5.8. Perform a hierarchical clustering of the data in Figure 5.17 using the average-linkage algorithm and Euclidean distance. Show the distance matrices and the dendrogram.

5.9. Perform a hierarchical clustering of the data in Figure 5.17 using the average-linkage algorithm and city block distance. Show the distance matrices and the dendrogram.

5.10. Perform a hierarchical clustering of the data in Figure 5.17 using Ward's algorithm. Show the values of the squared errors that are computed.

5.11. Perform a partitional clustering of the data in Figure 5.17 using Forgy's algorithm. Set $k = 2$ and use the first two samples in the list as seed points. Show the values of the centroids and the nearest seed points.

5.12. Perform a partitional clustering of the data in Figure 5.17 using Forgy's algorithm. Set $k = 3$ and use the first three samples in the list as seed points. Show the values of the centroids and the nearest seed points.

5.13. Perform a partitional clustering of the data in Figure 5.17 using the k-means algorithm. Set $k = 2$ and use the first two samples in the list as the initial samples. Show the values of the centroids and the distances from the samples to the centroids.

5.14. Perform a partitional clustering of the data in Figure 5.17 using the k-means algorithm. Set $k = 3$ and use the first three samples in the list as the initial samples. Show the values of the centroids and the distances from the samples to the centroids.

5.15. Perform a partitional clustering of the data in Figure 5.17 using the isodata algorithm. Use reasonable values of your own choosing for the parameters. Set $k = 3$ and use the first three samples in the list as the initial samples.

Chapter 6

Artificial Neural Networks

6.1 Introduction

Even though high-speed computers with central processing units capable of performing many millions of operations per second have become widely available, the human brain can still perform a variety of tasks much more efficiently than computers. Tasks such as recognizing human faces and understanding speech can be performed effortlessly by the human brain, but can only be performed in controlled situations by conventional computers. In this chapter we discuss some of the attempts to model the activity of biological brains using artificial neural networks. We concentrate on the fundamental principles of these models.

The reason that the human brain can perform so efficiently is that it uses parallel computation effectively. Thousands or even millions of nerve cells called **neurons** are organized to work simultaneously on the same problem. Figure 6.1 shows the structure of a neuron which is the basic building block of the brains of all animals. A neuron consists of a **cell body**, **dendrites** which receive input signals, and **axons** which send output signals. Dendrites receive input from sensory organs such as the eyes or ears and from axons of other neurons. Axons send output to organs such as muscles and to dendrites of other neurons.

Tens of billions of neurons are present in the human brain. A neuron typically receives input from several thousand dendrites and sends output through several hundred axonal branches. Because of the large number of connections between neurons and the redundant connections between them, the performance of the brain is relatively robust and is not significantly affected by the many neurons that die every day. Even when large numbers of neurons are damaged through trauma or stroke, remarkable rehabilitation is often possible by adaptation of the remaining neurons.

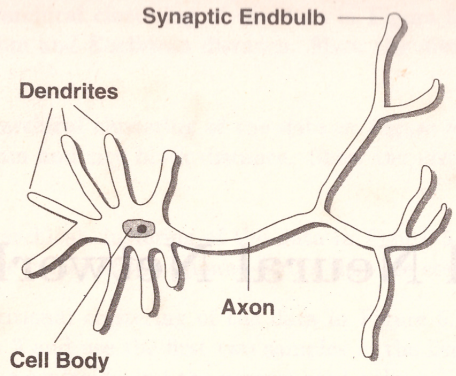

Figure 6.1: A diagram of a biological neuron.

On the other hand, hundreds of brain regions have been identified in which the neurons only perform specialized functions. For example, the Japanese physician Tatsuji Inouye, who treated soldiers during the Russo-Japanese War of 1904–1905, collected data concerning injuries to the visual cortex, which is located in the back of the brain [Glickstein]. These observations were possible in part because of the small bullets that were used in that war. The bullets pierced the skull without shattering it. Soldiers with injuries to the visual cortex suffered permanent partial loss of vision, but often recovered completely otherwise. By comparing the physical location of the injury with the patient's description of his visual impairment, progress was made in understanding the structure of the visual cortex. Other human observations and animal experimentation have produced extensive neurophysiological information. However, much more is still unknown about the human brain than is known.

The neurons in many regions of biological brains are organized into layers. A neuron in a layer usually receives its inputs from neurons in an adjacent layer. Furthermore, those inputs are usually from neurons in a relatively small region and the interconnection pattern is similar for each location. Connections between layers are mostly in one direction, moving from low-level processing to higher coordination and reasoning. The low-level processing may even be done outside the brain, for example in the eyes or ears.

An early attempt to form an abstract mathematical model of a neuron was by McCulloch and Pitts [McCulloch] in 1943 (see Figure 6.2a). Their model

- receives a finite number of inputs $x_1, \ldots, x_M$
- computes the weighted sum $s = \sum_{i=1}^{M} w_i x_i$ using the weights $w_1, \ldots, w_M$

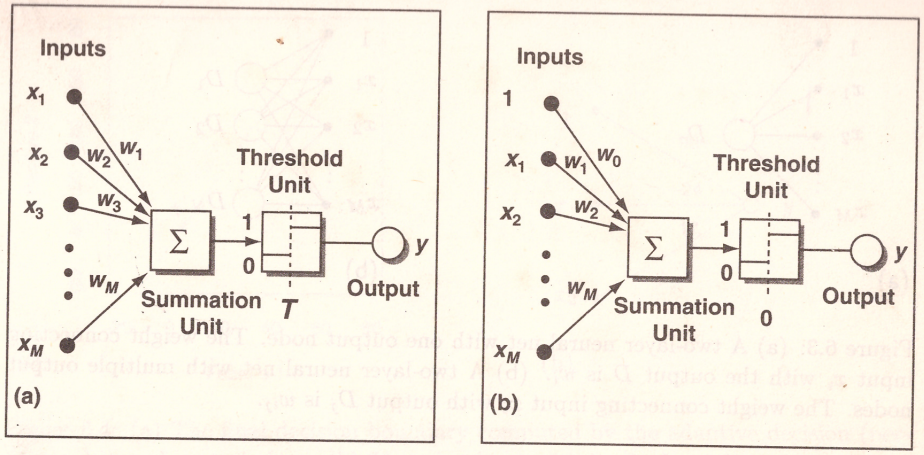

Figure 6.2: (a) The McCulloch-Pitts model of the neuron. (b) Model of a neuron with a bias weight.

- thresholds s and outputs 0 or 1 depending on whether the weighted sum is less than or greater than a given threshold value T.

Summation is indicated by a sigma and thresholding is shown by a small graph of the function. The combination of summation and thresholding is called a **node** in an artificial neural network. Nodes are denoted by open circles in the diagrams of neural nets later in this chapter. Node inputs with positive weights are called **excitory** and node inputs with negative weights are called **inhibitory**.

The action of the model neuron is output 1 if

$$w_1 x_1 + w_2 x_2 + \ldots + w_M x_M > T \tag{6.1}$$

and 0 otherwise. For convenience in describing the learning algorithms presented later in this chapter, we rewrite (6.1) as the sum

$$D = w_0 x_0 + w_1 x_1 + \ldots + w_M x_M \tag{6.2}$$

where $w_0 = -T$ and $x_0 = 1$. Output 1 if $D > 0$ and output 0 if $D \leq 0$. The weight w_0 in (6.2) is called the **bias weight**. Figure 6.2a may be modified to use a bias weight instead of explicitly using a threshold. This new model is shown in Figure 6.2b.

To produce electronic neural nets that imitate biological brains, the neurons of the network must be manufactured in hardware and connected as parallel computers

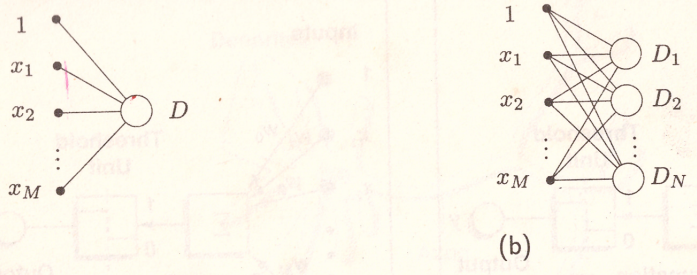

Figure 6.3: (a) A two-layer neural net with one output node. The weight connecting input x_i with the output D is w_i. (b) A two-layer neural net with multiple output nodes. The weight connecting input x_i with output D_j is w_{ij}.

such that the neurons perform their computations simultaneously. Although biological neurons are thousands of times slower than electronic neurons, biological brains are able to outperform electronic neural nets in many cases by using billions of neurons working in parallel using algorithms and network structures that vary significantly from lower to higher animals.

A difficult aspect of using a neural net is **training**, that is, finding weights that solve problems with acceptable performance. In Section 6.4, we discuss the back-propagation algorithm for training weights, and in Section 6.5, we discuss Hopfield's network. Once the weights have been correctly trained, using the net to classify samples is straightforward.

Because building neural nets in hardware is difficult and expensive, most of the neural net research is performed on nets simulated on conventional single-processor computers. In some cases the net itself is eventually built in hardware, but the design and training of the weights of the net and the evaluation of its performance are done in advance on a conventional computer.

6.2 Nets without Hidden Layers

The neural nets in Figure 6.3 are called **two-layer nets** because they have a layer of input nodes that are directly connected to the layer of output nodes. By contrast, the nets in Section 6.3 have at least one layer of nodes between the input and output layers. The layers of nodes between the input and output layers, which are not seen by the outside world, are called **hidden layers**.

