Bibliography

Aikens, J.S. (1984). A representation scheme using both frames and rules, *Rule-Based Expert Systems*, B.G. Buchanan and E.H. Shortliffe, eds, Addison-Wesley, Reading, MA, pp. 424–440.

Alpert, S.R., Woyak, S.W., Shrobe, H.J. and Arrowood, L.F. (1990). Guest Editors Introduction: Object-oriented programming in AI, *IEEE Expert*, 5(6), 6–7.

Brachman, R.J. and Levesque, H.J. (2004). *Knowledge Representation and Reasoning*. Morgan Kaufmann, Los Altos, CA.

Budd, T. (2002). *An Introduction to Object Oriented Programming*, 3rd edn. Addison-Wesley, Reading, MA.

Fikes, R. and Kehler, T. (1985). The role of frame-based representation in reasoning, *Communications of the ACM*, 28(9), 904–920.

Goldstein, I. and Papert, S. (1977). Artificial intelligence, language, and the study of knowledge, *Cognitive Science*, 1(1), 84–123.

Jackson, P. (1999). *Introduction to Expert Systems*, 3rd edn. Addison-Wesley, Harlow.

Levesque, H.J. and Brachman, R.J. (1985). A fundamental trade-off in knowledge representation and reasoning. *Readings in Knowledge Representation*, R.J. Brachman and H.J. Levesque, eds, Morgan Kaufmann, Los Altos, CA.

Luger, G.F. (2002). *Artificial Intelligence: Structures and Strategies for Complex Problem Solving*, 4th edn. Addison-Wesley, Harlow.

Payne, E.C. and McArthur, R.C. (1990). *Developing Expert Systems: A Knowledge Engineer's Handbook for Rules and Objects*. John Wiley, New York.

Rosson, M.B. and Alpert, S.R. (1990). The cognitive consequences of object-oriented design, *Human–Computer Interaction*, 5(4), 345–79.

Stefik, M.J. (1979). An examination of frame-structured representation systems, *Proceedings of the 6th International Joint Conference on Artificial Intelligence*, Tokyo, Japan, August 1979, pp. 845–852.

Stefik, M.J. (1995). *Introduction to Knowledge Systems*. Morgan Kaufmann, San Francisco, CA.

Stefik, M.J. and Bobrow, D.G. (1986). Object-oriented programming: themes and variations, *Al Magazine*, 6(4), 40–62.

Touretzky, D.S. (1986). *The Mathematics of Inheritance Systems*. Morgan Kaufmann, Los Altos, CA.

Waterman, D.A. (1986). *A Guide to Expert Systems*. Addison-Wesley, Reading, MA.

Winston, P.H. (1977). Representing knowledge in frames, Chapter 7 of *Artificial Intelligence*, Addison-Wesley, Reading, MA, pp. 181–187.

Winston, P.H. (1992). *Artificial Intelligence*, 3rd edn. Addison-Wesley, Reading, MA.

Artificial neural networks 6

In which we consider how our brains work and how to build and train artificial neural networks.

6.1 Introduction, or how the brain works

'The computer hasn't proved anything yet,' angry Garry Kasparov, the world chess champion, said after his defeat in New York in May 1997. 'If we were playing a real competitive match, I would tear down Deep Blue into pieces.'

But Kasparov's efforts to downplay the significance of his defeat in the six-game match was futile. The fact that Kasparov – probably the greatest chess player the world has seen – was beaten by a computer marked a turning point in the quest for intelligent machines.

The IBM supercomputer called Deep Blue was capable of analysing 200 million positions a second, and it appeared to be displaying intelligent thoughts. At one stage Kasparov even accused the machine of cheating!

> 'There were many, many discoveries in this match, and one of them was that sometimes the computer plays very, very human moves.
>
> It deeply understands positional factors. And that is an outstanding scientific achievement.'

Traditionally, it has been assumed that to beat an expert in a chess game, a computer would have to formulate a strategy that goes beyond simply doing a great number of 'look-ahead' moves per second. Chess-playing programs must be able to improve their performance with experience or, in other words, a machine must be capable of learning.

What is machine learning?

In general, machine learning involves adaptive mechanisms that enable computers to learn from experience, learn by example and learn by analogy. Learning capabilities can improve the performance of an intelligent system over time. Machine learning mechanisms form the basis for adaptive systems. The most popular approaches to machine learning are **artificial neural networks** and **genetic algorithms**. This chapter is dedicated to neural networks.

What is a neural network?

A neural network can be defined as a model of reasoning based on the human brain. The brain consists of a densely interconnected set of nerve cells, or basic information-processing units, called **neurons**. The human brain incorporates nearly 10 billion neurons and 60 trillion connections, **synapses**, between them (Shepherd and Koch, 1990). By using multiple neurons simultaneously, the brain can perform its functions much faster than the fastest computers in existence today.

Although each neuron has a very simple structure, an army of such elements constitutes a tremendous processing power. A neuron consists of a cell body, **soma**, a number of fibres called **dendrites**, and a single long fibre called the **axon**. While dendrites branch into a network around the soma, the axon stretches out to the dendrites and somas of other neurons. Figure 6.1 is a schematic drawing of a neural network.

