8.2.2 Example: XOR

We motivated the SVM by thinking about how we solved the XOR function in Section 3.4.3. So will the SVM actually solve the problem? We'll need to modify the problem to have targets -1 and 1 rather than 0 and 1, but that is not difficult. Then we'll introduce a basis of all terms up to quadratic in our two features: $1, \sqrt{2}x_1, \sqrt{2}x_2, x_1x_2, x_1^2, x_2^2$, where the $\sqrt{2}$ is to keep the multiplications simple. Then Equation (8.9) looks like:

$$\sum_{i=1}^{4} \lambda_i - \sum_{i,j}^{4} \lambda_i\lambda_j t_i t_j \Phi(\mathbf{x}_i)^T \Phi(\mathbf{x}_j), \tag{8.23}$$

subject to the constraints that $\lambda_1 - \lambda_2 + \lambda_3 - \lambda_4 = 0, \lambda_i \geq 0 \; i = 1 \ldots 4$. Solving this (which can be done algebraically) tells us that the classifier line is at $x_1x_2 = 0$. The margin that corresponds to this is $\sqrt{2}$. Unfortunately we can't plot it, since our four points have been transferred into a six-dimensional space. We know that this is not the smallest number that it can be solved in, since we did it in three dimensions in Section 3.4.3, but the dimensionality of the kernel space doesn't matter, as all the computations are in the 2D space anyway.

8.3 THE SUPPORT VECTOR MACHINE ALGORITHM

Quadratic programming solvers tend to be very complex (lots of the work is in identifying the active set), and we would be a long way off topic if we tried to write one. Fortunately, general purpose solvers have been written, and so we can take advantage of this. We will use cvxopt, which is a convex optimisation package that includes a wrapper for Python. There is a link to the relevant website on the book webpage. Cvxopt has a nice and clean interface so we can use this to do the computational heavy lifting for an implementation of the SVM. In essence, the approach is fairly simple: we choose a kernel and then for given data, assemble the relevant quadratic problem and its constraints as matrices, and then pass them to the solver, which finds the decision boundary and necessary support vectors for us. These are then used to build a classifier for that training data. This is given as an algorithm next, and then some parts of the implementation are highlighted, particularly those parts where some speed-up can be achieved by some linear algebra.

The Support Vector Machine Algorithm

- **Initialisation**
  - for the specified kernel, and kernel parameters, compute the kernel of distances between the datapoints
    - the main work here is the computation $\mathbf{K} = \mathbf{X}\mathbf{X}^T$
    - for the linear kernel, return $\mathbf{K}$, for the polynomial of degree d return $\frac{1}{\sigma}\mathbf{K}^d$
    - for the RBF kernel, compute $\mathbf{K} = \exp(-(\mathbf{x} - \mathbf{x}')^2/2\sigma^2)$
- **Training**
  - assemble the constraint set as matrices to solve:

$$\min_{\mathbf{x}} \frac{1}{2}\mathbf{x}^T t_i t_j \mathbf{K}\mathbf{x} + \mathbf{q}^T\mathbf{x} \text{ subject to } \mathbf{G}\mathbf{x} \leq \mathbf{h}, \mathbf{A}\mathbf{x} = b$$

  - pass these matrices to the solver

  - identify the support vectors as those that are within some specified distance of the closest point and dispose of the rest of the training data
  - compute b^* using equation (8.10)

- **Classification**
  - for the given test data $\mathbf{z}$, use the support vectors to classify the data for the relevant kernel using:
    * compute the inner product of the test data and the support vectors
    * perform the classification as $\sum_{i=1}^{n} \lambda_i t_i \mathbf{K}(\mathbf{x}_i, \mathbf{z}) + b^*$, returning either the label (hard classification) or the value (soft classification)

8.3.1 Implementation

In order to use the code on the website it is necessary to install the cvxopt package on your computer. There is a link to this on the website. However, we need to work out exactly what we are trying to solve. The key is Equation (8.9), which shows the dual problem, which had constraints $\lambda_i \geq 0$ and $\sum_{i=1}^{n} \lambda_i t_i = 0$. We need to modify it so that we are dealing with the case for slack variables, and using a kernel. Introducing slack variables changes this surprisingly little, basically swapping the first constraint to be $0 \leq \lambda_i \leq C$, while adding the kernel simply turns $\mathbf{x}_i^T \mathbf{x}_j$ into $\mathbf{K}(\mathbf{x}_i, \mathbf{x}_j)$. So we want to solve:

$$\max_{\boldsymbol{\lambda}} \quad = \sum_{i=1}^{n} \lambda_i - \frac{1}{2} \boldsymbol{\lambda}^T \boldsymbol{\lambda} \mathbf{t}\mathbf{t}^T \mathbf{K}(\mathbf{x}_i, \mathbf{x}_j) \boldsymbol{\lambda}, \tag{8.24}$$

$$\text{subject to} \quad 0 \leq \lambda_i \leq C, \sum_{i=1}^{n} \lambda_i t_i = 0. \tag{8.25}$$

The cvxopt quadratic program solver is `cvxopt.solvers.qp()`. This method takes the following inputs `cvxopt.solvers.qp(P, q, G, h, A, b)` and then solves:

$$\min \frac{1}{2} \mathbf{x}^T \mathbf{P} \mathbf{x} + \mathbf{q}^T \mathbf{x} \text{ subject to } \mathbf{G}\mathbf{x} \leq \mathbf{h}, \mathbf{A}\mathbf{x} = b, \tag{8.26}$$

where $\mathbf{x}$ is the variable we are solving for, which is $\boldsymbol{\lambda}$ for us. Note that this solves minimisation problems, whereas we are doing maximisation, which means that we need to multiply the objective function by -1. To make the equations match we set $\mathbf{P} = t_i t_j \mathbf{K}$ and $\mathbf{q}$ is just a column vector containing -1s. The second constraint is easy, since if $\mathbf{A} = \boldsymbol{\lambda}$ then we get the right equation. However, for the first constraint we need to do a little bit more work, since we want to include two constraints ($0 \leq \lambda_i$ and $\lambda_i \leq C$). To do this, we double up on the number of constraints, multiplying the ones where we want $\geq$ instead of $\leq$ by -1. In order to do this multiplication efficiently, it will also be better to use a matrix with the elements on the diagonal, so that we make the following matrix:

$$\begin{pmatrix} t_1 & 0 & \dots & 0 \\ 0 & t_2 & \dots & 0 \\ & & \dots & \\ 0 & 0 & \dots & t_n \\ -t_1 & 0 & \dots & 0 \\ 0 & -t_2 & \dots & 0 \\ & & \dots & \\ 0 & 0 & \dots & -t_n \end{pmatrix} \begin{pmatrix} \lambda_1 \\ \lambda_2 \\ \dots \\ \lambda_n \end{pmatrix} = \begin{pmatrix} C \\ C \\ \dots \\ C \\ 0 \\ 0 \\ \dots \\ 0 \end{pmatrix} \tag{8.27}$$

Assembling these, turning them into the matrices expected by the solver, and then calling it can then be written as:

```
# Assemble the matrices for the constraints
P = targets*targets.transpose()*self.K
q = -np.ones((self.N,1))
if self.C is None:
    G = -np.eye(self.N)
    h = np.zeros((self.N,1))
else:
    G = np.concatenate((np.eye(self.N),-np.eye(self.N)))
    h = np.concatenate((self.C*np.ones((self.N,1)),np.zeros((self.N,1))))
A = targets.reshape(1,self.N)
b = 0.0

# Call the quadratic solver
sol = cvxopt.solvers.qp(cvxopt.matrix(P),cvxopt.matrix(q),cvxopt.matrix(G),↩
cvxopt.matrix(h), cvxopt.matrix(A), cvxopt.matrix(b))
```

There are a couple of novelties in the implementation. One is that the training method actually returns a function that performs the classification, as can be seen here for the polynomial kernel:

```
if self.kernel == 'poly':
    def classifier(Y,soft=False):
        K = (1. + 1./self.sigma*np.dot(Y,self.X.T))**self.degree

        self.y = np.zeros((np.shape(Y)[0],1))
        for j in range(np.shape(Y)[0]):
            for i in range(self.nsupport):
                self.y[j] += self.lambdas[i]*self.targets[i]*K[j,i]
            self.y[j] += self.b

        if soft:
            return self.y
        else:
            return np.sign(self.y)
```

The reason for this is that the classification function has different forms for the different kernels, and so we need to create this function based on the kernel that is specified. A handle for the classifier is stored in the class, and the method can then be called as:

```
output = sv.classifier(Y,soft=False)
```

The other novelty is that some of the computation of the RBF kernel uses some linear algebra to make the computation faster, since NumPy is better at dealing with matrix manipulations than loops. The elements of the RBF kernel are $K_{ij} = \frac{1}{2\sigma}\exp(-\|x_i - x_j\|^2)$. We could go about forming this by using a pair of loops over i and j, but instead we can use some algebra.

The linear kernel has computed $K_{ij} = \mathbf{x}_i^T\mathbf{x}_j$, and the diagonal elements of this matrix are $\|\mathbf{x}_i\|^2$. The trick is to see how to use only these elements to compute the $\|\mathbf{x}_i - \mathbf{x}_j\|^2$ part, and it just requires expanding out the quadratic:

$$(\mathbf{x}_i - \mathbf{x}_j)^2 = \|\mathbf{x}_i\|^2 + \|\mathbf{x}_j^2\| - 2\mathbf{x}_i^T\mathbf{x}_j. \tag{8.28}$$

The only work involved now is to make sure that the matrices are the right shape. This would be easy if it wasn't for the fact that NumPy 'loses' the dimension of some $N \times 1$ matrices, so that they are of size N only, as we have seen before. This means that we need to make a matrix of ones and use the transpose operator a few times, as can be seen in the code fragment below.

```
self.xsquared = (np.diag(self.K)*np.ones((1,self.N))).T
b = np.ones((self.N,1))
self.K -= 0.5*(np.dot(self.xsquared,b.T) + np.dot(b,self.xsquared.T))
self.K = np.exp(self.K/(2.*self.sigma**2))
```

For the classifier we can use the same tricks to compute the product of the kernel and the test data:

```
elif self.kernel == 'rbf':
    def classifier(Y,soft=False):
        K = np.dot(Y,self.X.T)
        c = (1./self.sigma * np.sum(Y**2,axis=1)*np.ones((1,np.shape(Y)[0])))↩
        .T
        c = np.dot(c,np.ones((1,np.shape(K)[1])))
        aa = np.dot(self.xsquared[self.sv],np.ones((1,np.shape(K)[0]))).T
        K = K - 0.5*c - 0.5*aa
        K = np.exp(K/(2.*self.sigma**2))

        self.y = np.zeros((np.shape(Y)[0],1))
        for j in range(np.shape(Y)[0]):
            for i in range(self.nsupport):
                self.y[j] += self.lambdas[i]*self.targets[i]*K[j,i]
```

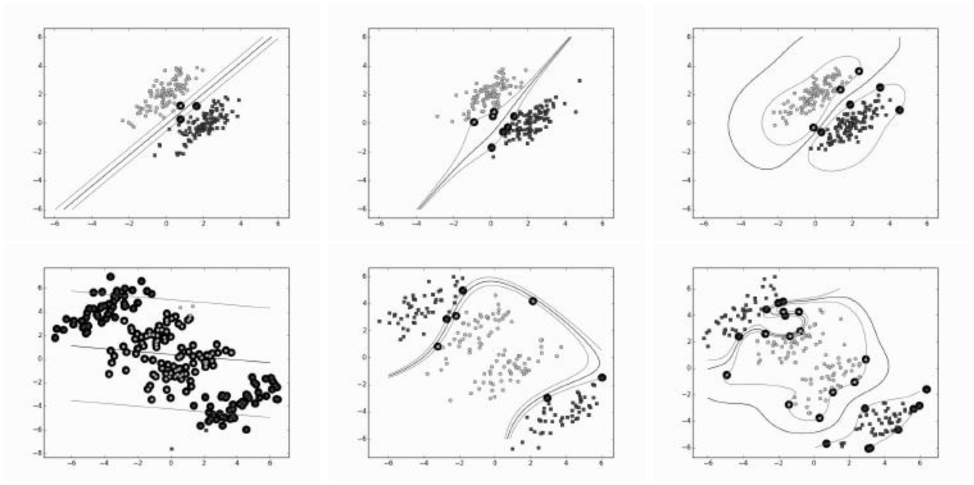
FIGURE 8.7 The SVM learning about a linearly separable dataset (*top row*) and a dataset that needs two straight lines to separate in 2D (*bottom row*) with *left* the linear kernel, *middle* the polynomial kernel of degree 3, and *right* the RBF kernel. $C = 0.1$ in all cases.

```
                self.y[j] += self.b

        if soft:
                return self.y
        else:
                return np.sign(self.y)
```

The first bit of computational work is in computing the kernel (which is $\mathcal{O}(m^2n)$, where m is the number of datapoints and n is the dimensionality), and the second part is inside the solver, which has to factorise a sum of the kernel matrix and a test matrix at each iteration. Factorisation costs $\mathcal{O}(m^3)$ in general, and this is why the SVM is very expensive to use for large datasets. There are some methods by which this can be improved, and there are some references to this at the end of the chapter.

8.3.2 Examples

In order to see the SVM working, and to identify the differences between the kernels, we will start with some very simple 2D datasets with two classes.

The first example (shown on the top row of Figure 8.7) simply checks that the SVM can learn accurately about data that is linearly separable, which it does successfully. Note that the different kernels produce different decision boundaries, which are not straight lines in the 2D plot for the polynomial kernel (centre) and RBF kernel (right), and that different numbers of support vectors (highlighted in bold) are needed for the different kernels as well.

On the second line of the figure is a dataset that cannot be separated by a single straight line, and which the linear kernel cannot then separate. However, the polynomial and RBF kernels deal with this data successfully with very few support vectors.

For the second example the data come from the XOR dataset with some spread around each of the four datapoints. The dataset is made by making four sets of random Gaussian samples with a small standard deviation, and means of $(0,0),(0,1),(1,0)$, and $(1,1)$. Figure 8.8 shows a series of outputs from this dataset with the standard deviations of each cluster being 0.1 on the left, 0.3 in the middle, and 0.4 on the right, and with 100 datapoints for training, and 100 datapoints for testing. The training set for the two classes is shown as black and white circles, with the support vectors marked with a thicker outline. The test set are shown as black and white squares.

The top row of the figure shows the polynomial kernel of degree 3 with no slack variables, while the second row shows the same kernel but with $C = 0.1$; the third row shows the RBF kernel with no slack variables, and the last row shows the RBF kernel with $C = 0.1$. It can be seen that where the classes start to overlap, the inclusion of slack variables leads to far simpler decision boundaries and a better model of the underlying data. Both the polynomial and RBF kernels perform well on this problem.

8.4 EXTENSIONS TO THE SVM

8.4.1 Multi-Class Classification

We've talked about SVMs in terms of two-class classification. You might be wondering how to use them for more classes, since we can't use the same methods as we have done to work out the current algorithm. In fact, you can't actually do it in a consistent way. The SVM only works for two classes. This might seem like a major problem, but with a little thought it is possible to find ways around the problem. For the problem of N-class classification, you train an SVM that learns to classify class one from all other classes, then another that classifies class two from all the others. So for N-classes, we have N SVMs. This still leaves one problem: how do we decide which of these SVMs is the one that recognises the particular input? The answer is just to choose the one that makes the strongest prediction, that is, the one where the basis vector input point is the furthest into the positive class region. It might not be clear how to work out which is the strongest prediction. The classifier examples in the code snippets return either the class label (as the sign of y) or the value of y, and this value of y is telling us how far away from the decision boundary it is, and clearly it will be negative if it is a misclassification. We can therefore use the maximum value of this soft boundary as the best classifier.

```
output = np.zeros((np.shape(test)[0],3))
output[:,0] = svm0.classifier(test[:,:2],soft=True).T
output[:,1] = svm1.classifier(test[:,:2],soft=True).T
output[:,2] = svm2.classifier(test[:,:2],soft=True).T

# Make a decision about which class
# Pick the one with the largest margin
bestclass = np.argmax(output,axis=1)
err = np.where(bestclass!=target)[0]
print len(err)/ np.shape(target)[0]
```

Figure 8.9 shows the first two dimensions of the iris dataset and the class decision boundaries for the three classes. It can be seen that using only two dimensions does not

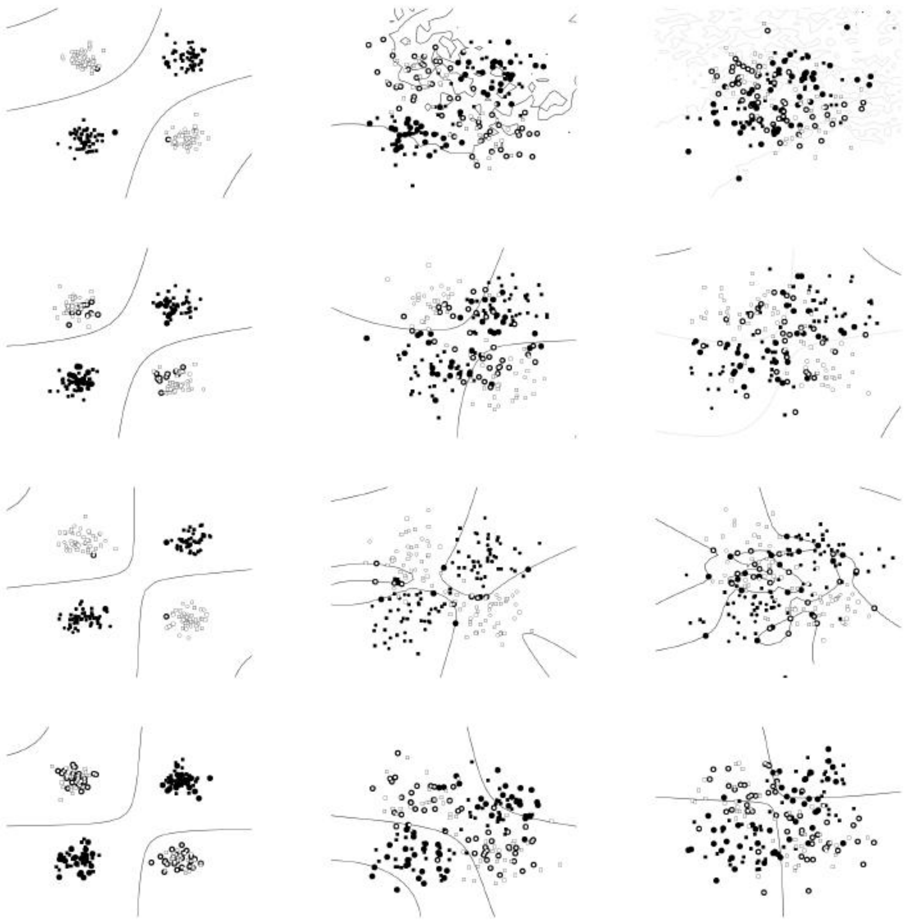

FIGURE 8.8 The effects of different kernels when learning a version of XOR with progressively more overlap (*left* to *right*) between the classes. *Top row:* polynomial kernel of degree 3 with no slack variables, *second row:* polynomial of degree 3 with $C = 0.1$, *third row:* RBF kernel, no slack variables, *bottom row:* RBF kernel with $C = 0.1$. The support vectors are highlighted, and the decision boundary is drawn for each case.

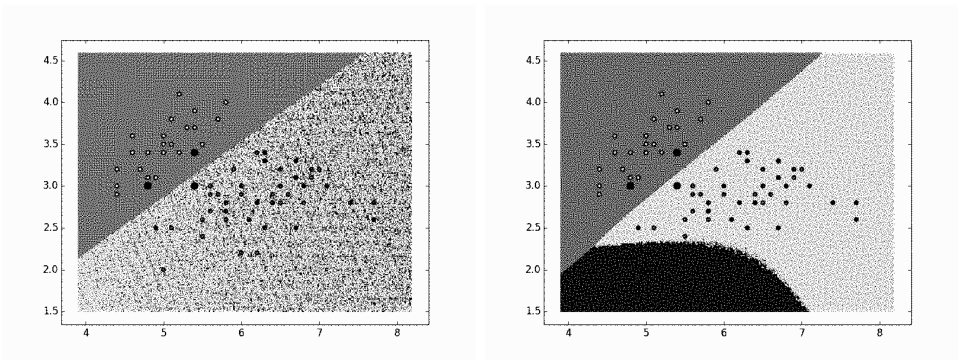

FIGURE 8.9 A linear (*left*) and polynomial, degree 3 (*right*) kernel learning the first two dimensions of the iris dataset, which separates one class very well from the other two, but cannot distinguish between the other two (for good reason). The support vectors are highlighted.

allow good separation of the data, and both kernels get about 33% accuracy, but allowing for all four dimensions, both the RBF and polynomial kernels reliably get about 95% accuracy.

8.4.2 SVM Regression

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

$$\frac{1}{2}\sum_{i=1}^{N}(t_i - y_i)^2 + \frac{1}{2}\boldsymbol{\lambda}\|\mathbf{w}\|^2, \tag{8.29}$$

and transform it using what is known as an ϵ-insensitive error function (E_ϵ) that gives 0 if the difference between the target and output is less than ϵ (and subtracts ϵ in any other case for consistency). The reason for this is that we still want a small number of support vectors, so we are only interested in the points that are not well predicted. Figure 8.10 shows the form of this error function, which is:

$$\sum_{i=1}^{N} E_\epsilon(t_i - y_i) + \boldsymbol{\lambda}\frac{1}{2}\|\mathbf{w}\|^2. \tag{8.30}$$

You might see this written in other texts with the constant $\boldsymbol{\lambda}$ in front of the second term replaced by a C in front of the first term. This is equivalent up to scaling. The picture to think of now is almost the opposite of Figure 8.3: we want the predictions to be inside the tube of radius ϵ that surrounds the correct line. To allow for errors, we again introduce slack variables for each datapoint (ϵ_i for datapoint i) with their constraints and follow the same procedure of introducing Lagrange multipliers, transferring to the dual problem, using a kernel function and solving the problem with a quadratic solver.

The upshot of all this is that the prediction we make for test point $\mathbf{z}$ is:

$$f(\mathbf{z}) = \sum_{i=1}^{n}(\mu_i - \lambda_i K(\mathbf{x}_i, \mathbf{z}) + b), \tag{8.31}$$

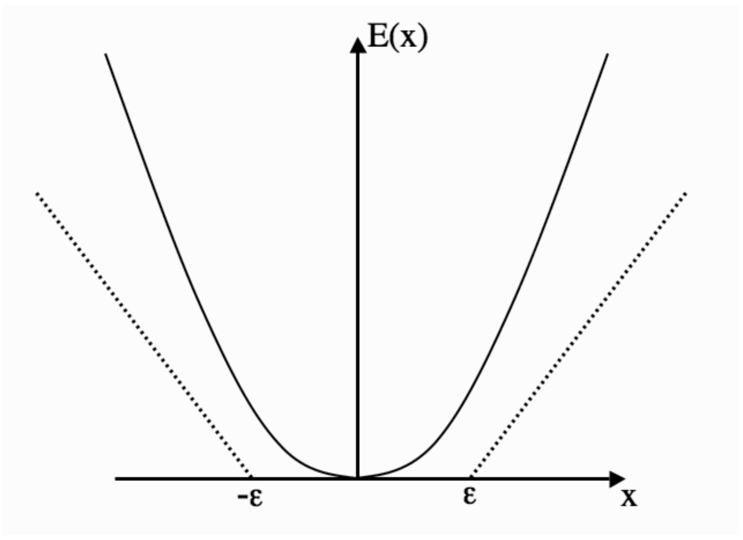

FIGURE 8.10 The ϵ-insensitive error function is zero for any error below ϵ.

where μ_i and λ_i are two sets of constraint variables.

8.4.3 Other Advances

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

There are several SVM implementations available via the Internet that are more advanced than the implementation on the book website. They are mostly written in C, but some include wrappers to be called from other languages, including Python. An Internet search will find you some possibilities to try, but some common choices are SVMLight, LIBSVM, and scikit-learn.

FURTHER READING

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

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

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

- V. Vapnik. *The Nature of Statistical Learning Theory.* Springer, Berlin, Germany, 1995.
- B. Schölkopf, C.J.C. Burges, and A.J. Smola. *Advances in Kernel Methods: Support Vector Learning.* MIT Press, Cambridge, MA, USA, 1999.

- J. Shawe-Taylor and N. Cristianini. *Kernel Methods for Pattern Analysis.* Cambridge University Press, Cambridge, UK, 2004.

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

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

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

- Chapter 12 of T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*, 2nd edition, Springer, Berlin, Germany, 2008.
- Chapter 7 of C.M. Bishop. *Pattern Recognition and Machine Learning.* Springer, Berlin, Germany, 2006.

PRACTICE QUESTIONS

Problem 8.1 Suppose that the following are a set of points in two classes:

$$\text{class 1} \quad : \quad \begin{pmatrix} 1 \\ 1 \end{pmatrix} \begin{pmatrix} 1 \\ 2 \end{pmatrix} \begin{pmatrix} 2 \\ 1 \end{pmatrix} \tag{8.32}$$

$$\text{class 2} \quad : \quad \begin{pmatrix} 0 \\ 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} \tag{8.33}$$

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

Problem 8.2 Suppose that the points are now:

$$\text{class 1} \quad : \quad \begin{pmatrix} 0 \\ 0 \end{pmatrix} \begin{pmatrix} 1 \\ 2 \end{pmatrix} \begin{pmatrix} 2 \\ 1 \end{pmatrix} \tag{8.34}$$

$$\text{class 2} \quad : \quad \begin{pmatrix} 1 \\ 1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} \tag{8.35}$$

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

Problem 8.3 Apply it to the `wine` dataset, trying out the different kernels. Compare the results to using an MLP. Do the same for the `yeast` dataset.

Problem 8.4 Use an SVM on the MNIST dataset.

Problem 8.5 Verify that introducing the slack variables does not change the dual problem much at all (only changing the constraint to be $0 \leq \lambda_i \leq C$). Start from Equation (8.12) and introduce the Lagrange multipliers and then compare the result to Equations (8.9).

CHAPTER 9

Optimisation and Search

In almost all of the algorithms that we've looked at in the previous chapters there has been some element of optimisation, generally by defining some sort of error function, and attempting to minimise it. We've talked about gradient descent, which is the optimisation method that forms the basis of many machine learning algorithms. In this chapter, we will look at formalising the gradient descent algorithm and understanding how it works, and then we will look at what we can do when there are no gradients in the problem, and so gradient descent doesn't work.

Whatever method we have used to solve the optimisation problem, the basic methodology has been the same: to compute the derivative of the error function to get the gradient and follow it downhill. What if that derivative doesn't exist? This is actually common in many problems—discrete problems are not defined on continuous functions, and hence can't be differentiated, and so gradient descent can't be used. In theory, it is possible to check all of the cases for a discrete problem to find the optimum, but the computations are infeasible for any interesting problem. We therefore need to think of some other approaches. Some examples of discrete problems are:

Chip design Lay a circuit onto a computer chip so that none of the tracks cross.

Timetabling Given a list of courses and which students are on each course, find a timetable with the minimum number of clashes (or given a number of planes and routes, schedule the planes onto the routes).

The Travelling Salesman Problem Given a set of cities, find a tour (that is, a solution that visits every city exactly once, and returns to the starting point) that minimises the total distance travelled.

One thing that is worth noting is that there is no one ideal solution to the search problem. That is, there is no one search algorithm that is guaranteed to perform the best on every problem that is presented to it—you always have to put some work into choosing the algorithm that will be most effective for your problem, and phrasing your problem to make the algorithm work as efficiently as possible. This is called the No Free Lunch theorem.

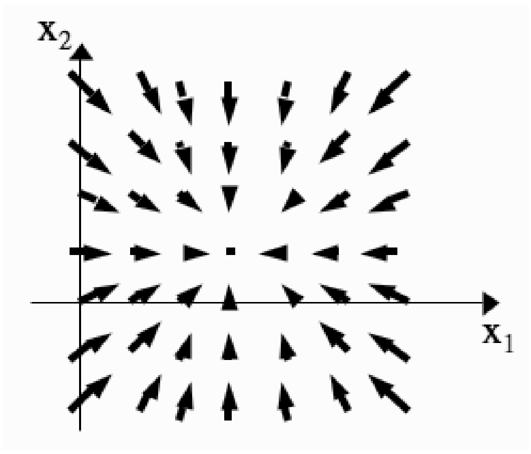

FIGURE 9.1 The downhill gradients to minimise a function. At the solution the gradient is 0. This is a nice example without local minima; they would also have gradient 0.

9.1 GOING DOWNHILL

We will start by trying to derive a better understanding of gradient descent, and seeing the algorithms that can be used for finding local optima for general problems. We will also look at the specific case of solving least-squares optimisation problems, which are the most common examples in machine learning.

The basic idea, as we have already seen, is that we want to minimise a function $f(\mathbf{x})$, where $\mathbf{x}$ is a vector $(x_1, x_2, \ldots, x_n)$ that has elements for each feature value, starting from some initial guess $\mathbf{x}(0)$. We try to find a sequence of new points $\mathbf{x}(i)$ that move downhill towards a solution. The methods that we are going to look at work in any number of dimensions. We will therefore have to take derivatives of the function in each of the different dimensions of $\mathbf{x}$. We write down this whole set of functions as $\nabla f(\mathbf{x})$, which is a vector with elements $(\frac{\partial f}{\partial \mathbf{x}_1}, \frac{\partial f}{\partial \mathbf{x}_2}, \ldots, \frac{\partial f}{\partial \mathbf{x}_n})$, so that it gives us the gradient in each dimension separately. Figure 9.1 shows a set of directions in two dimensions in order to minimise some function.

The first thing to think about is how we know when we have found a solution; in other words, how will we know when to stop? This is relatively easy: it is when $\nabla f = 0$, since then there is no more downhill to go. If you are walking down a hill, then you have reached the bottom when everything is flat around you (which might not be a very large space before things start going up again, but if the function is continuous, as we will assume here, then there must be a point where it is 0 inbetween where it is going down and where it starts going up). So we will know when to terminate the algorithm by checking whether or not $\nabla f = 0$. In practice, the algorithms will always have some numerical inaccuracy, since they are floating point numbers inside the computer, so we usually stop if $|\nabla f| < \epsilon$ where ϵ is some small number, maybe 10^{-5}. There is another concept that it can be useful to think about, which is the places that we can travel to without going up or down, i.e., the places that are at the same level as we are. The full set of places that have the same function value are known as level sets of the function, and some examples are shown in Figure 9.2. Often there will be several discrete parts to a level set, so it is not possible to explore it all without stepping off the set itself.

So from the current point $\mathbf{x}_i$ there are two things that we need to decide: what direction should we move in to go downhill as fast as possible, and how far should we move? Looking

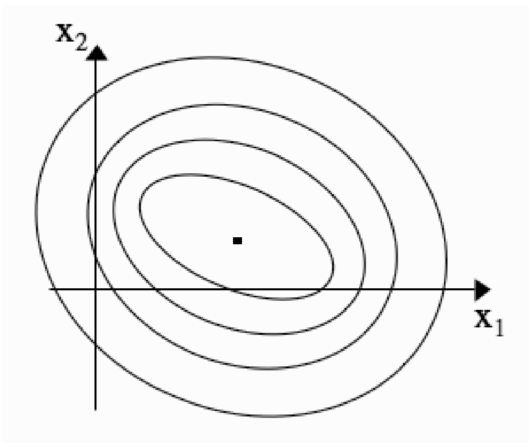

FIGURE 9.2 The lines show contours of equal value (level sets) for a function.

at the second of these questions first, there are two types of methods that can be used to solve it. The simplest approach is a line search: if we know what direction to look in, then we move along it until we reach the minimum in this direction. So this is just a search along the line we are moving along. Writing this down mathematically, if we are currently at $\mathbf{x}_k$ then the next guess will be $\mathbf{x}_{k+1}$, which is:

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k, \tag{9.1}$$

where $\mathbf{p}_k$ is the direction we have chosen to move in and α_k is the distance to travel in that direction, chosen by the line search. Finding a value for α_k can be computationally expensive and inaccurate, so it is generally just estimated.

The other method of choosing how far to move is known as a trust region. It is more complex, since it consists of making a local model of the function as a quadratic form and finding the minimum of that model. We will see one example of a trust region method in Section 9.2, and more information about general trust region methods is available in the books listed at the end of the chapter.

The direction $\mathbf{p}_k$ can also be chosen in several ways. The left of Figure 9.3 shows the ideal situation, which is that we point directly to the minimum, in which case the line search finds it straight away. Since we don't know the minimum (it is what we are trying to find!) this is virtually impossible. One thing that we can do is to make greedy choices and always go downhill as fast as possible at each point. This is known as steepest descent, and it means that $\mathbf{p}_k = -\nabla f(\mathbf{x}_k)$. The problem with it can be seen on the right of Figure 9.3, which is that many of the directions that it travels in are not directly towards the centre. In extreme cases they can be very different: across the valley, rather than down towards the global minimum (we saw this in Figure 4.7).

If we don't worry about the stepsize, and just set it as $\alpha_k = 1$, then we can perform the search using Equation (9.1) with a very simple program. All that is needed is to iterate the line search until the solution stops changing (or you decide that there have been too many iterations). The only other thing that you have to compute is the derivative of the function, which is the direction $\mathbf{p}_k$. This is the problem-specific part of the algorithm, and as a small example, we consider a simple three-dimensional function $f(\mathbf{x}) = (0.5x_1^2 + 0.2x_2^2 + 0.6x_3^2)$. We can differentiate once to compute the vector of derivatives, $\nabla f(\mathbf{x}) = (x_1, 0.4x_2, 1.2x_3)$, which is returned by the `gradient()` function in the code below:

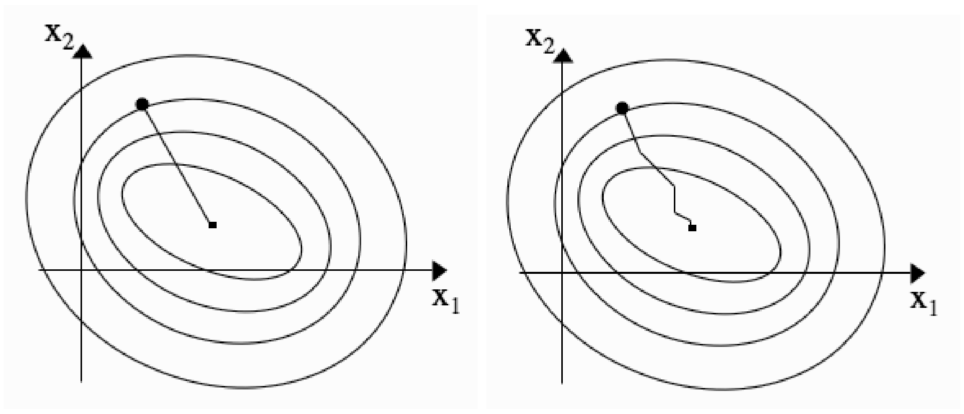

FIGURE 9.3 *Left:* In an ideal world we would know how to go to the minimum directly. In practice, we don't, so we have to approximate it by something like *right:* moving in the direction of steepest descent at each stage.

```
def gradient(x):
    return np.array([x[0], 0.4*x[1], 1.2*x[2]])

def steepest(x0):
    i = 0
    iMax = 10
    x = x0
    Delta = 1
    alpha = 1

    while i<iMax and Delta>10**(-5):
        p = -Jacobian(x)
        xOld = x
        x = x + alpha*p
        Delta = np.sum((x-xOld)**2)
        print x
        i += 1
```

To compute the minimum we now need to pick a start point, for example, $\mathbf{x}(0) = (-2, 2, -2)$, and then we can compute the steepest downhill direction as $(-2, 0.8, -2.4)$. Using the steepest descent method for this example gives fairly poor results, taking several steps before the answer gets close to the correct answer of $(0, 0, 0)$, and even then it is not that close:

```
[ 0. 1.20      0.40]
[ 0. 0.72     -0.08]
[ 0. 0.43      0.01]
[ 0. 0.26     -0.00]
[ 0. 0.16      0.00]
[ 0. 0.09     -0.00]
[ 0. 5.69-02   2.56-05]
```

To see how we can improve on this we need to examine the basics of function approximation.

9.1.1 Taylor Expansion

Steepest descent is based on the Taylor expansion of the function, which is a method of approximating the value of a function at a point in terms of its derivatives: a function $f(\mathbf{x})$ can be approximated by:

$$f(\mathbf{x}) \approx f(\mathbf{x}_0) + \mathbf{J}(f(\mathbf{x}))|_{\mathbf{x}_0}(\mathbf{x}-\mathbf{x}_0) + \frac{1}{2}(\mathbf{x}-\mathbf{x}_0)^T\mathbf{H}(f(\mathbf{x}))|_{\mathbf{x}_0}(\mathbf{x}-\mathbf{x}_0) + \ldots, \tag{9.2}$$

where $\mathbf{x}_0$ is a common, but potentially slightly confusing notation for the initial guess $\mathbf{x}(0)$, the $|_{\mathbf{x}_0}$ notation means that the function is evaluated at that point, and the $\mathbf{J}(\mathbf{x})$ term is the Jacobian, which is the vector of first derivatives:

$$\mathbf{J}(\mathbf{x}) = \frac{\partial f(\mathbf{x})}{\partial \mathbf{x}} = \left(\frac{\partial f(\mathbf{x})}{\partial x_1}, \frac{\partial f(\mathbf{x})}{\partial x_2}, \ldots, \frac{\partial f(\mathbf{x})}{\partial x_n}\right) \tag{9.3}$$

and $\mathbf{H}(\mathbf{x})$ is the Hessian matrix of second derivatives (the Jacobian of the gradient), which for a single function $f(x_1, x_2, \ldots x_n)$ is defined as:

$$\mathbf{H}(\mathbf{x}) = \frac{\partial}{\partial \mathbf{x}_i}\frac{\partial}{\partial \mathbf{x}_j} f(\mathbf{x}) = \begin{pmatrix} \frac{\partial^2 f(\mathbf{x})}{\partial x_1^2} & \frac{\partial^2 f(\mathbf{x})}{\partial x_1 \partial x_2} & \ldots & \frac{\partial^2 f(\mathbf{x})}{\partial x_1 \partial x_n} \\ & & \ldots & \\ \frac{\partial^2 f(\mathbf{x})}{\partial x_n \partial x_1} & \frac{\partial^2 f(\mathbf{x})}{\partial x_n \partial x_2} & \ldots & \frac{\partial f_n(\mathbf{x})}{\partial x_n^2} \end{pmatrix}. \tag{9.4}$$

If $f(\mathbf{x})$ is a scalar function (so that it returns just 1 number) then $\mathbf{J}(\mathbf{x}) = \nabla f(\mathbf{x})$ and is a vector and $\mathbf{H}(\mathbf{x}) = \nabla^2 f(\mathbf{x})$ is a two-dimensional matrix. For a vector $\mathbf{f}(\mathbf{x})$ with components $f_1(\mathbf{x}), f_2(\mathbf{x})$, etc., $\mathbf{J}(\mathbf{x})$ is a two-dimensional matrix and $\mathbf{H}(\mathbf{x})$ is three-dimensional.

If we ignore the Hessian term in Equation (9.2), then for scalar $f(\mathbf{x})$ we get precisely the steepest descent step. However, if we choose to minimise Equation (9.2) exactly as it is written (i.e., ignoring third derivatives and higher), then we find the Newton direction at the kth iteration to be: $\mathbf{p}_k = -(\nabla^2 f(\mathbf{x}_k))^{-1}\nabla f(\mathbf{x}_k)$. There is something important to notice about this equation, which is that we actually use the inverse of the Hessian. Computing this is generally of order $\mathcal{O}(N^3)$ (where N is the number of elements in the matrix) which makes this a computationally expensive method. The compensation for this cost is that we don't really have to worry about the stepsize at all; it is always set to 1.

Implementing this requires only 1 line of change to our basic steepest descent algorithm, plus the addition of a function that computes the Hessian. The line to change is the one that computes $\mathbf{p}_k$, which becomes:

```
p = -np.dot(np.linalg.inv(Hessian(x)),Jacobian(x))
```

For this simple example, this algorithm goes straight to the correct answer in one step, which is much better than the steepest descent method that we saw earlier. However, for more complicated functions it won't work as well, because the estimate of the Hessian is not as accurate. There are particular cases where we can, however, do better, as we shall see.

9.2 LEAST-SQUARES OPTIMISATION

For many of the algorithms that we have derived we have used a least-squares error function, such as the error of the MLP and the linear regressor. Least-squares problems turn out to be the most common optimisation problems in many fields, and this means that they have been very well studied and, fortunately, they have special structure in the problem that makes solving them easier than other problems. This leads to a set of special algorithms for solving least-squares problems, although they are mostly special cases of standard methods. One of these has become very well known, the Levenberg–Marquardt method, which is a trust region optimisation algorithm. We will derive the Levenberg–Marquardt algorithm, beginning by identifying why least-squares optimisation is a special case.

9.2.1 The Levenberg–Marquardt Algorithm

For least-squares problems, the objective function that we are optimising is:

$$f(\mathbf{x}) = \frac{1}{2}\sum_{j=1}^{m} r_j^2(\mathbf{x}) = \frac{1}{2}\|\mathbf{r}(\mathbf{x})\|_2^2, \tag{9.5}$$

where the $\frac{1}{2}$ makes the derivative nicer, and $\mathbf{r}(\mathbf{x}) = (r_1(\mathbf{x}), r_2(\mathbf{x}), \ldots, r_m(\mathbf{x}))^T$. In this last version, we can write the (transpose of the) Jacobian of $\mathbf{r}$ as:

$$\mathbf{J}^T(\mathbf{x}) = \left\{ \begin{array}{cccc} \frac{\partial r_1}{\partial x_1} & \frac{\partial r_2}{\partial x_1} & \cdots & \frac{\partial r_m}{\partial x_1} \\ \frac{\partial r_1}{\partial x_2} & \frac{\partial r_2}{\partial x_2} & \cdots & \frac{\partial r_m}{\partial x_2} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial r_1}{\partial x_n} & \frac{\partial r_2}{\partial x_n} & \cdots & \frac{\partial r_m}{\partial x_n} \end{array} \right\} = \left[\frac{\partial r_j}{\partial x_i}\right]_{j=1,\ldots,m,\ i=1,\ldots,n}, \tag{9.6}$$

which is useful because the function gradients that we want can mostly be computed directly:

$$\nabla f(\mathbf{x}) = \mathbf{J}(\mathbf{x})^T \mathbf{r}(\mathbf{x}) \tag{9.7}$$

$$\nabla^2 f(\mathbf{x}) = \mathbf{J}(\mathbf{x})^T \mathbf{J}(\mathbf{x}) + \sum_{j=1}^{m} r_j(\mathbf{x}) \nabla^2 r_j(\mathbf{x}). \tag{9.8}$$

The upshot of this is that knowing the Jacobian gives you the first (and usually, most important) part of the Hessian effectively without any additional computational cost, and it is this that special algorithms can exploit to solve least-squares problems efficiently. To see this, remember that, as in all of the other gradient-descent algorithms that we have looked at, we are approximating the function by the Taylor series (Equation (9.2)) up to second-order (Hessian) terms.

If $\|\mathbf{r}(\mathbf{x})\|$ is a linear function of $\mathbf{x}$ (which means that $f(\mathbf{x})$ is quadratic), then the Jacobian is constant and $\nabla^2 r_j(\mathbf{x}) = 0$ for all j. In this case, substituting Equations (9.7) and (9.8) into Equation (9.2) and taking derivatives, we see that at a solution:

$$\nabla f(\mathbf{x}) = \mathbf{J}^T(\mathbf{J}\mathbf{x} + \mathbf{r}) = 0, \tag{9.9}$$

and so:

$$\mathbf{J}^T\mathbf{J}\mathbf{x} = -\mathbf{J}^T\mathbf{r}(\mathbf{x}). \tag{9.10}$$

This is a linear least-squares problem and can be solved. In an ideal world we would be able to see that it is effectively just the statement $\mathbf{A}\mathbf{x} = \mathbf{b}$ (where $\mathbf{A} = \mathbf{J}^T\mathbf{J}$ is a square matrix and $\mathbf{b} = -\mathbf{J}^T\mathbf{r}(\mathbf{x})$ and so solve it directly as:

$$\mathbf{x} = -(\mathbf{J}^T\mathbf{J})^{-1}\mathbf{J}^T\mathbf{r}. \tag{9.11}$$

However, this is computationally expensive and numerically very unstable, so we need to use linear algebra to find $\mathbf{x}$ in a variety of different ways, such as Cholesky factorisation, QR factorisation, or using the Singular Value Decomposition. We will look at the last of these methods, since it uses eigenvectors, which we have already seen in Chapter 6, although we will see the first method in Chapter 18.

The Singular Value Decomposition (SVD) is the decomposition of a matrix $\mathbf{A}$ of size $m \times n$ into:

$$\mathbf{A} = \mathbf{U}\mathbf{S}\mathbf{V}^T, \tag{9.12}$$

where $\mathbf{U}$ and $\mathbf{V}$ are orthogonal matrices (i.e., the inverse of the matrix is its transpose, so $\mathbf{U}^T\mathbf{U} = \mathbf{U}\mathbf{U}^T = \mathbf{I}$, where $\mathbf{I}$ is the identity matrix). $\mathbf{U}$ is of size $m \times m$ and $\mathbf{V}$ is of size $n \times n$. $\mathbf{S}$ is a diagonal matrix of size $m \times n$, with the elements of this matrix, σ_i, being known as singular values.

To apply this to the linear least-squares problem we compute the SVD of $\mathbf{J}^T\mathbf{J}$ and substitute it into Equation (9.11):

$$\begin{aligned} \mathbf{x} &= \left[(\mathbf{U}\mathbf{S}\mathbf{V}^T)^T(\mathbf{U}\mathbf{S}\mathbf{V}^T)\right]^{-1}(\mathbf{U}\mathbf{S}\mathbf{V}^T)^T\mathbf{J}^T\mathbf{r} && (9.13)\\ &= \mathbf{V}\mathbf{S}\mathbf{U}^T\mathbf{J}^T\mathbf{r} && (9.14) \end{aligned}$$

using the fact that $\mathbf{A}\mathbf{B}^T = \mathbf{B}^T\mathbf{A}^T$ and similar linear algebraic identities.

We can actually go a bit further, and deal with the fact that $\mathbf{J}$ is probably not a square matrix. The size of the various matrices will be $m \times m$ for $\mathbf{U}$ and $m \times m$ for the other two (where m and n are defined in Equation (9.6); generally $n < m$). We can split $\mathbf{U}$ into two parts, $\mathbf{U}_1$ of size $m \times n$ and then the last few columns into a second part $\mathbf{U}_2$ of size $(n - m) \times n$. This lets us solve the linear least-squares equation as:

$$\mathbf{x} = \mathbf{V}\mathbf{S}^{-1}\mathbf{U}_1^T\mathbf{J}\mathbf{r}. \tag{9.15}$$

NumPy has an algorithm for linear least-squares in `np.linalg.lstsq()` and can compute the SVD decomposition using `np.linalg.svd()`.

We can now use this derivation to look at the most well-known method for solving non-linear least-squares problems, the Levenberg–Marquardt algorithm. The principal approximation that the algorithm makes is to ignore the residual terms in Equation (9.8), making each iteration a linear least-squares problem, so that $\nabla^2 f(\mathbf{x}) = \mathbf{J}(\mathbf{x})^T\mathbf{J}(\mathbf{x})$. Then the problem to be solved is:

$$\min_{\mathbf{p}} \frac{1}{2}\|\mathbf{J}_k\mathbf{p} + \mathbf{r}_k\|_2^2, \quad \|\mathbf{p}\| \leq \Delta_k, \tag{9.16}$$

where Δ_k is the radius of the trust region, which is the region where it is assumed that this approximation holds well. In normal trust region methods, the size of the region (Δ_k) is controlled explicitly, but in Levenberg–Marquardt it is used to control a parameter $\nu \geq 0$ that is added to the diagonal elements of the Jacobian matrix and is known as the damping factor. The minimum $\mathbf{p}$ then satisfies:

$$(\mathbf{J}^T\mathbf{J} + \nu\mathbf{I})\mathbf{p} = -\mathbf{J}^T\mathbf{r}. \tag{9.17}$$

This is a very similar equation to the one that we solved for the linear least-squares method, and so we can just use that solver here; effectively non-linear least-squares solvers solve a lot of linear problems to find the non-linear solution. There are very efficient Levenberg–Marquardt solvers, since it is possible to avoid computing the $\mathbf{J}^T\mathbf{J}$ term explicitly using the SVD composition that we worked out above.

The basic idea of the trust region method is to assume that the solution is quadratic about the current point, and use that assumption to minimise the current step. You then compute the difference between the actual reduction and the predicted one, based on the model, and make the trust region larger or smaller depending upon how well these two match, and if they do not match at all, then you reject that update. The Levenberg–Marquardt algorithm itself is very general, but it needs to have the function to be minimised, along with its gradient and Jacobian passed into it. The entire algorithm can be written as:

The Levenberg–Marquardt Algorithm

```
• Given start point $\mathbf{x}_0$
• While $\mathbf{J}^T\mathbf{r}(\mathbf{x})$ >tolerance and maximum number of iterations not exceeded:
    – repeat
        ∗ solve $(\mathbf{J}^T\mathbf{J} + \nu\mathbf{I})\mathbf{dx} = -\mathbf{J}^T\mathbf{r}$ for $\mathbf{dx}$ using linear least-squares
        ∗ set $\mathbf{x}_{\text{new}} = \mathbf{x} + \mathbf{dx}$
        ∗ compute the ratio of the actual and prediction reductions:
            · actual = $\|f(\mathbf{x}) - f(\mathbf{x}_{new})\|$
            · predicted = $\nabla f^T(\mathbf{x}) \times \mathbf{x}_{new} - \mathbf{x}$
            · $\rho$ = actual/predicted
        ∗ if $0 < \rho < 0.25$:
            · accept step: $\mathbf{x} = \mathbf{x}_{\text{new}}$
        ∗ else if $\rho > 0.25$:
            · accept step: $\mathbf{x} = \mathbf{x}_{\text{new}}$
            · increase trust region size (reduce $\nu$)
        ∗ else:
            · reject step
            · reduce trust region (increase $\nu$)
    – until $\mathbf{x}$ is updated or maximum number of iterations is exceeded
```

In SciPy the Levenberg–Marquardt optimiser is in the `optimize` module, and it can be called using `scipy.optimize.leastsq()`. Some general details about using the SciPy optimisers are given in Section 9.3.2.

We will look at two examples of using non-linear least-squares. One is a simple case of finding the minimum of a function that consists of two quadratic terms added together, i.e., a sum-of-squares problem, while the second is to minimise the fitting of a function to data.

The function that we will attempt to minimise is Rosenbrock's function:

$$f(x_1, x_2) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2. \tag{9.18}$$

This is a common problem to try since it has a long narrow valley, so finding the optimal solution is not especially easy (except by hand: if you look at the problem, then guessing that $x_1 = 1, x_2 = 1$ is the minimum is fairly obvious). You need to work out how to encode this in the form required for a sum-of-squares problem, which is basically to write:

$$\mathbf{r} = (10(x_2 - x_1^2), 1 - x_1)^T. \tag{9.19}$$

The Jacobian is then:

$$\mathbf{J} = \begin{pmatrix} -20x_1 & 10 \\ -1 & 0 \end{pmatrix}. \tag{9.20}$$

In this notation, $f(x_1, x_2) = \mathbf{r}^T\mathbf{r}$ and the gradient is $\mathbf{J}^T\mathbf{r}$. All of which can be written as a simple Python function:

```
def function(p):
    r = np.array([10*(p[1]-p[0]**2),(1-p[0])])
    fp = np.dot(transpose(r),r)
    J = (np.array([[-20*p[0],10],[-1,0]]))
    grad = np.dot(J.T,r.T)
    return fp,r,grad,J
```

Running the algorithm with starting point $(-1.92, 2)$ leads to the following outputs, where the numbers printed on each line are the function value, the parameters that gave it, the gradient, and the value of ν.

```
f(x)     Params        Grad        nu
292.92   [ 0.66 -6.22] 672.00   0.001
4421.20  [ 0.99  0.87] 1099.51  0.0001
1.21     [ 1.00  1.00] 24.40    1e-05
8.67-07  [ 1.00  1.00] 0.02     1e-06
6.18-17  [ 1.00  1.00] 1.57-07  1e-07
```

The second example is fitting a function to data. The function is a moderately complicated beast that is definitely not amenable to linear least-squares fitting:

$$y = f(p_1, p_2) = p_1 \cos(p_2 x) + p_2 \sin(p_1 x), \tag{9.21}$$

where the p_i are the parameters to be fitted and x is a datapoint from a set that are

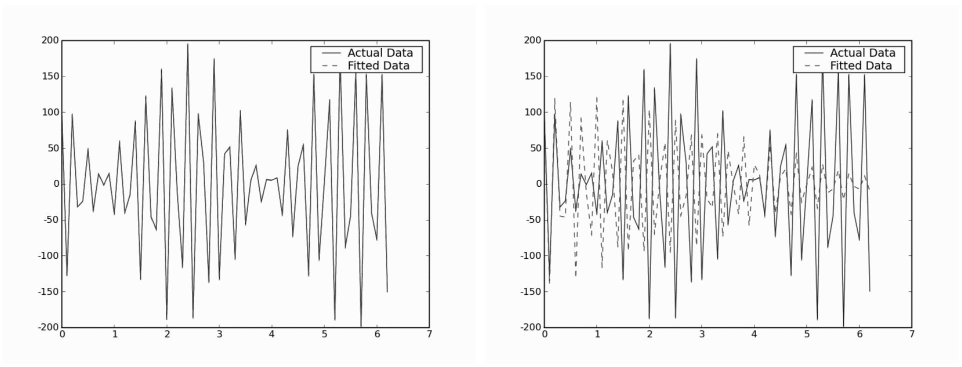

FIGURE 9.4 Using Levenberg–Marquardt for least-squares data fitting of data from Equation (9.21). The example on the left converges to the correct solution, while the one on the right, which still starts from a point close to the correct solution, fails to find it, resulting in significantly different output.

used to construct the function to be fitted. This is a difficult function to fit because it has lots of minima (since sin and cos are periodic, with period 2π). For data fitting problems, the assumption is often that data are generated at regular x points by a noisy process that produces the y values. Then the sum-of-squares error that we wish to minimise is the difference between the data (y) and the current fit (parameter estimates $\hat{p}_1, \hat{p}_2$):

$$\mathbf{r} = y - \hat{p}_1 \cos(\hat{p}_2 x) + \hat{p}_2 \sin(\hat{p}_1 x). \tag{9.22}$$

The Jacobian for this function requires some careful differentiating, and then the whole problem can be left to the optimiser. Figure 9.4 shows two examples of trying to recover values $p_1 = 100, p_2 = 102$. On the left, the starting point is $(100.5, 102.5)$, while on the right it is $(101, 101)$. It can be seen that on this problem, Levenberg–Marquardt is very susceptible to local minima, since while the example on the left works (converging after only 8 iterations), the example on the right, which still starts with parameter values very close to the correct ones, gets stuck and fails, with final parameter values $(100.89, 101.13)$.

9.3 CONJUGATE GRADIENTS

Not every problem that we want to solve is a least-squares problem. The good news is that we can do rather better than steepest descent even when we want to minimise an arbitrary objective function. The key to this is to look again at Figure 9.3, where you can see that there are several of the steepest gradient lines that are in pretty much the same direction. We would only need to go in that direction once if we knew how far to go the first time. And then we would go in a direction orthogonal (at right angles) to that one and, in two dimensions, we would be finished, as is shown on the right of Figure 9.5, where one step in the x direction and one in the y direction are enough to complete the minimisation. In n dimensions we would have to take n steps, and then we would have finished. This amazing scenario is the aim of the method of conjugate gradients. It manages to achieve it in the linear case, but in most non-linear cases, which are the kind we are usually interested in, it usually requires a few more iterations than it theoretically should, although still many less steps than most other methods for real problems.

It turns out that making the lines be orthogonal is generally impossible, since you don't

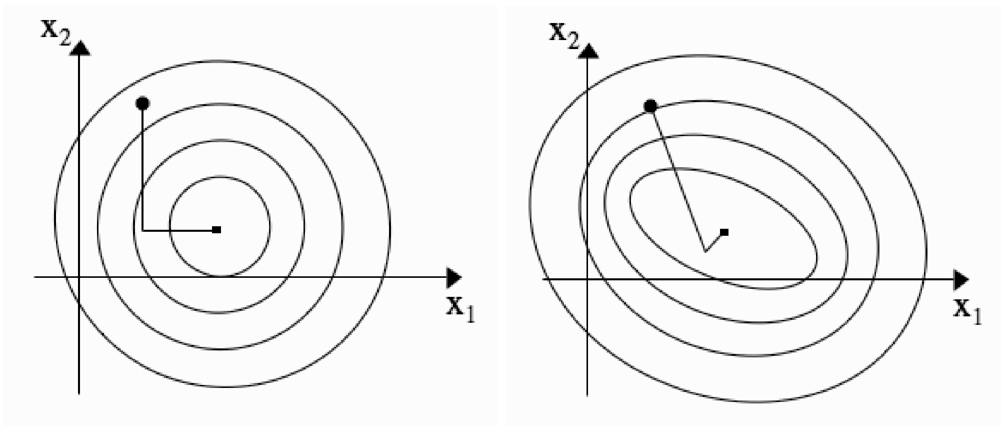

FIGURE 9.5 *Left:* If the directions are orthogonal to each other and the stepsize is correct, then only one step is needed for each dimension in the data, here two. *Right:* The conjugate directions are not orthogonal to each other on the ellipse.

have enough information about the solution space. However, it is possible to make them conjugate or A-orthogonal. Two vectors $\mathbf{p}_i, \mathbf{p}_j$ are conjugate if $\mathbf{p}_i^T \mathbf{A} \mathbf{p}_j = 0$ for some matrix $\mathbf{A}$. Conjugate lines for the ellipse contours in Figure 9.2 are shown on the right of Figure 9.5. Amazingly, the line search that we wrote down in Equation (9.1) is soluble along these directions, since they do not interfere with each other, with solution:

$$\alpha_i = \frac{\mathbf{p}_i^T(-\nabla f(\mathbf{x}_{i-1}))}{\mathbf{p}_i^T \mathbf{A} \mathbf{p}_i}. \tag{9.23}$$

We then need to use a function to find the zeros of this. The Newton–Raphson iteration, which is one method that will do it, is described below. So if we can find conjugate directions, then the line search is much better. The only question that remains is how to find them. This requires a Gram–Schmidt process, which constructs each new direction by taking a candidate solution and then subtracting off any part that lies along any of the directions that have already been used. We start by picking a set of mutually orthogonal vectors $\mathbf{u}_i$ (the basic coordinate axes will do; there are better options, but they are beyond the scope of this book) and then using:

$$\mathbf{p}_k = \mathbf{u}_k + \sum_{i=0}^{k-1} \beta_{ki} \mathbf{p}_i. \tag{9.24}$$

There are two possible β terms that can be used. They are both based on the ratios between the squared Jacobian before and after an update. The Fletcher–Reeves formula is:

$$\beta_{i+1} = \frac{\nabla f(\mathbf{x}_{x+1})^T \nabla f(\mathbf{x}_{i+1})}{\nabla f(\mathbf{x}_i)^T \nabla f(\mathbf{x}_i)}, \tag{9.25}$$

while the Polak–Ribiere formula is:

$$\beta_{i+1} = \frac{\nabla f(\mathbf{x}_{i+1})^T \left(\nabla f(\mathbf{x}_{i+1}) - \nabla f(\mathbf{x})_i\right)}{\nabla f(\mathbf{x}_i)^T \nabla f(\mathbf{x}_i)}. \tag{9.26}$$

The second one is often faster, but sometimes fails to converge (reach a stopping point).