In the nets in Figure 6.3, a weighted sum of the inputs is calculated by each output

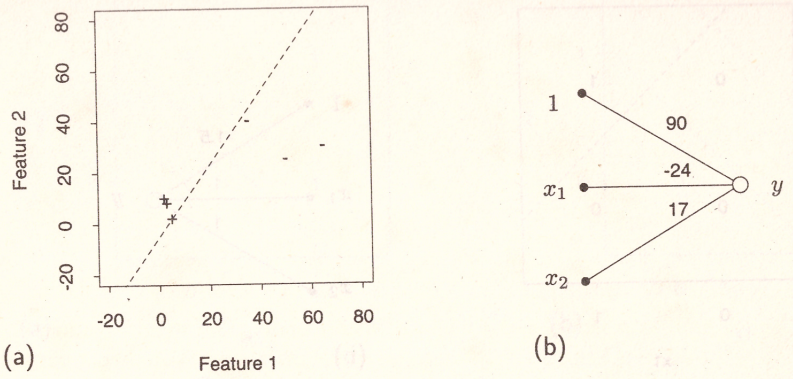

Figure 6.4: (a) The final decision boundary computed by the adaptive decision (perceptron) boundary technique. (b) The neural net with its final weights.

node. The output nodes then compare the weighted sum to a threshold. [Rosenblatt] used the McCulloch-Pitts neuron of Figure 6.2b in 1953 to form a trainable classifier that he called the **perceptron**. The algorithm used to adapt the weights for the perceptron is identical to the adaptive discriminant function technique discussed in Section 4.5. The final discriminant function obtained for Example 4.9 was

$$D = w_0 + w_1 x_1 + x_2 x_2 = 90 - 24x_1 + 17x_2. \tag{6.3}$$

If a sample produces a positive value for D, the sample is classified as belonging to class 1; otherwise, the sample is classified as belonging to class 0. For example, the sample $x_1 = 2, x_2 = 3$ gives the value $D = 93$, so this sample is classified into class 1.

The weights, 90, -24, 17, obtained for the neural net using the perceptron algorithm are the same as the coefficients of the discriminant function. In fact, the neural net in Figure 6.4b can be considered to be an alternative way of representing the algebraic equation (6.3). When $x_1 = 2, x_2 = 3$ are input to the net, its output is 93 before thresholding and 1 after thresholding.

A wide variety of patterns can be recognized using a two-layer net. For example, consider the problem of calculating the logical AND of two inputs (see Figure 6.5a):

x_1	x_2	Output
0	0	0
0	1	0
1	0	0
1	1	1

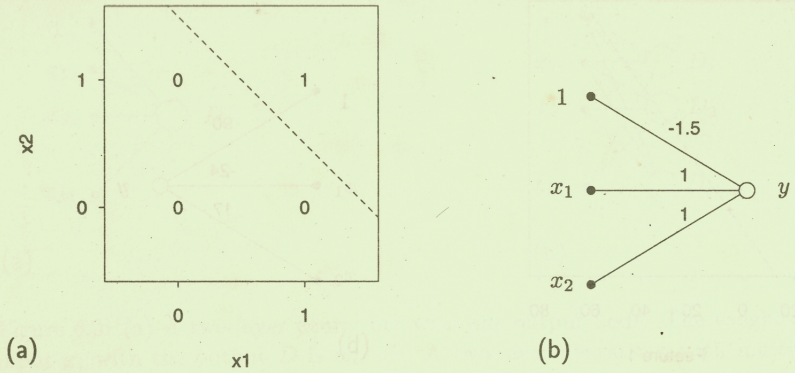

Figure 6.5: (a) The separating line $-1.5 + x_1 + x_2 = 0$ for the logical AND pattern. (b) A two-layer net implementing the logical AND.

In logical functions, 1 usually represents "true" or the presence of a binary feature, while 0 represents "false" or the absence of a binary feature. If x_1 and x_2 are binary features, $x_1 = 1$ and $x_2 = 1$ mean that both features are present, so x_1 AND $x_2 = 1$. However, if one of the features is absent (x_1, x_2, or both are 0), then x_1 AND $x_2 = 0$. One of the possible lines that separates the point $(1, 1)$ from the points $(0, 0)$, $(1, 0)$, and $(0, 1)$ has the equation $-1.5 + x_1 + x_2 = 0$ (see Figure 6.5b). Thus a set of weights for the two-layer net that produces the logical AND of its inputs is $w_0 = -1.5$, $w_1 = 1$, and $w_2 = 1$.

Figure 6.6a illustrates a similar situation for the logical OR function

x_1	x_2	Output
0	0	0
0	1	1
1	0	1
1	1	1

which can be produced by $D = -0.5 + x_1 + x_2$.

The exclusive OR function is 1 if exactly one of the features x_1 or x_2 is present, but 0 if none or both of the features is present:

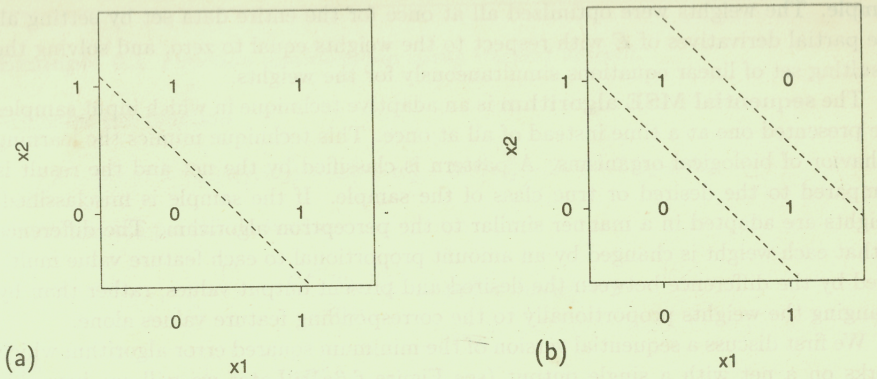

Figure 6.6: (a) The separating line $-0.5 + x_1 + x_2 = 0$ for the logical OR pattern. (b) No single separating line exists for the exclusive OR pattern.

x_1	x_2	Output
0	0	0
0	1	1
1	0	1
1	1	0

A separating line does not exist for the exclusive OR function (see Figure 6.6b). The pattern is classified as 0 only if it is between the two lines, so a two-layer net is not adequate for classification in this case because the middle decision region is not a half-plane. However, as we show in Section 6.3, the outputs of two linear decision boundaries can be combined in a three-layer net to produce the desired result.

The Sequential MSE Algorithm

The goal of the minimum squared error procedure of Section 4.7 was to find the values of the weights $w_0, \ldots, w_M$ that minimize the criterion function

$$E = (1/2)\sum_{p=1}^{N}(D_p - d_p)^2, \tag{6.4}$$

where

$$D_p = w_0 + w_1 x_{p1} + \ldots + w_M x_{pM},$$

d_p is the desired output for sample p, and $x_{p1}, \ldots, x_{pM}$ are the feature values of that sample. The weights were optimized all at once for the entire data set by setting all the partial derivatives of E with respect to the weights equal to zero, and solving the resulting set of linear equations simultaneously for the weights.

The **sequential MSE algorithm** is an adaptive technique in which input samples are presented one at a time instead of all at once. This technique mimics the learning behavior of biological organisms. A pattern is classified by the net and the result is compared to the desired or true class of the sample. If the sample is misclassified, weights are adapted in a manner similar to the perceptron algorithm. The difference is that each weight is changed by an amount proportional to each feature value multiplied by the difference between the desired and present output values, rather than by changing the weights proportionally to the corresponding feature values alone.

We first discuss a sequential version of the minimum squared error algorithm which works on a net with a single output (see Figure 6.3a). Later we will see how this algorithm can be modified to work with a net with multiple outputs.

The sequential MSE algorithm uses the **steepest descent minimization procedure** for adapting the weights for each sample. The weights are changed in directions that will decrease the error between the desired output of the network d and its actual output D. To find a local minimum of a function $F(w_1, \ldots, w_M)$ of M variables, pick a starting guess $w_1, \ldots, w_M$ and move a short distance in the direction of the steepest decrease of the function. From this new point, recompute the direction of steepest decrease, and move a short distance in this new direction. Continue until you are close to a local minimum. If everything works well, this local minimum will be the global minimum that is sought. However, it is possible to be trapped in a local minimum and never reach the absolute minimum. Avoiding local minima can be difficult, especially if the number of variables is large. It can be shown (see Problem 6.8) that the direction of steepest descent of a differentiable function F of M variables is the vector

$$\left(-c\frac{\partial F}{\partial w_1}, \ldots, -c\frac{\partial F}{\partial w_M}\right)$$

where c is a positive constant of proportionality. The steepest descent algorithm can be stated formally as follows:

1. Pick a starting guess $w_1, \ldots, w_M$ and choose a positive constant c.
2. Compute the partial derivatives $\partial F/\partial w_i$, for $i = 1, \ldots, M$. Replace w_i by $w_i - c\partial F/\partial w_i$ for $i = 1, \ldots, M$.
3. Repeat step 2 until $w_1, \ldots, w_M$ cease to change significantly.

If the partial derivatives are not available analytically, they may be estimated by finding the increase in E produced by increasing w_i a very small amount, that is, $\partial E/\partial w_i \approx \Delta E/\Delta w_i$.