Signals are propagated from one neuron to another by complex electro-chemical reactions. Chemical substances released from the synapses cause a change in the electrical potential of the cell body. When the potential reaches its threshold, an electrical pulse, **action potential**, is sent down through the axon. The pulse spreads out and eventually reaches synapses, causing them to increase or decrease their potential. However, the most interesting finding is that a neural network exhibits **plasticity**. In response to the stimulation pattern, neurons demonstrate long-term changes in the strength of their connections. Neurons also can form new connections with other neurons. Even entire collections of neurons may sometimes migrate from one place to another. These mechanisms form the basis for learning in the brain.

Our brain can be considered as a highly complex, nonlinear and parallel information-processing system. Information is stored and processed in a neural network simultaneously throughout the whole network, rather than at specific locations. In other words, in neural networks, both data and its processing are **global** rather than local.

Owing to the plasticity, connections between neurons leading to the 'right answer' are strengthened while those leading to the 'wrong answer' weaken. As a result, neural networks have the ability to learn through experience.

Learning is a fundamental and essential characteristic of biological neural networks. The ease and naturalness with which they can learn led to attempts to emulate a biological neural network in a computer.

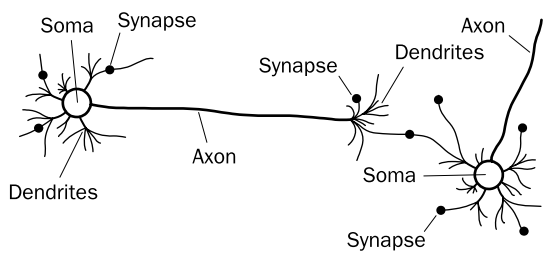


Figure 6.1 Biological neural network

Although a present-day artificial neural network (ANN) resembles the human brain much as a paper plane resembles a supersonic jet, it is a big step forward. ANNs are capable of 'learning', that is, they use experience to improve their performance. When exposed to a sufficient number of samples, ANNs can generalise to others they have not yet encountered. They can recognise handwritten characters, identify words in human speech, and detect explosives at airports. Moreover, ANNs can observe patterns that human experts fail to recognise. For example, Chase Manhattan Bank used a neural network to examine an array of information about the use of stolen credit cards – and discovered that the most suspicious sales were for women's shoes costing between $40 and $80.

How do artificial neural nets model the brain?

An artificial neural network consists of a number of very simple and highly interconnected processors, also called neurons, which are analogous to the biological neurons in the brain. The neurons are connected by weighted links passing signals from one neuron to another. Each neuron receives a number of input signals through its connections; however, it never produces more than a single output signal. The output signal is transmitted through the neuron's outgoing connection (corresponding to the biological axon). The outgoing connection, in turn, splits into a number of branches that transmit the same signal (the signal is not divided among these branches in any way). The outgoing branches terminate at the incoming connections of other neurons in the network. Figure 6.2 represents connections of a typical ANN, and Table 6.1 shows the analogy between biological and artificial neural networks (Medsker and Liebowitz, 1994).

How does an artificial neural network 'learn'?

The neurons are connected by **links**, and each link has a **numerical weight** associated with it. Weights are the basic means of long-term memory in ANNs. They express the strength, or in other words importance, of each neuron input. A neural network 'learns' through repeated adjustments of these weights.

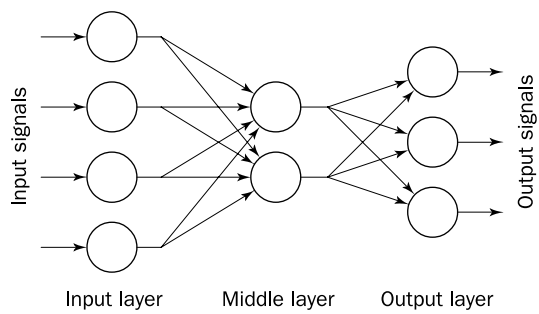


Figure 6.2 Architecture of a typical artificial neural network

Table 6.1 Analogy between biological and artificial neural networks

Biological neural network	Artificial neural network
Soma	Neuron
Dendrite	Input
Axon	Output
Synapse	Weight

But does the neural network know how to adjust the weights?
As shown in Figure 6.2, a typical ANN is made up of a hierarchy of layers, and the neurons in the networks are arranged along these layers. The neurons connected to the external environment form input and output layers. The weights are modified to bring the network input/output behaviour into line with that of the environment.

Each neuron is an elementary information-processing unit. It has a means of computing its **activation level** given the inputs and numerical weights.

To build an artificial neural network, we must decide first how many neurons are to be used and how the neurons are to be connected to form a network. In other words, we must first choose the network architecture. Then we decide which learning algorithm to use. And finally we train the neural network, that is, we initialise the weights of the network and update the weights from a set of training examples.

Let us begin with a neuron, the basic building element of an ANN.

6.2 The neuron as a simple computing element

A neuron receives several signals from its input links, computes a new activation level and sends it as an output signal through the output links. The input signal can be raw data or outputs of other neurons. The output signal can be either a final solution to the problem or an input to other neurons. Figure 6.3 shows a typical neuron.

