers are simplified. We will begin by assuming both $\mathbf{v}$ and $\mathbf{h}$ as binary vectors, i.e., elements of $\mathbf{v}$ and $\mathbf{h}$ can only take the value of 0 or 1. An extension of real-valued input $\mathbf{x}$ will be introduced 11.2.2. An RBM defines a joint probability over $\mathbf{v}$ and $\mathbf{h}$ as

$$P(\mathbf{v}, \mathbf{h}) = \frac{1}{Z} \exp(-E(\mathbf{v}, \mathbf{h})) \tag{11.1}$$

where Z is the partition function defined as $Z = \sum_{\mathbf{v}} \sum_{\mathbf{h}} \exp(-E(\mathbf{v}, \mathbf{h}))$, and E is an energy function given by

$$E(\mathbf{v}, \mathbf{h}) = -\mathbf{h}^T \mathbf{W} \mathbf{v} - \mathbf{b}^T \mathbf{h} - \mathbf{c}^T \mathbf{v} \tag{11.2}$$

where $\mathbf{W} \in \mathbb{R}^{d \times m}$ is a matrix of pairwise weights between elements of $\mathbf{v}$ and $\mathbf{h}$ (see Figure 11.1(a)), while $\mathbf{b} \in \mathbb{R}^{d \times 1}$ and $\mathbf{c} \in \mathbb{R}^{m \times 1}$ are biases for the hidden and visible variables, respectively.[1]

[1] For simplicity, bias terms are not shown in Figure 11.1.

Since there are no explicit connections between hidden units in an RBM, given randomly selected training data $\mathbf{v}$, the hidden units are independent of each other, which gives $P(\mathbf{h}|\mathbf{v}) = \prod_{i=1}^{d} P(h_i|\mathbf{v})$, and the binary state, h_i, $i = 1, \ldots, d$, is set to 1 with conditional probability given as,

$$P(h_i = 1|\mathbf{v}) = \sigma\big(\sum_{j=1}^{m} W_{ij} v_j + b_i\big) \tag{11.3}$$

where $\sigma(\cdot)$ is the sigmoid function defined as $\sigma(x) = (1 + \exp(-x))^{-1}$. Similarly, given $\mathbf{h}$, the visible units are independent of each other. Thus, we have $P(\mathbf{v}|\mathbf{h}) = \prod_{j=1}^{m} P(v_j|\mathbf{h})$, and the binary state, v_j, $j = 1, \ldots, m$, is set to 1 with conditional probability given as

$$P(v_j = 1|\mathbf{h}) = \sigma(\sum_{i=1}^{d} W_{ij} h_i + v_j). \tag{11.4}$$

With the simple conditional probabilities given by Eq.(11.3) and Eq.(11.4), sampling from $P(\mathbf{h}|\mathbf{v})$ and $P(\mathbf{v}|\mathbf{h})$ becomes very efficient. RBMs have generally been trained using gradient ascent to maximize the log-likelihood $l(\boldsymbol{\theta})$ for some set of training vectors $\mathbf{V} \in \mathbb{R}^{m \times n}$, where $\boldsymbol{\theta} = \{\mathbf{W}, \mathbf{b}, \mathbf{c}\}$ is the set of variables to be optimized. The log-likelihood $l(\boldsymbol{\theta})$ is written as

$$l(\boldsymbol{\theta}) = \frac{1}{n} \log P(\mathbf{V}) = \frac{1}{n} \sum_{i=1}^{n} \log P(\mathbf{v}_i). \tag{11.5}$$

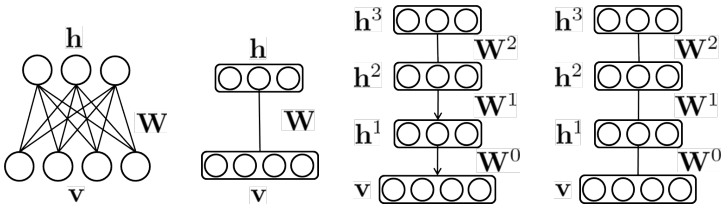
The derivative of $\log P(\mathbf{v})$ w.r.t variable $\mathbf{W}$ is given as

$$\frac{\partial \log P(\mathbf{v})}{\partial \mathbf{W}} = \sum_{\mathbf{h}} P(\mathbf{h}|\mathbf{v}) \mathbf{h} \mathbf{v}^T - \sum_{\tilde{\mathbf{v}}} \sum_{\mathbf{h}} P(\tilde{\mathbf{v}}, \mathbf{h}) \mathbf{h} \tilde{\mathbf{v}}^T \tag{11.6}$$

where $\tilde{\mathbf{v}} \in \{0, 1\}^m$ is an m-dimensional binary vector. The first term in Eq.(11.6) can be computed exactly. This term is often referred to as the *positive* gradient. It corresponds to the expected gradient of the energy with respect to $P(\mathbf{h}|\mathbf{v})$. The second term in Eq. (11.6) is known as the *negative* gradient, which is expectation over the model distribution $P(\mathbf{v}, \mathbf{h})$. It is intractable to compute the negative gradients exactly. Thus, we need to approximate the negative gradients by sampling $\mathbf{v}$ from $P(\mathbf{v}|\mathbf{h})$ and sampling $\mathbf{h}$ from $P(\mathbf{h}|\mathbf{v})$ by maintaining a Gibbs chain. For more details, we encourage readers to refere to Contrastive Divergence [62].

11.2.1 Deep Belief Networks and Deep Boltzmann Machine

RBMs can be stacked and trained in a greedy manner to form so-called Deep Belief Networks (DBN) [21] . DBNs are graphical models which learn



(a) RBM (b) RBM (c) 3-layer DBN (d) 3-layer DBM

Figure 11.1: An illustration of RBM, DBN and DBM.

to extract a deep hierarchical representation of the training data. They model the joint distribution between observed vector $\mathbf{v}$ and the l hidden layers as:

$$P(\mathbf{x}, \mathbf{h}^1, \mathbf{h}^2, \ldots, \mathbf{h}^l) = \left(\prod_{k=0}^{l-2} P(\mathbf{h^k} | \mathbf{h^{k+1}}) \right) P(\mathbf{h}^{l-1}, \mathbf{h}^l) \tag{11.7}$$

where $\mathbf{v} = \mathbf{h}^0$. $P(\mathbf{h}^{k-1}|\mathbf{h}^k)$ is a conditional distribution for the visible units conditioned on the hidden units of the RBM at level k, and $P(\mathbf{h}^{l-1}, \mathbf{h}^l)$ is the visible-hidden joint distribution in the top-level RBM. This is illustrated in Figure 11.1(c). DBN is able to learn hierarchical representation [33]. The low-level hidden representation such as $\mathbf{h}^1$ captures *low-level features* while the high-level hidden representation such as $\mathbf{h}^3$ captures more complex *high-level features*. Training of DBN is done by greedy layer-wise unsupervised training [21]. Specifically, we first train the first layer as an RBM with the raw input $\mathbf{v}$. From the first layer, we obtain the latent representation as the mean activations $P(\mathbf{h}^1|\mathbf{h}^0)$ or samples of $P(\mathbf{h}^1|\mathbf{h}^0)$, which will then be used as input to the second layer to update $\mathbf{W}^2$. After all the layers are trained, we can fine-tune all the parameters of DBN with respect to a proxy for the DBN log-likelihood, or with respect to a supervised training criterion by adding a classifier such as the softmax function on top of DBN.

A deep Boltzmann machine (DBM) [51] is another kind of deep generative model. Figure 11.1(d) gives an illustration of a DBM with 3 hidden layers. Unlike DBN, it is an entirely undirected model. Unlike RBM, the DBM has several layers of latent variables (RBMs have just one). Within each layer, each of the variables are mutually independent, conditioned on the variables in the neighboring layers. In the case of a deep Boltzmann machine with one visible layer $\mathbf{v}$, and l hidden layers, $\mathbf{h}^1$, $\mathbf{h}^2$ and $\mathbf{h}^l$, the joint probability is given by:

$$P(\mathbf{v}, \mathbf{h}^1, \mathbf{h}^2, \ldots, \mathbf{h}^n) = \frac{1}{Z} \exp(-E(\mathbf{v}, \mathbf{h}^1, \mathbf{h}^2, \ldots, \mathbf{h}^n)) \tag{11.8}$$

where the DBM energy function is defined as

$$E(\mathbf{v}, \mathbf{h}^1, \mathbf{h}^2, \ldots, \mathbf{h}^n) = -(\sum_{k=0}^{l-1} \mathbf{h}^k \mathbf{W}^k \mathbf{h}^{k+1}) - \sum_k \mathbf{b}^k \mathbf{h}^k \tag{11.9}$$

and $\mathbf{v} = \mathbf{h}^0$, $\mathbf{W}^k$ is the weight matrix to capture the interaction between $\mathbf{h}^k$ and $\mathbf{h}^{k+1}$, and $\mathbf{b}^k$ is the bias.

The conditional distribution over one DBM layer given the neighboring layers is factorial. In the example of the DBM with two hidden layers, these distributions are $P(\mathbf{v}|\mathbf{h}^1)$, $P(\mathbf{h}^1|\mathbf{v}, \mathbf{h}^2)$ and $P(\mathbf{h}^2|\mathbf{h}^1)$. The distribution over all hidden layers generally does not factorize because of interactions between layers. In the example with two hidden layers, $P(\mathbf{h}^1, \mathbf{h}^2|\mathbf{v})$ does not factorize due to the interaction weights $\mathbf{W}^1$ between $\mathbf{h}^1$ and $\mathbf{h}^2$ which render these variables mutually dependent. Therefore, sampling from $P(\mathbf{h}^1, \mathbf{h}^2|\mathbf{v})$ is difficult while training of DBM using gradient ascent methods require sampling from $P(\mathbf{h}^1, \mathbf{h}^2|\mathbf{v})$. To solve this problem, we use a mean-field approximation to approximate $P(\mathbf{h}^1, \mathbf{h}^2|\mathbf{v})$. Specifically, we define

$$Q(\mathbf{h}^1, \mathbf{h}^2) = \prod_j Q(h_j^1|\mathbf{v}) \prod_k Q(h_k^2|\mathbf{v}) \tag{11.10}$$

The mean field approximation attempts to find a member of this family of distributions that best fits the true posterior $P(\mathbf{h}^1, \mathbf{h}^2|\mathbf{v})$ by minimizing KL-divergence between $Q(\mathbf{h}^1, \mathbf{h}^2)$ and $P(\mathbf{h}^1, \mathbf{h}^2|\mathbf{v})$. With the approximation, we can easily sample $\mathbf{h}^1$ and $\mathbf{h}^2$ from $Q(\mathbf{h}^1, \mathbf{h}^2)$ and then update the parameters using gradient ascents with these samples [51].

11.2.2 RBM for Real-Valued Data

In many real-world applications such as image and audio modeling, the input features $\mathbf{v}$ are often real-valued data. Thus, it is important to extend RBM for modeling real-valued inputs. There are many variants of the RBM which defines the probability over real-valued data such as Gaussian-Bernoulli RBMs [69], mean and variance RBMs [22] and Spike and Slab RBMs [8].

The Gaussian-Bernoulli RBM (GBM) is the most common way to handle real-valued data, which has binary hidden units and real-valued visible units. It assumes the conditional distribution over the visible units being a Gaussian distribution whose mean is a function of the hidden units. Under this assumption, GRBM defines a joint probability over $\mathbf{v}$ and $\mathbf{h}$ as in Eq.(11.1) with the energy function given as

$$E(\mathbf{v}, \mathbf{h}) = -\mathbf{h}^T \mathbf{W}(\mathbf{v} \odot \boldsymbol{\beta}) - \mathbf{b}^T \mathbf{h} - \frac{1}{2}\mathbf{v} - \mathbf{c}^T(\boldsymbol{\beta} \odot (\mathbf{v} - \mathbf{c})) \tag{11.11}$$

where $\boldsymbol{\beta} \in \mathbb{R}^{m \times 1}$ is the precision vector with the i-th element β_i being the precision of v_i. $\odot$ is the Hadamard operation. Then the conditional probability of $P(\mathbf{v}|\mathbf{h})$ and $P(\mathbf{h}|\mathbf{v})$ are

$$P(\mathbf{h}|\mathbf{v}) = \prod_{i=1}^{d} P(h_i|\mathbf{v}) = \prod_{i=1}^{d} \sigma(b_i + \sum_{j=1}^{m} W_{ij} v_i \beta_i) \tag{11.12}$$

$$P(\mathbf{v}|\mathbf{h}) = \prod_{j=1}^{m} P(v_j|\mathbf{h}) = \prod_{j=1}^{m} \mathcal{N}(v_j|b_j + \sum_{i=1}^{d} W_{ij}h_i, \beta_i^{-1}) \tag{11.13}$$

where $\mathcal{N}(v_j|b_j + \sum_{i=1}^{d} W_{ij}h_i, \beta_i^{-1})$ is the Gaussian distribution with mean $b_j + \sum_{i=1}^{d} W_{ij}h_i$ and variance β_i^{-1}.

While the GRBM has been the canonical energy model for real-valued data, it is not well suited to the statistical variations present in some types of real-valued data, especially natural images [31]. The problem is that much of the information content present in natural images is embedded in the covariance between pixels rather than in the raw pixel values. To solve these problems, alternative models have been proposed that attempt to better account for the covariance of real-valued data. Mean and Covariance RBM (mcRBM) is one of the alternatives. The mcRBM uses its hidden units to independently encode the conditional mean and covariance of all observed units. Specifically, the hidden layer of mcRBM is divided into two groups of units: binary mean units $\mathbf{h}^{(m)}$ and binary covariance units $\mathbf{h}^{(c)}$. The energy function of mcRBM is defined as the combination of two energy functions:

$$E_{mc}(\mathbf{v}, \mathbf{h}^{(m)}, \mathbf{h}^{(c)}) = E_m(\mathbf{v}, \mathbf{h}^{(m)}) + E_c(\mathbf{v}, \mathbf{h}^{(c)}) \tag{11.14}$$

where $E_m(\mathbf{v}, \mathbf{h}^{(m)})$ is the standard Gaussian-Bernoulli energy function defined in Eq.(11.11), which models the interaction between real-valued $\mathbf{v}$ input and hidden units $\mathbf{h}^{(m)}$; and $E_c(\mathbf{v}, \mathbf{h}^{(c)})$ models the conditional covariance information, which is given as

$$E_c(\mathbf{v}, \mathbf{h}^{(c)}) = \frac{1}{2} \sum_j h_j^{(c)} (\mathbf{v}^T \mathbf{r}^{(j)})^2 - \sum_j b_j^{(c)} h_j^{(c)}. \tag{11.15}$$

The parameter $\mathbf{r}^{(j)}$ corresponds to the covariance weight vector associated with $h_j^{(c)}$ and $\mathbf{b}^{(c)}$ is a vector of covariance offsets.

11.3 AutoEncoder

An autoencoder (AE) is a neural network trained to learn latent representation that is good at reconstructing its input [4]. Generally, an autoencoder is composed of two parts, i.e., an encoder $f(\cdot)$ and a decoder $g(\cdot)$. An illustration of autoencoder is shown in Figure 11.2(a). The encoder maps the input $\mathbf{x} \in \mathbb{R}^m$ to latent representation $\mathbf{h} \in \mathbb{R}^d$ as $\mathbf{h} = f(\mathbf{x})$ and $f(\cdot)$ is usually a one-layer neural network, i.e., $f(\mathbf{x}) = s(\mathbf{W}\mathbf{x} + \mathbf{b})$, where $\mathbf{W} \in \mathbb{R}^{d \times m}$ and $\mathbf{b} \in \mathbb{R}^d$ are the weights and bias of the encoder. $s(\cdot)$ is a non-linear function such as sigmoid and tanh. A decoder maps back the latent representation $\mathbf{h}$ into a reconstruction $\tilde{\mathbf{x}} \in \mathbb{R}^m$ as $\tilde{\mathbf{x}} = g(\mathbf{h})$ and $g(\cdot)$ is given as $g(\mathbf{h}) = s(\mathbf{W}'\mathbf{h} + \mathbf{b}')$,

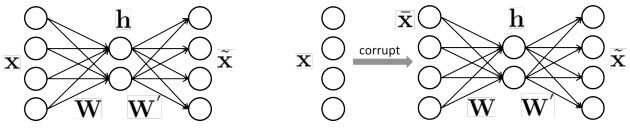
where $\mathbf{W}^{'} \in \mathbb{R}^{m \times d}$ and $\mathbf{b} \in \mathbb{R}^{m}$ are the weights and bias of the decoder. Note that the prime symbol does not indicate matrix transposition. The parameters of the autoencoder, i.e., $\theta = \{\mathbf{W}, \mathbf{b}, \mathbf{W}', \mathbf{b}'\}$ are optimized to minimize the reconstruction error. Depending on the appropriate distribution assumptions of the input, the reconstruction error can be measured in many ways. The most widely used reconstruction error is the squared error $\mathcal{L}(\mathbf{x}, \tilde{\mathbf{x}}) = \|\mathbf{x} - \tilde{\mathbf{x}}\|_2^2$. Alternatively, if the input is interpreted as either bit vectors or vectors of bit probabilities, cross-entropy of the reconstruction can be used

$$\mathcal{L}_H(\mathbf{x}, \tilde{\mathbf{x}}) = -\sum_{k=1}^{d}[\mathbf{x}_k \log \tilde{\mathbf{x}}_k + (1 - \mathbf{x}_k \log(1 - \tilde{\mathbf{x}}_k))]. \tag{11.16}$$

By training an autoencoder that is good at reconstructing input data, we hope that the latent representation $\mathbf{h}$ can capture some useful features. The identity function seems a particularly trivial function to try to learn, when it doesn't result in useful features. Therefore, we need to add constraints to the autoencoder to avoid trivial solution and learn useful features.

The autoencoder can be used to extract useful features by forcing $\mathbf{h}$ to have smaller dimension than $\mathbf{x}$, i.e., $d < m$. An autoencoder whose latent dimension is less than the input dimension is called an undercomplete autoencoder. Learning an undercomplete representation forces the autoencoder to capture the most salient features of the training data [15]. In other words, the latent representation $\mathbf{h}$ is a distributed representation which captures the coordinates along the main factors of variation in the data [15]. This is similar to the way that the projection on principal components would capture the main factors of variation in the data. Indeed, if there is one linear hidden layer, i.e., no activation function applied, and the mean squared error criterion is used to train the network, then the d hidden units learn to project the input in the span of the first d principal components of the data. If the hidden layer is non-linear, the autoencoder behaves differently from PCA, with the ability to capture multi-modal aspects of the input distribution.

Another choice is to constrain $\mathbf{h}$ to have a larger dimension than $\mathbf{x}$, i.e., $d > m$. An autoencoder whose latent dimension is larger than the input dimension is called an overcomplete autoencoder. However, due to the large dimension, the encoder and decoder are given too much capacity. In such cases, even a linear encoder and linear decoder can learn to copy the input to the output without learning anything useful about the data distribution. Fortunately, we can still discover interesting structure, by imposing other constraints on the network. One of the most widely used constraints is the sparsity constraint on $\mathbf{h}$. An overcomplete autoencoder with sparsity constraint is called a sparse autoencoder, which will be discussed next.



(a) Autoencoder (b) Denoising AE

Figure 11.2: An illustration of an autoencoder and a denoising autoencoder.

11.3.1 Sparse Autoencoder

A sparse autoencoder is an overcomplete authoencorder which tries to learn sparse overcomplete codes that are good at reconstruction [43]. A sparse overcomplete representation can be viewed as an alternative "compressed" representation: it has implicit straightforward compressibility due to the large number of zeros rather than an explicit lower dimensionality. Given the training data $\mathbf{X} \in \mathbb{R}^{m \times N}$, the objective function is given as

$$\min_{\mathbf{W},\mathbf{b},\mathbf{W}',\mathbf{b}'} \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}(\mathbf{x}_i, \tilde{\mathbf{x}}_i) + \alpha \Omega(\mathbf{h}_i) \tag{11.17}$$

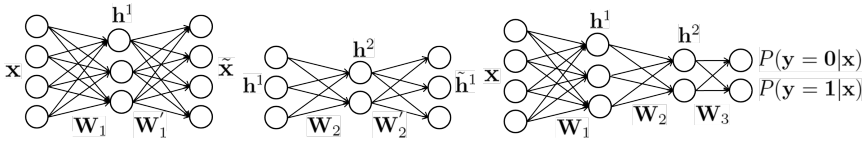
where N is the number of training instances, $\mathbf{x}_i$ is the i-th training instance, and $\mathbf{h}_i$ and $\tilde{\mathbf{x}}_i$ are the corresponding latent representation and reconstructed features. $\Omega(\mathbf{h}_i)$ is the sparsity regularizer to make $\mathbf{h}_i$ sparse and α is a scalar to control the sparsity. Many sparsity regularizers can be adopted. One popularly used is the ℓ_1-norm, i.e., $\Omega(\mathbf{h}_i) = \|\mathbf{h}_i\|_1 = \sum_{j=1}^{d} |h_i(j)|$. However, the ℓ_1-norm is non-smooth and not appropriate for gradient descent. An alternative is to use the smooth sparse constraint based on KL-divergence. Let ρ_j, $j = 1, \ldots, d$ be the average activation of hidden unit j (averaged over the training set) as

$$\rho_j = \frac{1}{N} \sum_{i=1}^{N} \mathbf{h}_i(j). \tag{11.18}$$

The essential idea is to force ρ_j to be close to ρ, where ρ is a small value close to zero (say $\rho = 0.05$). By forcing ρ_j be close to ρ, we would like the average activation of each hidden neuron j to be close to 0.05 (say). This constraint is satisfied when the hidden unit activations are mostly near 0. To achieve that ρ_j is close to ρ, we can use the KL-divergence as

$$\sum_{j=1}^{d} KL(\rho || \rho_j) = \sum_{j=1}^{d} \rho \log \frac{\rho}{\rho_j} + (1 - \rho) \log \frac{1 - \rho}{1 - \rho_j}. \tag{11.19}$$

$KL(\rho||\rho_j)$ is a convex function with its minimum of when $\rho_j = \rho$. Thus, minimizing this penalty term has the effect of causing ρ_j to be close to ρ, which achieves the sparse effect.