Example 6.1 *Function minimization using steepest descent.*

Minimize the function

$$F(w_1, w_2) = w_1^4 + w_2^4 + 3w_1^2w_2^2 - 2w_1 - 3w_2 + 4,$$

which has the partial derivatives

$$\frac{\partial F(w_1, w_2)}{\partial w_1} = 4w_1^3 + 6w_1w_2^2 - 2$$

and

$$\frac{\partial F(w_1, w_2)}{\partial w_2} = 4w_2^3 + 6w_1^2w_2 - 3.$$

If we choose $c = 0.1$, we obtain the following sequence of iterations:

Iteration	$F(w_1, w_2)$	$\partial F(w_1, w_2)/\partial w_1$	$\partial F(w_1, w_2)/\partial w_2$	w_1	w_2
1	4.000	−2.0000	−3.000	0.000	0.000
2	2.730	−1.8600	−2.820	0.200	0.300
3	1.770	0.9855	−1.691	0.386	0.582
4	1.548	0.9523	−2.468	0.485	0.751
5	1.543	0.1441	−0.082	0.475	0.776
10	1.539	0.0221	−0.0161	0.438	0.802
20	1.539	0.0005	0.0004	0.430	0.807
30	1.539	0.0000	0.0000	0.430	0.807
40	1.539	0.0000	0.0000	0.430	0.807

After 30 iterations, w_1 and w_2 cease to change significantly, so a local minimum has been reached.

Two difficulties with using the steepest descent procedure to minimize an arbitrary nonlinear function of $w_1, \ldots, w_M$ are choosing a good starting guess and choosing an appropriate constant c. If a starting guess too far from the values that produce the absolute global minimum is chosen, the procedure may converge to a local minimum that is not the desired solution. If the constant c is chosen too small, progress toward the minimum will be so slow that a huge number of iterations will be required to obtain the solution. If c is too large, the procedure may repeatedly overshoot the minimum and progress will also be slow. With c too large, the procedure may even cycle between two or more local minima and fail to converge. Finding a good starting guess and constant usually requires experimentation with the particular problem being solved.

Decreasing c as the iteration number increases can be useful as it is with adaptive discriminant functions.

We can now describe the sequential MSE algorithm for a single-output, two-layer net. Each of the N input samples has M features $x_1, \ldots, x_M$. The weights are $w_1, \ldots, w_M$. As in Section 6.1, it is convenient to define the auxiliary feature $x_0 = 1$ and bias weight w_0. Because we are updating the weights by considering one sample at a time, we use the error criterion function

$$E = \frac{1}{2}(D - d)^2,$$

where d is the correct output or class of the current sample and

$$D = \sum_{i=0}^{M} w_i x_i$$

is the output of the net. If the sample is misclassified, we want to adapt the weights so that D will be closer to d. The partial derivatives are

$$\frac{\partial E}{\partial w_i} = (D - d)x_i.$$

Using the steepest descent algorithm to minimize E gives us the sequential MSE algorithm.

1. Pick starting weights $w_0, \ldots, w_M$ and choose a positive constant c.
2. Present samples 1 through N repeatedly to the classifier, cycling back to sample 1 after sample N is encountered. For each sample, compute
$$D = w_0 + w_1 x_1 + \ldots + w_M x_M.$$
3. Replace w_i by $w_i - c(D - d)x_i$ for all i.
4. Repeat steps 2 and 3 until the weights cease to change significantly.

If the samples are linearly separable, the adaptive discriminant algorithm will eventually produce weights that will perfectly classify all samples. By contrast, the sequential MSE algorithm makes no such promise. However, if the samples are linearly separable and the distance between samples within each class is small with respect to the distance between the classes, the sequential MSE algorithm usually works well.

With minor modifications, the sequential MSE algorithm will also work for a two-layer net with multiple outputs (see Figure 6.3b). In this case, we have for each of the N samples the feature values $x_0, x_1, \ldots, x_M$ and desired outputs $d_1, \ldots, d_{M_1}$. The value w_{ij} is the weight on input i for output node j. We use the error criterion function

$$E = \frac{1}{2}\sum_{j=1}^{M_1}(D_j - d_j)^2.$$

Since $D_j = \sum_{i=0}^{M_1} w_{ij}x_i$,

$$\frac{\partial E}{\partial w_{ij}} = (D_j - d_j)x_i.$$

The Sequential MSE Algorithm for Multiple Outputs

1. Pick starting weights $w_0, \ldots, w_M$ and choose a positive constant c.
2. Present samples 1 through N repeatedly to the classifier, cycling back to sample 1 after sample N is encountered. For each sample, compute
$$D_j = w_{0j} + w_{1j}x_1 + \ldots + w_{Mj}x_M$$
for nodes $j = 1, \ldots, M_1$.
3. Replace w_{ij} by $w_{ij} - c(D_j - d_j)x_i$ for all i.
4. Repeat steps 2 and 3 until the weights cease to change significantly.

The algorithm usually works well if the classes are well separated.

6.3 Nets with Hidden Layers

After Rosenblatt developed the perceptron classifier in 1953 [Rosenblatt], some researchers had high expectations that it would be capable of modeling many simple brain functions. As research into the capabilities of the perceptron continued into the 1960s, it became apparent that the perceptron was too limited to use on any but the simplest learning situations. In 1969, the book *Perceptrons* by Minsky and Pappert ([Minsky]) was pessimistic regarding the capabilities of two-layer nets. They stated that multilayer nets were so general that they were equivalent to a universal computer and hence too complex for general analysis. However, other researchers continued to develop many types of multilevel adaptive networks. Some of these replaced the zero-one threshold function by an S-shaped curve, so that each weight had some effect on the final output. One such sigmoidal function is

$$R(s) = \frac{1}{1 + e^{-s}}$$

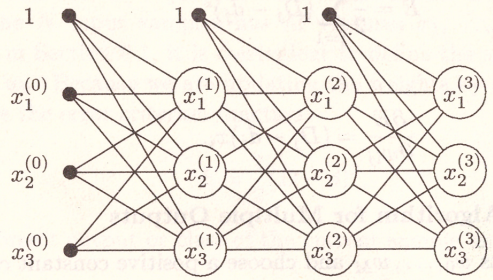

Figure 6.7: A four-layer neural net. The weight connecting node $x_i^{(k-1)}$ with node $x_j^{(k)}$ is $w_{ij}^{(k)}$.

discussed in Section 6.4. Much research is being conducted today on algorithms that will improve the speed and reliability of training multilayer nets.

In general, a multilayer net has $K+1$ layers of nodes, denoted $0, 1, \ldots, K$ as shown in Figure 6.7. The output of the ith node in layer k is denoted $x_i^{(k)}$. This is the value obtained after computing the weighted sum of the inputs and applying the threshold or other function. The layer of input nodes has index $k = 0$ and is called the **retina**. The nodes in layer K are called the output nodes, and those in layers 1 through $K-1$ are called the hidden nodes.

As mentioned in Section 6.2, a two-layer net can classify samples into two classes which are separated by a hyperplane. However, if the problem is to classify samples into two decision regions where class 1 is convex, and class 0 is the complement of class 1, three layers are required. (A region is **convex** if, whenever two points are in the region, the entire line segment joining the two points is also in the region.) Because a convex set can be approximated by the intersection of a finite number of half-planes, the nodes in layer 1 determine whether the sample lies in each of the half-planes making up the convex region, and layer 2 performs a logical AND to see if the pattern is in all of those half-planes simultaneously. Figure 6.6b shows how the exclusive OR mapping can be implemented as the intersection of the two hyperplanes

$$-0.5 + x_1 + x_2 \geq 0 \quad \text{AND} \quad 1.5 - x_1 - x_2 \geq 0. \tag{6.5}$$

The corresponding neural net is shown in Figure 6.8.

In principle, a three-layer network, called an **and-or network**, can always perfectly separate two classes if they can be distinguished by a set of binary features, no matter

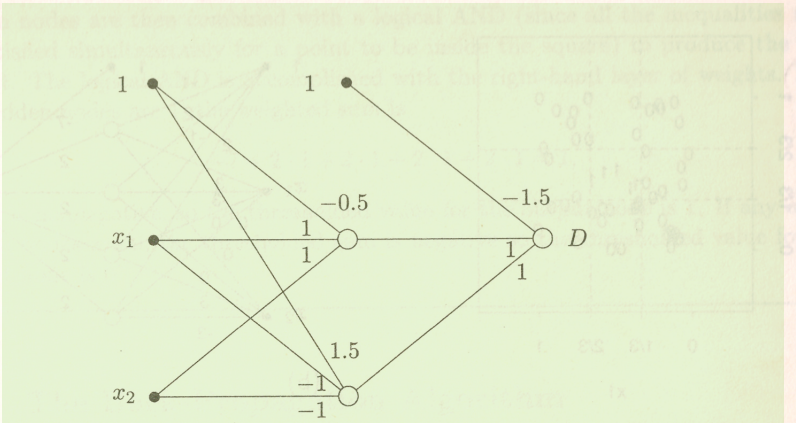

Figure 6.8: The implementation of the exclusive OR mapping.

how many there are. Even if the features can take on a finite number of discrete values, these values can be encoded as a large number of binary features. By computing the appropriate AND function, each node in the second layer specializes in detecting one of the possible input combinations that requires an output of 1. There are as many nodes in the second layer as there are input combinations that require an output of 1. The single node in the third layer computes the logical OR of the outputs from the second layer, so that if any second layer node outputs a 1, the third layer node will output a 1.

A problem with this approach is that the number of neurons required in layer 1 can be about as large as the number of different feature combinations that can occur or v^f, where f is the number of features, each with v possible values. There may be too many input combinations that require an output of 1 for it to be practical to build a node for each one of them. In many cases, the total number of nodes required in the network can be reduced by allowing the network to contain more than three layers.