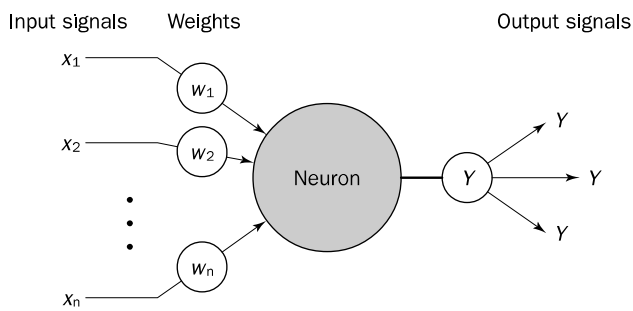


Figure 6.3 Diagram of a neuron

How does the neuron determine its output?

In 1943, Warren McCulloch and Walter Pitts proposed a very simple idea that is still the basis for most artificial neural networks.

The neuron computes the weighted sum of the input signals and compares the result with a threshold value, θ. If the net input is less than the threshold, the neuron output is -1. But if the net input is greater than or equal to the threshold, the neuron becomes activated and its output attains a value $+1$ (McCulloch and Pitts, 1943).

In other words, the neuron uses the following transfer or **activation function**:

$$X = \sum_{i=1}^{n} x_i w_i \tag{6.1}$$

$$Y = \begin{cases} +1 & \text{if } X \geqslant \theta \\ -1 & \text{if X} < \theta \end{cases}$$

where X is the net weighted input to the neuron, x_i is the value of input i, w_i is the weight of input i, n is the number of neuron inputs, and Y is the output of the neuron.

This type of activation function is called a **sign function**.

Thus the actual output of the neuron with a sign activation function can be represented as

$$Y = sign\left[\sum_{i=1}^{n} x_i w_i - \theta\right] \tag{6.2}$$

Is the sign function the only activation function used by neurons?

Many activation functions have been tested, but only a few have found practical applications. Four common choices – the step, sign, linear and sigmoid functions – are illustrated in Figure 6.4.

The **step** and **sign** activation functions, also called **hard limit functions**, are often used in decision-making neurons for classification and pattern recognition tasks.

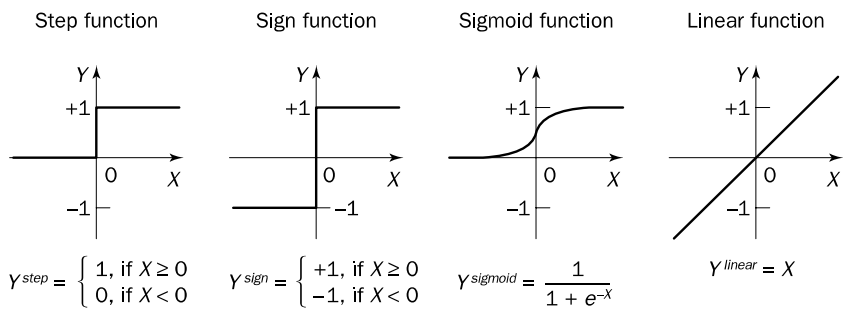


Figure 6.4 Activation functions of a neuron

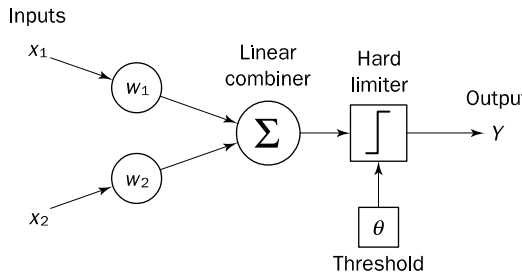


Figure 6.5 Single-layer two-input perceptron

The **sigmoid function** transforms the input, which can have any value between plus and minus infinity, into a reasonable value in the range between 0 and 1. Neurons with this function are used in the back-propagation networks.

The **linear activation function** provides an output equal to the neuron weighted input. Neurons with the linear function are often used for linear approximation.

Can a single neuron learn a task?

In 1958, Frank Rosenblatt introduced a training algorithm that provided the first procedure for training a simple ANN: a **perceptron** (Rosenblatt, 1958). The perceptron is the simplest form of a neural network. It consists of a single neuron with **adjustable** synaptic weights and a **hard limiter**. A single-layer two-input perceptron is shown in Figure 6.5.

6.3 The perceptron

The operation of Rosenblatt's perceptron is based on the McCulloch and Pitts neuron model. The model consists of a linear combiner followed by a hard limiter. The weighted sum of the inputs is applied to the hard limiter, which produces an output equal to $+1$ if its input is positive and -1 if it is negative. The aim of the perceptron is to classify inputs, or in other words externally applied stimuli $x_1, x_2, \ldots, x_n$, into one of two classes, say A_1 and A_2. Thus, in the case of an elementary perceptron, the n-dimensional space is divided by a **hyperplane** into two decision regions. The hyperplane is defined by the **linearly separable** function

$$\sum_{i=1}^{n} x_i w_i - \theta = 0 \tag{6.3}$$

For the case of two inputs, x_1 and x_2, the decision boundary takes the form of a straight line shown in bold in Figure 6.6(a). Point 1, which lies above the boundary line, belongs to class A_1; and point 2, which lies below the line, belongs to class A_2. The threshold θ can be used to shift the decision boundary.