(a) 1st Autoencoder (b) 2nd Autoencoder (c) Fine-tuning

Figure 11.3: An illustration of 2-layer stacked autoencoder.

11.3.2 Denoising Autoencoder

The aforementioned autoencoders add constraints on latent representations to learn useful features. Alternatively, denoising the autoencoder uses the denoising criteria to learn useful features. In order to force the hidden layer to discover more robust features and prevent it from simply learning the identity, the denoising autoencoder trains the autoencoder to reconstruct the input from a corrupted version of it [63]. An illustration of denoising autoencoder is shown in Figure 11.2(b). In the figure, the clean data $\mathbf{x}$ is first corrupted as a noisy data $\bar{\mathbf{x}}$ by means of a stochastic mapping $q_D(\bar{\mathbf{x}}|\mathbf{x})$. The corrupted data $\bar{\mathbf{x}}$ is then used as input to an autoencoder, which outputs the reconstructed data $\tilde{\mathbf{x}}$. The training objective of a denoising autoencoder is then to make reconstructed data $\tilde{\mathbf{x}}$ close to the clean data $\mathbf{x}$ as $\mathcal{L}(\mathbf{x}, \tilde{\mathbf{x}})$.

There are many choices of the stochastic mapping such as (1) additive isotropic Gaussian noise (GS): $\bar{\mathbf{x}}|\mathbf{x} \sim N(\mathbf{x}, \sigma\mathbf{I})$; this is a very common noise model suitable for real-valued inputs. (2) Masking noise (MN): a fraction ν of the elements of x (chosen at random for each example) is forced to 0. (3) Salt-and-pepper noise (SP): a fraction ν of the elements of $\mathbf{x}$ (chosen at random for each example) is set to their minimum or maximum possible value (typically 0 or 1) according to a fair coin flip. The masking noise and salt-and-pepper noise are natural choices for input domains which are interpretable as binary or near binary such as black-and-white images or the representations produced at the hidden layer after a sigmoid squashing function [63].

11.3.3 Stacked Autoencoder

Denoising autoencoders can be stacked to form a deep network by feeding the latent representation of the DAE found on the layer below as input to the current layer as shown in Figure 11.3, which are generally called stacked denoising autoencoders (SDAEs). The unsupervised pre-training of such an architecture is done one layer at a time. Each layer is trained as a DAE by minimizing the error in reconstructing its input. For example, in Figure 11.3(a), we train the first layer autoencoder. Once the first layer is trained, we can train the 2nd layer with the latent representation of the first autoencoder, i.e., $\mathbf{h}^1$, as input. This is shown in Figure 11.3(b). Once all layers are pre-trained, the network goes through a second stage of training called fine-tuning, which is typically to minimize prediction error on a supervised task. For fine-tuning,

we first add a logistic regression layer on top of the network as shown in Figure 11.3(c) (more precisely on the output code of the output layer). We then train the entire network as we would train a multilayer perceptron. At this point, we only consider the encoding parts of each autoencoder. This stage is supervised, since now we use the target class during training.

11.4 Convolutional Neural Networks

The Convolutional Neural Network (CNN or ConvNet) has achieved great success in many computer vision tasks such as image classification [32], segmentation [36] and video action recognition [55]. The specially designed architecture of the CNN is very powerful in extracting visual features from images, which can be used for various tasks. An example of a simplified CNN is shown in Figure 11.4. It is comprised of three basic types of layers, which are convolutional layers for extracting translation-invariant features from images, pooling layers for reducing the parameters and fully connected layers for classification tasks. CNNs are mainly formed by stacking these layers together. Recently, dropout layers [56] and residual layers [19] are also introduced to prevent CNN from overfitting and to ease the training of deep CNNs, respectively. Next, we will introduce the basic building blocks of CNNs and how CNNs can be used for feature learning.

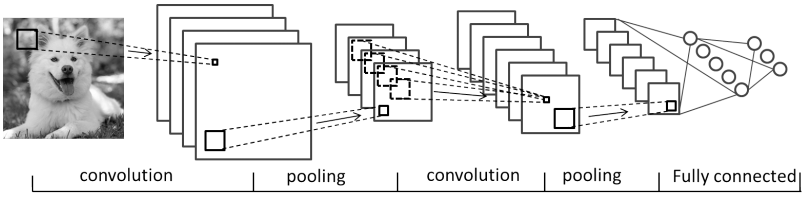


Figure 11.4: An illustration of CNN.

The Convolutional Layer: As the name implies, the Conv layer is the core building block of a CNN. The essential idea of a Conv layer is the observation that natural images have the property of being "stationary," which means that the statistics of one part of the image are the same as any other part. For example, a dog can appear in any location of an image. This suggests that the dog feature detector that we learn at one part of the image can also be applied to other parts of the image to detect dogs, and we can use the same features at all locations. More precisely, having learned features over small (say 3x3) patches sampled randomly from the larger image, we can then apply this learned 3x3 feature detector anywhere in the image. Specifically, we can take the learned 3x3 features and "convolve" them with the larger image, thus obtaining a different feature activation value at each location in

the image. The feature detector is called a filter or kernel in ConvNet and the feature obtained is called a feature map. Figure 11.5 gives an example of a convolution operation with the input as the 5x5 matrix and the kernel as the 3x3 matrix. The 3x3 kernel slides over the 5x5 matrix from left to right and from the top to down, which generates the feature map shown on the right. The convolution is done by multiplying the kernel by the sub-patch of the input feature map and then sum together. For example, the calculation of the gray sub-patch in the 5x5 matrix with the kernel is given in the figure. There

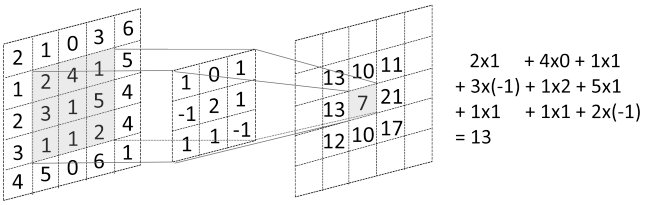


Figure 11.5: An illustration of convolution operation.

are three parameters in a Conv layer, i.e., the *depth*, *stride* and *zero-padding*. *Depth* corresponds to the number of filters we would like to use. A Conv layer can have many filters, each learning to look for something different in the input. For example, if the first Conv layer takes as input the raw image, then different neurons along the depth dimension may activate in the presence of various oriented edges, or blobs of color. In the simple ConvNet shown in Figure 11.4, the depth of the first convolution and second convolution layers are 4 and 6, respectively. *Stride* specifies how many pixels we skip when we slide the filter over the input feature map. When the stride is 1, we move the filters one pixel at a time as shown in Figure 11.5. When the stride is 2, the filters jump 2 pixels at a time as we slide them around. This will produce smaller output volumes spatially. It will be convenient to pad the input volume with zeros around the border, which is called *zero-padding*. The size of this zero-padding is a hyperparameter. The nice feature of zero-padding is that it will allow us to control the spatial size of the output volumes. Let the input volume be $W \times H \times K$, where W and H are width and height of the feature map and K is the number of feature maps. For example, for a color image with RGB channels, we have $K = 3$. Let the receptive field size (filter size) of the Conv Layer be F, number of filters be $\tilde{K}$, the stride with which they are applied be S, and the amount of zero padding used on the border be P; then the output volume after convolution is $\tilde{W} \times \tilde{H} \times \tilde{K}$, where $\tilde{W} = (W - F + 2P)/S + 1$ and $\tilde{H} = (H - F + 2P)/S + 1$. For example, for a $7 \times 7 \times 3$ input and a $4 \times 3 \times 3$ filter with stride 1 and pad 0, we would get a $5 \times 5 \times 4$ output.

Convolution using filters is a linear operation. After the feature maps are obtained in a Conv layer, a nonlinear activation function will be applied on these feature maps to learn non-linear features. Rectified linear unit (ReLU) is the most widely used activation function for ConvNet, which is demonstrated

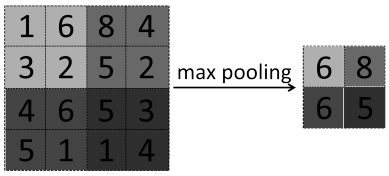


Figure 11.6: An illustration of a max pooling operation.

to be effective in alleviating the gradient vanishing problem. A rectifier is defined as $f(x) = \max(0, x)$.

The Pooling Layer: Pooling layers are usually periodically inserted between successive Conv layers in a CNN. They aim to progressively reduce the spatial size of the representation, which can help reduce the number of parameters and computation in the network, and hence also control overfitting. The pooling layer operates independently over each activation map in the input, and scales its dimensionality using the *max* function. The most common form is a pooling layer with filters of size 2x2 applied with a stride of 2, which downsamples every depth slice in the input by 2 along both width and height, discarding 75% of the activations. Every max operation would, in this case, be taking a max over 4 numbers and the maximum value of the 4 numbers will go to the next layer. An example of a max pooling operation is shown in Figure 11.6. During the forward pass of a pooling layer it is common to keep track of the index of the max activation (sometimes also called the switches) so that gradient routing is efficient during backpropagation.

Though max pooling is the most popular pooling layer, a CNN can also contain general pooling. General pooling layers are comprised of pooling neurons that are able to perform a multitude of common operations including L1/L2-normalization, and average pooling. An example of max pooling

The Fully Connected Layer: Neurons in a fully connected layer have full connections to all activations in the previous layer, as shown in Figure 11.4. The fully connected layers are put at the end of a CNN architecture, i.e., after several layers of Conv layer and max pooling layers. With the high-level features extracted by the previous layers, fully connected layers will then attempt to produce class scores from the activations, to be used for classification. The output of the fully connected layer will then be put in a softmax for classification. It is also suggested that ReLu may be used as the activation function in a fully connected layer to improve performance.

11.4.1 Transfer Feature Learning of CNN

In practice, training an entire Convolutional Network from scratch (with random initialization) is rare as (1) it is very time consuming and requires many computation resources and (2) it is relatively rare to have a dataset of sufficient size to train a ConvNet. Therefore, instead, it is common to

pre-train a ConvNet on a very large dataset (e.g., ImageNet, which contains 1.2 million images with 1000 categories), and then use the ConvNet either as an initialization or a fixed feature extractor for the task of interest [53]. There are mainly two major Transfer Learning scenarios, which are listed as follows:

- ConvNet as a fixed feature extractor: In this scenario, we take a ConvNet pretrained on ImageNet, remove the last fully connected layer, then treat the rest of the ConvNet as a fixed feature extractor for the new dataset. With the extracted features, we can train a linear classifier such as Linear SVM or logistic regression for the new dataset. This is usually used when the new dataset is small and similar to the original dataset. For such datasets, training or fine-tuning a ConvNet is not practical as ConvNets are prone to overfitting to small datasets. Since the new dataset is similar to the original dataset, we can expect higher-level features in the ConvNet to be relevant to this dataset as well.
- Fine-tuning the ConvNet: The second way is to not only replace and retrain the classifier on top of the ConvNet on the new dataset, but to also fine-tune the weights of the pretrained network using backpropagation. The essential idea of fine-tuning is that the earlier features of a ConvNet contain more generic features (e.g., edge detectors or color blob detectors) that should be useful in many tasks, but later layers of the ConvNet become progressively more specific to the details of the classes contained in the original dataset. If the new dataset is large enough, we can fine-tune all the layers of the ConvNet. If the new dataset is small but different from the original dataset, then we can keep some of the earlier layers fixed (due to overfitting concerns) and only fine-tune some higher-level portion of the network.

11.5 Word Embedding and Recurrent Neural Networks

Word embedding and recurrent neural networks are the state-of-the-art deep learning models for natural language processing tasks. Word embedding learns word representation and recurrent neural networks utilize word embedding for sentence or document feature learning. Next, we introduce the details of word embedding and recurrent neural networks.

11.5.1 Word Embedding

Word embedding, or distributed representation of words, is to represents each word as a low-dimensional dense vector such that the vector representation of words can capture synthetic and semantic meanings of words. The

low-dimensional representation can also alleviate the curse of dimensionality and data sparsity problems suffered by traditional representations such as bag-of-words and N-gram [66]. The essential idea of word embedding is the distributional hypothesis that "you shall know a word by the company it keeps" [13]. This suggests that a word has close relationships with its neighboring words. For example, the phrases *win the game* and *win the lottery* appear very frequently; thus the pair of words *win* and *game* and the pair of words *win* and *lottery* could have a very close relationship. When we are only given the word *win*, we would highly expect the neighboring words to be words like *game* or *lottery* instead of words as *light* or *air*. This suggests that a good word representation should be useful for predicting its neighboring words, which is the essential idea of Skip-gram [41]. In other words, the training objective of the Skip-gram model is to find word representations that are useful for predicting the surrounding words in a sentence or a document. More formally, given a sequence of training words $w_1, w_2, \ldots, w_T$, the objective of the Skip-gram model is to maximize the average log probability

$$\frac{1}{T}\sum_{t=1}^{T}\sum_{-c\leq j\leq c, j\neq 0}\log P(w_{t+j}|w_t) \tag{11.20}$$

where c is the size of the training context (which can be a function of the center word w_t). Larger c results in more training examples and thus can lead to higher accuracy, at the expense of training time. The basic Skip-gram formulation defines $P(w_{t+j}|w_t)$ using the softmax function:

$$P(w_O|w_I) = \frac{\exp(\mathbf{u}_{w_O}^T \mathbf{v}_{w_I})}{\sum_{w=1}^{W}\exp(\mathbf{u}_w{}^T \mathbf{v}_{w_I})} \tag{11.21}$$

where $\mathbf{v}_w$ and $\mathbf{u}_w$ are the "input" and "output" representations of w, and W is the number of words in the vocabulary. Learning the representation is usually done by gradient descent. However, Eq.(11.21) is impractical because the cost of computing $\nabla \log P(w_O|w_I)$ is proportional to W, which is often large. One way of making the computation more tractable is to replace the softmax in Eq. (11.21) with a hierarchical softmax. In a hierarchical softmax, the vocabulary is represented as a Huffman binary tree with words as leaves. With the Huffman tree, the probability of $P(w_O|w_I)$ is the probability of walking the path from root node to leaf node w_O given the word w_I, which is calculated as decision making in each node along the path with a simple function. Huffman trees assign short binary codes to frequent words, and this further reduces the number of output units that need to be evaluated. Another alternative to make the computation tractable is negative sampling [41]. The essential idea of negative sampling is that w_t should be more similar to its neighboring words, say w_{t+j}, than randomly sampled words. Thus, the objective function of negative sampling is to maximize the similarity between w_t and w_{t+j} and minimize the similarity between w_t and randomly sampled words. With

negative sampling, Eq. (11.21) is approximated as

$$\log \sigma(\mathbf{u}_{W_O}^T \mathbf{v}_{w_I}) + \frac{1}{K} \sum_{i=1}^{K} \log \sigma(-\mathbf{u}_{W_i}^T \mathbf{v}_{w_I}) \tag{11.22}$$

where K is the number of negative words sampled for each input word w_I. It is found that skip-gram with negative sampling is equivalent to implicitly factorizing a word-context matrix, whose cells are the pointwise mutual information (PMI) of the respective word and context pairs, shifted by a global constant [34].

Instead of using the center words to predict the context (surrounding words in a sentence), Continuous Bag-of-Words Model (CBOW) predicts the current word based on the context. More precisely, CBOW uses each current word as an input to a log-linear classifier with a continuous projection layer, and predicts words within a certain range before and after the current word [39]. The objective function of CBOW is to maximize the following log-likelihood function

$$\frac{1}{T} \sum_{t=1}^{T} \log P(w_t | w_{t-c}, \ldots, w_{t-1}, w_{t+1}, \ldots, w_{t+c}) \tag{11.23}$$

and $P(w_t | w_{t-c}, \ldots, w_{t-1}, w_{t+1}, \ldots, w_{t+c})$ is defined as

$$P(w_t | w_{t-c}, \ldots, w_{t-1}, w_{t+1}, \ldots, w_{t+c}) = \frac{\exp({\mathbf{u}_{w_t}}^T \tilde{\mathbf{v}}_t)}{\sum_{w=1}^{W} \exp({\mathbf{u}_w}^T \tilde{\mathbf{v}}_t)} \tag{11.24}$$

where $\tilde{\mathbf{v}}_t$ is the average representation of the contexts of w_t, i.e., $\tilde{\mathbf{v}}_t = \frac{1}{2c} \sum_{-c \le j \le c, j \ne 0} \mathbf{v}_{t+j}$.

Methods like skip-gram may do better on the analogy task, but they poorly utilize the statistics of the corpus since they train on separate local context windows instead of on global co-occurrence counts. Based on this observation, GloVe, proposed in [46], uses a specific weighted least squares model that trains on global word-word co-occurrence counts and thus makes efficient use of statistics. The objective function of GloVe is given as

$$\min \sum_{i,j} f(X_{ij}) (\mathbf{w}_i^T \tilde{\mathbf{w}}_j - \log X_{ij})^2 \tag{11.25}$$

where X_{ij} tabulates the number of times word j occurs in the context of word i. $\mathbf{w}_i \in \mathbb{R}^d$ is the word representation of w_i and $\tilde{\mathbf{w}}_j$ is a separate context word vector. $f()$ is the weighting function.

Word embedding can capture syntactic and semantic meanings of words. For example, it is found that vec(queen) is the closest vector representation to vec(king) - vec(man) + vec(woman), which implies that word representation learned by Skip-gram encodes semantic meanings of words. Word embedding can also be used for document representation by averaging the word vectors of words appearing in a document as the vector representation of the documents.

Following the distributional representation idea of word embedding, many network embedding algorithms are proposed. The essential idea of network embedding is to learn vector representations of network nodes that are good at predicting the neighboring nodes.

Since word representation learned by word embedding algorithms are low-dimensional dense vectors that capture semantic meanings, they are widely used as a preprocessing step in deep learning methods such as recurrent neural networks and recursive neural networks. Each word will be mapped to a vector representation before it is used as input to deep learning models.

11.5.2 Recurrent Neural Networks

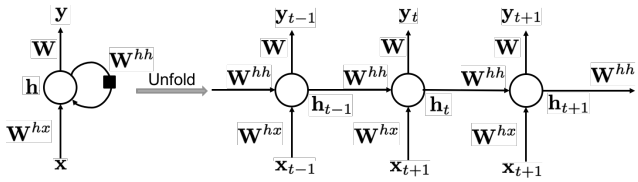


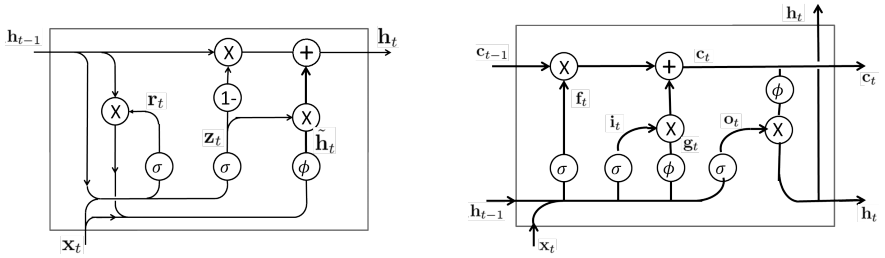
Figure 11.7: An illustration of an RNN.

Recurrent neural networks (RNN) are powerful concepts that allow the use of loops within the neural network architecture to model sequential data such as sentences and videos. Recurrent networks take as input a sequence of inputs, and produce a sequence of outputs. Thus, such models are particularly useful for sequence-to-sequence learning.

Figure 11.7 gives an illustration of the RNN architecture. The left part of the figure shows a folded RNN, which has a self-loop, i.e., the hidden state $\mathbf{h}$ is used to update itself given an input $\mathbf{x}$. To better show how RNN works, we unfold the RNN as a sequential structure, which is given in the right part of Figure 11.7. The RNN takes a sequence, $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_t, \ldots, \mathbf{x}_T$ as input, where at each step t, $\mathbf{x}_t$ is a d-dimensional feature vector. For example, if the input is a sentence, then each word w_i of the sentence is represented as a vector $\mathbf{x}_i$ using word embedding methods such as Skip-gram. At each time-step t, the output of the previous step, $\mathbf{h}_{t-1}$, along with the next word vector in the document, $\mathbf{x}_t$, are used to update the hidden state $\mathbf{h}_t$ as

$$\mathbf{h}_t = \sigma(\mathbf{W}^{hh}\mathbf{h}_{t-1} + \mathbf{W}^{hx}\mathbf{x}_t) \tag{11.26}$$

where $\mathbf{W}^{hh} \in \mathbb{R}^{d\times d}$ and $\mathbf{W}^{hx} \in \mathbb{R}^{d\times d}$ are the weights for inputs $\mathbf{h}_{t-1}$ and $\mathbf{x}_t$, respectively. *The hidden states $\mathbf{h}_t$ is the feature representation of the sequence up to time t for the input sequence.* The initial states $\mathbf{h}_0$ are usually initialized as all 0. Thus, we can utilize $\mathbf{h}_t$ to perform various tasks such as sentence completion and document classification. For example, for a sentence completion task, we are given the partial sentence, "The weather is..." and



(a) One GRU Cell (b) One LSTM Cell

Figure 11.8: An illustration of GRU and LSTM cells.

we want to predict the next word. We can predict the next word as

$$\mathbf{y}_t = softmax(\mathbf{W}\mathbf{h}_t + \mathbf{b}), \quad y_t = \arg\max \mathbf{y}_t \tag{11.27}$$

where $\mathbf{W} \in \mathbb{R}^{V \times d}$ are the weights of the softmax function with V being the size of the vocabulary and b the bias term. $\mathbf{y}_t$ is the predicted probability vector and y_t is the predicted label. We can think of an RNN as modeling the likelihood probability as $P(\mathbf{y}_t|\mathbf{x}_1, \ldots, \mathbf{x}_t)$.

Training of RNNs is usually done using Backpropagation Through Time (BPTT), which back propagates the error from time t to time 1 [70].