To determine whether or not a pattern with continuous features belongs to a class that is not convex, more than three layers are required. A finite set of any shape is contained in the union of a finite number of convex sets or "pieces" of the region. This implies that at most four layers are required: Layer 0 supplies the inputs, and layer 1 determines whether the sample is in each of the half-planes making up the convex components. The nodes in layer 3 compute the logical ANDs of the groups of half-planes, which define each of the convex pieces of the nonconvex regions, and a single node in layer 4 computes the logical OR to determine if the pattern lies in at least one of the convex components. See Problem 6.4 for an example of determining if a pattern

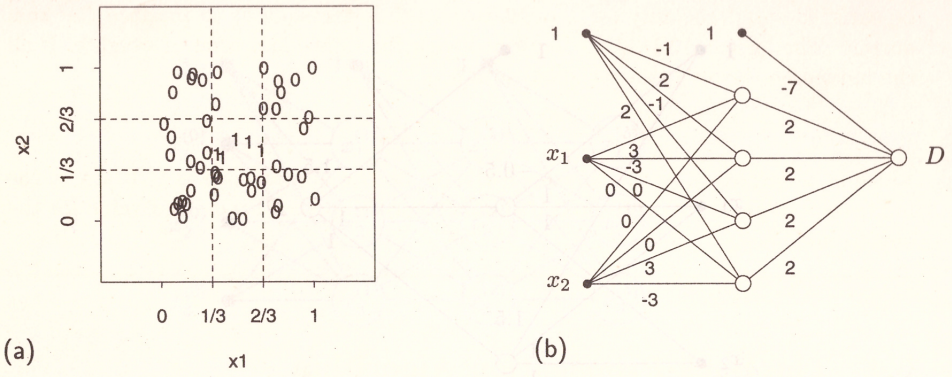

Figure 6.9: (a) Classification into a convex region. (b) Neural net for classification into the convex region in (a).

belongs to a nonconvex set.

Example 6.2 *A neural net that recognizes patterns in a convex region.*

Figure 6.9a shows samples from two classes: Class 1 lies inside the square decision region with vertices $(1/3, 1/3)$, $(2/3, 1/3)$, $(1/3, 2/3)$, and $(2/3, 2/3)$, and class 0 lies outside this square. One possible neural net that will perform this classification is shown in Figure 6.9b.

A point (x_1, x_2) is inside the square precisely when

$$\frac{1}{3} < x_1, \quad \frac{2}{3} > x_1, \quad \frac{1}{3} < x_2, \quad \text{and} \quad \frac{2}{3} > x_2.$$

These inequalities may be rewritten as

$$\begin{aligned} 0 &< -1 + 3x_1 + 0x_2 \\ 0 &< +2 - 3x_1 + 0x_2 \\ 0 &< -1 + 0x_1 + 3x_2 \\ 0 &< +2 + 0x_1 - 3x_2. \end{aligned}$$

The hidden nodes, from top to bottom, compute the right-hand sides of these inequalities. For example, the weights -1, 3, 0 of the inputs to the top hidden node are the coefficients of the right-hand side of the first inequality. The outputs from the hidden nodes are then combined with a logical AND (since all the inequalities must be satisfied simultaneously for a point to be inside the square) to produce the final output. The logical AND is accomplished with the right-hand layer of weights. If all the hidden nodes are 1, the weighted sum is

$$-7 + 2 \cdot 1 + 2 \cdot 1 + 2 \cdot 1 + 2 \cdot 1 = 1,$$

which is nonnegative, so the thresholded value for the output node is 1. If any of the hidden nodes are 0, the thresholded sum is negative so the thresholded value for the output node is 0.

6.4 The Back-Propagation Algorithm

Training a multilayer linear threshold network is complicated by the fact that a small change in one of the weights will usually not affect the outputs of the network at all, so steepest descent methods are not feasible. The output from a node will be affected only if one of the weights changes enough to cause its weighted sum to change its sign. Even if an output changes at one layer, the outputs at the next layer may not necessarily change, so the final outputs are very resistant to most small weight changes.

We might consider eliminating the threshold elements and simply compute weighted sums at each node. However, in this case, there would be no need to use a multilayer net, since it can be shown that any net without thresholds is equivalent to a two-layer net without thresholds. The following example illustrates this fact.

Example 6.3 *A neural net and an equivalent two-layer neural net.*

Consider the neural network in Figure 6.10a. We assume that each node outputs its value without thresholding. Thus the output of node N_1 is $2 - 3x_1 + 2x_2$, the output of node N_2 is $-1 + x_1 - 2x_2$, and the output of network is

$$D = 2 + 4(2 - 3x_1 + 2x_2) - 2(-1 + x_1 - 2x_2). \tag{6.6}$$

After simplifying, this last equation may be rewritten as

$$D = 12 - 14x_1 + 12x_2. \tag{6.7}$$

From this last equation, we see that the three-layer neural network of Figure 6.10a is equivalent to the two-layer neural network of Figure 6.10b in the sense that both networks produce the same output given the same input.

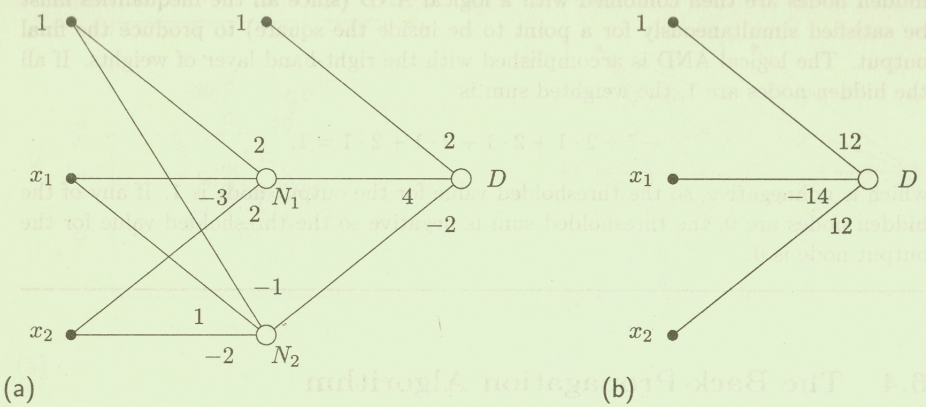

Figure 6.10: (a) A three-layer neural network without thresholds. (b) A two-layer network without thresholds which is equivalent to the neural network in (a).

This example demonstrates the fact that any network without thresholds is equivalent to a two-layer network without thresholds. This is true in general because combining linear expressions, such as in (6.6), results in a linear expression, such as (6.7), which can be computed by a two-layer network without thresholding.

A compromise between the two extremes of using a discontinuous threshold at each node and completely linear nodes is to use a smooth S-shaped function that gradually changes from 0 when the weighted sum s is much less than the threshold to 1 when s is much greater than the threshold. One such function is a **sigmoid function**, popularized by [Rummelhart]:

$$R(s) = \frac{1}{1+e^{-s}}.$$

It varies from 0 when s approaches $-\infty$, to 1 when s approaches $+\infty$. When $s = 0$, $R(s) = 1/2$. This function also has the convenient property that

$$\frac{dR}{ds} = R(1-R)$$

(see Problem 6.10), which is used in the back-propagation algorithm.

The use of the sigmoid threshold function R at each node in a net causes the outputs of the net to be differentiable functions of the weights, so the steepest descent method can be used to find a locally optimal set of weights. The outputs of such a net can be very complicated nonlinear functions of the inputs, so the weights to which the net converges may be dependent on the starting guess and step size.

The **back-propagation algorithm** for training the weights in a multilayer net uses the steepest descent minimization procedure (see Section 6.2) and the sigmoid threshold function. We assume that the layers are $k = 0, \ldots, K$, with $k = 0$ denoting the input layer, and $k = K$ denoting the output layer (see Figure 6.7). The output of node j in layer k is denoted $x_j^{(k)}$ for $j = 1, \ldots, M_k$, where M_k is the number of nodes in layer k (not counting the node with a bias weight). In particular, in the input layer node j passes its input x_j along as its output: $x_j^{(0)} = x_j$ for $j = 1, \ldots, M_0$. In every layer, except the output layer, the bias node outputs 1, so $x_0^{(k)} = 1$ for $k = 0, \ldots, K-1$. The outputs are $x_j^{(K)}$ for $j = 1, \ldots, M_K$. (The output layer does not have a node of index 0.) The weight of the connection from node i in layer $k-1$ to node j in layer k is denoted $w_{ij}^{(k)}$.

The back-propagation algorithm consists of two main steps: a feed-forward step in which the outputs of the nodes are computed starting at layer 1 and working forward to the output layer K, and a back-propagation step where the weights are updated in an attempt to get better agreement between the observed outputs $x_1^{(K)}, \ldots, x_{M_K}^{(K)}$ and the desired outputs $d_1, \ldots, d_{M_K}$. The feed-forward step begins at layer 1 and works forward to layer K. The back-propagation step begins at layer K and works backward to layer 1.

The Back-Propagation Algorithm

1. Initialize the weights $w_{ij}^{(k)}$ to small random values, and choose a positive constant c.

2. Repeatedly set $x_1^{(0)}, \ldots, x_{M_0}^{(0)}$ equal to the features of samples 1 to N, cycling back to sample 1 after sample N is reached.

3. *Feed-forward step.* For $k = 0, \ldots, K-1$, compute

$$x_j^{(k+1)} = R\left(\sum_{i=0}^{M_k} w_{ij}^{(k+1)} x_i^{(k)}\right), \tag{6.8}$$

for nodes $j = 1, \ldots, M_{k+1}$. We use the sigmoid threshold function $R(s) = 1/(1+e^{-s})$.