We can put these things together to form a complete algorithm. It starts by computing an initial search direction $\mathbf{p}_0$ (steepest descent will do), then finding the α_i that minimises the function $f(\mathbf{x}_i + \alpha_i \mathbf{p}_i)$, and using it to set $\mathbf{x}_{i+1} = \mathbf{x}_i + \alpha_i \mathbf{p}_i$. The next direction is then $\mathbf{p}_{i+1} = -\nabla f(\mathbf{x}_{i+1}) + \beta_{i+1}\mathbf{p}_i$ where β is set by one of the two formulas above.

It is common to restart the algorithm every n iterations (where n is the number of dimensions in the problem) because the algorithm has now generated the whole set of conjugate directions. The algorithm will then cycle through the directions again making incremental improvements.

The only thing that we don't know how to do yet is to find the α_is. The usual method of doing that is the Newton–Raphson iteration, which is a method of finding the zero points of a polynomial. It works by computing the Taylor expansion of the function $f(\mathbf{x} + \alpha\mathbf{p})$, which is:

$$f(\mathbf{x}+\alpha\mathbf{p}) \approx f(\mathbf{x})+\alpha\mathbf{p}\left(\frac{d}{d\alpha}f(\mathbf{x}+\alpha\mathbf{p})\right)\bigg|_{\alpha=0}+\frac{\alpha^2}{2}\mathbf{p}\cdot\mathbf{p}\left(\frac{d^2}{d\alpha^2}f(\mathbf{x}+\alpha\mathbf{p})\right)\bigg|_{\alpha=0}+\ldots, \quad (9.27)$$

and differentiating it with respect to α, which requires the Jacobian and Hessian matrices (here, these matrices are derivatives of $f(\cdot)$, not $\mathbf{r}$ as they were in Section 9.2):

$$\frac{d}{d\alpha}f(\mathbf{x}+\alpha\mathbf{p}) \approx \mathbf{J}(\mathbf{x})\mathbf{p} + \alpha\mathbf{p}^T\mathbf{H}(\mathbf{x})\mathbf{p}. \quad (9.28)$$

Setting this equal to zero tells us that the minimiser of $f(\mathbf{x} + \alpha\mathbf{p})$ is:

$$\alpha = \frac{\mathbf{J}(\mathbf{x})^T\mathbf{p}}{\mathbf{p}^T\mathbf{H}(\mathbf{x})\mathbf{p}}. \quad (9.29)$$

Unless $f(\mathbf{x})$ is an especially nice function, the second derivative approximation that we have made here won't get us to the bottom in one step, so we will have to iterate this step a few times to find the zero point, which is why it is known as the Newton–Raphson *iteration*, i.e., you have to put it into a loop that runs until the iterate stops changing.

Putting all of those things together gives the entire algorithm, which we'll look at before we work on an example:

The Conjugate Gradients Algorithm

```
• Given start point $\mathbf{x}_0$, and stopping parameter $\epsilon$, set $\mathbf{p}_0 = -\nabla f(\mathbf{x})$
• Set $\mathbf{p} = \mathbf{p}_0$
• While $\mathbf{p} > \epsilon^2\mathbf{p}_0$:
    – compute $\alpha_k$ and $\mathbf{x}_{\text{new}} = \mathbf{x} + \alpha_k\mathbf{p}$ using the Newton–Raphson iteration:
        ∗ while $\alpha^2 dp > \epsilon^2$:
            · $\alpha = -(\nabla f(\mathbf{x})^T\mathbf{p})/(\mathbf{p}^T\mathbf{H}(\mathbf{x})\mathbf{p})$
            · $\mathbf{x} = \mathbf{x} + \alpha\mathbf{p}$
            · $dp = \mathbf{p}^T\mathbf{p}$
    – evaluate $\nabla f(\mathbf{x}_{\text{new}})$
    – compute $\beta_{k+1}$ using Equation (9.25) or (9.26)
    – update $\mathbf{p} \leftarrow \nabla f(\mathbf{x}_{\text{new}}) + \beta_{k+1}\mathbf{p}$
    – check for restarts
```

9.3.1 Conjugate Gradients Example

Computing the conjugate gradients solution to the function $f(\mathbf{x}) = (0.5x_1^2 + 0.2x_2^2 + 0.6x_3^2)$ makes use of the Jacobian and Hessian again. The first Newton–Raphson step yields an α value of 0.931, so that the next step is:

$$\mathbf{x}(1) = \begin{pmatrix} -2 \\ 2 \\ 0 \end{pmatrix} + 0.931 \times \begin{pmatrix} 2 \\ -0.8 \\ 2.4 \end{pmatrix} = \begin{pmatrix} -0.138 \\ 1.255 \\ 0.235 \end{pmatrix} \tag{9.30}$$

Then $\beta = 0.0337$, so that the direction is:

$$\mathbf{p}(1) = \begin{pmatrix} 0.138 \\ -0.502 \\ -0.282 \end{pmatrix} + 0.0337 \times \begin{pmatrix} 2 \\ -0.8 \\ 2.4 \end{pmatrix} = \begin{pmatrix} 0.205 \\ -0.529 \\ -0.201 \end{pmatrix} \tag{9.31}$$

In the second step, $\alpha = 1.731$,

$$\mathbf{x}(2) = \begin{pmatrix} -0.138 \\ 1.255 \\ 0.235 \end{pmatrix} + 1.731 \times \begin{pmatrix} 0.205 \\ -0.529 \\ -0.201 \end{pmatrix} = \begin{pmatrix} -0.217 \\ -0.136 \\ 0.136 \end{pmatrix} \tag{9.32}$$

and the update is:

$$p(2) = \begin{pmatrix} -0.217 \\ -0.136 \\ 0.136 \end{pmatrix} + 0.240 \times \begin{pmatrix} 0.205 \\ -0.529 \\ -0.201 \end{pmatrix} = \begin{pmatrix} -0.168 \\ -0.263 \\ 0.088 \end{pmatrix} \tag{9.33}$$

A third step then gives the final answer as $(0, 0, 0)$.

9.3.2 Conjugate Gradients and the MLP

The scientific Python libraries SciPy include a set of general purpose optimisation algorithms in `scipy.optimize`, including an interface function (`scipy.optimize.minimize()`) that can call the others. In this section we will investigate using the methods that are provided within that library, particularly the conjugate gradient optimiser, in order to find the weights of the Multi-layer Perceptron (MLP) that was the main algorithm of Chapter 4. In that chapter we derived an algorithm based on gradient descent of the back-propagated error from first principles, but here we can use general methods.

In order to use any gradient descent algorithm we need to work out a function to minimise, an initial guess for where to start searching, and (preferably) the gradient of that function with respect to the variables. The reason for saying 'preferably the gradient' is that many of the algorithms will create a numerical estimate of the gradient if an explicit version is not given. However, since the gradient is fairly easy to compute for the MLP, numerical estimation is not necessary. We used the sum-of-squares error for the MLP, so we just need to work out the derivatives of that function for the three different activation functions that we allow: the normal logistic function, the linear activation that was used for regression problems, and the soft-max activation, and we've already done that in Section 4.6.5.

As was mentioned above, there is an interface function for most of the SciPy optimisers, which has the following form:

```
scipy.optimize.minimize(fun,  x0,  args=(),  method='BFGS',  jac=None,
hess=None,⤸
hessp=None, bounds=None, constraints=(), tol=None, callback=None, ⤸
options=None)}.
```

The choice of method, which is the actual gradient descent algorithm used can include `'BFGS'` (which is the Broyden, Fletcher, Goldfarb, and Shanno algorithm, a variation on Newton's method from Section 9.1.1 that computes an approximation to the Hessian rather than requiring the programmer to supply it) and `CG` which is the conjugate gradient algorithm.

Looking at the code snippet again we see that we need to pass in an error function and the function to compute the derivatives. Both of these functions take arguments, specifically the inputs to the network, and the targets that those inputs are meant to produce. There is one issue that we have to deal with here, which is that the SciPy optimisers find the minimum value for a vector of parameters, and we currently have two separate weights matrices. We need to reshape these two matrices into vectors and then concatenate them before they can be used, using:

```
w = np.concatenate((self.weights1.flatten(),self.weights2.flatten()))
```

and something similar for the gradients. When the optimiser has run we will need to separate them and put the values back into the weight matrices using:

```
split = (self.nin+1)*self.nhidden
self.weights1 = np.reshape(wopt[:split],(self.nin+1,self.nhidden))
self.weights2 = np.reshape(wopt[split:],(self.nhidden+1,self.nout))
```

The optimiser also needs an initial guess `x0` for the weights, but this is not an issue since in the original algorithm they are already set to have small positive and negative values, so we can just use those values.

In fact, there is a numerical detail that we need to deal with as well; technically it could be a problem with the version of the MLP that we implemented in Chapter 4 as well, but it doesn't usually seem to be an issue there. The problem is that when we use the sigmoid function and take the exponential we can get overflow in the floating point number, either from it becoming too large, or too close to 0. This is a particular problem when we use the cross-entropy error function of Section 4.6.6, because we then take the logarithm, and we need to make sure that the input is in the range of the log function. NumPy provides some useful constants to make these checks, and they can be seen in use in the following code snippet, which replaces the error calculation in the original MLP:

```
# Different types of output neurons
if self.outtype == 'linear':
    error = 0.5*np.sum((outputs-targets)**2)
elif self.outtype == 'logistic':
    # Non-zero checks
    maxval = -np.log(np.finfo(np.float64).eps)
    minval = -np.log(1./np.finfo(np.float64).tiny - 1.)
    outputs = np.where(outputs<maxval,outputs,maxval)
    outputs = np.where(outputs>minval,outputs,minval)
    outputs = 1./(1. + np.exp(-outputs))
    error = - np.sum(targets*np.log(outputs) + (1 - targets)*np.log(1 - ⤸
    outputs))
elif self.outtype == 'softmax':
    nout = np.shape(outputs)[1]
    maxval = np.log(np.finfo(np.float64).max) - np.log(nout)
    minval = np.log(np.finfo(np.float32).tiny)
    outputs = np.where(outputs<maxval,outputs,maxval)
    outputs = np.where(outputs>minval,outputs,minval)
    normalisers = np.sum(np.exp(outputs),axis=1)*np.ones((1,np.shape(outputs)⤸
    [0]))
    y =  np.transpose(np.transpose(np.exp(outputs))/normalisers)
    y[y<np.finfo(np.float64).tiny] = np.finfo(np.float32).tiny
    error = - np.sum(targets*np.log(y));
```

Finally, we need to decide how accurate we want the result to be, and how many iterations we are going to allow the algorithm to run for before calling a halt to the optimisation. When the algorithm has reached a minimum the gradient function will be 0, and so the normal convergence criterium is that the gradient is close to zero. The default value for this parameter is 1×10^{-5} and we will leave this unchanged. We will also specify that the algorithm can run for no more than 10,000 steps. Together, these lead to the following function call to the conjugate gradient optimiser (here the code uses an explicit call to the conjugate gradient method rather than the interface, but there is no real difference):

```
out =        so.fmin_cg(self.mlperror,        w,        fprime=self.mlpgrad,
args=(inputs,targets)⤸
, maxiter=10000, full_output=True, disp=1)
```

The `full_output` and `disp` parameters tell the optimiser to give a report on whether or not it was successful and how much work it did, something like:

```
Warning: Maximum number of iterations has been exceeded.
         Current function value: 7.487182
         Iterations: 10000
         Function evaluations: 250695
```

```
        Gradient evaluations: 140930
```

Now all that remains is to extract the new weight values from the values that the optimiser returns, which are in `out[0]`, and we are ready to use the algorithm. The demonstrations that we used in Chapter 4 are all perfectly suitable, of course, and all that needs changing is to import the conjugate gradient version of the MLP instead of the earlier version.

There are other methods of doing gradient descent, some of which are more effective on certain problems (but note that the No Free Lunch theorem tells us that no one solver will be the most effective for every problem). For example, the convex optimisation that was used for the Support Vector Machine in Chapter 8 is a gradient descent method for a particular type of constrained problem. We will next consider what happens when the problems that we wish to solve are discrete, which means that there is no gradient to find.

9.4 SEARCH: THREE BASIC APPROACHES

We are going to discuss three different ways to attempt optimisation without gradients. For each one, we will see how it works on the Travelling Salesman Problem (TSP), which is a classic discrete optimisation problem that consists of trying to find the shortest route through a set of cities that visits each city exactly once and returns to the start. For the first (starting) city we can choose any of the N that are available. For the next, there are $N-1$ choices, and for the next $N-2$. Using a brute force search in this way provides a $\mathcal{O}(N!)$ solution, which is obviously infeasible.

In fact, the TSP is an NP-hard problem. The best-known solution that is guaranteed to find the global maximum is using dynamic programming and its computational cost is $\mathcal{O}(n^2 2^n)$, but we won't be considering that here—the TSP is an example, not a problem we really want to solve here. The basic search methods are described next.

9.4.1 Exhaustive Search

Try out every solution and pick the best one. While this is obviously guaranteed to find the global optimum, because it checks every single solution, it is impractical for any reasonable size problem. For the TSP it would involve testing out every single possible way of ordering the cities, and calculating the distance for each ordering, so the computational complexity is $\mathcal{O}(N!)$, which is worse than exponential.

It is computationally infeasible to do the computations for more than about $N = 10$ cities. The basic part of the algorithm uses a helper function `permutation()` that computes possible orderings of the cities, but is otherwise fairly obvious:

```
for newOrder in permutation(range(nCities)):
    possibleDistanceTravelled = 0
    for i in range(nCities-1):
        possibleDistanceTravelled += distances[newOrder[i],newOrder[i+1]]
    possibleDistanceTravelled += distances[newOrder[nCities-1],0]

    if possibleDistanceTravelled < distanceTravelled:
        distanceTravelled = possibleDistanceTravelled
        cityOrder = newOrder
```

9.4.2 Greedy Search

Just make one pass through the system, making the best local choice at each stage. So for the TSP, choose the first city arbitrarily, and then repeatedly pick the city that is closest to where you are now that hasn't been visited yet, until you run out of cities. This is computationally very cheap ($\mathcal{O}(N \log N)$), but it is certainly not guaranteed to find the optimal solution, or even a particularly good one. The code is very simple, though:

```
for i in range(nCities-1):
    cityOrder[i+1] = np.argmin(dist[cityOrder[i],:])
    distanceTravelled  += dist[cityOrder[i],cityOrder[i+1]]
    # Now exclude the chance of travelling to that city again
    dist[:,cityOrder[i+1]] = np.Inf

# Now return to the original city
distanceTravelled += distances[cityOrder[nCities-1],0]
```

9.4.3 Hill Climbing

The basic idea of the hill climbing algorithm is to perform local search around the current solution, choosing any option that improves the result. (It might seem odd to talk about hill *climbing* when we've always talked about minimising a function. Of course, the difference between maximisation and minimisation is just whether you put a minus sign in front of the equation or not, and 'hill climbing' sounds much better than 'hollow descending.') The choice of how to do local search is called the move-set. It describes how the current solution can be changed to generate new solutions. So if we were to imagine moving about in 2D Euclidean space, possible moves might be to move 1 step north, south, east, or west.

For the TSP, the hill climbing solution would consist of choosing an initial solution randomly, and then swapping pairs of cities in the tour and seeing if the total length of the tour decreases. The algorithm would stop after some pre-defined number of swaps had occurred, or when no swap improved the result for some pre-defined length of time. As with the greedy search, there is no way to predict how good the solution will be: there is a chance that it will find the global maximum, but no guarantee of it; it could get stuck in the first local maxima. The central loop of the hill climbing algorithm just picks a pair of cities to swap, and keeps the change if it makes the total distance shorter:

```
for i in range(1000):
        # Choose cities to swap
        city1 = np.random.randint(nCities)
        city2 = np.random.randint(nCities)
```

```
    if city1 != city2:
        # Reorder the set of cities
        possibleCityOrder = cityOrder.copy()
        possibleCityOrder = np.where(possibleCityOrder==city1,-1,
        possibleCityOrder)
        possibleCityOrder = np.where(possibleCityOrder==city2,city1,
        possibleCityOrder)
        possibleCityOrder = np.where(possibleCityOrder==-1,city2,
        possibleCityOrder)

        # Work out the new distances
        # This can be done more efficiently
        newDistanceTravelled = 0
        for j in range(nCities-1):
            newDistanceTravelled += distances[possibleCityOrder[j],
            possibleCityOrder[j+1]]
        distanceTravelled += distances[cityOrder[nCities-1],0]

        if newDistanceTravelled < distanceTravelled:
            distanceTravelled = newDistanceTravelled
            cityOrder = possibleCityOrder
```

Hill climbing has three particular types of functions that it does badly on. They can all be imagined using the analogy of real hill climbing.

The first is when there are lots of foothills around the optimal solution. In that case the algorithm climbs the local maximum, and may get stuck there; certainly it will take a very long time to reach the optimal solution. The second is on a plateau, where no changes that the algorithm makes affect the solution. In this case the solution will just change randomly, if at all, and the maximum will probably not be found. The third case is when there is a very gently sloping ridge in the data. Most directions that the algorithm looks in will be downhill, and so it may decide that it has already reached the maximum.

9.5 EXPLOITATION AND EXPLORATION

The search methods above can be separated into methods that perform exploration of the search space, always trying out new solutions, like exhaustive search, and those performing exploitation of the current best solution, by trying out local variations of that current best solution, like hill climbing. Ideally, we would like some combination of the two—we should be trying to improve on the current best solution by local search, and also looking around in case there is an even better solution hiding elsewhere in the search space.

One way to think about this is known as the n-armed bandit problem. Suppose that we have a room full of one-armed bandit machines in some tacky Las Vegas casino (for those who don't know, a one-armed bandit is a slot machine with a lever that you pull, as in Figure 9.6). You don't know anything about the machines in advance, such as what the payouts are, and how likely you are to get the payout. You enter the room with a fistful of 50 cent coins from your student loan, aiming to generate enough beer money to get through the year. How do you choose which machine to use?

FIGURE 9.6 A one-armed bandit machine. It has one arm, and it steals your money.

At first, you have no information at all, so you choose randomly. However, as you explore, you pick up information about which machines are good (here, good means that you get a payout more often). You could carry on using them (exploiting your knowledge) or you could try out other machines in the hope of finding one that pays out even more (exploring further). The optimal approach is to trade off the two, always making sure that you have enough money to explore further by exploiting the best machines you know of, but exploring when you can.

One place where this combination of exploration and exploitation can be clearly seen is in evolution. We'll talk about that in the next chapter, but here we will look to physics instead of biology to act as our inspiration.

9.6 SIMULATED ANNEALING

In the field of statistical mechanics physicists have to deal with systems that are very large (tens of thousands of molecules and more) so that, while the computations are possible in principle, in practice the computational time is far too large. They have developed stochastic methods (that is, based on randomness) in order to get approximate solutions to the problems that, while still expensive, do not require the massive computational times that the full solution would.

The method that we will look at is based on the way in which real-world physical systems can be brought into very low energy states, which are therefore very stable. The system is heated, so that there is plenty of energy around, and each part of the system is effectively random. An annealing schedule is applied that cools the material down, allowing it to relax into a low energy configuration. We are going to model the same idea.

We start with an arbitrary temperature T, which is high. We will then randomly choose states and change their values, monitoring the energy of the system before and after. If the energy is lower afterwards, then the system will prefer that solution, so we accept the change. So far, this is similar to gradient descent. However, if the energy is not lower, then we still consider whether or not to accept the solution. We do this by evaluating $E_{\text{before}} - E_{\text{after}}$ and accepting the new solution if the value of $\exp((E_{\text{before}} - E_{\text{after}})/T)$ is bigger than a uniform random value between 0 and 1 (note that the expression is between 0 and 1 since it is the exponential of a negative value). This is called the Boltzmann distribution. The rationale behind sometimes accepting poorer states is that we might have found a local minimum, and by allowing this more expensive energy state we can escape from it.

After doing this a few times, the annealing schedule is applied in order to reduce the temperature and the method continues until the temperature reaches 0. As the temperature gets lower, so does the chance of accepting any particular higher energy state. The most common annealing schedule is $T(t+1) = cT(t)$, where $0 < c < 1$ (more commonly, $0.8 < c < 1$). The annealing needs to be slow to allow for lots of search to happen. For the TSP the best way to include simulated annealing is to modify the hill climbing algorithm above, changing the acceptance criteria for a change in the city ordering to:

```
if newDistanceTravelled < distanceTravelled or (distanceTravelled - ⤸
newDistanceTravelled) < T*np.log(np.random.rand()):
   distanceTravelled = newDistanceTravelled
   cityOrder = possibleCityOrder

# Annealing schedule
T = c*T
```

9.6.1 Comparison

Running all four methods above on the TSP for five cities gave the following results, where the best solution found and the distance are given in the first line and the time it took to run (in seconds) on the second:

```
>>> TSP.runAll()
Exhaustive search
((3, 1, 2, 4, 0), 2.65)
0.0036
Greedy search
((0, 2, 1, 3, 4), 3.27)
0.0013
Hill Climbing
((4, 3, 1, 2, 0]), 2.66)
0.1788
Simulated Annealing
((3, 1, 2, 4, 0]), 2.65)
0.0052
```

With ten cities the results were quite different, showing how important good approximations to search are, since even for this fairly small problem the exhaustive search takes a very long time. Note that the greedy search does nearly as well in this case, but this is simply chance.

```
Exhaustive search
((1, 5, 10, 6, 3, 9, 2, 4, 8, 7, 0), 4.18)
```

```
1781.0613
Greedy search
((3, 9, 2, 6, 10, 5, 1, 8, 4, 7, 0]), 4.49)
0.0057
Hill Climbing
((7, 9, 6, 2, 4, 0, 3, 8, 1, 5, 10]), 7.00)
0.4572
Simulated Annealing
((10, 1, 6, 9, 8, 0, 5, 2, 4, 7, 3]), 8.95)
0.0065
```

FURTHER READING

Two books on numerical optimization that provide much more information are:

- J. Nocedal and S.J. Wright. *Numerical Optimization.* Springer, Berlin, Germany, 1999.
- C.T. Kelley. *Iterative Methods for Optimization.* Number 18 in Frontiers in Applied Mathematics. SIAM, Philadelphia, USA, 1999.

A possible reference for the second half of the chapter is:

- J.C. Spall. *Introduction to Stochastic Search and Optimization: Estimation, Simulation, and Control.* Wiley-Interscience, New York, USA, 2003.

Some of the material is covered in:

- Section 6.9 and Sections 7.1–7.2 of R.O. Duda, P.E. Hart, and D.G. Stork. *Pattern Classification*, 2nd edition, Wiley-Interscience, New York, USA, 2001.

PRACTICE QUESTIONS

Problem 9.1 In the discussion after Equation (9.10) it is stated that the direct solution is unstable. Experiment with this and see that it is true.

Problem 9.2 Modify the code in `CG.py` in order to take a general function, together with its Jacobian (and if available its Hessian) and then compute the minimum.

Problem 9.3 Experiment with the Fletcher–Reeves and Polak–Ribiere formulas (Equations (9.25) and (9.26)) when solving Rosenbrock's function using conjugate gradients. Can you find places where one works better than the other?

Problem 9.4 Generate data from the equation $a(1-\exp(-b(x-c)))$ for choice of parameters a, b, c and x in the range -5 to 5 (with noise). Use Levenberg–Marquardt to fit the parameters.

Problem 9.5 Modify the conjugate gradient version of the MLP to use the other optimisation algorithms provided by SciPy and compare the results. Also, try stopping the optimiser from using the exact computation of the gradient and instead making a numerical estimate of it, and see how that changes the results.

Problem 9.6 By incorporating back-tracking into hill climbing, it is possible to escape from some poor local maxima. Add this into the code and test the results on the Travelling Salesman problem.

Problem 9.7 The logical satisfiability problem is an NP-complete problem that consists of finding truth assignments to sets of logical statements (e.g., $(a_1 \wedge a_2) \vee (\neg a_1 \vee a_3))$ so that they are true. It is an NP-complete problem to find truth assignments. Devise a way to use hill climbing and simulated annealing on the problem.

CHAPTER 10

Evolutionary Learning

In this chapter we are going to start by treating evolution the same way that we treated neuroscience earlier in the book—by cherry-picking a few useful concepts, and then filling in the gaps with computer science in order to make an effective learning method. To see why this might be interesting, you need to view evolution as a search problem. We don't generally think of it in this way, but animals are competing with each other in all kinds of ways—for example, eating each other—which encourages them to try to find camouflage colours, become toxic to certain predators, etc.

Evolution works on a population through an imaginary fitness landscape, which has an implicit bias towards animals that are 'fitter', i.e., those animals that live long enough to reproduce, are more attractive, and so get more mates, and generate more and healthier offspring. You can find out more from hundreds of books, such as Charles Darwin's *The Origin of Species* (the original book on the topic, still in print and very interesting) and Richard Dawkin's *The Blind Watchmaker*.

The genetic algorithm models the genetic process that gives rise to evolution. In particular, it models sexual reproduction, where both parents give some genetic information to their offspring. As is sketched in Figure 10.1, in biological organisms, each parent passes on one chromosome out of their two, and so there is a 50% chance of any gene making it into the offspring. Of the two versions of each gene (one from each parent) one allele (variation) is selected. Hence, children have similarities with their parents, and there is lots of genetic inheritance. However, there are also random mutations, caused by copying errors when the chromosome material is reproduced, which means that some things do change over time. Real genetics is obviously a lot more complicated than this, but we are taking only the things that we want for our model.

The genetic algorithm shows many of the things that are best and worst about machine learning: it is often, but not always, very effective, it has an array of parameters that are crucial, but hard to set, and it is impossible to guarantee that it will find a result that is any good at all. Having said all that, it often works very well, and it has become a very popular algorithm for people to use when they have no idea of any other way to find a reasonable solution.

In the terms that we saw at the end of the previous chapter, genetic algorithms perform both exploitation and exploration, so that they can make incremental improvements to current good solutions, but also find radically new solutions, some of which may be better than the current best.

We will also look briefly at two other topics in this chapter, a variation of the genetic algorithm that acts on trees that represent computer programs that is known as Genetic

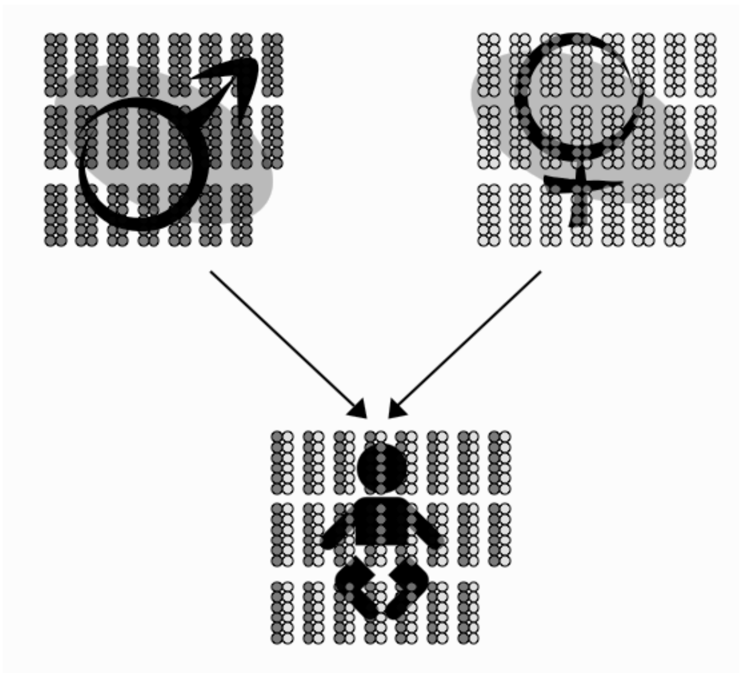

FIGURE 10.1 Each adult in the mating pair passes one of their two chromosomes to their offspring.

Programming, and a set of algorithms that use sampling from a probability distribution rather than an evolving population in order to find better solutions.

10.1 THE GENETIC ALGORITHM (GA)

The Genetic Algorithm is a computational approximation to how evolution performs search, which is by producing modifications of the parent genomes in their offspring and thus producing new individuals with different fitness. Like another mathematical model that we saw earlier in the book—the neuron—it attempts to abstract away everything except the important parts that we need to understand what evolution does. From this principle, the things that we need to model simple genetics inside a computer and solve problems with it are:

- a method for representing problems as chromosomes
- a way to calculate the fitness of a solution
- a selection method to choose parents
- a way to generate offspring by breeding the parents

These items are all described in the following sections, and the basic algorithm is described. We are going to use an example to describe the methods, which is an NP-complete problem (if you are not familiar with the term NP-complete, its practical implication is that the problem runs in exponential time or worse in the number of inputs) known as the knapsack problem (a knapsack is a rather old name for a rucksack or bag). Sections 10.3.1 and 10.3.4 provide other examples. The knapsack problem is easy to describe, but difficult to solve in general. Here is the version of it that we will use:

Suppose that you are packing for your holidays. You've bought the biggest and best rucksack that was for sale, but there is still no way that you are going to fit in everything you want to take (camera, money, addresses of friends, etc.) and the things that your mum is insisting you take (spare underwear, phrasebook, stamps to write home with, etc.). As a good computer scientist you decide to measure how much space it takes up and then write a program to work out how to fill as much of the bag as possible, so that you get the best value for your airfare.

This problem, and variations of it, appear in various disguises in cryptography, combinatorics, applied mathematics, logistics, and business, so it is an important problem. Unfortunately, since it is NP-complete, finding the optimal solution for interesting cases (pretty much anything above 10 items) is computationally impossible. There is an obvious greedy algorithm that finds solutions to the knapsack problem. At each stage it takes the largest thing that hasn't been packed yet and that will still fit into the bag, and iterates that rule. This will not necessarily return the optimal solution (unless each thing is larger than the sum of all the ones smaller than it, in which case it will), but it is very quick and simple. So a GA should be getting a much better solution than the greedy rule most of the time to be worth all the effort involved in writing and running it.

10.1.1 String Representation

The first thing that we need is some way to represent the individual solutions, in analogy to the chromosome. GAs use a string, with each element of the string (equivalent to the gene) being chosen from some alphabet. The different values in the alphabet, which is often just binary, are analogous to the alleles. For the problem we are trying to solve we have to work out a way of encoding the description of a solution as a string. We then create a set of random strings to be our initial population.

It is possible to modify the GA so that the alphabet it uses runs over the real numbers. While purists don't think that this is a GA at all, it is quite popular, because of the number of applications, but it is not as elegant as using a discrete alphabet. It also tends to make the mutation operator that we will see later less useful.

For the knapsack problem the alphabet is very simple, since we can make it binary, since for each item we just need to say whether or not we want to take it. We make the string L units long, where L is the total number of things we would like to take with us, and make each unit a binary digit. We then encode a solution using 0 for the things we will not take and 1 for the things we will. So if there were four things we wanted to take, then $(0, 1, 1, 0)$ would mean that we take the middle two, but not the first or last.

Note that this does not tell us whether or not this string is possible (that is, whether the things that we have said we will take will actually fit into the knapsack), nor whether it is a good string (whether it fills the knapsack). To work these out we need some way to decide how well each string fulfills the problem criteria. This is known as the fitness of the string.

10.1.2 Evaluating Fitness

The fitness function can be seen as an oracle that takes a string as an argument and returns a value for that string. Together with the string encoding the fitness function forms the problem-specific part of the GA. It is worth thinking about what we want from our fitness function. Clearly, the best string should have the highest fitness, and the fitness should decrease as the strings do less well on the problem. In real evolution, the fitness landscape is not static: there is competition between different species, such as predators and prey, or medical cures for certain diseases, and so the measure of fitness changes over time. We'll ignore that in the genetic algorithm.

For the knapsack problem, we decided that we wanted to make the bag as full as possible. So we would need to know the volume of each item that we want to put into the knapsack, and then for a given string that says which things should be taken, and which should not, we can compute the total volume. This is then a possible fitness function. However, it does not tell us anything about whether they will fit into the bag—with this fitness function the optimal solution is to take everything. So we need to check that they will fit, and if they will not, reduce the fitness of that solution. One option would be to set the fitness to 0 if the things in that string will not all fit. However, suppose that the solution is almost perfect, it is just that there is one thing too many in the knapsack. By setting the fitness to 0 we are reducing the chance of this solution being allowed to evolve and improve during later iterations. For this reason we will make the fitness function be the sum of the values of the items to be taken if they fit into the knapsack, but if they do not we will subtract twice the amount by which they are too big for the knapsack from the size of the knapsack. This allows solutions that are only just over to be considered for improvement, but tries to ensure that they are not the fittest solutions around.

10.1.3 Population

We can now measure the fitness of any string. The GA works on a population of strings, with the first generation usually being created randomly. The fitness of each string is then evaluated, and that first generation is bred together to make a second generation, which is then used to generate a third, and so on. After the initial population is chosen randomly, the algorithm evolves to produce each successive generation, with the hope being that there will be progressively fitter individuals in the populations as the number of generations increases.

To make the initial population for the knapsack problem, we will now create a set of random binary strings of length L by using the random number generator, which is very easy in NumPy using the uniform random number generator and the `np.where()` function:

```
pop = np.random.rand(popSize,stringLength)
pop = np.where(pop<0.5,0,1)
```

We now need to choose parents out of this population, and start breeding them.

10.1.4 Generating Offspring: Parent Selection

For the current generation we need to select those strings that will be used to generate new offspring. The idea here is that average fitness will improve if we select strings that are already relatively fit compared to the other members of the population (following natural selection), which is exploitation of our current population. However, it is also good to allow some exploration in there, which means that we have to allow some possibility of weak strings being considered. If strings are chosen proportionally to their fitness, so that fitter strings are more likely to be chosen to enter the 'mating pool', then this allows for both options. There are three commonly employed ways to do this, although the last one tends to produce better results:

Tournament Selection Repeatedly pick four strings from the population, with replacement and put the fittest two of them into the mating pool.

Truncation Selection Pick some fraction f of the best strings and ignore the rest. For example, $f = 0.5$ is often used, so the best 50% of the strings are put into the mating pool, each twice so that the pool is the right size. The pool is randomly shuffled to make the pairs. This is obviously very easy to implement, but it does limit the amount of exploration that is done, biasing the GA towards exploitation.

Fitness Proportional Selection The better option is to select strings probabilistically, with the probability of a string being selected being proportional to its fitness. The function that is generally used is (for string α):

$$p^{\alpha} = \frac{F^{\alpha}}{\sum_{\alpha'} F^{\alpha'}}, \tag{10.1}$$

where F^{α} is the fitness. If the fitness is not positive then F needs to be replaced by $\exp(sF)$ throughout, where s is the selection strength, a parameter, and you might recognise the equation as the soft-max activation from Chapter 4:

$$p^{\alpha} = \frac{\exp(sF^{\alpha})}{\sum_{\alpha'} \exp(sF^{\alpha'})}. \tag{10.2}$$

There is an implementation issue here. We want to pick each string with probability proportional to its fitness, but if we only have one copy of each string, then the probability of picking each string is the same. One way around this is to add more copies of the fitter strings, so that they are more likely to get chosen. This is sometimes called 'roulette selection', because if you imagine that each string gets an area on a roulette wheel, then the larger the area associated to one number, the more likely it is that the ball will land there. You can then just randomly pick strings from this larger set. A method of doing this is shown in the following code snippet, which uses the `np.kron()` function. We've seen this before (in Section 6.5); it is a NumPy function that multiplies each element of its first array argument by every element of the second, putting all of the results together into one multi-dimensional output array. It is useful here in order to populate the new and much larger `newPopulation` array, which contains multiple copies of each string.

```
# Put in repeated copies of each string according to fitness
# Deal with strings with very low fitness
j=0
while np.round(fitness[j])<1:
 j = j+1

newPop = np.kron(np.ones((np.round(fitness[j]),1)),pop[j,:])

# Add multiple copies of strings into the newPop
for i in range(j+1,self.popSize):
 if np.round(fitness[i])>=1:
  newPop = np.concatenate((newPop,np.kron(np.ones((np.round(fitness[i]),1)),
  pop[i,:])),axis=0)

# Shuffle the order (note that there are still too many)
indices = range(np.shape(newPop)[0])
np.random.shuffle(indices)
newPop = newPop[indices[:popSize],:]
return newPop
```

However we select the strings to put into the mating pool, the next operation is to put them into pairs. Since the order that they are in is random, we can simply pair up the strings so that each even-indexed string takes the following odd-indexed one as its mate.

10.2 GENERATING OFFSPRING: GENETIC OPERATORS

Having selected our breeding pairs, we now need to decide how to combine their two strings to generate the offspring, which is the genetics part of the algorithm. There are two genetic operators that are generally used, and they are discussed now. There are others, but these were the original choices, and are far and away the most common.

10.2.1 Crossover

In biology, organisms have two chromosomes, and each parent donates one of them. Members of our GA population only have one chromosome-equivalent, the string. Thus, we generate the new string as part of the first parent and part of the second. The most common way of doing this is to pick one point at random in the string, and to use parent 1 for the first part of the string, up to the crossover point and parent 2 for the rest. We actually generate two offspring, with the second one consisting of the first part of parent 2 and the second part of parent 1. This scheme is known as single point crossover, and the extension to multi-point crossover is hopefully obvious. The most 'extreme' version is known as uniform crossover and consists of independently selecting each element of the string at random from the two parents. The three types of crossover are shown in Figure 10.2.

Crossover is the operator that performs global exploration, since the strings that are produced are radically different to both parents in at least some places. The hope is that sometimes we will take good parts of both solutions and put them together to make an even better solution. A nice picture example is to imagine a bird that has webbed feet for good swimming, but that cannot fly, breeding with a bird that can fly, but not swim. The offspring? A duck! Obviously, this is not biologically plausible, but it is a good picture of how crossover works. One interesting feature of the GA that obviously isn't true in real genetics is that in addition to the duck the algorithm would produce the bird that can't fly or swim, although it is unlikely to last long since its fitness will presumably not be high. In fact, there are exceptions to this, such as the great New Zealand Kiwi, which can neither swim nor fly, but is happily not extinct.

The following code snippet shows a NumPy implementation of single point crossover. The extension to multi-point and uniform crossover is not particularly difficult.

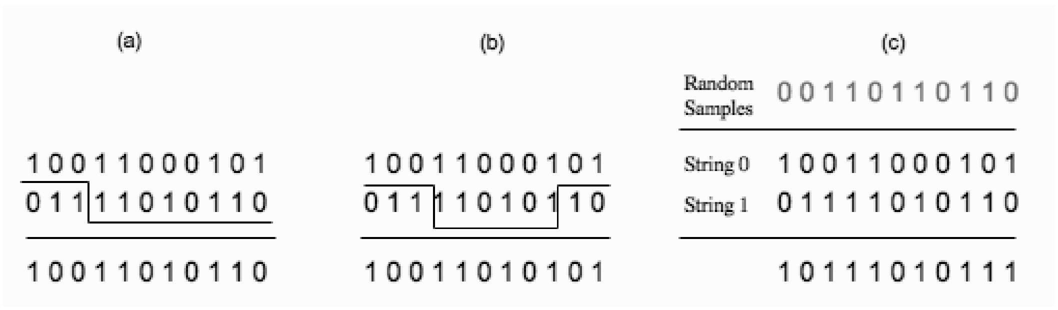

FIGURE 10.2 The different forms of the crossover operator. (a) Single point crossover. A position in the string is chosen at random, and the offspring is made up of the first part of parent 1 and the second part of parent 2. (b) Multi-point crossover. Multiple points are chosen, with the offspring being made in the same way. (c) Uniform crossover. Random numbers are used to select which parent to take each element from.

```
def spCrossover(pop):
    newPop = np.zeros(shape(pop))
    crossoverPoint = np.random.randint(0,stringLength,popSize)
    for i in range(0,self.popSize,2):
 newPop[i,:crossoverPoint[i]] = pop[i,:crossoverPoint[i]]
 newPop[i+1,:crossoverPoint[i]] = pop[i+1,:crossoverPoint[i]]
 newPop[i,crossoverPoint[i]:] = pop[i+1,crossoverPoint[i]:]
 newPop[i+1,crossoverPoint[i]:] = pop[i,crossoverPoint[i]:]
    return newPop
```

Crossover is not always useful, depending upon the problem; for example, in the Travelling Salesman Problem that we talked about in Chapter 9, the strings that are generated by crossover might not even be valid tours. However, when it is useful, it is often the more powerful of the genetic operators, and has led to the building block hypothesis of how GAs work. The idea is that GAs work well on problems where the solution comes from putting together lots of little solutions, so that different strings assemble each separate building block, and then crossover puts those substrings together to make the final solution.

10.2.2 Mutation

The other genetic operator is mutation, which effectively performs local random search. The value of any element of the string can be changed, governed by some (usually low) probability p. For our binary alphabet in the knapsack problem, mutation causes a bit-flip, as is shown in Figure 10.3. For chromosomes with real values, some random number is generally added or subtracted from the current value. Often, $p \approx 1/L$ where L is the string length, so that there is approximately one mutation in each string. This might seem quite high, but it is often found to be a good choice given that the mutation rate has to trade off doing lots of local search with the risk of disrupting the good solutions.

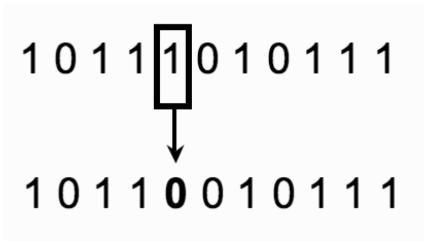

FIGURE 10.3 The effects of mutation on a string.

10.2.3 Elitism, Tournaments, and Niching

At this stage we have taken pairs of parents, and produced pairs of offspring. There is now a choice of what to do with them. The simplest option would be simply replace the parents by their children to make a completely new population, and carry on from there. However, this means that the maximum fitness in each generation can decrease, at least temporarily, and since in the end we are only interested in the 'best' solution, this seems a bit risky: we could potentially lose a really good string that we find early on in the search, and that we never see again.

There is a variety of ways to avoid this, of which the simplest is to use elitism, which takes some number of the fittest strings from one generation and puts them directly into the next population, replacing strings that are already there either at random, or by choosing the least fit to replace. Note that at every iteration the population stays the same size, something else that is unlike real evolution. Another solution is to implement a tournament, where the two parents and their two offsprings compete, with the two fittest out of the four being put into the new population.

The implementation of these functions continues along the same lines as the previous ones; the `np.argsort()` function returns the indices of the array that sorts them into order, but does not actually sort the array. It returns an array the same size as the one that is sorted, and we only want to extract the first few elite ones. When we do this we will be left with a matrix with a singleton dimension, which is why the `np.squeeze()` function is needed to reduce the array to the right size.

```
def elitism(oldPop,pop,fitness):
    best = np.argsort(fitness)
    best = np.squeeze(oldPop[best[-nElite:],:])
    indices = range(np.shape(pop)[0])
    np.random.shuffle(indices)
    pop = pop[indices,:]
    pop[0:nElite,:] = best
    return pop
```

While elitism and tournaments both ensure that good solutions aren't lost, they both have the problem that they can encourage premature convergence, where the algorithm settles down to a constant population that never changes even though it hasn't found an optimum. This happens because the GA favours fitter members of the population, which means that a solution that reaches a local maximum will generally be favoured, and this solution will be exploited. Tournaments and elitism encourage this, because they reduce the amount of diversity in the population by allowing the same individuals to remain over many generations. This means that the exploration aspect of the GA stops occurring. Exploration will be downplayed, making it hard to escape from the local maximum—most strings will have worse fitness, and will therefore be replaced in the population. Eventually, the majority of the strings in the population will be the same, but will represent a local maximum, not the global maximum. The randomness in the GA is a very large part of why it works, and schemes to reduce that randomness often harm the overall results.

One way to solve the problem of premature convergence is through niching (also known as using island populations), where the population is separated into several subpopulations, which all evolve independently for some period of time, so that they are likely to have converged to different local maxima, and a few members of one subpopulation are occasionally injected as 'immigrants' into another subpopulation. Another approach is known as fitness sharing, where the fitness of a particular string is averaged across the number of times that that string appears in the population. This biases the fitness function towards uncommon strings, but can also mean that very common good solutions are selected against.

There are other methods that have been developed to improve the convergence and final results of GAs, but they aren't useful for a basic understanding of how the basic algorithm works, so we'll ignore them. Anybody who wants to know more is directed to one of the books in the references at the end of the chapter.

The complete algorithm for the GA consists of simply putting together the pieces that we have looked at individually. Extending the basic algorithm to include some of the methods mentioned above, such as tournaments and niching, can improve the performance of the algorithm, but does not change the description much. The algorithm is often run for a fixed number of generations. It is a computationally very expensive algorithm, especially if the fitness function is non-trivial to evaluate. After seeing a complete description of the GA, we'll have a look at an example of how the algorithm works by considering the problem of graph colouring, and then look at how to use the GA to solve two sample problems.

The Basic Genetic Algorithm

- **Initialisation**
  - generate N random strings of length L with the chosen alphabet
- **Learning**
  - repeat:
    * create an (initially empty) new population
    * repeat:
      · select two strings from current population, preferably using fitness-proportional selection
      · recombine them in pairs to produce two new strings
      · mutate the offspring
      · either add the two offspring to the population, or use tournaments to put two strings from the four of parents and offspring into the population
    * until N strings for the new population are generated
    * optionally, use elitism to take the fittest strings from the parent generation and replace some others from the child generation

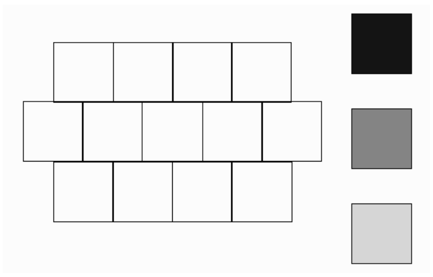

FIGURE 10.4 A sample map that we wish to colour using the three colours shown, without any two adjacent squares having the same colour.

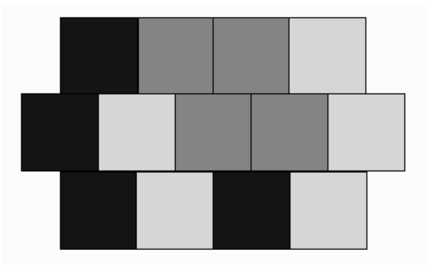

FIGURE 10.5 A possible colouring with several adjacent squares having the same colour.

```
        * keep track of the best string in the new population
        * replace the old population with the new one
    – until stopping criteria met
```

10.3 USING GENETIC ALGORITHMS

10.3.1 Map Colouring

Graph colouring is a typical discrete optimisation problem. We want to colour a graph using only k colours, and choose them in such a way that adjacent regions have different colours. It has been mathematically proven that any two-dimensional planar graph can be coloured with four colours, which was the first ever proof that used a computer program to check the cases. Even though it might be impossible, we are going to try to solve the three-colour problem using a genetic algorithm, we just won't be upset if the solution isn't perfect (this is a good idea with a GA anyway, of course). With all problems where you want to apply a genetic algorithm, there are three basic tasks that need to be performed:

Encode possible solutions as strings For this problem, we'll choose our alphabet to consist of the three possible shades (black (b), dark (d), and light (l), say). So for a six-region map, a possible string is $\alpha = \{bdblbb\}$. This says that the first region is black, the second dark grey, etc. We choose an order to record the regions in and stick to it for all the strings, and now we can encode any way of colouring in those six regions. An example problem and a colouring are given in Figures 10.4 and 10.5.

Choose a suitable fitness function The thing that we want to minimise (a cost function) is the number of times that two adjacent regions have the same colour. We could count these up fairly simply, but it is not a fitness function, because the best solution has the lowest number, not the highest. One easy way to turn it into a fitness function would be to multiply all the scores by minus one and use Equation (10.2) to turn them into fitnesses, or to count the total number of lines between regions and subtract off the number where the two regions on either side of the line have the same colour.

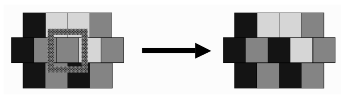

FIGURE 10.6 The way that mutation is performed on a colour, changing it into one of the other colours.

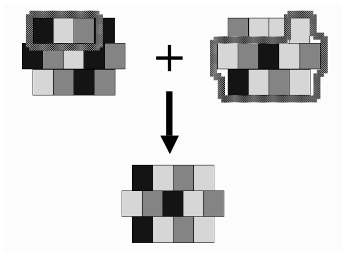

FIGURE 10.7 The effects of crossover on a map.

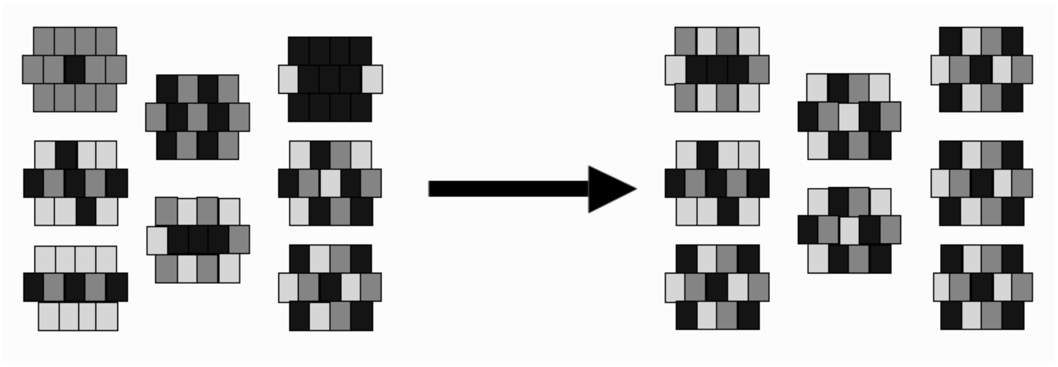

FIGURE 10.8 One generation of the GA working on the map colouring problem.

However, we could also just count the number of correct edges. The example in Figure 10.5 has 16 out of the 26 boundaries correct (where a boundary is the intersection between any two squares), so its fitness is 16.

Choose suitable genetic operators We'll use the standard genetic operators for this, since this example makes the operations of crossover and mutation clear. The way that they are used is shown in Figures 10.6 and 10.7. In general, people just use the standard operators for most problems, but if they don't work well, it can be worth putting some effort into thinking of new ones.

Having made those choices, we can let the GA run on the problem, with a possible population and their offspring shown in Figure 10.8, and look at the best solutions after some preset number of iterations. The GA produces good solutions to this problem, and implementing it for yourself is one of the suggested exercises for this chapter.

10.3.2 Punctuated Equilibrium

For a long time, one thing that creationists and others who did not believe in evolution used as an argument against it was the problem of the lack of intermediate animals in the fossil record. The argument runs that if humans evolved from apes, then there should be some evidence of a whole set of intermediary species that existed during the transition phase, and there aren't. Interestingly, GAs demonstrate one of the explanations why this is not correct, which is that the way that evolution actually seems to work is known as punctuated

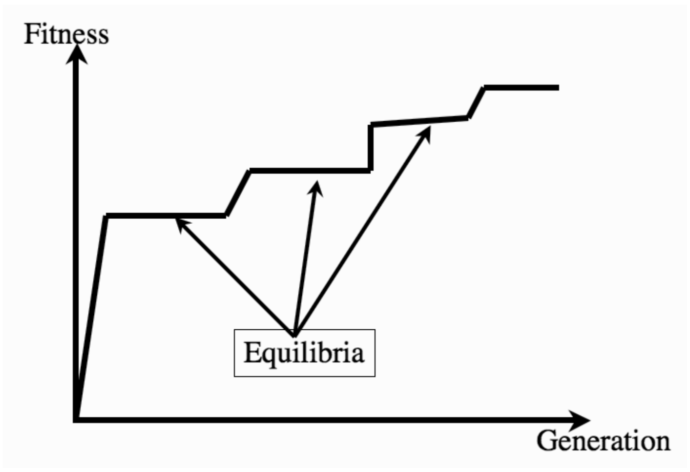

FIGURE 10.9 A graph showing punctuated equilibrium in a genetic algorithm. There is an effectively steady state where fitness does not improve, followed by rapid improvements in fitness until another steady state is reached.

equilibrium. There is basically a steady population of some species for a long time, and then something changes and over a very short (in evolutionary terms... still hundreds or thousands of years) period, there is a big change, and then everything settles down again. So the chance of finding fossils from the intermediary stage is quite small. There is a graph showing this effect in Figure 10.9.

10.3.3 Example: The Knapsack Problem

We used the knapsack problem as an example while we were looking at components of the GA. It is now time to see it being solved. Before we do that, we can use some of the methods from Section 9.4 to solve it. We've already mentioned the greedy algorithm solution, and we can of course use exhaustive search, as well, or any of the other methods we discussed in the last chapter, such as simulated annealing or hillclimbing.

The website has a simple example with 20 different packages, which have a total size of 2436.77 and a maximum knapsack size of 500. The greedy algorithm finds a solution of 487.47, while the optimal solution is eventually found by the exhaustive search as 499.98. The question is how well the GA does on the same problem. We will use the fitness function that was described in Section 10.1.2, where solutions that are too large are penalised by having twice the amount they are over subtracted from the maximum size. Figure 10.10 shows a graph of the output when the GA is run on this problem for 100 iterations. The GA rapidly finds a near-optimal solution (of 499.94) to this relatively simple problem, although in this run it did not find the global optimum.

10.3.4 Example: The Four Peaks Problem

The four peaks is a toy problem (that is, simple problem that isn't useful itself, but is good for testing algorithms) that is quite often used to test out GAs and various developments of them. It is an invented fitness function that rewards strings with lots of consecutive 0s at the start of the string, and lots of consecutive 1s at the end. The fitness consists of counting

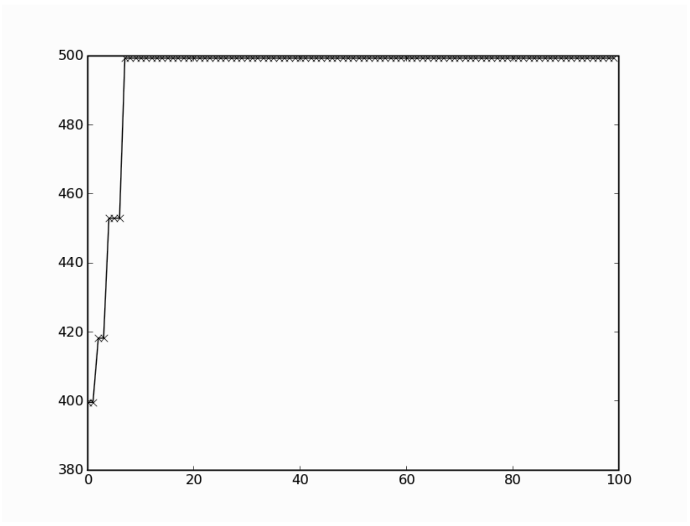

FIGURE 10.10 Evolution of the solution to the knapsack problem. The GA finds a very good solution to this simple problem within a few iterations, but never finds the optimal solution.

the number of 0s at the start, and the number of 1s at the end and returning the maximum of them as the fitness. However, if both the number of 0s and the number of 1s are above some threshold value T then the fitness function gets a bonus of 100 added to it. This is where the name 'four peaks' comes from: there are two small peaks where there are lots of 0s, or lots of 1s, and then there are two larger peaks, where the bonus is included. The GA should find these larger peaks for a successful run.

In NumPy the four peaks fitness function can be written as:

```
def fourpeaks(population,T=15):

    start = np.zeros((np.shape(population)[0],1))
    finish = np.zeros((np.shape(population)[0],1))

    fitness = np.zeros((np.shape(population)[0],1))

    for i in range(np.shape(population)[0]):
        s = np.where(population[i,:]==1)
        f = np.where(population[i,:]==0)
        if np.size(s)>0:
            start = s[0][0]
        else:
            start = 0

        if np.size(f)>0:
            finish = np.shape(population)[1] - f[-1][-1] -1
        else:
```

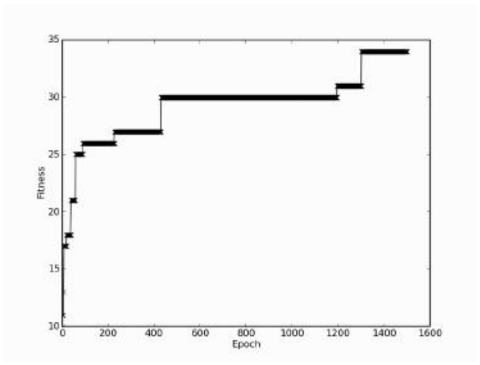

FIGURE 10.11 Evolution of a solution to the four peaks problem. The solution never reaches the bonus score in the fitness function.

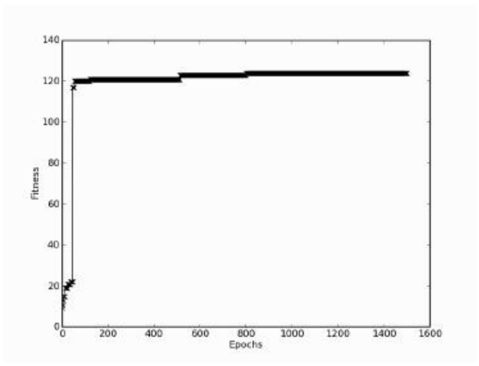

FIGURE 10.12 Another solution to the four peaks problem. This solution does reach the bonus score, but does not get the global maximum.

```
                finish = 0

            if start>T and finish>T:
                fitness[i] = np.maximum(start,finish)+100
            else:
                fitness[i] = np.maximum(start,finish)

        fitness = np.squeeze(fitness)
        return fitness
```

Figures 10.11 and 10.12 show the outputs of two runs for a chromosome length of 100 and with $T = 15$. In the second the GA reaches the bonus point, while in the first it does not. Both of these runs used a mutation rate of 0.01, which is $1/L$, and single point crossover. They also used elitism.