11.5.3 Gated Recurrent Unit

Though in theory, RNN is able to capture long-term dependency, in practice, the old memory will fade away as the sequence becomes longer. To make it easier for RNNs to capture long-term dependencies, gated recurrent units (GRUs) [7] are designed to have more persistent memory. Unlike an RNN, which uses a simple affine transformation of $\mathbf{h}_{t-1}$ and $\mathbf{h}_t$ followed by $tanh$ to update $\mathbf{h}_t$, GRU introduces the Reset Gate to determine if it wants to forget past memory and the Update Gate to control if new inputs are introduced to $\mathbf{h}_t$. The mathematical equations of how this is achieved are given as follows and an illustration of a GRU cell is shown in Figure 11.8(a):

$$\begin{aligned}
\mathbf{z}_t &= \sigma(\mathbf{W}_{zx}\mathbf{x}_t + \mathbf{W}_{zh}\mathbf{h}_{t-1} + \mathbf{b}_z) && \text{(Update gate)} \\
\mathbf{r}_t &= \sigma(\mathbf{W}_{rx}\mathbf{x}_t + \mathbf{W}_{rh}\mathbf{h}_{t-1} + \mathbf{b}_r) && \text{(Reset gate)} \\
\tilde{\mathbf{h}}_t &= \tanh(\mathbf{r}_t \odot \mathbf{U}\mathbf{h}_{t-1} + \mathbf{W}\mathbf{x}_t) && \text{(New memory)} \\
\mathbf{h}_t &= (1 - \mathbf{z}_t) \odot \tilde{\mathbf{h}}_t + \mathbf{z}_t \odot \mathbf{h}_{t-1} && \text{(Hidden state)}
\end{aligned} \tag{11.28}$$

From the above equation and Figure 11.8(a), we can treat the GRU as four fundamental operational stages, i.e., new memory, update gate, reset gate and hidden state. A new memory $\tilde{h}_t$ is the consolidation of a new input word $\mathbf{x}_t$ with the past hidden state $\mathbf{h}_{t-1}$, which summarizes this new word in light of the contextual past. The reset signal $\mathbf{r}_t$ is used to determining how important

$\mathbf{h}_{t-1}$ is to the summarization $\tilde{\mathbf{h}}_t$. The reset gate has the ability to completely diminish a past hidden state if it finds that $\mathbf{h}_{t-1}$ is irrelevant to the computation of the new memory. The update signal $\mathbf{z}_t$ is responsible for determining how much of past state $\mathbf{h}_{t-1}$ should be carried forward to the next state. For instance, if $\mathbf{z}_t \approx 1$, then $\mathbf{h}_{t-1}$ is almost entirely copied out to $\mathbf{h}_t$. The hidden state $\mathbf{h}_t$ is finally generated using the past hidden input $\mathbf{h}_t$ and the new memory generated $\tilde{\mathbf{h}}_t$ with the control of the update gate.

11.5.4 Long Short-Term Memory

Long Short-Term Memories, LSTMs [23], are another variant of the RNN, which can also capture long-term dependency. Similar to GRUs, an LSTM introduces more complex gates to control if it should accept new information or forget previous memory, i.e., the input gate, forget gate, output gate and new memory cell. The update rules of LSTMs are given as follows:

$$
\begin{aligned}
\mathbf{i}_t &= \sigma(\mathbf{W}_{ix}\mathbf{x}_t + \mathbf{W}_{ih}\mathbf{h}_{t-1} + \mathbf{b}_i) && \text{(Input gate)} \\
\mathbf{f}_t &= \sigma(\mathbf{W}_{fx}\mathbf{x}_t + \mathbf{W}_{fh}\mathbf{h}_{t-1} + \mathbf{b}_f) && \text{(Forget gate)} \\
\mathbf{o}_t &= \sigma(\mathbf{W}_{ox}\mathbf{x}_t + \mathbf{W}_{oh}\mathbf{h}_{t-1} + \mathbf{b}_o) && \text{(Output gate)} \\
\mathbf{g}_t &= \tanh(\mathbf{W}_{gx}\mathbf{x}_t + \mathbf{W}_{gh}\mathbf{h}_{t-1} + \mathbf{b}_g) && \text{(New memory cell)} \\
\mathbf{c}_t &= \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \mathbf{g}_t && \text{(Final memory cell)} \\
\mathbf{h}_t &= \mathbf{o}_t \odot \tanh(\mathbf{c}_t) && (11.29)
\end{aligned}
$$

where $\mathbf{i}_t$ is the input gate, $\mathbf{f}_t$ is the forget gate, $\mathbf{o}_t$ is the forget fate, $\mathbf{c}_t$ is the memory cell state at t and $\mathbf{x}_t$ is the input features at t. $\sigma(\cdot)$ means the sigmoid function and $\odot$ denotes the Hadamard product. The main idea of the LSTM model is the memory cell $\mathbf{c}_t$, which records the history of the inputs observed up to t. $\mathbf{c}_t$ is a summation of (1) the previous memory cell $\mathbf{c}_{t-1}$ modulated by a sigmoid gate $\mathbf{f}_t$, and (2) $\mathbf{g}_t$, a function of previous hidden states and the current input modulated by another sigmoid gate $\mathbf{i}_t$. The sigmoid gate $\mathbf{f}_t$ is to selectively forget its previous memory while $\mathbf{i}_t$ is to selectively accept the current input. $\mathbf{i}_t$ is the gate controlling the output. The illustration of a cell of LSTM at the time step t is shown in Figure 11.8(b).

11.6 Generative Adversarial Networks and Variational Autoencoder

In this section, we introduce two very popular deep generative models proposed recently, i.e., generative adversarial networks and the variational autoencoder.

11.6.1 Generative Adversarial Networks

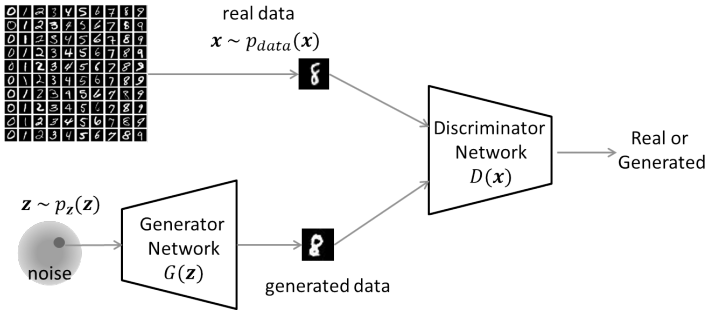


Figure 11.9: An illustration of the framework of the GAN.

The generative adversarial network (GAN) [16] is one of the most popular generative deep models. The core of a GAN is to play a min-max game between a discriminator D and a generator G, i.e., adversarial training. The discriminator D tries to differentiate if a sample is from the real world or generated by the generator, while the generator G tries to generate samples that can fool the discriminator, i.e., make the discriminator believe that the generated samples are from the real world. Figure 11.9 gives an illustration of the framework of the GAN. The generator takes a noise $\mathbf{z}$ sampled from a prior distribution $p_{\mathbf{z}}(\mathbf{z})$ as input, and maps the noise to the data space as $G(\mathbf{z};\theta_g)$. Typical choices of the prior $p(\mathbf{z})$ can be a uniform distribution or Gaussian distribution. We also define a second multilayer perceptron $D(\mathbf{x};\theta_d)$ that outputs a single scalar. $D(\mathbf{x})$ represents the probability that $\mathbf{x}$ came from the real-world data rather than generated data. D is trained to maximize the probability of assigning the correct label to both training examples and samples from G. We simultaneously train G to minimize $\log(1 - D(G(\mathbf{z})))$. In other words, D and G play the following two-player minimax game with value function $V(D, G)$:

$$\min_G \max_D V(D,G) = \mathbb{E}_{\mathbf{x}\sim p_{data}(\mathbf{x})}[\log D(\mathbf{x})] + \mathbb{E}_{\mathbf{z}\sim p_{\mathbf{z}(\mathbf{z})}}[\log(1 - D(G(\mathbf{z})))]. \tag{11.30}$$

The training of a GAN can be done using minibatch stochastic gradient descent training by updating the parameters of G and D alternatively. *After the model is trained without supervision, we can treat the discriminator as a feature extractor*: The first few layers of D extract features from $\mathbf{x}$ while the last few layers are to map the features to the probability that $\mathbf{x}$ is from real data. Thus, we can remove the last few layers, then the output of D is the features extracted. In this sense, we treat GANs as unsupervised feature learning algorithms, though the main purpose of the GAN is to learn $p(\mathbf{x})$.

The GAN is a general adversarial training framework, which can be used for various domains by designing a different generator, discriminator and loss function [6, 65, 74]. For example, InfoGAN [6] learns disentangled representa-

tion by dividing the noise into two parts, i.e., disentangled codes $\mathbf{c}$ and incompressible noise $\mathbf{z}$ so that the disentangled codes $\mathbf{c}$ can control the properties such as the identity and illumination of the images generated. SeqGAN [74] models the data generator as a stochastic policy in reinforcement learning and extends the GAN for text generation.

11.6.2 Variational Autoencoder

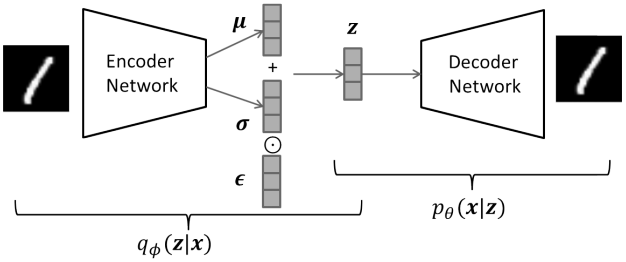


Figure 11.10: An illustration of the framework of the VAE.

The variational autoencoder (VAE) [30] is a popular generative model for unsupervised representation learning. It can be trained purely with gradient-based methods. Typically, a VAE has a standard autoencoder component which encodes the input data into a latent code space by minimizing reconstruction error, and a Bayesian regularization over the latent space, which forces the posterior of the hidden code vector to match a prior distribution. Figure 11.10 gives an illustration of a VAE. To generate a sample from the model, the VAE first draws a sample $\mathbf{z}$ from the prior distribution $p_\theta(\mathbf{z})$. The sample $\mathbf{z}$ is used as input to a differentiable generator network $g(\mathbf{z})$. Finally, $\mathbf{x}$ is sampled from a distribution $p_\theta(\mathbf{x}|g(\mathbf{z})) = p_\theta(\mathbf{x}|\mathbf{z})$. During the training, the approximate inference network, i.e., encoder network $q_\phi(\mathbf{z}|\mathbf{x})$, is then used to obtain $\mathbf{z}$ and $p_\theta(\mathbf{x}|\mathbf{z})$ is then viewed as a decoder network. The core idea of variational autoenoder is that they are trained by maximizing the variational lower bound $\mathcal{L}(\theta, \phi; \mathbf{x})$:

$$\mathcal{L}(\theta, \phi; \mathbf{x}) = -D_{KL}(q_\phi(\mathbf{z}|\mathbf{x})||p_\theta(\mathbf{z})) + \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}[\log p_\theta(\mathbf{x}|\mathbf{z})] \leq \log p_\theta(\mathbf{x}) \tag{11.31}$$

where $D_{KL}(q_\phi(\mathbf{z}|\mathbf{x})||p_\theta(\mathbf{z}))$ is the KL divergence which measures the similarity of two distributions $q_\phi(\mathbf{z}|\mathbf{x})$ and $p_\theta(\mathbf{z})$. ϕ and θ are the variational parameters and generative parameters, respectively. We want to differentiate and optimize the lower bound w.r.t both the ϕ and θ. However, directly using a gradient estimator for the above objective function will exhibit high variance. Therefore, VAE adopts the reparametric trick. That is, under certain mild conditions for a chosen approximate posterior $q_\phi(\mathbf{z}|\mathbf{x})$, the random variable $\tilde{z} \sim q_\phi(\mathbf{z}|\mathbf{x})$ can be parameterized as

$$\tilde{z} = g_\phi(\epsilon, \mathbf{x}) \quad \text{with} \quad \epsilon \sim p(\epsilon) \tag{11.32}$$

where $g_\phi(\epsilon, \mathbf{x})$ is a differentiable transformation function of a noise variable ϵ. The nonparametric trick is also shown in Figure 11.10. With this technique, the variational lower bound in Eq.(11.31) can be approximated as

$$\begin{aligned} L^A(\theta, \phi; \mathbf{x}) &= \frac{1}{L} \sum_{l=1}^{L} \log p_\theta(\mathbf{x}, \mathbf{z}^{(l)}) - \log q_\phi(\mathbf{z}^{(l)}|\mathbf{x}) \\ \tilde{z}^{(l)} &= g_\phi(\epsilon^{(l)}, \mathbf{x}) \quad \text{with} \quad \epsilon^{(l)} \sim p(\epsilon). \end{aligned} \tag{11.33}$$

Then the parameters can be learned via stochastic gradient descent efficiently. It is easy to see that the encoder is a feature extractor which learns latent representations for $\mathbf{x}$.

11.7 Discussion and Further Readings

We have introduced representative deep learning models for feature engineering. In this section, we'll discuss how they can be used for hierarchical representation learning and disentangled representation, and how they can be used for popular domains such as text, image and graph.

Table 11.1: Hierarchical and disentangled representation learning

Method	Hierarchical Fea. Rep.	Disentangled Fea. Rep.
RBM/DBM	[59]	[48]
DBN	[33]	N/A
AE/DAE/SDAE	[37]	[28]
RNN/GRU/LSTM	[68, 73]	[11, 54]
CNN	[12, 14]	[25, 49]
GAN	[47]	[6, 38]
VAE	[75]	[54, 72]

Hierarchical Representation Generally, hierarchical representation lets us learn features of hierarchy and combine top-down and bottom-up processing of an image (or text). For instance, lower layers could support object detection by spotting low-level features indicative of object parts. Conversely, information about objects in the higher layers could resolve lower-level ambiguities in the image or infer the locations of hidden object parts. Features at different hierarchy levels may be good for different tasks. The high-level features captures the main objects, resolve ambiguities, and thus are good for classification, while mid-level features include many details and may be good for segmentation. Hierarchical feature representation is very common in deep learning models. We list some representative literature on how the introduced model can be used for hierarchical feature learning in Table 11.1.

Disentangled Representation Disentangled representation is a popular way to learn explainable representations. The majority of existing representation learning frameworks learn representation $\mathbf{h} \in \mathbb{R}^{d\times 1}$, which is difficult to explain, i.e., the d-latent dimensions are entangled together and we don't know the semantic meaning of the d-dimensions. Instead, disentangled representation learning tries to disentangle the latent factors so we know the semantic meaning of the latent dimensions. For example, for handwritten digits such as the MNIST dataset, we may want to disentangle the digit shape from writing style so that some part of $\mathbf{h}$ controls digit shapes while the other part represents writing style. The disentangled representation not only explains latent representation but also helps generate controlled realistic images. For example, by changing the part of codes that controls digit shape, we can generate new images of target digit shapes using a generator with this new latent representation. Therefore, disentangled representation learning is attracting increasing attention. Table 11.1 also lists some representative deep learning methods for disentangled representation learning. This is still a relatively new direction that needs further investigation.

Table 11.2: Deep learning methods for different domains

Method	Text	Image	Audio	Linked Data) (Graph)
RBM/DBM	[57, 58]	[57]	[9]	[67]
DBN	[52]	[33]	[42]	N/A
AE/DAE/SDAE	[3]	[63]	[44]	[64]
CNN	[29]	[19, 45, 71]	[1]	[10]
Word2Vec	[35, 41]	N/A	N/A	[61, 66]
RNN/GRU/LSTM	[27, 40, 60]	[2]	[17, 50]	N/A
GAN	[74]	[6, 47]	[74]	N/A
VAE	[5, 26]	[18, 30]	[24]	N/A

Deep Feature Representation for Various Domains Many deep learning algorithms were initially developed for specific domains. For example, the CNN was initially developed for image processing and Word2Vec was initially proposed for learning word representation. Due to the great success of these methods, they were further developed to be applicable to other domains. For example, in addition to images, the CNN has also been successfully applied on texts, audio and linked data. Each domain has its own unique property. For example, text data are inherently sequential and graph data are non-i.i.d. Thus, directly applying CNN is impractical. New architectures are proposed to adapt the CNN for these domains. The same holds for the other deep learning algorithms. Therefore, we summarize the application of the discussed deep learning models in four domains in Table 11.2. We encourage interested users to read these papers for further understanding.

Combining Deep Learning Models We have introduced various deep learning models, which can be applied to different domains. For example, LSTM is mainly used for dealing with sequential data such as texts while the CNN is powerful for images. It is very common that we need to work on tasks which are related to different domains. In such cases, we can combine different deep learning models to propose a new framework that can be applied for the task at hand. For example, for an information retrieval task given a text query, we want to retrieve images that match the query. We will need to use LSTM or Word2Vec to learn a representation that captures the semantic meanings of the query. At the same time, we need to use CNN to learn features that describe the image. We can then train LSTM and CNN so that the similarity of the representations for the matched query-image pairs are maximized while the similarity of the representations for the non-matched query-image pairs are minimized. Another example is video action recognition, where we want to classify the action of the video. Since the video is composed of frames and nearby frames have dependency, a video is inherently sequential data and LSTM is a good fit for modeling such data. However, LSTM is not good at extracting images. Therefore, we will first need to use CNN to extract features from each frame of the video, which are then used as input to LSTM for learning representation of the video [68]. Similarly, for image captioning, we can use CNN to extract features and use LSTM to generate image captions based on the image features [71]. We just list a few examples and there are many other examples. In general, we can treat deep learning algorithms as feature extracting tools that can extract features from certain domains. We can then design a loss function on top of these deep learning algorithms for the problem we want to study. One thing to note is that when we combine different models together, they are trained end-to-end. In other words, we don't train these models separately. Instead, we treat the new model as a unified model. This usually gives better performance than training each model separately and then combining them.

Bibliography

[1] Ossama Abdel-Hamid, Abdel-Rahman Mohamed, Hui Jiang, Li Deng, Gerald Penn, and Dong Yu. Convolutional neural networks for speech recognition. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 22(10):1533–1545, 2014.

[2] Stanislaw Antol, Aishwarya Agrawal, Jiasen Lu, Margaret Mitchell, Dhruv Batra, C. Lawrence Zitnick, and Devi Parikh. VQA: Visual question answering. In *CVPR*, pages 2425–2433, 2015.

[3] Sarath Chandar AP, Stanislas Lauly, Hugo Larochelle, Mitesh Khapra, Balaraman Ravindran, Vikas C Raykar, and Amrita Saha. An autoencoder approach to learning bilingual word representations. In *NIPS*, pages 1853–1861, 2014.

[4] Yoshua Bengio et al. Learning deep architectures for AI. *Foundations and Trends in Machine Learning*, 2(1):1–127, 2009.

[5] Samuel R Bowman, Luke Vilnis, Oriol Vinyals, Andrew M Dai, Rafal Jozefowicz, and Samy Bengio. Generating sentences from a continuous space. *CoNLL*, 2016.

[6] Xi Chen, Yan Duan, Rein Houthooft, John Schulman, Ilya Sutskever, and Pieter Abbeel. Infogan: Interpretable representation learning by information maximizing generative adversarial nets. In *NIPS*, pages 2172–2180, 2016.

[7] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[8] Aaron Courville, James Bergstra, and Yoshua Bengio. A spike and slab restricted boltzmann machine. In *AISTAS*, pages 233–241, 2011.

[9] George Dahl, Abdel-Rahman Mohamed, Geoffrey E Hinton, et al. Phone recognition with the mean-covariance restricted Boltzmann machine. In *NIPS*, pages 469–477, 2010.

[10] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. In *NIPS*, pages 3844–3852, 2016.

[11] Emily Denton and Vighnesh Birodkar. Unsupervised learning of disentangled representations from video. pages 4417–4426, NIPS 2017.

[12] Clement Farabet, Camille Couprie, Laurent Najman, and Yann LeCun. Learning hierarchical features for scene labeling. *IEEE TPAMI*, 35(8):1915–1929, 2013.

[13] John R Firth. *A Synopsis of Linguistic Theory*, 1930-1955. 1957.

[14] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *CVPR*, pages 580–587, 2014.

[15] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[16] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *NIPS*, pages 2672–2680, 2014.

[17] Alex Graves, Abdel-Rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. In *ICASSP*, pages 6645–6649. IEEE, 2013.

[18] Karol Gregor, Ivo Danihelka, Alex Graves, Danilo Jimenez Rezende, and Daan Wierstra. Draw: A recurrent neural network for image generation. *arXiv preprint arXiv:1502.04623*, 2015.

[19] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, pages 770–778, 2016.

[20] Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-Rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N Sainath, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, 29(6):82–97, 2012.

[21] Geoffrey E Hinton. Deep belief networks. *Scholarpedia*, 4(5):5947, 2009.

[22] Geoffrey E Hinton et al. Modeling pixel means and covariances using factorized third-order Boltzmann machines. In *CVPR*, pages 2551–2558. IEEE, 2010.

[23] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[24] Wei-Ning Hsu, Yu Zhang, and James R. Glass. Learning latent representations for speech generation and transformation. Annual Conference of the International Speech Communication Association, (INTERSPEECH). pages 1273–1277, 2017.

[25] Wei-Ning Hsu, Yu Zhang, and James R. Glass. Unsupervised learning of disentangled and interpretable representations from sequential data. In *NIPS*, 2017.

[26] Zhiting Hu, Zichao Yang, Xiaodan Liang, Ruslan Salakhutdinov, and Eric P Xing. Toward controllable text generation. In *ICML*, 2017.

[27] Ozan Irsoy and Claire Cardie. Opinion mining with deep recurrent neural networks. In *EMNLP*, pages 720–728, 2014.

[28] Michael Janner, Jiajun Wu, Tejas Kulkarn, Ilker Yildirim, and Josh Tenenbaum. Learning to generalize intrinsic images with a structured disentangling autoencoder. In *NIPS*, 2017.

[29] Nal Kalchbrenner, Edward Grefenstette, and Phil Blunsom. A convolutional neural network for modelling sentences. In *ACL*, 2014.

[30] Diederik P Kingma and Max Welling. Auto-encoding variational Bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[31] Alex Krizhevsky, Geoffrey E Hinton, et al. Factored 3-way restricted Boltzmann machines for modeling natural images. In *AISTATS*, pages 621–628, 2010.

[32] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *NIPS*, pages 1097–1105, 2012.

[33] Honglak Lee, Roger Grosse, Rajesh Ranganath, and Andrew Y Ng. Unsupervised learning of hierarchical representations with convolutional deep belief networks. *Communications of the ACM*, 54(10):95–103, 2011.