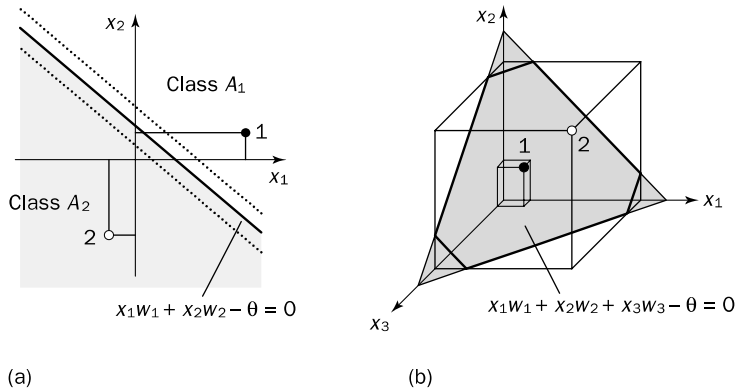


Figure 6.6 Linear separability in the perceptrons: (a) two-input perceptron; (b) three-input perceptron

With three inputs the hyperplane can still be visualised. Figure 6.6(b) shows three dimensions for the three-input perceptron. The separating plane here is defined by the equation

$$x_1w_1 + x_2w_2 + x_3w_3 - \theta = 0$$

But how does the perceptron learn its classification tasks?
This is done by making small adjustments in the weights to reduce the difference between the actual and desired outputs of the perceptron. The initial weights are randomly assigned, usually in the range $[-0.5, 0.5]$, and then updated to obtain the output consistent with the training examples. For a perceptron, the process of weight updating is particularly simple. If at iteration p, the actual output is $Y(p)$ and the desired output is $Y_d(p)$, then the error is given by

$$e(p) = Y_d(p) - Y(p) \qquad \text{where } p = 1, 2, 3, \ldots \tag{6.4}$$

Iteration p here refers to the pth training example presented to the perceptron.

If the error, $e(p)$, is positive, we need to increase perceptron output $Y(p)$, but if it is negative, we need to decrease $Y(p)$. Taking into account that each perceptron input contributes $x_i(p) \times w_i(p)$ to the total input $X(p)$, we find that if input value $x_i(p)$ is positive, an increase in its weight $w_i(p)$ tends to increase perceptron output $Y(p)$, whereas if $x_i(p)$ is negative, an increase in $w_i(p)$ tends to decrease $Y(p)$. Thus, the following **perceptron learning rule** can be established:

$$w_i(p+1) = w_i(p) + \alpha \times x_i(p) \times e(p), \tag{6.5}$$

where α is the **learning rate**, a positive constant less than unity.

The perceptron learning rule was first proposed by Rosenblatt in 1960 (Rosenblatt, 1960). Using this rule we can derive the perceptron training algorithm for classification tasks.

Step 1: ***Initialisation***

Set initial weights $w_1, w_2, \ldots, w_n$ and threshold θ to random numbers in the range $[-0.5, 0.5]$.

Step 2: ***Activation***

Activate the perceptron by applying inputs $x_1(p), x_2(p), \ldots, x_n(p)$ and desired output $Y_d(p)$. Calculate the actual output at iteration $p = 1$

$$Y(p) = step\left[\sum_{i=1}^{n} x_i(p)w_i(p) - \theta\right], \tag{6.6}$$

where n is the number of the perceptron inputs, and *step* is a step activation function.

Step 3: ***Weight training***

Update the weights of the perceptron

$$w_i(p+1) = w_i(p) + \Delta w_i(p), \tag{6.7}$$

where $\Delta w_i(p)$ is the weight correction at iteration p.

The weight correction is computed by the **delta rule**:

$$\Delta w_i(p) = \alpha \times x_i(p) \times e(p) \tag{6.8}$$

Step 4: ***Iteration***

Increase iteration p by one, go back to Step 2 and repeat the process until convergence.

Can we train a perceptron to perform basic logical operations such as AND, OR or Exclusive-OR?

The truth tables for the operations AND, OR and Exclusive-OR are shown in Table 6.2. The table presents all possible combinations of values for two variables, x_1 and x_2, and the results of the operations. The perceptron must be trained to classify the input patterns.

Let us first consider the operation AND. After completing the initialisation step, the perceptron is activated by the sequence of four input patterns representing an **epoch**. The perceptron weights are updated after each activation. This process is repeated until all the weights converge to a uniform set of values. The results are shown in Table 6.3.