4. *Back-propagation step.* For the nodes in the output layer, $j = 1, \ldots, M_K$, compute

$$\delta_j^{(K)} = x_j^{(K)}(1 - x_j^{(K)})(x_j^{(K)} - d_j). \tag{6.9}$$

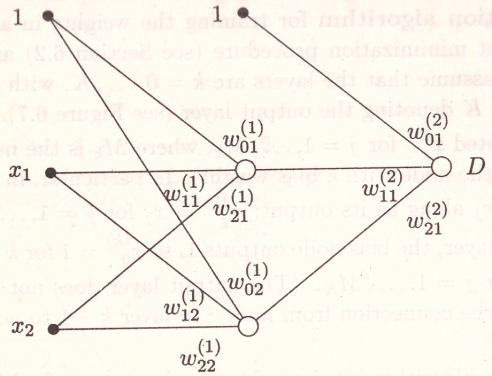

Figure 6.11: The neural net for derivation of back-propagation weights in Example 6.4.

For layers $k = K-1, \ldots, 1$ compute

$$\delta_i^{(k)} = x_i^{(k)}(1 - x_i^{(k)}) \sum_{j=1}^{M_{k+1}} \delta_j^{(k+1)} w_{ij}^{(k+1)} \tag{6.10}$$

for $i = 1, \ldots, M_k$.

5. Replace $w_{ij}^{(k)}$ by $w_{ij}^{(k)} - c\delta_j^{(k)} x_i^{(k-1)}$ for all i, j, k.

6. Repeat steps 2 to 5 until the weights $w_{ij}^{(k)}$ cease to change significantly.

Note that if a node output value is close to 0 or 1, the value $\delta_j^{(k)}$ is close to 0, so the changes in the input weights of that node will be small. This contributes to the stability of the net because if a node output is close to its final value of 0 or 1, its weights will not tend to change very much, but the weights in undecided nodes with outputs close to 0.5 can easily move in either direction.

In the feed-forward step, we compute the output of each node in layer 1, then the output of each node in layer 2, and so on. The value of $x_j^{(k+1)}$ is the output of the jth node in the $(k+1)$st layer. The back-propagation step uses the steepest descent method to change the weights so that the computed output becomes closer to the desired output. More precisely, it uses the steepest descent method to minimize the error function

$$E = \frac{1}{2} \sum_{j=1}^{M_K} (x_j^{(K)} - d_j)^2,$$

where d_j is the desired output of the jth node in the last layer. The partial derivatives of E are first computed with respect to the weights in the last layer, then in the next-to-last layer, and so on. Since the partial derivatives of E in layer k involve the partial derivatives of E in layer $k+1$, when the partial derivatives of E in layer k are computed, all of the terms involved in the expression will already have been computed.

We show how (6.9) results from using the steepest descent method to minimize the error function; we leave the similar derivation of (6.10) as an exercise (see Problem 6.12). To derive (6.9), we compute

$$\frac{\partial E}{\partial w_{ij}^{(K)}} = (x_j^{(K)} - d_j) \frac{\partial x_j^{(K)}}{\partial w_{ij}^{(K)}} = (x_j^{(K)} - d_j) x_j^{(K)} (1 - x_j^{(K)}) x_i^{(K-1)}.$$

We have used the facts that

$$\frac{\partial R(s)}{\partial s} = R(s)(1 - R(s))$$

(see Problem 6.10) and

$$\frac{\partial x_j^{(K)}}{\partial w_{ij}^{(K)}} = \frac{\partial}{\partial w_{ij}^{(K)}} R\left(\sum_{i=0}^{M_{K-1}} w_{ij}^{(K)} x_i^{(K-1)} \right) = x_j^{(K)} (1 - x_j^{(K)}) x_i^{(K-1)}.$$

Therefore $w_{ij}^{(K)}$ should be replaced by

$$w_{ij}^{(K)} - c \frac{\partial E}{\partial w_{ij}^{(K)}} = w_{ij}^{(K)} - c(x_j^{(K)} - d_j) x_j^{(K)} (1 - x_j^{(K)}) x_i^{(K-1)},$$

which becomes

$$w_{ij}^{(K)} - c\delta_j^{(K)} x_i^{(K-1)}$$

if we let

$$\delta_j^{(K)} = (x_j^{(K)} - d_j) x_j^{(K)} (1 - x_j^{(K)}) = x_j^{(K)} (1 - x_j^{(K)}) (x_j^{(K)} - d_j).$$

Example 6.4 *The back-propagation algorithm for the neural net of Figure 6.11.*

We show how the back-propagation algorithm works for the neural net of Figure 6.11. We assume that the positive constant c of step 1 is equal to 1.

Since this net has only one output, we denote the output $x_1^{(2)}$ as D and the desired output as d. At the feed-forward step (step 3), we calculate the output of each node. The output of node 1 in layer 1 is

$$x_1^{(1)} = R(w_{01}^{(1)} + w_{11}^{(1)}x_1^{(0)} + w_{21}^{(1)}x_2^{(0)}),$$

where R denotes the sigmoid function. Similarly, we obtain

$$x_2^{(1)} = R(w_{02}^{(1)} + w_{12}^{(1)}x_1^{(0)} + w_{22}^{(1)}x_2^{(0)})$$

and

$$\begin{aligned} x_1^{(2)} &= R(w_{01}^{(2)} + w_{11}^{(2)}x_1^{(1)} + w_{21}^{(2)}x_2^{(1)}) \\ &= R(w_{01}^{(2)} + w_{11}^{(2)}R(w_{01}^{(1)} + w_{11}^{(1)}x_1^{(0)} + w_{21}^{(1)}x_2^{(0)}) \\ &\quad + w_{21}^{(2)}R(w_{02}^{(1)} + w_{12}^{(1)}x_1^{(0)} + w_{22}^{(1)}x_2^{(0)})). \end{aligned}$$

For step 4, we first compute

$$\delta_1^{(2)} = x_1^{(2)}(1 - x_1^{(2)})(x_1^{(2)} - d).$$

Next for step 4, we compute

$$\begin{aligned} \delta_1^{(1)} &= x_1^{(1)}(1 - x_1^{(1)})\delta_1^{(2)}w_{11}^{(2)}, \\ \delta_2^{(1)} &= x_2^{(1)}(1 - x_2^{(1)})\delta_1^{(2)}w_{21}^{(2)}. \end{aligned}$$

These are the formulas that are used to update the weights for step 5. For example, weight $w_{11}^{(2)}$ will be updated as

$$w_{11}^{(2)} - \delta_1^{(2)}x_1^{(1)} = w_{11}^{(2)} - D(1 - D)(D - d)R(w_{01}^{(1)} + w_{11}^{(1)}x_1^{(0)} + w_{21}^{(1)}x_2^{(0)}). \tag{6.11}$$

The formula (6.11) in the preceding discussion was obtained using formula (6.9). Instead of deriving (6.9) directly, we present the simpler derivation of (6.11). Both derivations are obtained by using the steepest descent method to minimize the error function

$$E = \frac{1}{2}(D - d)^2.$$

with respect to the weights.

According to the steepest descent algorithm, weight $w_{11}^{(2)}$ should be updated to

$$w_{11}^{(2)} - \frac{\partial E}{\partial w_{11}^{(2)}}. \tag{6.12}$$

Now,

$$\begin{aligned} \frac{\partial E}{\partial w_{11}^{(2)}} &= (D - d)\frac{\partial D}{\partial w_{11}^{(2)}} \\ &= (D - d)D(1 - D)\frac{\partial}{\partial w_{11}^{(2)}}(w_{01}^{(2)} + w_{11}^{(2)}x_1^{(1)} + w_{21}^{(2)}x_2^{(1)}) \\ &= (D - d)D(1 - D)x_1^{(1)} \\ &= (D - d)D(1 - D)R(w_{01}^{(1)} + w_{11}^{(1)}x_1^{(0)} + w_{21}^{(1)}x_2^{(0)}). \end{aligned} \tag{6.13}$$

We see that substituting (6.13) into (6.12) gives (6.11). Similarly,

$$\begin{aligned} \frac{\partial E}{\partial w_{01}^{(2)}} &= (D - d)D(1 - D) \\ \frac{\partial E}{\partial w_{21}^{(2)}} &= (D - d)D(1 - D)R(w_{02}^{(1)} + w_{12}^{(1)}x_1^{(0)} + w_{22}^{(1)}x_2^{(0)}), \end{aligned}$$

which agree with step 5 of the back-propagation algorithm.

Next we show, as another example, how the updating formula for the weight $w_{21}^{(1)}$, which feeds into the hidden layer, can be derived. According to the back-propagation algorithm, $w_{21}^{(1)}$ is updated as

$$\begin{aligned} w_{21}^{(1)} - \delta_1^{(1)}x_2^{(0)} &= w_{21}^{(1)} - x_1^{(1)}(1 - x_1^{(1)})\delta_1^{(2)}w_{11}^{(2)}x_2^{(0)} \\ &= w_{21}^{(1)} - x_1^{(1)}(1 - x_1^{(1)})x_1^{(2)}(1 - x_1^{(2)})(x_1^{(2)} - d)w_{11}^{(2)}x_2^{(0)} \\ &= w_{21}^{(1)} - x_1^{(1)}(1 - x_1^{(1)})D(1 - D)(D - d)w_{11}^{(2)}x_2^{(0)}. \end{aligned} \tag{6.14}$$

Using the steepest descent method to minimize the error function, weight $w_{21}^{(1)}$ should be updated as

$$w_{21}^{(1)} - \frac{\partial E}{\partial w_{21}^{(1)}}. \tag{6.15}$$

Now,

$$\begin{aligned} \frac{\partial E}{\partial w_{21}^{(1)}} &= (D - d)\frac{\partial D}{\partial w_{21}^{(1)}} \\ &= (D - d)D(1 - D)\frac{\partial}{\partial w_{21}^{(1)}}(w_{01}^{(2)} + w_{11}^{(2)}x_1^{(1)} + w_{21}^{(2)}x_2^{(1)}) \\ &= (D - d)D(1 - D)\frac{\partial}{\partial w_{21}^{(1)}}(w_{11}^{(2)}x_1^{(1)}) \\ &= (D - d)D(1 - D)w_{11}^{(2)}x_1^{(1)}(1 - x_1^{(1)})\frac{\partial}{\partial w_{21}^{(1)}}(w_{01}^{(1)} + w_{11}^{(1)}x_1^{(0)}x_1 + w_{21}^{(1)}x_2^{(0)}) \\ &= (D - d)D(1 - D)w_{11}^{(2)}x_1^{(1)}(1 - x_1^{(1)})x_2^{(0)}. \end{aligned} \tag{6.16}$$

We see that (6.15) and (6.16) agree with (6.14).