[34] Omer Levy and Yoav Goldberg. Neural word embedding as implicit matrix factorization. In *NIPS*, pages 2177–2185, 2014.

[35] Yang Li, Quan Pan, Tao Yang, Suhang Wang, Jiliang Tang, and Erik Cambria. Learning word representations for sentiment analysis. *Cognitive Computation*, 2017.

[36] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. In *CVPR*, pages 3431–3440, 2015.

[37] Jonathan Masci, Ueli Meier, Dan Cireşan, and Jürgen Schmidhuber. Stacked convolutional auto-encoders for hierarchical feature extraction. *Artificial Neural Networks and Machine Learning–ICANN 2011*, pages 52–59, 2011.

[38] Michaël Mathieu, Junbo Jake Zhao, Pablo Sprechmann, Aditya Ramesh, and Yann LeCun. Disentangling factors of variation in deep representation using adversarial training. In *NIPS*, pages 5041–5049, 2016.

[39] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[40] Tomas Mikolov, Martin Karafiát, Lukás Burget, Jan Cernocký, and Sanjeev Khudanpur. Recurrent neural network based language model. In *INTERSPEECH*, pages 1045–1048, 2010.

[41] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *NIPS*, pages 3111–3119, 2013.

[42] Abdel-Rahman Mohamed, George Dahl, and Geoffrey Hinton. Deep belief networks for phone recognition. In *NIPS Workshop on Deep Learning for Speech Recognition and Related Applications*, 2009.

[43] Andrew Ng. Sparse autoencoder. *CS294A Lecture Notes*, 72(2011):1–19, 2011.

[44] Jiquan Ngiam, Aditya Khosla, Mingyu Kim, Juhan Nam, Honglak Lee, and Andrew Y Ng. Multimodal deep learning. In *ICML*, pages 689–696, 2011.

[45] Maxime Oquab, Leon Bottou, Ivan Laptev, and Josef Sivic. Learning and transferring mid-level image representations using convolutional neural networks. In *CVPR*, pages 1717–1724, 2014.

[46] Jeffrey Pennington, Richard Socher, and Christopher D Manning. Glove: Global vectors for word representation. In *EMNLP*, volume 14, pages 1532–1543, 2014.

[47] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*, 2015.

[48] Scott Reed, Kihyuk Sohn, Yuting Zhang, and Honglak Lee. Learning to disentangle factors of variation with manifold interaction. In *ICML*, pages 1431–1439, 2014.

[49] Salah Rifai, Yoshua Bengio, Aaron Courville, Pascal Vincent, and Mehdi Mirza. Disentangling factors of variation for facial expression recognition. *ECCV*, pages 808–822, 2012.

[50] Haşim Sak, Andrew Senior, and Françoise Beaufays. Long short-term memory recurrent neural network architectures for large scale acoustic modeling. In *Fifteenth Annual Conference of the International Speech Communication Association*, 2014.

[51] Ruslan Salakhutdinov and Geoffrey Hinton. Deep Boltzmann machines. In *Artificial Intelligence and Statistics*, pages 448–455, 2009.

[52] Ruhi Sarikaya, Geoffrey E. Hinton, and Anoop Deoras. Application of deep belief networks for natural language understanding. *IEEE/ACM Trans. Audio, Speech & Language Processing*, 22(4):778–784, 2014.

[53] Ali Sharif Razavian, Hossein Azizpour, Josephine Sullivan, and Stefan Carlsson. CNN features off-the-shelf: an astounding baseline for recognition. In *CVPR Workshops*, pages 806–813, 2014.

[54] N. Siddharth, Brooks Paige, Jan-Willem van de Meent, Alban Desmaison, Frank Wood, Noah D. Goodman, Pushmeet Kohli, and Philip H. S. Torr. Learning disentangled representations with semi-supervised deep generative models. In *NIPS*, 2017.

[55] Karen Simonyan and Andrew Zisserman. Two-stream convolutional networks for action recognition in videos. In *NIPS*, pages 568–576, 2014.

[56] Nitish Srivastava, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[57] Nitish Srivastava and Ruslan R Salakhutdinov. Multimodal learning with deep Boltzmann machines. In *NIPS*, pages 2222–2230, 2012.

[58] Nitish Srivastava, Ruslan R Salakhutdinov, and Geoffrey E Hinton. Modeling documents with deep Boltzmann machines. In *UAI*, 2013.

[59] Heung-Il Suk, Seong-Whan Lee, Dinggang Shen, Alzheimer's Disease Neuroimaging Initiative, et al. Hierarchical feature representation and multimodal fusion with deep learning for AD/MCI diagnosis. *NeuroImage*, 101:569–582, 2014.

[60] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *NIPS*, pages 3104–3112, 2014.

[61] Jian Tang, Meng Qu, Mingzhe Wang, Ming Zhang, Jun Yan, and Qiaozhu Mei. Line: Large-scale information network embedding. In *WWW*, pages 1067–1077, 2015.

[62] Tijmen Tieleman. Training restricted Boltzmann machines using approximations to the likelihood gradient. In *ICML*, pages 1064–1071. ACM, 2008.

[63] Pascal Vincent, Hugo Larochelle, Isabelle Lajoie, Yoshua Bengio, and Pierre-Antoine Manzagol. Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *Journal of Machine Learning Research*, 11(Dec):3371–3408, 2010.

[64] Daixin Wang, Peng Cui, and Wenwu Zhu. Structural deep network embedding. In *SIGKDD*, pages 1225–1234. ACM, 2016.

[65] Jun Wang, Lantao Yu, Weinan Zhang, Yu Gong, Yinghui Xu, Benyou Wang, Peng Zhang, and Dell Zhang. Irgan: A minimax game for unifying generative and discriminative information retrieval models. In *SIGIR*, 2017.

[66] Suhang Wang, Jiliang Tang, Charu Aggarwal, and Huan Liu. Linked document embedding for classification. In *CIKM*, pages 115–124. ACM, 2016.

[67] Suhang Wang, Jiliang Tang, Fred Morstatter, and Huan Liu. Paired restricted Boltzmann machine for linked data. In *CIKM*, pages 1753–1762. ACM, 2016.

[68] Yilin Wang, Suhang Wang, Jiliang Tang, Neil O'Hare, Yi Chang, and Baoxin Li. Hierarchical attention network for action recognition in videos. *CoRR*, abs/1607.06416, 2016.

[69] Max Welling, Michal Rosen-Zvi, and Geoffrey E Hinton. Exponential family harmoniums with an application to information retrieval. In *NIPS*, volume 4, pages 1481–1488, 2004.

[70] Paul J Werbos. Backpropagation through time: What it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.

[71] Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhudinov, Rich Zemel, and Yoshua Bengio. Show, attend and tell: Neural image caption generation with visual attention. In *ICML*, pages 2048–2057, 2015.

[72] Xinchen Yan, Jimei Yang, Kihyuk Sohn, and Honglak Lee. Attribute2image: Conditional image generation from visual attributes. In *ECCV*, pages 776–791. Springer, 2016.

[73] Zichao Yang, Diyi Yang, Chris Dyer, Xiaodong He, Alexander J Smola, and Eduard H Hovy. Hierarchical attention networks for document classification. In *HLT-NAACL*, pages 1480–1489, 2016.

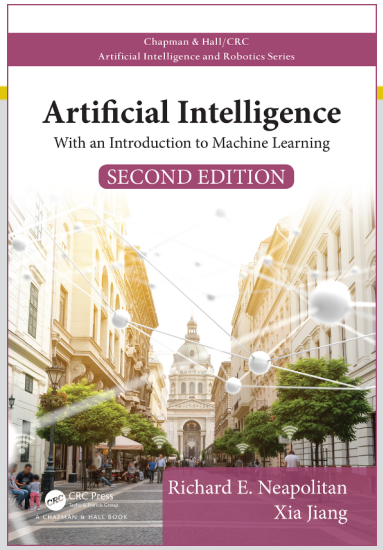
[74] Lantao Yu, Weinan Zhang, Jun Wang, and Yong Yu. Seqgan: Sequence generative adversarial nets with policy gradient. In *AAAI*, pages 2852–2858, 2017.

[75] Shengjia Zhao, Jiaming Song, and Stefano Ermon. Learning hierarchical features from deep generative models. In *ICML*, pages 4091–4099, 2017.

CHAPTER

6

NEURAL NETWORKS AND DEEP LEARNING



This chapter is excerpted from

Artificial Intelligence: With and Introduction to Machine Learning, Second Edition

by Richard E. Neapolitan and Xia Jiang.

© 2018 Taylor & Francis Group. All rights reserved.

Learn more

Neural Networks and Deep Learning

The previous three parts modeled intelligence at either a human cognitive level or at a population-based level. The intelligence is removed from the physiological processes involved in intelligent reasoning. In this part, we model the neuronal processes involved when the brain is "intelligently" controlling the thoughts and behavior of a life form. The networks we construct in this fashion are called **artificial neural networks**. Neural networks have been used effectively in applications such as image recognition and speech recognition, which are hard to model with the structured approach used in rule-based systems and Bayesian networks. In the case of image recognition, for example, they learn to identify images of cars by being presented with images that have been labeled "car" and "no car". We start by modeling a single neuron.

15.1 The Perceptron

Figure 15.1 (a) shows a biological neuron, which has dendrites that transmit signals to a cell body, a cell body that processes the signal, and an axon that sends signals out to other

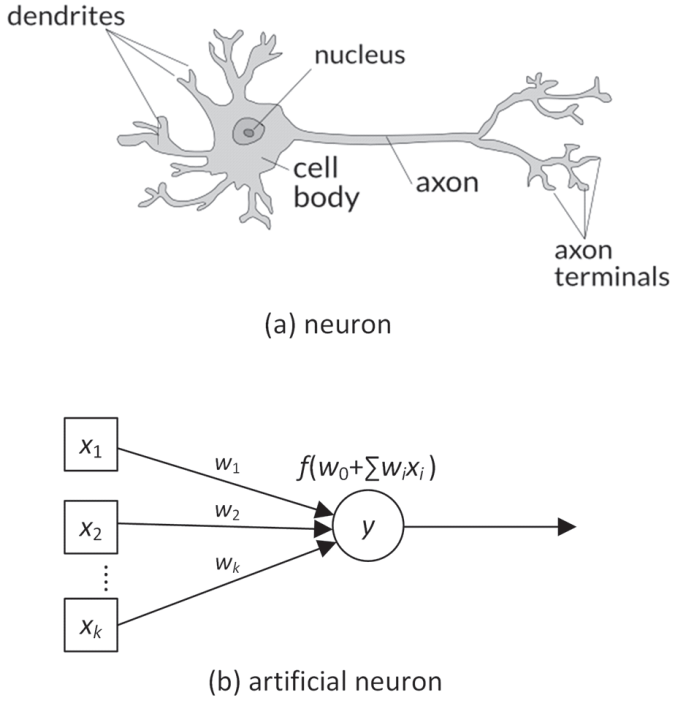


Figure 15.1 A neuron is in (a); an artficial neuron is in (b).

neurons. The input signals are accumulated in the cell body of the neuron, and if the accumulated signal exceeds a certain threshold, an output signal is generated which is passed on by the axon. Figure 15.1 (b) show an artificial neuron that mimics this process. The artificial neuron takes as input a vector $(x_1, x_2, ..., x_k)$, and then applies **weights** $(w_0, w_1, w_2, ..., w_k)$ to that input yielding a weighted sum:

$$w_0 + \sum_{i=1}^{k} w_i x_i.$$

Next the neuron applies an activation function f to that sum, and outputs the value y of f. Note that the inputs x_i are square nodes to distinguish them from an artificial neuron, which is a computational unit.

A **neural network** consists of one to many artificial neurons, which communicate with each other. The output of one neuron is the input to another neuron. The simplest neural network is the **perceptron**, which consists of a single artificial neuron, as shown in Figure 15.1 (b). The activation function for the perceptron is as follows:

$$f(z) = \begin{cases} 1 & \text{if } z > 0 \\ -1 & \text{otherwise} \end{cases}$$

Therefore, the complete expression for the output y of the perceptron is as follows:

$$y = \begin{cases} 1 & \text{if } w_0 + \sum_{i=1}^{k} w_i x_i > 0 \\ -1 & \text{otherwise} \end{cases} \tag{15.1}$$

The perceptron is a binary classifier. It returns 1 if the activation function exceeds 0; otherwise it returns -1.

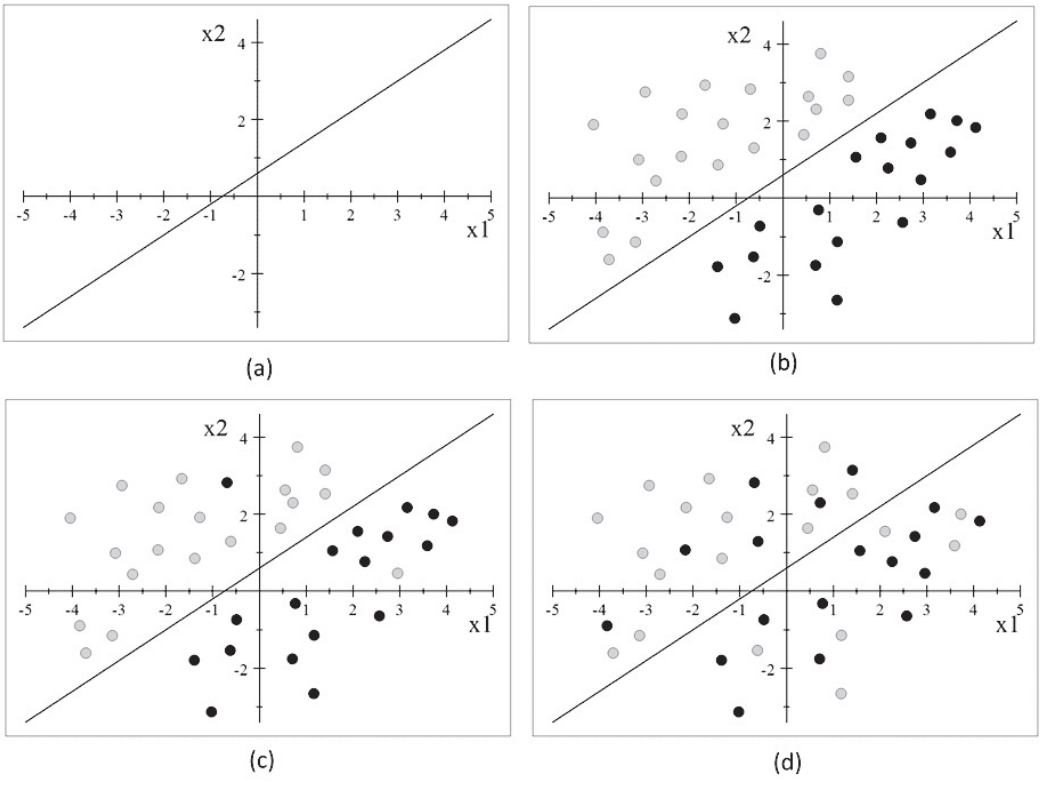


Figure 15.2 The line $-3 - 4x_1 + 5x_2 = 0$ is in (a). This line linearly separates the data in (b), and approximately linearly separates the data in (c). It does not approximately linearly separate the data in (d).

Let's look at the case where the input is a two-dimensional vector (x_1, x_2). Suppose the weighted sum in our perceptron is as follows:

$$w_0 + w_1 x_1 + w_2 x_2 = -3 - 4x_1 + 5x_2. \tag{15.2}$$

Figure 15.2 (a) plots the line $-3 - 4x_1 + 5x_2 = 0$. If we color 2-dimensional points gray or black, the set of gray points is **linearly separable** from the set of black points if there exists at least one line in the plane with all the gray points on one side of the line and all the black points on the other side. This definition extends readily to higher-dimensional data. The points in Figure 15.2 (b) are linearly separable by the line $-3 - 4x_1 + 5x_2 = 0$. So, the perceptron with the weights in Equality 15.2 maps all the gray points to $y = 1$, and all the black points to $y = -1$. This perceptron is a perfect binary classifier for these data. If we have the data in Figure 15.2 (c), this perceptron is a pretty good classifier, as it only misclassifies two cases. It is a very poor classifier for the data in Figure 15.2 (d). The gray and black points in that figure are not approximately linearly separable; so no perceptron would be a good classifier for this data. The perceptron is a **linear binary classifier** because it uses a linear function to classify an instance.

15.1.1 Learning the Weights for a Perceptron

When learning a perceptron for binary classification, our goal is to determine weights determining a line that as close as possible linearly separates the two classes. Next we develop a

gradient descent algorithm for learning these weights (See Section 5.3.2 for an introduction to gradient descent.) If we have a data item $(x_1, x_2, ..., x_k, y)$, our loss function is

$$Loss(y, \hat{y}) = (\hat{y} - y)(w_0 + \sum_{i=1}^{k} w_i x_i),$$

where $\hat{y}$ is the estimate of y using Equation 15.1. The idea behind this loss function is that if $\hat{y} = y$ there is no loss. If $\hat{y} = 1$ and $y = -1$, then $w_0 + \sum_{i=1}^{k} w_i x_i > 1$, and the loss is $2\left(w_0 + \sum_{i=1}^{k} w_i x_i\right)$. This loss is a measure of how far off we are from obtaining a value < 0, which would have given a correct answer. Similarly, if $\hat{y} = -1$ and $y = 1$, then $w_0 + \sum_{i=1}^{k} w_i x_i < 1$, and the loss is again $2\left(w_0 + \sum_{i=1}^{k} w_i x_i\right)$. The cost function follows:

$$Cost([y^1, \hat{y}^1], ..., [y^n, \hat{y}^n]) = \sum_{j=1}^{n} Loss(y^j, \hat{y}^j) = \sum_{j=1}^{n} \left((\hat{y}^j - y^j)(w_0 + \sum_{i=1}^{k} w_i x_i^j) \right). \quad (15.3)$$

Note that x_i^j denotes the ith vector element in the jth data item. This is different from the notation used in Section 5.3. The partial derivatives of the cost function are as follows:

$$\frac{\partial \left(\sum_{j=1}^{n} (\hat{y}^j - y^j)(w_0 + \sum_{i=1}^{k} w_i x_i^j) \right)}{\partial w_0} = \sum_{j=1}^{n} (\hat{y}^j - y^j)$$

$$\frac{\partial \sum_{j=1}^{n} \left((\hat{y}^j - y^j)(w_0 + \sum_{i=1}^{k} w_i x_i^j) \right)}{\partial w_m} = \sum_{j=1}^{n} (\hat{y}^j - y^j) x_m^j.$$

When Rosenblatt [1958] developed the perceptron and the algorithm we are presenting, he updated based on each item in sequence as in stochastic gradient descent (Section 5.3.4). We show that version of the algorithm next.

Algorithm 15.1 Gradient_Descent_Perceptron

Input: Set of real predictor data and binary outcome data: $\{(x_1^1 x_2^1, ...x_k^1, y^1)$, $(x_1^2 x_2^2, ...x_k^2, y^2)..., (x_1^n x_2^n, ...x_k^n, y^n)\}$.

Output: Weights $w_0, w_1, ..., w_k$ that minimize the cost function in Equality 15.3.

Function *Minimizing_Values*;
for $i = 0$ to k
    $w_i = arbitrary_value$;
endfor
$\lambda = learning_rate$;
repeat *number_iterations* times
    for $j = 1$ to n
        $y = \begin{cases} 1 \text{ if } w_0 + \sum_{i=1}^{k} w_i x_i^j > 0 \\ -1 \text{ otherwise} \end{cases}$
        $w_0 = w_0 - \lambda(y - y^j)$;
        for $m = 1$ to k
            $w_m = w_m - \lambda(y - y^j) x_m^j$;
        endfor
    endfor
endrepeat

Example 15.1 Suppose we set $\lambda = 0, 1$, and we have the following data:

x_1	x_2	y
1	2	1
3	4	-1

The algorithm through 2 iterations of the repeat loop follows:

// Initialize weights to arbitrary values.
$w_0 = 1; w_1 = 1, w_2 = 1;$

// First iteration of repeat loop.

// $j = 1$ in the for$-j$ loop.
$w_0 + w_1 x_1^1 + w_2 x_2^1 = 1 + 1(1) + 1(2) = 4 > 0;$
$y = 1;$
$w_0 = w_0 - \lambda(y - y^1) = 1 - (0.1)(1 - 1) = 1;$
$w_1 = w_1 - \lambda(y - y^1)x_1^1 = 1 - (0.1)(1 - 1)1 = 1;$
$w_2 = w_2 - \lambda(y - y^1)x_2^1 = 1 - (0.1)(1 - 1)2 = 1;$

// $j = 2$ in the for$-j$ loop.
$w_0 + w_1 x_1^2 + w_2 x_2^2 = 1 + 1(3) + 1(4) = 8 > 0;$
$y = 1;$
$w_0 = w_0 - \lambda(y - y^1) = 1 - (0.1)(1 - (-1)) = 0.8;$
$w_1 = w_1 - \lambda(y - y^1)x_1^1 = 1 - (0.1)(1 - (-1))3 = 0.4;$
$w_2 = w_2 - \lambda(y - y^1)x_2^1 = 1 - (0.1)(1 - (-1))4 = 0.2;$

// Second iteration of repeat loop.