10.3.5 Limitations of the GA

There are lots of good things about genetic algorithms, and they work amazingly well a lot of the time. However, they are not without problems, a significant one of which is they can be very slow. The main problem is that once a local maximum has been reached, it can often be a long time before a string is produced that escapes from the local maximum and finds another, higher, maximum. In addition, because we generally do not know anything about the fitness landscape, we can't see how well the GA is doing.

A more basic criticism of genetic algorithms is that it is very hard (read basically impossible) to analyse the behaviour of the GA. We expect that the mean fitness of the population will increase until an equilibrium of some kind is reached. This equilibrium is between the selection operator, which makes the population less diverse, but increases the mean fitness (exploitation), and the genetic operators, which usually reduce the mean fitness, but increase the diversity in the population (exploration). However, proving that this is guaranteed to happen has not been possible so far, which means that we cannot guarantee that the algorithm will converge at all, and certainly not to the optimal solution. This bothers a lot of researchers. That said, genetic algorithms are widely used when other methods do not work, and they are usually treated as a black box—strings are pushed in one end, and eventually an answer emerges. This is risky, because without knowledge of how the algorithm works it is not possible to improve it, nor do you know how cautiously you should treat the results.

10.3.6 Training Neural Networks with Genetic Algorithms

We trained our neural networks, most notably the MLP, using gradient descent. However, we could encode the problem of finding the correct weights as a set of strings, with the fitness function measuring the sum-of-squares error. This has been done, and with good reported results. However, there are some problems with this approach. The first is that we turn all the local information from the targets about the error at each output node of the network into just one number, the fitness, which is throwing away useful information, and the second is that we are ignoring the gradient information, which is also throwing away useful information.

A more sensible use for GAs with neural networks is to use the GA to choose the topology of the network. Previously, we chose the structure in a completely ad hoc way by trying out different structures and choosing the one that worked best. We can use a GA for this problem, although the crossover operator doesn't make a great deal of sense, so we just consider mutation. However, we allow for four different types of mutation: delete a neuron, delete a weight connection, add a neuron, add a connection. The deletion operators bias the learning towards simple networks. Making the GA more complicated by adding extra mutation operators might make you wonder if you can make it more complicated again. And you can; one example of where this can lead is discussed next.

10.4 GENETIC PROGRAMMING

One extension of genetic algorithms that has had a lot of attention is the idea of genetic programming. This was introduced by John Koza, and the basic idea is to represent a computer program as a tree (imagine a flow chart of the code). For certain programming languages, notably LISP, this is actually a very natural way to represent a program, but it doesn't work very well in Python, so we will have a quick look at the idea, but not get into writing any explicit algorithms for the method. Tree-based variants on mutation and crossover are defined (replace subtrees by other subtrees, either randomly generated (mutation, Figure 10.13) or swapped from another tree (crossover, Figure 10.14)), and then the genetic program runs just like a normal genetic algorithm, but acting on these program trees rather than strings.

Figure 10.15 shows a set of simple trees that perform arithmetic operations, and some possible developments of them, made using these operators.

Genetic programming has been used for many different tasks, from recognising skin melanomas to circuit design, and lots of very impressive results have been claimed for it. However, the search space is unbelievably large, and the mutation operator not especially useful, and so a lot depends upon the initial population. A set of possibly useful subtrees is usually chosen by the system developer first in order to give the system a head start. There

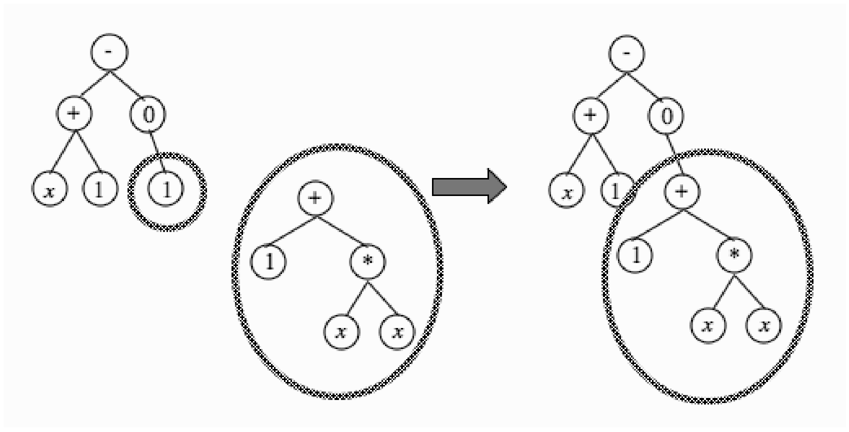

FIGURE 10.13 Example of a mutation in genetic programming.

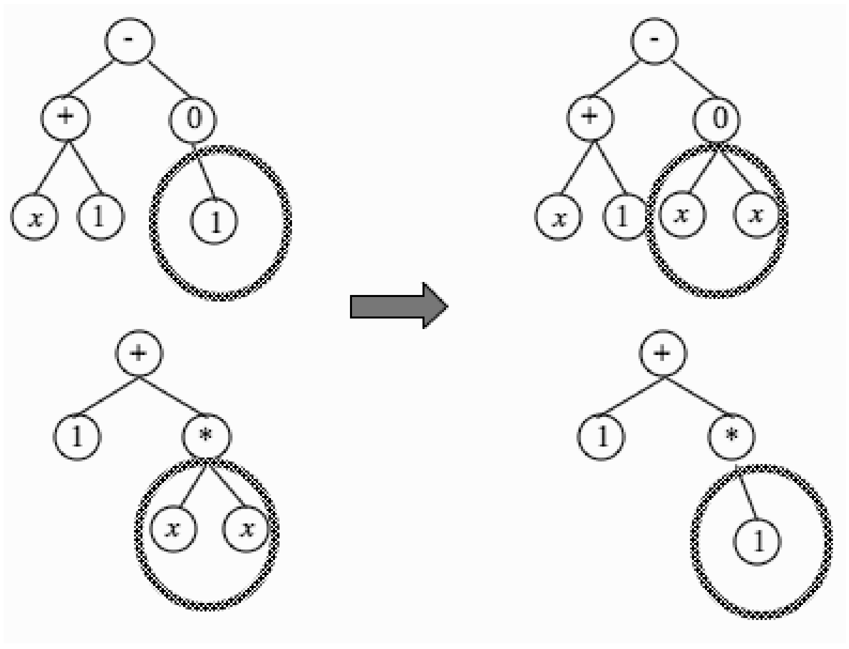

FIGURE 10.14 Example of a crossover in genetic programming.

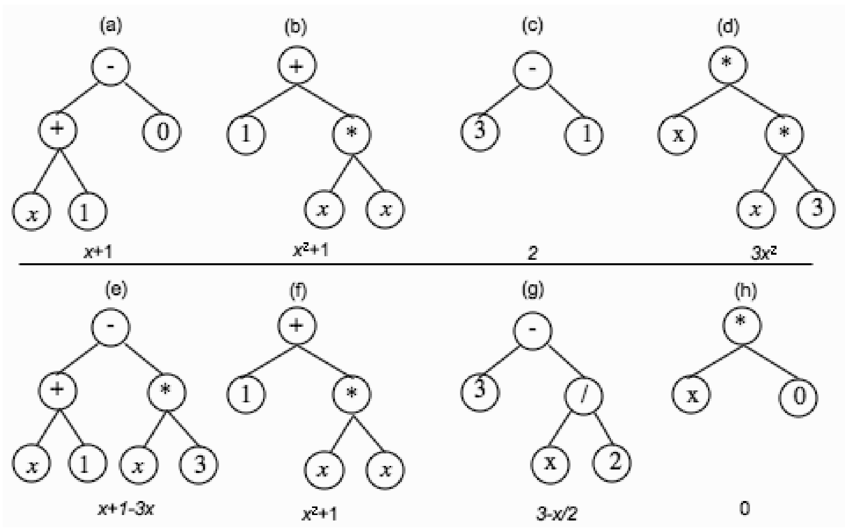

FIGURE 10.15 *Top:* Four arithmetical trees. *Bottom:* Example developments of the four trees: (e) and (h) are a possible crossover of (a) and (d), (f) is a copy of (b), and (g) is a mutation of (c).

are a couple of places where you can find more information on genetic programming in the Further Reading section.

10.5 COMBINING SAMPLING WITH EVOLUTIONARY LEARNING

The last machine learning method in this chapter is an interesting variation on the theme of evolutionary learning, combined with probabilistic models of the type that are described in Chapter 16, namely Bayesian networks. They are often known as estimation of distribution algorithms (EDA).

The most basic version is known as Population-Based Incremental Learning (PBIL), and it is amazingly simple. It works on a binary alphabet, just like the basic GA, but instead of maintaining a population, it keeps a probability vector p that gives the probability of each element being a 0 or 1. Initially, each value of this vector is 0.5, so that each element has equal chance of being 0 or 1. A population is then constructed by sampling from the distribution specified vector, and the fitness of each member of the population is computed. A subset of this population (typically just the two fittest vectors) is chosen to update the probability vector, using a learning rate η, which is often set to 0.005 (where `best` and `second` represent the best and second-best elements of the population):

$$p = p \times (1-\eta) + \eta(\text{best} + \text{second})/2. \tag{10.3}$$

The population is then thrown away, and a new one sampled from the updated probability vector. The results of using this simple algorithm on the four-peaks problem with $T = 11$ are shown in Figure 10.16 using strings of length 100 with 200 strings in each population. This is directly comparable with Figure 10.12.

The centre of the algorithm is simply the code to find the strings with the two highest fitnesses and use them to update the vector. Everything else is directly equivalent to the genetic algorithm.

```
# Pick best
best[count] = np.max(fitness)
bestplace = np.argmax(fitness)
fitness[bestplace] = 0
secondplace = np.argmax(fitness)

# Update vector
p  = p*(1-eta) + eta*((pop[bestplace,:]+pop[secondplace,:])/2)
```

The probabilistic model that is used in PBIL is very simple: it is assumed that each element of the probability vector is independent, so that there is no interaction. However, there is no reason why more complicated interactions between variables cannot be considered, and several methods have been developed that do exactly this. The first option is to construct a chain, so that each variable depends only on the one to its left. This might involve sorting the order of the probability vector first, but then the algorithm simply needs to measure the mutual information (see Section 12.2.1) between each pair of neighbouring variables. This use of mutual information gives the algorithm its name: MIMIC. There are also more complicated variants using full Bayesian networks, such as the Bayesian Optimisation Algorithm (BOA) and Factorised Distribution Algorithm (FDA).

The power of these developments of the GA is that they use probabilistic models and are therefore more amenable to analysis than normal GAs, which have steadfastly withstood many attempts to better understand their behaviour. They also enable the algorithm to discover correlations between input variables, which can be useful if you want to understand the solution rather than just apply it.

It is important to remember that there is no guarantee that a genetic algorithm will find a good solution, although it often will, and certainly no guarantee that it will find the optimum. The vast majority of applications of genetic algorithms and the other algorithms described in this chapter do not consider this, but use the algorithms as a way to avoid having to understand the problem. Recall the No Free Lunch theorem of the last chapter—there is no universally good solution to the search problem—before using the GA or genetic program as the only search method that you use. Having said that, providing that you are prepared to accept the long running time and the fact that there are no guarantees of a good solution, they are frequently very useful methods.

FURTHER READING

There are entire books written about genetic algorithms, including:

- J.H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence.* MIT Press, Cambridge, MA, USA, 1992.

- M. Mitchell. *An Introduction to Genetic Algorithms.* MIT Press, Cambridge, MA, USA, 1996.

- D.E. Goldberg. *Genetic Algorithms in Search, Optimisation, and Machine Learning.* Addison-Wesley, Reading, MA, USA, 1999.

There are also entire books on genetic programming, including:

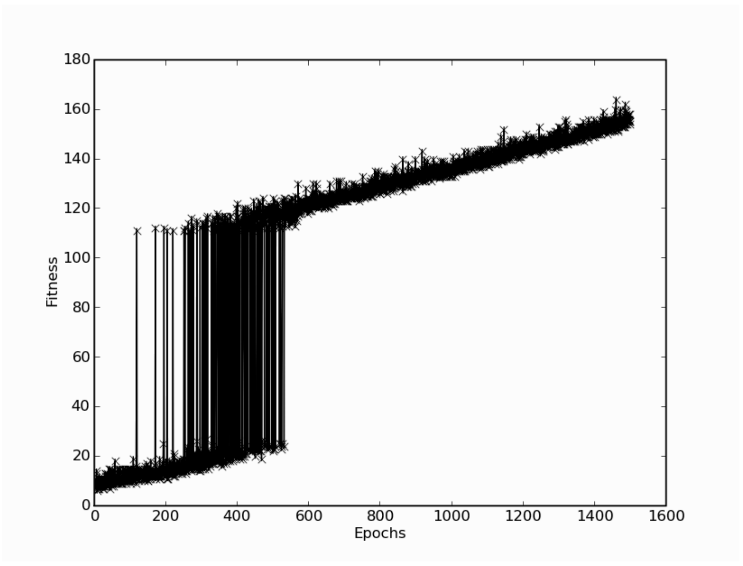

FIGURE 10.16 The evolution of the best fitness using PBIL on the four peaks problem.

- J.R. Koza. *Genetic Programming: On the Programming of Computers by the Means of Natural Selection.* MIT Press, Cambridge, MA, USA, 1992.
- Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*, 3rd edition, Springer, Berlin, Germany, 1999.

For more on Estimation of Distribution algorithms, look at:

- S. Baluja and R. Caruana. Removing the genetics from the standard genetic algorithm. In A. Prieditis and S. Russel, editors, *The International Conference on Machine Learning*, pages 38–46, Morgan Kaufmann Publishers, San Mateo, CA, USA, 1995.
- M. Pelikan, D.E. Goldberg, and F. Lobo. A survey of optimization by building and using probabilistic models. *Computational Optimization and Applications*, 21(1):5–20, 2002. Also IlliGAL Report No. 99018.

Details of the two books mentioned about real evolution are:

- C. Darwin. *On the Origin of Species by Means of Natural Selection*, 6th edition, Wordsworth, London, UK, 1872.
- R. Dawkins. *The Blind Watchmaker: Why the Evidence of Evolution Reveals a Universe without Design.* Penguin, London, UK, 1996.

PRACTICE QUESTIONS

Problem 10.1 Suppose that you want to archive your data files, but you have only got one CD, and more data files than will fit on it. You decide to choose the files you will save so as to try to maximise the amount of space you fill on the disk, so that the most data is backed up, but you can't split a data file. Write a greedy algorithm and a hill-climbing algorithm to solve this problem. What guarantees can you make about efficiency of the solutions?

Problem 10.2 (from Jon Shapiro)
In video poker, you are dealt five cards face up. You have one chance to replace any of the cards (or all or none) with cards drawn from the deck. You then get a payout related to the value of your hand as a poker hand. Say your stake is \$1. The lowest hand which pays is pair of jacks or better; this pays \$1 (so your net gain is 0). Two pair pays \$2, three-of-a-kind pays \$3, and so forth. Your goal is to make as much money as possible.

In order to play this game, you need a strategy for deciding which cards to keep and which to replace. For example, if your hand contains two face cards, but is currently worthless, should you hold them both or hold only one? If one is held, there are four chances to match one card; if two are held there are only three chances but there are two cards to match. If the hand contains a pair of low cards, is it better to keep the pair in the hopes of drawing another pair or a card which turns the pair into three-of-a-kind, or is it better to draw five new cards? It is unclear what is the best strategy for replacing cards in losing hands. Devise a way to use a genetic algorithm to search for good strategies for playing this game. Assume that you have a computer version of the game, so that any strategies which the GA proposes can be tested on the computer over many plays. Could an MLP using gradient descent learning be used to learn a good strategy? Why or why not?

Problem 10.3 You have 5000 MP3 files sitting on your computer's hard disk. Unfortunately, the hard disk has started making noises, and you decide that you had better back up the MP3s. Equally unfortunately, you can only burn CDs, not DVDs, on your computer. You need to minimise the number of CDs that you use, so you decide to design a genetic algorithm to choose which MP3s to put onto each CD in order to fill each CD as completely as possible.

Design a genetic algorithm to solve the problem. You will need to consider how you would encode the inputs, which genetic operators are suitable, and how you would make the genetic algorithm deal with the fact that you have multiple CDs, not just one CD.

Problem 10.4 Convert the GA to use real-valued chromosomes and use it to find the minima in Rosenbrock's function (Equation (9.18)).

Problem 10.5 Implement the map colouring fitness function (you will have to design a map first, of course) and see how good the solutions that the GA finds are. Compare maps that are three-colourable with some that are not. Can you think of any other algorithm that could be used to find solutions to this problem?

Problem 10.6 The Royal Road fitness function is designed to test the building block hypothesis, which says that GAs work by assembling small building blocks and then put them together by crossover. The function splits the binary string into l sequential pieces, all b bits long. The fitness of the piece is b for blocks that are all 1s, and 0 for others, and the total fitness is the sum of the fitness for each block. Implement this fitness function and test it on strings of length 16, with blocks of lengths 1, 2, 4, 8. Run your GAs for 10,000 iterations. Compare the results to using PBIL.

CHAPTER 11

Reinforcement Learning

Reinforcement learning fills the gap between supervised learning, where the algorithm is trained on the correct answers given in the target data, and unsupervised learning, where the algorithm can only exploit similarities in the data to cluster it. The middle ground is where information is provided about whether or not the answer is correct, but not how to improve it. The reinforcement learner has to try out different strategies and see which work best. That 'trying out' of different strategies is just another way of describing search, which was the subject of Chapters 9 and 10. Search is a fundamental part of any reinforcement learner: the algorithm searches over the state space of possible inputs and outputs in order to try to maximise a reward.

Reinforcement learning is usually described in terms of the interaction between some agent and its environment. The agent is the thing that is learning, and the environment is where it is learning, and what it is learning about. The environment has another task, which is to provide information about how good a strategy is, through some reward function.

Think about a child learning to stand up and walk. The child tries out many different strategies for staying upright, and it gets feedback about which work by whether or not it ends up flat on its face. The methods that seem to work are tried over and over again, until they are perfected or better solutions are found, and those that do not work are discarded. This analogy has another useful aspect: it may well not be the last thing that the child does before falling that makes it fall over, but something that happened earlier on (it can take several desperate seconds of waving your arms around before you fall over, but the fall was caused by tripping over something, not by waving your arms about). So it can be difficult to work out which action (or combination of actions) made you fall over, because there are many actions in the chain.

The importance of reinforcement learning for psychological learning theory comes from the concept of trial-and-error learning, which has been around for a long time, and is also known as the Law of Effect. This is exactly what happens in reinforcement learning, as we'll see, and it was described in a book by Thorndike in 1911 as:

> Of several responses made to the same situation, those which are accompanied or closely followed by satisfaction to the animal will, other things being equal, be more firmly connected with the situation, so that, when it recurs, they will be more likely to recur; those which are accompanied or closely followed by discomfort to the animal will, other things being equal, have their connections with that situation weakened, so that, when it recurs, they will be less likely to occur. The greater the satisfaction or discomfort, the greater the strengthening or weakening of the bond. (E. L. Thorndike, *Animal Intelligence*, page 244.)

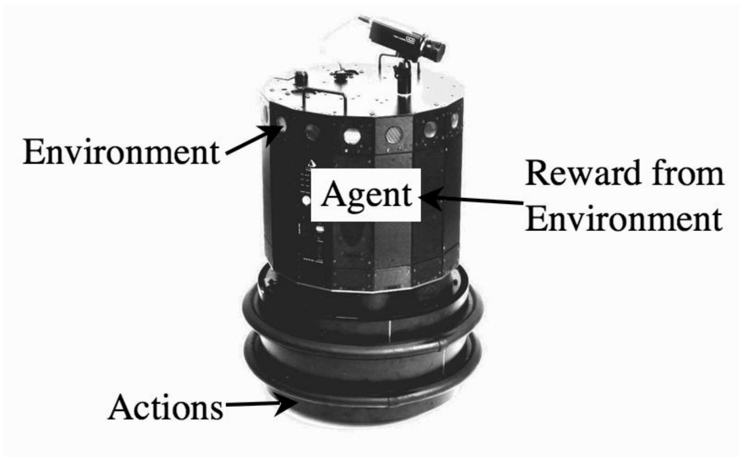

FIGURE 11.1 A robot perceives the current state of its environment through its sensors, and performs actions by moving its motors. The reinforcement learner (agent) within the robot tries to predict the next state and reward.

This is where the name 'reinforcement learning' comes from, since you repeat actions that are reinforced by a feeling of satisfaction. To see how it can be applied to machine learning, we will need some more notation.

11.1 OVERVIEW

Reinforcement learning maps states or situations to actions in order to maximise some numerical reward. That is, the algorithm knows about the current input (the state), and the possible things it can do (the actions), and its aim is to maximise the reward. There is a clear distinction drawn between the agent that is doing the learning and the environment, which is where the agent acts, and which produces the state and the rewards. The most common way to think about reinforcement learning is on a robot. The current sensor readings of the robot, or processed versions of them, could define the state. They are a representation of the environment around the robot in some way. Note that the state doesn't necessarily tell us everything that it would be useful to know (the robot's sensors don't tell it its location, only what it can see about it), and there can be noise and inaccuracies in the state data. The possible ways that the robot can drive its motors are the actions, which move the robot in the environment, and the reward could be how well it does its task without crashing into things. Figure 11.1 shows the idea of state, actions, and environment to a robot, while Figure 11.2 shows how they are linked with each other and with the reward.

In reinforcement learning the algorithm gets feedback in the form of the reward about how well it is doing. In contrast to supervised learning, where the algorithm is 'taught' the correct answer, the reward function evaluates the current solution, but does not suggest how to improve it. Just to make the situation a little more difficult, we need to think about the possibility that the reward can be delayed, which means that you don't actually get the reward until a long time in the future. (For example, think about a robot that is learning to traverse a maze. It doesn't know whether it has found the centre of the maze until it gets there, and it doesn't get the reward until it reaches the centre of the maze.) We therefore need to allow for rewards that don't appear until long after the relevant actions have been

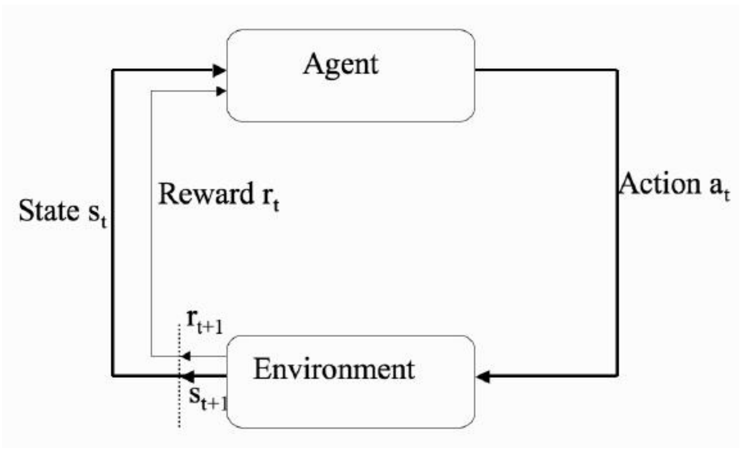

FIGURE 11.2 The reinforcement learning cycle: the learning agent performs action a_t in state s_t and receives reward r_{t+1} from the environment, ending up in state s_{t+1}.

taken. Sometimes we think of the immediate reward and the total expected reward into the future.

Once the algorithm has decided on the reward, it needs to choose the action that should be performed in the current state. This is known as the policy. This is done based on some combination of exploration and exploitation (remember, reinforcement learning is basically a search method), which in this case means deciding whether to take the action that gave the highest reward last time we were in this state, or trying out a different action in the hope of finding something even better.

11.2 EXAMPLE: GETTING LOST

You arrive in a foreign city exhausted after many hours of flying, catch the train into town and stagger into a backpacker's hostel without noticing much of your surroundings. When you wake up it is dark and you are starving, so you set off for a wander around town looking for somewhere to eat. Unfortunately, it is 3 a.m. and, even more unfortunately, you soon realise that you are completely lost. To make matters worse, you discover that you can't remember the name of the backpacker's, or much else about it except that it is in one of the old squares. Of course, that doesn't help much because this part of the city pretty much consists of old squares. There are only two things in your favour: you are fairly sure that you'll recognise the building, and you've studied reinforcement learning and decide to apply it (yes, this book can save your life!).

You are sure that you've only walked through the old part of the city, so you don't need to worry about any street that takes you out of the old part. So at the next bus stop you come to, you have a proper look at the map, and note down the map of the old town squares, which turns out to look like Figure 11.3.

As you finish drawing the map you notice a 24-hour shop and buy as many bags of potato chips as you can fit into your pockets. As a reinforcement learner you decide to reward yourself when you take actions that lead to the backpacker's rather than stuff your face immediately (this is a delayed reward). After thinking about a reward structure you decide that the only one that will work is to eat until you can eat no more when you

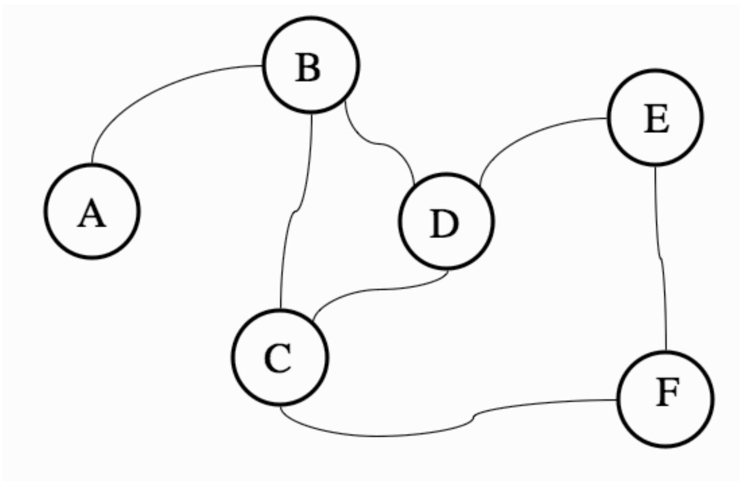

FIGURE 11.3 The old town that you find yourself lost in.

actually get to the backpacker's, and not to reward yourself at all until then. You'll just have to hope that you don't faint from hunger first!

Inspired by the idea of food, you decide that the backpacker's is almost definitely in the square labelled F on the map, because its name seems vaguely familiar. You decide to work out a reward structure so that you can follow a reinforcement learning algorithm to get to the backpacker's. The first thing you work out is that staying still means that you are sleeping on your feet, which is bad. So you assign a reward of -5 for that (while negative reinforcement can be viewed as punishment, it doesn't necessarily correspond clearly, but you might want to imagine it as pinching yourself so that you stay awake). Of course, once you reach state F you are in the backpacker's and will therefore stay there. This is known as an absorbing state, and is the end of the problem, when you get the reward of eating all the chips you bought. Now moving between two squares could be good, because it might take you closer to F. But without looking at the map you won't know that, so you decide to just apply a reward when you actually reach F, and leave everything else as neutral. Where there is no direct road between two squares (so that no action takes you from one to the other) there is no reward because it is not a viable action. This results in the reward matrix R shown below (where '-' shows that there is no link) and also in Figure 11.4.

	Next State					
Current State	A	B	C	D	E	F
A	-5	0	-	-	-	-
B	0	-5	0	0	-	-
C	-	0	-5	0	-	100
D	-	0	0	-5	0	-
E	-	-	-	0	-5	100
F	-	-	0	-	0	-

Of course, as a reinforcement learner you don't actually know the reward matrix. That's pretty much what you are trying to discover, but it doesn't make for a very good example. We'll assume that you have now reached a stage of tiredness where you can't even read what is on your paper properly. Having got this set up we've reached the stage where we

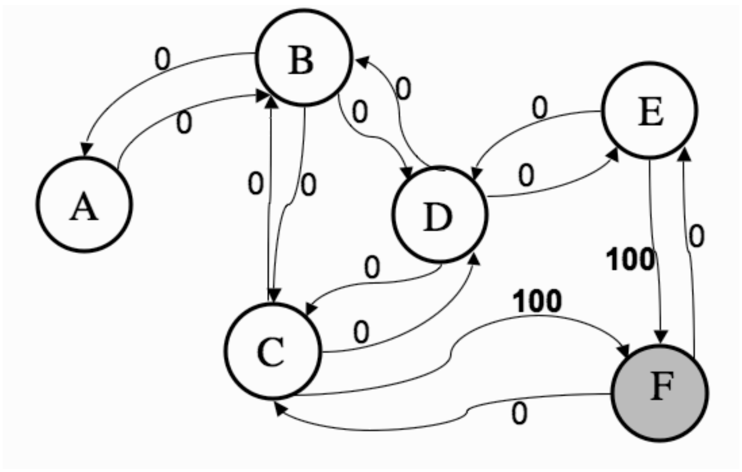

FIGURE 11.4 The *state diagram* if you are correct and the backpacker's is in square (state) F. The connections from each state back into itself (meaning that you don't move) are not shown, to avoid the figure getting too complicated. They are each worth -5 (except for staying in state F, which means that you are in the backpacker's).

need to do some learning, but for now we'll leave you stranded in that foreign city and flesh out a few of the things that we've talked about so far.

11.2.1 State and Action Spaces

Our reinforcement learner is basically a search algorithm, and obviously the larger the number of states that the algorithm has to search through, the longer it will take to find a good solution. The set of all states that are possible for the learner to experience is known as the state space. There is a corresponding action space that contains all of the possible actions. If we can reduce the size of the state space and action space, then it is almost always a good idea, providing that it does not oversimplify the problem. In the example there are only six states, but still, look at Figure 11.4 and imagine wandering through all of the squares over and over again while we search: it seems like this learning is going to take a long time. And it generally does.

Computing the size of the state space (and the corresponding action space) is relatively simple. For example, suppose that there are five inputs, each an integer between 0 and 100. The size of the state space is then $100 \times 100 \times 100 \times 100 \times 100 = 100^5$, which is incredibly large, so the curse of dimensionality is really kicking in here. However, if we decide that we can quantize the data so that instead of 100 numbers there are only two for each input (for example, by assigning every number less than 50 to class 1, and every number 50 and above to class 2), then the size of the state space is a more manageable $2^5 = 32$. Choosing the state space and action space carefully is therefore a crucial part of making a successful reinforcement learner. You want them to be as small as possible without losing accuracy in the results—by reducing the scale of each input from 100 to 2, we have obviously thrown away a lot of information that might have made the quality of the answer better. As is usually the case, there is some element of compromise between the two.

11.2.2 Carrots and Sticks: The Reward Function

The basic idea of the learner is that it will choose the action that gets the maximum expected reward. In the example, we worked out what the rewards would be in a fairly ad hoc way, by saying what we wanted and then thinking about how to get it. That's pretty much the way that it works in practice, too: in Chapter 10 where we looked at genetic algorithms, we had to carefully craft the fitness function to solve the problem that we wanted, and the same thing is true of the reward function. In fact, they can be seen as the same thing.

The reward function takes the current state and the chosen action and produces a numerical reward based on them. So in the example, if we are in state A, and choose the action of doing nothing, so that we remain in state A, we get a reward of -5. Note that the reward can be positive or negative, with the latter corresponding to 'punishment', showing that particular actions should be avoided. The reward is generated by the environment around the learner; it is not internal to the learner itself (this is what makes it difficult to describe in our example: the environment doesn't give you rewards in the real world, only when there is a computer (or brain) as part of the environment to help out). In effect, the reward function makes the goal of the learner explicit—the learner is trying to maximise the reward, which means behaving in exactly the way that the reward function expects. The reward tells the learner what the goal is, not how the goal should be achieved, which would be supervised learning. It is therefore usually a bad idea to include sub-goals (extra things that the learner should achieve along the way, which are meant to speed up learning), because the learner can find methods of achieving the sub-goals without actually achieving the real goal.

The choice of a suitable reward function is absolutely crucial, with very different behaviours resulting from varying the reward function. For example, consider the difference between these two reward functions for a maze-traversing robot (try to work out the difference before reading the paragraph that follows them):

- receive a reward of 50 when you find the centre of the maze
- receive a reward of -1 for each move and a reward of +50 when you find the centre of the maze

In the first version, the robot will learn to get to the centre of the maze, just as it will in the second version, but the second reward function is biased towards shorter routes through the maze, which is probably a good thing. The maze problem is episodic: learning is split into episodes that have a definite endpoint when the robot reaches the centre of the maze. This means that the rewards can be given at the end and then propagated back through all the actions that were performed to update the learner. However, there are plenty of other examples that are not episodic (continual tasks), and there is no cut off when the task stops. An example is the child learning to walk that was mentioned at the start of the chapter. A child can walk successfully when it doesn't fall over at all, not when it doesn't fall over for 10 minutes.

Now that the reward has been broken into two parts—an immediate part and a pay-off in the end—we need to think about the learning algorithm a bit more. The thing that is driving the learning is the total reward, which is the expected reward from now until the end of the task (when the learner reaches the terminal state or accepting state—the backpacker's in our example). At that point there is generally a large pay-off that signals the successful completion of the task. However, the same thing does not work for continual tasks, because there is no terminal state, so we want to predict the reward forever into the infinite future, which is clearly impossible.

11.2.3 Discounting

The solution to this problem is known as discounting, and means that we take into account how certain we can be about things that happen in the future: there is lots of uncertainty in the learning anyway, so we should discount our predictions of rewards in the future according to how much chance there is that they are wrong. The rewards that we expect to get very soon are probably going to be more accurate predictions than those a long time in the future, because lots of other things might change. So we add an additional parameter $0 \leq \gamma \leq 1$, and then discount future rewards by multiplying them by γ^t, where t is the number of timesteps in the future this reward is from. As γ is less than 1, so γ^2 is smaller again, and $\gamma^k \to 0$ as $k \to \infty$ (i.e., γ gets smaller and smaller as k gets larger and larger), so that we can ignore most of the future predictions. This means that our prediction of the total future reward is:

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots + \gamma^{k-1} r_k + \ldots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}. \tag{11.1}$$

Obviously, the closer γ is to zero, the less distance we look into the future, while with $\gamma = 1$ there is no discounting, as in the episodic case above (in fact, discounting is sometimes used for episodic learning as well, since the eventual reward could be a very long way off and we have to deal with that uncertainty in learning somehow). We can apply discounting to the example of learning to walk. When you fall over you give yourself a reward of -1, and otherwise there are no rewards. The -1 reward is discounted into the future, so that a reward k steps into the future has reward $-\gamma^k$. The learner will therefore try to make k as large as possible, resulting in proper walking.

The point of the reward function is that it gives us a way to choose what to do next—our predictions of the reward let us exploit our current knowledge and try to maximise the reward we get. Alternatively, we can carry on exploring and trying out new actions in the hope that we find ways to get even larger rewards. The methods of exploration and exploitation that we carry out are the methods of action selection that we perform.

11.2.4 Action Selection

At each stage of the reinforcement learning process, the algorithm looks at the actions that can be performed in the current state and computes the value of each action; that is, the average reward that is expected for carrying out that action in the current state. The simplest way to do this is to compute the average reward that has been received each time in the past. This is known as $Q_{s,t}(a)$, where s is the state, a is the action, and t is the number of times that the action has been taken before in this state. This will eventually converge to the true prediction of the reward for that action. Based on the current average reward predictions, there are three methods of choosing action a that are worth thinking about for reinforcement learning. We've seen the first and third of them before:

Greedy Pick the action that has the highest value of $Q_{s,t}(a)$, so always choose to exploit your current knowledge.

ϵ-greedy This is similar to the greedy algorithm, but with some small probability ϵ we pick some other action at random. So nearly every time we take the greedy option, but occasionally we try out an alternative in the hope of finding a better action. This throws some exploration into the mix. ϵ-greedy selection finds better solutions over time than the pure greedy algorithm, since it can explore and find better solutions.

Soft-max One refinement of ϵ-greedy is to think about which of the alternative actions to select when the exploration happens. The ϵ-greedy algorithm chooses the alternatives with uniform probability. Another possibility is to use the soft-max function (which we've seen repeatedly, e.g., as Equation (4.12)) to make the selection:

$$P(Q_{s,t}(a)) = \frac{\exp(Q_{s,t}(a)/\tau)}{\sum_b \exp(Q_{s,t}(b)/\tau)}. \tag{11.2}$$

Here, there is a new parameter τ, which is known as the temperature because of the link to simulated annealing, see Section 9.6. When τ is large, all actions have similar probabilities, and when τ is small, the selection probabilities matter more. In soft-max selection, the current best (greedy) action will be chosen most of the time, but the others will be chosen proportional to their estimated reward, which is updated whenever they are used.

11.2.5 Policy

We have just considered different action selection methods, such as ϵ-greedy and soft-max. The aim of the action selection is to trade off exploration and exploitation in such a way as to maximise the expected reward into the future. Instead, we can make an explicit decision that we are going to always take the optimal choice at each stage, and not do exploration any more. This choice of which action to take in each state in order to get optimal results is known as the policy, π. The hope is that we can learn a better policy that is specific to the current state s_t. This is the crux of the learning part of reinforcement learning—learn a policy π from states to actions. There is at least one optimal policy that gives the maximum reward, and that is what we want to find. In order to find a policy, there are a few things that we need to worry about. The first is how much information we need to know regarding how we got to the current state, and the second is how we ascribe a value to the current state. The first one is important enough both for here and for Chapter 16 that we are going to go into some detail now.

11.3 MARKOV DECISION PROCESSES

11.3.1 The Markov Property

Let's go back to the example. Standing in the square labelled D you need to make a choice of what action to take next. There are four possible options (see Figure 11.4): standing still, or moving to one of B, C, or E. The question is whether or not that is enough information for you to predict the reward accurately and so to choose the best possible action. Or do you also need to know where you have been in the past? Let's say that you know that you came to D from B. In that case, maybe it does not make sense to move back to B, since your reward won't change. However, if you came to D from E then it does actually make sense to go back there, since it moves you closer to F. So in this case, it appears that knowing your previous action doesn't actually help very much, because you don't have enough information to work out what was useful.

Another example where this is usually true is in a game of chess, where the current situation of all the pieces on the board (the state) is enough to predict whether or not the next move is a good one—it does not depend on precisely how each piece got to the current location. Thus, the current state provides enough information. A state that has this property, which is that the current state provides enough information for the reward to be

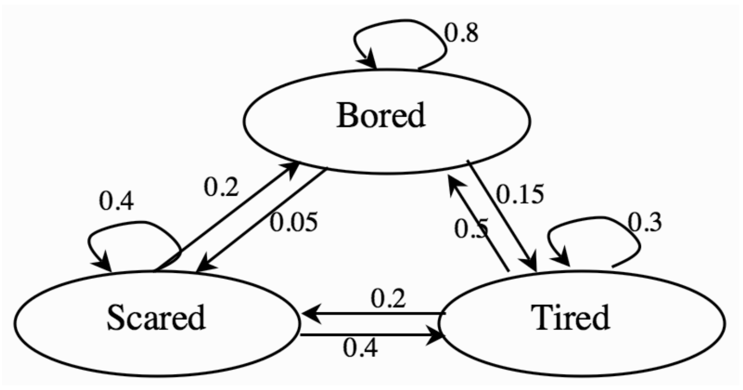

FIGURE 11.5 A simple example of a Markov decision process to decide on the state of your mind tomorrow given your state of mind today.

computed without looking at previous states, is known as a Markov state. The importance of this can be seen from the following two equations, the first of which is what is required when the Markov property is not true, while for the second one it is true. The equation is the computation of the probability that the next reward is r' and the next state is s'.

$$Pr(r_t = r', s_{t+1} = s'|s_t, a_t, r_{t-1}, s_{t-1}, a_{t-1}, \dots r_1, s_1, a_1, r_0, s_0, a_0), \tag{11.3}$$

$$Pr(r_t = r', s_{t+1} = s'|s_t, a_t). \tag{11.4}$$

Clearly, Equation (11.4), which depends only on where you are now, and what you choose to do now, is much simpler to compute, less likely to suffer from rounding errors, and does not require that the whole history of the learner is stored. In fact, it makes the computation possible, whereas the first is not possible for any interesting problem. A reinforcement learning problem that follows Equation (11.4) (that is, that has the Markov property) is known as a Markov Decision Process (MDP). It means that we can compute the likely next reward, and what the next state will be, from only the current state and action, based on previous experience. We can make decisions (predictions) about the likely behaviour of the learner, and its expected rewards, using just the current state data.

11.3.2 Probabilities in Markov Decision Processes

We have now reduced our reinforcement learning problem to learning about Markov Decision Processes. We will only talk about the case where the number of possible states and actions is finite, because reasoning about the infinite case makes your head hurt. There is a very simple example of an MDP in Figure 11.5, showing predictions for your state-of-mind while preparing for an exam, together with the (transition probabilities) for moving between each pair of states shown. This is known as a Markov chain. The diagram can be extended into something called a transition diagram, which shows the dynamics of a finite Markov Decision Process and usually includes information about the rewards.

We can make a transition diagram for our example, based on Figure 11.4. We'll make the situation a little bit more complicated now by adding in the assumption that you are so tired that even though you are in state B and trying to get to state A, there is a small probability that you will actually take the wrong street and end up in either C or D. We'll

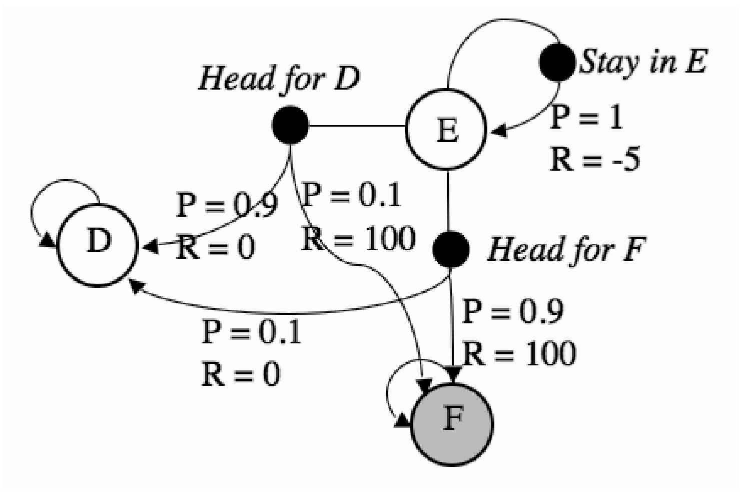

FIGURE 11.6 A small part of the transition diagram for the example. From state E there are three possible actions, and the states in which they end up, together with the rewards, are shown here.

make those probabilities be 0.1 for each extra exit that there is from each state, and we'll assume that you can stand in one place without fear of ending up elsewhere. A very tiny bit of the transition diagram, centred on state E, is shown in Figure 11.6. There are three actions that can be taken in state E (shown by the black circles), with associated probabilities and expected rewards. Learning and using this transition diagram can be seen as the aim of any reinforcement learner.

The Markov Decision Process formalism is a powerful one that can deal with additional uncertainties. For example, it can be extended to deal with the case where the true states are not known, only an observation of the state can be made, which is probabilistically related to the state, and possibly the action. These are known as partially observable Markov Decision Processes (POMDPs), and they are related to the Hidden Markov Models that we will see in Section 16.3. POMDPs are commonly used for robotics, where the sensors of the robots are usually far too inexact and noisy for places the robot visits to be identified with any degree of success. Methods to deal with these problems maintain an estimate of belief of their current state and use that in the reinforcement learning calculations. It is now time to get back to the reinforcement learner and the concept of values.

11.4 VALUES

The reinforcement learner is trying to decide on what action to take in order to maximise the expected reward into the future. This expected reward is known as the value. There are two ways that we can compute a value. We can consider the current state, and average across all of the actions that can be taken, leaving the policy to sort this out for itself (the state-value function, $V(s)$), or we can consider the current state and each possible action that can be taken separately, the action-value function, $Q(s,a)$. In either case we are thinking about what the expected reward would be if we started in state s (where $E(\cdot)$ is the statistical expectation):

$$V(s) = E(r_t|s_t = s) = E\left\{\sum_{i=0}^{\infty} \gamma^i r_{t+i+1}|s_t = s\right\}, \tag{11.5}$$

$$Q(s,a) = E(r_t|s_t = s, a_t = a) = E\left\{\sum_{i=0}^{\infty} \gamma^i r_{t+i+1}|s_t = s, a_t = a\right\}. \tag{11.6}$$

It should be fairly obvious that the second estimate is more accurate in the long run, because we have more information: we know which action we are going to take. However, because of that we need to collect a lot more data, and so it will take a long time to learn. In other words, the action-value function is even more susceptible to the curse of dimensionality than the state-value function. In situations where there are lots of states it will not be possible to store either, and some other method, such as using a parameterised solution space (i.e., having a set of parameters that are controlled by the learner, rather than explicit solutions), will be needed. This is more complicated than we will consider here.

There are now two problems that we need to solve, predicting the value function, and then selecting the optimal policy. We'll think about the second one first. The optimal policy is the one in which the value function is the greatest over all possible states. We label this (not necessarily unique) policy with a star: π^*. The optimal state-value function is then $V^*(s) = \max_\pi V^\pi(s)$ for all possible states s, and the optimal action-value function is $Q^*(s,a) = \max_\pi Q^\pi(s,a)$ for all possible states s and actions a. We can link these two value functions, because the first considers taking the optimal action in each case (since the policy π^* is optimal), while the second considers taking action a this time, and then following the optimal policy from then on. Hence we only need to worry about the current reward and the (discounted) estimate of the future rewards:

$$\begin{aligned} Q^*(s,a) &= E(r_{t+1}) + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) \\ &= E(r_{t+1}) + \gamma V^*(s_{t+1}|s_t = s, a_t = a). \end{aligned} \tag{11.7}$$

Of course, there is no guarantee that we will ever manage to learn the optimal policy. There is bound to be noise and other inaccuracies in the system, and the curse of dimensionality is liable to limit the amount of exploration that we do. However, it will be enough to learn good approximations to the optimal policy. One thing that will work to our advantage is that reinforcement learning operates on-line, in that the learner is exploring the different states as it learns, which means that it is getting more chances to learn about the states that are seen more often, and so will have a better chance of finding the optimal policy for those states.

The question is how you actually update the value function ($V(s)$ or $Q(s,a)$). The idea is to make a look-up table of all the possible states or state-action pairs, and set them all to zero to start with. Then we will use experience to fill them in. Returning to your foreign trip, you wander around until eventually you stumble upon the backpacker's. Gorging yourself on the chips you remember that at the last timestep you were in square E (this remembering is known as a backup). Now you can update the value for E (the reward is $\gamma \times 100$). That is all we do, since your memory is so shot that you can't remember anything else. And there we stop until the next night, when you wake up again and the same thing happens. Except now, you have information about E, although not about any other state. However, when you reach E now, say from D, then you can update the value for D to have reward $\gamma^2 \times 100$. And so it continues, until all of the states (and possibly actions) have values attached.

The obvious problem with this approach is that we have to wait until we reach the goal before we can update the values. Instead, we can use the same trick that we used in Equation (11.7) and use the current reward and the discounted prediction instead, so that the update equation looks like (where μ is the learning rate as usual):

$$V(s_t) \leftarrow V(s_t) + \mu(r_{t+1} + \gamma V(s_{t+1}) - V(s_t)). \tag{11.8}$$

The $Q(s,a)$ version looks very similar, except that we have to include the action information. In both cases we are using the difference between the current and previous estimates, which is why these methods have the name of temporal difference (TD) methods. Suppose that we knew rather more about where we had been. In that case, we could have updated more states when we got to the backpacker's, which would have been rather more efficient. The trouble is that we don't know if those states were useful or not—it might have been chance that we visited them. The answer to this is similar to discounting: we introduce another parameter $0 \leq \lambda \leq 1$ that we apply to reduce the amount that being in that particular state matters. The way this works is known as an eligibility trace, where an eligible state is one that you have visited recently, and it is computed by setting:

$$e_t(s',a') = \begin{cases} 1 & \text{if } s' = s, a' = s \\ \gamma\lambda e_{t-1}(s',a') & \text{otherwise.} \end{cases} \tag{11.9}$$

If $\lambda = 0$ then you only use the current state, which was the algorithm we had above. For $\lambda = 1$ you retain all the knowledge of where you have been. It can be shown that the TD(0) algorithm (i.e., the TD(λ)indexTD(λ) algorithm with $\lambda = 0$) is optimal, in the sense that it converges to the correct value function V^π for the current policy π. There are some provisos on the values of the parameter μ, which are usually satisfied by incrementally reducing the size of μ as learning progresses. The TD(0) algorithm for Q values is also known as the Q-learning algorithm.

The Q-Learning Algorithm

```
• Initialisation
    – set $Q(s,a)$ to small random values for all $s$ and $a$
• Repeat:
    – initialise $s$
    – repeat:
        ∗ select action $a$ using $\epsilon$-greedy or another policy
        ∗ take action $a$ and receive reward $r$
        ∗ sample new state $s'$
        ∗ update $Q(s,a) \leftarrow Q(s,a) + \mu(r + \gamma \max_{a'} Q(s',a') - Q(s,a))$
        ∗ set $s \leftarrow s'$
    – For each step of the current episode
• Until there are no more episodes
```

Note that we can do exactly the same thing for $V(s)$ values instead of $Q(s,a)$ values. There is one thing in this algorithm that is slightly odd, which is in the computation of

$Q(s', a')$. We do not use the policy to find the value of a', but instead choose the one that gives the highest value. This is known as an off-policy decision. Modifying the algorithm to work on-policy is very easy. It gets an interesting name based on the fact that it uses the set of values $(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1})$, which reads 'sarsa':

The Sarsa Algorithm

```
• Initialisation
    – set $Q(s, a)$ to small random values for all $s$ and $a$
• Repeat:
    – initialise $s$
    – choose action $a$ using the current policy
    – repeat:
        ∗ take action $a$ and receive reward $r$
        ∗ sample new state $s'$
        ∗ choose action $a'$ using the current policy
        ∗ update $Q(s, a) \leftarrow Q(s, a) + \mu(r + \gamma Q(s', a') - Q(s, a))$
        ∗ $s \leftarrow s'$, $a \leftarrow a'$
    – for each step of the current episode
• Until there are no more episodes
```

The two algorithms are very similar. They are both bootstrap methods, because they start from poor estimates of the correct answers and iteratively update them as the algorithm progresses. The algorithms work on-line, with the values of Q being updated as soon as r_{t+1} and s_{t+1} are known. In both cases, we are updating the estimates based only on the next state and reward. We could delay our updating for longer, until we knew values of r_{t+n} and s_{t+n}, and then use a TD(λ) algorithm. The only difficulty with this is that there are many different actions a that could be taken between s_t and s_{t+n}.

Once the details of the reward and transition matrices have been sorted out, the implementation of the algorithms doesn't hold many surprises. For example, the central part of the sarsa algorithm using the ϵ-greedy policy can be written in this form:

```
# Stop when the accepting state is reached
while inEpisode:
    r = R[s,a]
    # For this example, new state is the chosen action
    sprime = a

    # epsilon-greedy selection
    if (np.random.rand()<epsilon):
        indices = np.where(t[sprime,:]!=0)
        pick = np.random.randint(np.shape(indices)[1])
        aprime = indices[0][pick]
    else:
        aprime = np.argmax(Q[sprime,:])

    Q[s,a] += mu * (r + gamma*Q[sprime,aprime] - Q[s,a])
    s = sprime
    a = aprime

    # Check if accepting state reached
    if s==5:
      inEpisode = 0
```

11.5 BACK ON HOLIDAY: USING REINFORCEMENT LEARNING

As an example of how to use the reinforcement algorithms we will finish our example of finding the way to the backpacker's by using the ϵ-greedy policy. The specification of the problem is set up in the reward matrix R and transition matrix t, so the first thing to do is to work out how to describe those, neither of which is very difficult since they were given in Section 11.2. It might not be obvious that t is there, but it is shown in Figure 11.4, and can be written out as (where 1 means that there is a link, and 0 means that there is not):

	Next State					
Current State	A	B	C	D	E	F
A	1	1	0	0	0	0
B	1	1	1	1	0	0
C	0	1	1	1	0	1
D	0	1	1	1	1	0
E	0	0	0	1	1	1
F	0	0	1	0	1	1

It is then just a question of running the algorithm for parameter choices of γ, μ, ϵ, and the number of iterations. A run with $\gamma = 0.4$, $\mu = 0.7$, $\epsilon = 0.1$, and 1,000 iterations (with either sarsa or Q-learning produced the following Q matrix:

1.4	16.0	0	0	0	0
6.4	11.0	40.0	16.0	0	0
0	16.0	35.0	16.0	0	100.0
0	16.0	40.0	11.0	40.0	0
0	0	0	16.0	35.0	100.0
0	0	0	0	0	0

Note that NumPy has a useful `np.inf` value, so `-np.inf` can be used as rewards for impossible actions. Some of the 0s can become `-np.inf`s eventually. The question is how to interpret and use this matrix, and the answer is to simply apply the policy at each point, choosing the maximum available Q value for the current state most of the time until you reach the goal state. So from A, the policy will direct you to move to B ($Q = 16$) then on to C ($Q = 40$) and so to F. From D you can go to either C or E ($Q = 40$) and from either of those, directly to F.

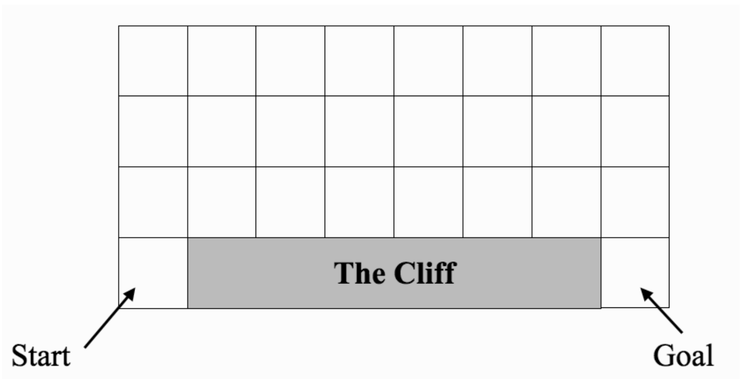

FIGURE 11.7 The example environment.

11.6 THE DIFFERENCE BETWEEN SARSA AND Q-LEARNING

It might not be clear what the difference is between the two algorithms in practice. We're going to consider the little environment that is shown in Figure 11.7, where the agent has to learn a route from the start location on the left to the final location on the right (the example comes from Section 6.5 of Sutton and Barto's book, which is in the readings at the end of the chapter). The reward structure is that every move gets a reward of -1, except for moves that end up on the cliff. These get a reward of -100, and the agent gets put back at the start location. This is clearly an episodic problem, since there is a clear end state.

Both algorithms will start out with no information about the environment, and will therefore explore randomly, using the ϵ-greedy policy. However, over time, the strategies that the two algorithms produce are quite different. The main reason for the difference is that Q-learning always attempts to follow the optimal path, which is the shortest one. This takes it close to the cliff, and the ϵ-greedy part means that inevitably it will sometimes fall over. By way of contrast, the sarsa algorithm will converge to a much safer route that keeps it well away from the cliff, even though it takes longer. The two solutions are shown in Figures 11.8 and 11.9. The sarsa algorithm produces the safe route because it includes information about action selection in its estimates of Q, while Q-learning produces the riskier, but shorter, route. The choice of which is better is up to you, and it depends on how serious the effects of falling off the cliff are.

The reason for the difference between the algorithms is that Q-learning always assumes that the policy will pick the optimal action, and while this is true most of the time, the ϵ-greedy policy does occasionally choose a different action, which can cause problems here. However, the algorithm ignores these dangers because it only focuses on the optimal solution. Sarsa does not take this maximum, and so it will be biased against solutions that take it close to the cliff, because these allow for cases where the agent fell off the cliff, and that therefore have very large negative rewards.

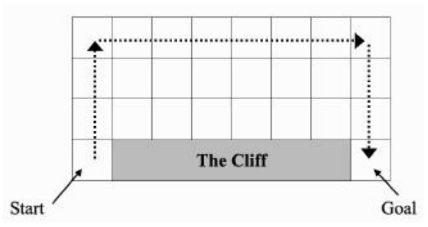

FIGURE 11.8 The sarsa solution is far from optimal, but it is safe.

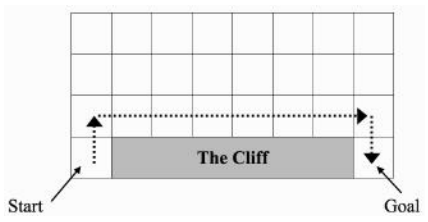

FIGURE 11.9 The Q-learning solution is optimal, but occasionally the random search will tip it over the cliff.

11.7 USES OF REINFORCEMENT LEARNING

Reinforcement learning has been used successfully for many problems, and the results of computer modelling of reinforcement learning have been of great interest to psychologists, as well as computer scientists, because of the close links to biological learning. However, the place where it has been most popular is in intelligent robotics, because of the fact that the robot can be left to attempt to solve the task without human intervention.

For example, reinforcement learning has been used to get robots to learn to clear a room by pushing boxes to the edges. This isn't exactly the most exciting task in the world, but the fact that the robot can learn to do it using reinforcement learning is impressive. Reinforcement learning has been used in other robotic applications, including robots learning to follow each other, travel towards bright lights, and even navigate.

This is not to say that reinforcement learning does not have problems. Since it is, in essence, a search strategy, reinforcement learning suffers from the same difficulties as the search algorithms that we talked about in the last two chapters: it can become stuck in local minima, and if the current search region is effectively flat, then the algorithm does not find any better solution. There are several reports of researchers training robots having the batteries run out before the robot has learnt anything, and even of the researchers giving up and kicking the robot in the right direction to give it a start. In general, reinforcement learning is fairly slow, because it has to build up all of the information through exploration and exploitation in order to find the better solutions. It is also very dependent upon a carefully chosen reward function: get that wrong and the algorithm will do something completely unexpected.

A famous example of reinforcement learning was TD-Gammon, which was produced by Gerald Tesauro. His idea was that reinforcement learning should be very good at learning to play games, because games were clearly episodic—you played until somebody won—and there was a clear reward structure, with a positive reward for winning. There was another benefit, which was that you could set the learner to play against itself. This is actually very important, since the version of TD-Gammon that was actually bundled with the IBM operating system OS/2 Warp had played 1,500,000 games against itself before it stopped improving.

FURTHER READING

A detailed book on reinforcement learning is:

- R.S. Sutton and A.G. Barto. *Reinforcement Learning: An Introduction.* MIT Press, Cambridge, MA, USA, 1998.

An interesting article concerning the use of reinforcement learning is:

- G. Tesauro. Temporal difference learning and TD-gammon. *Communications of the ACM*, 38(3):58–68, 1995.

Alternative treatments are:

- Chapter 13 of T. Mitchell. *Machine Learning.* McGraw-Hill, New York, USA, 1997.
- Chapter 18 of E. Alpaydin. *Introduction to Machine Learning*, 2nd edition, MIT Press, Cambridge, MA, USA, 2009.