Table 6.2 Truth tables for the basic logical operations

Input variables		AND	OR	Exclusive-OR
x_1	x_2	$x_1 \cap x_2$	$x_1 \cup x_2$	$x_1 \oplus x_2$
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

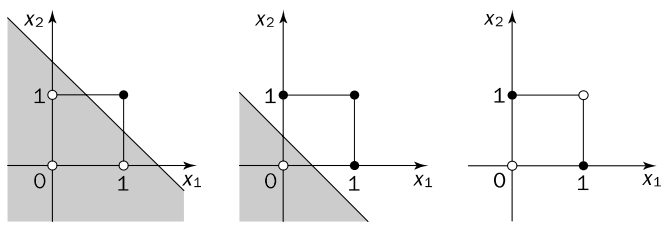
Table 6.3 Example of perceptron learning: the logical operation AND

Epoch	Inputs		Desired output	Initial weights		Actual output	Error	Final weights	
	x_1	x_2	Y_d	w_1	w_2	Y	e	w_1	w_2
1	0	0	0	0.3	−0.1	0	0	0.3	−0.1
	0	1	0	0.3	−0.1	0	0	0.3	−0.1
	1	0	0	0.3	−0.1	1	−1	0.2	−0.1
	1	1	1	0.2	−0.1	0	1	0.3	0.0
2	0	0	0	0.3	0.0	0	0	0.3	0.0
	0	1	0	0.3	0.0	0	0	0.3	0.0
	1	0	0	0.3	0.0	1	−1	0.2	0.0
	1	1	1	0.2	0.0	1	0	0.2	0.0
3	0	0	0	0.2	0.0	0	0	0.2	0.0
	0	1	0	0.2	0.0	0	0	0.2	0.0
	1	0	0	0.2	0.0	1	−1	0.1	0.0
	1	1	1	0.1	0.0	0	1	0.2	0.1
4	0	0	0	0.2	0.1	0	0	0.2	0.1
	0	1	0	0.2	0.1	0	0	0.2	0.1
	1	0	0	0.2	0.1	1	−1	0.1	0.1
	1	1	1	0.1	0.1	1	0	0.1	0.1
5	0	0	0	0.1	0.1	0	0	0.1	0.1
	0	1	0	0.1	0.1	0	0	0.1	0.1
	1	0	0	0.1	0.1	0	0	0.1	0.1
	1	1	1	0.1	0.1	1	0	0.1	0.1

Threshold: $\theta = 0.2$; learning rate: $\alpha = 0.1$.

In a similar manner, the perceptron can learn the operation OR. However, a single-layer perceptron cannot be trained to perform the operation Exclusive-OR.

A little geometry can help us to understand why this is. Figure 6.7 represents the AND, OR and Exclusive-OR functions as two-dimensional plots based on the values of the two inputs. Points in the input space where the function output is 1 are indicated by black dots, and points where the output is 0 are indicated by white dots.



(a) AND ($x_1 \cap x_2$) (b) OR ($x_1 \cup x_2$) (c) Exclusive-OR ($x_1 \oplus x_2$)

Figure 6.7 Two-dimensional plots of basic logical operations

In Figures 6.7(a) and (b), we can draw a line so that black dots are on one side and white dots on the other, but dots shown in Figure 6.7(c) are not separable by a single line. A perceptron is able to represent a function only if there is some line that separates all the black dots from all the white dots. Such functions are called **linearly separable**. Therefore, a perceptron can learn the operations AND and OR, but not Exclusive-OR.

But why can a perceptron learn only linearly separable functions?

The fact that a perceptron can learn only linearly separable functions directly follows from Eq. (6.1). The perceptron output Y is 1 only if the total weighted input X is greater than or equal to the threshold value θ. This means that the entire input space is divided in two along a boundary defined by $X = \theta$. For example, a separating line for the operation AND is defined by the equation

$$x_1 w_1 + x_2 w_2 = \theta$$

If we substitute values for weights w_1 and w_2 and threshold θ given in Table 6.3, we obtain one of the possible separating lines as

$$0.1x_1 + 0.1x_2 = 0.2$$

or

$$x_1 + x_2 = 2$$

Thus, the region below the boundary line, where the output is 0, is given by

$$x_1 + x_2 - 2 < 0,$$

and the region above this line, where the output is 1, is given by

$$x_1 + x_2 - 2 \geqslant 0$$

The fact that a perceptron can learn only linear separable functions is rather bad news, because there are not many such functions.

Can we do better by using a sigmoidal or linear element in place of the hard limiter?

Single-layer perceptrons make decisions in the same way, regardless of the activation function used by the perceptron (Shynk, 1990; Shynk and Bershad, 1992). It means that a single-layer perceptron can classify only linearly separable patterns, regardless of whether we use a hard-limit or soft-limit activation function.

The computational limitations of a perceptron were mathematically analysed in Minsky and Papert's famous book *Perceptrons* (Minsky and Papert, 1969). They proved that Rosenblatt's perceptron cannot make global generalisations on the basis of examples learned locally. Moreover, Minsky and Papert concluded that

the limitations of a single-layer perceptron would also hold true for multilayer neural networks. This conclusion certainly did not encourage further research on artificial neural networks.

How do we cope with problems which are not linearly separable?
To cope with such problems we need multilayer neural networks. In fact, history has proved that the limitations of Rosenblatt's perceptron can be overcome by advanced forms of neural networks, for example multilayer perceptrons trained with the back-propagation algorithm.

6.4 Multilayer neural networks

A multilayer perceptron is a feedforward neural network with one or more hidden layers. Typically, the network consists of an **input layer** of source neurons, at least one middle or **hidden layer** of computational neurons, and an **output layer** of computational neurons. The input signals are propagated in a forward direction on a layer-by-layer basis. A multilayer perceptron with two hidden layers is shown in Figure 6.8.