// $j = 1$ in the for$-j$ loop.
$w_0 + w_1 x_1^1 + w_2 x_2^1 = 0.8 + 0.4(1) + 0.2(2) = 1.6 > 0;$
$y = 1;$
$w_0 = w_0 - \lambda(y - y^1) = 0.8 - (0.1)(1 - 1) = 0.8;$
$w_1 = w_1 - \lambda(y - y^1)x_1^1 = 0.4 - (0.1)(1 - 1)1 = 0.4;$
$w_2 = w_2 - \lambda(y - y^1)x_2^1 = 0.2 - (0.1)(1 - 1)2 = 0.2;$

// $j = 2$ in the for$-j$ loop.
$w_0 + w_1 x_1^2 + w_2 x_2^2 = 0.8 + 0.4(3) + 0.2(4) = 2.8 > 0;$
$y = 1;$
$w_0 = w_0 - \lambda(y - y^1) = 0.8 - (0.1)(1 - (-1)) = 0.6;$
$w_1 = w_1 - \lambda(y - y^1)x_1^1 = 0.4 - (0.1)(1 - (-1))3 = -0.2;$
$w_2 = w_2 - \lambda(y - y^1)x_2^1 = 0.2 - (0.1)(1 - (-1))4 = -0.6;$

Table 15.1 SAT Scores, Parental Income, and Graduation Status for 12 Students

SAT (100)	Income ($10,000)	Graduate
4	18	no
6	7	no
8	4	no
10	6	no
12	2	no
10	10	no
6	6	yes
7	20	yes
8	16	yes
12	16	yes
14	7	yes
16	4	yes

15.1.2 The Perceptron and Logistic Regression

The perceptron is similar to logistic regression (See Section 5.3.3) in that they both map continuous predictors to a binary outcome. The difference is that the perceptron deterministically reports that $Y = 1$ or $Y = -1$, while logistic regression reports the probability that $Y = 1$. Recall this logistic regression computes this probability as follows:

$$P(Y = 1|\mathbf{x}) = \frac{\exp(b_0 + \sum_{i=1}^{k} b_i x_i)}{1 + \exp(b_0 + \sum_{i=1}^{k} b_i x_i)}$$
$$P(Y = -1|\mathbf{x}) = \frac{1}{1 + \exp(\sum_{i=1}^{k} b_i x_i)}.$$

We can use the logistic regression equation as a binary classifier if we say $Y = 1$ if and only if $P(Y = 1) > P(Y = -1)$. The following sequence of steps shows that if we do this, we have a linear binary classifier.

$$\begin{aligned}
P(Y = 1|x) &= P(Y = -1|\mathbf{x}) \\
\frac{\exp(b_0 + \sum_{i=1}^{k} b_i x_i)}{1 + \exp(b_0 + \sum_{i=1}^{k} b_i x_i)} &= \frac{1}{1 + \exp(\sum_{i=1}^{k} b_i x_i)} \\
\exp\left(b_0 + \sum_{i=1}^{k} b_i x_i\right) &= 1 \\
\ln\left(\exp(b_0 + \sum_{i=1}^{k} b_i x_i)\right) &= 0 \\
b_0 + \sum_{i=1}^{k} b_i x_i &= 0.
\end{aligned}$$

So, we set $Y = 1$ if and only if $b_0 + \sum_{i=1}^{k} b_i x_i > 0$.

Example 15.2 Suppose we suspect that SAT scores and parental income have an affect on whether a student graduates college, and we obtain the data in Table 15.1. These data are plotted in Figure 15.3 (a). If we learn a logistic regression model from these data (using an algorithm such as the one outlined in Section 5.3.3), we obtain

$$P(Graduate = yes|SAT, Income) = \frac{\exp(-6.24 + 0.439SAT + 0.222Income)}{1 + \exp(-6.24 + 0.439SAT + 0.222Income)},$$

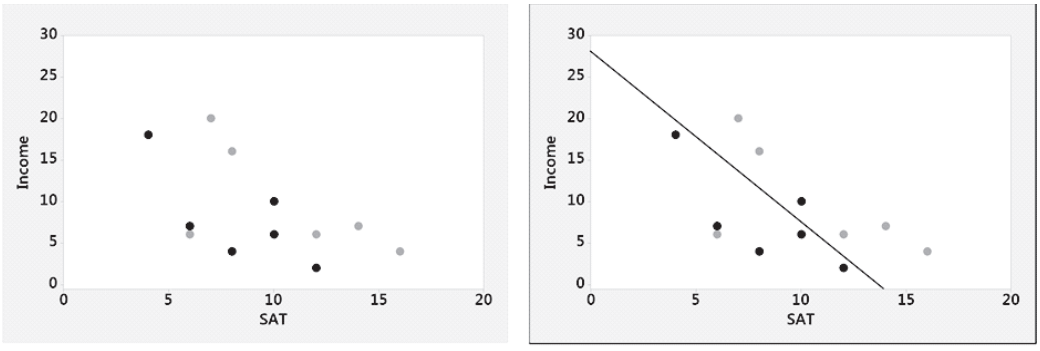


Figure 15.3 The plot in (a) shows individuals who graduated college as gray points and individuals who did not graduate college as black points. The plot in (b) shows the same individuals and includes the line $6.14 + 0.439SAT + 0.222Income$.

and so the line we obtain for a linear classifier is

$$-6.24 + 0.439SAT + 0.222Income = 0.$$

That line is plotted with the data in Figure 15.3 (b). Note that the line does not perfectly linearly separate the data, and two points are misclassified. These data are not linearly separable.

It is left as an exercise to implement Algorithm 15.1, apply it to the data in Table 15.1, and compare the results to those obtained with logistic regression.

15.2 Feedforward Neural Networks

If we want to classify the objects in Figure 15.2 (d), we need to go beyond a simple perceptron. Next we introduce more complex networks, which can classify objects that are not linearly separable. We start with a simple example, the XOR function.

15.2.1 Modeling XOR

The domain of the XOR function is $\{(0,0),(0,1)(1,0),(1,1)\}$. The XOR mapping is then as follows:

$$\begin{aligned} XOR(0,0) &= 0 \\ XOR(0,1) &= 1 \\ XOR(1,0) &= 1 \\ XOR(1,1) &= 0. \end{aligned}$$

Figure 15.4 plots the domain of the XOR function, and shows points mapping to 0 as black points and points mapping to 1 as gray points. Clearly, the black and gray points are not linearly separable. So, no perceptron could model the XOR function. However, the more complex network in Figure 15.5 does model it. That network is a **2-layer neural network** because there are two layers of artificial neurons. The first layer, containing the nodes h_1 and h_2, is called a hidden layer because it represents neither input nor output. The second layer contains the single output node y. The perceptron only has this layer. Note

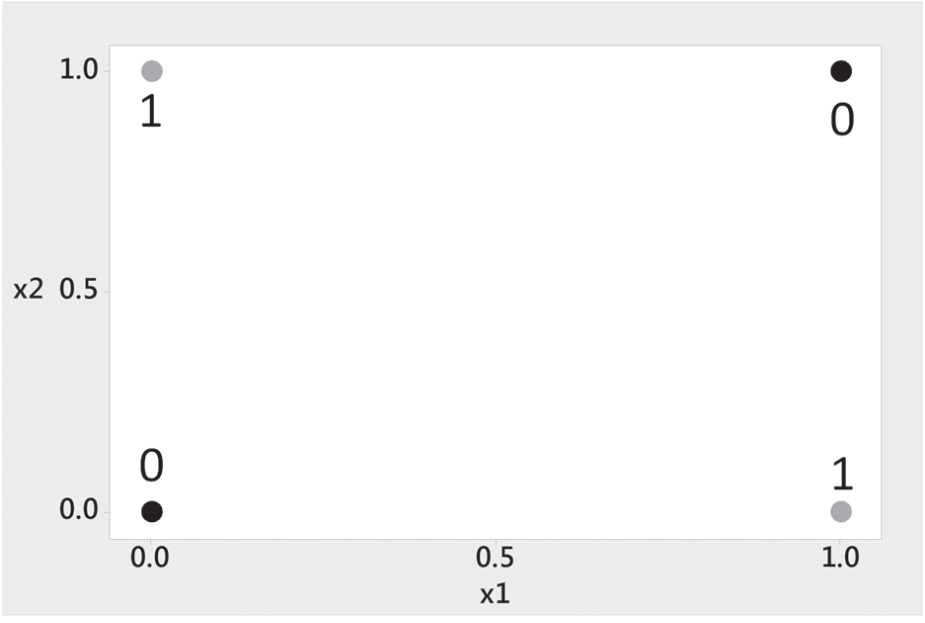


Figure 15.4 The XOR function. The black points map to 0, and the gray points map to 1.

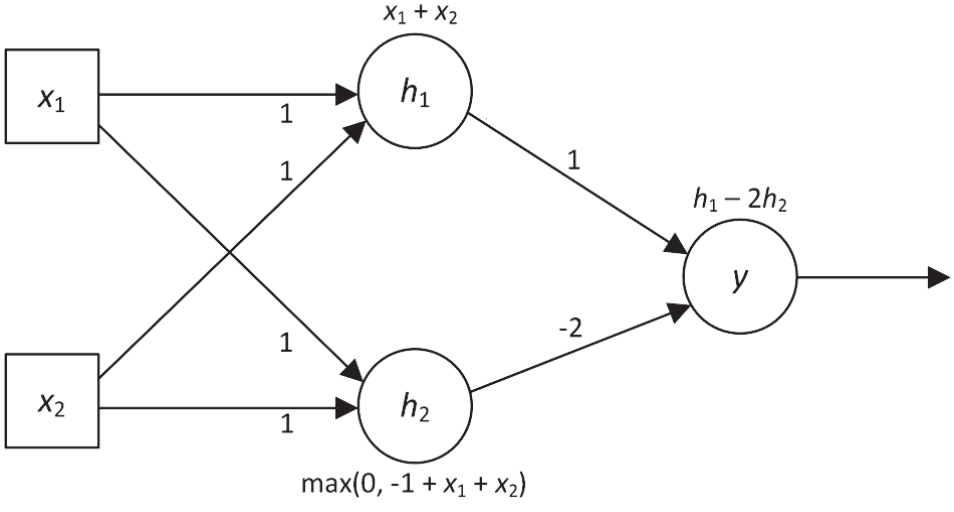


Figure 15.5 A neural network modeling the XOR function.

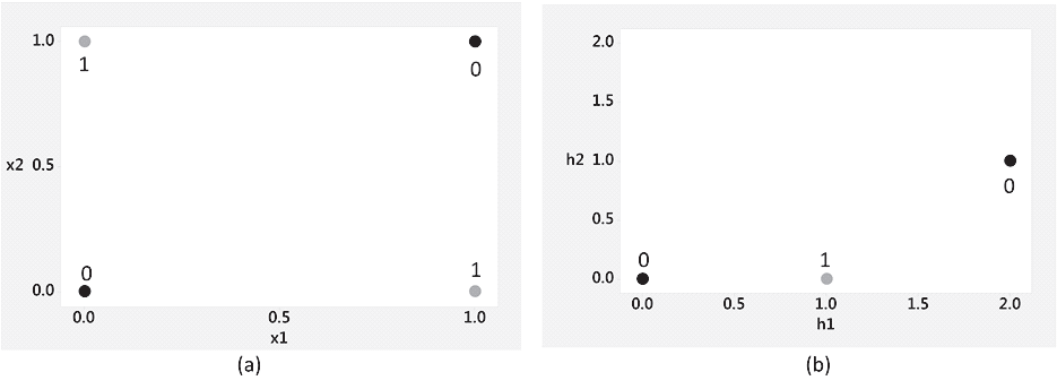


Figure 15.6 The original x-space is in (a), and the transformed h-space is in (b).

that the activation function in the hidden node h_2 in Figure 15.5 is $\max(0, z)$; this function is called the **rectified linear activation function**.

Let's show that the network in Figure 15.5 does indeed model the XOR function:

$$\begin{aligned}(x_1, x_2) &= (0,0)\\ h_1 &= 0+0=0\\ h_2 &= \max(0,-1+0+0)=0\\ y &= 0-2(0)=0\end{aligned}$$

$$\begin{aligned}(x_1, x_2) &= (0,1)\\ h_1 &= 0+1=1\\ h_2 &= \max(0,-1+0+1)=0\\ y &= 1-2(0)=1\end{aligned}$$

$$\begin{aligned}(x_1, x_2) &= (1,0)\\ h_1 &= 1+0=1\\ h_2 &= \max(0,-1+1+0)=0\\ y &= 1-2(0)=1\end{aligned}$$

$$\begin{aligned}(x_1, x_2) &= (1,1)\\ h_1 &= 1+1=2\\ h_2 &= \max(0,-1+1+1)=1\\ y &= 2-2(1)=0.\end{aligned}$$

The "trick" in this network is that h_1 and h_2 together map (0,0) to (0,0), (0,1) to (1,0), (1,0) to (1,0) and (1,1) to (2,1). So our black points are now (0,0) and (2,1), while our single gray point is (1,0). The data is now linearly separable. Figure 15.6 shows the transformation from the x-space to the h-space.

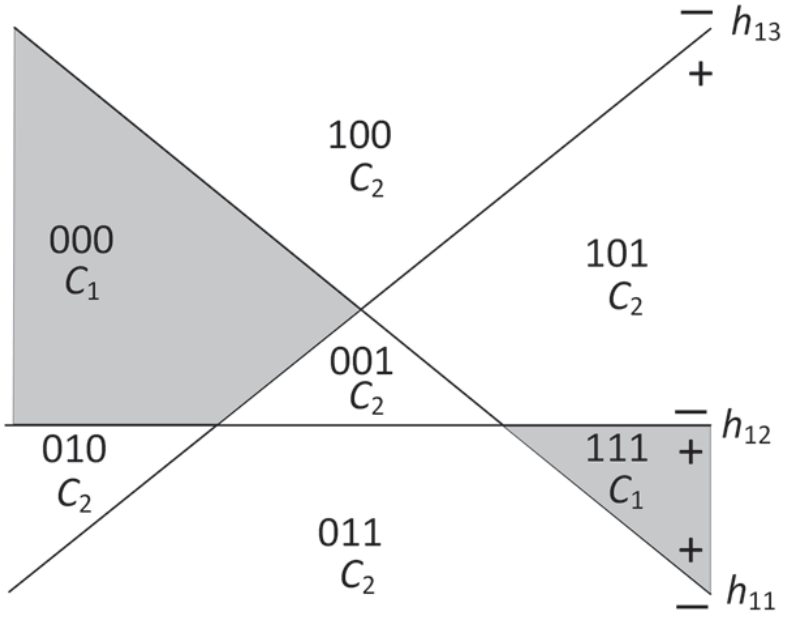


Figure 15.7 Class C_1 consists of the shaded area, and class C_2 consists of the white area. The notation 100, for example, means that the region lies on the plus side of line h_{11}, on the minus side of line h_{12}, and on the minus side of line h_{13}.

15.2.2 Example with Two Hidden Layers

Suppose we are classifying points in the plane, and all points in the grey area in Figure 15.7 are in class C_1, while all points in the white area are in class C_2. This is a difficult classification problem because the regions that comprise class C_1 are not even adjacent. The neural network in Figure 15.8, which has two hidden layers, correctly accomplishes this classification with appropriate weights and activation functions. Next, we show how this is done.

The lines h_1, h_2, and h_3 in Figure 15.7 separate the plane into 7 regions. The notation $+/-$ in Figure 15.7 indicates the region is on the $+$ side of the given line or on the $-$ side. We assign the region value 1 if it is on the $+$ side of the line and value 0 if it is on the $-$ side. Region 100 is therefore labeled as such because it is on the $+$ side of line h_{11}, on the $-$ side of line h_{12}, and on the $-$ side of line h_{13}. The other regions are labeled in a similar fashion. We can create a hidden node h_{11} with weights representing line h_{11}. Then we use an activation function that returns 0 if (x_1, x_2) is on the $-$ side of line h_{11} and 1 otherwise. We create hidden nodes h_{12} and h_{13} in a similar fashion. Table 15.2 shows the values output by each of the nodes h_{11}, h_{12}, and h_{13} when (x_1, x_2) resides in each of the 7 regions in Figure 15.7 (note that 110 does not determine a region). So, the regions map to the 7 of the corners of the unit cube in 3-space. The two points that represent class C_1 are (0,0,0) and (1,1,1). We can linearly separate point 0,0,0) from the other 6 points with a plane in 3-space. So we create a hidden node h_{21} that does this. We use an activation function that returns 1 for point (0,0,0) and 0 for all other points on the cube. Similarly, we create a hidden node h_{22} that outputs 1 for point (1,1,1) and 0 for all other points on the cube. Table 15.2 shows these values. So, the output of these two hidden nodes is (1,0)

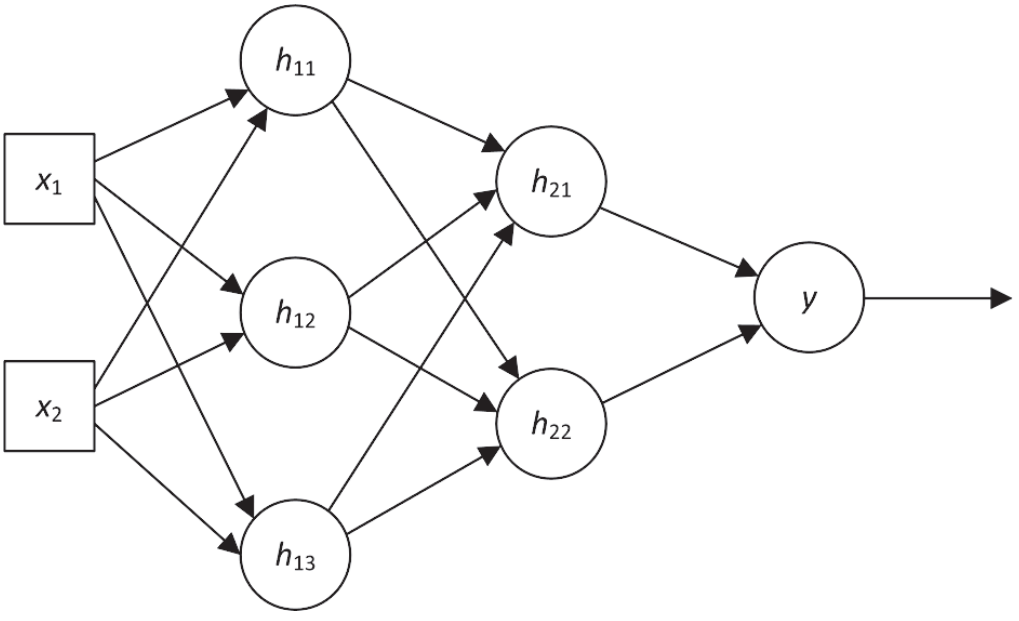


Figure 15.8 A neural network that correctly classifies points in classes C_1 and C_2 in Figure 15.7.

Table 15.2 Values of the Nodes in the Neural Network in Figure 15.7 for the Input Tuple Located in Each of the Regions in Figure 15.6

Region	Class	h_{11}	h_{12}	h_{13}	h_{21}	h_{22}	y
000	C_1	0	0	0	1	0	1
001	C_2	0	0	1	0	0	0
010	C_2	0	1	0	0	0	0
011	C_2	0	1	1	0	0	0
100	C_2	1	0	0	0	0	0
101	C_2	1	0	1	0	0	0
110	–	–	–	–	–	–	–
111	C_1	1	1	1	0	1	1

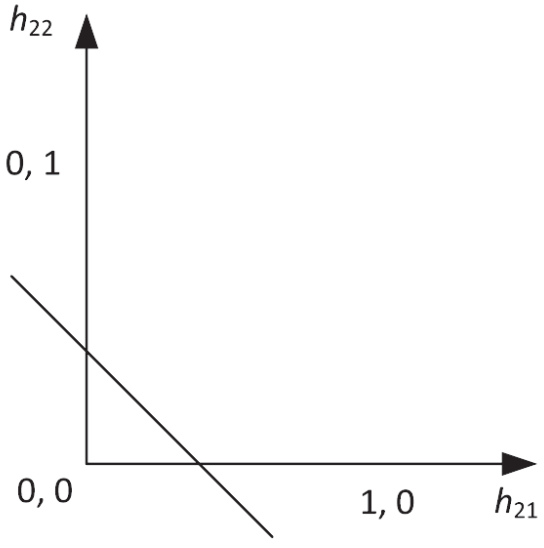


Figure 15.9 The points in region 000 map to (0,1), the points in region 111 map to (1,0), and all other points map to 0,0).

if (x_1, x_2) is in region 000 and (0,1) if (x_1, x_2) is in region 111. It is (0,0) if (x_1, x_2) is in any other region. These three points are shown in Figure 15.9. Next, we create weights for our output node y that yield a line that separates (1,0) and (0,1) from (0,0). Such a line is shown in Figure 15.9. We then use an activation function that returns 1 if the point lies above that line and 0 otherwise. In this way, all values of (x_1, x_2) in class C_1 map to 1 and all values of (x_1, x_2) in class C_2 map to 0.

Example 15.3 Suppose the three lines determining the regions in Figure 15.7 are as follows:

$$\begin{aligned} h_{11} &: 2 - x_1 - x_2 = 0 \\ h_{12} &: 0.5 - x_2 = 0 \\ h_{13} &: x_1 - x_2 = 0. \end{aligned}$$

These three lines are plotted in Figure 15.10. Given these lines, the activation functions for hidden nodes h_{11}, h_{12}, and h_{13} are as follows:

$$h_{11} = \begin{cases} 1 & \text{if } 2 - x_1 - x_2 > 0 \\ 0 & \text{otherwise} \end{cases}$$

$$h_{12} = \begin{cases} 1 & \text{if } 0.5 - x_2 < 0 \\ 0 & \text{otherwise} \end{cases}$$

$$h_{12} = \begin{cases} 1 & \text{if } x_1 - x_2 < 0 \\ 0 & \text{otherwise} \end{cases} .$$

Hidden node h_{21} must provide a plane that linearly separates $(0, 0, 0)$ from all other points on the unit cube. The following is one such plane:

$$h_{11} + h_{12} + h_{13} = 0.5.$$

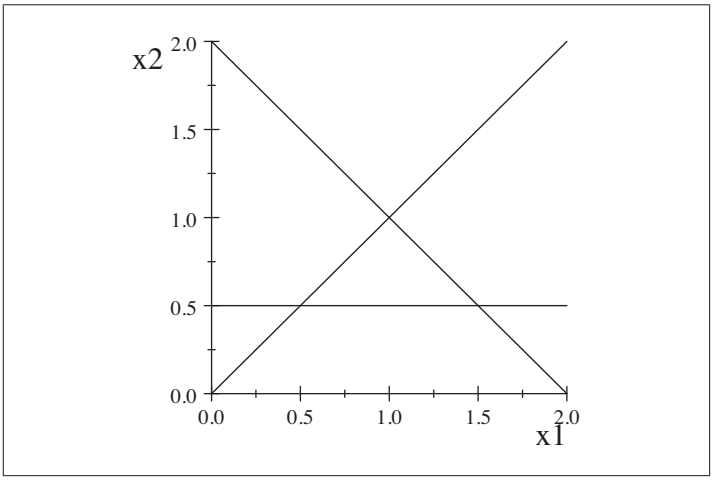


Figure 15.10 The three lines determining the regions in Figure 15.7 for Example 15.3.