PRACTICE QUESTIONS

Problem 11.1 Work through the first few steps of the hill by hand for both sarsa and Q-learning. Then modify the code to run on this example and ensure that they match.

Problem 11.2 Design a Q-learner for playing noughts-and-crosses (also known as Tic-Tac-Toe). Run the algorithm by hand, describing the states, transitions, rewards, and Q-values. Assume that the opponent picks a random (but valid) square for each move. How would your learner change if the opponent played optimally? Would a TD learner behave differently?

Problem 11.3 A robot has 8 range-finding sensors and 2 motors. The range sensors return an integer between 0 and 127 inclusive that represents the distance in centimetres to the nearest object. If the nearest object is further than 127 centimetres away, then 127 is returned. The motors receive an integer input between -100 (full speed backwards) and 100 (full speed forwards).

You want to train the robot to follow the right-hand wall using reinforcement learning. The robot should stay between 15 and 30 centimetres away from the right-hand wall, and if it reaches corners should be able to turn to follow the wall.

Compute the state space, decide if this is a continuous or episodic problem, and then design a suitable reinforcement learner of the problem, considering:

- Any quantisation of the input and output spaces.
- The reward system you choose.
- A description of your chosen learning algorithm.
- Any problems that you anticipate with the system, and what the final result of the learning will be.

Problem 11.4 There are 5 lifts in a 10-storey office building. On each floor there are call buttons for somebody wishing to go up or down, except for the top and bottom floors where there is only 1 call button. When a lift arrives and somebody enters the lift they press the number of the floor on which they wish to stop. Each lift stores the numbers and travels up or down, stopping at each floor that is requested.

Calculate the state and action spaces for the system, and then describe a suitable reinforcement learner for this system. You need to devise a reward function as well as describe the learning method that you believe to be most appropriate. Should the system use delayed rewards? A good reinforcement learning system provides a very effective algorithm for this problem (as compared to standard naïve methods for lift scheduling). Explain why this could be the case, and give possible problems with using a reinforcement learner.

Problem 11.5 It is possible to write a learning Connect-4 player. In case you don't remember Connect-4, the game is played on a grid board of 7×6. Two players take it in turns to drop tokens into the grid where they fill the lowest available spot in the chosen column. The aim is to get four of your coloured tokens in a row. In case that doesn't make sense, or just because you are feeling nostalgic, there are plenty of versions of the game on the Internet.

The state space of Connect-4 is not easy to think about. There is 1 state with no counters on it, 7 states with 1 counter in them (assuming that the same colour counter always starts): one state for the counter being in each row, and 7 again for 2 counters being on the board. However, from there the number of states mushrooms. In the case where the game is a draw, so that all of the squares are full, there is something less than $2^{7\times 6} = 2^{42}$ states. I say something less because this counts all the cases that include a line of 4, and also ignores the fact that there are only 21 counters of each colour. The fact remains that the state space is immense, so it is probably going to take a long time to learn.

However, programming the game is relatively simple. There are two absorbing states: when the board is full, and when somebody wins. In either of these cases a reward is given. So you will have to decide on rewards, and write some code that detects when one or the other state has happened. The choice that is made at each turn is simply which column to add the new counter in, so there are only seven possible actions. You need to represent the board, for which I'd recommend a 2D array with 0 meaning empty, 1 meaning contains a red counter, and 2 meaning contains a yellow counter. This should make it easy to detect the absorbing states.

Having set up that lot, you need to make a number of modifications to the Q-learning code. Firstly, you are not going to pass in transition and reward matrices, since making them would be crazy. You are probably going to give a reward of 0 to every move except a win and a loss, so change the code to present those rewards. You then need to change the ϵ-greedy search strategy to simply pick a random (but not full) column, rather than look at the transition matrix. Then that's it; set it running (and be prepared to wait for a very long time. I trained the algorithm for 20,000 games against a purely random player, and at the end of that the Q-learner was winning about 80% of the games).

CHAPTER 12

Learning with Trees

We are now going to consider a rather different approach to machine learning, starting with one of the most common and powerful data structures in the whole of computer science: the binary tree. The computational cost of making the tree is fairly low, but the cost of using it is even lower: $\mathcal{O}(\log N)$, where N is the number of datapoints. This is important for machine learning, since querying the trained algorithm should be as fast as possible since it happens more often, and the result is often wanted immediately. This is sufficient to make trees seem attractive for machine learning. However, they do have other benefits, such as the fact that they are easy to understand (following a tree to get a classification answer is transparent, which makes people trust it more than getting an answer from a 'black box' neural network).

For these reasons, classification by decision trees has grown in popularity over recent years. You are very likely to have been subjected to decision trees if you've ever phoned a helpline, for example for computer faults. The phone operators are guided through the decision tree by your answers to their questions.

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

In terms of optimisation and search, decision trees use a greedy heuristic to perform search, evaluating the possible options at the current stage of learning and making the one that seems optimal at that point. This works well a surprisingly large amount of the time.

12.1 USING DECISION TREES

As a student it can be difficult to decide what to do in the evening. There are four things that you actually quite enjoy doing, or have to do: going to the pub, watching TV, going to a party, or even (gasp) studying. The choice is sometimes made for you—if you have an assignment due the next day, then you need to study, if you are feeling lazy then the pub isn't for you, and if there isn't a party then you can't go to it. You are looking for a nice algorithm that will let you decide what to do each evening without having to think about it every night. Figure 12.1 provides just such an algorithm.

Each evening you start at the top (root) of the tree and check whether any of your friends know about a party that night. If there is one, then you need to go, regardless. Only if there is not a party do you worry about whether or not you have an assignment deadline coming up. If there is a crucial deadline, then you have to study, but if there is nothing that is urgent for the next few days, you think about how you feel. A sudden burst of energy

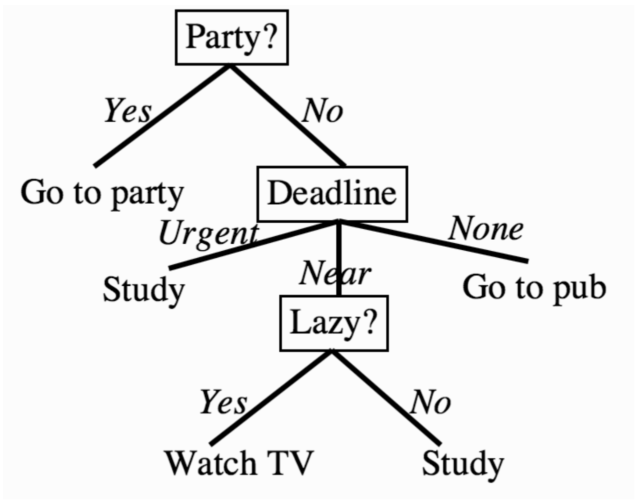

FIGURE 12.1 A simple decision tree to decide how you will spend the evening.

might make you study, but otherwise you'll be slumped in front of the TV indulging your secret love of Shortland Street (or other soap opera of your choice) rather than studying. Of course, near the start of the semester when there are no assignments to do, and you are feeling rich, you'll be in the pub.

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

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

That's all that there is to using the decision tree. Compare it to the previous use of this data, with the Naïve Bayes Classifier in Section 2.3.2. The far more interesting part is how to construct the tree from data, and that is the focus of the next section.

12.2 CONSTRUCTING DECISION TREES

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

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

12.2.1 Quick Aside: Entropy in Information Theory

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

$$\text{Entropy}(p) = -\sum_i p_i \log_2 p_i, \tag{12.1}$$

where the logarithm is base 2 because we are imagining that we encode everything using binary digits (bits), and we define $0 \log 0 = 0$. A graph of the entropy is given in Figure 12.2. Suppose that we have a set of positive and negative examples of some feature (where the feature can only take 2 values: positive and negative). If all of the examples are positive, then we don't get any extra information from knowing the value of the feature for any particular example, since whatever the value of the feature, the example will be positive. Thus, the entropy of that feature is 0. However, if the feature separates the examples into 50% positive and 50% negative, then the amount of entropy is at a maximum, and knowing about that feature is very useful to us. The basic concept is that it tells us how much *extra* information we would get from knowing the value of that feature. A function for computing the entropy is very simple, as here:

```
def calc_entropy(p):

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

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

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

12.2.2 ID3

Now that we have a suitable measure for choosing which feature to choose next, entropy, we just have to work out how to apply it. The important idea is to work out how much the entropy of the whole training set would decrease if we choose each particular feature for the next classification step. This is known as the information gain, and it is defined as

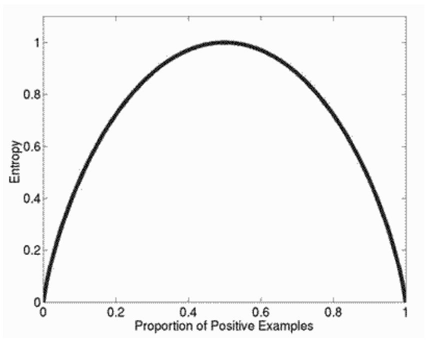

FIGURE 12.2 A graph of entropy, detailing how much information is available from finding out another piece of information given what you already know.

the entropy of the whole set minus the entropy when a particular feature is chosen. This is defined by (where S is the set of examples, F is a possible feature out of the set of all possible ones, and $|S_f|$ is a count of the number of members of S that have value f for feature F):

$$\text{Gain}(S, F) = \text{Entropy}(S) - \sum_{f \in values(F)} \frac{|S_f|}{|S|} \text{Entropy}(S_f). \tag{12.2}$$

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

$$\begin{aligned}
\text{Entropy}(S) &= -p_\oplus \log_2 p_\oplus - p_\ominus \log_2 p_\ominus \\
&= -\frac{1}{4} \log_2 \frac{1}{4} - \frac{3}{4} \log_2 \frac{3}{4} \\
&= 0.5 + 0.311 = 0.811.
\end{aligned} \tag{12.3}$$

The function Entropy(S_f) is similar, but only computed with the subset of data where feature F has values f.

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

We now want to compute the information gain of F, so we now need to compute each of the values inside the summation in Equation (12.2), $\frac{|S_f|}{|S|}\text{Entropy}(S)$ (in our example, the features are ‘Deadline’, ‘Party’, and ‘Lazy’):

$$\begin{aligned}
\frac{|S_{f_1}|}{|S|}\text{Entropy}(S_{f_1}) &= \frac{1}{4}\times\left(-\frac{0}{1}\log_2\frac{0}{1}-\frac{1}{1}\log_2\frac{1}{1}\right) \\
&= 0 \qquad (12.4) \\
\frac{|S_{f_2}|}{|S|}\text{Entropy}(S_{f_2}) &= \frac{2}{4}\times\left(-\frac{1}{2}\log_2\frac{1}{2}-\frac{1}{2}\log_2\frac{1}{2}\right) \\
&= \frac{1}{2} \qquad (12.5) \\
\frac{|S_{f_3}|}{|S|}\text{Entropy}(S_{f_3}) &= \frac{1}{4}\times\left(-\frac{0}{1}\log_2\frac{0}{1}-\frac{1}{1}\log_2\frac{1}{1}\right) \\
&= 0 \qquad (12.6)
\end{aligned}$$

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

$$\text{Gain}(S,F) = 0.811 - (0 + 0.5 + 0) = 0.311. \qquad (12.7)$$

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

```
def calc_info_gain(data,classes,feature):
    gain = 0
    nData = len(data)
    # List the values that feature can take
    values = []
    for datapoint in data:
        if datapoint[feature] not in values:
            values.append(datapoint[feature])

    featureCounts = np.zeros(len(values))
    entropy = np.zeros(len(values))
    valueIndex = 0
    # Find where those values appear in data[feature] and the corresponding ⤸
    class
    for value in values:
        dataIndex = 0
        newClasses = []
        for datapoint in data:
            if datapoint[feature]==value:
                featureCounts[valueIndex]+=1
                newClasses.append(classes[dataIndex])
            dataIndex += 1

        # Get the values in newClasses
        classValues = []
        for aclass in newClasses:
            if classValues.count(aclass)==0:
                classValues.append(aclass)
        classCounts = np.zeros(len(classValues))
        classIndex = 0
        for classValue in classValues:
            for aclass in newClasses:
                if aclass == classValue:
                    classCounts[classIndex]+=1
            classIndex += 1

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

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

The ID3 Algorithm

- If all examples have the same label:
  - return a leaf with that label
- Else if there are no features left to test:
  - return a leaf with the most common label
- Else:
  - choose the feature $\hat{F}$ that maximises the information gain of S to be the next node using Equation (12.2)
  - add a branch from the node for each possible value f in $\hat{F}$
  - for each branch:
    - calculate S_f by removing $\hat{F}$ from the set of features
    - recursively call the algorithm with S_f, to compute the gain relative to the current set of examples

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

12.2.3 Implementing Trees and Graphs in Python

Trees are really just a restricted version of graphs, since they both consist of nodes and edges between the nodes. Graphs are a very useful data structure in many different areas of computer science. There are two reasonable ways to represent a graph computationally. One is as an $N \times N$ matrix, where N is the number of nodes in the network. Each element of the matrix is a 1 if there is a link between the two nodes, and a 0 otherwise. The benefit of this approach is that it is easy to give weights to the links by changing the 1s to the values of the weights. The alternative is to store a list of nodes, following each by a list of nodes that it is linked to. Both are fairly natural in Python, with the second making use of the dictionary, a basic data structure that we have not used much, except for very simply in the decision tree (Chapter 12) that consists of a set of keys and values. For a graph, the key to each dictionary entry is the name of the node, and its value is a list of the nodes that it is connected to, as in this example:

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

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

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

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

12.2.4 Implementation of the Decision Tree

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

```
def make_tree(data,classes,featureNames):
    # Various initialisations suppressed
    default = classes[np.argmax(frequency)]
    if nData==0 or nFeatures == 0:
        # Have reached an empty branch
        return default
    elif classes.count(classes[0]) == nData:
        # Only 1 class remains
        return classes[0]
    else:
        # Choose which feature is best
        gain = np.zeros(nFeatures)
        for feature in range(nFeatures):
            g = calc_info_gain(data,classes,feature)
            gain[feature] = totalEntropy - g
        bestFeature = np.argmax(gain)
        tree = {featureNames[bestFeature]:{}}
        # Find the possible feature values
        for value in values:
            # Find the datapoints with each feature value
            for datapoint in data:
                if datapoint[bestFeature]==value:
                        if bestFeature==0:
                            datapoint = datapoint[1:]
                            newNames = featureNames[1:]
                        elif bestFeature==nFeatures:
                            datapoint = datapoint[:-1]
                            newNames = featureNames[:-1]
                        else:
                            datapoint = datapoint[:bestFeature]
                            datapoint.extend(datapoint[bestFeature+1:])
                            newNames = featureNames[:bestFeature]
                            newNames.extend(featureNames[bestFeature+1:])
                        newData.append(datapoint)
                        newClasses.append(classes[index])
                index += 1
            # Now recurse to the next level
            subtree = make_tree(newData,newClasses,newNames)
            # And on returning, add the subtree on to the tree
            tree[featureNames[bestFeature]][value] = subtree
        return tree
```

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

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

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

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

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

12.2.5 Dealing with Continuous Variables

One thing that we have not yet discussed is how to deal with continuous variables, we have only considered those with discrete sets of feature values. The simplest solution is to discretise the continuous variable. However, it is also possible to leave it continuous and modify the algorithm. For a continuous variable there is not just one place to split it: the variable can be broken between any pair of datapoints, as shown in Figure 12.3. It can, of course, be split in any of the infinite locations along the line as well, but they are no different to this smaller set of locations. Even this smaller set makes the algorithm more expensive for continuous variables than it is for discrete ones, since as well as calculating the information gain of each variable to pick the best one, the information gain of many points within each variable has to be computed. In general, only one split is made to a continuous

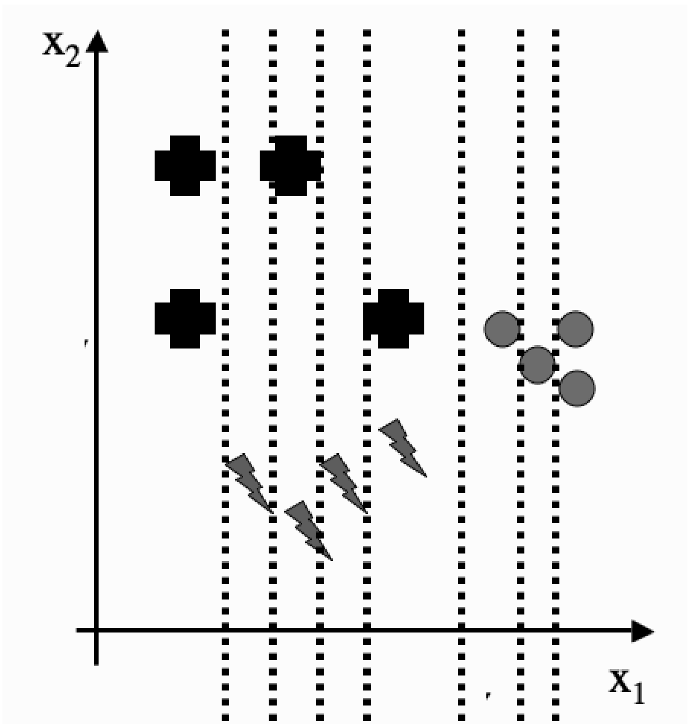

FIGURE 12.3 Possible places to split the variable x_1, between each of the datapoints as the feature value increases.

variable, rather than allowing for threeway or higher splits, although these can be done if necessary.

The trees that these algorithms make are all univariate trees, because they pick one feature (dimension) at a time and split according to that one. There are also algorithms that make multivariate trees by picking combinations of features. This can make for considerably smaller trees if it is possible to find straight lines that separate the data well, but are not parallel to any axis. However, univariate trees are simpler and tend to get good results, so we won't consider multivariate trees any further. This fact that one feature is chosen at a time provides another useful way to visualise what the decision tree is doing. Figure 12.4 shows the idea. Given a dataset that contains three classes, the algorithm picks a feature and value for that feature to split the remaining data into two. The final tree that results from this is shown in Figure 12.5.

12.2.6 Computational Complexity

The computational cost of constructing binary trees is well known for the general case, being $\mathcal{O}(N \log N)$ for construction and $\mathcal{O}(\log N)$ for returning a particular leaf, where N is the number of nodes. However, these results are for balanced binary trees, and decision trees are often not balanced; while the information measures attempt to keep the tree balanced by finding splits that separate the data into two even parts (since that will have the largest entropy), there is no guarantee of this. Nor are they necessarily binary, especially for ID3 and C4.5, as our example shows.

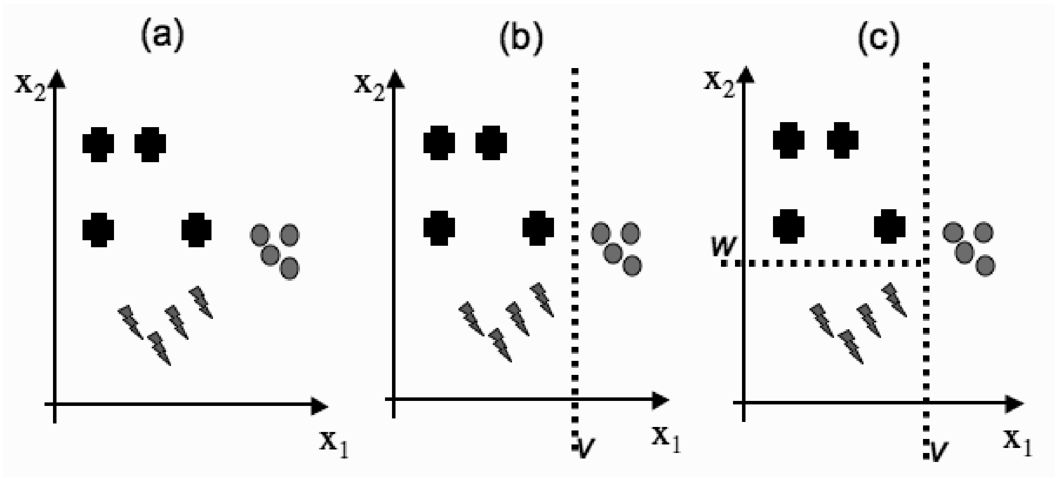

FIGURE 12.4 The effect of decision tree choices. The two-dimensional dataset shown in (a) is split first by choosing feature x_1 (b) and then x_2, (c) which separates out the three classes. The final tree is shown in Figure 12.5.

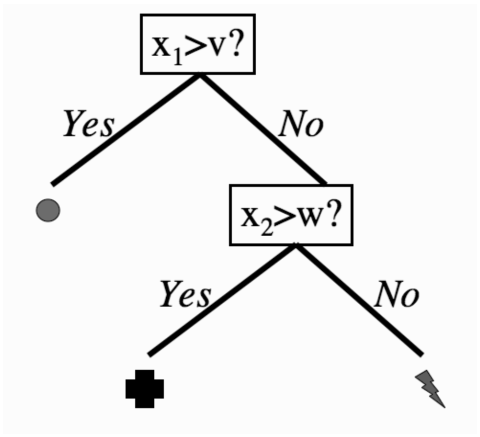

FIGURE 12.5 The final tree created by the splits in Figure 12.4.

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

12.3 CLASSIFICATION AND REGRESSION TREES (CART)

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

12.3.1 Gini Impurity

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

$$G_k = \sum_{i=1}^{c} \sum_{j \neq i} N(i)N(j), \tag{12.8}$$

where c is the number of classes. In fact, you can reduce the algorithmic effort required by noticing that $\sum_i N(i) = 1$ (since there has to be some output class) and so $\sum_{j \neq i} N(j) = 1 - N(i)$. Then Equation (12.8) is equivalent to:

$$G_k = 1 - \sum_{i=1}^{c} N(i)^2. \tag{12.9}$$

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

The information measure can be changed in another way, which is to add a weight to the misclassifications. The idea is to consider the cost of misclassifying an instance of class i as class j (which we will call the risk in Section 2.3.1) and add a weight that says how important each datapoint is. It is typically labelled as λ_{ij} and is presented as a matrix, with element λ_{ij} representing the cost of misclassifying i as j. Using it is simple, modifying the Gini impurity (Equation (12.8)) to be:

$$G_i = \sum_{j \neq i} \lambda_{ij} N(i) N(j). \tag{12.10}$$

We will see in Section 13.1 that there is another benefit to using these weights, which is to successively improve the classification ability by putting higher weight on datapoints that the algorithm is getting wrong.

12.3.2 Regression in Trees

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

12.4 CLASSIFICATION EXAMPLE

We'll work through an example using ID3 in this section. The data that we'll use will be a continuation of the one we started the chapter with, about what to do in the evening.

When we want to construct the decision tree to decide what to do in the evening, we start by listing everything that we've done for the past few days to get a suitable dataset (here, the last ten days):

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

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

$$\begin{aligned}
\text{Entropy}(S) &= -p_{\text{party}} \log_2 p_{\text{party}} - p_{\text{study}} \log_2 p_{\text{study}} \\
&- p_{\text{pub}} \log_2 p_{\text{pub}} - p_{\text{TV}} \log_2 p_{\text{TV}} \\
&= -\frac{5}{10} \log_2 \frac{5}{10} - \frac{3}{10} \log_2 \frac{3}{10} - \frac{1}{10} \log_2 \frac{1}{10} - \frac{1}{10} \log_2 \frac{1}{10} \\
&= 0.5 + 0.5211 + 0.3322 + 0.3322 = 1.6855 \qquad (12.11)
\end{aligned}$$

and then find which feature has the maximal information gain:

$$\begin{aligned}
\text{Gain}(S, \text{Deadline}) &= 1.6855 - \frac{|S_{\text{urgent}}|}{10} \text{Entropy}(S_{\text{urgent}}) \\
&- \frac{|S_{\text{near}}|}{10} \text{Entropy}(S_{\text{near}}) - \frac{|S_{\text{none}}|}{10} \text{Entropy}(S_{\text{none}}) \\
&= 1.6855 - \frac{3}{10} \left(-\frac{2}{3} \log_2 \frac{2}{3} - \frac{1}{3} \log_2 \frac{1}{3} \right) \\
&- \frac{4}{10} \left(-\frac{2}{4} \log_2 \frac{2}{4} - \frac{1}{4} \log_2 \frac{1}{4} - \frac{1}{4} \log_2 \frac{1}{4} \right) \\
&- \frac{3}{10} \left(-\frac{1}{3} \log_2 \frac{1}{3} - \frac{2}{3} \log_2 \frac{2}{3} \right) \\
&= 1.6855 - 0.2755 - 0.6 - 0.2755 \\
&= 0.5345 \qquad (12.12)
\end{aligned}$$

$$\begin{aligned}
\text{Gain}(S, \text{Party}) &= 1.6855 - \frac{5}{10} \left(-\frac{5}{5} \log_2 \frac{5}{5} \right) \\
&- \frac{5}{10} \left(-\frac{3}{5} \log_2 \frac{3}{5} - \frac{1}{5} \log_2 \frac{1}{5} - \frac{1}{5} \log_2 \frac{1}{5} \right) \\
&= 1.6855 - 0 - 0.6855 \\
&= 1.0 \qquad (12.13)
\end{aligned}$$

$$\begin{aligned}
\text{Gain}(S, \text{Lazy}) &= 1.6855 - \frac{6}{10} \left(-\frac{3}{6} \log_2 \frac{3}{6} - \frac{1}{6} \log_2 \frac{1}{6} - \frac{1}{6} \log_2 \frac{1}{6} - \frac{1}{6} \log_2 \frac{1}{6} \right) \\
&- \frac{4}{10} \left(-\frac{2}{4} \log_2 \frac{2}{4} - \frac{2}{4} \log_2 \frac{2}{4} \right) \\
&= 1.6855 - 1.0755 - 0.4 \\
&= 0.21 \qquad (12.14)
\end{aligned}$$

Therefore, the root node will be the party feature, which has two feature values ('yes' and 'no'), so it will have two branches coming out of it (see Figure 12.6). When we look at the 'yes' branch, we see that in all five cases where there was a party we went to it, so we just put a leaf node there, saying 'party'. For the 'no' branch, out of the five cases there are three different outcomes, so now we need to choose another feature. The five cases we are looking at are:

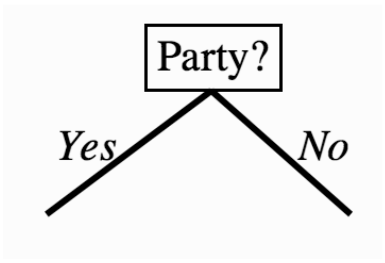

FIGURE 12.6 The decision tree after one step of the algorithm.

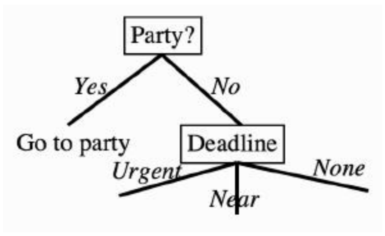

FIGURE 12.7 The tree after another step.

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

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

$$\begin{aligned}
\text{Gain}(S, \text{Deadline}) &= 1.371 - \frac{2}{5}\left(-\frac{2}{2}\log_2\frac{2}{2}\right) \\
&- \frac{2}{5}\left(-\frac{1}{2}\log_2\frac{1}{2} - \frac{1}{2}\log_2\frac{1}{2}\right) - \frac{1}{5}\left(-\frac{1}{1}\log_2\frac{1}{1}\right) \\
&= 1.371 - 0 - 0.4 - 0 \\
&= 0.971 \qquad (12.15) \\
\text{Gain}(S, \text{Lazy}) &= 1.371 - \frac{4}{5}\left(-\frac{2}{4}\log_2\frac{2}{4} - \frac{1}{4}\log_2\frac{1}{4} - \frac{1}{4}\log_2\frac{1}{4}\right) \\
&- \frac{1}{5}\left(-\frac{1}{1}\log_2\frac{1}{1}\right) \\
&= 1.371 - 1.2 - 0 \\
&= 0.1710 \qquad (12.16)
\end{aligned}$$

This leads to the tree shown in Figure 12.7. From this point it is relatively simple to complete the tree, leading to the one that was shown in Figure 12.1.

FURTHER READING

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

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

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

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

The original paper that started the field is:

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

A book that covers information theory and machine learning is:

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

Other machine learning textbooks that cover decision trees include:

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

PRACTICE QUESTIONS

Problem 12.1 Suppose that the probability of five events are P(first) = 0.5, and P(second) = P(third) = P(fourth) = P(fifth) = 0.125. Calculate the entropy. Write down in words what this means.

Problem 12.2 Make a decision tree that computes the logical AND function. How does it compare to the Perceptron solution?

Problem 12.3 Turn this politically incorrect data from Quinlan into a decision tree to classify which attributes make a person attractive, and then extract the rules.

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

Problem 12.4 When you arrive at the pub, your five friends already have their drinks on the table. Jim has a job and buys the round half of the time. Jane buys the round a quarter of the time, and Sarah and Simon buy a round one eighth of the time. John hasn't got his wallet out since you met him three years ago.

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

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

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

Problem 12.5 Use the naïve Bayes classifier from Section 2.3.2 on the datasets that you used for the decision tree (this will involve some effort in turning the textual data into probabilities) and compare the results.

Problem 12.6 The `CPU` dataset in the UCI repository is a very good regression problem for a decision tree. You will need to modify the decision tree code so that it does regression, as discussed in Section 12.3.2. You will also have to work out the Gini impurity for multiple classes.

Problem 12.7 Modify the implementation to deal with continuous variables, as discussed in Section 12.2.5.

Problem 12.8 The misclassification impurity is:

$$N(i) = 1 - \max_j P(w_j). \qquad (12.17)$$

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

CHAPTER 13

Decision by Committee: Ensemble Learning

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

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

Figure 13.1 shows the basic idea of ensemble learning, as these methods are collectively called. Given a relatively simple binary classification problem and some learner that puts an ellipse around a subset of the data, combining the ellipses can provide a considerably more complex decision boundary.

There are then only a couple of questions to ask: which learners should we use, how should we ensure that they learn different things, and how should we combine their results? The methods that we are investigating in this chapter can use any classifier at all. Although in general they only use one type of classifier at a time, they do not have to. A common choice of classifier is the decision tree (see Chapter 12).

Ensuring that the learners see different things can be performed in different ways, and it is the primary difference between the algorithms that we shall see. However, it can also come about naturally depending upon the application area. Suppose that you have lots and lots of data. In that case you could simply randomly partition the data and give different sets of data to different classifiers. Even here there are choices: do you make the partitions separate, or include overlaps? If there is no overlap, then it could be difficult to work out how to combine the classifiers, or it might be very simple: if your doctor always asks for opinions from two colleagues, one specialising in heart problems and one in sports injuries,

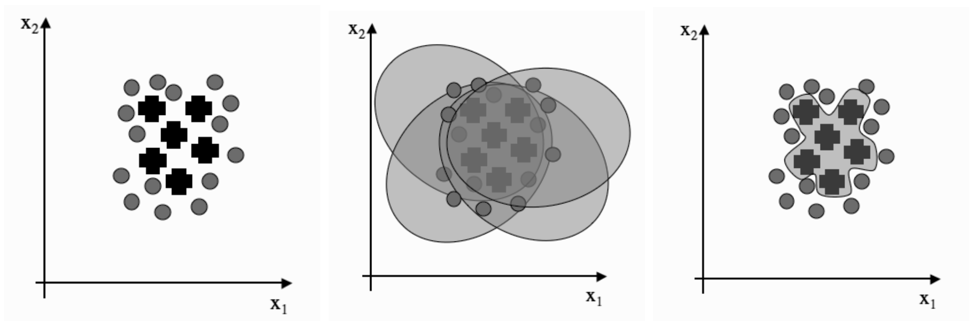

FIGURE 13.1 By combining lots of simple classifiers (here that simply put an elliptical decision boundary onto the data), the decision boundary can be made much more complicated, enabling the difficult separation of the pluses from the circles.

then upon discovering that your leg started hurting after you went for a run she would likely accord more weight to the diagnosis of the sports injury expert.

Interestingly, ensemble methods do very well when there is very little data as well as when there is too much. To see why, think cross-validation (Section 2.2.2). We used cross-validation when there was not enough data to go around, and trained lots of neural networks on different subsets of the data. Then we threw away most of them. With an ensemble method we keep them all, and combine their results in some way. One very simple way to combine the results is to use majority voting — if it's good enough for electing governments in elections, it's good enough for machine learning. Majority voting has the interesting property that for binary classification, the combined classifier will only get the answer wrong if more than half of the classifiers were wrong. Hopefully, this isn't going to happen too often (although you might be able to think of government elections where this has been the case in your view). There are alternative ways to combine the results, as we'll discuss. These things will become clearer as we look at the algorithms, so let's get started.

13.1 BOOSTING

At first sight the claim of the most popular ensemble method, boosting, seems amazing. If we take a collection of very poor (weak in the jargon) learners, each performing only just better than chance, then by putting them together it is possible to make an ensemble learner that can perform arbitrarily well. So we just need lots of low-quality learners, and a way to put them together usefully, and we can make a learner that will do very well.

The principal algorithm of boosting is named AdaBoost, and is described in Section 13.1.1. The algorithm was first described in the mid-1990s by Freund and Shapiro, and while it has had many variations derived from it, the principal algorithm is still one of the most widely used. The algorithm was proposed as an improvement on the original 1990 boosting algorithm, which was rather data hungry. In that algorithm, the training set was split into three. A classifier was trained on the first third, and then tested on the second third. All of the data that was misclassified during that testing was used to form a new dataset, along with an equally sized random selection of the data that was correctly classified. A second classifier was trained on this new dataset, and then both of the classifiers were tested on the final third of the dataset. If they both produced the same output, then that datapoint was ignored, otherwise the datapoint was added to yet another new

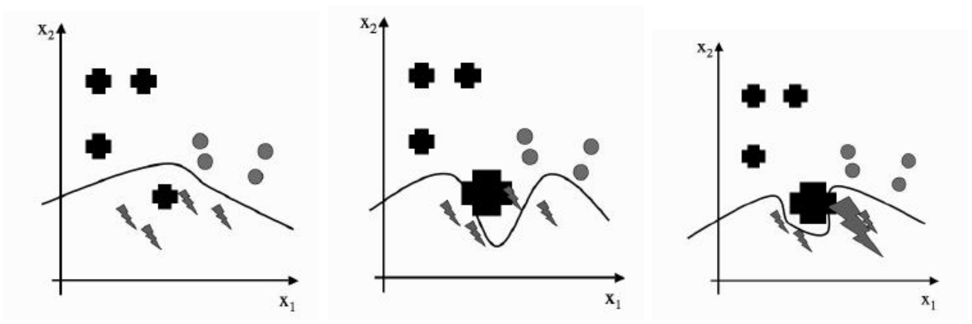

FIGURE 13.2 As points are misclassified, so their weights increase in boosting (shown by the datapoint getting larger), which makes the importance of those datapoints increase, making the classifiers pay more attention to them.

dataset, which formed the training set for a third classifer. Rather than looking further at this version, we will look at the more common algorithm.

13.1.1 AdaBoost

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

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

Figure 13.2 shows the effect of weighting incorrectly classified examples as training proceeds, with the size of each datapoint being a measure of its importance. As an algorithm this looks like (where $I(y_n \neq h_t(x_n))$ is an indicator function that returns 1 if the target and output are not equal, and 0 if they are):

AdaBoost Algorithm

```
• Initialise all weights to $1/N$, where $N$ is the number of datapoints
• While $0 < \epsilon_t < \frac{1}{2}$ (and $t < T$, some maximum number of iterations):
    – train classifier on $\{S, w^{(t)}\}$, getting hypotheses $h_t(x_n)$ for datapoints $x_n$
    – compute training error $\epsilon_t = \sum_{n=1}^{N} w_n^{(t)} I(y_n \neq h_t(x_n))$
    – set $\alpha_t = \log\left(\frac{1-\epsilon_t}{\epsilon_t}\right)$
    – update weights using:
          $w_n^{(t+1)} = w_n^{(t)} \exp(\alpha_t I(y_n \neq h_t(x_n))/Z_t,$    (13.1)
      where $Z_t$ is a normalisation constant
• Output $f(x) = \text{sign}\left(\sum_{t=1}^{T} \alpha_t h_t(x)\right)$
```

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

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

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

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

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

Most of the work of the algorithm is done by the classification algorithm, which is given new weights at each iteration. In this respect, boosting is not quite a stand-alone algorithm: the classifiers need to consider the weights when they perform their classifications. It is not always obvious how to do this for a particular classifier, but we have seen methods of doing it for a few classifiers. For the decision tree we saw a method in Section 12.3.1,

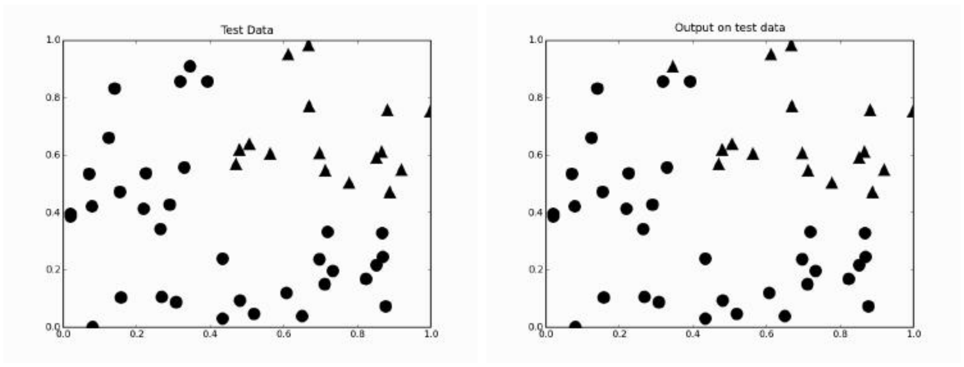

FIGURE 13.3 Boosting learns this simple dataset very successfully, producing an ensemble classifier that is rather more complicated than the simple horizontal or vertical line classifier that the algorithm boosts. On the independent test set shown here, the algorithm gets only 1 datapoint wrong, and that is one that is coincidentally close to one that was misclassified to simulate noise in the training data.

when we looked at the Gini impurity. There, we allowed for a $\boldsymbol{\lambda}$ matrix that encoded the risks associated with misclassification, and these are a perfect place in which to introduce weights. Modification of the decision tree algorithm to deal with these weights is suggested as an exercise for this chapter. A similar argument can be used for the Bayes' classifier; this was discussed in Section 2.3.1.

As a very simple example showing how boosting works, a very simple classifier was created that can only separate data by fitting one either horizontal or vertical line, with it choosing which to fit at the current iteration at random. A two-dimensional dataset was created with data in the top right-hand corner being in one class, and the rest in another, plus a couple of the datapoints were randomly mislabelled to simulate noise. Clearly, this dataset cannot be separated by a single horizontal or vertical decision boundary. However, Figure 13.3 shows the output of the classifier on an independent test set, where the algorithm gets only one datapoint wrong, and that is one that is coincidentally close to one of the 'noisy' datapoints in the training data. Figure 13.4 shows the training data, the error curve on both the training and testing sets, and the first few iterations of the classifier, which can only put in one horizontal or linear classification line.

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

$$G_t(\alpha) = \sum_{n=1}^{N} \exp\left(-y_n(\alpha h_t(x_n) + f_{t-1}(x_n))\right), \tag{13.2}$$

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

$$f_{t-1}(x_n) = \sum_{\tau=0}^{t-1} \alpha_\tau h_\tau(x_n). \tag{13.3}$$

Exponential loss functions are well behaved and robust to outliers. The weights $w^{(t)}$

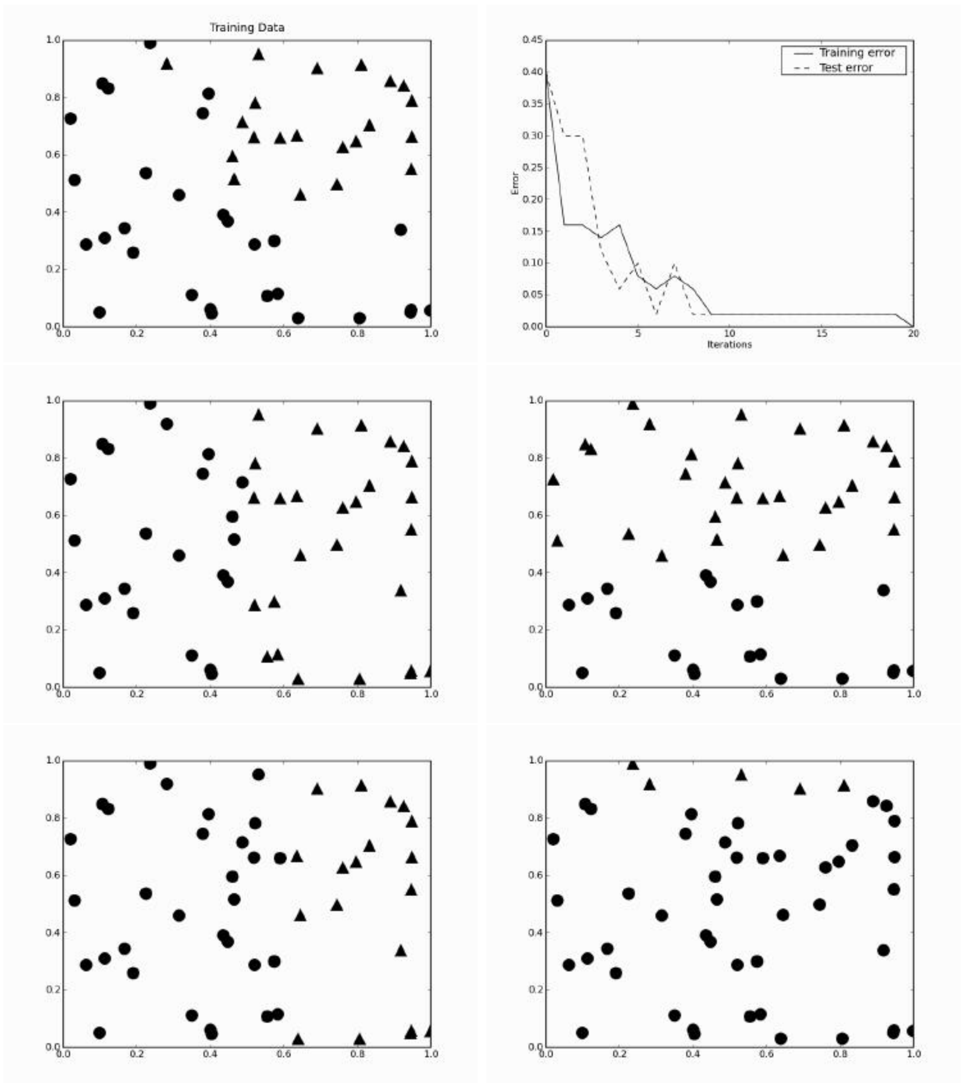

FIGURE 13.4 *Top:* the training data and the error curve. *Middle and bottom:* The first few iterations of the classifier; each plot shows the output of one of the weak classifiers that are boosted by the algorithm.

in the algorithm are nothing more than the second term in Equation (13.2), which can therefore be rewritten as:

$$G_t(\alpha) = \sum_{n=1}^{N} w^{(t)} \exp\left(-y_n \alpha h_t(x_n)\right). \tag{13.4}$$

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

AdaBoost can be modified to perform regression rather than classification (known as real adaboost, or sometimes adaboost.R). There is another variant on boosting (also called AdaBoost, confusingly) that uses the weights to sample from the full dataset, training on a sample of the data rather than the full weighted set, with more difficult examples more likely to be in the training sample. This is more in line with the original boosting algorithm, and is obviously faster, since each training run has fewer data to learn about.

13.1.2 Stumping

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

13.2 BAGGING

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

Bootstrap sampling seems like a very strange thing to do. We've taken a perfectly good dataset, mucked it up by sampling from it, which might be good if we had made a smaller dataset (since it would be faster), but we still ended up with a dataset the same size. Worse, we've done it lots of times. Surely this is just a way to burn up computer time without gaining anything. The benefit of it is that we will get lots of learners that perform slightly differently, which is exactly what we want for an ensemble method. Another benefit is that estimates of the accuracy of the classification function can be made without complicated analytic work, by throwing computer resources at the problem (technically, bagging is a variance reducing algorithm; the meaning of this will become clearer when we talk about bias and variance in Section 2.5). This is a standard technique in modern statistics; we'll see another example in Chapter 15 when we look at Markov Chain Monte Carlo methods. It is sufficiently common to have inspired the comment that "statistics is defined as the discipline where those that think don't count and those that count don't think."

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

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

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

The example consists of taking the party data that was used in Section 12.4 to demonstrate the decision tree, and restricting the trees to stumps, so that they can make a classification based on just one variable. The output of a decision tree that uses the whole dataset for this is not surprising: it takes the two largest classes, and separates them. However, using just stumps of trees and 20 samples, bagging can separate the data perfectly, as this output shows:

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

13.2.1 Subagging

For some reason, ensemble methods often have good names, such as boosting and bagging (and we will see my choice for best-named, bragging, in Section 13.4). However, the method of subagging wins the prize for the oddest sounding word. It is a combination of 'subsample' and 'bagging,' and it is the fairly obvious idea that you don't need to produce samples that are the same size as the original data. If you make smaller datasets, then it makes sense to sample without replacement, but otherwise the implementation is only very slightly different from the bagging one, except that in NumPy you use `np.random.shuffle()` to produce the samples. It is common to use a dataset size that is half that of the original data, and the results of this can often be comparable to a full bagging simulation.

13.3 RANDOM FORESTS

If there is one method in machine learning that has grown in popularity over the last few years, then it is the idea of random forests. The concept has been around for longer than that, with several different people inventing variations, but the name that is most strongly attached to it is that of Breiman, who also described the CART algorithm that was discussed in Section 12.2, and also gave bagging its name.

The idea is largely that if one tree is good, then many trees (a forest) should be better, provided that there is enough variety between them. The most interesting thing about a random forest is the ways that it creates randomness from a standard dataset. The first of the methods that it uses is the one that we have just seen: bagging. If we wish to create a forest then we can make the trees different by training them on slightly different data, so we take bootstrap samples from the dataset for each tree. However, this isn't enough randomness yet. The other obvious place where it is possible to add randomness is to limit the choices that the decision tree can make. At each node, a random subset of the features is given to the tree, and it can only pick from that subset rather than from the whole set.

As well as increasing the randomness in the training of each tree, it also speeds up the training, since there are fewer features to search over at each stage. Of course, it does introduce a new parameter (how many features to consider), but the random forest does not seem to be very sensitive to this parameter; in practice, a subset size that is the square root of the number of features seems to be common. The effect of these two forms of randomness is to reduce the variance without effecting the bias. Another benefit of this is that there is no need to prune the trees. There is another parameter that we don't know how to choose yet, which is the number of trees to put into the forest. However, this is fairly easy to pick if we want optimal results: we can keep on building trees until the error stops decreasing.

Once the set of trees are trained, the output of the forest is the majority vote for classification, as with the other committee methods that we have seen, or the mean response for regression. And those are pretty much the main features needed for creating a random forest. The algorithm is given next before we see some results of using the random forest.

The Basic Random Forest Training Algorithm

- For each of N trees:
  - create a new bootstrap sample of the training set
  - use this bootstrap sample to train a decision tree
  - at each node of the decision tree, randomly select m features, and compute the information gain (or Gini impurity) only on that set of features, selecting the optimal one
  - repeat until the tree is complete

The implementation of this is very easy: we modify the decision to take an extra parameter, which is m, the number of features that should be used in the selection set at each stage. We will look at an example of using it shortly as a comparison to boosting.

Looking at the algorithm you might be able to see that it is a very unusual machine learning method because it is embarrassingly parallel: since the trees do not depend upon each other, you can both create and get decisions from different trees on different individual processors if you have them. This means that the random forest can run on as many processors as you have available with nearly linear speedup.

There is one more nice thing to mention about random forests, which is that with a little bit of programming effort they come with built-in test data: the bootstrap sample will miss out about 35% of the data on average, the so-called out-of-bootstrap examples. If we keep track of these datapoints then they can be used as novel samples for that particular tree, giving an estimated test error that we get without having to use any extra datapoints. This avoids the need for cross-validation.

As a brief example of using the random forest, we start by demonstrating that the random forest gets the correct results on the Party example that has been used in both this and the previous chapters, based on 10 trees, each trained on 7 samples, and with just two levels allowed in each tree:

```
RF prediction
['Party', 'Study', 'Party', 'Party', 'Pub', 'Party', 'Study', 'TV', 'Party', ↩
'Study']
```

As a rather more involved example, the car evaluation dataset in the UCI Repository contains 1,728 examples aiming to classify whether or not a car is a good purchase based on six attributes. The following results compare a single decision tree, bagging, and a random forest with 50 trees, each based on 100 samples, and with a maximum depth of five for each tree. It can be seen that the random forest is the most accurate of the three methods.

```
Tree
Number correctly predicted 777.0
Number of testpoints  864
Percentage Accuracy  89.9305555556

Number of cars rated as good or very good 39.0
Number correctly identified as good or very good 18.0
Percentage Accuracy 46.1538461538
-----
Bagger
Number correctly predicted 678.0
Number of testpoints  864
Percentage Accuracy  78.4722222222

Number of cars rated as good or very good 39.0
Number correctly identified as good or very good 0.0
Percentage Accuracy 0.0
-----
```

```
Forest
Number correctly predicted 793.0
Number of testpoints  864
Percentage Accuracy  91.7824074074

Number of cars rated as good or very good 39.0
Number correctly identified as good or very good 20.0
Percentage Accuracy 51.28205128
```

13.3.1 Comparison with Boosting

There are some obvious similarities to boosting (Section 13.1), but it is the differences that are most telling. The most general thing is that boosting is exhaustive, in that it searches over the whole set of features at each stage, and each stage depends on the previous one. This means that boosting has to run sequentially, and the individual steps can be expensive to run. By way of contrast, the parallelism of the random forest and the fact that it only searches over a fairly small set of features at each stage speed the algorithm up a lot.

Since the algorithm only searches a small subset of the data at each stage, it cannot be expected to be as good as boosting for the same number of trees. However, since the trees are cheaper to train, we can make more of them in the same computational time, and often the results are amazingly good even on very large and complicated datasets.

In fact, the most amazing thing about random forests is that they seem to deal very well with really big datasets. It is fairly clear that they should do well computationally, since both the reduced number of features to search over and the ability to parallelise should help there. However, they seem to also produce good outputs based on surprisingly small parts of the problem space seen by each tree.

13.4 DIFFERENT WAYS TO COMBINE CLASSIFIERS

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

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

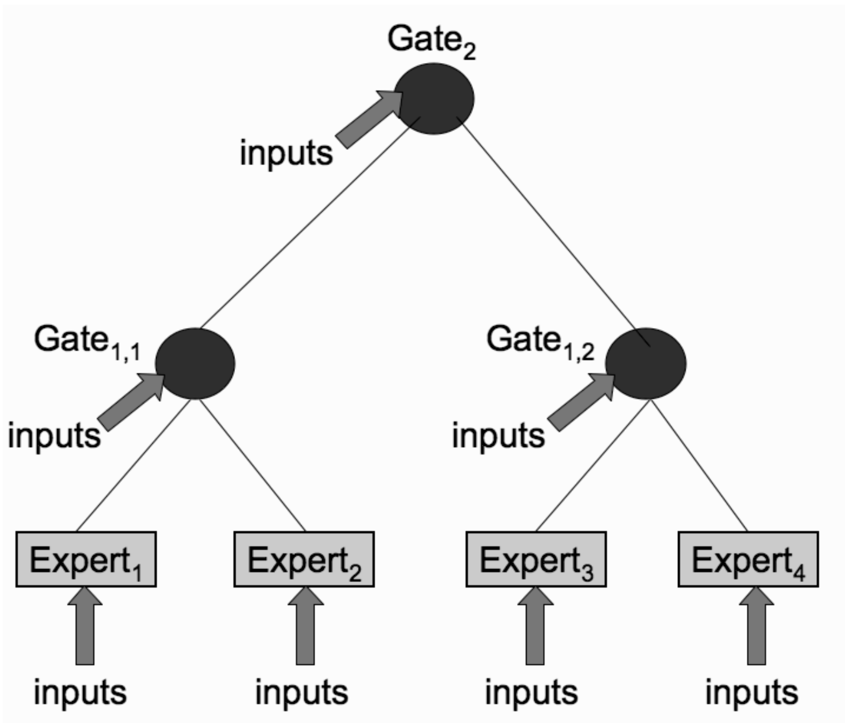

FIGURE 13.5 The Hierarchical Mixture of Networks network, consisting of a set of classifiers (experts) with gating systems that also use the inputs to decide which classifiers to trust.

$$\sum_{k=T/2+1}^{T} \left(\begin{array}{c} T \\ k \end{array} \right) p^k (1-p)^{T-k}, \tag{13.5}$$

where T is the number of classifiers. If $p > 0.5$, then this sum approaches 1 as $T \to \infty$. This is a lot of the power behind ensemble methods: even if each classifier only gets about half the answers right, if we use a decent number of classifiers (maybe 100), then the probability of the ensemble being correct gets close to 1. In fact, even with less than 50% chance of success for each individual classifier, the ensemble can often do very well indeed.

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

There is one more thing that can be done to combine classifiers, and that is to learn how to do it. There is an algorithm that does precisely this, known as the mixture of experts. Inputs are presented to the network, and each individual classifier makes an assessment. These outputs from the classifiers are then weighted by the relevant gate, which produces a weight w using the current inputs, and this is propagated further up the hierarchy. The most common version of the mixture of experts works as follows:

The Mixture of Experts Algorithm

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

$$o_i(\mathbf{x}, \mathbf{w}_i) = \frac{1}{1 + \exp(-\mathbf{w}_i \cdot \mathbf{x})}. \tag{13.6}$$

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

$$g_i(\mathbf{x}, \mathbf{v}_i) = \frac{\exp(\mathbf{v}_i \mathbf{x})}{\sum_l \exp(\mathbf{v}_l \mathbf{x})}. \tag{13.7}$$

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

$$\sum_k o_j g_j. \tag{13.8}$$

The most common way to train this network is using an EM algorithm. This is a general statistical approximation algorithm that is discussed in Section 7.1.1. It is also possible to use gradient descent on the parameters.

There are a couple of other ways to view these mixture of experts methods. One is to regard them as trees, except that the splits are not the hard splits that we performed in Chapter 12, but rather soft, because they are based on probability. The other is to compare them with radial basis function (RBF) networks (see Section 5.2). Each RBF gave a constant output within its receptive field. If, instead, each node were to give a linear approximation to the data, then the result would be the mixture of experts network.

FURTHER READING

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

- R.E. Schapire. The boosting approach to machine learning: An overview. In D. D. Denison, M. H. Hansen, C. Holmes, B. Mallick, and B. Yu, editors, *Nonlinear Estimation and Classification*, Springer, Berlin, Germany, 2003.
- L. Breiman. Bagging predictors. *Machine Learning*, 26(2):123–140, 1996.
- M.I. Jordan and R.A. Jacobs. Hierarchical mixtures of experts and the EM algorithm. *Neural Computation*, 6(2):181–214, 1994.

An overview of the whole area is provided by:

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

For an alternative viewpoint, see:

- Sections 17.4 and 17.6–17.7 of E. Alpaydin. *Introduction to Machine Learning*, 2nd edition, MIT Press, Cambridge, MA, USA, 2009.

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

The original paper on Random Forests is still a very useful resource:
Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

PRACTICE QUESTIONS

Problem 13.1 Modify the decision tree implementation to use weights in the computation of the Gini impurity. This is not trivial, since you have to modify the total value of the Gini impurity, too. Once you have done it, use stump trees on the party data.

Problem 13.2 Implement the alternative form of boosting that uses the weights to sample the dataset. Does this make any difference to the outputs?

Problem 13.3 Stumping picks out the single most informative feature in the dataset and uses this. For a binary classification problem this will typically get at least half of the dataset correct. Why? How does this statement generalise to multiple classes?

Problem 13.4 Compare and contrast bagging and cross-validation.

Problem 13.5 The `Breastcancer` dataset in the UCI Machine Learning repository gives ten features and asks for a classification of breast tumours into benign and malignant. It is a difficult dataset, and provides a good comparison of the standard decision tree with boosted and bagged versions. Use all of the methods, using stumping and more advanced trees and see which work better.

Problem 13.6 The Mixture of Experts algorithm works with any kind of expert. Suppose that the experts were each MLPs. Implement this algorithm and see how well it does on the `Breastcancer` dataset above.

Problem 13.7 In Section 13.3 on the random forest, it was mentioned that there exists out-of-bootstrap data that can be used for validation and testing. Modify the code to keep track of this data.

Problem 13.8 Use the boosting code from Problem 13.2 above and compare it with the random forest on the cars dataset from the UCI Repository.

CHAPTER 14

Unsupervised Learning

Many of the learning algorithms that we have seen to date have made use of a training set that consists of a collection of labelled target data, or at least (for evolutionary and reinforcement learning) some scoring system that identifies whether or not a prediction is good or not. Targets are obviously useful, since they enable us to show the algorithm the correct answer to possible inputs, but in many circumstances they are difficult to obtain—they could, for instance, involve somebody labelling each instance by hand. In addition, it doesn't seem to be very biologically plausible: most of the time when we are learning, we don't get told exactly what the right answer should be. In this chapter we will consider exactly the opposite case, where there is no information about the correct outputs available at all, and the algorithm is left to spot some similarity between different inputs for itself.

Unsupervised learning is a conceptually different problem to supervised learning. Obviously, we can't hope to perform regression: we don't know the outputs for any points, so we can't guess what the function is. Can we hope to do classification then? The aim of classification is to identify similarities between inputs that belong to the same class. There isn't any information about the correct classes, but if the algorithm can exploit similarities between inputs in order to cluster inputs that are similar together, this might perform classification automatically. So the aim of unsupervised learning is to find clusters of similar inputs in the data without being explicitly told that these datapoints belong to one class and those to a different class. Instead, the algorithm has to discover the similarities for itself. We have already seen some unsupervised learning algorithms in Chapter 6, where the focus was on dimensionality reduction, and hence clustering of similar datapoints together.

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

To see how to work out a general error criterion that we can use, we need to go back to some of the important concepts that were discussed in Section 2.1.1: input space and weight space. If two inputs are close together then it means that their vectors are similar, and so the distance between them is small (distance measures were discussed in Section 7.2.3, but here we will stick to Euclidean distance). Then inputs that are close together are identified as being similar, so that they can be clustered, while inputs that are far apart are not clustered together. We can extend this to the nodes of a network by aligning weight space with input space. Now if the weight values of a node are similar to the elements of an input vector then that node should be a good match for the input, and any other inputs that are similar. In order to start to see these ideas in practice we'll look at a simple clustering algorithm, the k-Means Algorithm, which has been around in statistics for a long time.