We may also directly compute the partial derivatives of E with respect to the other weights and verify directly that they agree with step 5 of the back-propagation algorithm:

$$\begin{aligned}
\frac{\partial E}{\partial w_{01}^{(1)}} &= (D-d)D(1-D)w_{11}^{(2)}x_1^{(1)}(1-x_1^{(1)}) \\
\frac{\partial E}{\partial w_{02}^{(1)}} &= (D-d)D(1-D)w_{21}^{(2)}x_2^{(1)}(1-x_2^{(1)}) \\
\frac{\partial E}{\partial w_{11}^{(1)}} &= (D-d)D(1-D)w_{11}^{(2)}x_1^{(1)}(1-x_1^{(1)})x_1^{(0)} \\
\frac{\partial E}{\partial w_{12}^{(1)}} &= (D-d)D(1-D)w_{21}^{(2)}x_2^{(1)}(1-x_2^{(1)})x_1^{(0)} \\
\frac{\partial E}{\partial w_{22}^{(1)}} &= (D-d)D(1-D)w_{21}^{(2)}x_2^{(1)}(1-x_2^{(1)})x_2^{(0)}.
\end{aligned}$$

Example 6.5 *Learning to recognize patterns in a convex region through back-propagation.*

The goal in this example is to find decision boundaries for classifying samples into either the convex region labeled 1 or the complement of this region labeled 0 as shown in Figure 6.12a. The configuration of the net is shown in Figure 6.12b. The weights are initialized to small random values and the inputs to the net are as shown in Figure 6.12a. The back-propagation algorithm is then applied to train the weights of the three decision boundaries. The locations of the three decision boundaries are shown after 1,000 iterations. These decision boundaries correspond to the three hidden nodes $x_1^{(1)}$, $x_2^{(1)}$, and $x_3^{(1)}$. The output node $x_1^{(2)}$ computes the logical *AND* of the three half-planes

$$\begin{aligned}
w_{10}^{(1)}x_0^{(1)} + w_{11}^{(1)}x_1^{(1)} + w_{12}^{(1)}x_2^{(1)} + w_{13}^{(1)}x_3^{(1)} &\geq 0 \\
w_{20}^{(1)}x_0^{(1)} + w_{21}^{(1)}x_1^{(1)} + w_{22}^{(1)}x_2^{(1)} + w_{23}^{(1)}x_3^{(1)} &\geq 0 \\
w_{30}^{(1)}x_0^{(1)} + w_{31}^{(1)}x_1^{(1)} + w_{32}^{(1)}x_2^{(1)} + w_{33}^{(1)}x_3^{(1)} &\geq 0,
\end{aligned}$$

which forms the convex region containing the 1s in Figure 6.12a.

Starting from a random initial set of weights, the step size $c = 0.47$ was the largest step size for which the weights converged to values that correctly classify all the samples for the trial runs. If c was greater than 0.47, the weights converged to values that did not correctly classify the samples. If c was chosen to be too small, convergence to the correct weights was slow.

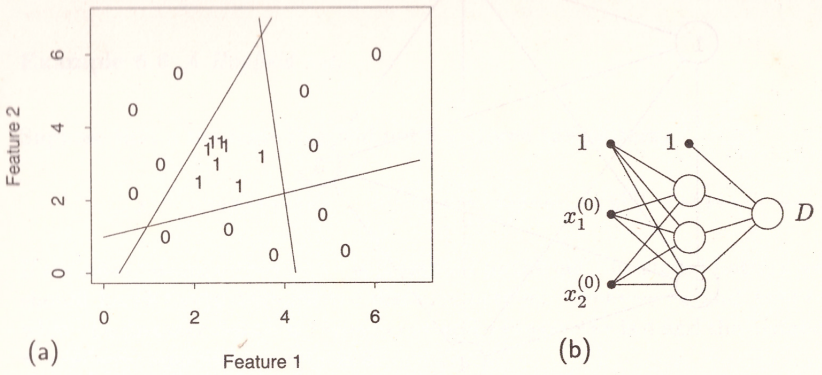

Figure 6.12: (a) The samples and decision boundaries for Example 6.5. (b) The net for Example 6.5.

Using a momentum term in the back-propagation algorithm may increase stability when using a large step size. If $\Delta w_{ij}^{(k)}(t)$ represents the change in the weight $w_{ij}^{(k)}$ which would occur during the tth iteration of the usual back-propagation algorithm, the change in the weight $w_{ij}^{(k)}$ using the momentum term is $\Delta w_{ij}^{(k)}(t) + \alpha \Delta w_{ij}^{(k)}(t-1)$, where α is a positive constant that controls how much importance to give to the momentum term. The momentum term can decrease oscillations that may slow convergence.

6.5 Hopfield Nets

A distinctive feature of biological brains is their ability to recognize a pattern given only an imperfect representation of it. A person can recognize a friend that he or she has not seen for years given only a poor quality photograph of that person; the memory of that friend can then evoke other related memories. This process is an example of **associative memory**. A memory or pattern is accessed by associating it with another pattern, which may be imperfectly formed. By contrast, ordinary computer memory is retrieved by precisely specifying the location or address where the information is stored. A **Hopfield net** is a simple example of an associative memory. A pattern consists of an n-bit sequence of 1s and -1s.

While the nets discussed in the previous sections are strictly feed-forward (a node

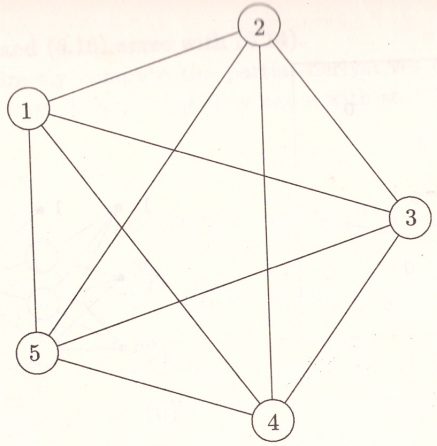

Figure 6.13: The connections of a Hopfield net.

always provides input to another node in the next higher layer), in a Hopfield net every node is connected to every other node in the network. Each connection has an associated weight. Signals can flow in either direction on these connections, and the same weight is used in both directions. Assigning weights to a Hopfield net is interpreted as *storing* a set of example patterns. We explain this in detail in the next subsection. After the storage step, whenever a pattern is presented to the net, the net is supposed to output the stored pattern that is most similar to the input pattern.

In the classification step, the n nodes of a Hopfield net are initialized to the n feature values of a pattern. Each node concurrently computes a weighted sum of the values of all the other nodes as shown in Figure 6.13. It then changes its own value to $+1$ if the weighted sum is nonnegative or to -1 if the weighted sum is negative. When the net ceases to change after a certain point in the iteration process, the net is said to have reached a **stable state**. In practice, a stable state will be detected if the state of the net does not change for some fixed time period depending on the speed and size of the net. If the net functions correctly, it converges to the stored pattern that is closest to the presented pattern, where the distance between two bit strings x and y is the number of positions in which the corresponding bits of x and y disagree. In practice, however, if the net's capacity is not large enough, the net may not produce the stored pattern closest to the input pattern. Experimental results [Hopfield] show that a Hopfield net can usually store approximately $0.15n$ patterns, where n is the number of nodes in the net. If too many patterns are stored, the net will not function correctly. In what follows, we will represent 1 by + and -1 by -.

Example 6.6 *A Hopfield net.*

Suppose that a ten-node Hopfield net has stored the patterns

$$\mathbf{x}^{(1)} = \texttt{++++++++++}, \quad \mathbf{x}^{(2)} = \texttt{+-+-+-+-+-}, \quad \mathbf{x}^{(3)} = \texttt{+++++-----}.$$

If the new pattern ++++++---+- is presented to the net, the stored pattern +++++----- should be retrieved because the distance between +++++----- and ++++++---+- is 1 while the distance between ++++++++++ and ++++++---+- is 4 and the distance between +-+-+-+-+- and ++++++---+- is 3.

Although using a nearest neighbor algorithm to compute the stored pattern closest to the input pattern might be more straightforward, one attractive feature of the Hopfield net is that all the nodes in the net can operate asynchronously in parallel. Suppose that the nodes asynchronously compute the weighted sum of the other nodes and that they recompute and change their states at completely random times, such that this occurs on the average once every time unit. We say the node **fires** when it reads the states of the other nodes, updates its state after recomputing its (possibly) new value, and sends its recomputed state to the other nodes. Note that the state of a node does not necessarily change when the node fires, because the recomputed state may be the same as its previous state. In general the probability that two or more nodes fire at exactly the same time is 0, so the firings can be listed in order of their occurrence. However, even though the firings occur in sequential order, they occur more quickly than they would occur if a conventional computer with a single processor were used. Entry ij in the matrix in Figure 6.14 represents the connection weight between nodes i and j after the three patterns from Example 6.6 are stored. Example 6.7 shows how the values of the net change at successive iterations during a retrieval step.