So, we can make the activation function for hidden node h_{21} as follows:

$$h_{21} = \begin{cases} 1 & \text{if } h_{11} + h_{12} + h_{13} - 0.5 < 0 \\ 0 & \text{otherwise} \end{cases}$$

Hidden node h_{22} must provide a plane that linearly separates $(1, 1, 1)$ from all other points on the unit cube. The following is one such plane:

$$h_{11} + h_{12} + h_{13} = 2.5.$$

So, we can make the activation function for hidden node h_{22} as follows:

$$h_{22} = \begin{cases} 1 & \text{if } h_{11} + h_{12} + h_{13} - 2.5 > 0 \\ 0 & \text{otherwise} \end{cases} .$$

Finally node y must provide a line that linearly separates $(0, 1)$ and $(1, 0)$ from $(1, 1)$. The following is one such line:

$$h_{21} + h_{22} = 0.5.$$

So, we can make the activation function for node y as follows:

$$y = \begin{cases} 1 & \text{if } h_{21} + h_{22} - 0.5 > 0 \\ 0 & \text{otherwise} \end{cases} .$$

15.2.3 Structure of a Feedforward Neural Network

Having shown a neural network with one hidden layer (Figure 15.5) and a neural network with two hidden layers (Figure 15.8), we now present the general structure of a feedforward neural network. This structure appears in Figure 15.11. On the far left is the input, which consists of $x_1, x_2, ..., x_k$. Next there are 0 or more hidden layers. Then on the far right is the output layer, which consists of 1 or more nodes $y_1, y_2, ...y_m$. Each hidden layer can contain a different number of nodes.

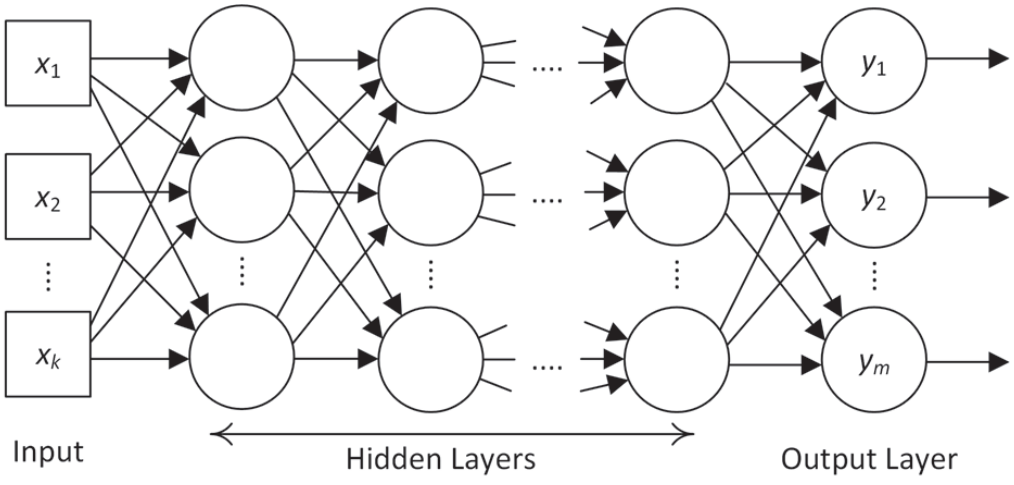


Figure 15.11 The general structure of a feedforward neural network.

In Sections 15.2.1 and 15.2.2 we constructed the neural network, and assigned values to the weights to achieve our desired results. Typically, we do not do this but rather learn the weights using a gradient descent algorithm similar to, but more complex than, the Algorithm 15.1 for the perceptron. For example, we could first construct the network in Figure 15.8 and then provide the form of the activation functions for the hidden nodes and the output node (discussed in the next section). Finally, we provide many data items to the algorithm, each of which has value $(x_1, x_2, ..., x_k, C)$, where C is the class to which point $(x_1, x_2, ..., x_k)$ belongs. The algorithm then learns weights such that the resultant network can classify new points.

A difficulty is that we don't know which network structure will work for a given problem beforehand. For example, without a detailed analysis of the problem in Section 15.2.2, we would not know to assign two hidden layers where the first layer contains 3 nodes, and the second layer contains 2 nodes. So, in general we experiment with different network configurations, and using a technique such as cross validation (Section 5.2.3), we find the configuration that gives the best results. There are various strategies for configuring the network. One strategy is to make the number of hidden nodes about equal to the number of input variables, and assign various layer and node per layer configurations. However, this will probably result in over-fitting if the dataset size is small compared to the input size. Another strategy is to make the number of hidden nodes no greater than the number of data items, and again try various layer and node per layer configurations.

In summary, to develop a neural network application we need data on the input variable(s) and output variable(s). We then construct a network with some configuration of hidden node layers and an output layer. The final step is to specify the activation functions, which we discuss next. Note that, if implementing the neural network from scratch, we would also need to program the gradient descent algorithm that finds the optimal values of the weights. However, henceforth we will assume that we are using a neural network package which has these algorithms implemented. We will present such packages in the final section.

15.3 Activation Functions

Next we discuss the activation functions that are ordinarily used in neural

networks.

15.3.1 Output Nodes

We have different activation functions for the output nodes depending on the task. If we are classifying data into one of two different classes, we need output nodes that represent binary classification. If we are classifying data in one of three or more possible classes, we need output nodes that represent multinomial classification. If our output is a continuous distribution such as the normal distribution, we need nodes that represent properties of that distribution. We discuss each in turn.

15.3.1.1 Binary Classification

In binary classification we want to classify or predict a variable that has two possible values. For example, we may want to predict whether a patient's cancer will metastasize based on features of the patient's cancer. In most such cases we do not want the system to simply say "yes" or "no". Rather we want to know, for example, the probability of the cancer metastasizing. So rather than using the discrete activation function that was used in the perceptron, we ordinarily use the **sigmoid function**, which is also used in logistic regression (Section 5.3.3). So, assuming the single output node y has the vector of hidden nodes $\mathbf{h}$ as parents, the **sigmoid activation function** for a binary outcome is

$$f(\mathbf{h}) = \frac{\exp(w_0 + \sum w_i h_i)}{1 + \exp(w_0 + \sum w_i h_i)}, \tag{15.4}$$

which yields $P(Y = 1|\mathbf{h})$.

Example 15.4 Suppose our output function is the sigmoid function, and

$$w_0 = 2, w_1 = -4, w_2 = -3, w_3 = 5.$$

Suppose further that for a particular input

$$h_1 = 6, h_2 = 7, h_3 = 9.$$

Then

$$\begin{aligned} f(\mathbf{h}) &= \frac{\exp(w_0 + w_1 \times h_1 + w_2 \times h_2 + w_3 \times h_3)}{1 + \exp(w_0 + w_1 \times h_1 + w_2 \times h_2 + w_3 \times h_3)} \\ &= \frac{\exp(2 - 4 \times 6 - 3 \times 7 + 5 \times 9)}{1 + \exp(2 - 4 \times 6 - 3 \times 7 + 5 \times 9)} \\ &= 0.881. \end{aligned}$$

So,

$$P(Y = 1|\mathbf{h}) = 0.881.$$

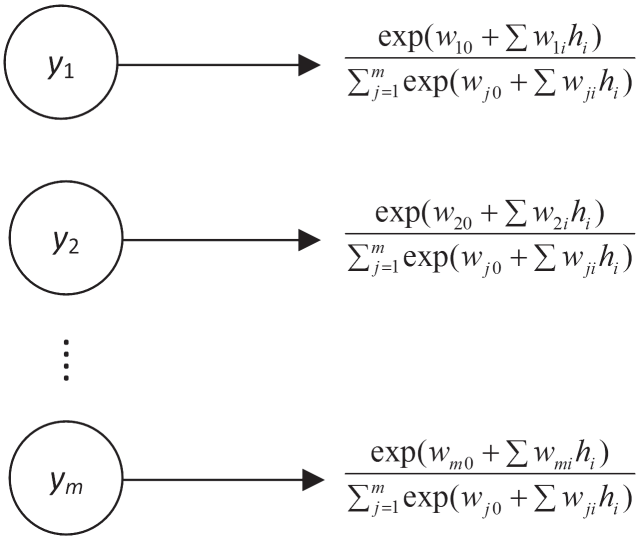


Figure 15.12 The output layer for classifying a variable with m values using the softmax function.

15.3.1.2 Multinomial Classification

In multinomial classification we want to classify or predict a variable that has m possible values. An important example, in which neural networks are often applied, is image recognition. For example, we may have a handwritten symbol of one of the digits 0-9, and our goal is to classify the symbol as the digit intended by the writer. To do multinomial classification, we use an extension of the sigmoid function called the **softmax function**. When using this function, we develop one output node for each of the m possible outcomes, and we assign the following activation function to the kth output node:

$$f_k(\mathbf{h}) = \frac{\exp(w_{k0} + \sum w_{ki}h_i)}{\sum_{j=1}^{m} \exp(w_{j0} + \sum w_{ji}h_i)}.$$

In this way, the kth output node provides $P(Y = k|\mathbf{h})$. Figure 15.12 shows the output layer for classifying a variable with m values using the softmax function.

Example 15.5 Suppose our output function is the softmax function, we have 3 outputs, y_1, y_2, and y_3, and

$$w_{10} = 2, w_{11} = -4, w_{12} = 3$$
$$w_{20} = 1, w_{21} = 9, w_{22} = -3$$
$$w_{30} = 10, w_{31} = 7, w_{32} = -4.$$

Suppose further that for a particular input

$$h_1 = 2, h_2 = 4.$$

Then

$$w_{10} + w_{11}h_1 + w_{12}h_2 = 2 - 4 \times 2 + 3 \times 4 = 6$$
$$w_{20} + w_{21}h_1 + w_{22}h_2 = 1 + 9 \times 2 - 3 \times 4 = 7$$
$$w_{30} + w_{31}h_1 + w_{32}h_2 = 10 + 7 \times 2 - 4 \times 4 = 8.$$

$$f_1(\mathbf{h}) = \frac{\exp(w_{10} + \sum w_{1i}h_i)}{\sum_{j=1}^{3} \exp(w_{j0} + \sum w_{ji}h_i)} = \frac{\exp(6)}{\exp(6) + \exp(7) + \exp(8)} = 0.090$$

$$f_2(\mathbf{h}) = \frac{\exp(w_{20} + \sum w_{2i}h_i)}{\sum_{j=1}^{3} \exp(w_{j0} + \sum w_{ji}h_i)} = \frac{\exp(7)}{\exp(6) + \exp(7) + \exp(8)} = 0.245$$

$$f_3(\mathbf{h}) = \frac{\exp(w_{30} + \sum w_{3i}h_i)}{\sum_{j=1}^{3} \exp(w_{j0} + \sum w_{ji}h_i)} = \frac{\exp(8)}{\exp(6) + \exp(7) + \exp(8)} = 0.665.$$

So,

$$\begin{aligned} P(Y &= 1|\mathbf{h}) = 0.090 \\ P(Y &= 2|\mathbf{h}) = 0.245 \\ P(Y &= 3|\mathbf{h}) = 0.665. \end{aligned}$$

15.3.1.3 Normal Output Distributions

Instead of modeling that the output can only have one of m values (discrete), we can assume it is normally distributed. We have then that

$$\rho(y|\mathbf{x}) = \text{NormalDen}(y; \mu, \sigma^2),$$

where $\mathbf{x}$ is the input vector. For example, we may want to predict a normal distribution of systolic blood pressure based on a patient's features. In this case, we can have a single output node, where its value is the mean of the normal distribution. So, the activation function is simply the **linear activation function**:

$$\mu = f(\mathbf{h}) = w_0 + \sum w_i h_i.$$

Example 15.6 Suppose our output function is the mean of a normal distribution, and

$$w_0 = 2, w_1 = -4, w_2 = -3, w_3 = 5.$$

Suppose further that for a particular input

$$h_1 = 3, h_2 = 7, h_3 = 8.$$

Then the mean of the output normal distribution is as follows:

$$\begin{aligned} \mu = f(\mathbf{h}) &= w_0 + w_1 \times h_1 + w_2 \times h_2 + w_3 \times h_3 \\ &= 2 - 4 \times 3 - 3 \times 7 + 5 \times 8 \\ &= 9. \end{aligned}$$

15.3.2 Hidden Nodes

The most popular activation function for hidden nodes (especially in applications involving image recognition) is the **rectified linear activation function,** which we have already used. That function is as follows:

$$f(\mathbf{h}) = \max(0, w_0 + \sum w_i h_i).$$

Notice that this activation function is similar to the linear activation function just discussed, except that it returns 0 when the linear combination is negative.

Example 15.7 Suppose our activation function is the rectified linear function, and

$$w_0 = -5, w_1 = -4, w_2 = -3, w_3 = 5.$$

Suppose further that for a particular input

$$h_1 = 8, h_2 = 7, h_3 = 9.$$

Then

$$\begin{aligned} f(\mathbf{h}) &= \max(0, w_0 + w_1 \times h_1 + w_2 \times h_2 + w_3 \times h_3) \\ &= \max(0, -5 - 4 \times 8 - 3 \times 7 + 5 \times 9) \\ &= \max(0, -13) \\ &= 0. \end{aligned}$$

Another activation function used for hidden nodes is the **maxout activation function**. For this function, we have r weight vectors, where r is a parameter, and we take the maximum of the weighted sums. That is, we have

$$\begin{aligned} z_1 &= w_{10} + \sum w_{1i} h_i \\ z_2 &= w_{20} + \sum w_{2i} h_i \\ &\vdots \\ z_r &= w_{r0} + \sum w_{ri} h_i, \end{aligned}$$

and

$$f(\mathbf{h}) = \max(z_1, z_2, ..., z_r).$$

Example 15.8 Suppose our activation function is the maxout function, $r = 3$, and

$$w_{10} = 2, w_{11} = -4, w_{12} = 3$$

$$w_{20} = 1, w_{21} = 9, w_{22} = -3$$

$$w_{30} = 10, w_{31} = 7, w_{32} = -4.$$

Suppose further that for a particular input

$$h_1 = 2, h_2 = 4.$$

Then

$$z_1 = w_{10} + w_{11} h_1 + w_{12} h_2 = 2 - 4 \times 2 + 3 \times 4 = 6$$

$$z_2 = w_{20} + w_{12} h_1 + w_{22} h_2 = 1 + 9 \times 2 - 3 \times 4 = 7$$

$$z_3 = w_{30} + w_{31} h_1 + w_{32} h_2 = 10 + 7 \times 2 - 4 \times 4 = 8.$$

$$f(\mathbf{h}) = \max(z_1, z_2, z_3) = \max(6, 7, 8) = 8.$$

The sigmoid activation function (Equation 15.4) can also be used as the activation function for hidden notes. A related function, which is also used, is the **hyperbolic tangent activation function**, which is as follows:

$$f(h) = \tanh(w_0 + \sum w_i h_i).$$

Figure 15.13 Examples of the handwritten digits in the MNIST dataset.

15.4 Application to Image Recognition

The MNIST dataset (http://yann.lecun.com/exdb/mnist/) is an academic dataset used to evaluate the performance of classification algorithms. The dataset consists of 60,000 training images and 10,000 test images. Each image is one of the digits 0-9, which is handwritten. It is a standardized 28 by 28 pixel greyscale image. So, there are 784 pixels in all. Figure 15.13 shows examples of the digits in the dataset.

Candel et al. [2015] developed a neural network to solve the problem of classifying images in the test dataset by learning a system from the training dataset. The network has 784 inputs (one for each pixel), 10 outputs (one for each digit) and three hidden layers, with each layer containing 200 hidden nodes. Each hidden node uses the rectified linear activation function, and the output nodes use the softmax function. Figure 15.14 depicts the network.

The neural network had a classification error rate of 0.0083, which ties the best error rate previously achieved by Microsoft.

15.5 Discussion and Further Reading

The title of this chapter is "Neural Networks and Deep Learning." Yet we never mentioned "deep learning" in the text. As mentioned in Section 1.1.2, in the 1940s foundational efforts at AI involved modeling the neurons in the brain, which resulted in the field of **neural networks** [Hebb, 1949]. Once the logical approach to AI became dominant in the 1950s, neural networks fell from popularity. However, new algorithms for training neural networks and dramatically increased computer processing speed resulted in a re-emergence of the use of neural nets in the field called **deep learning** [Goodfellow et al., 2016]. **Deep learning neural network** architectures differ from older neural networks in that they often have more hidden layers. Furthermore, deep learning networks can be trained using both unsupervised and supervised learning. We presented only the supervised learning approach. **Convolutional neural networks** are ones whose architectures make the explicit assumption that the inputs are images, and therefore encode specific properties of images. Recurrent neural networks are a class of neural nets that feed their state at the previous time step into the current time step. They are applied, for example, to automatic text generation (discussed below).

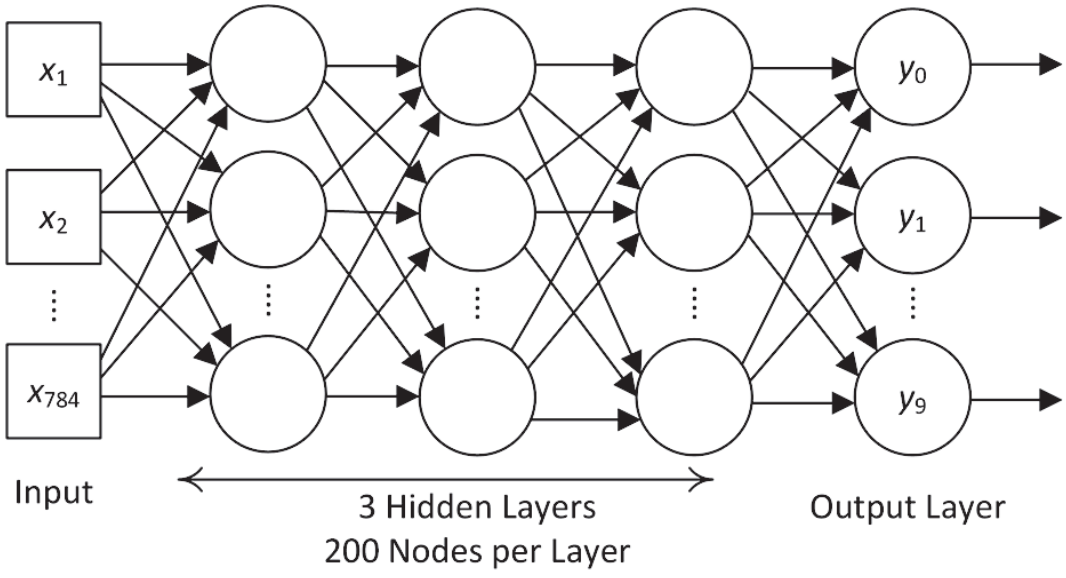


Figure 15.14 The neural network used to classify digits in the MNIST dataset.

This has only been a brief introduction to the basics of neural networks. For a more thorough coverage, including a discussion of the gradient descent algorithms used to learn neural network parameters, you are referred to [Goodfellow et al., 2016] and [Theodoridis, 2015]. You can download software that implements neural networks. Two such products are H20 (https://www.h2o.ai/) and tenserflow (https://www.tensorflow.org/).

Deep learning has been used to solve a variety of problems, which were difficult with other approaches. We close by listing some specific applications.

1. Object detection and classification in a photograph [Krizhevsky et al., 2012]. This problem involves classifying objects in a photograph as one of a set of previously known objects.

2. Adding color to black and white photographs [Zang et al., 2016]. This problem concerns adding color to black and white photographs.

3. Automatic image caption generation [Karpathy and Fei-Fei, 2015]. This task concerns generating a caption that describes the contents of an image.

4. Adding sound to silent videos [Owens et al., 2016]. This problem concerns synthesizing sounds that best match what is happening in a scene in a video.

5. Automatic language translation [Sutskever et al., 2014]. This task involves translating a word, phrase, or sentence from one language to another language.

6. Automatic handwriting generation [Graves, 2014]. This problem involves generating new handwriting for a given word or phrase based on a set of handwriting examples.

7. Automatic text generation [Sutskever et al., 2011]. This problem involves using a large amount of text to learn a model that can generate a new word, phrase, or sentence based on a partial word or text.

8. Automatic game playing [Mnih et al., 2015]. This problem concerns learning how to play a computer game based on the pixels on the screen.

When modeling a problem using a neural network, the model is a black box in the sense that the structure and parameters in the layers do not provide us with a model of reality, which we can grasp. Bayesian networks, on the other hand, provide a relationship among variables, which can often be interpreted as causal. Furthermore, Bayesian networks enable us to model and understand complex human decisions. So, although both architectures can be used to model many of the same problems, neural networks have more often been successfully applied to problems that involve human intelligence which cannot be described at the cognitive level. These problems include computer vision, image processing, and text analysis. Bayesian networks, on the other hand, have more often been applied successfully to problems that involve determining the relationships among related random variables, and exploiting these relationships to do inference and make decisions. A classic example is a medical decision support system (See Section 7.7).

EXERCISES

Section 15.1

Exercise 15.1 Does the resultant line in Example 15.1 linearly separate the data? If not, work through more iterations of the repeat loop until it does.

Exercise 15.2 Suppose we set $\lambda = 0, 2$, and we have the following data:

x_1	x_2	y
2	3	-1
4	5	1

Work through iterations of the repeat loop in Algorithm 15.1 until the resultant line linearly separates the data.

Exercise 15.3 It was left as an exercise to implement Algorithm 15.1, apply it to the data in Table 15.1, and compare the results to those obtained with logistic regression. Do this.

Section 15.2

Exercise 15.4 Suppose the three lines determining the regions in Figure 15.7 are as follows:

$$\begin{aligned} h_{11} &: 4 - 2x_1 - x_2 = 0 \\ h_{12} &: 1 - x_1 - x_2 = 0 \\ h_{13} &: 3 + 2x_1 - x_2 = 0. \end{aligned}$$

Plot the three lines and show the regions corresponding to classes C_1 and C_2. Develop parameters for the neural network in Figure 15.8 such that the network properly classifies the points in the classes C_1 and C_2.