14.1 THE K-MEANS ALGORITHM

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

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

A distance measure In order to talk about distances between points, we need some way to measure distances. It is often the normal Euclidean distance, but there are other alternatives; we've covered some other alternatives in Section 7.2.3.

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

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

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

We start by positioning the cluster centres randomly through the input space, since we don't know where to put them, and then we update their positions according to the data. We decide which cluster each datapoint belongs to by computing the distance between each datapoint and all of the cluster centres, and assigning it to the cluster that is the closest. Note that we can reduce the computational cost of this procedure by using the KD-Tree algorithm that was described in Section 7.2.2. For all of the points that are assigned to a cluster, we then compute the mean of them, and move the cluster centre to that place. We iterate the algorithm until the cluster centres stop moving. Here is the algorithmic description:

The k-Means Algorithm

```
• Initialisation
    – choose a value for $k$
    – choose $k$ random positions in the input space
    – assign the cluster centres $\boldsymbol{\mu}_j$ to those positions
• Learning
    – repeat
        * for each datapoint $\mathbf{x}_i$:
            · compute the distance to each cluster centre
            · assign the datapoint to the nearest cluster centre with distance
                  $d_i = \min_j d(\mathbf{x}_i, \boldsymbol{\mu}_j).$        (14.1)
        * for each cluster centre:
            · move the position of the centre to the mean of the points in that cluster
              ($N_j$ is the number of points in cluster $j$):
                  $\boldsymbol{\mu}_j = \frac{1}{N_j} \sum_{i=1}^{N_j} \mathbf{x}_i$        (14.2)
    – until the cluster centres stop moving
• Usage
    – for each test point:
        * compute the distance to each cluster centre
        * assign the datapoint to the nearest cluster centre with distance
              $d_i = \min_j d(\mathbf{x}_i, \boldsymbol{\mu}_j).$        (14.3)
```

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

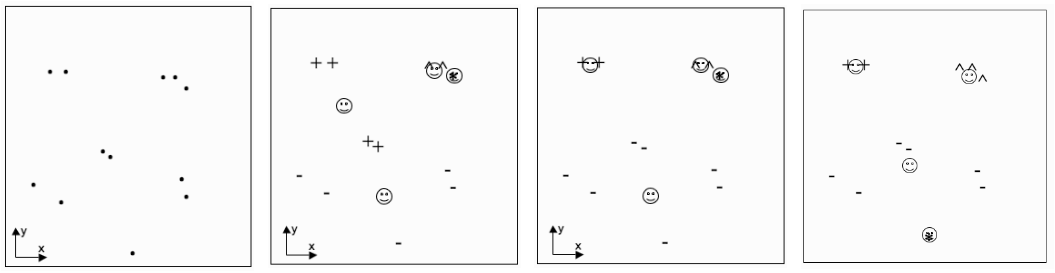

FIGURE 14.1 *Left:* A two-dimensional dataset. *Right:* Three possible ways to position 4 centres (drawn as faces) using the k-means algorithm, which is clearly susceptible to local minima.

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

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

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

To see how this works in practice, Figures 14.1 and 14.2 show some data and some different ways to cluster that data computed by the k-means algorithm. It should be clear that the algorithm is susceptible to local minima: depending upon where the centres are initially positioned in the space, you can get very different solutions, and many of them look very unlikely to our eyes. Figure 14.2 shows examples of what happens when you choose the number of centres wrongly. There are certainly cases where we don't know in advance how many clusters we will see in the data, but the k-means algorithm doesn't deal with this at all well.

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

By running the algorithm with lots of different values of k, we can see which values give us the best solution. Of course, we need to be careful with this. If we still just measure the sum-of-squares error between each datapoint and its nearest cluster centre, then when

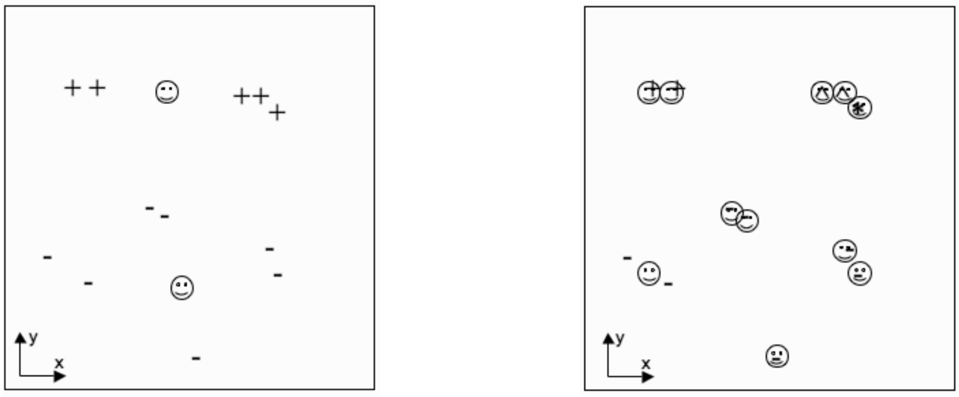

FIGURE 14.2 *Left:* A solution with only 2 classes, which does not match the data well. *Right:* A solution with 11 classes, showing severe overfitting.

we set k to be equal to the number of datapoints, we can position one centre on every datapoint, and the sum-of-squares error will be zero (in fact, this won't happen, since the random initialisation will mean that several clusters will end up coinciding). However, there is no generalisation in this solution: it is a case of serious overfitting. However, by computing the error on a validation set and multiplying the error by k we can see something about the benefit of adding each extra cluster centre.

14.1.1 Dealing with Noise

There are lots of reasons for performing clustering, but one of the more common ones is to deal with noisy data readings. These might be slightly corrupted, or occasionally just plain wrong. If we can choose the clusters correctly, then we have effectively removed the noise, because we replace each noisy datapoint by the cluster centre (we will use this way of representing datapoints for other purposes in Section 14.2). Unfortunately, the mean average, which is central to the k-means algorithm, is very susceptible to outliers, i.e., very noisy measurements. One way to avoid the problem is to replace the mean average with the median, which is what is known as a robust statistic, meaning that it is not affected by outliers (the mean of $(1, 2, 1, 2, 100)$ is 21.2, while the median is 2). The only change that is needed to the algorithm is to replace the computation of the mean with the computation of the median. This is computationally more expensive, as we've discussed previously, but it does remove noise effectively.

14.1.2 The k-Means Neural Network

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

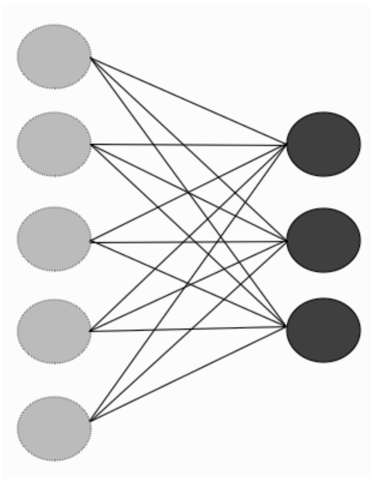

FIGURE 14.3 A single-layer neural network can implement the k-means solution.

just make the activation of a node be the distance between that node in weight space and the current input, as we did for Radial Basis Functions in Chapter 5. Then training is just moving the position of the node, which means adjusting the weights.

So, we can implement the k-means algorithm using a set of neurons. We will use just one layer of neurons, together with some input nodes, and no bias node. The first layer will be the inputs, which don't do any computation, as usual, and the second layer will be a layer of competitive neurons, that is, neurons that 'compete' to fire, with only one of them actually succeeding. Only one cluster centre can represent a particular input vector, and so we will choose the neuron with the highest activation h to be the one that fires. This is known as winner-takes-all activation, and it is an example of competitive learning, since the set of neurons compete with each other to fire, with the winner being the one that best matches (i.e., is closest to) the input. Competitive learning is sometimes said to lead to grandmother cells, because each neuron in the network will learn to recognise one particular feature, and will fire only when that input is seen. You would then have a specific neuron that was trained to recognise your grandmother (and others for anybody else/anything else that you see often).

We will choose k neurons (for hopefully obvious reasons) and fully connect the inputs to the neurons, as usual. There is a picture of this network in Figure 14.3. We will use neurons with a linear transfer function, computing the activation of the neurons as simply the product of the weights and inputs:

$$h_i = \sum_j w_{ij} x_j. \tag{14.4}$$

Providing that the inputs are normalised so that their absolute size is the same (a point that we'll come back to in Section 14.1.3), this effectively measures the distance between the input vector and the cluster centre represented by that neuron, with larger numbers (higher activations) meaning that the two points are closer together.

So the winning neuron is the one that is closest to the current input. The question is how can we then change the position of that neuron in weight space, that is, how do we update its weights? In the k-means algorithm that was described earlier it was easy: we just set the cluster centre to be the mean of all the datapoints that were assigned to that

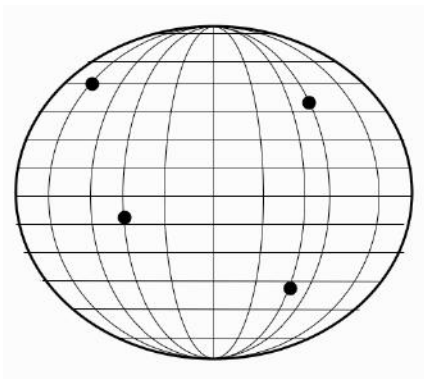

FIGURE 14.4 A set of neurons positioned on the unit sphere in 3D.

centre. However, when we do neural network training, we often feed in just one input vector at a time and change the weights (that is, we use the algorithm on-line, rather than batch). We therefore do not know the mean because we don't know about all the datapoints, just the current one. So we approximate it by moving the winning neuron closer to the current input, making that centre even more likely to be the best match next time that input is seen. This corresponds to:

$$\Delta w_{ij} = \eta x_j. \tag{14.5}$$

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

14.1.3 Normalisation

Suppose that the weights of all the neurons are small (maybe less than 1) except for those to one particular neuron. We'll make those weights be 10 for the example. If an input vector with values $(0.2, 0.2, -0.1)$ is presented, and it happens to be an exact match for one of the neurons, then the activation of that neuron will be $0.2 \times 0.2 + 0.2 \times 0.2 + -0.1 \times -0.1 = 0.09$. The other neurons are not perfect matches, so their activations should all be less. However, consider the neuron with large weights. Its activation will be $10 \times 0.2 + 10 \times 0.2 + 10 \times -0.1 = 3$, and so it will be the winner. Thus, we can only compare activations if we know that the weights for all of the neurons are the same size. We do this by insisting that the weight vector is normalised so that the distance between the vector and the origin (the point $(0, 0, \ldots 0)$) is one. This means that all of the neurons are positioned on the unit hypersphere, which we described in Section 2.1.2 when we talked about the curse of dimensionality: it is the set of all points that are distance one from the origin, so it is a circle in 2D, a sphere in 3D (as shown in Figure 14.4), and a hypersphere in higher dimensions.

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

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

The neuronal activation (Equation (14.4)) can be written as:

$$h_i = \mathbf{W}_i^T \cdot \mathbf{x}, \tag{14.6}$$

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

14.1.4 A Better Weight Update Rule

The weight update rule given in Equation (14.5) lets the weights grow without any bound, so that they do not lie on the unit hypersphere any more. If we normalise the inputs as well, which certainly seems reasonable, then we can use the following weight update rule:

$$\Delta w_{ij} = \eta(x_j - w_{ij}), \tag{14.7}$$

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

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

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

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

The On-Line k-Means Algorithm

```
• Initialisation
    – choose a value for $k$, which corresponds to the number of output nodes
    – initialise the weights to have small random values
• Learning
    – normalise the data so that all the points lie on the unit sphere
    – repeat:
        ∗ for each datapoint:
            · compute the activations of all the nodes
            · pick the winner as the node with the highest activation
            · update the weights using Equation (14.7)
        ∗ until number of iterations is above a threshold
• Usage
    – for each test point:
        ∗ compute the activations of all the nodes
        ∗ pick the winner as the node with the highest activation
```

14.1.5 Example: The Iris Dataset Again

Now that we have a method of training the k-means algorithm we can use it to learn about data. Except we need to think about how to understand the results. If there aren't any labels in the data, then we can't really do much to analyse the results, since we don't have anything to compare them with. However, we might use unsupervised learning methods to cluster data where we know at least some of the labels. For example, we can use the algorithm on the iris dataset that we looked at in Section 4.4.3, where we classified three types of iris flowers using the MLP. All we need to do is to give some of the data to the algorithm and train it, and then use some more to test the output. However, the output of the algorithm isn't as clear now, because we don't use the labels that come with the data, since we aren't doing supervised learning anymore. To get around that, we need to work out some way of turning the results from the algorithm, which is the index of the cluster that best matches it, into a classification output that we can compare with the labels. This is relatively easy if we used three clusters in the algorithm, since there should hopefully be a one-to-one correspondence between them, but it might turn out that using more clusters gets better results, although this will make the analysis more difficult. You can do this by hand if there are relatively small numbers of datapoints, or you could use a supervised learning algorithm to do it for you, as is discussed next.

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

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

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

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

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

14.1.6 Using Competitive Learning for Clustering

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

There is an alternative solution to this problem of assigning labels, and it is one that we have seen before. In Chapter 5 we considered using the k-means network in order to train the positions of the RBF nodes. It is now possible to see how this works. The k-means part positions the RBFs in the input space, so that they represent the input data well. A Perceptron is then used on top of this in order to provide the match to the outputs in the supervised learning part of the network. Since this is now supervised learning, it ensures that the output categories match the target data classes. It also means that you can use lots of clusters in the k-means network without having to work out which datapoints belong to which cluster, since the Perceptron will do this for you.

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

14.2 VECTOR QUANTISATION

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

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

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

Figure 14.5 shows an interpretation of prototype vectors in two dimensions. The dots at the centre of each cell are the prototype vectors, and any datapoint that lies within a cell is represented by the dot. The name for each cell is the Voronoi set of a particular prototype. Together, they produce the Voronoi tesselation of the space. If you connect together every pair of points that share an edge, as is shown by the dotted lines, then you get the Delaunay triangulation, which is the optimal way to organise the space to perform function approximation.

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

14.3 THE SELF-ORGANISING FEATURE MAP

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

There are two novel departures in this for us: firstly, the relative locations of the neurons in the network matters (this property is known as feature mapping—nearby neurons

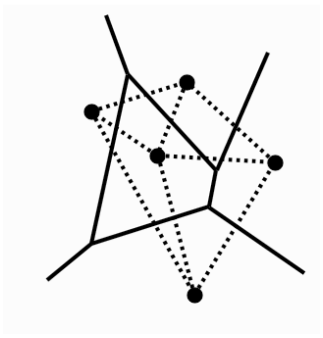

FIGURE 14.5 The Voronoi tesselation of space that performs vector quantisation. Any datapoint is represented by the dot within its cell, which is the prototype vector.

correspond to similar input patterns), and secondly, the neurons are arranged in a grid with connections between the neurons, rather than in layers with connections only between the different layers. In the auditory cortex there appears to be sheets of neurons arranged in 2D, and that is the typical arrangement of neurons for the SOM: a grid of neurons arranged in 2D, as can be seen in Figure 14.6. A 1D line of neurons is also sometimes used. In mathematical terms, the SOM demonstrates relative ordering preservation, which is sometimes known as topology preservation. The relative ordering of the inputs should be preserved by the ordering in the neurons, so that neurons that are close together represent inputs that are close together, while neurons that are far apart represent inputs that are far apart.

This topology preservation is not necessarily possible, because the SOM typically uses a 1D or 2D array of neurons, and most of our input spaces are of much higher dimensionality than that. This means that the ordering cannot be preserved. We have seen this in Figure 1.2, where one view of some wind turbines made it look like they are on top of each other, when they clearly are not, because we used a two-dimensional representation of three-dimensional reality. You've probably seen the same thing in other photos, where trees appear to be growing out of somebody's head. A different way to see the same thing is given in Figure 14.7, where mismatches between the topology of the input space and map lead to changes in the relative ordering. The best that can be said is that SOM is perfectly topology-preserving, which means that if the dimensionality of the input and the map correspond, then the topology of the input space will be preserved. We are going to look at other methods of performing dimensionality reduction in Chapter 6.

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

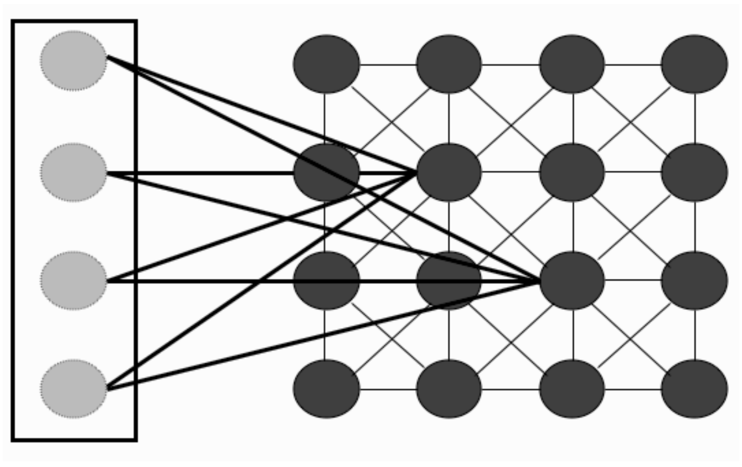

FIGURE 14.6 The Self-Organising Map network. As usual, input nodes (on the left) do no computation, and the weights are modified to change the activations of the neurons (weights are only shown to two nodes for clarity). However, the nodes within the SOM affect each other in that the winning node also changes the weights of neurons that are close to it. Connections are shown in the figure to the eight closest nodes, but this is a parameter of the network.

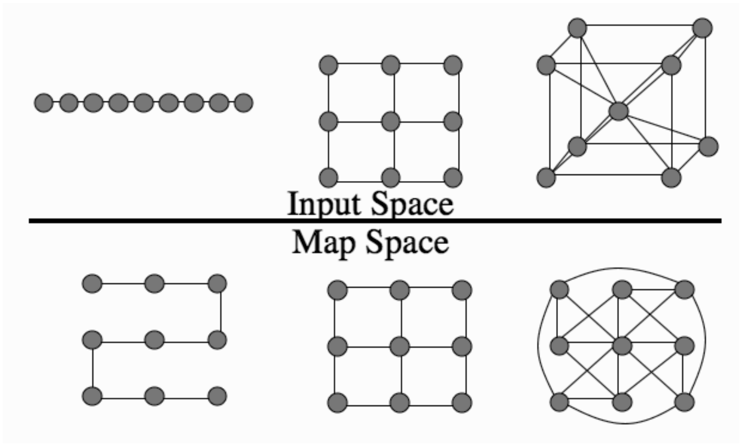

FIGURE 14.7 When inputs in 1D (a straight line), a 2D grid, and a 3D cube are represented by a 2D grid of neurons, the relative ordering is not perfectly preserved. The 1D line is bent, which means that points that used to be a long way apart (such as the first and sixth on the line) are now close together, while the cube becomes very complicated. The lines in the bottom part of the figure represent connections that are meant to be close.

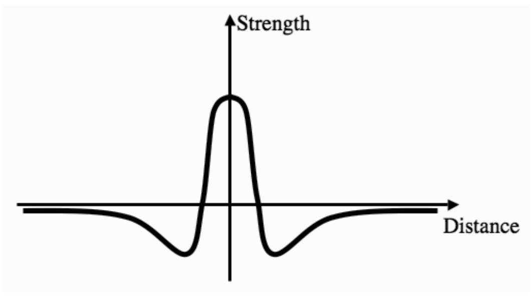

FIGURE 14.8 Graph of the strength of lateral connections for a feature mapping algorithm known as the 'Mexican Hat'.

This is known as the 'Mexican Hat' form of lateral connections, for reasons that should be clear from the picture in Figure 14.8. We can then just use ordinary competitive learning, just like we did for the k-means network in Section 14.1.2. The Self-Organising Map does pretty much exactly this.

14.3.1 The SOM Algorithm

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

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

The Self-Organising Feature Map Algorithm

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

$$n_b = \min_j \|\mathbf{x} - \mathbf{w}_j^T\|. \tag{14.8}$$

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

$$\mathbf{w}_j^T \leftarrow \mathbf{w}_j^T + \eta(t)(\mathbf{x} - \mathbf{w}_j^T), \tag{14.9}$$

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

  * update the weight vector of all other neurons using:

$$\mathbf{w}_j^T \leftarrow \mathbf{w}_j^T + \eta_n(t)h(n_b,t)(\mathbf{x} - \mathbf{w}_j^T), \tag{14.10}$$

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

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

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

• **Usage**

- for each test point:

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

$$n_b = \min_j \|\mathbf{x} - \mathbf{w}_j^T\| \tag{14.11}$$

14.3.2 Neighbourhood Connections

The size of the neighbourhood is thus another parameter that we need to control. How large should the neighbourhood of a neuron be? If we start our network off with random weights, as we did for the MLP, then at the beginning of learning, the network is pretty well unordered (as the weights are random, two nodes that are very close in weight space could be on opposite sides of the map, and vice versa) and so it makes sense that the neighbourhoods should be large, so that we get the rough ordering of the network correct. However, once the network has been learning for a while, the rough ordering has already been created, and the algorithm starts to fine-tune the individual local regions of the network. At this stage, the neighbourhoods should be small, as is shown in Figure 14.9. It therefore makes sense to reduce the size of the neighbourhood as the network adapts. These two phases of learning are also known as ordering and convergence. Typically, we reduce the neighbourhood size by a small amount at each iteration of the algorithm. We control the learning rate η in exactly the same way, so that it starts off large and decreases over time, as is shown in the algorithm below.

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

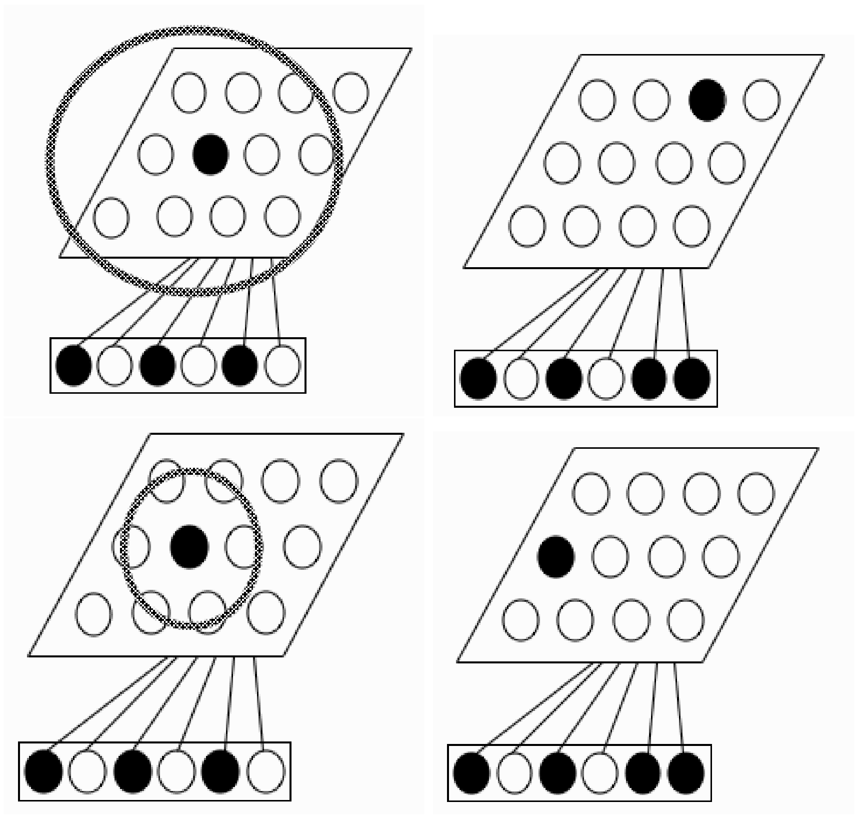

FIGURE 14.9 *Top:* Initially, similar input vectors excite neurons that are far apart, so that the neighbourhood (shown as a circle) needs to be large. *Bottom:* Later on during training the neighbourhood can be smaller, because similar input vectors excite neurons that are close together.

```
# Set up the map distance matrix
mapDist = np.zeros((self.x*self.y,self.x*self.y))
for i in range(self.x*self.y):
    for j in range(i+1,self.x*self.y):
        mapDist[i,j] = np.sqrt((self.map[0,i] - self.map[0,j])**2 + (self.↩
        map[1,i] - self.map[1,j])**2)
 mapDist[j,i] = mapDist[i,j]

# Within the loop, select the neighbours
# Find the neighbours and update their weights
neighbours = np.where(mapDist[best[i]]<=self.nSize,1,0)
neighbours[best[i]] = 0
self.weights += self.eta_n * neighbours*np.transpose((inputs[i,:] - np.↩
transpose(self.weights)))
```

There is another way to initialise the weights in the network, which is to use Principal Components Analysis (which is described in Section 6.2) to find the two (assuming that the map is two-dimensional) largest directions of variation in the data and to initialise the weights so that they increase along these two directions:

```
dummy1,dummy2,evals,evecs = pca.pca(inputs,2)
self.weights = np.zeros((self.nDim,x*y))
for i in range(x*y):
    for j in range(self.mapDim):
        self.weights[:,i] += (self.map[j,i]-0.5)*2*evecs[:,j]
```

This means that the ordering part of the training has already been done in the initialisation, and so the algorithm can be trained with small neighbourhood size from the start. Obviously, this is only possible if the training of the algorithm is in batch mode, so that you have all of the data available for training right from the start. This should be true for the SOM anyway—it is not designed for on-line learning. This can be a bit of a limitation, because there are many cases where we would like to do unsupervised on-line learning.

There are a couple of different things that we can do. One is to ignore that constraint and use the SOM anyway. This is fairly common. However, the size of the map really starts to matter, and there is no guarantee that the SOM will converge to a solution unless batch learning is applied. The alternative is to use one of a variety of networks that are designed to deal with exactly this situation. There are a fair number of these, but Fritzke's "Growing Neural Gas" and Marsland's "Grow When Required" Network are two of the more common ones.

14.3.3 Self-Organisation

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

14.3.4 Network Dimensionality and Boundary Conditions

We typically think about applying the SOM algorithm to a 2D rectangular array of neurons (as shown in Figure 14.6), but there is nothing in the algorithm to force this. There are cases where a line of neurons (1D) works better, or where three dimensions are needed. It depends on the dimensionality of the inputs (actually on the intrinsic dimensionality, the number of dimensions that you actually need to represent the data), not the number that it is embedded in. As an example, consider a set of inputs spread through the room you are in, but all on the plane that connects the bottom of the wall to your left with the top of the wall to your right. These points have intrinsic dimensionality two since they are all on the plane, but they are embedded in your three-dimensional room. Noise and other inaccuracies in data often lead to it being represented in more dimensions than are actually required, and so finding the intrinsic dimensionality can help to reduce the noise.

We also need to consider the boundaries of the network. In some cases, it makes sense that the edges of the map of neurons is strictly defined — for example, if we are arranging sounds from low pitch to high pitch, then the lowest and highest pitches we can hear are obvious endpoints. However, it is not always the case that such boundaries are clearly defined. In this case we might want to remove the boundary conditions. We can do this by removing the boundary by tying the ends together. In 1D this means that we turn a line into a circle, while in 2D we turn a rectangle into a torus. To see this, try taking a piece of paper and bend it so that the top and bottom edges line up. You've now got a tube. If you bend the tube round so that the two open ends meet up you have a circle of tube known as a torus. Pictures of these effects are shown in Figure 14.10. In effect, it means that there are no neurons on the edge of the feature map. The choice of the number of dimensions and the boundary conditions depends on the problem that we are considering, but it is usually the case that the torus works better than the rectangle, although it is not always clear why.

The one cost that this has is that the map distances get more complicated to calculate, since we now need to calculate the distances allowing for the wrap around. This can be done using modulo arithmetic, but it is easier to think about taking copies of the map and putting them around the map, so that the original map has copies of itself all around: one above, one below, to the right and left, and also diagonally above and below, as is shown in Figure 14.11. Now we keep one of the points in the original map, and the distance to the second node is the smallest of the distances between the first node and the copies of the second node in the different maps (including the original). By treating the distances in x and y separately, the number of distances that has to be computed can be reduced.

As with the competitive learning algorithm that we considered earlier, the size of the SOM is defined before we start learning. The size of the network (that is, the number of neurons that we put into it) decides how fine-grained the learning is. If there are very few neurons, then the best that the network can do is to find gross generalisations that link the data. However, if there are very large numbers of neurons, then the network can

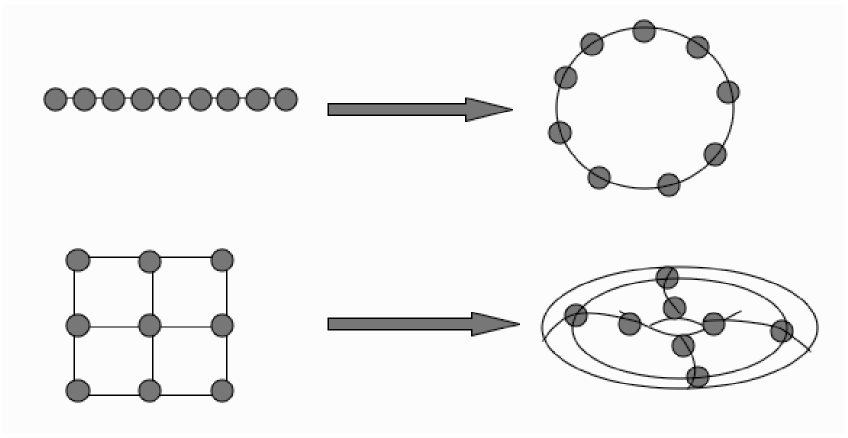
FIGURE 14.10 Using circular boundary conditions in 1D turns a line into a circle, while in 2D it turns a rectangle into a torus.

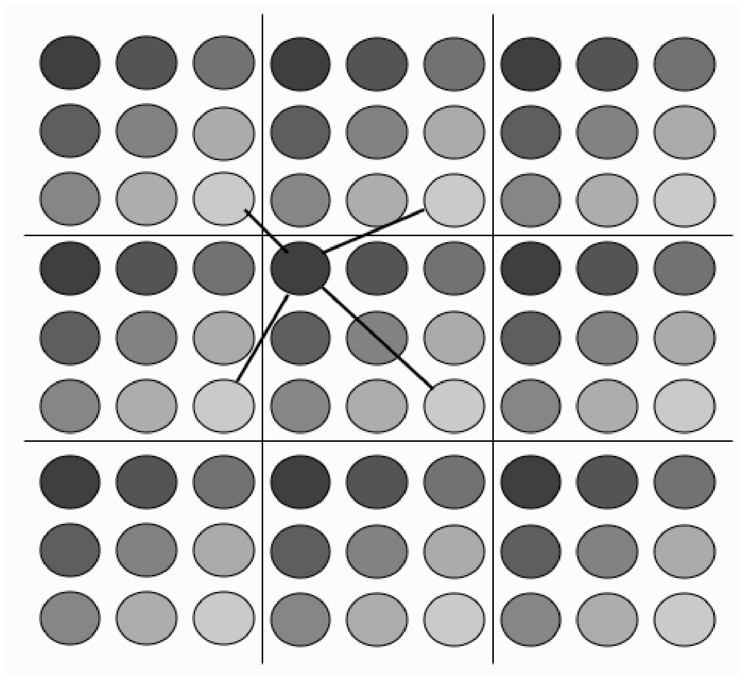
FIGURE 14.11 One way to compute distances between points without any boundary on the map is to imagine copies of the entire map being placed around the original, and picking the shortest of the distances between a node and any of the copies of the other node.

represent every input without ever needing to generalise at all. This is yet another example of overfitting. Clearly, then, choosing the correct size of network is important. The common approach is to test out several different sizes of network, such as 5×5 and 10×10 and see how well the network learns.

14.3.5 Examples of Using the SOM

As a first example of using the SOM, and one that shows the topological ordering of the network, consider training the network on a set of two-dimensional data drawn at random from a uniform distribution in $[-1, 1]$ in both directions. If the network weights are started off randomly, then initially the network is completely disordered (as shown in the top-left picture in Figure 14.12), but after 10 iterations of training the network is ordered so that neighbouring nodes map to data that is close together (bottom-left). Using PCA to initialise the map is not especially useful for this dataset, but it does speed things up: only five iterations through the dataset produce the output shown on the bottom-right of the figure, where it started from the version on the top-right.

For two examples of using the SOM on non-random data, where we can expect to see some actual learning, we will first look at the iris data that we used with the k-means algorithm earlier in this chapter. Figure 14.13 shows a plot of which node of a 5×5 Self-Organising Map was the best match on a set of test data after training for 100 iterations. The three different classes are shown as different shapes (squares, plus triangles pointing up and down), but remember that the network did not receive any information about these target classes. It can be seen that the examples in each of the three classes form different clusters in the map. Looking at the figure, you might be wondering if it is possible to use the plot to identify the different classes by assuming that they are separated in the map. This has been investigated—often by using methods similar to those of Linear Discriminant Analysis that are described in Section 6.1—with some success, and a reference is provided at the end of the chapter.

A more difficult problem is shown in Figure 14.14. The data are the `ecoli` dataset from the UCI Machine Learning repository, and the class is the localisation site of the protein, based on a set of protein measurements. The results with this dataset when testing are not as clearly impressive (but note that the MLP gets about 50% accuracy on this dataset, and that has the target data, which the SOM doesn't). However, the clusters can still be seen to some extent, and they are very clear in the training data. Note that the boundary conditions can make things a little more complicated, since the cluster does not necessarily respect the edges of the map.

FURTHER READING

There is a book by Kohonen, the inventor of the SOM, that provides a very good overview (if rather dated, now) of the area:

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

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

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

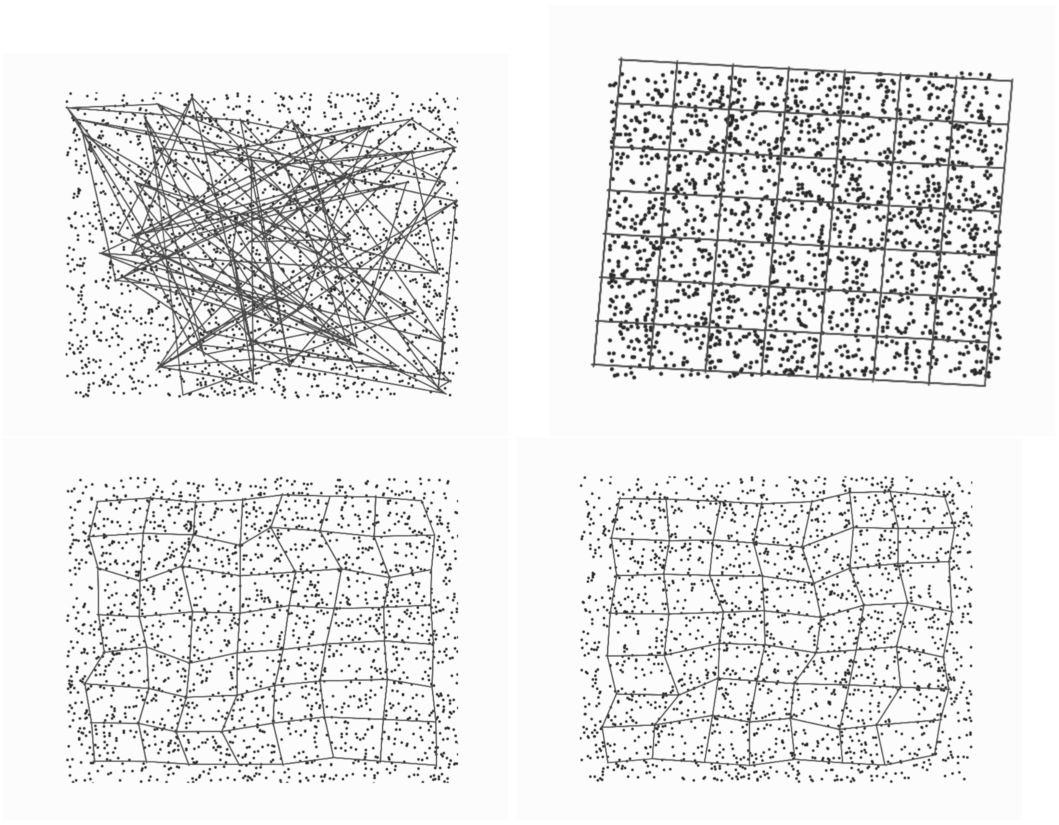
FIGURE 14.12 Training the SOM on a set of uniformly randomly sampled two-dimensional data in the range $[-1, 1]$ in both dimensions. *Top:* Initialisation of the map using *left:* random weights and *right:* PCA (the randomness in the data means that the directions of variation are not necessarily along the obvious directions). *Bottom:* The output after just 10 iterations of training on the left, and 5 on the right, both with typical parameter values.

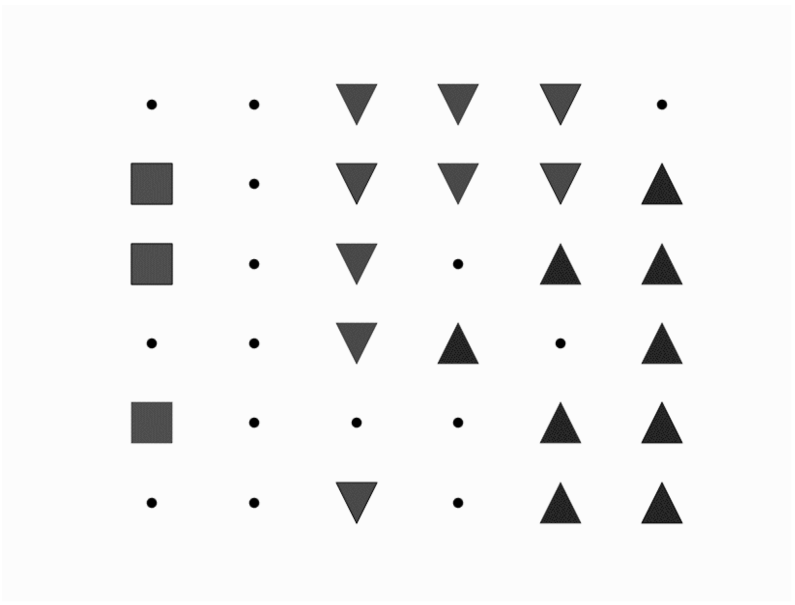
FIGURE 14.13 Plot showing which nodes are the best match according to class, with the three shapes corresponding to three different classes in the iris dataset. The small dots represent nodes that did not fire.

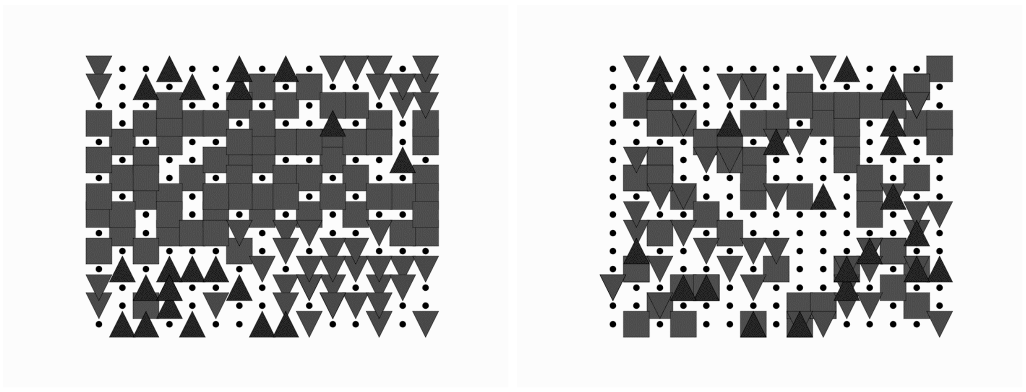
FIGURE 14.14 Plots showing which nodes are the best match according to class, with the three shapes corresponding to three different classes in the *E. coli* dataset, tested on *left:* the training set and *right:* a separate test set. The small dots represent nodes that did not fire.

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

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

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

Books that cover the area include:

- Section 10.14 of R.O. Duda, P.E. Hart, and D.G. Stork. *Pattern Classification*, 2nd edition, Wiley-Interscience, New York, USA, 2001.
- Chapter 9 of S. Haykin. *Neural Networks: A Comprehensive Foundation*, 2nd edition, Prentice-Hall, New Jersey, USA, 1999.
- Section 9.3 of B.D. Ripley. *Pattern Recognition and Neural Networks.* Cambridge University Press, Cambridge, UK, 1996.

PRACTICE QUESTIONS

Problem 14.1 What is the purpose of the neighbourhood function in the SOM? How does it change the learning?

Problem 14.2 A simplistic intruder detection system for a computer network consists of an attempt to categorise users according to (i) the time of day they log in, (ii) the length of time they log in for, (iii) the types of programs they run while logged in, (iv) the number of programs they run while logged in. Suggest how you would train a SOM and the naïve Bayes' classifier to perform the categorisation. What preprocessing of the data would you do, how much data would you need, and how large would you make the SOM? Do you think that such a system would work for intruder detection?

Problem 14.3 The Music Genome Project (`http://www.pandora.com`) does not work by using a SOM. But it could. Describe how you would implement it.

Problem 14.4 A bank wants to detect fraudulent credit card transactions. They have data for lots and lots of transactions (each transaction is an amount of money, a shop, and the time and date) and some information about when credit cards were stolen, and the transactions that were performed on the stolen card. Describe how you could use a competitive learning method to cluster people's transactions together to identify patterns, so that stolen cards can be detected as changes in pattern. How well do you think this would work? There is much more data of transactions when cards are not stolen, compared to stolen transactions. How does this affect the learning, and what can you do about it?

Problem 14.5 It is possible to use any competitive learning method to position the basis functions of a Radial Basis Function network. The example code used k-means. Modify it to use the SOM instead and compare the results on the `wine` and `yeast` datasets.

Problem 14.6 For the `wine` dataset, experiment with different sizes of map, and boundary conditions. How much difference does it make? Can you use the principal components in order to set the size automatically?

CHAPTER 15

Markov Chain Monte Carlo (MCMC) Methods

In this chapter we are going to look at a method that has revolutionised statistical computing and statistical physics over the past 20 years. The principal algorithm has been around since 1953, but only when computers became fast enough to be able to perform the computations on real-world examples in hours instead of weeks did the methods become really well known. However, this algorithm has now been cited as one of the most influential ever created.

There are two basic problems that can be solved using these methods, and they are the two that we have been wrestling with for pretty much the entire book: we may want to compute the optimum solution to some objective function, or compute the posterior distribution of a statistical learning problem. In either case the state space may well be very large, and we are only interested in finding the best possible answer—the steps that we go through along the way are not important. We've seen several methods of solving these types of problems during the book, and here we are going to look at one more. We will see a place where MCMC methods are very useful in Section 16.1.

The idea behind everything that we are going to talk about in this chapter is that as we explore the state space, we can also construct samples as we go along in such a way that the samples are likely to come from the most probable parts of the state space. In order to see what this means, we will discuss what Monte Carlo sampling is, and look at Markov chains.

15.1 SAMPLING

We have produced samples from probability distributions in almost all of the algorithms we have looked at, for example, for initialisation of weights. In many cases, the probability distribution we have used has been the uniform one on $[0, 1)$, and we have done it using the `np.random.rand()` function in NumPy, although we have also seen sampling from Gaussian distributions using `np.random.normal()`.

15.1.1 Random Numbers

The basis of all of these sampling methods is in the generation of random numbers, and this is something that computers are not really capable of doing. However, there are plenty of algorithms that produce pseudo-random numbers, the simplest of which is the linear congruential generator. This is a very simple function that is defined by a recurrence relation (i.e., you put one number in to get the second number, and then feed that back in to get the third, and then repeat the cycle):

$$x_{n+1} = (ax_n + c) \mod m, \tag{15.1}$$

where a, c, and m are parameters that have to be chosen carefully. All of them, and the initial input x_0 (which is known as the seed), are integers, and so are all of the outputs. The modulus function means that the largest number that can be produced is m, and so there are at most m numbers that can be produced by the algorithm. Once one number appears a second time, the whole pattern will repeat again since the equation only uses the current output as input. The length of the sequence between repeats is the period, and it should obviously be as long as possible, since it is the most obvious non-randomness in the algorithm. There has been a lot of investigation of choices of the parameters so that the period is length m, so that every integer between 0 and m is produced before the pattern cycles. There are various choices of the parameters that have been selected to work well, including $m = 2^{32}$; $a = 1,664,525$; and $c = 1,013,904,223$. Clearly, just picking numbers at random isn't going to be that useful.

There has been a lot of effort put into different random number generators, since they are important not just for statistical computing, but also cryptography and security. The industry-standard algorithm for generating random samples is the Mersenne Twister, which is based on Mersenne prime numbers. It is the random number generator used in NumPy. No matter what algorithm generates the numbers, though, it is important to remember that they are not genuinely random, and to genuflect to the wisdom of John von Neumann, one of the fathers of modern computing, who stated:

> Anyone who considers arithmetic methods of producing random digits is, of course, in a state of sin.

The other troublesome thing about random numbers is that it is not actually possible to prove that a sequence of numbers is truly random. There are several tests that can be made of a sequence of numbers to see if they seem to be random. Examples include calculating the entropy of the sequence (entropy was described in Section 12.2.1), using a compression algorithm on the sequence (since compression algorithms exploit redundancy, i.e., predictability, in the input, if the compression algorithms fail to make the input smaller, then it might be because they are random), and just checking how many numbers are odd compared to even. However, you can never guarantee that a sequence is random, just that it hasn't failed most of the tests yet (but just because it fails one or two of them at some point in the sequence doesn't mean that the sequence isn't random; truly random numbers can look deterministic for a long time... this is part of the joy of randomness!). I'll leave the last word on this to von Neumann again:

> In my experience it was more trouble to test random sequences than to manufacture them.

15.1.2 Gaussian Random Numbers

The Mersenne twister produces uniform random numbers. However, often we might want to produce samples from other distributions, e.g., Gaussian. The usual method of doing this is the Box–Muller scheme, which uses a pair of uniformly randomly distributed numbers in order to make two independent Gaussian-distributed numbers with zero mean and unit variance. Let's see how it works.

Suppose that we had two independent zero mean, unit variance normals. Then their product is:

$$f(x, y) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2} \frac{1}{\sqrt{2\pi}} e^{-y^2/2} = \frac{1}{2\pi} e^{-(x^2+y^2)/2}. \tag{15.2}$$

If we use polar coordinates instead (so $x = r\sin(\theta)$ and $y = r\cos(\theta)$) then we would have $r^2 = x^2 + y^2$ and $\theta = \tan^{-1}(y/x)$. Both of these are uniformly distributed random variables ($0 \leq r \leq 1$ and $0 \leq \theta < 2\pi$). In other words, $\theta = 2\pi U_1$ where U_1 is a uniformly distributed random variable. Now we just need a similar expression for r.

We can write that:

$$P(r \leq R) = \int_{r'=0}^{R} \int_{\theta=0}^{2\pi} \frac{1}{2\pi} e^{-} r'^2 r' dr' d\theta = \int_{r'=0}^{r} e^{-} r'^2 r' dr'. \tag{15.3}$$

If we use the change of variables $\frac{1}{2} r'2 = s$ (so that $r'dr' = ds$) then:

$$P(r \leq R) = \int_{s=0}^{r^2/2} e^{-} s ds = 1 - e^{-r^2/2}. \tag{15.4}$$

So to sample r we just need to solve $1 - e^{(} - r^2/2) = 1 - U_2$ where U_2 is another uniformly distributed random variable, and which has solution $r = \sqrt{-2\ln(U_2)}$. So one algorithm to generate the Gaussian variables is:

The Box–Muller Scheme

- Pick two uniformly distributed random numbers $0 \leq U_1, U_2 \leq 1$
- Set $\theta = 2\pi U_1$ and $r = \sqrt{(} - 2\ln(U_2))$
- Then $x = r\cos(\theta)$ and $y = r\sin(\theta)$ are independent Gaussian-distributed variables with zero mean and unit variance

An alternative approach to computing these random variables is to pick the two uniform random values and scale them to lie between -1 and 1, and to interpret them as describing a point in the plane. If this point is outside the unit circle (so if the variables are U_1 and U_2 as above then if $w^2 = U_1^2 + U_2^2 > 1$) then it is discarded, and another point picked until it is within the circle. Then the transformation $x = U_1 \left(\frac{-2\ln w^2}{w^2} \right)^{\frac{1}{2}}$ and similarly for y with U_2 also provides the variables.

The difference between the methods is that one requires the computation of $\sin(\theta)$, while the other requires that some points are sampled and discarded. Which is faster depends upon the programming language and computer architecture.

A plot of 1,000 samples created by the Box–Muller scheme along with the zero mean, unit variance Gaussian line is shown in Figure 15.1. There is a more efficient algorithm for computing Gaussian-distributed random numbers known as the Ziggurat algorithm that should be investigated further if you require lower computational cost.

There may well be many other distributions that we want to sample from. For common statistical distributions people have worked out schemes like the Box–Muller scheme, but we might want to sample from distributions that we can't describe in those terms. We will see examples of this in Chapter 16. We would like a general method of sampling from any distribution that doesn't have to be tailored to the distribution. There is one important

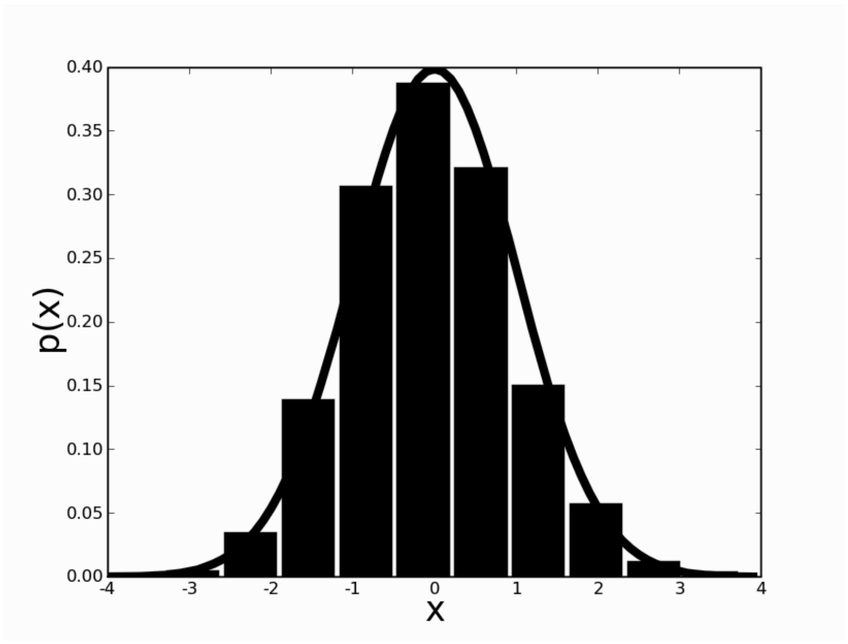

FIGURE 15.1 Histogram of 1,000 Gaussian samples created by the Box–Muller scheme. The line gives the Gaussian distribution with zero mean and unit variance.

concept that can be seen in the Box–Muller scheme, and that is the idea of rejection. When the original samples were not inside the unit circle they were rejected and another one computed to replace them. This is a bit like simulated annealing as we saw it in Section 9.6: we constructed a possible solution and then decided whether or not to use it. Rejection adds computational cost to the procedure, since if we were unlucky this algorithm could run for a long time before it found a pair of numbers that satisfied the criteria. However, it also means that we find samples that satisfy our requirements without having to design any tricky code, and it is generally faster as well, since the computational cost of generating some random numbers is rather less than the cost of doing the complicated transform.

We are going to see rejection used a lot more in this chapter, but before we get there, we should set the idea of sampling onto a proper theoretical footing.

15.2 MONTE CARLO OR BUST

Monte Carlo, a tiny principality on the Mediterranean coast between France and Italy, is famous mostly for its casino and Grand Prix race. As the rich and famous flock to lose money there, they are unlikely to know that the principality also has the dubious honour of having an important statistical principle named after it. The Monte Carlo principle states that if you take independent and identically distributed (i.e., well-behaved) samples $\mathbf{x}^{(i)}$ from an unknown high-dimensional distribution $p(\mathbf{x})$, then as the number of samples gets larger the sample distribution will converge to the true distribution. In other words, sampling works. Written mathematically, this says:

$$\begin{aligned} p_N(\mathbf{x}) &= \frac{1}{N}\sum_{i=1}^{N}\delta(\mathbf{x}^(i) = \mathbf{x}) \\ &\rightarrow \lim_{N\to\infty} p_N(\mathbf{x}) = p(\mathbf{x}), \end{aligned} \tag{15.5}$$

where $\delta(\mathbf{x}_i = \mathbf{x})$ is the Dirac delta function that is 0 everywhere except at the point $\mathbf{x}_i$ and has $\int \delta(x)dx = 1$. This can be used to compute the expectation as well (where $f(\mathbf{x})$ is some function and $\mathbf{x}$ has discrete values, and the superscript $\cdot^{(i)}$ represents the index of the sample):

$$\begin{aligned} E_N(f) &= \frac{1}{N}\sum_{i=1}^{N} f(\mathbf{x}^{(i)}) \\ &\rightarrow \lim_{N\to\infty} E_N(f) = \sum_{\mathbf{x}} f(\mathbf{x})p(\mathbf{x}). \end{aligned} \tag{15.6}$$

The fact that the sample distribution becomes more and more like the true one as we take more and more samples tells us that samples are more likely to be drawn from parts of the distribution that have high probability. This is very useful, since places where there are more samples will allow us to approximate the function well in those regions, and we only really care about the appearance of the function in those places—if the probability is small, then it doesn't matter that the number of samples is small (the area is sparsely covered) since the probability is low there anyway. If we use methods that don't know anything about the probability (such as sampling based on a uniform grid and using splines or something similar), then we have to treat all areas of the space as equally likely, which means that there is going to be a lot of computational resources wasted. There is another benefit, too. In addition to using the samples to approximate the expectation, we can also find a maximum, that is, the most likely outcome, from the samples:

$$\hat{\mathbf{x}} = \arg\max_{\mathbf{x}^{(i)}} p\left(\mathbf{x}^{(i)}\right). \tag{15.7}$$

Allegedly, the idea of Monte Carlo sampling (and the reason that it got its name) first came about when Stan Ulam was considering the probabilities of particular hands of cards. In fact, the whole of probability theory was originally developed by some of the great French mathematicians, such as Fermat, in order to reason about games of chance, so Monte Carlo sampling is in pretty good company. Suppose that you want to do something relatively simple, such as to predict how many times you should expect to win at the patience game that came free with your computer. All you need to do is work out the rules for when you win based on the initial setup, and then look at how many of these setups there are. In a standard deck there are 52 cards, so there are 52! ($\approx 8 \times 10^{67}$) different ways in which the cards can be distributed. So even before we start thinking about the specific rules for the game, we know that the number of different layouts is so large it is basically impossible to think about. Despairing of working it out, you might decide to play a couple of hands of patience and see how well you do. In fact, the Monte Carlo principle tells you that that is exactly what you should be doing. Suppose that you play ten games of patience and six of them come out. You might be able to argue that approximately 60% of the patience games will do well. To believe this, you will have to play far more than ten games with the same success rate, or course; and even then it assumes that you are a good patience player, and don't cheat.

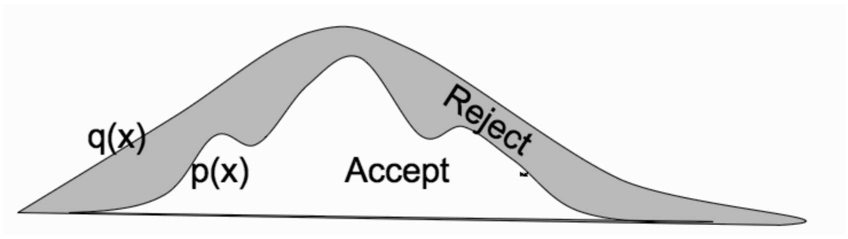

FIGURE 15.2 The proposal distribution method.

15.3 THE PROPOSAL DISTRIBUTION

We now have everything that we need if the distribution $p(\mathbf{x})$ that we are sampling from is easy (that is, not computationally expensive) to sample from. Unfortunately, this is very rarely the case, but fortunately there is a way to get around this problem, which is to cheat by inventing a simpler distribution $q(\mathbf{x})$ that we can sample from easily, and picking samples from there. Obviously we can't just pick any distribution $q(\mathbf{x})$, there has to be some relation between them. So we assume that even though we don't know $p(\mathbf{x})$, we can evaluate some related distribution $\tilde{p}(\mathbf{x})$ for a given $\mathbf{x}$, where:

$$p(\mathbf{x}) = \frac{1}{Z_p}\tilde{p}(\mathbf{x}), \tag{15.8}$$

where Z_p is some normalisation constant that we don't know. This is not usually an unreasonable assumption; we are not saying that we do not know $p(\mathbf{x})$, just that we can't sample from it easily. Now we can pick a number M so that $\tilde{p}(\mathbf{x}) \leq Mq(\mathbf{x})$ for all values of $\mathbf{x}$. We generate a random number $\mathbf{x}^*$ from $q(\mathbf{x})$, and we want this to look like a sample from $p(\mathbf{x})$. We therefore turn to the idea of rejection again, looking at how likely it is that the sample comes from $p(\mathbf{x})$, and discarding it if it turns out to be unlikely.