Example 6.7 *Iterations of a Hopfield net.*

The following table shows the successive firings of the Hopfield net having the weight matrix defined in Figure 6.14 and input pattern +++-+++++-.

	1	2	3	4	5	6	7	8	9	10
1	0	1	3	1	3	−1	1	−1	1	−1
2	1	0	1	3	1	1	−1	1	−1	1
3	3	1	0	1	3	−1	1	−1	1	−1
4	1	3	1	0	1	1	−1	1	−1	1
5	3	1	3	1	0	−1	1	−1	1	−1
6	−1	1	−1	1	−1	0	1	3	1	3
7	1	−1	1	−1	1	1	0	1	3	1
8	−1	1	−1	1	−1	3	1	0	1	3
9	1	−1	1	−1	1	1	3	1	0	1
10	−1	1	−1	1	−1	3	1	3	1	0

Figure 6.14: The weight matrix of the Hopfield net in Example 6.7.

Iteration Number	Node	Fire Time	Current Node Values
1	7	0.123	+++-+++++-
2	10	0.134	+++-++++++
3	5	0.328	+++-++++++
4	10	0.441	+++-++++++
5	4	0.471	++++++++++
6	8	0.524	++++++++++

In this case, the net converges to the stored pattern ++++++++++, which is the closest stored pattern to the input pattern. This example required five iterations for convergence. However, the firings of the nodes occur randomly, so another run might require a different number of iterations to reach a stable state because the firing sequence may be different for another run.

The Storage and Retrieval Algorithms

There are two distinct phases in using a Hopfield net. The first phase uses a **storage algorithm** to compute the matrix of connection weights from the patterns to be stored. This is not an iterative adaptive process; the weight between nodes i and j depends only on the similarity between bits i and j in the set of patterns to be stored. Let

$$\mathbf{x}^{(p)} = (x_1^{(p)}, x_2^{(p)}, \ldots, x_n^{(p)})$$

be the pth pattern that is stored in the net, where $1 \le p \le m$. The weight between nodes i and j is calculated as

$$w_{ij} = \begin{cases} \sum_{p=1}^{m} x_i^{(p)} x_j^{(p)} & i \neq j \\ 0 & i = j \end{cases} \tag{6.17}$$

Thus, the weight between node i and node j is the number of times that the ith component of a stored pattern is the same as the jth component of that same pattern, minus the number of times that they differ, summed over all the stored patterns.

For example, to obtain the weight matrix entry in the second row and the sixth column in Figure 6.14, note that $x_2^{(1)} = 1$, $x_6^{(1)} = 1$, $x_2^{(2)} = -1$, $x_6^{(2)} = -1$, $x_2^{(3)} = 1$, and $x_6^{(3)} = -1$, so (6.17) gives

$$w_{26} = (1)(1) + (-1)(-1) + (1)(-1) = 1.$$

The second phase of using a Hopfield net is the **retrieval algorithm**. Let the input pattern to be retrieved be

$$\mathbf{y} = (y_1, y_2, \ldots, y_n).$$

The processing begins by initializing the value of node i to y_i. Next the nodes begin firing. When node i fires, y_i is set to

$$\operatorname{sgn}\left(\sum_{j=1}^{n} w_{ij} y_j\right), \tag{6.18}$$

where sgn $s = +1$, if $s \ge 0$, and sgn $s = -1$, if $s < 0$. These calculations are repeated until the states stop changing. To see how an input pattern can be iteratively updated to retrieve a stored pattern, we rewrite the sum in (6.18) as

$$\sum_{j=1}^{n} w_{ij} y_j = \sum_{j=1, j\neq i}^{n} \sum_{p=1}^{m} x_i^{(p)} x_j^{(p)} y_j = \sum_{p=1}^{m} x_i^{(p)} \left(\sum_{j=1, j\neq i}^{n} x_j^{(p)} y_j \right). \tag{6.19}$$

Now if the input pattern $\mathbf{y}$ is close to the stored pattern $\mathbf{x}^{(p)}$, the quantity in parentheses in (6.19) will be close to n. If $\mathbf{y}$ and $\mathbf{x}^{(p)}$ are very different from each other, this quantity will be close to $-n$. The pros and cons are then summed for each pattern. If $x_i^{(p)}$ is $+1$ and the quantity in parentheses is positive, then the $+1$ is probably correct, so a positive number is contributed to the sum in (6.19). If the quantity in parentheses is negative, then the $+1$ is probably incorrect, so a negative number is contributed to the sum in (6.19). The situation is reversed if $x_i^{(p)}$ is -1. In either case, if (6.19) is nonnegative, the most likely value of $x_i^{(p)}$ is $+1$, so $x_i^{(p)}$ is changed to $+1$. If (6.19) is negative, $x_i^{(p)}$ is changed to -1.

In order to show that the net eventually converges, we define

$$E = \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} w_{ij}y_i y_j, \tag{6.20}$$

which is called the **Liapunov function** of the net. We claim that when a node, say k, fires, E does not increase. To see this, first suppose that node k fires and y_k changes from $+1$ to -1. We may rewrite E as

$$E = y_k \sum_{j=1}^{n} w_{kj}y_j + \frac{1}{2}\sum_{i=1, i\neq k}^{n}\sum_{j=1, j\neq k}^{n} w_{ij}y_i y_j.$$

If we let ΔE denote the original value of E (when $y_k = +1$) minus the new value of E (when $y_k = -1$) and Δy denote the original value of y_k $(+1)$ minus the new value of y_k (-1), then

$$\Delta E = \Delta y \sum_{j=1}^{n} w_{kj}y_j = 2\left(\sum_{j=1}^{n} w_{kj}y_j\right). \tag{6.21}$$

Since y_k changed from $+1$ to -1, the term in brackets is nonnegative. Therefore $\Delta E \geq 0$. Thus if node k fires and y_k changes from $+1$ to -1, E does not increase.

Similarly, if node k fires and y_k changes from -1 to $+1$,

$$\Delta E = -2\left(\sum_{j=1}^{n} w_{kj}y_j\right).$$

In this case, since y_k changed from -1 to $+1$, the term in brackets is negative. Therefore $\Delta E > 0$. Thus if node k fires and y_k changes from -1 to $+1$, E decreases.

Since the weights w_{ij} are integers and each y_i is ± 1, it follows from (6.21) that when E decreases, it decreases by at least 2. Further, since E is bounded below, it cannot decrease indefinitely. (A crude lower bound for E is $-\frac{1}{2}n^2W$, where W is the maximum of $\{|w_{ij}|\}$. The term n^2 occurs because there are n^2 summands in (6.20).)

Suppose, by way of contradiction, that nodes continue to fire and the y_i continue to change. Then E will never increase. Furthermore, eventually nodes will change from -1 to $+1$, and E will, therefore, decrease infinitely often. Since this is impossible, it follows that eventually the net will converge to a **stable state**, that is, a state that will not change as the nodes continue to fire. The stable state to which the net converges may not be unique, and it may not even be one of the patterns that were stored (see Problem 6.14). Such situations arise when the number of patterns stored is too large compared with the number of nodes in the network. The following example shows what happens when too many patterns are stored in a net.

Example 6.8 *A Hopfield net with too many stored patterns.*

Suppose the eight patterns

---, --+, -+-, -++, +--, +-+, ++-, +++

are stored in a net with three nodes. The matrix of weights w_{ij} becomes

	1	2	3
1	0	0	0
2	0	0	0
3	0	0	0

.

This means that every value computed by (6.18) is 0 which has sign 1 by convention. The result is that every input pattern will eventually converge to +++. In this example, the net has only one stable state (namely +++).

If we attempt to store the five patterns

--- -++ +-- ++- +++

in the net, the following weight matrix is produced:

	1	2	3
1	0	1	−1
2	1	0	3
3	−1	3	0

.

This results in a Hopfield net with only two stable states +-- and -++.

Hopfield nets have been used to recognize characters in binary images [Schalkoff]. Although a major advantage of Hopfield nets is that all the nodes can operate asynchronously, a major disadvantage is that every node must be connected to every other node. This can become unwieldy in a net with a large number of nodes.

6.6 An Application: Classifying Sex from Facial Images

Among applications that use neural networks are optical character recognition, handwritten character recognition, speech recognition, robot control of moving vehicles, control of industrial processes such as the operation of nuclear power plants, and screening of loan applications. Some of these systems are in prototype stage; others are in use in industrial settings. At least 20 researchers have designed networks that

can distinguish facial photographs of men and women, for example, with error rates in the 10 to 30 percent range (e.g., [Brunelli], [Burton], [Fleming], [O'Toole]). In this section, we describe neural networks used by Golomb, Lawrence, and Sejnowski to determine the sex of a human from a picture of the person's face ([Golomb]). Although this problem may not have immediate applicability, facial recognition could be useful in security and law enforcement applications. Also, certain diseases such as Down's syndrome are associated with facial characteristics.

The data were images of 45 male and 45 female faces. No facial hair, jewelry, or makeup was present in any of the images. Hints of the correct sex due to clothes were eliminated by placing a white cloth around the neck of each subject. Before the images were analyzed by the neural nets, they were normalized so that the eyes were the same distance apart on a horizontal line, and the mouth was located at a given distance below this line. After normalization, each image was 30×30 pixels with 256 gray levels.

If an image was directly input to a neural network, the network would have 900 inputs—one for each pixel. In this system, however, the number of inputs was reduced from 900 to 40 with minimal loss of information in the sense that the original image could be adequately reconstructed from the reduced set of 40 inputs. Such a technique is called **data compression** and is very important in image processing because digital images tend to be so large. We explain how the image was compressed later, but here we note that the compression itself used a neural network.