Section 15.3

Exercise 15.5 Suppose our output function is the sigmoid function for binary output and

$$w_0 = 1, w_1 = -4, w_2 = -5, w_3 = 4.$$

Suppose further that for a particular input

$$h_1 = 4, h_2 = 5, h_3 = 6.$$

Compute $f(\mathbf{h})$. What is $P(Y = 1|\mathbf{h})$?

Exercise 15.6 Suppose our output function is the softmax function, and we have 4 outputs, y_1, y_2, y_3, y_4. Suppose further that

$$\begin{aligned} w_{10} = 1, w_{11} = -3, w_{12} = 2 \\ w_{20} = 2, w_{21} = 7, w_{22} = -2 \\ w_{30} = 6, w_{31} = 5, w_{32} = -4. \\ w_{40} = 5, w_{41} = -3, w_{42} = 6. \end{aligned}$$

Suppose further that for a particular input

$$h_1 = 3, h_2 = 4, h_3 = 5.$$

Compute $f_1(\mathbf{h}), f_2(\mathbf{h}), f_3(\mathbf{h}), f_4(\mathbf{h})$. What is $P(Y = i|\mathbf{h})$ for $i = 1, 2, 3, 4$?

Exercise 15.7 Suppose our activation function is the rectified linear function, and

$$w_0 = 5, w_1 = -4, w_2 = 2, w_3 = 4, w_4 = 8.$$

Suppose further that for a particular input

$$h_1 = 8, h_2 = 7, h_3 = 6, h_4 = 5.$$

Compute $f(\mathbf{h})$.

Exercise 15.8 Suppose our activation function is the maxout function, $r = 2$, and

$$\begin{aligned} w_{10} = 1, w_{11} = -2, w_{12} = 6, w_{13} = 5 \\ w_{20} = 2, w_{21} = 8, w_{22} = -2, w_{23} = 4. \end{aligned}$$

Suppose further that for a particular input

$$h_1 = 3, h_2 = 2, h_3 = 5.$$

Compute $f(\mathbf{h})$.

Section 15.4

Exercise 15.9 The Metabric dataset is introduced in [Curtis, et al. 2012]. It provides data on breast cancer patient features such as tumor size and outcomes such as death. This dataset can be obtained at https://www.synapse.org/#!Synapse:syn1688369/wiki/27311. Gain access to the dataset. Then download one of the neural network packages discussed in Section 15.5. Divide the dataset into a training dataset containing 2/3 of the data, and a test dataset containing 1/3 of the data. Apply various parameter settings (e.g., number of hidden layers and number of hidden nodes per layer) to the training set. For each setting do a 5-fold cross validation analysis, where the goal is to classify/predict whether the patient dies. Determine the Area Under an ROC Curve (AUROC) for each of the settings, and apply the settings with the best AUROC to the test data. Determine the AUROC for the test data.

A **naive Bayesian network** is a network in which there is one root, and all other nodes are children of the root. There are no edges among the children. Naive Bayesian network are used for discrete classification by making the target the root, and the predictors the children. XLSTAT includes a naive Bayesian network module. Free naive Bayesian network software is available at various sites including http://www.kdnuggets.com/software/bayesian.html. Download a naive Bayesian network software package, and do the same study as outlined above using this package. Vary whatever parameters are available in the software, and do the 5-fold cross validation analysis for each parameter setting. Compare the AUROCs obtained by the neural network method and the naive Bayesian network method when applied to the test data.

Exercise 15.10 As discussed in Section 15.4, the MNIST dataset is an academic dataset used to evaluate the performance of classification algorithms. The dataset consists of 60,000 training images and 10,000 test images. Each image is one of the digits 0-9, which is handwritten. It is a standardized 28 by 28 pixel greyscale image. So, there are 784 pixels in all. Download this dataset. Then download one of the neural network packages discussed in Section 15.5. Apply various parameter settings (e.g., number of hidden layers and number of hidden nodes per layer) to the training set. For each setting do a 5-fold cross validation analysis, where the goal is to classify/predict the correct digit. Determine the Area Under an ROC Curve (AUROC) for each of the settings, and apply the settings with the best AUROC to the test data. Determine the AUROC for the test data.

Than apply a naive Bayesian network to the dataset. Again use various parameter settings and 5-fold cross validation on the training dataset, and apply the best parameter values to the test dataset. Compare the AUROCs obtained by the neural network method and the naive Bayesian network method when applied to the test dataset.

CHAPTER

7

AI-COMPLETENESS: THE PROBLEM DOMAIN OF SUPERINTELLIGENT MACHINES

This chapter is excerpted from
Artificial Superintelligence: A Futuristic Approach
by Roman V. Yampolskiy.

© 2018Taylor & Francis Group. All rights reserved.

Learn more

AI-Completeness

The Problem Domain of Superintelligent Machines[*]

1.1 INTRODUCTION

Since its inception in the 1950s, the field of artificial intelligence (AI) has produced some unparalleled accomplishments while failing to formalize the problem space that concerns it. This chapter addresses this shortcoming by extending previous work (Yampolskiy 2012a) and contributing to the theory of AI-Completeness, a formalism designed to do for the field of AI what the notion of NP-Completeness (where NP stands for nondeterministic polynomial time) did for computer science in general. It is my belief that such formalization will allow for even faster progress in solving remaining problems in humankind's quest to build an intelligent machine.

According to Wikipedia, the term *AI-Complete* was proposed by Fanya Montalvo in the 1980s ("AI-Complete" 2011). A somewhat general definition of the term included in the 1991 "Jargon File" (Raymond 1991) states:

> AI-complete: [MIT, Stanford, by analogy with "NP-complete"] adj. Used to describe problems or subproblems in AI, to indicate that the solution presupposes a solution to the "strong AI

[*] Reprinted from Roman V. Yampolskiy, Artificial intelligence, evolutionary computation and metaheuristics. *Studies in Computational Intelligence* 427:3–17, 2013, with kind permission of Springer Science and Business Media. Copyright 2013, Springer Science and Business Media.

problem" (that is, the synthesis of a human-level intelligence). A problem that is AI-complete is, in other words, just too hard.

As such, the term *AI-Complete* (or sometimes AI-Hard) has been a part of the field for many years and has been frequently brought up to express the difficulty of a specific problem investigated by researchers (see Mueller 1987; Mallery 1988; Gentry, Ramzan, and Stubblebine 2005; Phillips and Beveridge 2009; Bergmair 2004; Ide and Véronis 1998; Navigli and Velardi 2005; Nejad 2010; Chen et al. 2009; McIntire, Havig, and McIntire 2009; McIntire, McIntire, and Havig 2009; Mert and Dalkilic 2009; Hendler 2008; Leahu, Sengers, and Mateas 2008; Yampolskiy 2011). This informal use further encouraged similar concepts to be developed in other areas of science: Biometric-Completeness (Phillips and Beveridge 2009) or Automatic Speech Recognition (ASR)-Complete (Morgan et al. 2003). Although recently numerous attempts to formalize what it means to say that a problem is AI-Complete have been published (Ahn et al. 2003; Shahaf and Amir 2007; Demasi, Szwarcfiter, and Cruz 2010), even before such formalization attempts, systems that relied on humans to solve problems perceived to be AI-Complete were utilized:

- **AntiCaptcha** systems use humans to break the CAPTCHA (Completely Automated Public Turing Test to Tell Computers and Humans Apart) security protocol (Ahn et al. 2003; Yampolskiy 2007a, 2007b; Yampolskiy and Govindaraju 2007) either by directly hiring cheap workers in developing countries (Bajaj 2010) or by rewarding correctly solved CAPTCHAs with presentation of pornographic images (Vaas 2007).
- The **Chinese room** philosophical argument by John Searle shows that including a human as a part of a computational system may actually reduce its perceived capabilities, such as understanding and consciousness (Searle 1980).
- **Content development** online projects such as encyclopedias (Wikipedia, Conservapedia); libraries (Project Gutenberg, video collections [YouTube]; and open-source software [SourceForge]) all rely on contributions from people for content production and quality assurance.
- **Cyphermint**, a check-cashing system, relies on human workers to compare a snapshot of a person trying to perform a financial

transaction to a picture of a person who initially enrolled with the system. Resulting accuracy outperforms any biometric system and is almost completely spoof proof (see http://cyphermint.com for more information).

- **Data-tagging** systems entice a user into providing metadata for images, sound, or video files. A popular approach involves developing an online game that, as a by-product of participation, produces a large amount of accurately labeled data (Ahn 2006).
- **Distributed Proofreaders** employs a number of human volunteers to eliminate errors in books created by relying on Optical Character Recognition process (see http://pgdp.net/c/ for more information).
- **Interactive evolutionary computation** algorithms use humans in place of a fitness function to make judgments regarding difficult-to-formalize concepts such as aesthetic beauty or taste (Takagi 2001).
- **Mechanical Turk** is an attempt by Amazon.com to create Artificial AI. Humans are paid varying amounts for solving problems that are believed to be beyond current abilities of AI programs (see https://www.mturk.com/mturk/welcome for more information). The general idea behind the Turk has broad appeal, and the researchers are currently attempting to bring it to the masses via the generalized task markets (GTMs) (Shahaf and Horvitz 2010; Horvitz and Paek 2007; Horvitz 2007; Kapoor et al. 2008).
- **Spam prevention** is easy to accomplish by having humans vote on e-mails they receive as spam or not. If a certain threshold is reached, a particular piece of e-mail could be said with a high degree of accuracy to be spam (Dimmock and Maddison 2004).

Recent work has attempted to formalize the intuitive notion of AI-Completeness. In particular, three such endowers are worth reviewing next (Yampolskiy 2012a). In 2003, Ahn et al. attempted to formalize the notion of an AI-Problem and the concept of AI-Hardness in the context of computer security. An AI-Problem was defined as a triple:

$\mathcal{P} = (S, D, f)$, where S is a set of problem instances, D is a probability distribution over the problem set S, and $f : S \rightarrow \{0; 1\}^*$

> answers the instances. Let $\delta \in (0; 1]$. We require that for an $\alpha > 0$ fraction of the humans H, $Pr_{x \leftarrow D}[H(x) = f(x)] > \delta$. … An AI problem $\mathcal{P}$ is said to be (δ, τ)-*solved* if there exists a program A, running in time at most τ on any input from S, such that $Pr_{x \leftarrow D, r}[A_r(x) = f(x)] \geq \delta$. ($A$ is said to be a (δ, τ) solution to $\mathcal{P}$.) $\mathcal{P}$ is said to be a (δ, τ)-*hard AI problem* if no current program is a (δ, τ) solution to $\mathcal{P}$. (Ahn et al. 2003, 298).

It is interesting to observe that the proposed definition is in terms of democratic consensus by the AI community. If researchers say the problem is hard, it must be so. Also, time to solve the problem is not taken into account. The definition simply requires that some humans be able to solve the problem (Ahn et al. 2003).

In 2007, Shahaf and Amir presented their work on the theory of AI-Completeness. Their work puts forward the concept of the human-assisted Turing machine and formalizes the notion of different human oracles (HOs; see the section on HOs for technical details). The main contribution of the paper comes in the form of a method for classifying problems in terms of human-versus-machine effort required to find a solution. For some common problems, such as natural language understanding (NLU), the work proposes a method of reductions that allow conversion from NLU to the problem of speech understanding via text-to-speech software.

In 2010, Demasi et al. (Demasi, Szwarcfiter, and Cruz 2010) presented their work on problem classification for artificial general intelligence (AGI). The proposed framework groups the problem space into three sectors:

- **Non-AGI-Bound**: problems that are of no interest to AGI researchers
- **AGI-Bound**: problems that require human-level intelligence to be solved
- **AGI-Hard**: problems that are at least as hard as any AGI-Bound problem.

The work also formalizes the notion of HOs and provides a number of definitions regarding their properties and valid operations.

String Human (String input) {

return output; }

FIGURE 1.1 Human oracle: $\text{Human}_{\text{Best}}$, a union of minds.

1.2 THE THEORY OF AI-COMPLETENESS

From people with mental disabilities to geniuses, human minds are cognitively diverse, and it is well known that different people exhibit different mental abilities. I define a notion of an HO function capable of computing any function computable by the union of all human minds. In other words, any cognitive ability of any human being is repeatable by my HO. To make my HO easier to understand, I provide Figure 1.1, which illustrates the *Human* function.

Such a function would be easy to integrate with any modern programming language and would require that the input to the function be provided as a single string of length N, and the function would return a string of length M. No encoding is specified for the content of strings N or M, so they could be either binary representations of data or English language phrases—both are computationally equivalent. As necessary, the Human function could call regular Turing Machine (TM) functions to help in processing data. For example, a simple computer program that would display the input string as a picture to make human comprehension easier could be executed. Humans could be assumed to be cooperating, perhaps because of a reward. Alternatively, one can construct a Human function that instead of the union of all minds computes the average decision of all human minds on a problem encoded by the input string as the number of such minds goes to infinity. To avoid any confusion, I propose naming the first HO $\text{Human}_{\text{Best}}$ and the second HO $\text{Human}_{\text{Average}}$. Problems in the AI domain tend to have a large degree of ambiguity in terms of acceptable correct answers. Depending on the problem at hand, the simplistic notion of an average answer could be replaced with an aggregate

answer as defined in the wisdom-of-crowds approach (Surowiecki 2004). Both functions could be formalized as human-assisted Turing machines (Shahaf and Amir 2007).

The human function is an easy-to-understand and -use generalization of the HO. One can perceive it as a way to connect and exchange information with a real human sitting at a computer terminal. Although easy to intuitively understand, such description is not sufficiently formal. Shahaf et al. have formalized the notion of HO as an Human-Assisted Turing Machine (HTM) (Shahaf and Amir 2007). In their model, a human is an oracle machine that can decide a set of languages L_i in constant time: $H \subseteq \{L_i \mid L_i \subseteq \sum^{\star}\}$. If time complexity is taken into account, answering a question might take a nonconstant time, $H \subseteq \{<L_i, f_i> \mid L_i \subseteq \sum^{\star}, f_i : \mathbb{N} \rightarrow \mathbb{N}\}$, where f_i is the time-complexity function for language L_i, meaning the human can decide if $x \in L_i$ in f_i ($|x|$) time. To realistically address capabilities of individual humans, a probabilistic oracle was also presented that provided correct answers with probability p: $H \subseteq \{<L_i, p_i> \mid L_i \subseteq \sum^{\star}, 0 \leq p_i \leq 1\}$. Finally, the notion of reward is introduced into the model to capture humans' improved performance on "paid" tasks: $H \subseteq \{<L_i, u_i> \mid L_i \subseteq \sum^{\star}, u_i : \mathbb{N} \rightarrow \mathbb{N}\}$ where u_i is the utility function (Shahaf and Amir 2007).

1.2.1 Definitions

Definition 1: A problem *C* is **AI-Complete** if it has two properties:

1. It is in the set of AI problems (HO solvable).
2. Any AI problem can be converted into *C* by some polynomial time algorithm.

Definition 2: **AI-Hard:** A problem *H* is AI-Hard if and only if there is an AI-Complete problem *C* that is polynomial time Turing reducible to *H*. ■

Definition 3: **AI-Easy:** The complexity class AI-Easy is the set of problems that are solvable in polynomial time by a deterministic Turing machine with an oracle for some AI problem. In other words, a problem *X* is AI-Easy if and only if there exists some AI problem *Y* such that *X* is polynomial time Turing reducible to *Y*. This means that given an oracle for *Y*, there exists an algorithm that solves *X* in polynomial time. ■

Figure 1.2 illustrates the relationship between different AI complexity classes. The right side of the figure shows the situation if it is ever proven that AI problems = AI-Complete problems. The left side shows the converse.

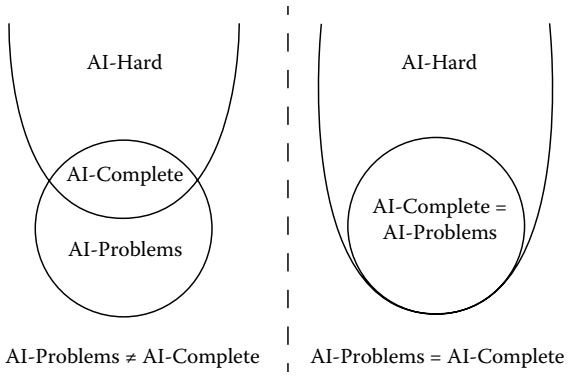


FIGURE 1.2 Relationship between AI complexity classes.

1.2.2 Turing Test as the First AI-Complete Problem

In this section, I show that a Turing test (TT; Turing 1950) problem is AI-Complete. First, I need to establish that a TT is indeed an AI problem (HO solvable). This trivially follows from the definition of the test itself. The test measures if a human-like performance is demonstrated by the test taker, and HOs are defined to produce human-level performance. While both *human* and *intelligence test* are intuitively understood terms, I have already shown that HOs could be expressed in strictly formal terms. The TT itself also could be formalized as an interactive proof (Shieber 2006, 2007; Bradford and Wollowski 1995).

The second requirement for a problem to be proven to be AI-Complete is that any other AI problem should be convertible into an instance of the problem under consideration in polynomial time via Turing reduction. Therefore, I need to show how any problem solvable by the Human function could be encoded as an instance of a TT. For any HO-solvable problem *h*, we have a string *input* that encodes the problem and a string *output* that encodes the solution. By taking the *input* as a question to be used in the TT and *output* as an answer to be expected while administering a TT, we can see how any HO-solvable problem could be reduced in polynomial time to an instance of a TT. Clearly, the described process is in polynomial time, and by similar algorithm, any AI problem could be reduced to TT. It is even theoretically possible to construct a complete TT that utilizes all other problems solvable by HO by generating one question from each such problem.

1.2.3 Reducing Other Problems to a TT

Having shown a first problem (TT) to be AI-Complete, the next step is to see if any other well-known AI problems are also AI-Complete. This is an effort similar to the work of Richard Karp, who showed some 21 problems were NP-Complete in his 1972 work and by doing so started a new field of computational complexity (Karp 1972). According to the *Encyclopedia of Artificial Intelligence* (Shapiro 1992), the following problems are all believed to be AI-Complete and so will constitute primary targets for our effort of proving formal AI-Completeness on them (Shapiro 1992, 54–57):

- **Natural Language Understanding:** "Encyclopedic knowledge is required to understand natural language. Therefore, a complete Natural Language system will also be a complete Intelligent system."
- **Problem Solving:** "Since any area investigated by AI researchers may be seen as consisting of problems to be solved, all of AI may be seen as involving Problem Solving and Search."
- **Knowledge Representation and Reasoning:** "The intended use is to use explicitly stored knowledge to produce additional explicit knowledge. This is what reasoning is. Together Knowledge representation and Reasoning can be seen to be both necessary and sufficient for producing general intelligence—it is another AI-complete area."
- **Vision or Image Understanding:** "If we take 'interpreting' broadly enough, it is clear that general intelligence may be needed to do this interpretation, and that correct interpretation implies general intelligence, so this is another AI-complete area."

Now that the TT has been proven to be AI-Complete, we have an additional way of showing other problems to be AI-Complete. We can either show that a problem is both in the set of AI problems and all other AI problems can be converted into it by some polynomial time algorithm or can reduce any instance of TT problem (or any other problem already proven to be AI-Complete) to an instance of a problem we are trying to show to be AI-Complete. This second approach seems to be particularly powerful. The general heuristic of my approach is to see if all information encoding the question that could be asked during administration of a TT could be encoded as an instance of a problem in question and likewise if any potential solution to that problem would constitute an answer to the

relevant TT question. Under this heuristic, it is easy to see that, for example, chess is not AI-Complete as only limited information can be encoded as a starting position on a standard-size chessboard. Not surprisingly, chess has been one of the greatest successes of AI; currently, chess-playing programs dominate all human players, including world champions.

Question answering (QA) (Hirschman and Gaizauskas 2001; Salloum 2009) is a subproblem in natural language processing. Answering questions at a level of a human is something HOs are particularly good at based on their definition. Consequently, QA is an AI-Problem that is one of the two requirements for showing it to be AI-Complete. Having access to an oracle capable of solving QA allows us to solve TT via a simple reduction. For any statement *S* presented during administration of TT, transform said statement into a question for the QA oracle. The answers produced by the oracle can be used as replies in the TT, allowing the program to pass the TT. It is important to note that access to the QA oracle is sufficient to pass the TT only if questions are not restricted to stand-alone queries, but could contain information from previous questions. Otherwise, the problem is readily solvable even by today's machines, such as IBM's Watson, which showed a remarkable performance against human *Jeopardy* champions (Pepitone 2011).

Speech understanding (SU) (Anusuya and Katti 2009) is another subproblem in natural language processing. Understanding speech at a level of a human is something HOs are particularly good at based on their definition. Consequently, SU is an AI-Problem that is one of the two requirements for showing it to be AI-Complete. Having access to an oracle capable of solving SU allows us to solve QA via a simple reduction. We can reduce QA to SU by utilizing any text-to-speech software (Taylor and Black 1999; Chan 2003), which is both fast and accurate. This reduction effectively transforms written questions into the spoken ones, making it possible to solve every instance of QA by referring to the SU oracle.

1.2.4 Other Probably AI-Complete Problems

I hope that my work will challenge the AI community to prove other important problems as either belonging or not belonging to that class. Although the following problems have not been explicitly shown to be AI-Complete, they are strong candidates for such classification and are problems of great practical importance, making their classification a worthy endeavor. If a problem has been explicitly conjectured to be AI-Complete in a published paper, I include a source of such speculation: dreaming (Salloum 2009);

commonsense planning (Shahaf and Amir 2007); foreign policy (Mallery 1988); problem solving (Shapiro 1992); judging a TT (Shahaf and Amir 2007); commonsense knowledge (Andrich, Novosel, and Hrnkas 2009); SU (Shahaf and Amir 2007); knowledge representation and reasoning (Shapiro 1992); word sense disambiguation (Chen et al. 2009; Navigli and Velardi 2005); Machine Translation ("AI-Complete" 2011); ubiquitous computing (Leahu, Sengers, and Mateas 2008); change management for biomedical ontologies (Nejad 2010); NLU (Shapiro 1992); software brittleness ("AI-Complete" 2011); and vision or image understanding (Shapiro 1992).