We make the decision of whether or not to accept the sample by picking a uniformly distributed random number u between 0 and $Mq(\mathbf{x}^*)$. If this random number is less than $\tilde{p}(\mathbf{x}^*)$, then we accept $\mathbf{x}^*$, otherwise we reject it. The reason why this works is known as the envelope principle: the pair $(\mathbf{x}^*, u)$ is uniformly distributed under $Mq(\mathbf{x}^*)$, and the rejection part throws away samples that don't match the uniform distribution on $p(\mathbf{x}^*)$, so $Mq(\mathbf{x})$ forms an envelope on $p(\mathbf{x})$. Figure 15.2 shows the idea: we sample from $Mq(\mathbf{x})$ and reject any sample that lies in the grey area. The smaller M is, the more samples we get to keep, but we need to ensure that $\tilde{p}(\mathbf{x}) \leq Mq(\mathbf{x})$. This method is known as rejection sampling, and the algorithm can be written as:

The Rejection Sampling Algorithm

- Sample $\mathbf{x}^*$ from $q(\mathbf{x})$ (e.g., using the Box–Muller scheme if $q(\mathbf{x})$ is Gaussian)
- Sample u from uniform$(0, \mathbf{x}^*)$
- If $u < p(\mathbf{x}^*)/Mq(\mathbf{x}^*)$:
  - add $\mathbf{x}^*$ to the set of samples
- Else:
  - reject $\mathbf{x}$ and pick another sample

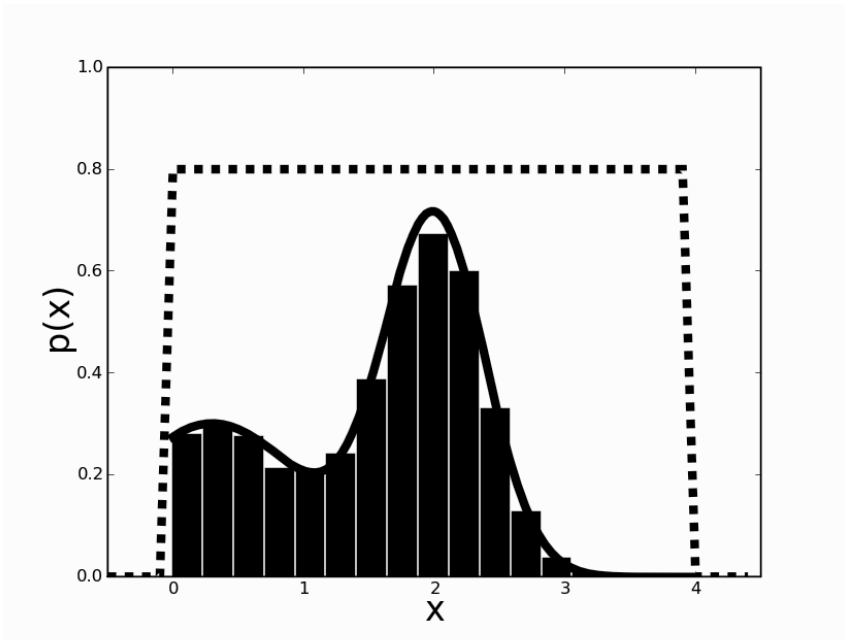

FIGURE 15.3 The histogram shows samples of a mixture of two Gaussians (given by the solid line) as sampled from the uniform box shown as a dotted line by using rejection sampling.

As an example of using rejection sampling, Figure 15.3 shows the results of using it to sample from the mixture of two Gaussians by using the uniform distribution shown by the dotted line. Using $M = 0.8$, as shown in the figure, the algorithm rejects about half of the samples. Using $M = 2$ the algorithm rejects about 85% of samples. So with rejection sampling, you have to throw away samples, and if you don't pick M properly, you will have to reject a lot of them. The curse of dimensionality makes the problem even worse. There are two things that we can do to get over this problem. One is to develop some more sophisticated methods of understanding the space that we are sampling, and the other is to try to ensure that the samples are taken from areas of the space that have high probability.

The reason why we are using these methods at all is that we can't sample from the actual distribution we want, since that is too difficult and/or expensive, but it might be possible to understand it in other ways. In Section 15.4.1 we will look at methods that allow us to travel around within the space by using simple local moves. Before we get to that we will look at a method that tries to ensure that the samples come from regions of high probability. The method is known as importance sampling, because it attaches a weight that says how important each sample is.

Suppose that we want to compute the expectation of a function $f(\mathbf{x})$ for a continuous random variable $\mathbf{x}$ distributed according to unknown distribution $p(\mathbf{x})$. Starting from the expression of the expectation that we wrote out earlier, we can introduce another distribution $q(\mathbf{x})$:

$$\begin{aligned} E(f) &= \int p(\mathbf{x}) f(x)\, d\mathbf{x} \\ &= \int p(\mathbf{x}) f(\mathbf{x}) \frac{q(\mathbf{x})}{q(\mathbf{x})} d\mathbf{x} \\ &\approx \frac{1}{N} \sum_{i=1}^{N} \frac{p\left(\mathbf{x}^{(i)}\right)}{q\left(\mathbf{x}^{(i)}\right)} f\left(x^{(i)}\right), \end{aligned} \tag{15.9}$$

where we have used the fact that $q(\mathbf{x})$ is the density of a random variable, and so if we perform $\int q(\mathbf{x})d\mathbf{x}$ over all values of $\mathbf{x}$, then it must equal 1. The ratio $w(\mathbf{x}^{(i)}) = p(\mathbf{x}^{(\mathbf{i})})/q(\mathbf{x}^{(\mathbf{i})})$ is called the importance weight, and it corrects for sampling from the grey region in Figure 15.2 without having to reject samples. While this can be used to estimate the expectation directly, the real benefit of computing the importance weights is that they can be used in order to resample the data. This leads to an algorithm known descriptively as Sampling-Importance-Resampling. In the words of the advert, it 'does exactly what it says on the tin':

The Sampling-Importance-Resampling Algorithm

- Produce N samples $\mathbf{x}^{(i)},\ i = 1 \ldots N$ from $q(\mathbf{x})$
- Compute normalised importance weights

$$w^{(i)} = \frac{p(\mathbf{x}^{(i)})/q(\mathbf{x}^{(i)})}{\sum_j p(\mathbf{x}^{(j)})/q(\mathbf{x}^{(j)})} \tag{15.10}$$

- Resample from the distribution $\{\mathbf{x}^{(i)}\}$ with probabilities given by the weights $w^{(i)}$

An implementation of this in Python is shown next, and the results of using sampling-importance-resampling on the example in Figure 15.3 are given in Figure 15.4. Note that this method does not reject any samples, but it does involve two separate sampling steps and a relatively expensive loop. Like the other algorithms we have seen, it is sensitive to the quality of the match between the proposal distribution $q(\mathbf{x})$ and the actual distribution $p(\mathbf{x})$.

```
# Sample from q
sample1 = np.random.rand(n)*4

# Compute weights
w = p(sample1)/q(sample1)
w /= np.sum(w)

# Sample from sample1 according to w
cumw = np.zeros(n)
cumw[0] = w[0]
for i in range(1,n):
    cumw[i] = cumw[i-1]+w[i]
```

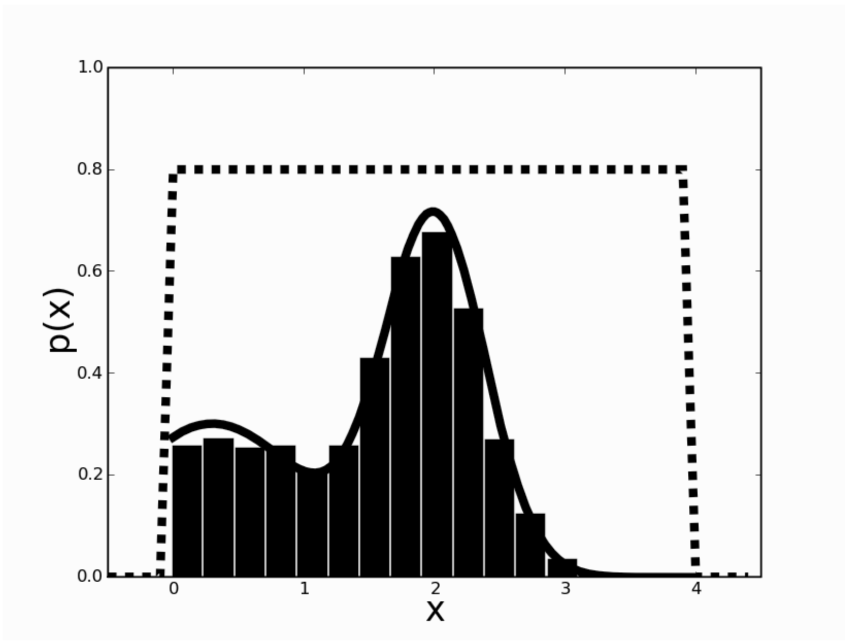

FIGURE 15.4 The histogram shows samples created using sampling-importance-resampling from a mixture of two Gaussians (given by the solid line) as sampled from the uniform box shown as a dotted line.

```
u = np.random.rand(n)

index = 0
for i in range(n):
    indices = np.where(u<cumw[i])
    sample2[index:index+size(indices)] = sample1[i]
    index += np.size(indices)
    u[indices]=2
```

In Section 16.4.2 we will see a method that uses sampling-importance-resampling in an on-line application, known as a particle filter or sequential Monte Carlo method. However, we will first turn our attention to how we can find out more about the sample space. The basic idea is to keep track of the sequence of samples and modify the proposal distribution to take advantage of this, for which we will have to use some more complicated machinery.

15.4 MARKOV CHAIN MONTE CARLO

15.4.1 Markov Chains

In probabilistic terms a chain is a sequence of possible states, where the probability of being in state s at time t is a function of the previous states. A Markov chain is a chain with the Markov property, i.e., the probability at time t depends only on the state at $t-1$, as discussed in Section 11.3. The set of possible states are linked together by transition probabilities that say how likely it is that you move from the current state to each of the others, and they are generally written as a matrix T. They might be constant, or functions of some other variables, but here we will assume that they are constant. Note that, unlike the Markov Decision Processes that we saw in Section 11.3, there is no action here that affects the probability of moving into a particular state.

Given a chain, we can perform a random walk on the chain by choosing a start state and randomly choosing each successive state according to the transition probabilities. The link to sampling that we need is that if the transition probabilities reflect the distribution that we wish to sample from, then a random walk will explore that distribution. One problem with this is that random walks are very inefficient at exploring space, since they move back towards the start as often as they move away, which means the distance they move from the start scales as $\sqrt{t}$, where t is the number of samples. We therefore want to explore more efficiently than just using a random walk.

We do this by setting up our Markov chain so that it reflects the distribution we wish to sample from, and we want the distribution $p(\mathbf{x}^{(i)})$ to converge to the actual distribution $p(\mathbf{x})$ no matter what state we start from. Since we can start from any state, this tells us that every state is reachable from every other state, which means that the chain is irreducible so that the transition matrix can't be cut up into smaller matrices. The chain also has to be ergodic, which means that we will revisit every state, so that the probability of visiting any particular state in the future never goes to zero, but is not periodic, which means that we can visit at any time, not just every k iterations for some constant k.

We also want the distribution $p(\mathbf{x})$ to be invariant to the Markov chain, which means that the transition probabilities don't change the distribution:

$$p(\mathbf{x}) = \sum_{\mathbf{y}} T(\mathbf{y}, \mathbf{x}) p(\mathbf{y}). \tag{15.11}$$

Finding the transition probabilities to make this true requires that we can move backwards and forwards along the chain with equal probability, so that the chain is reversible. This says that the probability of being in an unlikely state s (sampling datapoint $\mathbf{x}$), but heading for a likely state s' (datapoint $\mathbf{x}'$) should be the same as being in the likely state s' and heading for the unlikely state s, so that:

$$p(\mathbf{x})T(\mathbf{x}, \mathbf{x}') = p(\mathbf{x}')T(\mathbf{x}', \mathbf{x}). \tag{15.12}$$

This is known as the detailed balance condition and the fact that it leaves the distribution $p(\mathbf{x})$ alone is fairly obvious with a little calculation. If the chain satisfies the detailed balance condition, then it must be ergodic, since $\sum_{\mathbf{y}} T(\mathbf{x}, \mathbf{y}) = 1$, since you must have come from some state, and so:

$$\sum_{\mathbf{y}} p(\mathbf{y}) T(\mathbf{y}, \mathbf{x}) = p(\mathbf{x}), \tag{15.13}$$

which means that $p(\mathbf{x})$ must be an invariant distribution of T. So if we can work out how to construct a Markov chain with detailed balance we can sample from it in order to sample from our distribution. This is known as Markov Chain Monte Carlo (MCMC) sampling, and the most popular algorithm that is used for MCMC is the Metropolis–Hastings algorithm after the two people who were directly involved in its creation.

15.4.2 The Metropolis–Hastings Algorithm

We assume that we have a proposal distribution of the form $q(\mathbf{x}^{(i)}|\mathbf{x}^{(i-1)})$ that we can sample from. The idea of Metropolis–Hastings is similar to that of rejection sampling: we take a sample $\mathbf{x}^*$ and choose whether or not to keep it. Except, unlike rejection sampling, rather than picking another sample if we reject the current one, instead we add another copy of the previous accepted sample. Here, the probability of keeping the sample is $u(\mathbf{x}^*|\mathbf{x}^{(i-1)})$:

$$u(\mathbf{x}^*|\mathbf{x}^{(i)}) = \min\left(1, \frac{\tilde{p}(\mathbf{x}^*)q(\mathbf{x}^{(i)}|\mathbf{x}^*)}{\tilde{p}(\mathbf{x}^{(i)})q(\mathbf{x}^*|\mathbf{x}^{(i)})}\right). \tag{15.14}$$

The Metropolis–Hastings Algorithm

```
• Given an initial value $x_0$
• Repeat
    – sample $\mathbf{x}^*$ from $q(\mathbf{x}_i|\mathbf{x}_{i-1})$
    – sample $u$ from the uniform distribution
    – if $u <$ Equation (15.14):
        ∗ set $\mathbf{x}[i+1] = \mathbf{x}^*$
    – otherwise:
        ∗ set $\mathbf{x}[i+1] = \mathbf{x}[i]$
• Until you have enough samples
```

So why does this algorithm work? Each step involves using the current value to sample from the proposal distribution. These values are accepted if they move the Markov chain towards more likely states, and because the Markov chain is reversible (since it satisfies the detailed balance condition) the algorithm explores states that are proportional to the difficult distribution $p(x)$.

The Python implementation is still very simple:

```python
u = np.random.rand(N)
y = np.zeros(N)
y[0] = np.random.normal(mu,sigma)
for i in range(N-1):
    ynew = np.random.normal(mu,sigma)
    alpha = min(1,p(ynew)*q(y[i])/(p(y[i])*q(ynew)))
    if u[i] < alpha:
        y[i+1] = ynew
    else:
        y[i+1] = y[i]
```

The Metropolis–Hastings (and variants of it) are by far the most commonly used MCMC methods, and it is also the most general. It requires that you choose the proposal distribution $q(x^*|x)$ carefully, but it is a very simple algorithm to use. Figure 15.5 shows 5,000

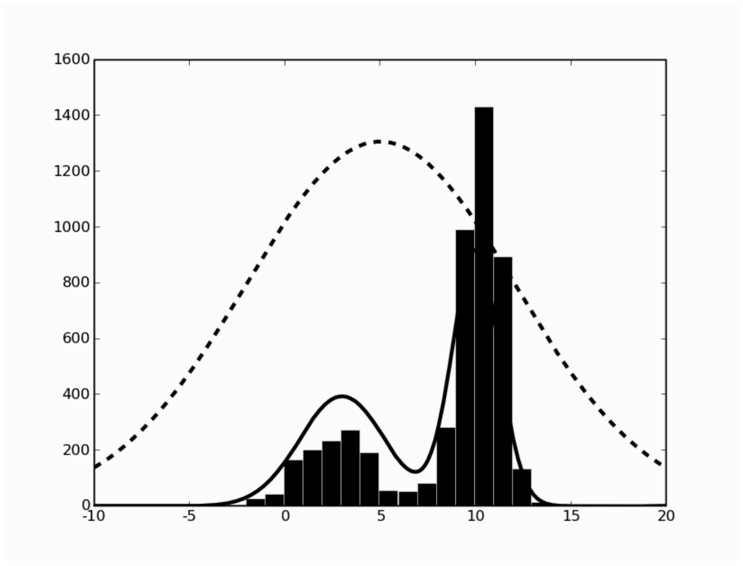

FIGURE 15.5 The results of the Metropolis–Hastings algorithm when the true distribution is a mixture of two Gaussians (shown by the solid line) and the proposal distribution is a single Gaussian (the dotted line).

samples computed using the algorithm on a mixture of two Gaussians based on a proposal distribution that is a single Gaussian.

Note that if the proposal distribution is symmetric, then it drops out of the test in Equation (15.14). This is the original Metropolis algorithm, and it is much closer to the pure random walk. The results of using this algorithm on the same data can be seen in Figure 15.6.

There are other choices of proposal distribution, and they lead to variants on the Metropolis–Hastings algorithm. We will consider the two most common choices next.

15.4.3 Simulated Annealing (Again)

There are lots of times when we might just want to find the maximum of a distribution rather than approximate the distribution itself. We can do this in calculating $\arg\max_{\mathbf{x}^{(i)}} p(\mathbf{x}^{(i)})$ (that is, the $\mathbf{x}^{(i)}$ with the largest probability), but while doing this we will have computed samples from many parts of the space, not just around the maximal region. A possible solution is to use simulated annealing as we did in Section 9.6. This changes the Markov chain so that its invariant distribution is not $p(\mathbf{x})$, but rather $p^{1/T_i}(\mathbf{x})$, where $T_i \to 0$ as $i \to \infty$. We need an annealing schedule that cools the system down over time so that we are progressively less likely to accept solutions that are worse over time.

There are only two modifications needed to the Metropolis–Hastings algorithm, and both are trivial: we extend the acceptance criterion to include the temperature and add a line into the loop to include the annealing schedule. The results of using simulated annealing on the example where the true distribution is a mixture of two Gaussians and the proposal distribution is just one is shown in Figure 15.7.

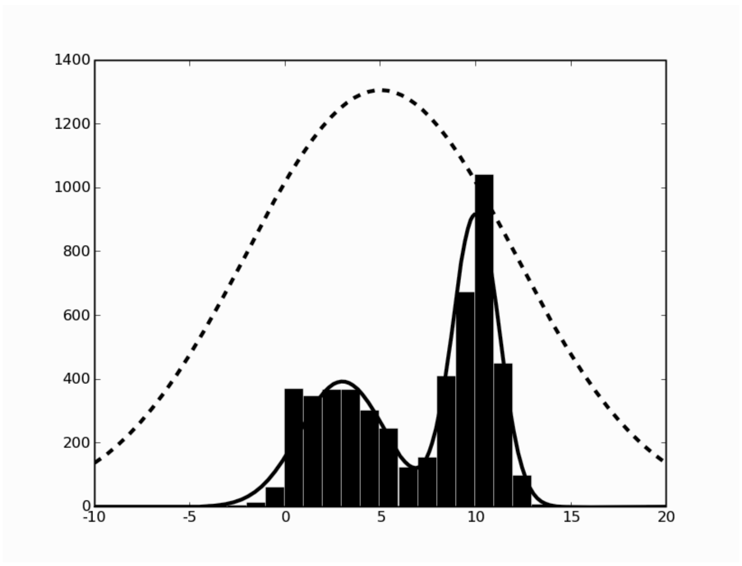

FIGURE 15.6 The results of the Metropolis algorithm when the true distribution is a mixture of two Gaussians (shown by the solid line) and the proposal distribution is a single Gaussian (the dotted line).

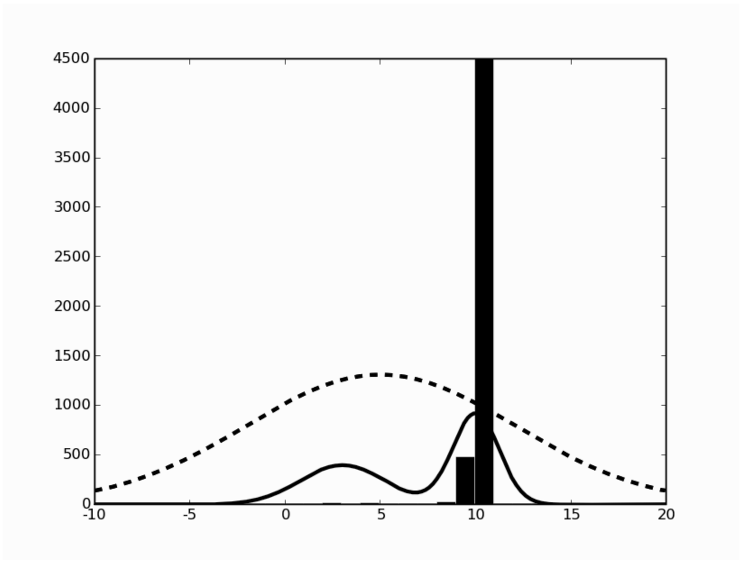

FIGURE 15.7 Using simulated annealing gives the maximum rather than an approximation to the distribution, as is shown here for the same example as in Figures 15.5 and 15.6.

15.4.4 Gibbs Sampling

Another variation on the Metropolis–Hastings algorithm comes when we already know the full conditional probability $p(x_j|x_1, \dots x_{j-1}, x_{j+1}, \dots x_n)$ (which is often written as $p(x_j|x_{-j})$ for convenience). We are going to see some examples of this in the next chapter: Bayesian networks. In Section 16.1.2 we will deal with a set of probabilities from a network that looks like:

$$p(\mathbf{x}) = \prod_j p(x_j|x_{\alpha j}), \tag{15.15}$$

where $x_{\alpha j}$ is the parents of x_j (as will become clear in that section).

Given that we know $p(x_j|x_{\alpha j}) \prod_{k \in \beta(j)} p(x_k|x_{\alpha(k)})$ (which is $p(x_j|x_{-j})$), maybe we should try using it as the proposal distribution, giving:

$$q(x^*|x^{(i)}) = \begin{cases} p\left(x_j^*, x_{-j}^{(i)}\right) & \text{if } x_{-j}^* = x_{-j}^{(i)} \\ 0 & \text{otherwise.} \end{cases} \tag{15.16}$$

If we then use Metropolis–Hastings, we find that the acceptance probability P_a is:

$$P_a = \min\left\{1, \frac{p(x^*)p(x_j^{(i)}|x_{-j}^{(i)})}{p(x^{(i)})p(x_j^*|x_{-j}^*)}\right\}, \tag{15.17}$$

and looking carefully at this and expanding out the conditional probabilities we get:

$$P_a = \min\left\{1, \frac{p(x^*)p(x_j^{(i)}, x_{-j}^{(i)})p(x_{-j}^{(i)})}{p(x^{(i)})p(x_j^*, x_{-j}^*)p(x_{-j}^*)}\right\}. \tag{15.18}$$

Since $p(x_j^*, x_{-j}^*) = p(x^*)$, and similarily for $p^{(i)}$, we only have to worry about $\frac{p(x^{(i)})}{p(x_{-j}^*)}$. From the definition of the proposal distribution we know that $x*_{-j} = x_{-j}^{(i)}$, and so the computation is actually $\min 1, 1 = 1$. So we always accept the proposal, which makes things much simpler.

The total algorithm is given by choosing each variable and sampling from its conditional distribution. That's it! The only option that you have is whether to go through the variables in order, or whether to update them in a random order. Rather than running up to some maximum value N, it is not uncommon to run until the joint distribution stops changing. This algorithm is known as the Gibbs sampler and it forms the basis of the software package BUGS (Bayesian Updating with Gibbs Sampling) that is commonly used in statistics. It is also a very useful algorithm for Bayesian networks, as we shall see in the next chapter.

The Gibbs Sampler

```
• For each variable $x_j$:
    – initialise $x_j^{(0)}$
• Repeat
    – for each variable $x_j$:
        ∗ sample $x_1^{(i+1)}$ from $p(x_1|x_2^{(i)}, \dots x_n^{(i)})$
        ∗ sample $x_2^{(i+1)}$ from $p(x_2|x_1^{(i+1)}, x_3^{(i)}, \dots x_n^{(i)})$
```

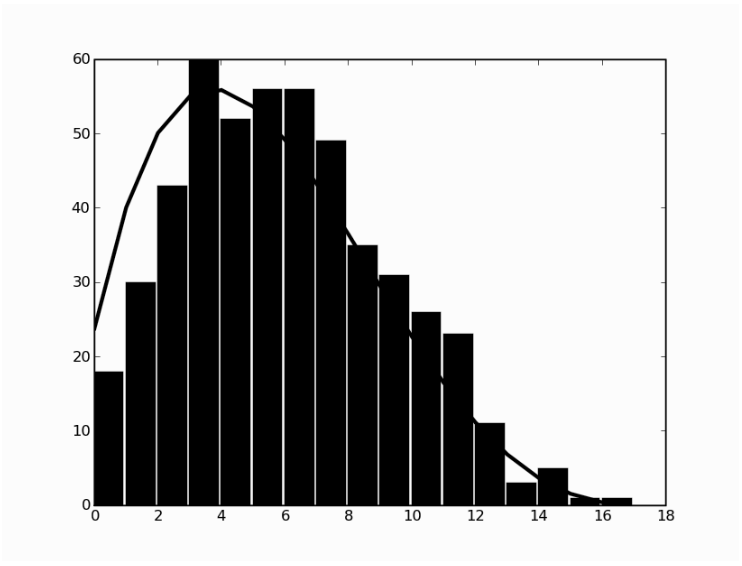

FIGURE 15.8 The Gibbs sampler output for the beta-binomial distribution.

  * ...
  * sample $x_n^{(i+1)}$ from $p(x_n|x_1^{(i+1)}, \ldots x_{n-1}^{(i+1)})$

- Until you have enough samples

As an example, suppose that we have a distribution that is made up of two different distributions, a binomial one in x and a beta in y. If you don't know what these distributions are, the combined distribution can be written as:

$$p(x, y, n) = \left(\frac{n!}{x!(n-x)!}\right) y^{x+\alpha-1} + (1-y)^{n-x+\beta-1}. \tag{15.19}$$

The important point is that the overall distribution is a product of two separate ones that can be sampled from separately. Figure 15.8 shows the output of the sampling using the Gibbs sampler, with the line being the correct distribution as usual. There is another example of Gibbs sampling in Section 16.1.2.

FURTHER READING

The historical perspective in this area is provided by:

- N. Metropolis and S. Ulam. The Monte Carlo method. *Journal of the American Statistical Association*, 44(247):335–341, 1949.

Since MCMC is a very useful, but fairly difficult area, there is a good number of review and tutorial articles available. Some that you may find helpful are:

- W.R. Gilks, S. Richardson, and D.J. Spiegelhalter, editors. *Markov Chain Monte Carlo in Practice.* Chapman & Hall, London, UK, 1996.

- C. Andrieu, C. de Freitas, A. Doucet, and M. Jordan. An introduction to MCMC for machine learning. *Machine Learning*, 50:5–43, 2003.
- G. Casella and E.I. George. Explaining the Gibbs sampler. *The American Statistician*, 46(3):167–174, 1992.
- Chib. S. and E. Greenberg. Understanding the Metropolis-Hastings algorithm. *The American Statistician*, 49(4):327–335, 1995.

There is also a more complete treatment of sampling methods in:

- Chapter 11 of C.M. Bishop. *Pattern Recognition and Machine Learning.* Springer, Berlin, Germany, 2006.

PRACTICE QUESTIONS

Problem 15.1 Implement the alternative algorithm for the Box–Muller scheme and compare their times on your computer.

Problem 15.2 Use rejection sampling and importance sampling to sample from a Gaussian distribution using a uniform distribution as the proposal distribution. How many samples do you have to reject with the rejection sampler?

Problem 15.3 The Gibbs sampler can be used in place of the EM algorithm in order to fit the mixtures of a Gaussian Mixture Model (Section 7.1). The idea is to use the samples to introduce the mixing variable π as we did then, and to use the Gibbs sampler to sample from the current estimates of the Gaussians. The algorithm will then look something like:

The Gibbs Sampler for Gaussian Mixtures

```
• Given some estimates of $\mu_1$, $\mu_2$
• Repeat until the distribution stops changing:
    – for i = 1 to N:
        ∗ sample $\pi$ according to the E-step of the EM algorithmEM algorithm
        ∗ update:
              $$\hat{\mu}_i = \frac{\sum_{i=1}^{N}(1-\pi_i^{(t)})x_i}{\sum_{i=1}^{N}(1-\pi_i^{(t)})}.$$   (15.20)
        ∗ sample from the Gaussians with these estimates in order to produce new
          estimates of the means
```

Implement this and compare the results to using the EM algorithm.

Problem 15.4 Show that the Gibbs sampler satisfies the detailed balance equation.

Problem 15.5 Modify the Metropolis–Hastings algorithm in order to resample when it rejects the current sample. How does it affect the results? Explain the result in terms of the effect on the Markov chain.

CHAPTER 16

Graphical Models

Throughout this book we have seen that machine learning brings together computer science and statistics. Nowhere is this more clearly shown than in one of the most popular areas of current research in machine learning: graphical models (or more completely, probabilistic graphical models), which use graph theory with all its underlying computational and mathematical machinery in order to explain probabilistic models.

The graphs used in graphical models are the exact ones that are taught in basic algorithms classes: a set of nodes, together with links between them, which can be either directed (i.e., have arrows on them so that you can only go one way along them) or not. There are two basic types of graphical models, depending upon whether or not the edges are directed. We will focus primarily on directed graphs, but the undirected kind (known as Markov Random Fields) are described in Section 16.2. For such a simple data structure, graphs have turned out to be incredibly powerful in many different parts of computer science, from constructing compilers to managing computer networks. For this reason, there are lots of readily available algorithms for finding shortest paths (Floyd's and Djiksta's algorithms, which we've already discussed briefly in Section 6.6), determining cycles, etc. Any good book on algorithms will give details of these and many other graph algorithms.

For our part, we are interested in using graphs to encode probability distributions and so we need to decide what nodes and links are in this context. The nodes are fairly obvious. We generate a node for each random variable, and label it accordingly. In this book, we will only consider discrete variables, so that there is a finite number of possible values that the random variable can take. Given a continuous variable we will discretise it into a finite set. While this loses information, it makes the problem much simpler. The alternative is to specify the variable by a probability density function, which can be done, but makes the whole thing harder to describe and understand.

The question is what to make the links represent. Perhaps the best way to think about this is to ask what it means if two nodes are not linked. In this case we are saying that there is no connection between those two variables, which is the same as saying that they are independent. Except it isn't quite as simple as that, because two nodes could be linked through a third node. Have a look at the right of Figure 16.1, where C is not directly linked to B, but there is a link through A. For this reason we have to be careful and talk about conditional independence: C is conditionally independent of B, given A.

We use directed links because these relationships are not symmetrical (unless the variables are independent, in which case there is no link). What does the simplest connected graph that we can make, the one on the left of Figure 16.1, mean? There is a rather loose interpretation of the link, which is to say that 'A' causes 'B' (but note that this isn't quite the same semantic usage that we normally have for 'causes', since there may be several

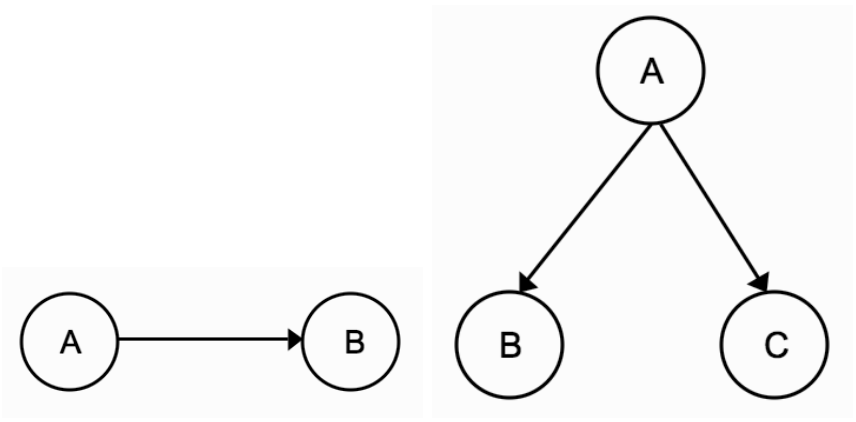

FIGURE 16.1 Two simple graphical models. The arrows denote causal relationships between nodes that represent features.

variables that are all involved in causing B). This is a useful intuition to have, but it is not really correct. More properly, the graph tells us that the probability of A and B is the same as the probability of A times the probability of B conditioned on A: $P(a, b) = P(b|a)P(a)$. If there is no direct link between two nodes then they are conditionally independent of each other.

There is a third thing that we need in order to specify the problem properly, which is the conditional probability table for each variable. This specifies what the probabilities are for each of the nodes, conditioned on any nodes that are its parents.

If we wanted to work out a value for $P(a, b)$, then we would need a distribution table for $P(a)$ and one for $P(b|a)$. The nodes are separated into those where we can see their values directly—observed nodes—and hidden or latent nodes, whose values we hope to infer, and which may not have clear meanings in all cases.

The basic concept of the graphical model is very simple, which makes it all the more amazing that it produces a powerful set of tools for understanding and creating machine learning algorithms. We will start by looking at the most general model, the Bayesian Belief Network or more simply, Bayesian Network, and see how they are represented, and the difficulties involved in dealing with them. Following this, we will identify a few places where these difficulties can be overcome, resulting in some very important algorithms that solve a variety of different tasks. In particular, we will look at Markov Random Fields (MRFs), Hidden Markov Models (HMMs), the Kalman Filter, and particle filter.

16.1 BAYESIAN NETWORKS

To start with, we will consider directed graphs, and make one restriction to them, namely that they must not contain cycles, that is, there cannot be any loops in the graphs. These graphs go by the rather unlovely name of DAGs: directed, acyclic graphs, but for graphical models, when they are paired with the conditional probability tables, they are called Bayesian networks. In order to see what we can do with such a network, we need an example.

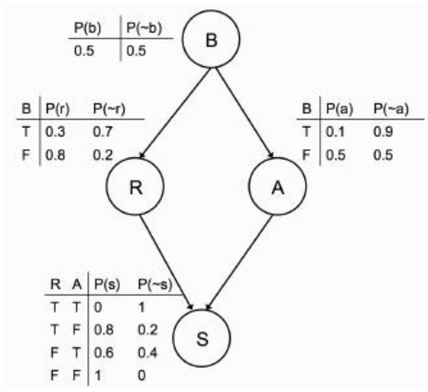

FIGURE 16.2 The sample graphical model. 'B' denotes a node stating whether the exam was boring, 'R' whether or not you revised, 'A' whether or not you attended lectures, and 'S' whether or not you will be scared before the exam.

16.1.1 Example: Exam Fear

Figure 16.2 shows a graph with a full set of distribution tables specified. It is a handy guide to whether or not you will be scared before an exam based on whether or not the course was boring ('B'), which was the key factor you used to decide whether or not to attend lectures ('A') and revise ('R'). We can use it to perform inference in order to decide the likelihood of you being scared before the exam ('S'). There are two kinds of inferences, depending on whether the observations that are made come from the top of the graph or the bottom. If we have a set of observations that can be used to predict an unknown outcome, then we are doing top-down inference or prediction, whereas if the outcome is known, but the causes are hidden, then we are doing bottom-up inference or diagnosis. Either way, we are working out the values of the hidden (unknown) nodes given information about the observed nodes. For the example in Figure 16.2 we will start by predicting whether or not you will be scared before the exam, so it is the outcome that is hidden.

In order to compute the probability of being scared, we need to compute $P(b, r, a, s)$, where the lower-case letters indicate particular values that the upper-case variables can take. The wonderful thing about the graphical model is that we can read the conditional probabilities from the graph—if there is no direct link, then variables are conditionally independent given a node that is already included, so those variables are not needed. For this reason, the computation we need for Figure 16.2 is:

$$\begin{aligned} P(s) &= \sum_{b,r,a} P(b,r,a,s) \\ &= \sum_{b,r,a} P(b) \times P(r|b) \times P(a|b) \times P(s|r,a) \\ &= \sum_{b} P(b) \times \sum_{r,a} P(r|b) \times P(a|b) \times P(s|r,a). \end{aligned} \tag{16.1}$$

If we know particular values for the three observable nodes, then we can plug them in and work out the probability. In fact, the conditional independence gives us even more: if I know both whether or not you attended lectures and whether or not you revised, then I don't need to know if the course was boring, since there is no direct connection between 'B' and 'S'. Suppose that you didn't attend lectures, but did revise. In that case, the probability of you being scared can be read off the final distribution table as 0.8. The power of the graphical model is when you don't have full information. It is possible to marginalise over any of those variables by summing up the values. So suppose that you know that the course was boring, and want to work out how likely it is that you will be scared before the exam. In that case you can ignore the $P(b)$ terms, and just need to sum up the probabilities for r and a using Equation (16.1):

$$\begin{aligned} P(s) &= 0.3 \times 0.1 \times 0 + 0.3 \times 0.9 \times 0.8 + 0.7 \times 0.1 \times 0.6 + 0.7 \times 0.9 \times 1 \\ &= 0.328. \end{aligned} \tag{16.2}$$

The backwards inference, or diagnosis, can also be useful. Suppose that I see you looking very scared outside the exam. You look vaguely familiar, but I'm not sure whether or not you came to the lectures. I might want to work out why you are scared—was it because you didn't come to the lectures, or because you didn't revise? To perform this calculation I need to use Bayes' rule to turn the conditional probabilities around, just as was done for the Bayes' classifier in Chapter 7. So the computations that I need are (where $P(s)$ is the normalising constant found by summing over all values of r, a, and b, i.e., Equation (16.2)):

$$\begin{aligned} P(r|s) &= \frac{P(s|r)P(r)}{P(s)} \\ &= \frac{\sum_{b,a} P(b,a,r,s)}{P(s)} \\ &= \frac{0.5 \cdot (0.3 \cdot 0.1 \cdot 0 + 0.3 \cdot 0.9 \cdot 0.8) + 0.5 \cdot (0.8 \cdot 0.5 \cdot 0 + 0.8 \cdot 0.5 \cdot 0.8)}{P(s)} \\ &= \frac{0.268}{0.684} = 0.3918. \qquad (16.3) \\ P(a|s) &= \frac{P(s|a)P(a)}{P(s)} \\ &= \frac{0.144}{0.684} = 0.2105. \qquad (16.4) \end{aligned}$$

This use of Bayes' rule is the reason why this type of graphical model is known as a Bayesian network. Even in this very simple example, the inference was not trivial, since there were a lot of calculations to do. However, the problem is actually rather worse than that. The computational cost of the simple algorithm we used (start at the root, and follow each link through the graph to perform the computation) is $\mathcal{O}(2^N)$ for a graph with N nodes where each node can be either true or false. In general the problem of exact inference on Bayesian networks is NP-hard (technically, it is actually #P-hard, which is even worse). However, for so-called polytrees where there is at most one path between any two nodes, the computational cost is much smaller—linear in the size of the network.

Unfortunately, it is rare to find such polytrees in real examples, so we can either try to turn other networks into polytrees, or consider only approximate inference, which is the most common solution to the problem, and the method that we'll consider next. We can speed things up a little by getting things into the form of Equation (16.1), where the summations were carefully placed as far to the right as possible, so that program loops can be minimised. By doing this the algorithm is as efficient as possible, but it is still NP-hard.

This is sometimes known as the variable elimination algorithm, which is a variation on the bucket elimination algorithm. The idea is to convert the conditional probability tables into what are called λ tables, which simply list all of the possible values for all variables, and which initially contain the conditional probabilities. For example, the λ table for the 'S' variable in Figure 16.2 is:

R	A	S	λ
T	T	T	0
T	T	F	1
T	F	T	0.8
T	F	F	0.2
F	T	T	0.6
F	T	F	0.4
F	F	T	1
F	F	F	0

If I see you looking scared outside the exam (so that S is true), then I can eliminate it from the graph by removing from each table all rows that have S false in them, and deleting the S column. This simplifies things a little, but I have to do rather more in order to compute the probability of you having attended lectures. I don't know whether you revised or not, and I don't know if you found the lectures boring, so I have to marginalise over these variables. The order in which we marginalise doesn't change the correctness (although more advanced algorithms can improve the speed by taking advantage of conditional independence) so we'll pick R first. To eliminate it from the graph, we have to find all of the λ tables that contain it (there will be two of them containing R: the one for R itself and the one that we have just modified to remove S). To remove R, we have to add together the products of the λ values that correspond to places where the other values match. So to complete the entry where B is true and A is false, we have to multiply together the values where B, A, R are respectively true, false, true in the two tables and then add to that the product of where B, A, R are respectively true, false, false. In other words:

$$\left(\begin{array}{cc|c} B & R & \lambda \\ T & T & 0.3 \\ T & F & 0.7 \\ F & T & 0.8 \\ F & F & 0.2 \end{array}\right) \times \left(\begin{array}{cc|c} R & A & \lambda \\ T & T & 0 \\ T & F & 0.8 \\ F & T & 0.6 \\ F & F & 1 \end{array}\right) \Rightarrow \left(\begin{array}{cc|c} B & A & \lambda \\ T & T & 0.3 \cdot 0 + 0.7 \cdot 0.6 = 0.42 \\ T & F & 0.3 \cdot 0.8 + 0.7 \cdot 1 = 0.94 \\ F & T & 0.8 \cdot 0 + 0.2 \cdot 0.6 = 0.12 \\ F & F & 0.8 \cdot 0.8 + 0.2 \cdot 1 = 0.84 \end{array}\right) \tag{16.5}$$

We can do the same thing in order to eliminate B, which involves all three of the tables, and this will enable the computation of the conditional probability of you attending lectures given that I saw you looking scared before the exam. The benefit of doing things this way is that the whole thing can be written as a general algorithm:

The Variable Elimination Algorithm

- Create the λ tables:
  - for each variable v:
    - make a new table
    - for all possible true assignments x of the parent variables:
      - add rows for $P(v|x)$ and $1 - P(v|x)$ to the table
    - add this table to the set of tables

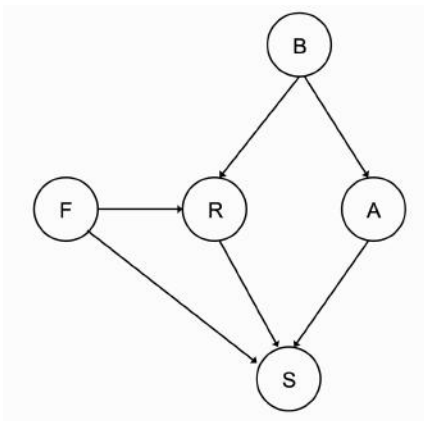

FIGURE 16.3 Adding just one extra node ('F', information about whether or not this is your final year) makes the conditional probability tables significantly more complicated.

```
• Eliminate known variables $v$:
    – for each table:
        * remove rows where $v$ is incorrect
        * remove column for $v$ from table
• Eliminate other variables (where $x$ is the variable to keep):
    – for each variable $v$ to be eliminated:
        * create a new table $t'$
        * for each table $t$ containing $v$:
            · $v_{\text{true},t} = v_{\text{true},t} \times P(v|x)$
            · $v_{\text{false},t} = v_{\text{false},t} \times P(\neg v|x)$
        * $v_{\text{true},t'} = \sum_t (v_{\text{true},t})$
        * $v_{\text{false},t'} = \sum_t (v_{\text{false},t})$
    – replace tables $t$ with the new one $t'$
• Calculate conditional probability:
    – for each table:
        * $x_{\text{true}} = x_{\text{true}} \times P(x)$
        * $x_{\text{false}} = x_{\text{false}} \times P(\neg x)$
        * probability is $x_{\text{true}}/(x_{\text{true}} + x_{\text{false}})$
```

To see that these algorithms do not scale well, consider Figure 16.3, which shows a very simple development of the example in Figure 16.2 by adding just one extra node to the network: whether or not this is your final year ('F'). This makes the network significantly more complicated, since we need another table and extra entries in two of the other tables, and therefore the variable elimination algorithm will take rather longer to run.

16.1.2 Approximate Inference

Since the variable elimination algorithm will only take you so far, for reasonably sized Bayesian networks there is no choice but to perform approximate inference. Fortunately, we have already seen a set of algorithms that are ideally suited to the problem: the Markov Chain Monte Carlo methods that we saw in Chapter 15. There are two other methods of doing approximate inference (loopy belief propagation and mean field approximation), but we will not consider them further; there are references to descriptions of these methods at the end of the chapter.

The basic idea of using MCMC methods in Bayesian networks is to sample from the hidden variables, and then (depending upon the MCMC algorithm employed) weight the samples by their likelihoods. Creating the samples is very easy: for prediction, we start at the top of the graph and sample from each of the known probability distributions. Using Figure 16.2 again, we generate a sample from $P(b)$, and then use that value in the conditional probability tables for 'R' and 'A' to compute $P(r|b = \text{sample value})$ and $P(a|b = \text{sample value})$. These three values are then used to sample from $P(p|b, a, r)$. We can take as many samples as we like in this way, and expect that as the number of samples gets large, so the frequency of specific samples will converge to their expected values.

In this sampling method, we have to work through the graph from top to bottom and select rows from the conditional probability table that match the previous case. This is not what we would do if we were constructing the table by hand. Suppose that you wanted to know how many courses you did not attend the lectures for because the course was boring. You would simply look back through your courses and count the number of boring courses where you didn't go to lectures, ignoring all the interesting courses. We can use exactly this idea if we use rejection sampling (see Section 15.3). The method samples from the unconditional distribution and simply rejects any samples that don't have the correct prior probability. It means that we can sample from each distribution independently, and then throw away any samples that don't match the other variables. This is obviously computationally easier, but we might have to reject a lot of samples.

The solution to this problem is to work out what evidence we already have and use this evidence to assign likelihoods to the other variables that are sampled. Suppose that we sample from $P(b)$ and get value 'true'. If we already know that we did revise, then we weight the observation $P(r|b)$ by the appropriate probability, which is 0.3. We continue through the other variables, sampling where there is no evidence and using the tables to find the probability if we do have evidence. However, we can do rather better than this by using the full MCMC framework. We start by setting values for all of the possible probabilities, based on either evidence or random choices. This gives us an initial state for a Markov chain. Now Gibbs sampling (Section 15.4.4) will find us the maxima of our probability distribution given enough samples.

The probabilities in the network are:

$$p(x) = \prod_j p(x_j|x_{\alpha j}), \tag{16.6}$$

where $x_{\alpha j}$ are the parent nodes of x_j. In a Bayesian network, any given variable is independent of any node that is not their child, given their parents. So we can write:

$$p(x_j|x_{-j}) = p(x_j|x_{\alpha j}) \prod_{k \in \beta(j)} p(x_k|x_{\alpha(k)}), \tag{16.7}$$

where $\beta(j)$ is the set of children of node x_j and x_{-j} signifies all values of x_i except x_j. For

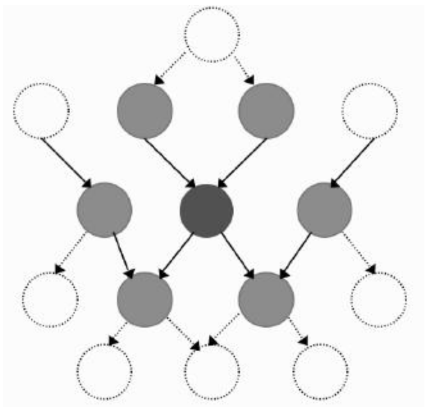

FIGURE 16.4 The Markov blanket of a node is the set of nodes (shaded light grey) that are either parents or children of the node, or other parents of its children (shaded dark grey).

any node we only need to consider its parents, its children, and the other parents of the children, as shown in Figure 16.4. This set is known as the Markov blanket of a node.

Given these calculations, computing the inference on any real Bayesian network generally consists of using Gibbs sampling in order to approximate the inference. For the exam fear example, the algorithm to perform the Gibbs sampling consists of computing the probability distributions (possibly by using parts of the variable elimination algorithm) and then sampling from it:

```
for i in range(nsamples):
    # values contain current samples of b, r, a, s
    values = np.where(np.random.rand(4)<0.5,0,1)
    for j in range(nsteps):
        values=pb_ras(values)
        values=pr_bas(values)
        values=pa_brp(values)
        values=ps_bra(values)
    distribution[values[0]+2*values[1]+4*values[2]+8*values[3]] += 1
distribution /= nsamples
```

For the example, a sample distribution (based on 500 samples, with 10 iterations of each chain) is:

```
b r a s:   dist
1 1 1 1    0.0
1 1 1 0    0.086
1 1 0 1    0.038
1 1 0 0    0.052
1 0 1 1    0.048
```

```
1 0 1 0    0.116
1 0 0 1    0.274
1 0 0 0    0.0
0 1 1 1    0.0
0 1 1 0    0.088
0 1 0 1    0.068
0 1 0 0    0.114
0 0 1 1    0.03
0 0 1 0    0.076
0 0 0 1    0.01
0 0 0 0    0.0
```

16.1.3 Making Bayesian Networks

If we are given the structure and conditional probability tables of the Bayesian network, then we can perform inference on it by using Gibbs sampling or, if the network is simple enough, exactly. However, this raises the important question about where the Bayesian network itself comes from. Unfortunately, the news in this area isn't particularly good: the computational costs of searching over trees are immense, as we shall see. It is not uncommon for people to create the entire network by hand, and only then to use algorithms in order to perform inference on the network. Constructing Bayesian networks by hand is obviously very boring to do, and unless it is based on real data, then it is subjective: putting a whole lot of effort into inference is a waste of time if the data you are inferring about bears no resemblance to reality!

So why is it so difficult to construct Bayesian networks? First, we have already seen that the problem of exact inference on Bayesian networks was NP-hard, which is why we had to use approximate inference. Now let's think about the structure of the graph a little. If there are N nodes (i.e., N random variables in the graph), then how many different graphs are there? For just three nodes ('A', 'B', 'C') we can leave the three unconnected, connect 'A' to 'B' and leave 'C' alone, connect 'B' to 'A' and leave 'C' alone (remember that the links are directional) and lots of variations of that, so that there are seven possible graphs before we have even connected all three nodes to each other. For ten nodes there are $\mathcal{O}(10^{18})$ possible graphs, so we are not going to be searching over all of them. Further, we might want our algorithm to be able to include latent variables, i.e., hidden nodes, which might be a sensible thing to do in terms of explaining the data, but it does make the problem of search even worse.

We've talked about search before: Chapters 9 and 10 are full of search methods. So can we use those methods to solve this problem? The answer is a cautious yes, once we have worked out an objective function to maximise. We want to reward graphs that explain the data well, but we also want to appeal to Occam's razor (which we saw in Section 12.2.2) to ensure that the graphs are as simple as possible. Typical methods are to use an objective function based on the Minimum Description Length (MDL) (which is based on the argument that the solution with the shortest description, i.e., fewest parameters that explains the data, is the best one) or related information-theoretic measures. Then hill climbing or similar algorithms are used to perform local search around a set of random starting graphs. As usual for optimisation problems, getting the scoring function right is critical. You might be wondering why it is not possible to use a genetic algorithm. It is, but given the number of iterations of the GA, each of which would involve constructing hundreds of possible networks, testing them by performing inference, and then combining them, the computational expense rules it out as a practical possibility. As it is such an important problem, there has been a lot of very advanced work on it, which is beyond our scope here. However, there are references at the end of the chapter that contain more information should you want it.

Given that we cannot make the entire graph, we will consider the compromise situation, where we try to compute the conditional probability tables for a known graph based on data. This is quite a sensible compromise: you assume that some expert can put together a network that shows how variables relate to each other, effectively a 'cartoon' of the data generating process, and then you use data in order to compute the conditional probability tables. However, it is still difficult. The idea is to choose the probability distributions to maximise the likelihood of the training data. If there are no hidden nodes, then it is possible to compute the likelihood directly:

$$\begin{aligned} L &= \frac{1}{M}\log\prod_{m=1}^{N} P(D_m|G) \\ &= \frac{1}{M}\sum_{n=1}^{N}\sum_{m=1}^{M}\log P(X_n|\text{parents}(X_n), D_m), \end{aligned} \qquad (16.8)$$

where M is the number of training data examples D_m, and X_n is one of the N nodes in graph G. Equation (16.8) has broken everything into sums over each node individually, which means that we can compute each separate conditional probability table. To compute the values of the table, you just need to count how often you have been scared before an exam given each of the possible values for having revised and attended lectures, and normalise it to make it into a probability. The danger with this is that with small amounts of data there could be examples that have not happened in training, and that will therefore have probability 0, although this can be dealt with by including prior probabilities and using Bayes' rule to update the estimates using the real data.

Obviously, this doesn't work if there are hidden nodes, since we don't know values for them in the data. Surprisingly, getting around this problem isn't as difficult as might be expected. The key is to see that if we did have values for them, then Equation (16.8) could be used. We can estimate values for them by inference, and then we can iterate these two steps: an estimation step using inference followed by a maximisation step, making this an EM algorithm (Section 7.1.1).

There is lots more work on Bayesian networks, and the references at the end of the chapter include entire books on the topic for anybody wishing to explore more in this area. We will now turn our attention to some other types of graphical models, starting with the variation where the edges are undirected.

16.2 MARKOV RANDOM FIELDS

Bayesian networks are inherently asymmetric, since each edge had an arrow on it. If we remove this constraint, then there is no longer any idea of children and parent nodes. It also makes the idea of conditional independence that we saw for the Bayesian network easier: two nodes in a Markov Random Field (MRF) are conditionally independent of each other, given a third node, if there is no path between the two nodes that doesn't pass through the third node. This is actually a variation on the Markov property, which is how the networks got their name: the state of a particular node is a function only of the states of its immediate neighbours, since all other nodes are conditionally independent given its neighbours. You might think that this fact would make inference on MRFs simpler, but unfortunately it doesn't; in general it is still a #P-hard problem. However, there are particular applications where MRF methods have turned out to be particularly useful, often for images.

The most well-known example is image denoising, something that we have already seen in Section 4.4.5 when we talked about auto-associative learning in the MLP. Suppose that we have a binary image I with pixel values $I_{x_i,x_j} \in \{-1, 1\}$. This image is a representation of an 'ideal' image I'_{x_i,x_j} that has no noise in it, which is what we want to recover. If we assume that the amount of noise is small, then there should be a good correlation between the values of each pixel in the two images, so I_{x_i,x_j} and I'_{x_i,x_j} should be correlated. We also assume that within a small 'patch' or region in an image, there is good correlation between pixels (so I_{x_i,x_j} should correlate well with I_{x_i+1,x_j} and its other neighbouring pixels (I_{x_i,x_j-1}, etc.). This assumption says that there are lots of places in the image where all of the pixels are of the same value, and this is (at least approximately) true for most images, and says that the pixels are correlated (and that other pixels in the image are conditionally independent of I_{x_i,x_j} given the neighbours of that pixel, which is the MRF bit).

The original theory of MRFs was worked out by physicists, initially by looking at the Ising model, which is a statistic description of a set of atoms connected in a chain, where each can spin up (+1) or down (-1) and whose spin affects those connected to it in the chain. Physicists tend to think of the energy of such systems, and argue that stable states are those with the lowest energy, since the system needs to get extra energy if it wants to move out of this state. For this reason, the jargon of MRFs is in terms of energies, and we therefore want the energy of our pair of images to be low when the pixels match, and higher when they do not. So we write the energy of the same pixel in two images as $-\eta I_{x_i,x_j} I'_{x_i,x_j}$, where η is a positive constant. Note that if the two pixels have the same sign then the energy is negative, while if they have opposite signs then the energy is positive and therefore larger. The energy of two neighbouring pixels is $-\zeta I_{x_i,x_j} I_{x_i+1,x_j}$, and we can just add these components together to get the total energy:

$$E(I, I') = -\zeta \sum_{i,j}^{N} I_{x_i,x_j} I_{x_i \pm 1, x_j \pm 1} - \eta \sum_{i,j=1}^{N} I_{x_i,x_j} I'_{x_i,x_j}, \qquad (16.9)$$

where the index of the pixels is assumed to run from 1 to N in both the x and y directions in both images and we are only interested in locally flat patches of the image we are changing, which is I.

There is now a simple iterative update algorithm, which is to start with noisy image I and ideal I', and update I so that at each step the energy calculation is lower. So you pick one pixel I_{x_i,x_j} for some values of x_i, x_j at a time, and compute the energies with this pixel being set to -1 and 1, picking the lower one. In probabilistic terms, we are making the probability $p(I, I')$ higher. The algorithm then moves on to another pixel, either choosing a random pixel at each step or moving through them in some pre-determined order, running through the set of pixels until their values stop changing. Figure 16.5 shows an original black and white image, a version corrupted with 10% noise, and the MRF-reconstructed version using parameters $\eta = 2.0, \zeta = 1.5$. This reduces the error from 10% to less than 1%, although it also removes my home country of New Zealand from the map!

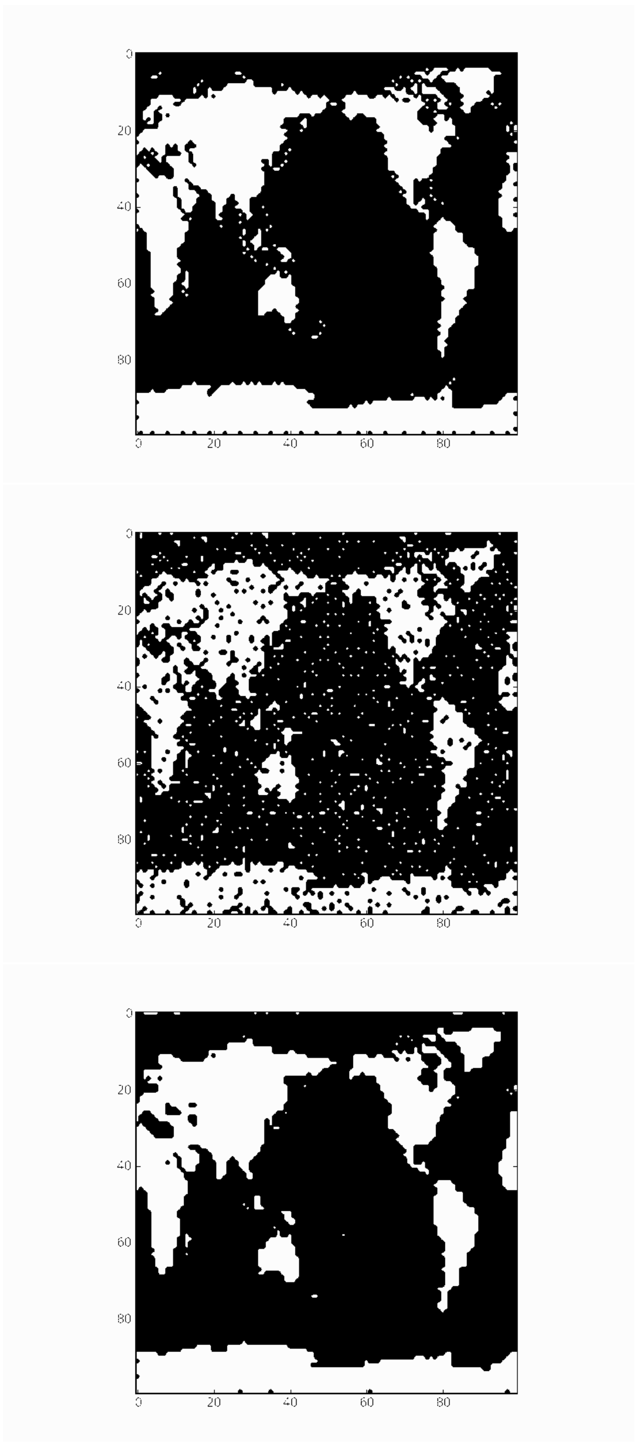

FIGURE 16.5 Using the MRF image denoising algorithm with $\eta = 2.1, \zeta = 1.5$ on a map of the world (*top left*) corrupted by 10% uniformly distributed random noise (*top right*) gives the image below which has about 1% error, although it has smoothed out the edges of all the continents.

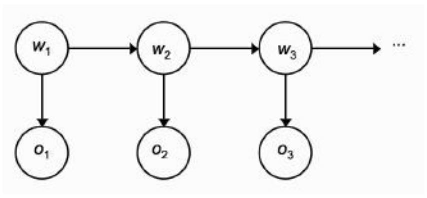

FIGURE 16.6 The Hidden Markov Model is an example of a dynamic Bayesian network. The figure shows the first three states and the related observations unrolled as time progresses.