The classification network (see Figure 6.15a) had 40 input nodes, a hidden layer of 2 to 40 nodes, and one output node. Every input node was connected to every node in the hidden layer, and every node in the hidden layer was connected to the output node. In addition to changing the weights, the performance of the network was optimized by changing the number of nodes in the hidden layer. The input to the network was the compressed image, and the output was a value between 0 and 1, with values greater than 0.5 indicating a male and values less than 0.5 indicating a female. The network was trained and tested using a version of the leaving-one-out technique; the network was trained on 80 faces using the back-propagation algorithm (see Section 6.4), and then tested on the 10 remaining faces. Eight tests of the classification network were run, each using a different set of faces for training and classification.

To compare the performance of the network with humans, five people each classified all 90 faces. The average error by the humans was 11.6 percent. By contrast, the classification network with 10 nodes in the hidden layer had an average error of 8.1 percent (averaged over the eight tests). The authors noted that humans and the network tended to misclassify the same faces.

The compression network (see Figure 6.15b) had 900 input nodes that represented the original image, and 40 output nodes that represented the compressed image. There was no hidden layer. Every input node is connected to every output node. This network was trained by using an auxiliary network obtained from the compression network by

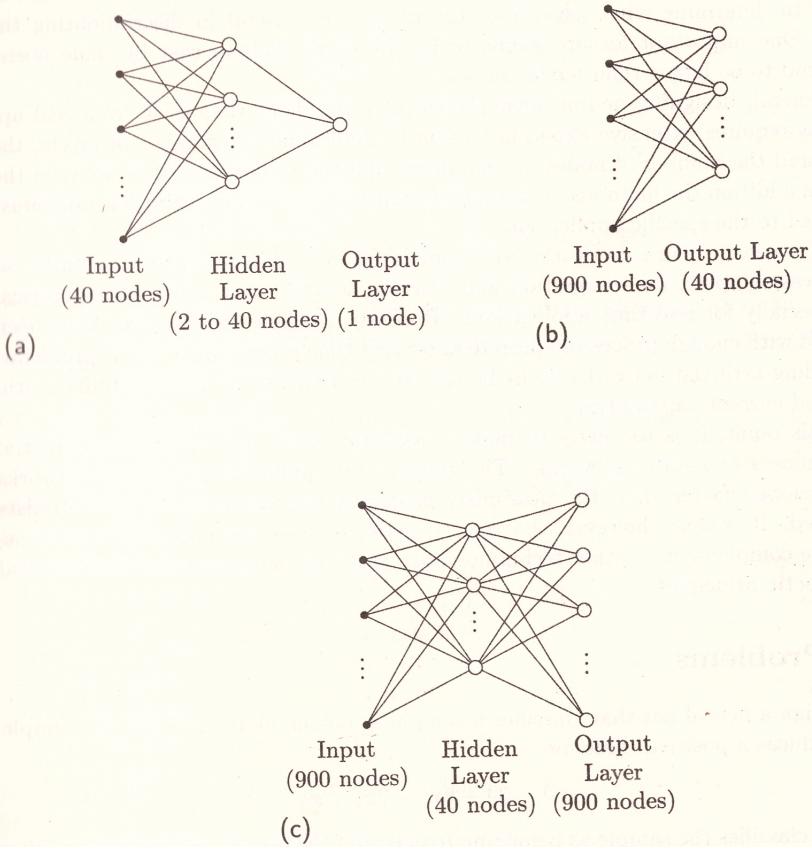

Figure 6.15: (a) The classification network. (b) The compression network. (c) The auxiliary network. The auxiliary network is used to train the compression network.

putting the 40 output nodes into a hidden layer, adding an output layer of 900 nodes, and connecting every node in the hidden layer to every output node (see Figure 6.15c). This auxiliary network was trained using the back-propagation algorithm on all 90 faces. Here the goal was to have the output equal to the input at each of the 900 nodes. The weights obtained for the connections between the nodes in the input and hidden layers in the auxiliary network were then used for the corresponding connections in the compression network. By examining the weights in the trained networks, it was possible to determine what aspects of the image were useful in discriminating the classes. One important feature seems to be the dark shadows cast by male noses, which tend to be larger than female noses.

In general, designing neural networks to solve problems that arise from real applications requires extensive experimentation to determine the number of layers, the weights and the number of nodes in each layer, and the interconnections between the nodes. In addition the methods of storing information and implementing learning must be tailored to the specific application.

Until recently neural networks were simulated on single-processor machines, so the processing speed for recognition and learning was not fast enough for practical use—especially for real-time applications. Recently, parallel architectures have been developed with enough processor speed to allow real-time processing on some problems, but building artificial networks with the size and complexity of human brains is still far beyond current capabilities.

At this point, it is too early to make a realistic assessment of the full potential and usefulness of neural networks. The theory and application of neural networks have developed faster than the pessimists projected but slower than the optimists anticipated. It is clear, however, that neural networks have the potential to serve as a valuable complement to other techniques such as those that are based on statistical and syntactic principles.

6.7 Problems

6.1. Design a neural net that classifies a sample as belonging to class 1 if the sample produces a positive value for

$$D = 34 + 8x_1 - 7x_2 + x_3,$$

and classifies the sample as belonging to class 0 if the sample produces a negative value for D. [**Ans:** $w_{01}^{(1)} = 34$, $w_{11}^{(1)} = 8$, $w_{21}^{(1)} = -7$, $w_{31}^{(1)} = 1$.]

6.2. Construct a two-layer net with two input values x_1 and x_2 that outputs 1 if $x_1 > x_2$ and 0 otherwise.

6.3. Construct a two-layer net with two input values x_1 and x_2 that outputs 1 if $x_1 \leq x_2$ and 0 otherwise.

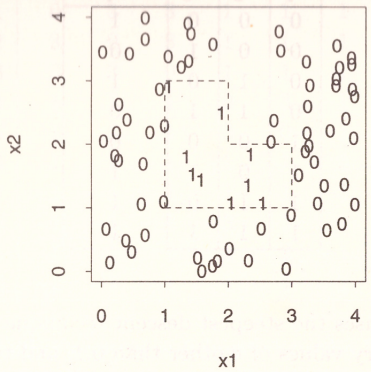

Figure 6.16: The nonconvex decision region for Problem 6.4.

6.4. Construct a neural net using a 0/1, discontinuous threshold function that outputs 1 if the pattern is in the nonconvex class containing the ones shown in Figure 6.16 and 0 otherwise.

6.5. Design a neural network to compute the following function:

x_1	x_2	x_3	Output
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

[**Ans:** Build a hidden layer that computes

$$\begin{aligned} D_1 &= x_1 + x_2 + x_3 \\ D_2 &= x_1 + (1 - x_2) + x_3 \\ D_3 &= (1 - x_1) + x_2 + x_3 \\ D_4 &= (1 - x_1) + x_2 + (1 - x_3). \end{aligned}$$

Then AND the output of the hidden layers.]

6.6. Design a neural network to compute the following function:

x_1	x_2	x_3	Output
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

6.7. Write a program that uses the steepest descent technique to minimize the function of Example 6.1. Try values of c other than 0.1, and try initial values for x_1 and x_2 other than 0.0.

6.8. Show that the direction of steepest descent of a differentiable function F of M variables is the vector

$$\left(-c\frac{\partial F}{\partial w_1}, \ldots, -c\frac{\partial F}{\partial w_M}\right)$$

where c is a positive constant of proportionality.

6.9. Show that the sigmoid threshold function $R(s)$ is antisymmetric about the point $(0, 1/2)$.

6.10. Show that the derivative of the sigmoid function $R(s)$ is $R(s)(1 - R(s))$.

6.11. For the neural net of Figure 6.11, compute the partial derivatives of E with respect to

(a) $w_{01}^{(1)}$ (b) $w_{02}^{(1)}$ (c) $w_{11}^{(1)}$ (d) $w_{12}^{(1)}$ (e) $w_{22}^{(1)}$ (f) $w_{01}^{(2)}$
(g) $w_{21}^{(2)}$

and verify that they agree with step 5 of the back-propagation algorithm.

6.12. Derive equation (6.10).

6.13. Compute the weight matrix for a ten-node Hopfield net that stores the patterns

+++−−+−+−−, −−−−−−++−−, −−−−−+−+−+.

Ans:

	1	2	3	4	5	6	7	8	9	10
1	0	3	3	1	1	1	−1	−1	1	−1
2	3	0	3	1	1	1	−1	−1	1	−1
3	3	3	0	1	1	1	−1	−1	1	−1
4	1	1	1	0	3	−1	1	−3	3	1
5	1	1	1	3	0	−1	1	−3	3	1
6	1	1	1	−1	−1	0	−3	1	−1	1
7	−1	−1	−1	1	1	−3	0	−1	1	−1
8	−1	−1	−1	−3	−3	1	−1	0	−3	−1
9	1	1	1	3	3	−1	1	−3	0	1
10	−1	−1	−1	1	1	1	−1	−1	1	0

6.14. Show a set of possible successive firings when the Hopfield net of Problem 6.13 receives the input pattern ++++++++. To which pattern does the network converge?

[**Ans:** +++++++−−++ or +++++−+−+−.]

6.15. Find a set of patterns that, when stored in a four-node Hopfield net, produces a weight matrix with all entries zero.

6.16. Is there a set of patterns that, when stored in a four-node Hopfield net, produces a weight matrix with all entries, except those on the main diagonal, one? If so, give such a set of patterns; otherwise, give an argument that shows there is no such set of patterns.

6.17. Find a set of patterns that, when stored in a four-node Hopfield net, produces exactly three stable states. List the stable states.

6.18. Suggest a way to classify sex from facial images (as in Section 6.6) using statistical techniques.