1.3 FIRST AI-HARD PROBLEM: PROGRAMMING

I define the problem of programming as taking a natural language description of a program and producing a source code, which then is compiled on some readily available hardware/software to produce a computer program that satisfies all implicit and explicit requirements provided in the natural language description of the programming problem assignment. Simple examples of programming are typical assignments given to students in computer science classes, for example, "Write a program to play tic-tac-toe." Successful students write source code that, if correctly compiled, allows the grader to engage the computer in an instance of that game. Many requirements of such an assignment remain implicit, such as that response time of the computer should be less than a minute. Such implicit requirements are usually easily inferred by students who have access to culture-instilled common sense. As of this writing, no program is capable of solving programming outside strictly restricted domains.

Having access to an oracle capable of solving programming allows us to solve TT via a simple reduction. For any statement *S* presented during TT, transform said statement into a programming assignment of the form: "Write a program that would respond to *S* with a statement indistinguishable from a statement provided by an average human" (a full transcript of the TT may also be provided for disambiguation purposes). Applied to the set of all possible TT statements, this procedure clearly allows us to pass TT; however, programming itself is not in AI-Problems as there are many instances of programming that are not solvable by HOs. For example, "Write a program to pass a Turing test" is not known to be an AI-Problem under the proposed definition. Consequently, programming is an AI-Hard problem.

1.4 BEYOND AI-COMPLETENESS

The HO function presented in this chapter assumes that the human behind it has some assistance from the computer in order to process certain human unfriendly data formats. For example, a binary string representing a video is completely impossible for a human to interpret, but it could easily be played by a computer program in the intended format, making it possible for a human to solve a video understanding-related AI-Complete problem. It is obvious that a human provided with access to a computer (perhaps with Internet connection) is a more powerful intelligence compared to an unenhanced, in such a way, human. Consequently, it is important to limit help from a computer to a human worker "inside" a HO function to assistance in the domain of input/output conversion, but not beyond, as the resulting function would be both AI-Complete and "Computer Complete".

Figure 1.3 utilizes a Venn diagram to illustrate subdivisions of problem space produced by different types of intelligent computational devices. Region 1 represents what is known as a Universal Intelligence (Legg and Hutter 2007) or a Super Intelligence (Legg 2008; Yampolskiy 2011a, 2011b, 2012b)—a computational agent that outperforms all other intelligent

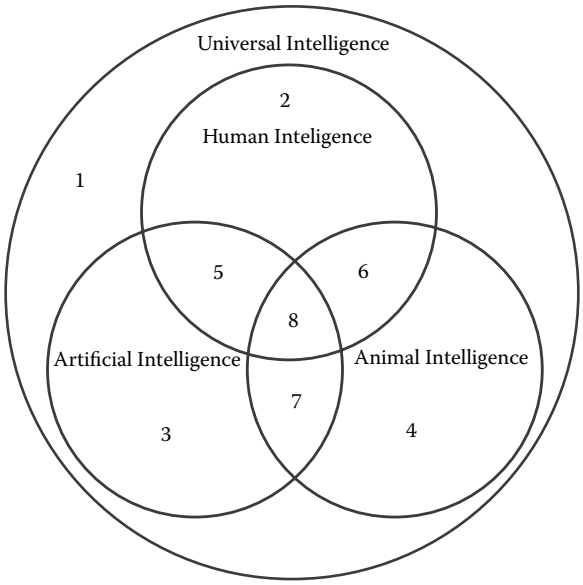


FIGURE 1.3 Venn diagram for four different types of intelligence.

agents over all possible environments. Region 2 is the standard unenhanced Human-level intelligence of the type capable of passing a TT, but at the same time incapable of computation involving large numbers or significant amount of memorization. Region 3 is what is currently possible to accomplish via state-of-the-art AI programs. Finally, Region 4 represents an abstract view of animal intelligence.

AI intelligence researchers strive to produce Universal Intelligence, and it is certainly likely to happen, given recent trends in both hardware and software developments and the theoretical underpinning of the Church/Turing Thesis (Turing 1936). It is also likely that if we are able to enhance human minds with additional memory and port those to a higher-speed hardware we will essentially obtain a Universal Intelligence (Sandberg and Boström 2008).

While the Universal Intelligence incorporates abilities of all the lower intelligences, it is interesting to observe that Human, AI and Animal intelligences have many interesting regions of intersection (Yampolskiy and Fox 2012). For example, animal minds are as good as human minds at visual understanding of natural scenes. Regions 5, 6, and 7 illustrate common problem spaces between two different types of intelligent agents. Region 8 represents common problem solving abilities of humans, computers and animals. Understanding such regions of commonality may help us to better separate the involved computational classes, which are represented by abilities of a specific computational agent minus the commonalities with a computational agent with which we are trying to draw a distinction. For example, CAPTCHA (Ahn et al. 2003) type tests rely on the inability of computers to perform certain pattern recognition tasks with the same level of accuracy as humans in order to separate AI agents from Human agents. Alternatively, a test could be devised to tell humans not armed with calculators from AIs by looking at the upper level of ability. Such a test should be easy to defeat once an effort is made to compile and formalize the limitations and biases of the human mind.

It is also interesting to consider the problem solving abilities of hybrid agents. I have already noted that a human being equipped with a computer is a lot more capable compared to an unaided person. Some research in Brain Computer Interfaces (Vidal 1973) provides a potential path for future developments in the area. Just as interestingly, combining pattern recognition abilities of animals with symbol processing abilities of AI could produce a computational agent with a large domain of human-like abilities (see work on RoboRats by Talwar et al. (2002) and on monkey controlled robots by Nicolelis

et al. 2000). It is very likely that in the near future different types of intelligent agents will be combined to even greater extent. While such work is under way, I believe that it may be useful to introduce some additional terminology into the field of AI problem classification. For the complete space of problems I propose that the computational agents which are capable of solving a specific subset of such problems get to represent the set in question. Therefore, I propose additional terms: "Computer-Complete" and "Animals-Complete" to represent computational classes solvable by such agents. It is understood that just as humans differ in their abilities, so do animals and computers. Aggregation and averaging utilized in my Human function could be similarly applied to the definition of respective oracles. As research progresses, common names may be needed for different combinations of regions from Figure 1.3 illustrating such concepts as Human-AI hybrid or Animal-Robot hybrid.

Certain aspects of human cognition do not map well onto the space of problems which have seen a lot of success in the AI research field. Internal states of the human mind, such as consciousness (stream of), self-awareness, understanding, emotions (love, hate), feelings (pain, pleasure), etc., are not currently addressable by our methods. Our current state-of-the-art technologies are not sufficient to unambiguously measure or detect such internal states, and consequently even their existence is not universally accepted. Many scientists propose ignoring such internal states or claim they are nothing but a byproduct of flawed self-analysis. Such scientists want us to restrict science only to measurable behavioral actions; however, since all persons have access to internal states of at least one thinking machine, interest in trying to investigate internal states of the human mind is unlikely to vanish.

While I am able to present a formal theory of AI-Completeness based on the concept of HOs, the theory is not strong enough to address problems involving internal states of the mind. In fact, one of the fundamental arguments against our ability to implement understanding in a system that is based on symbol manipulation, Searle's Chinese Room thought experiment, itself relies on a generalized concept of a human as a part of a computational cycle. It seems that the current Turing/Von Neumann architecture is incapable of dealing with the set of problems which are related to internal states of human mind. Perhaps a new type of computational architecture capable of mimicking such internal states will be developed in the future. It is likely that it will be inspired by a better understanding of human biology and cognitive science. Research on creating Artificial Consciousness (AC) is attracting a lot of attention, at least in terms of number of AC papers published.

As a part of my ongoing effort to classify AI related problems, I propose a new category specifically devoted to problems of reproducing internal states of the human mind in artificial ways. I call this group of problems Consciousness-Complete or C-Complete for short. An oracle capable of solving C-Complete problems would be fundamentally different from the Oracle Machines proposed by Turing. C-Oracles would take input in the same way as their standard counterparts but would not produce any symbolic output. The result of their work would be a novel internal state of the oracle, which may become accessible to us if the new type of hardware discussed above is developed.

Just as SAT was shown to be the first NP-Complete problem and TT to be the first AI-Complete problem, I suspect that Consciousness will be shown to be the first C-Complete problem, with all other internal-state related problems being reducible to it. Which of the other internal state problems are also C-Complete is beyond the scope of this preliminary work. Even with no consciousness-capable hardware available at the moment of this writing, the theory of C-Completeness is still a useful tool, as it allows for formal classification of classical problems in the field of Artificial Intelligence into two very important categories: potentially *solvable* (with current technology) and unsolvable (with current technology). Since the only information available about HOs is their output and not internal states, they are fundamentally different from C-Oracles, creating two disjoint sets of problems.

The history of AI research is full of unwarranted claims of anticipated breakthroughs and, conversely, overestimations regarding the difficulty of some problems. Viewed through the prism of my AI-Complete/C-Complete theories, the history of AI starts to make sense. Solutions for problems that I classify as AI-Complete have been subject to continuous steady improvement, while those falling in the realm of C-Completeness have effectively seen zero progress (computer pain, Bishop 2009 and Dennett 1978; artificial consciousness, Searle 1980 and Dreyfus 1972; etc.). To proceed, science needs to better understand what the difference between a feeling and a thought is. Feeling pain and knowing about pain are certainly not the same internal states. I am hopeful that future research in this area will bring some long-awaited answers.

1.5 CONCLUSIONS

Progress in the field of artificial intelligence requires access to well-defined problems of measurable complexity. The theory of AI-Completeness aims to provide a base for such formalization. Showing certain problems to be

AI-Complete/-Hard is useful for developing novel ways of telling computers from humans. Also, any problem shown to be AI-Complete would be a great alternative way of testing an artificial intelligent agent to see if it attained human level intelligence (Shahaf and Amir 2007).

REFERENCES

Ahn, Luis von. June 2006. Games with a purpose. *IEEE Computer Magazine* 96–98.

Ahn, Luis von, Manuel Blum, Nick Hopper, and John Langford. 2003. CAPTCHA: Using Hard AI Problems for Security. Paper read at Eurocrypt. Advances in Cryptology — EUROCRYPT 2003. International Conference on the Theory and Applications of Cryptographic Techniques, Warsaw, Poland, May 4–8, 2003. Published in *Lecture Notes in Computer Science* 2656 (2003): 294–311.

AI-Complete. 2011. Accessed January 7. http://en.wikipedia.org/wiki/AI-complete.

Andrich, Christian, Leo Novosel, and Bojan Hrnkas. 2009. Common Sense Knowledge. Exercise Paper—Information Search and Retrieval. http://www.iicm.tu-graz.ac.at/cguetl/courses/isr/uearchive/uews2009/Ue06-CommonSenseKnowledge.pdf

Anusuya, M. A. and S. K. Katti. 2009. Speech recognition by machine: a review. *International Journal of Computer Science and Information Security (IJCSIS)* no. 6(3):181–205.

Bajaj, Vikas. April 25, 2010. Spammers pay others to answer security tests. *New York Times*.

Bergmair, Richard. December 2004. Natural Language Steganography and an "AI-Complete" Security Primitive. In 21st Chaos Communication Congress, Berlin.

Bishop, Mark. 2009. Why computers can't feel pain. *Minds and Machines* 19(4):507–516.

Bradford, Philip G. and Michael Wollowski. 1995. A formalization of the Turing Test. *SIGART Bulletin* 6(4):3–10.

Chan, Tsz-Yan. 2003. Using a text-to-speech synthesizer to generate a reverse Turing test. Paper presented at the 15th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'03), Washington, DC, November 3–5.

Chen, Junpeng, Juan Liu, Wei Yu, and Peng Wu. November 30, 2009. Combining Lexical Stability and Improved Lexical Chain for Unsupervised Word Sense Disambiguation. Paper presented at the Second International Symposium on Knowledge Acquisition and Modeling (KAM '09), Wuhan, China.

Demasi, Pedro, Jayme L. Szwarcfiter, and Adriano J. O. Cruz. March 5–8, 2010. A Theoretical Framework to Formalize AGI-Hard Problems. Paper presented at the Third Conference on Artificial General Intelligence, Lugano, Switzerland.

Dennett, Daniel C. July 1978. Why you can't make a computer that feels pain. *Synthese* 38(3):415–456.

Dimmock, Nathan and Ian Maddison. December 2004. Peer-to-peer collaborative spam detection. *Crossroads* 11(2): 17–25.

Dreyfus, Hubert L. 1972. *What Computers Can't Do: A Critique of Artificial Reason*: New York: Harper & Row.

Gentry, Craig, Zulfikar Ramzan, and Stuart Stubblebine. June 5–8, 2005. Secure Distributed Human Computation. Paper presented at the 6th ACM Conference on Electronic Commerce, Vancouver, BC, Canada.

Hendler, James. September 2008. We've come a long way, maybe … . *IEEE Intelligent Systems* 23(5):2–3.

Hirschman, L., and R Gaizauskas. 2001. Natural language question answering. The view from here. *Natural Language Engineering* 7(4):275–300.

Horvitz, E. 2007. Reflections on challenges and promises of mixed-initiative interaction. *AI Magazine—Special Issue on Mixed-Initiative Assistants* 28(2): 11–18.

Horvitz, E. and T. Paek. 2007. Complementary computing: policies for transferring callers from dialog systems to human receptionists. *User Modeling and User Adapted Interaction* 17(1):159–182.

Ide, N. and J. Véronis. 1998. Introduction to the special issue on word sense disambiguation: the state of the art. *Computational Linguistics* 24(1):1–40.

Kapoor, A., D. Tan, P. Shenoy, and E. Horvitz. September 17–19, 2008. Complementary Computing for Visual Tasks: Meshing Computer Vision with Human Visual Processing. Paper presented at the IEEE International Conference on Automatic Face and Gesture Recognition, Amsterdam.

Karp, Richard M. 1972. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, edited by R. E. Miller and J. W. Thatcher, 85–103. New York: Plenum.

Leahu, Lucian, Phoebe Sengers, and Michael Mateas. September 21–24, 2008. Interactionist AI and the Promise of ubicomp, or, How to Put Your Box in the World Without Putting the World in Your Box. Paper presented at the *Tenth International Conference on Ubiquitous Computing*. Seoul, South Korea.

Legg, Shane. June 2008. Machine Super Intelligence. PhD thesis, University of Lugano, Switzerland. http://www.vetta.org/documents/Machine_Super_Intelligence.pdf

Legg, Shane and Marcus Hutter. December 2007. Universal intelligence: a definition of machine intelligence. *Minds and Machines* 17(4):391–444.

Mallery, John C. 1988. Thinking about Foreign Policy: Finding an Appropriate Role for Artificial Intelligence Computers. Ph.D. dissertation, MIT Political Science Department, Cambridge, MA.

McIntire, John P., Paul R. Havig, and Lindsey K. McIntire. July 21–23, 2009. Ideas on Authenticating Humanness in Collaborative Systems Using AI-Hard Problems in Perception and Cognition. Paper presented at the IEEE National Aerospace and Electronics Conference (NAECON), Dayton, OH.

McIntire, John P., Lindsey K. McIntire, and Paul R. Havig. May 18–22, 2009. A Variety of Automated Turing tests for Network Security: Using AI-Hard Problems in Perception and Cognition to Ensure Secure Collaborations. Paper presented at the International Symposium on Collaborative Technologies and Systems (CTS '09), Baltimore.

Mert, Ezgi, and Cokhan Dalkilic. September 14–16, 2009. Word Sense Disambiguation for Turkish. Paper presented at the 24th International Symposium on Computer and Information Sciences (ISCIS 2009), Guzelyurt, Turkey.

Morgan, Nelson, D. Baron, S. Bhagat, H. Carvey, R. Dhillon, J. Edwards, D. Gelbart, A. Janin, A. Krupski, B. Peskin, T. Pfau, E. Shriberg, A. Stolcke, and C. Wooters. April 6–10, 2003. Meetings about Meetings: Research at ICSI on Speech in Multiparty Conversations. Paper presented at the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP '03), Hong Kong.

Mueller, Erik T. March 1987. Daydreaming and Computation. PhD dissertation, University of California, Los Angeles.

Navigli, Roberto, and Paola Velardi. July 2005. Structural semantic interconnections: a knowledge-based approach to word sense disambiguation. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 27(7):1075–1086.

Nejad, Arash Shaban. April 2010. A Framework for Analyzing Changes in Health Care Lexicons and Nomenclatures. PhD dissertation, Concordia University, Montreal, QC, Canada.

Nicolelis, Miguel A. L., Johan Wessberg, Christopher R. Stambaugh, Jerald D. Kralik, Pamela D. Beck, Mark Laubach, John K. Chapin, and Jung Kim. 2000. Real-time prediction of hand trajectory by ensembles of cortical neurons in primates. *Nature* 408(6810):361.

Pepitone, Julianne. 2011. IBM's Jeopardy supercomputer beats humans in practice bout. *CNNMoney*. http://money.cnn.com/2011/01/13/technology/ibm_jeopardy_watson. Accessed January 13.

Phillips, P. Jonathon, and J. Ross Beveridge. September 28–30, 2009. An Introduction to Biometric-Completeness: The Equivalence of Matching and Quality. Paper presented at the IEEE 3rd International Conference on Biometrics: Theory, Applications, and Systems (BTAS '09), Washington, DC.

Raymond, Eric S. March 22, 1991. Jargon File Version 2.8.1. http://catb.org/esr/jargon/oldversions/jarg282.txt

Salloum, W. November 30, 2009. A Question Answering System Based on Conceptual Graph Formalism. Paper presented at the 2nd International Symposium on Knowledge Acquisition and Modeling (KAM 2009), Wuhan, China.

Sandberg, Anders, and Nick Boström. 2008. Whole Brain Emulation: A Roadmap. Technical Report 2008-3. Future of Humanity Institute, Oxford University. http://www.fhi.ox.ac.uk/Reports/2008-3.pdf

Searle, John. 1980. Minds, brains and programs. *Behavioral and Brain Sciences* 3(3):417–457.

Shahaf, Dafna, and Eyal Amir. March 26–28, 2007. Towards a Theory of AI Completeness. Paper presented at the 8th International Symposium on Logical Formalizations of Commonsense Reasoning (Commonsense 2007), Stanford University, Stanford, CA.

Shahaf, D., and E. Horvitz. July 2010. Generalized Task Markets for Human and Machine Computation. Paper presented at the Twenty-Fourth AAAI Conference on Artificial Intelligence, Atlanta, GA.

Shapiro, Stuart C. 1992. Artificial Intelligence. In *Encyclopedia of Artificial Intelligence*, edited by Stuart C. Shapiro, 54–57. New York: Wiley.

Shieber, Stuart M. July 16–20, 2006. Does the Turing Test Demonstrate Intelligence or Not? Paper presented at the Twenty-First National Conference on Artificial Intelligence (AAAI-06), Boston.

Shieber, Stuart M. December 2007. The Turing test as interactive proof. *Nous* 41(4):686–713.

Surowiecki, James. 2004. *The Wisdom of Crowds: Why the Many Are Smarter Than the Few and How Collective Wisdom Shapes Business, Economies, Societies and Nations.* New York: Little, Brown.

Takagi, H. 2001. Interactive evolutionary computation: fusion of the capacities of EC optimization and human evaluation. *Proceedings of the IEEE 89* 9:1275–1296.

Talwar, Sanjiv K., Shaohua Xu, Emerson S. Hawley, Shennan A. Weiss, Karen A. Moxon, and John K. Chapin. May 2, 2002. Behavioural neuroscience: rat navigation guided by remote control. *Nature* 417:37–38.

Taylor, P., and A. Black. 1999. Speech Synthesis by Phonological Structure Matching. Paper presented at Eurospeech99, Budapest, Hungary.

Turing, A. 1950. Computing machinery and intelligence. *Mind* 59(236):433–460.

Turing, Alan M. 1936. On computable numbers, with an application to the Entscheidungs problem. *Proceedings of the London Mathematical Society* 42:230–265.

Vaas, Lisa. December 1, 2007. Striptease used to recruit help in cracking sites. *PC Magazine*. http://www.pcmag.com/article2/0,2817,2210671,00.asp

Vidal, J. J. 1973. Toward direct brain-computer communication. *Annual Review of Biophysics and Bioengineering* 2:157–180.

Yampolskiy, R. V. 2011. AI-Complete CAPTCHAs as zero knowledge proofs of access to an artificially intelligent system. *ISRN Artificial Intelligence* 2012:271878.

Yampolskiy, Roman V. April 13, 2007a. Embedded CAPTCHA for Online Poker. Paper presented at the 20th Annual CSE Graduate Conference (Grad-Conf2007), Buffalo, NY.

Yampolskiy, Roman V. September 28, 2007b. Graphical CAPTCHA Embedded in Cards. Paper presented at the Western New York Image Processing Workshop (WNYIPW)—IEEE Signal Processing Society, Rochester, NY.

Yampolskiy, Roman V. October 3–4, 2011a. Artificial Intelligence Safety Engineering: Why Machine Ethics Is a Wrong Approach. Paper presented at Philosophy and Theory of Artificial Intelligence (PT-AI2011), Thessaloniki, Greece.

Yampolskiy, Roman V. October 3–4, 2011b. What to Do with the Singularity Paradox? Paper presented at Philosophy and Theory of Artificial Intelligence (PT-AI2011), Thessaloniki, Greece.

Yampolskiy, Roman V. April 21–22, 2012a. AI-Complete, AI-Hard, or AI-Easy—Classification of Problems in AI. Paper presented at the 23rd Midwest Artificial Intelligence and Cognitive Science Conference, Cincinnati, OH.

Yampolskiy, Roman V. 2012b. Leakproofing singularity—artificial intelligence confinement problem. *Journal of Consciousness Studies (JCS)* 19(1–2):194–214.

Yampolskiy, Roman V., and Joshua Fox. 2012. Artificial general intelligence and the human mental model. In *In the Singularity Hypothesis: A Scientific and Philosophical Assessment*, edited by Amnon Eden, Jim Moor, Johnny Soraker, and Eric Steinhart, 129–146. New York: Springer.

Yampolskiy, Roman V., and Venu Govindaraju. 2007. Embedded non-interactive continuous bot detection. *ACM Computers in Entertainment* 5(4):1–11.