But why do we need a hidden layer?
Each layer in a multilayer neural network has its own specific function. The input layer accepts input signals from the outside world and redistributes these signals to all neurons in the hidden layer. Actually, the input layer rarely includes computing neurons, and thus does not process input patterns. The output layer accepts output signals, or in other words a stimulus pattern, from the hidden layer and establishes the output pattern of the entire network.

Neurons in the hidden layer detect the features; the weights of the neurons represent the features hidden in the input patterns. These features are then used by the output layer in determining the output pattern.

With one hidden layer, we can represent any continuous function of the input signals, and with two hidden layers even discontinuous functions can be represented.

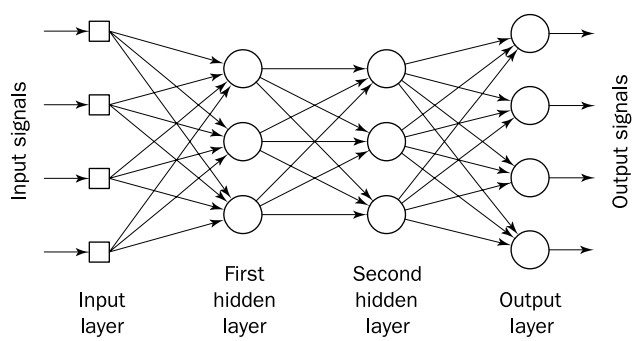


Figure 6.8 Multilayer perceptron with two hidden layers

Why is a middle layer in a multilayer network called a 'hidden' layer? What does this layer hide?

A hidden layer 'hides' its desired output. Neurons in the hidden layer cannot be observed through the input/output behaviour of the network. There is no obvious way to know what the desired output of the hidden layer should be. In other words, the desired output of the hidden layer is determined by the layer itself.

Can a neural network include more than two hidden layers?

Commercial ANNs incorporate three and sometimes four layers, including one or two hidden layers. Each layer can contain from 10 to 1000 neurons. Experimental neural networks may have five or even six layers, including three or four hidden layers, and utilise millions of neurons, but most practical applications use only three layers, because each additional layer increases the computational burden exponentially.

How do multilayer neural networks learn?

More than a hundred different learning algorithms are available, but the most popular method is back-propagation. This method was first proposed in 1969 (Bryson and Ho, 1969), but was ignored because of its demanding computations. Only in the mid-1980s was the back-propagation learning algorithm rediscovered.

Learning in a multilayer network proceeds the same way as for a perceptron. A training set of input patterns is presented to the network. The network computes its output pattern, and if there is an error – or in other words a difference between actual and desired output patterns – the weights are adjusted to reduce this error.

In a perceptron, there is only one weight for each input and only one output. But in the multilayer network, there are many weights, each of which contributes to more than one output.

How can we assess the blame for an error and divide it among the contributing weights?

In a back-propagation neural network, the learning algorithm has two phases. First, a training input pattern is presented to the network input layer. The network then propagates the input pattern from layer to layer until the output pattern is generated by the output layer. If this pattern is different from the desired output, an error is calculated and then propagated backwards through the network from the output layer to the input layer. The weights are modified as the error is propagated.

As with any other neural network, a back-propagation one is determined by the connections between neurons (the network's architecture), the activation function used by the neurons, and the learning algorithm (or the learning law) that specifies the procedure for adjusting weights.

Typically, a back-propagation network is a multilayer network that has three or four layers. The layers are **fully connected**, that is, every neuron in each layer is connected to every other neuron in the adjacent forward layer.

A neuron determines its output in a manner similar to Rosenblatt's perceptron. First, it computes the net weighted input as before:

$$X = \sum_{i=1}^{n} x_i w_i - \theta,$$

where n is the number of inputs, and θ is the threshold applied to the neuron.

Next, this input value is passed through the activation function. However, unlike a percepron, neurons in the back-propagation network use a sigmoid activation function:

$$Y^{sigmoid} = \frac{1}{1 + e^{-X}} \tag{6.9}$$

The derivative of this function is easy to compute. It also guarantees that the neuron output is bounded between 0 and 1.

What about the learning law used in the back-propagation networks?

To derive the back-propagation learning law, let us consider the three-layer network shown in Figure 6.9. The indices i, j and k here refer to neurons in the input, hidden and output layers, respectively.

Input signals, $x_1, x_2, \ldots, x_n$, are propagated through the network from left to right, and error signals, $e_1, e_2, \ldots, e_l$, from right to left. The symbol w_{ij} denotes the weight for the connection between neuron i in the input layer and neuron j in the hidden layer, and the symbol w_{jk} the weight between neuron j in the hidden layer and neuron k in the output layer.