The Markov Random Field Image Denoising Algorithm

- Given a noisy image I and an original image I', together with parameters η, ζ:
- Loop over the pixels of image I:
  - compute the energies with the current pixel being -1 and 1
  - pick the one with lower energy and set its value in I accordingly

We will now focus on a type of graphical model that is in very common use, and that has computationally tractable algorithms for doing exact inference on it.

16.3 HIDDEN MARKOV MODELS (HMMS)

The Hidden Markov Model is one of the most popular graphical models. It is used in speech processing and in a lot of statistical work. The HMM generally works on a set of temporal data. At each clock tick the system moves into a new state, which can be the same as the previous one. Its power comes from the fact that it deals with situations where you have a Markov model, but you do not know exactly which state of the Markov model you are in—instead, you see observations that do not uniquely identify the state. This is where the *hidden* in the title comes from. Performing inference on the HMM is not that computationally expensive, which is a big improvement over the more general Bayesian network. The applications that it is most commonly applied to are temporal: a set of measurements made at regular time intervals, which comprise the observations of the state. In fact, the HMM is the simplest dynamic Bayesian network, a Bayesian network that deals with sequential (often time-series) data. Figure 16.6 shows the HMM as a graphical model.

The example that we will use is this: As a caring teacher I want to know whether or not you are actually working towards the exam. I know from Chapter 12 that there are four things that you do in the evenings (go to the pub, watch TV, go to a party, study) and I want to work out whether or not you are studying. However, I can't just ask you, because you would probably lie to me. So all I can do is try to make observations about your behaviour and appearance. Specifically, I can probably work out if you look tired, hungover, scared, or fine. I want to use these observations to try to work out what you did last night. The problem is that I don't know why you look the way you do, but I can guess by assigning probabilities to those things. So if you look hungover, then I might give probability 0.5 to the guess that you went to the pub last night, 0.25 to the guess that you went to a party, 0.2 to watching TV, and 0.05 to studying. In fact, we will use these the other way round, using the probability that you look hungover given what you did last night. These are known as observation or emission probabilities.

I don't have access to the other information that was used in Chapter 12, such as what parties are on and what other assignments you have (one of the worst things about stopping being a student is that the number of parties you get invited to drops off), but based on my own experience of being a student I can guess how likely parties are, etc., and knowing what student finances are, I can guess things like the probability of you going to the pub tonight if you went to the pub last night. So now it is just a question of putting these things into a form where I can work with them, and I can prepare my lectures according to how well you are working.

Each day that I see you in lectures I make an observation of your appearance, $o(t)$, and I want to use that observation to guess the state $\omega(t)$. This requires me to build up some kind of probabilities $P(o_k(t)|\omega_j(t))$, which is the probability that I see observation o_k (e.g., you are tired) given that you were in state ω_j (e.g., you went to a party) last night. These are usually labelled as $b_j(o_k)$. The other information that I have, or think I have, is the transition probability, which tells me how likely you are to be in state ω_j tonight given that you were in state ω_i last night. So if I think you were at the pub last night I will probably guess that the probability of you being there again tonight is small because your student loan won't be able to handle it. This is written as $P(\omega_j(t+1)|\omega_i(t))$ and is usually labelled as $a_{i,j}$.

I can add one more constraint to each of the probability distributions $a_{i,j}$ and b_i. I know that you did something last night, so $\sum_j a_{i,j} = 1$ and I know that I will make some observation (since if you aren't in the lecture I'll assume you were too tired), so $\sum_k b_j(o_k) = 1$. There is one other thing that is generally assumed, which is that the Markov chain is ergodic, something that we saw in Section 15.4.1: it means that there is a non-zero probability of reaching every state eventually, no matter what the starting state.

After a couple of weeks of the course I have made observations about you, and I am ready to sort out my HMM. There are three things that I might want to do with the data:

- see how well the sequence of observations that I've made match my current HMM (Section 16.3.1)
- work out the most probable sequence of states that you've been in based on my observations (Section 16.3.2)
- given several sets of observations (for example, by watching several students) generate a good HMM for the data (Section 16.3.3)

We will start by assuming that I invent a model and want to see how good it is. So I use my own knowledge of being a student to work out the probability distributions and then I can test the observations I make of you against my model. At this point I will probably find out that my student life was different to yours, or things have changed since I was a student, and I will have to generate a new model to match current data. I can then use this improved model to work out what you've been doing each evening. These problems are dealt with in the next three sections.

The HMM itself is made up of the transition probabilities $a_{i,j}$ and the observation probabilities $b_j(o_k)$, and the probability of starting in each of the states, π_i. So these are the things that I need to specify for myself, starting with the transition probabilities (which are also shown in Figure 16.7):

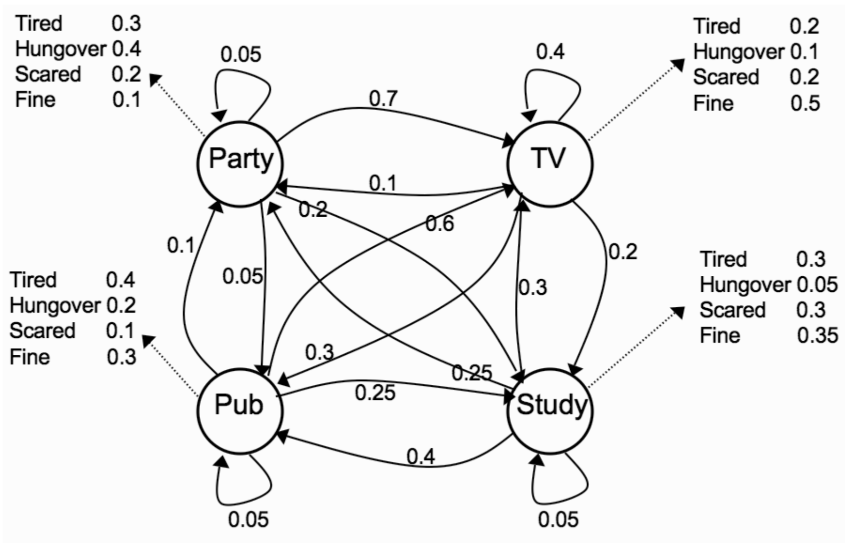

FIGURE 16.7 The example HMM with transition and observation probabilities shown.

	Previous night			
	TV	Pub	Party	Study
TV	0.4	0.6	0.7	0.3
Pub	0.3	0.05	0.05	0.4
Party	0.1	0.1	0.05	0.25
Study	0.2	0.25	0.2	0.05

and then the observation probabilities:

	TV	Pub	Party	Study
Tired	0.2	0.4	0.3	0.3
Hungover	0.1	0.2	0.4	0.05
Scared	0.2	0.1	0.2	0.3
Fine	0.5	0.3	0.1	0.35

16.3.1 The Forward Algorithm

Suppose that I see the following observations: $O =$ (tired, tired, fine, hungover, hungover, scared, hungover, fine) and I want to work out the likely run of states that generated it. The probability that my observations $O = \{o(1), \ldots, o(T)\}$ come from the model can be computed using simple conditional probability. I know you were doing something last night, so for an observation $o(t) =$tired (say) I just need to compute the probability that I made that observation given you were in a particular state (say watching TV) and multiply it by the probability that you were in that state given the state I thought you were in the night before (say partying). So for the example, I compute the probability that you were tired given that you were watching TV, which is 0.2, and then multiply it by the probability that you spent last night watching TV given that I thought you were partying the night before, which is 0.1. So this yields probability 0.02 for this particular state change. There is one extra thing that we need which is to decide which state you actually start in. I don't know this, so I assign probability 0.25 to each state.

Now I need to do this over every possible sequence of states to find out the most likely one based on what I actually saw. Note that I have used O to denote the whole sequence of observations that I made. In the same way, Ω is an entire sequence of possible states (this is a change in notation from the rest of the methods we have looked at, but it is consistent with the way that other authors describe HMMs). This can be written as:

$$P(O) = \sum_{r=1}^{R} P(O|\Omega_r)P(\Omega_r). \tag{16.10}$$

The r index here describes a possible sequence of states, so Ω_1 is one sequence, Ω_2 another, and so on. We'll consider this in a minute, but first we will use the Markov property to write:

$$P(\Omega_r) = \prod_{t=1}^{T} P(\omega_j(t)|\omega_i(t-1)) = \prod_{t=1}^{T} a_{i,j}, \tag{16.11}$$

and

$$P(O|\Omega_r) = \prod_{t=1}^{T} P(o_k(t)|\omega_j(t)) = \prod_{t=1}^{T} b_j(o_k). \tag{16.12}$$

So Equation (16.10) can be written as:

$$\begin{aligned} P(O) &= \sum_{r=1}^{R}\prod_{t=1}^{T} P(o_k(t)|\omega_j(t))P(\omega_j(t)|\omega_i(t-1)) \\ &= \sum_{r=1}^{R}\prod_{t=1}^{T} b_j(o_k)a_{i,j}. \end{aligned} \tag{16.13}$$

This looks fairly easy now. The only problem is in that sum over r, which runs over all possible sequences of hidden states. If there are N hidden states then there are N^T possible sequences, and for each one we have to compute a product of T probabilities. Not only will these probabilities be incredibly small, but the computational cost of getting them will be astronomical: $\mathcal{O}(N^T T)$.

Fortunately, the Markov property comes to our rescue again. Since the probability of each state only depends on the data at the current and previous timestep $(o(t), \omega(t), \omega(t-1))$ we can build up our computation of $P(O)$ one timestep at a time. This is known as the forward trellis by some people, since it looks like a garden trellis in Figure 16.8. To construct the trellis we introduce a new variable $\alpha_i(t)$ that describes the probability that at time t the state is ω_i and that the first $(t-1)$ steps all matched the observations $o(t)$:

$$\alpha_j(t) = \begin{cases} 0 & t=0, j \neq \text{initial state} \\ 1 & t=0, j = \text{initial state} \\ \sum_i \alpha_i(t-1)a_{i,j}b_j(o_t) & \text{otherwise.} \end{cases} \tag{16.14}$$

where $b_j(o_t)$ means the particular emission probability of output o_t. This ensures that only the observation probability that has the index that matches the observation o_t contributes to the sum. Computing $P(O)$ now requires only $\mathcal{O}(N^2 T)$, which is a substantial improvement, in a very simple algorithm, as will be seen shortly.

We will use the following notation: $a_{i,j}$ is the transition probability of going from state i

to state j, so if there are N states, then it is of size $N \times N$; $b_i(o)$ is the transition probability of emitting observation o in state i, so it is of size $N \times O$, where O is the number of different observations that there are (four in the example). It will be useful to introduce four more variables, all of which are probabilities that are conditioned on the observation sequence and the model:

- $\alpha_{i,t}$, which is the probability of getting the observation sequence up to time t and being in state i at time t (size $N \times t$),
- $\beta_{i,t}$, which is the probability of the sequence from $t+1$ to the end given that the state is i at time t,
- $\delta_{i,t}$, which is the highest probability of any path that reaches state i at time t,
- $\xi_{i,j,t}$, which is the probability of being in state i at time t, state j at time $t+1$, and so is an $N \times N \times T$ matrix.

Since $\alpha_{i,t}$ is the probability of getting the observation sequence up to time t and being in state i at time t conditioned on the model and the observations, the probability of the whole observation sequence given the model is just $\sum_{i=1}^{N} \alpha_{i,T}$.

The HMM Forward Algorithm

- Initialise with $\alpha_{i,0} = \pi_i b_i(o_0)$
- For each observation in order $o_t,\ t = 1, \ldots, T$
  - for each of the N_s possible states s:
    - $\alpha_{s,t+1} = b_s(o_{t+1}) \left(\sum_{i=1}^{N} (\alpha_{i,t} a_{i,s})\right)$

Let's look at the first two states of our example HMM. In both, the observation is that you were tired, so we need to compute $\alpha_{i,t}$ and so construct the trellis. Figure 16.8 shows the idea, with the initial $\alpha_{i,t=0}$ coming from my guesses about how likely each state is (the π variable), and we just need to run through the set of computations to compute $\alpha_{i,t=2}$ and so on, getting the numbers that are shown in the figure. We then repeat this for the next step and so on until we reach the final state. At this stage we can sum up all of the possible probabilities, which tells me in this case that you were most likely watching TV last night. We will also need to be able to go backwards through the trellis, which is a very similar algorithm that works backwards to compute β values, also based on the transmission probability and observation probability matrices by:

$$\beta_{i,t} = \sum_{j=1}^{N} a_{i,j} b_j(o_{t+1}) \beta_{j,t+1}. \tag{16.15}$$

16.3.2 The Viterbi Algorithm

The next problem that we want to solve is the decoding problem of working out the hidden states: I can use my model of how students are expected to behave and match them with my observations to guess what you have been doing each evening. The algorithm is known as the Viterbi algorithm after its creator, although he actually derived it for error correction, a

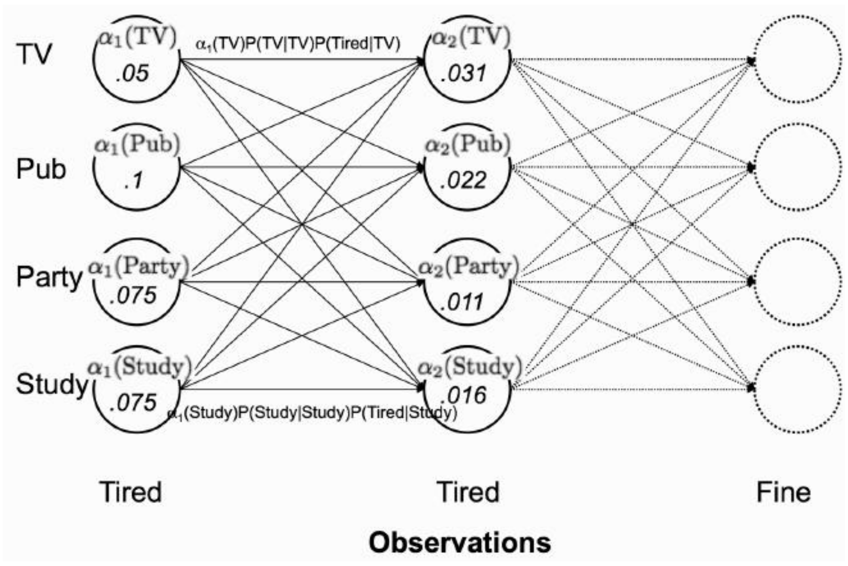

FIGURE 16.8 The forward trellis for the first two observations of the example HMM.

completely different application! We want to work out the $\delta_{i,t}$ variable. This requires finding the maximum probability of any path that gets us to state i at time t that has the right observations for the sequence up to t; apart from being a maximisation instead of a sum, it is pretty similar to the forward algorithm. The initialisation is $\delta_{i,t=0} = \pi_i b_i(o_0)$, since this tells us about the first observations we see, and then we work from there computing each new δ as:

$$\delta_{j,t+1} = \max_i \left(\delta_{i,t} a_{i,j}\right) b_j(o_{t+1}). \tag{16.16}$$

It will also be useful to keep track of which state seems to be the best at each stage: $\phi_{j,t} = \arg\max_i(\delta_{i,t-1}a_{i,j})$, because at the end of the sequence we want to go backwards through the matrix that we have built up and work out the actual most probable path from it.

So once we reach the end, we can work out the most likely state as the one with the highest $q^*_T = \delta\cdot, T$, and then work back through the lattice using $q^*_t = \phi_{q^*_{t+1},t+1}$ until we reach the start of the sequence. As an algorithm this can be written as:

The HMM Viterbi Algorithm

```
• Start by initialising $\delta_{i,0}$ by $\pi_i bi(o_0)$ for each state $i$, $\phi_0 = 0$
    – run forward in time $t$:
        ∗ for each possible state $s$:
            · $\delta_{s,t} = \max_i(\delta_{i,t-1}a_{i,s})b_s(o_t)$
            · $\phi_{s,t} = \arg\max_i(\delta_{i,t-1}a_{i,s})$
    – set $q^*_T$, the most likely end hidden state to be $q^*_T = \arg\max_i \delta_{i,T}$
    – run backwards in time computing:
        ∗ $q^*_{t-1} = \phi_{q^*_t,t}$
```

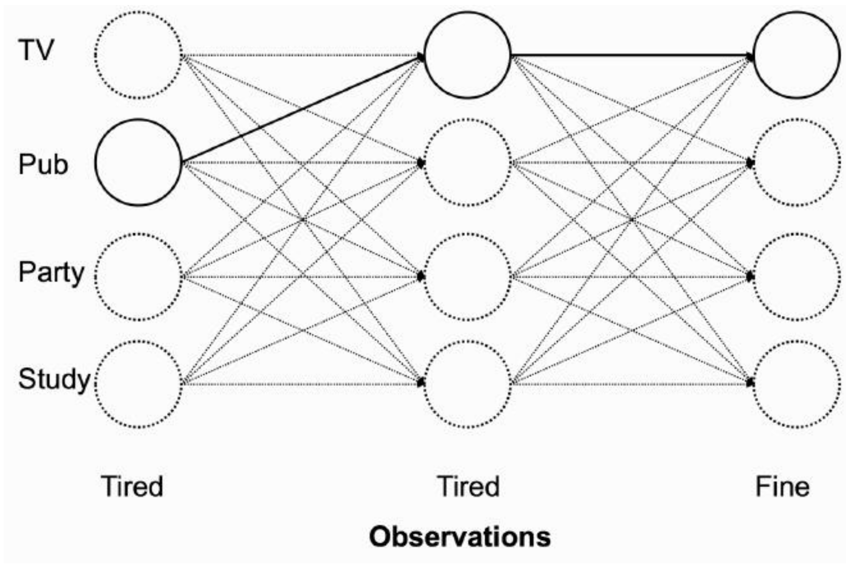

FIGURE 16.9 The Viterbi trellis for the first three observations of the example HMM.

Figure 16.9 shows the path for the first three states of the example. Using the numbers from the example, I can use the Viterbi algorithm to find the most likely explanation of a set of observations such as (fine, hungover, hungover, fine, tired, fine, fine, fine, hungover, hungover, tired, scared, scared), which tells me that you seem to have spent a lot of time in the pub. However, even for this most likely sequence, the probability of it is only 7.65×10^{-9}, which doesn't seem very likely. This is one of the problems with HMMs: the state space is so large that the probabilities tend to 0 very quickly. This is both an interpretation problem and a computational one, since we get problems with rounding errors very quickly because the probabilities are so small. This is discussed briefly at the end of this section.

16.3.3 The Baum–Welch or Forward–Backward Algorithm

In the example I had to invent the transition and observation probabilities from my experience, and the result is that the best path is not very likely. It would obviously be better to generate the HMM from sets of observations rather than by making up the transition probabilities. This is a learning process, and it is an unsupervised learning problem since we don't have any target solutions to go on. In fact, finding the optimal probabilities is an NP-complete problem, since we have to search over all the possible sets of probabilities for all the possible sequences. Instead, we will use an EM algorithm known as the Baum–Welch algorithm (see Section 7.1.1 for a previous example of an EM algorithm). This is not quite as good as the previous one in that it is not guaranteed to find even a local optima, but in general it works fairly well.

The key to the algorithm is in its second name: Forward–Backward. We introduced a variable α above that took us forward through the HMM above, and we mentioned that it had a complementary variable β that takes us backwards through the HMM, i.e., $\beta_i(t)$ tells us the probability that at time t we are in state ω_i and the result of the target sequence (times $t+1$ to T) will be generated correctly. So we can now pick any point in the middle of a sequence, and run forwards from the beginning and backwards from the end to see the possible paths.

To see what computations are needed we will work out what the three variables that we are interesting in fitting—π_i, $a_{i,j}$ and $b_i(o_k)$—are, which are respectively the number of times we expect to be in state i at the first observation, the expected number of times we transition from state i to state j divided by the number of times we leave state i, and the expected number of times we see observation o_k when in state i, divided by the number of times we are in state i.

Thinking about the $\xi_{i,j,t}$ variable that we talked about, but haven't used yet, and which is the probability of being in state i at time t, state j at time $t+1$, we can see that (where the $\hat{}$ is to make it clear that these are estimates based on the sequences that we see):

$$\hat{\pi}_i = \sum_{j=1}^{N} \xi_{i,j,0} \tag{16.17}$$

$$\hat{a}_{i,j} = \sum_{t=1}^{T-1} \xi_{i,j,t} / \sum_{t=1}^{T-1} \sum_{j=1}^{N} \xi_{i,j,t} \tag{16.18}$$

$$\hat{b}_i(o_k) = \sum_{t=1,o_t=k}^{T} \sum_{j=1}^{N} \xi_{i,j,t} / \sum_{t=1}^{T} \sum_{j=1}^{N} \xi_{i,j,t} \tag{16.19}$$

The notation in the last line means that in the numerator we only include those times t where the observation k was seen, and note that the sums over time in the middle line only go up to $T-1$, since it isn't possible to move on from the last state.

So now the algorithm needs to start by computing $\xi_{i,j,t}$, and then make an estimate of π, a, b, which can then be iterated until the values stop changing. This has the flavour of an EM algorithm: we work out how many times we can expect to transition between states, which is the expectation, and then we try to maximise them.

The only thing that we haven't done yet is to work out how to compute $\xi_{i,j,t}$, although looking back at the definitions of the α and β variables, and working out that what we are doing is running forwards until time t, when we get to state i, then transitioning to state j carrying on from there to the end (or equivalently, going backwards from the end to state j at time $t+1$, we see that the form of $\xi_{i,j,t}$ is:

$$\xi_{i,j,t} = \frac{\alpha_{i,t} a_{i,j} b_j(o_{t+1}) \beta_{j,t+1}}{\sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_{i,t} a_{i,j} b_j(o_{t+1}) \beta_{j,t+1}}, \tag{16.20}$$

where the numerator is simply a normaliser, and the values at T are a little different as there is no b or β then. This leads to the complete Baum–Welch algorithm:

The HMM Baum–Welch (Forward–Backward) Algorithm

```
• Initialise $\pi$ to be equal probabilities for all states, and $a$, $b$ randomly unless you have
  prior knowledge
• While updates have not converged:
    – E-step:
    – use forward and backward algorithms to get $\alpha$ and $\beta$
    – for each observation in the sequence $o_t, t = 1 \dots T$
        ∗ for each state $i$:
            · for each state $j$:
            ·   compute $\xi$ using Equation (16.20)
    – M-step:
    – for each state $i$:
        ∗ compute $\hat{\pi}_i$ using Equation (16.17)
        ∗ for each state $j$:
            · compute $\hat{a}_{i,j}$ using Equation (16.18)
    – for each different possible observation $o$:
        ∗ compute $\hat{b}_i(o)$ using Equation (16.19)
```

Since this is the most important algorithm, here is a Python implementation as well:

```python
def BaumWelch(obs,nStates):

        T = np.shape(obs)[0]
        xi = np.zeros((nStates,nStates,T))

        # Initialise pi, a, b randomly
        pi = 1./nStates*np.ones((nStates))
        a = np.random.rand(nStates,nStates)
        b = np.random.rand(nStates,np.max(obs)+1)

        tol = 1e-5
        error = tol+1
        maxits = 100
        nits = 0
        while ((error > tol) & (nits < maxits)):
                nits += 1
                oldpi = pi.copy()
                olda = a.copy()
                oldb = b.copy()

                # E step
                alpha,c = HMMfwd(pi,a,b,obs)
                beta = HMMbwd(a,b,obs,c)

                for t in range(T-1):
                        for i in range(nStates):
                                for j in range(nStates):
                                        xi[i,j,t] = alpha[i,t]*a[i,j]*b[j,⤸
                                        obs[t+1]]*beta[j,t+1]
                        xi[:,:,t] /= np.sum(xi[:,:,t])

                # The last step has no b, beta in
                for i in range(nStates):
                        for j in range(nStates):
                                xi[i,j,T-1] = alpha[i,T-1]*a[i,j]
                xi[:,:,T-1] /= np.sum(xi[:,:,T-1])

                # M step
                for i in range(nStates):
                        pi[i] = np.sum(xi[i,:,0])
                        for j in range(nStates):
                                a[i,j] = np.sum(xi[i,j,:T-1])/np.sum(xi[i,:,:⤸
                                T-1])

                        for k in range(max(obs)):
                                found = (obs==k).nonzero()
                                b[i,k] = np.sum(xi[i,:,found])/np.sum(xi[i,:,⤸
                                :])

                error = (np.abs(a-olda)).max() + (np.abs(b-oldb)).max()
                print nits, error, 1./np.sum(1./c), np.sum(alpha[:,T-1])

        return pi, a, b
```

We can't really use this algorithm very well on the simple example, since we would need rather more data to do justice to the training. However, if we do apply it and then compute the Viterbi path, then it gives the same answer as with the invented data.

One final thing to mention with the HMM is that there are ways to deal with the fact that the probabilities get so small, which can cause round-off errors inside the computer. One approach is to renormalise the α values by dividing by the sum of the α values for each time step. If the same values are used in the β calculation as well, then they cancel out beautifully in the final calculations and things work very well. This is the c variable in the following code for the forward algorithm:

```
def HMMfwd(pi,a,b,obs):

    nStates = np.shape(b)[0]
    T = np.shape(obs)[0]
```

```
alpha = np.zeros((nStates,T))
alpha[:,0] = pi*b[:,obs[0]]

for t in range(1,T):
    for s in range(nStates):
        alpha[s,t] = b[s,obs[t]] * np.sum(alpha[:,t-1] * a[:,s])

c = np.ones((T))
if scaling:
    for t in range(T):
        c[t] = np.sum(alpha[:,t])
        alpha[:,t] /= c[t]
return alpha,c
```

That pretty much sums it up for the HMM. It is worth mentioning two limitations of it, which are that the probability distributions are not time dependent, and that the probabilities can get very small. The second of these problems is an implementation detail that needs careful monitoring, while the first can be dealt with by using more general graphical models, although with the additional computational costs that come with that.

16.4 TRACKING METHODS

We will now look at two methods of performing tracking. You perform tracking fairly easily, keeping tabs on where something is and how it is moving. This has an obvious evolutionary benefit, since keeping track of where predators were and whether they were coming towards you could keep you alive. It is also useful for a machine to be able to do this, both for similar reasons to a human or animal (watching something moving and predicting what path it will follow, for example in radar or other imaging method) and to keep track of a changing probability distribution. We will look at two methods of doing it, the Kalman filter and the particle filter.

16.4.1 The Kalman Filter

The Kalman filter (named for E. Kalman; although he was not the original inventor he did do quite a lot of work on it) is a recursive estimator. It makes an estimate of the next step, then computes an error term based on the value that was actually produced in the next step, and tries to correct it. It then uses both of those to make the next prediction, and iterates this procedure. It can be seen as a simple cycle of predict-correct behaviour, where the error at each step is used to improve the estimate at the next iteration. The Kalman filter can be represented by the graphical model shown in Figure 16.10.

Much of the jargon that is associated with the Kalman filter is familiar to us: the state, which is hidden, consists of the variables that we want to know, which we see through noisy observations over time. There is a transition model that tells us how states change from one to another, and an observation model (also called the sensor model here) that tells us how states lead to observations.

The underlying idea is that there is some time-varying process that is generating a set of noisy outputs, where there are two sources of noise: process noise, which represents

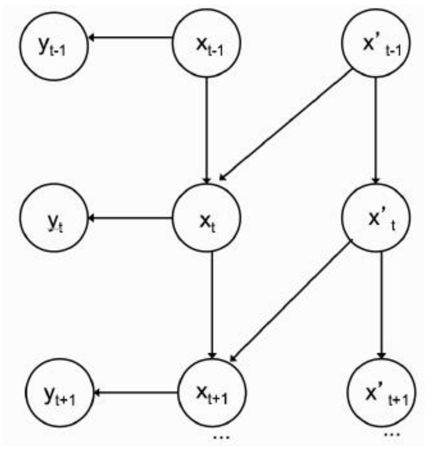

FIGURE 16.10 A representation of the Kalman filter with time derivatives (such as for tracking) as a graphical model.

the fact that the process changes over time, but we don't know how, and observation (or measurement) noise, which is the errors that are made in the readings. Both are assumed to be independent of each other, and zero mean Gaussians. We write the process as a stochastic difference equation in $\mathbf{x}$, which has n dimensions:

$$\mathbf{x}_{t+1} = \mathbf{A}\mathbf{x}_t + \mathbf{B}\mathbf{u}_t + \mathbf{w}_t, \tag{16.21}$$

where $\mathbf{A}_t$ is an $n \times n$ matrix that represents the non-driven part of the underlying process, $\mathbf{B}$ is an $n \times l$ matrix that represents the driving force, and $\mathbf{u}$ is the l-dimensional driving force. $\mathbf{w}$ is the process noise, which is assumed to be zero mean with standard deviation $\mathbf{Q}$.

The observations that we make are m-dimensional:

$$\mathbf{y}_t = \mathbf{H}\mathbf{x}_t + \mathbf{v}_t, \tag{16.22}$$

where $m \times n$ matrix $\mathbf{H}$ describes how measurements of the state are measured, and $\mathbf{v}$ is the measurement noise, which is also assumed to be zero mean, but with standard deviation $\mathbf{R}$.

As an example of this, consider a particle moving at a constant speed in one dimension. The state is two dimensional, consisting of the position and velocity of the particle (so $\mathbf{x} = [x, \dot{x}]^T$). Since there is no driving force the next term is $\mathbf{B}\mathbf{u}_t = 0$. The process equation is then derived by using Newton's laws of motion (so $x_{t+1} = x_t + \Delta t \dot{x}_t$, and since we are making indexing by time, $\Delta_t = 1$, and the velocity is constant):

$$\begin{pmatrix} x_{t+1} \\ \dot{x}_{t+1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x_t \\ \dot{x}_t \end{pmatrix} + \mathbf{w}_t. \tag{16.23}$$

We can observe the position of the particle, up to measurement noise, but not the velocity, and so the measurement equation is:

$$y_t = \begin{pmatrix} 1 & 0 \end{pmatrix} \begin{pmatrix} x_t \\ \dot{x}_t \end{pmatrix} + \mathbf{v}_t. \tag{16.24}$$

The principal simplifying assumptions of the Kalman filter are that the process is linear and that all of the distributions are Gaussian with constant covariance. Since the convolution of Gaussians is also Gaussian, this means that we can put them together to form new

Gaussians, and so the model stays well behaved. This was a significant advantage over previous methods of tracking, which tended to stop working fairly quickly, since the estimates broke down because the probability distribution stopped being well-defined. We assume that both the transition model and the observation model are Gaussians with means based on the previous observations, and fixed covariances $\mathbf{Q}$ and $\mathbf{R}$ which can be written mathematically as (with the same parameters as the state and measurement update equations):

$$\begin{aligned} P(\mathbf{x}_{t+1}|\mathbf{x}_t) &= \mathcal{N}(\mathbf{x}_{t+1}|\mathbf{A}\mathbf{x}_t, \mathbf{Q}) && (16.25) \\ P(\mathbf{z}_t|\mathbf{x}_t) &= \mathcal{N}(\mathbf{z}_t|\mathbf{H}\mathbf{x}_t, \mathbf{R}). && (16.26) \end{aligned}$$

Having described the set-up, what do we do? The basic idea is to make a prediction and then correct it when the next observation is available, i.e., at the next timestep. We will use $\hat{\mathbf{x}}$ and $\hat{\mathbf{y}}$ as the estimates, so $\hat{\mathbf{y}}_{t+1} = \mathbf{H}\mathbf{A}\hat{\mathbf{x}}_{t+1}$ and so the error is $\mathbf{y}_{t+1} - \hat{y}_{t+1}$; that is the difference between what was actually observed and what we predicted (without measurement noise). Since this is a probabilistic process with Gaussian distributions, we can also keep a predicted covariance matrix that goes with it: $\hat{\mathbf{\Sigma}}_{t+1} = \mathbf{A}\mathbf{\Sigma}_t\mathbf{A}^T + \mathbf{Q}$ (which is $E[(\mathbf{x}_k - \hat{\mathbf{x}}_k)(\mathbf{x}_k - \hat{\mathbf{x}}_k)^T])$. The Kalman filter weights these error computations by how much trust the filter currently has in its predictions; these weights are known as the Kalman gain and are computed by:

$$\mathbf{K}_{t+1} = \hat{\mathbf{\Sigma}}_{t+1}\mathbf{H}^T \left(\mathbf{H}\hat{\mathbf{\Sigma}}_{t+1}\mathbf{H}^T + \mathbf{R}\right)^{-1}. \quad (16.27)$$

This equation comes from minimising the mean-square error; we will not derive it, but the Further Reading section gives further references for you to follow up if you wish. Using it, the update for the estimate is:

$$\mathbf{x}_{t+1} = \hat{\mathbf{x}}_{t+1} + \mathbf{K}_{t+1}\left(\mathbf{z}_{t+1} - \mathbf{H}\hat{\mathbf{x}}_{t+1}\right), \quad (16.28)$$

All that is then required is to update the covariance estimate:

$$\mathbf{\Sigma}_{t+1} = (\mathbf{I} - \mathbf{K}_{t+1}\mathbf{H})\hat{\mathbf{\Sigma}}_{t+1}, \quad (16.29)$$

where $\mathbf{I}$ is the identity matrix of the relevant size. Putting these equations together leads to a simple algorithm.

The Kalman Filter Algorithm

- Given an initial estimate $\mathbf{x}(0)$
- For each timestep:
  - **predict the next step**
    - predict state as $\hat{\mathbf{x}}_{t+1} = \mathbf{A}\mathbf{x}_t + \mathbf{B}\mathbf{u}_t$
    - predict covariance as $\hat{\mathbf{\Sigma}}_{t+1} = \mathbf{A}\mathbf{\Sigma}_t\mathbf{A}^T + \mathbf{Q}$
  - **update the estimate**
    - compute the error in the estimate, $\boldsymbol{\epsilon} = \mathbf{y}_{t+1} - \mathbf{H}\mathbf{A}\mathbf{x}_{t+1}$
    - compute the Kalman gain using Equation (16.27)
    - update the state using Equation (16.28)
    - update the covariance using Equation (16.29)

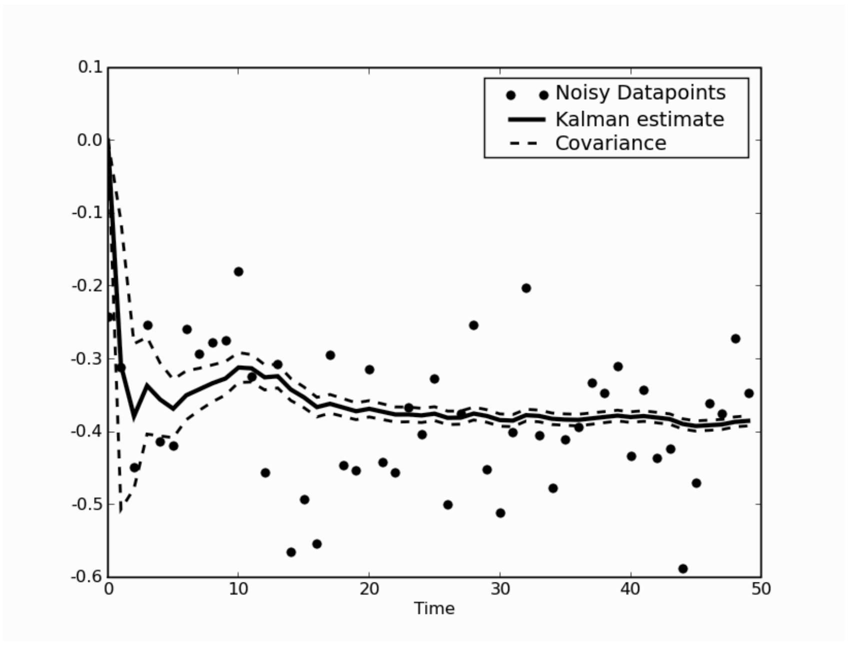

FIGURE 16.11 Estimates of a 1D constant noisy process using a Kalman filter. The filter settles to representing the unchanging mean of the process fairly quickly, and the estimated error (shown as dashed lines) drops accordingly.

Implementing this principally involves repeated use of `np.dot()` to multiply matrices together.

Figure 16.11 shows a simple 1D example of using the Kalman filter, where there is no time variation (so $x_{t+1} = x_t + w_k$). The dots are the noisy data from the process, and the line is the Kalman filter estimate, with the dashed lines representing one standard deviation. It can be seen that the initial estimate was not very good, but the algorithm quickly converges to a good estimate of the mean of the data, and the error drops accordingly.

Now that we have seen the Kalman filter in action, we need to work out how to use it for tracking. We worked out the process in 1D in Equations (16.23) and (16.24), and Figure 16.12 shows an example of 1D tracking, but we will write them in 2D here, to make sure that everything is clear. Again. we will assume that there is no control of the particle, so that it moves at constant velocity (up to noise).

The state of the particle is then:

$$\mathbf{x}_t = (x_1, x2, \dot{x}_1, \dot{x}_2)^T, \ \mathbf{y}_t = (y_1, y_2)^T, \tag{16.30}$$

$$\mathbf{A} = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \ \mathbf{H} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \tag{16.31}$$

In the absence of any other knowledge, we can assume that **Q** and **R** are proportional to the 4×4 and 2×2 identity matrices, respectively.

Figure 16.13 shows an example of a point moving in two spatial dimensions, starting at (10,10) and moving to the right at speed 1 for 15 steps, with noise standard deviation 0.1 in

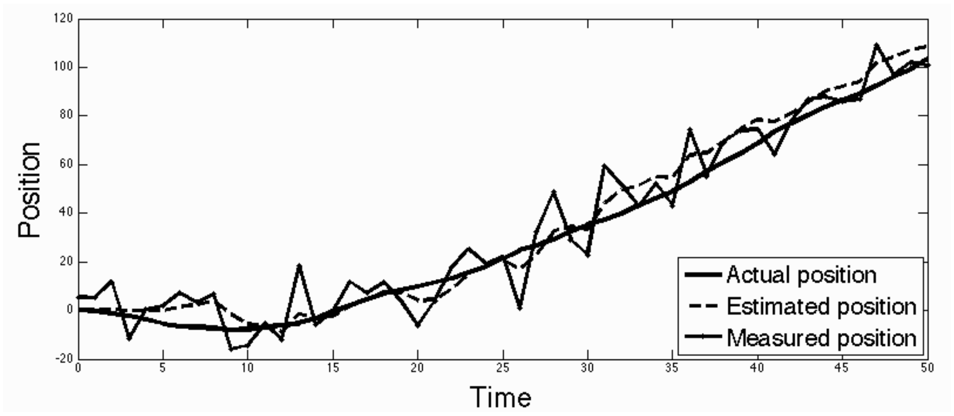

FIGURE 16.12 Demonstration of the Kalman filter tracking an object moving in one spatial dimension.

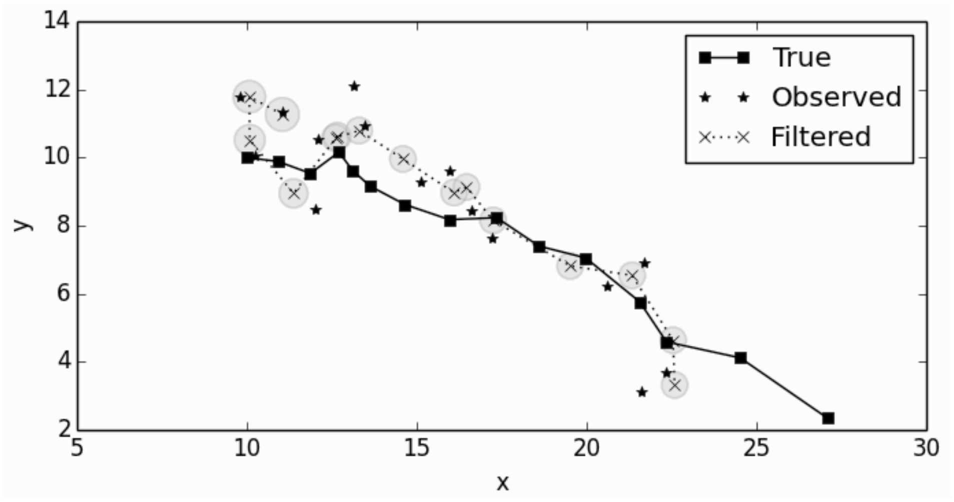

FIGURE 16.13 Demonstration of the Kalman filter tracking an object moving in two spatial dimensions. The grey circles show the covariance around each estimated point.

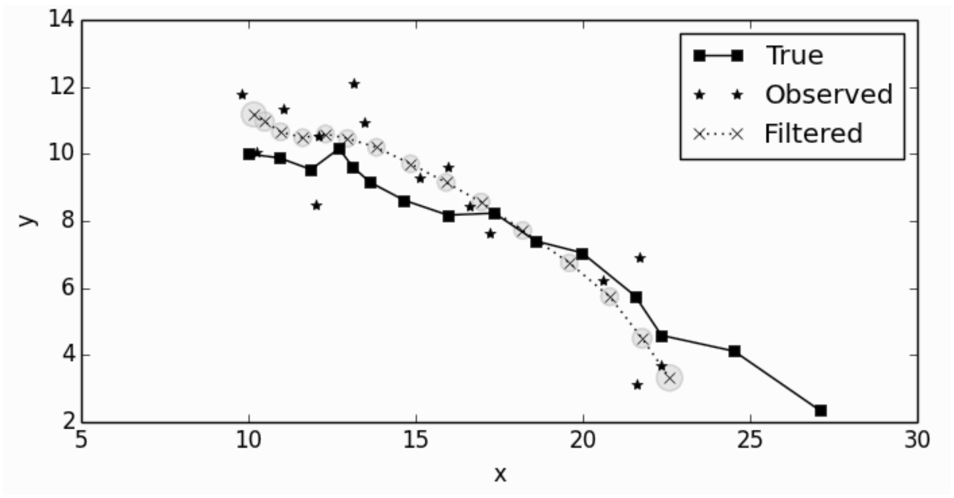

FIGURE 16.14 A smoothed version of the path from Figure 16.13.

both x_1 and x_2, with the grey circles representing the covariance matrix (at one standard deviation). It can be seen that the filter initially tracks the observations, but then learns more of the underlying process. However, the trajectory that is shown is rather 'jumpy' as the high level of noise affects the estimates. One way around this is to use the Kalman smoother, which performs a backwards smoothing of the trajectory after the filter has been used to estimate the positions. So the filter is run to predict the points, and then the predictions are updated to be a smoother path along the trajectory of the particle.

There are a variety of ways to do this smoothing, but one option, known as the Rauch–Tung–Striebel smoother is to use the following update equations from the endpoint back to the beginning (where the $\hat{(\cdot)}$ variables are the filtered versions):

$$\begin{aligned}
\mathbf{x}' &= \mathbf{A}\hat{\mathbf{x}} \\
\mathbf{\Sigma}' &= \mathbf{A}\hat{\mathbf{\Sigma}}\mathbf{A}^T + \mathbf{Q} \\
\mathbf{J} &= \hat{\mathbf{\Sigma}}\mathbf{A}\mathbf{\Sigma}' \\
\mathbf{x}_s &= \hat{\mathbf{x}} + \mathbf{J}(\hat{\mathbf{x}} - \mathbf{x}') \\
\mathbf{\Sigma}_s &= \hat{\mathbf{\Sigma}} + \mathbf{J}(\hat{\mathbf{\Sigma}} - \mathbf{\Sigma}')\mathbf{J}^T
\end{aligned} \tag{16.32}$$

Figure 16.14 shows the smoothed trajectory from Figure 16.13.

One of the main assumptions of the Kalman filter was that the process was linear. There are many cases where this is not true. One option to deal with non-linearity is to linearise about the current estimate $(\mathbf{x}_t, \mathbf{\Sigma}_t)$ and this leads to the Extended Kalman Filter. There are a lot of similarities with the original Kalman filter, so let's try and pick out the differences.

We start with some non-linear stochastic difference equation:

$$\mathbf{x}_{t+1} = f(\mathbf{x}_t, \mathbf{u}_t, \mathbf{w}_t), \tag{16.33}$$

where the variables are the same as for the Kalman filter, except that we have a non-linear function $f(\cdot)$ in place of the nice linear matrix, and we also have a measurement function:

$$\mathbf{y}_t = h(\mathbf{x}_t, \mathbf{v}_t). \tag{16.34}$$

If we have a current estimate $\hat{\mathbf{x}}_t$ then we can evaluate the function at that point by computing $\tilde{\mathbf{x}} = f(\hat{\mathbf{x}}, \mathbf{u}_t, 0)$, where we are assuming that the mean of the noise is 0. We now linearise about this point as:

$$x_{t+1} \approx \tilde{\mathbf{x}}_{t+1} + \mathbf{J}_{f,x}(\hat{\mathbf{x}}_t, \mathbf{u}_t, 0)(\mathbf{x}_t - \hat{\mathbf{x}}_t) + \mathbf{J}_{f,w}(\hat{\mathbf{x}}_t, \mathbf{u}_t, 0)\mathbf{w}_t), \tag{16.35}$$

and similarly for $h(\cdot)$ to get:

$$y_t \approx \tilde{\mathbf{y}}_{t+1} + \mathbf{J}_{h,x}(\tilde{\mathbf{x}}_t, 0)(\mathbf{x}_t - \tilde{\mathbf{x}}_t) + \mathbf{J}_{h,v}(\tilde{\mathbf{x}}_t, \mathbf{v}_t). \tag{16.36}$$

In both of these equations $\mathbf{J}$ refers to the Jacobian of the subscripted function with regard to the variable in the second subscript, so:

$$\mathbf{J}_{f,x}\big|_{i,j} = \frac{\partial f_i}{\partial \mathbf{x}_j}(\hat{\mathbf{x}}_t, \mathbf{u}_t, 0). \tag{16.37}$$

So providing that we can compute these two functions and their derivatives, we can use them to make an estimate of the error, which is a linear function and so can be estimated using the normal Kalman filter. This leads to the following algorithm:

The Extended Kalman Filter Algorithm

- Given an initial estimate $\mathbf{x}(0)$
- For each timestep:
  - **predict the next step**
    - predict state as $\hat{\mathbf{x}}_{t+1} = f(\hat{\mathbf{x}}_t, \mathbf{u}_t, 0$
    - compute the Jacobians $\mathbf{J}_{f,x}$ and $\mathbf{J}_{f,w}$
    - predict covariance as $\hat{\boldsymbol{\Sigma}}_{t+1} = \mathbf{J}_{f,x}\boldsymbol{\Sigma}_t\mathbf{J}^T_{f,x} + \mathbf{J}_{f,w}\mathbf{Q}\mathbf{J}^T_{f,w}$
  - **update the estimate**
    - compute the error in the estimate, $\boldsymbol{\epsilon} = \mathbf{y}_t - h(\hat{\mathbf{x}}_t, 0)$
    - compute the Jacobians $\mathbf{J}_{h,x}$ and $\mathbf{J}_{h,w}$
    - compute the Kalman gain using $\mathbf{K} = \mathbf{J}_{f,x}\mathbf{J}^T_{h,x}(\mathbf{J}_{h,x}\mathbf{J}_{f,x}\mathbf{J}^T_{h,x} + \mathbf{J}_{h,w}\mathbf{R}\mathbf{J}^T_{h,w})^{-1}$
    - update the state using $\hat{\mathbf{x}} = \hat{\mathbf{x}} + \mathbf{K}\boldsymbol{\epsilon}$
    - update the covariance using $(\mathbf{I} - \mathbf{K}\mathbf{J}_{h,x})\mathbf{J}_{f,x}$

Figure 16.15 shows an example of the extended Kalman filter tracking the function $h(x, y, z) = x + y$ with $f(x, y, z) = (y, z, -.5x(y + z))$.

The Extended Kalman Filter is not an optimal estimator. Further, if the assumptions about local linearity that underlie the linearisation are not true, then the estimate is very poor, and even where it is good, it requires the computation of the Jacobians, which is potentially difficult. There have been various attempts to improve upon it; one method that may be of interest is to choose a set of points that represent the statistics of the data and to transform them by passing them through the non-linear functions ($f(\cdot)$ and $h(\cdot)$), and then to compute the statistics of those points in order to estimate the statistics of the transformed data. This has the great name of the unscented transform, and can be used to produce an Unscented Kalman Filter. For more information on this, see the Further Reading section; instead we will look at a common MCMC algorithm for performing tracking, the particle filter.

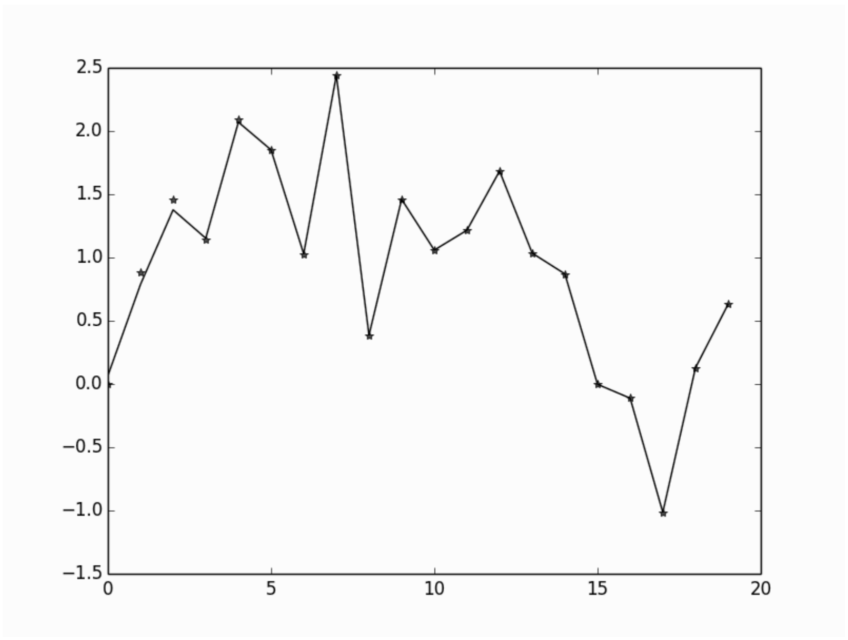

FIGURE 16.15 The extended Kalman filter tracking the function $h(x, y, z) = x + y$ with $f(x, y, z) = (y, z, -.5x(y + z))$.

16.4.2 The Particle Filter

As well as the linearity of the function, the Kalman filter also assumes that the distributions are Gaussian, so that they can be convolved and stay as Gaussians. In order to get around this problem, we return to the methods that have underpinned many of the algorithms in this chapter: sampling. The particular sampling technique that we will use is the sampling-importance-resampling algorithm of Section 15.3, which forms the basis of the particle filter, or condensation method. This is a relatively recent development, and has been finding many successful applications in tracking, including in image and signal analysis. The idea is to use sampling to keep track of the state of the probability distribution. This is known as sequential sampling, since we are using a set of samples for time t to estimate the process at time $t + 1$, and then resampling from there.

One benefit of sampling methods is that we don't have to hold on to the Markov assumption. In tracking, prior history can be useful, which means that the Markov assumption is a bad one. The proposal distribution is generally written as $q(\mathbf{x}_{t+1}|\mathbf{x}_{0:t}, \mathbf{y}_{0:t})$ to make this dependence clear, and the proposal distribution that is generally used in the estimated transition probabilities $p(\hat{\mathbf{x}}_{t+1}|\mathbf{x}_{0:t}, \mathbf{y}_{01:t})$, since it is a simple distribution that is related to the process. With this decided, there is very little else to the basic particle filter: a schematic of one iteration of the particle filter is shown in Figure 16.16. The basic algorithm is given next, followed by some points about the implementation and some examples.

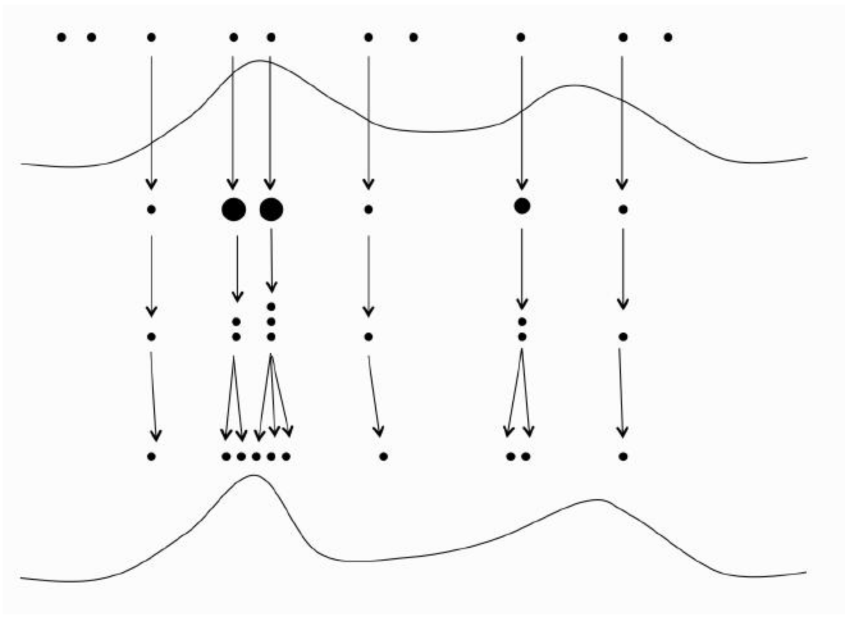

FIGURE 16.16 A schematic of one iteration of a particle filter. A set of random particle positioned have importance weights computed according to the distribution, and then new particles are created and modified based on these weights, and the estimated distribution is updated.

The Particle Filter Algorithm

- Sample $\mathbf{x}_0^{(i)}$ from $p(\mathbf{x}_0$ for $i = 1 \dots N$
- For each timestep:
  - **importance sample**
  - for each datapoint:
    * sample $\hat{\mathbf{x}}_t^{(i)}$ from $q(\mathbf{x}_t^{(i)}|\mathbf{x}_{0:t-1}^{(i)}, \mathbf{y}_{1:t})$
    * add $\hat{\mathbf{x}}_t^{(i)}$ onto the list of samples to get $\mathbf{x}_{0:t}^{(i)}$ from $\mathbf{x}_{0:t-1}^{(i)}$
    * compute the importance weights:

$$w_t^{(i)} = w_{t-1}^{(i)} \frac{p(\mathbf{y}_t|\hat{\mathbf{x}}_t^{(i)}) p(\mathbf{x}_t^{(i)}|\hat{\mathbf{x}}_{t-1}^{(i)})}{q(\mathbf{x}_t^{(i)}|\mathbf{x}_{0:t-1}^{(i)}, \mathbf{y}_{1:t})} \tag{16.38}$$

  - normalise the importance weights by dividing by their sum
  - **resample the particles**
    * retain particles according to their importance weights, so that there might be several copies of some particles, and none of others to get the same number of particles approximately sampled from $p(\mathbf{x}_{0:t}^{(i)}|\mathbf{y}_{1:t})$

The resampling part of this algorithm deserves a little more consideration, because there are several ways in which it can be done, and they differ in their computational time and the variance of the estimates. The code on the website provides two implementations: systematic resampling and residual resampling. The basic idea of systematic resampling is to use the cumulative sum of the weights (which will be 1 at the very end by definition) and a set of uniform random numbers $\tilde{u}_k$, put together so that they are in order using $u_k = (k = 1 + \tilde{u_k})/N$ and put n_i copies of particles i into the next set, where n_i is the number of u_k for which $\sum_{s=1}^{i-1} w_s \leq u_k < \sum_{s=1}^{i} w_s$. One way to implement this is using the following code:

```
def systematic(w,N):
    # Systematic resampling
    N2 = np.shape(w)[0]
    # One too many to make sure it is >1
    samples = np.random.rand(N+1)
    indices = np.arange(N+1)
    u = (samples+indices)/(N+1)
    cumw = np.cumsum(w)
    keep = np.zeros((N))
    # ni copies of particle xi where ni = number of u between ws[i-1] and ws[
    i]
    j = 0
    for i in range(N2):
        while((u[j]<cumw[i]) & (j<N)):
            keep[j] = i
            j+=1

    return keep
```

Residual sampling tries to speed this up (although it is an $\mathcal{O}(n)$ algorithm, it is called often, and so making it fast is a very good idea) by first using the integer part of Nw_i as an idea of how many copies of each particle i to keep, and then using stratified samples for the rest.

Figure 16.17 shows the particle filter keeping track of a distribution that changes at $t = 30$. The positions of the particles are shown as dots, the underlying state process is shown as the dashed line, and the observations are shown as pluses. The solid line is the hypothesised observation based on the mean of the particles. It can be seen that this tracks the observations very well.

As an example of using the Particle Filter for tracking, Figure 16.18 shows an object moving at constant speed being tracked in 2D. The weights are computed based on the Euclidean distance between each particle and the object, and the particles are resampled if the average distance between them grows too large.

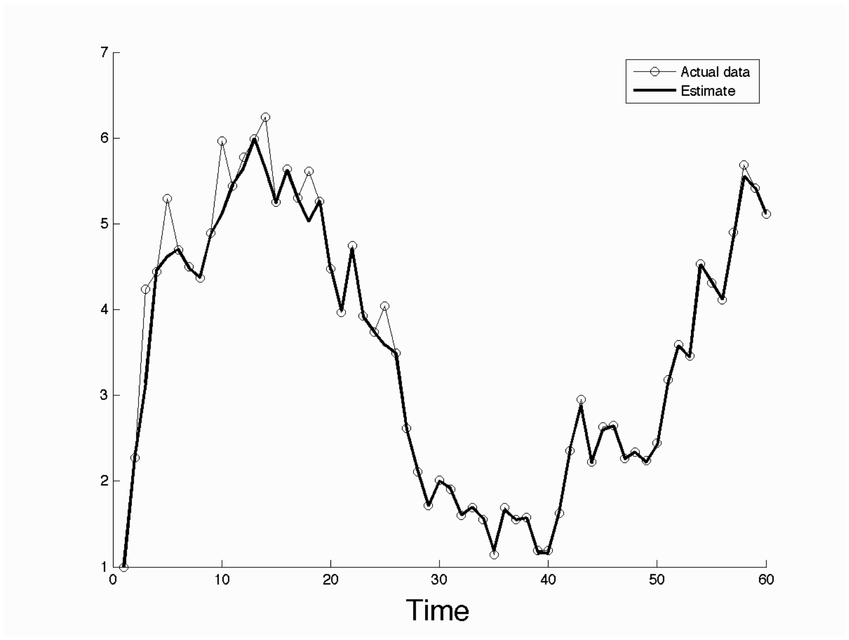

FIGURE 16.17 A particle filter tracking a 1D distribution that changes at $t = 30$. The observations are marked as pluses, the underlying state is shown by the dashed line, the particles at each iteration are shown as dots, and the hypothesised observation based on the mean of the particles is shown as a solid line.