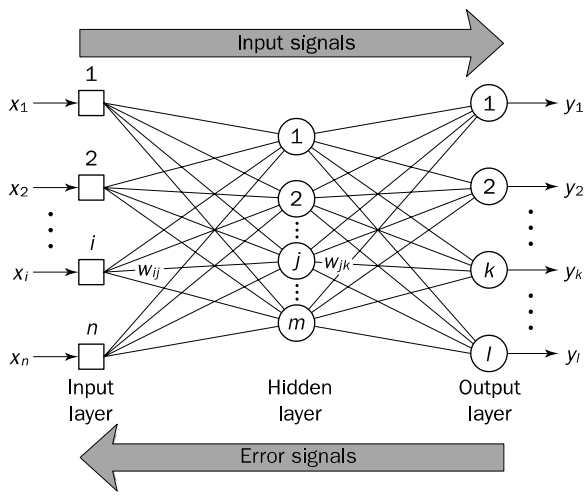


Figure 6.9 Three-layer back-propagation neural network

To propagate error signals, we start at the output layer and work backward to the hidden layer. The error signal at the output of neuron k at iteration p is defined by

$$e_k(p) = y_{d,k}(p) - y_k(p), \tag{6.10}$$

where $y_{d,k}(p)$ is the desired output of neuron k at iteration p.

Neuron k, which is located in the output layer, is supplied with a desired output of its own. Hence, we may use a straightforward procedure to update weight w_{jk}. In fact, the rule for updating weights at the output layer is similar to the perceptron learning rule of Eq. (6.7):

$$w_{jk}(p+1) = w_{jk}(p) + \Delta w_{jk}(p), \tag{6.11}$$

where $\Delta w_{jk}(p)$ is the weight correction.

When we determined the weight correction for the perceptron, we used input signal x_i. But in the multilayer network, the inputs of neurons in the output layer are different from the inputs of neurons in the input layer.

As we cannot apply input signal x_i, what should we use instead?

We use the output of neuron j in the hidden layer, y_j, instead of input x_i. The weight correction in the multilayer network is computed by (Fu, 1994):

$$\Delta w_{jk}(p) = \alpha \times y_j(p) \times \delta_k(p), \tag{6.12}$$

where $\delta_k(p)$ is the error gradient at neuron k in the output layer at iteration p.

What is the error gradient?

The error gradient is determined as the derivative of the activation function multiplied by the error at the neuron output.

Thus, for neuron k in the output layer, we have

$$\delta_k(p) = \frac{\partial y_k(p)}{\partial X_k(p)} \times e_k(p), \tag{6.13}$$

where $y_k(p)$ is the output of neuron k at iteration p, and $X_k(p)$ is the net weighted input to neuron k at the same iteration.

For a sigmoid activation function, Eq. (6.13) can be represented as

$$\delta_k(p) = \frac{\partial \left\{ \dfrac{1}{1 + \exp[-X_k(p)]} \right\}}{\partial X_k(p)} \times e_k(p) = \frac{\exp[-X_k(p)]}{\{1 + \exp[-X_k(p)]\}^2} \times e_k(p)$$

Thus, we obtain:

$$\delta_k(p) = y_k(p) \times [1 - y_k(p)] \times e_k(p), \tag{6.14}$$

where

$$y_k(p) = \frac{1}{1 + \exp[-X_k(p)]}.$$

How can we determine the weight correction for a neuron in the hidden layer?

To calculate the weight correction for the hidden layer, we can apply the same equation as for the output layer:

$$\Delta w_{ij}(p) = \alpha \times x_i(p) \times \delta_j(p), \tag{6.15}$$

where $\delta_j(p)$ represents the error gradient at neuron j in the hidden layer:

$$\delta_j(p) = y_j(p) \times [1 - y_j(p)] \times \sum_{k=1}^{l} \delta_k(p) w_{jk}(p),$$

where l is the number of neurons in the output layer;

$$y_j(p) = \frac{1}{1 + e^{-X_j(p)}};$$

$$X_j(p) = \sum_{i=1}^{n} x_i(p) \times w_{ij}(p) - \theta_j;$$

and n is the number of neurons in the input layer.

Now we can derive the back-propagation training algorithm.

Step 1: ***Initialisation***

Set all the weights and threshold levels of the network to random numbers uniformly distributed inside a small range (Haykin, 1999):

$$\left(-\frac{2.4}{F_i}, +\frac{2.4}{F_i}\right),$$

where F_i is the total number of inputs of neuron i in the network. The weight initialisation is done on a neuron-by-neuron basis.

Step 2: ***Activation***

Activate the back-propagation neural network by applying inputs $x_1(p), x_2(p), \ldots, x_n(p)$ and desired outputs $y_{d,1}(p), y_{d,2}(p), \ldots, y_{d,n}(p)$.

(a) Calculate the actual outputs of the neurons in the hidden layer:

$$y_j(p) = sigmoid\left[\sum_{i=1}^{n} x_i(p) \times w_{ij}(p) - \theta_j\right],$$

where n is the number of inputs of neuron j in the hidden layer, and *sigmoid* is the sigmoid activation function.

(b) Calculate the actual outputs of the neurons in the output layer:

$$y_k(p) = sigmoid\left[\sum_{j=1}^{m} x_{jk}(p) \times w_{jk}(p) - \theta_k\right],$$

where m is the number of inputs of neuron k in the output layer.

Step 3: ***Weight training***

Update the weights in the back-propagation network propagating backward the errors associated with output neurons.