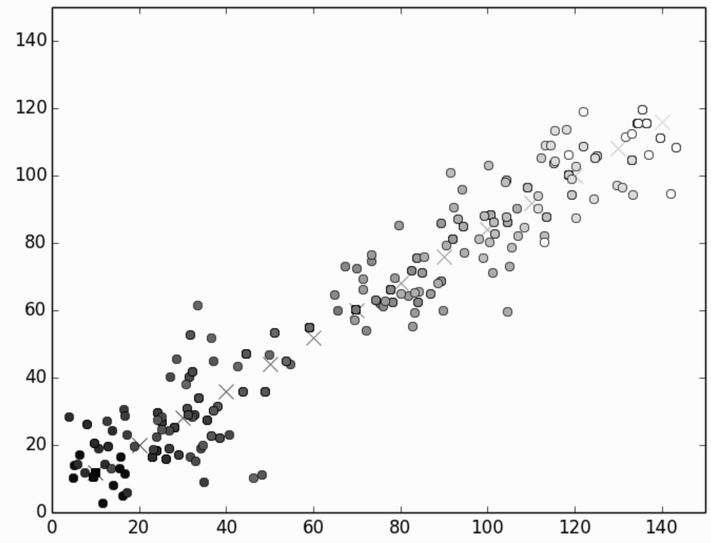

FIGURE 16.18 A particle filter tracking an object moving at a constant speed upwards and to the right in 2D. The crosses show the position of the object, and the circles show 15 particles at each iteration, starting at black ($t = 0$) and fading to white ($t = 10$). It can be seen that the particles track the object successfully.

FURTHER READING

Graphical models are a growth area at the moment, with lots of interesting research being done in the area. The original work in the area, and the motivations for it, are described in:

- J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference.* Morgan Kaufmann, San Mateo, CA, USA, 1988.

Alternative overviews of Bayesian networks can be found in the following papers and books, the last of which is a collection of papers that provides a good overview of the area:

- W.L. Buntine. Operations for learning with graphical models. *Journal of Artificial Intelligence Research*, 2:159–225, 1994.
- D. Husmeier. Introduction to learning Bayesian networks from data. In D. Husmeier, R. Dybowski, and S. Roberts, editors, *Probabilistic Modelling in Bioinformatics and Medical Informatics*, Springer, Berlin, Germany, 2005.
- Chapters 8 and 13 of C.M. Bishop. *Pattern Recognition and Machine Learning.* Springer, Berlin, Germany, 2006.
- M.I. Jordan, editor. *Learning in Graphical Models.* MIT Press, Cambridge, MA, USA, 1999.

In the area of Markov Random Fields, the image denoising example comes from:

- S. Geman and D. Geman. Stochastic relaxation, Gibbs distributions and the Bayesian restoration of images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 6:721–741, 1984.

Markov Random Fields are most commonly used in imaging. There are good overviews in:

- P. Pèrez. Markov random fields and images. *CWI Quarterly*, 11(4):413–437, 1998.
- R. Kindermann and J.L. Snell. *Markov Random Fields and Their Applications.* American Mathematical Society, Providence, RI, USA, 1980.

For more details on the Hidden Markov Model, the Kalman filter and the particle filter, you might want to look at:

- L.R. Rabiner. A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–268, 1989.
- Z. Ghahramani. An introduction to Hidden Markov Models and Bayesian networks. *International Journal of Pattern Recognition and Artificial Intelligence*, 15:9–42, 2001.
- G. Welch and G. Bishop. An introduction to the Kalman filter, 1995. URL `http://www.cs.unc.edu/~welch/kalman/`. Technical Report TR 95-041, Department of Computer Science, University of North Carolina at Chapel Hill, USA.
- M.S. Arulampalam, S. Maskell, N. Gordon, and T. Clapp. A tutorial on particle filters for online nonlinear/non-Gaussian Bayesian tracking. *IEEE Transactions on Signal Processing*, 50(2):174–188, 2002.

- S.J. Julier and J.K. Uhlmann. A new method for the nonlinear transformation of means and covariances in nonlinear filters. *IEEE Transactions on Automatic Control*, 45(3):477–482, 2000.
- R. van der Merwe, A. Doucet, N. de Freitas, and E. Wan. The unscented particle filter. In *Advances in Neural Information Processing Systems*, 2000. (the technical report version of this paper is particularly helpful).

A more detailed treatment is given in:

- Chapters 8 and 13 of C.M. Bishop. *Pattern Recognition and Machine Learning.* Springer, Berlin, Germany, 2006.

PRACTICE QUESTIONS

Problem 16.1 Compute the probability of taking notes (N) in the Bayesian network shown in Figure 16.19. The problem describes the chance of you taking notes in the lecture or sleeping (S) according to whether or not the course was boring (B) based on whether or not the professor is boring (L) and the content is dull (C). Compute the chance of falling asleep in a lecture given that both the professor and the course are boring.

Problem 16.2 Use MCMC in order to compute the chance of taking notes in a lecture given only that you know the lecture is interesting.

Problem 16.3 Compute the most likely path through the HMM shown in Figure 16.20 using the Viterbi algorithm.

Problem 16.4 Suppose that you notice a fairground show where the showman demonstrates that a coin is fair, but then makes it turn up heads many times in a row. You notice that he actually has two coins, and swaps between them with sleight-of-hand. Watching, you start to see that he sticks to the fair coin with probability 0.4, and to the biased coin with probability 0.1, and that the biased coin seems to come up heads about 85% of the time. Make a hidden Markov model for this problem, construct an observation sequence, and use the Viterbi algorithm to estimate the states.

Problem 16.5 On the website are a series of robot sensor readings. The aim is to predict the next reading from the current one by using a Perceptron, and then monitor the output of the Perceptron by using a Kalman filter in order to identify problem places where the prediction does not work.

Problem 16.6 Figure 16.18 performs tracking of an object in 2D using the Euclidean distance between each particle and the object to set the weights. Modify the code to compare this to using a binary weight based on proximity to the object where you will need to set a threshold to define proximity.

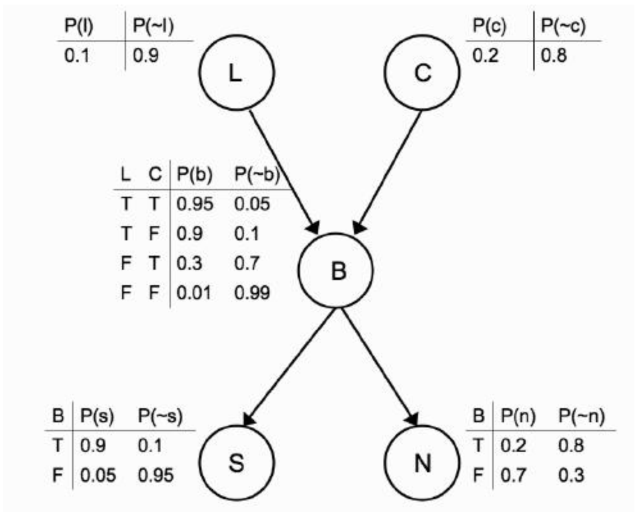

FIGURE 16.19 The Bayesian Network example for Problem 16.1.

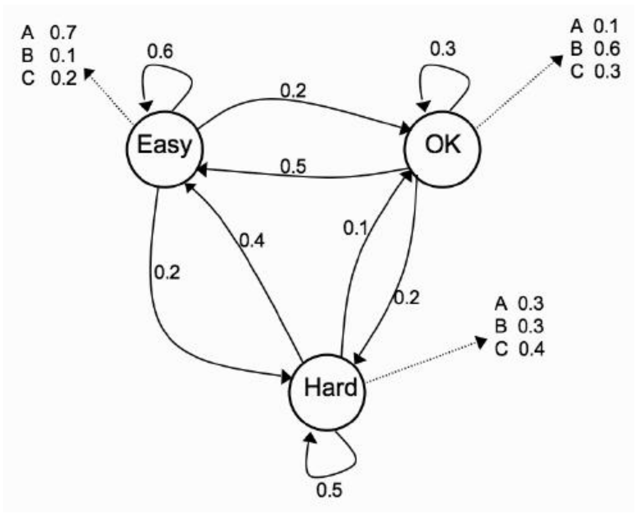

FIGURE 16.20 The Hidden Markov Model example for Problem 16.3.

CHAPTER 17

Symmetric Weights and Deep Belief Networks

Let's return to the model of the neuron that was the basis for the neural network algorithms such as the Perceptron in Chapter 3 and the multi-layer Perceptron in Chapter 4. These algorithms were based on what are effectively an integrate-and-fire model of a neuron, where the product of the inputs and the weights was compared with a threshold (generally zero when a bias node was also used) and the neuron produced a continuous approximation to firing (output one) or not firing (output zero) for each input. The approximation was typically the logistic function. The algorithms based on neurons that we have seen have been asymmetric in the sense that the values of the inputs and weights made the (hidden) neurons fire, or not, but the values of the neurons would never affect the inputs (in fact, these input nodes were never considered as neurons).

If we were to think of these networks as graphs, the edges would be directed, with the arrowhead pointing at the neurons from the inputs. This is shown on the left of Figure 17.1, where the shading suggests that the two sets of nodes are different, since the light-coloured nodes affect the firing of the dark-coloured nodes (based on the direction arrows on the links), but not vice-versa. On the right of the figure, the links are not directed, and so the nodes are all the same colour, since the firing of the nodes in the top layer, together with the weights on the edges, can be used to decide if the nodes in the lower layer fire or not, as well as the other way round.

However, the first learning rule that we saw (Hebb's rule in Section 3.1.1) was entirely symmetrical. It said that if two neurons fire at the same time then the synaptic connection between them gets stronger, while if they do not fire at the same time then the connection gets weaker. As we saw in that section, this means that if we only see one of two neurons with a connection between them, then we can decide what the other one does: if they are positively connected, then if the first one fires, so does the second one, and if they are negatively connected, then if the first one fires, the second does not. It doesn't matter which of the two neurons we see, since the connection between them is symmetric — if we use the usual notation of writing w_{mn} for the weight (strength) of the connection between neuron m and neuron n, then $w_{mn} = w_{nm}$, and so the weight matrix is symmetric.

In this chapter we explore a series of algorithms that are based on learning and using these symmetric weights. While in general such networks are potentially very complicated and hard to train, there are certain cases that are tractable and that turn out to be useful. The first network that we look into is an early—and conceptually very simple—network,

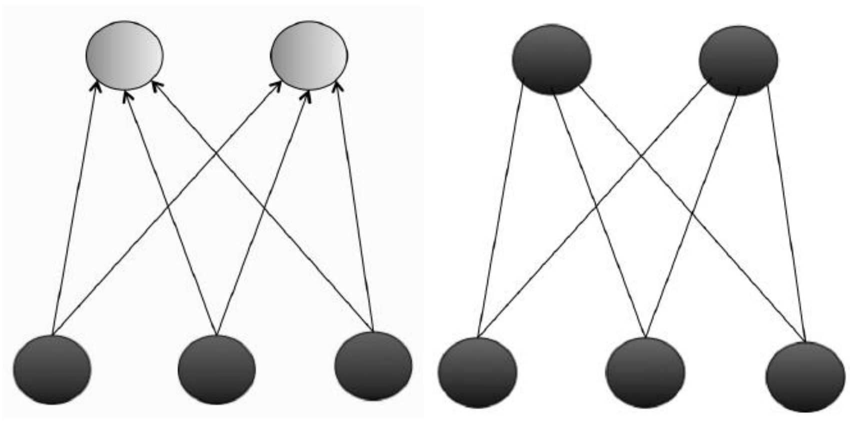

FIGURE 17.1 Two neural networks made up of two layers of nodes. On the *left* the weighted links are directional so that whether or not the lighter-coloured nodes fire affects the firing of the darker-coloured nodes, but not vice-versa, while on the *right* the two layers are symmetric, so that the firing of the upper layer can change the firing of the lower layer in the same way as from bottom to top.

which was described by John Hopfield in 1982. However, before seeing the network we will look at what it does.

17.1 ENERGETIC LEARNING: THE HOPFIELD NETWORK

17.1.1 Associative Memory

One thing that we know our brains are very good at is remembering things. This is clearly a crucial part of learning: recognising things that we have seen before. There are many different types of memory, but one of the most useful is associative memory. An associative memory, also known as a context-addressable memory, works by learning a set of patterns in such a way that if you see a new pattern, the memory reproduces whichever of the stored patterns most closely resembles it. We use this all the time — for example, when we recognise the letters of the alphabet that are written in some font that we haven't seen before, or when we recognise a person when we see them from an unusual angle, or decide that one person looks like another. It can be thought of as completing or correcting inputs (depending upon whether there are missing values, or incorrect values in the input).

So it would be worth studying context-addressable memories just because they are clearly important in the brain. However, there are some more practical applications of them, too. For example, we can use them to remove the noise from images, or reconstruct the full image. We train the memory by showing it the complete pictures, and then, when we show the memory a noisy (i.e., corrupted with some errors), or partial (with missing values) version it will reproduce the original version.

As an example of the things that you can do with an associative memory, suppose that we learn a list of pairs of words that are associated together, such as:

```
Humphrey - Bogart
Ingrid - Bergman
Paul - Henreid
```

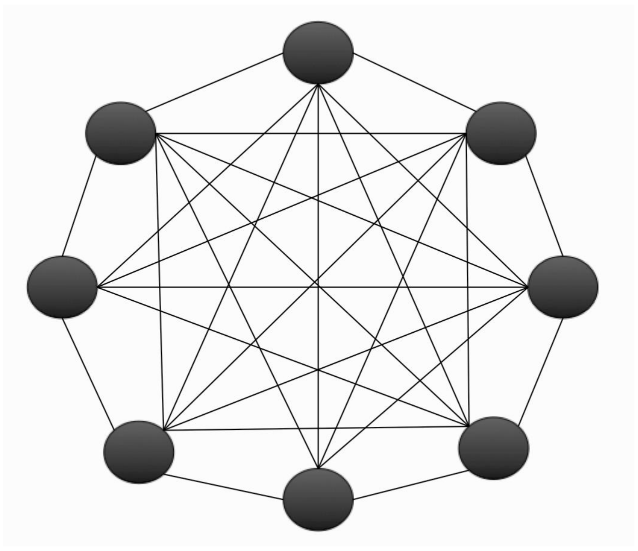

FIGURE 17.2 Schematic of the Hopfield network, which is a fully connected set of neurons with symmetric weights.

```
Claude - Rains
Omar - Sharif
Julie - Christie
```

The two things that we can do with an associative memory are pattern completion, so that if we saw ‘Ingrid’ then the memory can provide ‘Bergman’, and pattern denoising, so that if we saw ‘Hungry – Braggart’ then the memory can correct it to ‘Humphrey – Bogart’. Of course, we can confuse it by adding memories that overlap. So if

```
Paul - Newman
```

was added to the memory, then looking for ‘Paul’ would produce one of the two (‘Henreid’ or ‘Newman’) arbitrarily.

Having understood what the memory does, let’s work out how to make one.

17.1.2 Making an Associative Memory

The Hopfield network consists of a set of McCulloch and Pitts neurons that are fully connected with symmetric weights, so that every neuron is connected to every other neuron, except for itself (so that $w_{ii} = 0$). In the brain there are cases where neurons have synapses that connect back to themselves, but we’ll ignore that here. There is a picture of the Hopfield network in figure 17.2.

McCulloch and Pitts neurons are binary, they either fire or don’t fire, but rather than using 1 and 0 to encode these outputs, it is more convenient to use 1 and -1 instead. One reason for this is that it gives us Hebb’s rule very simply. If $s_i^{(t)}$ is the activation of neuron i at time t, then we can write Hebb’s rule as:

$$\frac{dw_{ij}^t}{dt} = s_i^{(t)} s_j^{(t)}, \tag{17.1}$$

so that if two neurons have the same behaviour (firing or not firing) then the weight increases, while if they have opposite behaviour then the weight decreases.

Note the time superscripts in Equation (17.1), which imply that the weight update is based on the activations of the neurons at the current time. However, things are not as clear when we consider the update of each neuron to decide whether or not it is firing. The reason for this is that the weights are symmetric, and so there is no pre-defined order in which the neurons should be updated. For a directed network, we start at the inputs, and use those values and the weights to decide whether or not the next layer fire, and so on until we get to the outputs. However, for symmetric weights there is no clearly defined order. This means that there are two different ways in which we can do the update, and they can sometimes produce different output behaviours. One method is to use:

$$s_i^{(t)} = \text{sign}\left(\sum_j w_{ij} s_j^{(t-1)}\right), \tag{17.2}$$

which describes synchronous update where effectively every neuron has its value updated at the same time; imagine all of the neurons making a simultaneous decision about whether or not to fire at the next time step. The second version is asynchronous, each neuron makes a decision about when to fire based on the current values — either $s_j^{(t-1)}$ or $s_j^{(t)}$ depending which is available — of all of the other neurons. The order in which the neurons are updated could be either random or in some fixed sequence. In either case, it can be necessary to run the update for several steps to ensure that the network settles into a steady state. This is sufficient to enable the network to recall previous inputs.

Regardless of which form of update is used, it is still normal to set the threshold for each neuron to fire at 0, so that the decision about whether or not each will fire is:

$$s_i = \text{sign}\left(\sum_j w_{ij} s_j\right), \tag{17.3}$$

where sign(·) is a function that returns 1 if the input is greater than 0, and -1 otherwise. It is possible to include a bias node (which is a node with a constant value, usually ± 1) to change this value in the normal way if required.

We give an input to the Hopfield network by setting the activations of the neurons (the s_i) and then running the update equation until the neurons stop changing values. So once the weights are set, the remembering is very simple. The learning is also simple: the Hopfield network learns the weights using Hebb's rule:

$$w_{ij} = \frac{1}{N}\sum_{n=1}^{N} s_i(n) s_j(n), \tag{17.4}$$

where N is the number of patterns that we want the network to learn and $s_i(n)$ is the activation of neuron i for input pattern n. There is no t superscript since we are setting the values of the neurons, not waiting for them to update.

The $\frac{1}{N}$ takes the place of the learning rate η in previous learning algorithms. It doesn't have much effect for the Hopfield network, since learning is effectively one-shot, but by using $\frac{1}{N}$ the maximum size of the weights is independent of N.

Suppose that we are learning to reproduce binary images, as in Section 4.4.5. We have one neuron for each pixel of the picture, and we equate $s_i = 1$ with black pixels and $s_i = -1$ with white pixels. By considering the case of one input pattern, we can now see that the

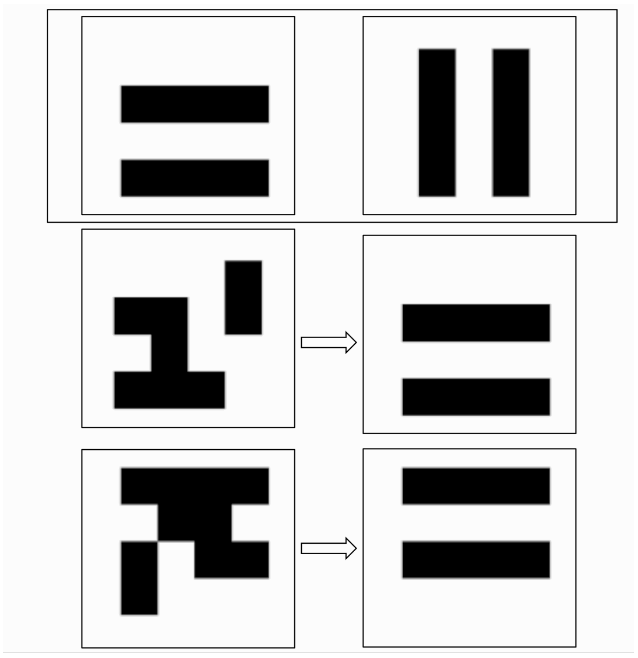

FIGURE 17.3 The Hopfield network is trained on the top two images. When the first image on the second row is presented it settles back to the first image on the top row. However, when the second image is presented, it settles to the third image, which is the inverse of the first input pattern.

Hopfield network does perform image correction. If we set the weights based on an image and then present a noisy version of that image to the network, then each of the neurons will have its activity updated by all of the other neurons. Many of the neurons will change state from 1 to -1 and back again several times until the network settles into its final state (stabilises). So what will the network stabilise to? Well, providing that more than half of the initial bits are correct, on average the total input to each neuron will be more than half correct, and this will overwhelm the errors. This means that the correct pattern is an attractor: if we let the network settle into its final state, that final state will correspond to the pattern that the network learned.

What if more than half of the inputs are incorrect? That means that black pixels are white, and white pixels are black. In this case, the network will settle to the inverse of the pattern, i.e., the pattern where black and white have been swapped throughout the image (we will see an example of this in Figure 17.3). If we label the first, correct pattern as x, this second pattern is $-x$. This is also an attractor.

Of course, the network actually learns about many input patterns (N of them) and so there will be many attractors. For this reason, it is not guaranteed to find the 'correct' answer, since the noisy input may actually be closer to another of the trained patterns that the one you expect. This suggests a question that we need to consider, which is: how many memories can the Hopfield network hold; that is, how many different patterns can the network remember?

However, before analysing the network further, and answering this question, we will need a sidetrack, so first here is the complete algorithm:

The Hopfield Algorithm

- **Learning**
  - take a training set of N d-dimensional inputs $\mathbf{x}(1), \mathbf{x}(2), \ldots, \mathbf{x}(N)$ with elements ± 1
  - create a set of d neurons (or $d+1$ including a bias node that is permanently set to 1) and set the weights to:

$$w_{ij} = \begin{cases} \frac{1}{N}\sum_{n=1}^{N} x_i(n)x_j(n) & \forall i \neq j \\ 0 & \forall i = j \end{cases} \tag{17.5}$$

- **Recall**
  - present the new input $\mathbf{x}$ by setting the states s_i of the neurons to x_i
  - **repeat**
    - update the neurons using:

$$s_i^{(t)} = \text{sign}\left(\sum_j w_{ij} s_j^*\right), \tag{17.6}$$

      where s_j^* is $s_j^{(t)}$ if that neuron has been updated already, and $s_j^{(t-1)}$ if it has not. This means that for synchronous update $s_j^* = s_j^{(t-1)}$ for every node, while for asynchronous update it could be either value.
  - **until** the network stabilises
  - read off the states s_i of the neurons as the output

One way to implement the different forms of update is shown in the following code snippet, which makes clear the difference between the synchronous and asynchronous versions:

```
def update_neurons(self):
    if self.synchronous:
        act = np.sum(self.weights*self.activations,axis=1)
        self.activations = np.where(act>0,1,-1)
    else:
        order = np.arange(self.nneurons)
        if self.random:
            np.random.shuffle(order)
            for i in order:
                if np.sum(self.weights[i,:]*self.activations)>0:
                    self.activations[i] = 1
                else:
                    self.activations[i] = -1
    return self.activations
```

Figure 17.4 shows an example of a Hopfield network being used. The network is trained on an image of each of the digits 0 to 9 taken from the 'Binary Alphadigits' dataset (a web link is provided on the book website). These are 20 × 16 binary images, of the digits '0' to '9' and 'A' to 'Z' and the images used are shown in the top row of the figure.

Following the setting of the weights, the network was presented with the corrupted version of the image of the '2', as shown in the next line, and then performed asynchronous random updating to settle back to the original, trained, image.

17.1.3 An Energy Function

We will return to the question of the capacity of the network shortly, but first let's investigate one of the major contributions that Hopfield made, which was to write down an energy function for his network. Energy functions are used in physics to compute how much energy a system has, with the idea that systems relax into low energy states, for example the way that sets of chemicals mix together to form stable compounds. The energy function for the Hopfield network with d neurons (where d may or may not include a bias node) is (using the usual matrix notation in the second line):

$$\begin{aligned} H &= -\frac{1}{2}\sum_{i=1}^{d}\sum_{j=1}^{d} w_{ij}s_is_j \\ &= -\frac{1}{2}\mathbf{s}\mathbf{W}\mathbf{s}^T \end{aligned} \tag{17.7}$$

The reason for the $\frac{1}{2}$ is that each weight is counted twice in the double sum, since $w_{ij}s_is_j$ is the same as $w_{ji}s_js_i$, and the negative signs are because we consider that finding lower energies is better. Note that in the Hopfield network nodes are not connected to themselves, and so $w_{ii} = 0$.

To see what the energy function computes, consider the case where the network has learnt one training example, and it sees it again. In this case, $w_{ij} = s_is_j$, and since $s_i^2 = 1$ no matter whether $s_i = -1$ or $s_i = 1$:

$$\begin{aligned} H &= -\frac{1}{2}\sum_{i=1}^{d}\sum_{j=1}^{d} w_{ij}s_is_j \\ &= -\frac{1}{2}\sum_{i=1}^{d}\sum_{j=1,j\neq i}^{d} s_is_js_is_j \\ &= -\frac{1}{2}\sum_{i=1}^{d}\sum_{j=1,j\neq i}^{d} s_i^2s_j^2 \\ &= -\frac{1}{2}\sum_{i=1}^{d}\sum_{j=1,j\neq i}^{d} 1 \\ &= -\frac{d(d-1)}{2}. \end{aligned} \tag{17.8}$$

Note the insistence that the second sum is only over $j \neq i$ from the second line on to account for the fact that $w_{ii} = 0$.

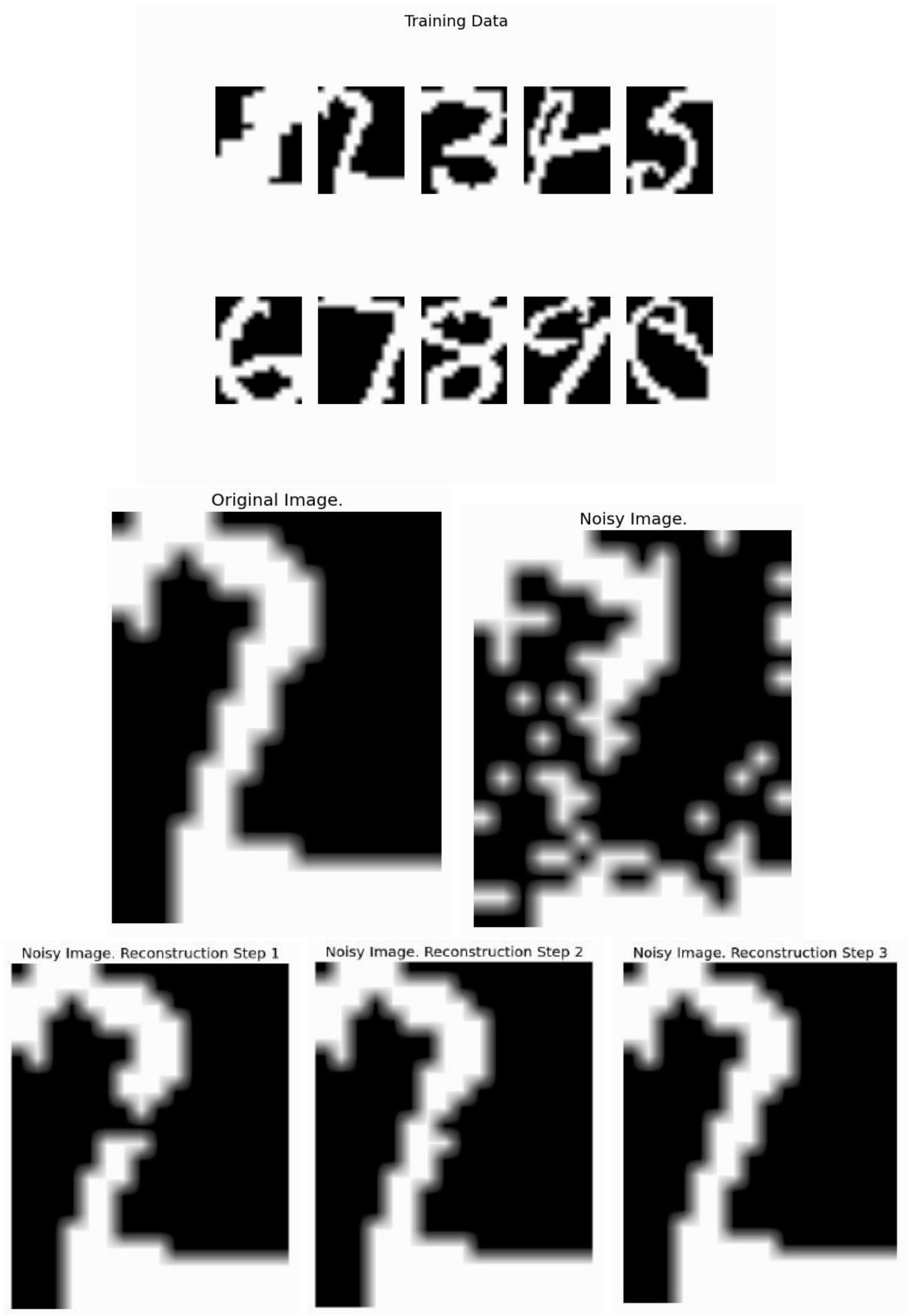
Training Data
Original Image.
Noisy Image.
Noisy Image. Reconstruction Step 1
Noisy Image. Reconstruction Step 2
Noisy Image. Reconstruction Step 3

FIGURE 17.4 Example of the Hopfield Network reconstructing a noisy version of an image. The network was trained on the images at the top. It was then presented with the image on the right of the second row, which is the image on the left of that row, but with 50 bits corrupted by being inverted (so black becomes white and vice-versa). Three iterations of the algorithm running are shown in the third row.

Clearly, any place where the weights and the neurons disagree will give a contribution with +1 instead of -1, and so the total sum will have a higher (less negative) value. Thus, the energy function describes the amount of mismatch between the values of the neurons and what the weights say they should be.

An energy function sets the neural activity of the network into terms that physicists have used for a long time, and indeed, physicists immediately recognised this equation – it is (a slight variation of) the energy of the Ising Spin Glasses, which are a simple model of magnetic materials, and have been very well studied. Once we have an energy function, we can watch how it changes as the network stabilises.

We have just seen that once the network is stable so that the weights and neuron activations agree, the energy is at a minimum, and while we wait for it to stabilise, the energy is higher. This means that the attractors (stable patterns) are local minima of the error function, just as in previous discussions about such functions. We can imagine the change in energy as the network learns as an energy landscape (for the example shown in Figure 17.4 the energy went from -1119.0 to -6447.8). If we think about a ball bearing rolling around on this landscape under gravity, then the ball will roll downhill until it gets to the bottom of some hollow, where it will stop. If there are lots of hollows, then which hollow it will end up in depends on where it starts in the landscape. This is what happens when many images are stored in the network — there are many different hollows relating to each of the different images. The area around an attractor, from where the ball bearing will roll into that hollow, is called the basin of attraction.

Another benefit of having an energy function is that we can see that the network will eventually reach a stable state at a local minimum. (In fact, this is only true for asynchronous update; for synchronous update there are cases that never converge.) To see that the network will reach a local minima, we need to consider an update step where the value of bit i changes. In this case, the network has gone from state $\mathbf{s} = (s_1, \ldots s_i, \ldots s_d)$ to state $\mathbf{s}' = (s_1, \ldots s_i', \ldots s_d)$ and the energy has gone from $H(\mathbf{s})$ to $H(\mathbf{s}')$. If we consider this difference, we get:

$$\begin{aligned} H(\mathbf{s}) - H(\mathbf{s}') &= -\frac{1}{2}\sum_{j=1}^{d} w_{ij} s_i s_j + \frac{1}{2}\sum_{j=1}^{d} w_{ij} s_i' s_j \qquad (17.9) \\ &= -\frac{1}{2}(s_i - s_i')\sum_{j=1}^{d} w_{ij} s_j \qquad (17.10) \end{aligned}$$

The reason that the bit flipped is because the combination of the weights and the values of the other neurons disagreed with its current value, so s_i and $\sum_{j=1}^{d} w_{ij} s_j$ must have opposite signs. Likewise, s_i and s_i' have opposite signs since the bit flipped. Hence, $H(\mathbf{s}) - H(\mathbf{s}')$ must be positive, and so the total energy of the network has decreased. This means that while the energy of the network continues to change, it will decrease towards a minimum. There is no guarantee that the network will reach a global optimum, though.

17.1.4 Capacity of the Hopfield Network

The question of how many different memories the Hopfield network can store is obviously an important one. Fortunately, it is fairly simple to answer by considering the stability of a single neuron in the network, which we will label as the ith one. Suppose that the network has learnt about input $\mathbf{x}(n)$ already. If that input is presented to the network again, then the output of neuron i will be (using Equation 17.4 and ignoring the $\frac{1}{N}$ throughout since it is just a scaling factor):

$$\begin{aligned} s_i &= \sum_{j=1}^{d} w_{ij} x_j(n) \\ &= \sum_{j=1, j\neq i}^{d} \left((x_i(n)x_j(n))\, x_j(n) + \left(\sum_{m=1, m\neq n}^{N} x_i(m)x_j(m) \right) x_j(n) \right) \\ &= x_i(n)(d-1) + \sum_{j=1, j\neq i}^{d} \sum_{m=1, m\neq n}^{N} x_i(m)x_j(m)x_j(n) \end{aligned} \tag{17.11}$$

The output that we want is $s_i = x_i(n)$, which is the ($d-1$ times) the value of the first term, and so we ideally want the second term to be zero. Note that there are $(d-1)\times(N-1)$ terms to be summed up.

The possible values for each x_i are ± 1, and we assume that each neuron has an equal chance of either value, which means that we can model them as random binary variables with zero mean and unit variance. Thus, the total sum has mean $(d-1)x_i(n)$ (that is, either $-(d-1)$ or $d-1$) and variance $(d-1)\times(N-1)$ (and hence standard deviation $\sqrt{(d-1)(N-1)}$) and it is approximately Gaussian distributed. So what is the probability that bit i flips state? This happens in the tails of the Gaussian distribution, and we can compute it using the Gaussian cumulative distribution function (where $t = \frac{-(d-1)}{\sqrt{(d-1)(N-1)}}$) as:

$$P(\text{bit } i \text{ flips}) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{t} e^{-t^2/2} dt \tag{17.12}$$

This means that it is the ratio of $d-1$ and $N-1$ that is of interest, which is not surprising since more neurons should be able to hold more memories. There is always a chance that a bit will flip. If we are prepared to accept a 1% chance of this, then Equation (17.12) tells us that we can store about $N \approx 0.18d$ patterns. However, we aren't quite finished. The computation we just did was about the probability of the bit flipping on the first iteration of the update rule. A more involved analysis eventually tells us that with that same error rate of 1% the network can store about $N \approx 0.138d$ patterns. So for the 20×16 images that are learnt in the example dataset, there are 320 neurons, so it should be able to hold around 44 images.

17.1.5 The Continuous Hopfield Network

There is a continuous version of the Hopfield network, which can be adapted to deal with more interesting data, such as grayscale images. The basic difference is that the neurons need to have an activation function that runs between -1 and 1. This could be done with a scaled version of the logistic (sigmoidal) function that was used in earlier chapters, but instead it is convenient to use the hyperbolic tangent function, which is already in that range. In NumPy this is computed with `np.tanh()`. Changing the Hopfield network to use these neurons is one of the exercises for this chapter.

In the continuous Hopfield network we make the same type of transformation as we did for the MLP, using a continuous activation function instead of the binary threshold. This changes the function of the network so that instead of minimising the energy function

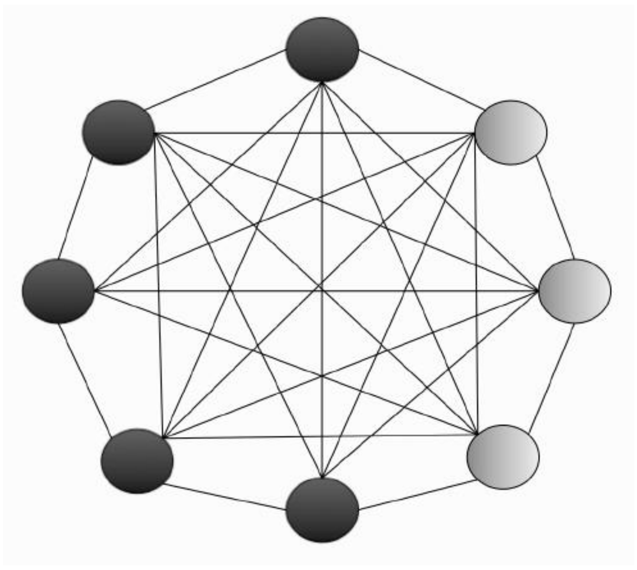

FIGURE 17.5 A schematic of a BM network. All the nodes are fully connected, but there are visible nodes (in dark grey) and hidden nodes.

in Equation (17.7), it approximates the probability distribution that matches that energy function:

$$p(\mathbf{x}|\mathbf{W}) = \frac{1}{Z(\mathbf{W})} \exp\left[\frac{1}{2}\mathbf{x}^T\mathbf{W}\mathbf{x}\right], \tag{17.13}$$

where $Z(\mathbf{W}) = \sum_{\mathbf{x}} \exp\left(\frac{1}{2}\mathbf{x}^T\mathbf{W}\mathbf{x}\right)$ is a normalising function.

We can actually compute the distribution in Equation (17.13) directly if we transform the neuron activations to lie between 0 and 1 instead of -1 and 1, and then interpret them as the probability of the neuron firing, so that 0 means that the neuron does not fire and 1 means that it definitely does. This means that we have a stochastic neuron, which fires with probability $1/(1+e^{-x})$. A network based on stochastic neurons like this is known as a Boltzmann machine. It produces Gibbs samples (see Section 15.4.4) of precisely this probability distribution, and it is described next.

17.2 STOCHASTIC NEURONS — THE BOLTZMANN MACHINE

The original Boltzmann machine is fully connected just like the Hopfield network. However, in addition to the neurons where the input can be specified (and the output read off), which are termed visible neurons, there are also hidden neurons, which are free to take on any role in the computational model. Figure 17.5 shows a schematic of a possible Boltzmann machine, which is fully connected, with a set of visible neurons and a set of hidden nodes.

In order to see how to train a Boltzmann machine, let's start with a stochastic Hopfield network, where there are no hidden nodes. We will label the states of these visible nodes as $\mathbf{v}$. We know that the network is sampling from:

$$P(\mathbf{v}|\mathbf{W}) = \frac{1}{Z(\mathbf{W})} \exp\left(\frac{1}{2}\mathbf{v}^T\mathbf{W}\mathbf{v}\right), \tag{17.14}$$

Therefore, learning consists of modifying the weight matrix $\mathbf{W}$ so that the generative model in Equation 17.14 matches the training data $(\mathbf{x}^n)$ well. We want to maximise the likelihood of this, which we can do by computing the derivative of the log of it with respect to each weight w_{ij}:

$$\begin{aligned}\frac{\partial}{\partial w_{ij}} \log \prod n = 1^N P(\mathbf{x}^n|\mathbf{W}) &= \frac{\partial}{\partial w_{ij}} \sum_{n=1}^{N} \frac{1}{2}(\mathbf{x}^n)^T \mathbf{W}\mathbf{x}^n - \log Z(\mathbf{W}) & (17.15)\\ &= \sum_{n=1}^{N} \left(x_i^n x_j^n - x_i^n x_j^n P(\mathbf{x}|\mathbf{W})\right) & (17.16)\\ &= N\langle x_i x_j\rangle_{\text{data}} - \langle x_i x_j\rangle_{P(\mathbf{x}|\mathbf{W})}, & (17.17)\end{aligned}$$

where in the last line the $\langle , \rangle$ notation means the average, so that the first term is the average over the data, and the second term is the average over samples from the probability distribution.

This means that the gradient with respect to each weight is proportional to the difference between two correlations: the correlation between two values in the data (the empirical correlation) and the correlation between them in the current model. Clearly, the first of these is easy to calculate, but the second isn't easy to evaluate directly. However, it can be estimated by sampling from the model as the Boltzmann machine iterates.

Hinton, who first described the Boltzmann machine, describes the algorithm for computing these two parts as a 'wake-sleep' algorithm. Initially the algorithm is awake, so it computes the correlation in the data. The network then falls asleep and 'dreams' about the data that it has seen, which lets it estimate the correlations in the model.

We can extend this derivation to a network with hidden neurons as well. We will label the states of the set of hidden neurons as $\mathbf{h}$, so that the state of the whole network is $\mathbf{y} = (\mathbf{v}, \mathbf{h})$. Remember that the hidden states are unknown. The likelihood of the weight matrix $\mathbf{W}$ given a data example $\mathbf{x}^n$ is (where Z now has to sum over the hidden nodes $\mathbf{h}$ as well as the visible nodes):

$$P(\mathbf{x}^n|\mathbf{W}) = \sum_{h} P(\mathbf{x}^n, \mathbf{h}|\mathbf{W}) = \sum_{\mathbf{h}} \frac{1}{Z(\mathbf{W})} \exp\left(\frac{1}{2}(\mathbf{y}^n)^T \mathbf{W}\mathbf{y}^n\right). \tag{17.18}$$

Computing the derivative of the log likelihood as before, the gradient with respect to a weight w_{ij} has a similar form of the difference between two terms that are found while the network is asleep and awake:

$$\frac{\partial}{\partial w_{ij}} \log P(\{\mathbf{x}^n\}|\mathbf{W}) = \sum_{n} \left(\langle y_i y_j\rangle_{P(\mathbf{h}|\mathbf{x}^n, \mathbf{W})} - \langle y_i y_j\rangle_{P(\mathbf{v}, \mathbf{h}|\mathbf{W})}\right). \tag{17.19}$$

Unfortunately, now both of these correlations have to be sampled by running the network for many iterations, where each iteration has two stages: in the first, awake, stage the visible neurons have the values clamped to the input (that is, they are held fixed), while the hidden nodes are allowed to take any values under their model, while in the second, asleep, stage both sets are sampled from the model.

In Chapter 15 we saw one possible way to get around problems like this, by setting up a Markov chain that converges to the correct probability distribution. If we let the chain run until equilibrium then the result will be samples from the distribution, and so we can approximate the average over the distribution by an average over these samples.

Unfortunately, this is still computationally very expensive since it will take lots of steps for the Markov chain to converge at each stage.

However, using this, training the algorithm consists of clamping the visible nodes to the input, and then using Gibbs sampling (see Section 15.4.4) until the network settles, then computing the first of the two terms in Equation (17.19). Following this, the visible nodes are allowed to go free as well and the whole distribution is sampled from, first the visible nodes and then the hidden nodes, up and down through the network, until it converges again, and then computing the second term. The weights can then be trained by using Equation (17.19).

The Boltzmann machine is a very interesting neural network, particularly because it produces a generative probabilistic model (which is what is sampled from for the weight updates). However, it is computationally very expensive since every learning step involves Monte Carlo sampling from the two distributions. In fact, since it is based on symmetric connections the Boltzmann machine is actually a Markov Random Field (MRF) (see Section 16.2). The computational expense has meant that the normal Boltzmann machine has never been popular in practical use, which is why we haven't examined an implementation of it. However, a simplification of it has, and we will consider that next.

17.2.1 The Restricted Boltzmann Machine

The main computational expense in the algorithm above is that both of the correlations have to be found by sampling, which means that the algorithm has to run for lots of iterations so that it converges at each step of the learning. This is progressively more expensive as the number of nodes grows, so that the algorithm scales very badly. However, some of this sampling was unnecessary for the machine that didn't have hidden nodes, where it is possible to compute the first term from the data alone. Ideally, we would like to find some simplification of the machine that allows for this to happen, and that also makes the number of sampling steps required to estimate the second correlation reasonably small.

It turns out that it is the interconnections *within* each layer that cause some of the problems. Without these interconnections, the nodes in a layer are conditionally independent of each other, given that the nodes in the other layer are held constant. In other words, by treating each hidden neuron as an individual 'expert' and multiplying together their distributions for each visible neuron (or taking logs and adding them) it is possible to compute the probability distribution for each visible neuron. Similarly, the distribution over the hidden nodes can be computed given the visible nodes.

So the restricted Boltzmann machine (RBM) consists of two layers of nodes, the visible ones (which are clamped to the inputs during the 'awake' training) and the hidden ones. There are no interconnections within either of the two layers, but they are fully connected between them using the symmetric weights. This is known as a bipartite graph and an example for three input nodes and two hidden nodes is shown in Figure 17.6. This machine was originally known by the delightful name of Harmonium, but the RBM name has stuck, not least because the tractable training algorithm is conceptually very similar to that for the full Boltzmann machine.

The algorithm that is used to train RBMs is known as the Contrastive Divergence (CD) algorithm, and it was created by Hinton and colleagues, who also first proposed the Boltzmann machine. The algorithm is a wake-sleep algorithm, and it will be given shortly, before we consider a numerical example, derive the update equations, and consider some implementation issues.

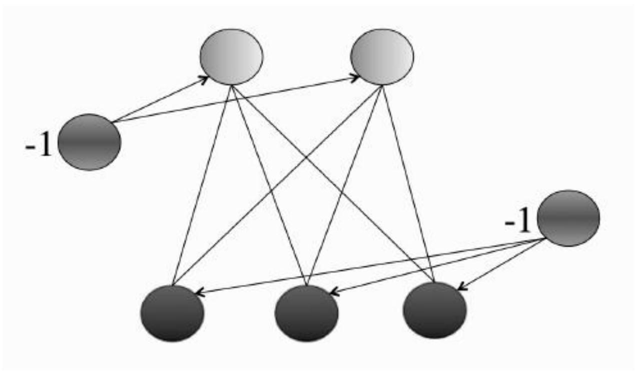

FIGURE 17.6 A schematic of the RBM network. There are two layers of nodes (the visible ones and the hidden ones) that are connected with symmetric weights.

The Restricted Boltzmann Machine

- **Initialisation**
  - initialise all weights to small (positive and negative) random values, usually with zero means and standard deviation about 0.01
- **Contrastive Divergence Learning**
  - take a training set of d-dimensional inputs $\mathbf{x}(1), \mathbf{x}(2), \ldots, \mathbf{x}(N)$ with elements ± 1
  - for some number of training epochs or until the error gets small:
    * Awake phase
    * clamp the visible nodes to the values of one of the input vectors
    * compute the probability of each hidden node j firing as (where b_j is the bias input to hidden node h_j):

$$p(h_j = 1|\mathbf{v}, \mathbf{W}) = 1/(1 + \exp(-b_j - \sum_{i=1}^{d} v_i w_{ij})) \qquad (17.20)$$

    * create a random sample to decide if each hidden node h_j fires
    * compute CDpos $= \frac{1}{N} \sum_i \sum_j v_i h_j$
    * Asleep phase
    * for some number of CD steps:
      · re-estimate v_i using (where a_i is the bias input to hidden node h_i):

$$p(v'_i = 1|\mathbf{h}, \mathbf{W}) = 1/(1 + \exp(-a_i - \sum_{j=1}^{n} w_{ij} h_j)) \qquad (17.21)$$

      · create a random sample to decide if each visible node v_i fires

            · re-estimate h_j using:

$$p(h'_j = 1|\mathbf{v}, \mathbf{W}) = 1/(1 + \exp(-\sum_{i=1}^{d} v'_i w_{ij})) \tag{17.22}$$

            · create a random sample to decide if each hidden node h_j fires

        * use the current values of v_i and h_j to compute CDneg $= \frac{1}{N}\sum_i \sum_j v_i h_j$

    – Weight Update

    – update the weights with (where η is the learning rate, m is the momentum size, and τ is the current step and $\tau - 1$ is the previous one):

$$\begin{aligned} \Delta w_{ij}^{\tau} &= \eta(CDpos - CDneg) + m\Delta w_{ij}^{\tau-1} && (17.23)\\ w_{ij} &\leftarrow \Delta w_{ij}^{\tau} && (17.24)\end{aligned}$$

    – update the bias weights with the same learning rule, but based on (where $\mathbf{v}^n$ is the vector of visible values for the nth input, and similarly for $\mathbf{h}^n$):

$$\begin{aligned} CDpos_{\text{visible bias}} &= \sum_{n=1}^{N} \mathbf{x}(n) && (17.25)\\ CDneg_{\text{visible bias}} &= \sum_{n=1}^{N} \mathbf{v}^n && (17.26)\\ CDpos_{\text{hidden bias}} &= \sum_{n=1}^{N} \mathbf{x}(n) && (17.27)\\ CDneg_{\text{hidden bias}} &= \sum_{n=1}^{N} \mathbf{h}^n && (17.28)\\ & && (17.29)\end{aligned}$$

    – compute an error term (such as the reconstruction error $\sum_i (v_i - v'_i)^2$)

- **Recall**

    – clamp an input $\mathbf{x}$ by setting the states of the nodes v_i to x_i

        * compute the activation of each hidden node j as:

$$p(h_j = 1|\mathbf{v}, \mathbf{W}) = 1/(1 + \exp(-\sum_{i=1}^{d} v_i w_{ij})) \tag{17.30}$$

        * create a random sample to decide if each hidden node h_j fires

        * re-estimate v_i using:

$$p(v'_i = 1|\mathbf{h}, \mathbf{W}) = 1/(1 + \exp(-\sum_{j=1}^{n} w_{ij} h_j)) \tag{17.31}$$

        * create a random sample to decide if each visible node v_i fires

This is a pretty complicated algorithm, so let's consider a simple example of using an RBM to illustrate it. Suppose that students need to choose three out of five possible courses for a particular part of their degree. The courses are Software Engineering (SE), Machine Learning (ML), Human-Computer Interaction (HCI), Discrete Maths (DM), and Databases (DB). If a tutor wants to help students by recommending which courses would suit them better, then it could be useful to separate the students according to whether they prefer programming or information technology. Looking at a few examples from previous years, the tutor might see examples like: (ML, DM, DB) and (SE, ML, DB) for the more programming-orientated students and (SE, HCI, DB) for those that prefer IT. There will, of course, be students who like both types of courses, and their choices would look more mixed.

Given only the list of which courses students took, the RBM can be used to identify the clusters in the data and then, given the information about what kind of student they are, it can suggest which courses they might choose.

Courses				
SE	ML	HCI	DM	DB
0	1	0	1	1
1	1	0	1	0
1	1	0	0	1
1	1	0	0	1
1	0	1	0	1
1	1	1	0	0

On a very simple, small example like this we can see the effects of the computations very simply. The wake phase of the algorithm uses these inputs to compute the probabilities, and then sample activations, for the hidden layer, initially based on random weights. It then works out the number of times that a visible node and hidden node fired at the same time, which is an approximation to the expectation of the data. So if for our particular inputs the computed activations of the hidden nodes were:

Hidden Node 1	Hidden Node 2
1	0
0	1
1	1
0	1
0	1
0	0

then for each of the five courses the CDpos values will be:

Course	Hidden Node 1	Hidden Node 2
SE	1	4
ML	2	3
HCI	0	1
DB	1	1
DB	2	3

There are also similar computations for the bias nodes.

The algorithm then takes some small number of update steps sampling the visible nodes and then the hidden nodes, and makes the same estimation to get CDneg values, which are compared in order to get the weight updates, which consist of subtracting these two matrices and including any momentum term to keep the weights moving in the same direction.

After a few iterations of learning the probabilities start to be significantly different to 0.5 as the algorithm identifies the appropriate structure. For example, after 10 iterations with the dataset, the probabilities could be (to 2 decimal places):

Courses				
SE	ML	HCI	DM	DB
0.78	0.63	0.23	0.29	0.49
0.90	0.76	0.19	0.22	0.60
0.90	0.76	0.19	0.22	0.60
0.90	0.76	0.19	0.22	0.60
0.89	0.75	0.34	0.30	0.63
0.89	0.75	0.34	0.30	0.63

At this point we can look at either the probabilities or activations of the hidden nodes when the visible nodes are clamped to the inputs, or to new test data, or we can clamp the hidden nodes to particular values and look at the probabilities or activations of the visible nodes.

If we look at the hidden node activations for the training inputs, then we might well see something like:

Hidden Node 1	Hidden Node 2
0	1
0	1
1	1
1	1
1	0
1	0

This suggests that the algorithm has identified the same categories that were used to generate the data, and also seen that the two students in the middle chose a mixture of the courses.

We can also feed in a new student to the visible nodes, who took (SE, ML, DB) and the algorithm will turn on both of the hidden nodes. Further, if we turn on only the first of the two hidden nodes, and sample a few times from the visible nodes we will see outputs like $(1, 1, 1, 0, 1)$ and $(1, 1, 1, 0, 0)$, since the algorithm does not know to only choose three courses.

17.2.2 Deriving the CD Algorithm

Having seen the algorithm and a small example of using it, it is now time to look at the derivation of it. In order to understand the ideas behind this algorithm, let's recap a little about what we are doing, which is trying to compute the probability distribution $p(\mathbf{y}, \mathbf{W})$, which can be written as:

$$p(\mathbf{y}, \mathbf{W}) = \frac{1}{Z(\mathbf{W})} \exp(\mathbf{y}^T \mathbf{W} \mathbf{y}). \qquad (17.32)$$

In order to find the maximum likelihood solution to this equation, as we did previously, we would use a training set, compute the derivative of the log likelihood based on this training set, and use gradient ascent. We saw what this derivative looked like in Equation (17.15), but we will write it in a slightly different way here.

The log likelihood based on N training inputs is:

$$\begin{aligned}\mathcal{L} &= \frac{1}{N}\sum_{n=1}^{N}\log p(\mathbf{x}^n, \mathbf{W}) \\ &= \langle \log p(\mathbf{x}, \mathbf{W})\rangle_{\text{data}} \\ &= -\frac{1}{2}\langle (\mathbf{x}^n)^T\mathbf{W}\mathbf{x}^n\rangle_{\text{data}} - \log Z(\mathbf{W}) \end{aligned} \tag{17.33}$$

When we compute the derivative of this with respect to the weights the second term goes away and we get:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}} = -\langle\frac{\partial \mathcal{L}}{\partial \mathbf{W}}\rangle_{\text{data}} + \langle\frac{\partial \mathcal{L}}{\partial \mathbf{W}}\rangle_{p(\mathbf{x},\mathbf{W})} \tag{17.34}$$

Looking at it in this form it is clear what the problem is: the second term is averaged over the full probability distribution, which includes the normalisation term $Z(\mathbf{W})$, which is a sum over all of the possible values of $\mathbf{x}$, and so it is very expensive to compute.

The last insight we need is to see an alternative description of what maximising the likelihood computes. This is that it minimises what is known as the Kullback–Leibler (KL) divergence, which is a measure of the difference between two probability distributions. In fact, it is precisely the information gain that we used in Chapter 12. The KL divergence between two probability distributions p and q is defined as:

$$KL(p||q) = \int p(x)\log\frac{p(x)}{q(x)}dx \tag{17.35}$$

Note that it is asymmetric, in that $KL(p||q) \neq KL(q||p)$ in general.

To see that minimising this is equivalent to maximising the log likelihood we just need to write it out:

$$\begin{aligned}KL(p(\mathbf{x}, \mathbf{W})_{\text{data}}||p(\mathbf{x}, \mathbf{W})_{\text{model}}) &= \sum_{n=1}^{N} p_{\text{data}}\log\frac{p_{\text{data}}}{p_{\text{model}}}dx \\ &= \sum_{n=1}^{N} p_{\text{data}}\log p_{\text{data}} - \sum_{n=1}^{N} p_{\text{data}}\log p_{\text{model}} \\ &= \sum_{n=1}^{N} p_{\text{data}}\log p_{\text{data}} - \frac{1}{N}\sum_{n=1}^{N}\log p(\mathbf{x}^n|\mathbf{W}) \end{aligned} \tag{17.36}$$

Since the first term is independent of the weights it can be ignored for the optimisation, and the second term is the definition of the log likelihood.

Hinton's insight was that if, instead of minimising the KL divergence, the difference between two different KL divergences $(KL(p_{\text{data}}||p_{\text{model}}) - KL(p_{\text{MCMC}(n)}||p_{\text{model}})$ was minimised, then the expensive term in Equation 17.34 would cancel out. The term $p_{\text{MCMC}(n)}$ denotes the probability distribution after n samples of the Markov chain; often $n = 1$.

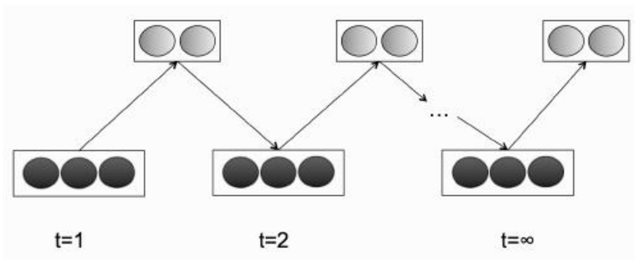

FIGURE 17.7 The form of alternating Gibbs sampling. The initial values of the visible nodes are used (with the weights) to compute the probability distribution for the hidden nodes, and then these are used to re-estimate the values of the visible nodes, and the process iterates.

That is the idea of the contrastive divergence algorithm: we compute the expectation of the data, and then use Gibbs sampling starting at the data distribution for a small number of steps to get the next term. In the terms of a wake-sleep algorithm, we don't allow the network to dream for long before we wake it up and demand to know what it was dreaming about.

In order to see what the difference is between them, consider the (non-restricted) Boltzmann machine that had only visible nodes. In that case we worked out the full computation using maximum likelihood, and the resulting weight update equation is:

$$w_{ij} \leftarrow w_{ij} + \eta(\langle v_i v_j \rangle_{\text{data}} - \langle v_i v_j \rangle_{p(\mathbf{x},\mathbf{W})} \tag{17.37}$$

Using the CD algorithm instead doesn't change things much:

$$w_{ij} \leftarrow w_{ij} + \eta(\langle v_i v_j \rangle_{\text{data}} - \langle v_i v_j \rangle_{\text{MCMC}(n)}) \tag{17.38}$$

For the restricted Boltzmann machine the CD weight update is:

$$w_{ij} \leftarrow w_{ij} + \eta(\langle v_i h_j \rangle_{p(\mathbf{h}|\mathbf{v},\mathbf{W})} - \langle v_i h_j \rangle_{\text{MCMC}(n)}) \tag{17.39}$$

The first term estimates the distribution of the hidden nodes based on the training data, while the second runs a few steps of the Markov chain to estimate the distribution. This running of the Markov chain is in the form of alternating Gibbs sampling, where the values of the visible nodes and the weights are used to estimate the hidden nodes, and then these are used to estimate the visible nodes, and the process iterates. This is shown in Figure 17.7. As with other algorithms based on gradient ascent (or descent), such as the MLP, it can be very helpful to include a momentum term in the weight update, and this is included in the description of the RBM algorithm that is given below. Following that are a few notes about the implementation of the RBM.

Looking at this algorithm you might notice that the bias weights are kept separate to the others, and have their own update rules. This is because there are two sets of bias weights: separate ones for the visible nodes and the hidden nodes. These are not symmetrical weights for the simple reason that they aren't really weights like the other connections, but just a convenient way of encoding the different activation thresholds for each neuron.

Most of the computational steps are simple to compute in NumPy. However, one particular step justifies a little bit of thought, which is the decision about whether a neuron