(a) Calculate the error gradient for the neurons in the output layer:

$$\delta_k(p) = y_k(p) \times [1 - y_k(p)] \times e_k(p)$$

where

$$e_k(p) = y_{d,k}(p) - y_k(p)$$

Calculate the weight corrections:

$$\Delta w_{jk}(p) = \alpha \times y_j(p) \times \delta_k(p)$$

Update the weights at the output neurons:

$$w_{jk}(p+1) = w_{jk}(p) + \Delta w_{jk}(p)$$

(b) Calculate the error gradient for the neurons in the hidden layer:

$$\delta_j(p) = y_j(p) \times [1 - y_j(p)] \times \sum_{k=1}^{l} \delta_k(p) \times w_{jk}(p)$$

Calculate the weight corrections:

$$\Delta w_{ij}(p) = \alpha \times x_i(p) \times \delta_j(p)$$

Update the weights at the hidden neurons:

$$w_{ij}(p+1) = w_{ij}(p) + \Delta w_{ij}(p)$$

Step 4: ***Iteration***

Increase iteration p by one, go back to Step 2 and repeat the process until the selected error criterion is satisfied.

As an example, we may consider the three-layer back-propagation network shown in Figure 6.10. Suppose that the network is required to perform logical operation Exclusive-OR. Recall that a single-layer perceptron could not do this operation. Now we will apply the three-layer net.

Neurons 1 and 2 in the input layer accept inputs x_1 and x_2, respectively, and redistribute these inputs to the neurons in the hidden layer without any processing:

$x_{13} = x_{14} = x_1$ and $x_{23} = x_{24} = x_2$.

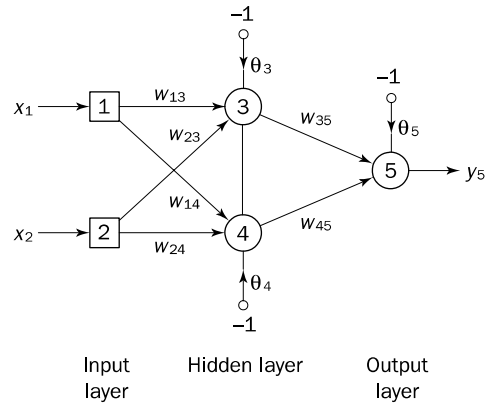


Figure 6.10 Three-layer network for solving the Exclusive-OR operation

The effect of the threshold applied to a neuron in the hidden or output layer is represented by its weight, θ, connected to a fixed input equal to -1.

The initial weights and threshold levels are set randomly as follows:

$w_{13} = 0.5$, $w_{14} = 0.9$, $w_{23} = 0.4$, $w_{24} = 1.0$, $w_{35} = -1.2$, $w_{45} = 1.1$,
$\theta_3 = 0.8$, $\theta_4 = -0.1$ and $\theta_5 = 0.3$.

Consider a training set where inputs x_1 and x_2 are equal to 1 and desired output $y_{d,5}$ is 0. The actual outputs of neurons 3 and 4 in the hidden layer are calculated as

$$y_3 = sigmoid\,(x_1 w_{13} + x_2 w_{23} - \theta_3) = 1/[1 + e^{-(1\times 0.5 + 1\times 0.4 - 1\times 0.8)}] = 0.5250$$

$$y_4 = sigmoid\,(x_1 w_{14} + x_2 w_{24} - \theta_4) = 1/[1 + e^{-(1\times 0.9 + 1\times 1.0 + 1\times 0.1)}] = 0.8808$$

Now the actual output of neuron 5 in the output layer is determined as

$$y_5 = sigmoid\,(y_3 w_{35} + y_4 w_{45} - \theta_5) = 1/[1 + e^{-(-0.5250\times 1.2 + 0.8808\times 1.1 - 1\times 0.3)}] = 0.5097$$

Thus, the following error is obtained:

$$e = y_{d,5} - y_5 = 0 - 0.5097 = -0.5097$$

The next step is weight training. To update the weights and threshold levels in our network, we propagate the error, e, from the output layer backward to the input layer.

First, we calculate the error gradient for neuron 5 in the output layer:

$$\delta_5 = y_5(1 - y_5)e = 0.5097 \times (1 - 0.5097) \times (-0.5097) = -0.1274$$

Then we determine the weight corrections assuming that the learning rate parameter, α, is equal to 0.1:

$$\Delta w_{35} = \alpha \times y_3 \times \delta_5 = 0.1 \times 0.5250 \times (-0.1274) = -0.0067$$

$$\Delta w_{45} = \alpha \times y_4 \times \delta_5 = 0.1 \times 0.8808 \times (-0.1274) = -0.0112$$

$$\Delta \theta_5 = \alpha \times (-1) \times \delta_5 = 0.1 \times (-1) \times (-0.1274) = 0.0127$$

Next we calculate the error gradients for neurons 3 and 4 in the hidden layer:

$$\delta_3 = y_3(1 - y_3) \times \delta_5 \times w_{35} = 0.5250 \times (1 - 0.5250) \times (-0.1274) \times (-1.2) = 0.0381$$

$$\delta_4 = y_4(1 - y_4) \times \delta_5 \times w_{45} = 0.8808 \times (1 - 0.8808) \times (-0.1274) \times 1.1 = -0.0147$$

We then determine the weight corrections:

$$\Delta w_{13} = \alpha \times x_1 \times \delta_3 = 0.1 \times 1 \times 0.0381 = 0.0038$$

$$\Delta w_{23} = \alpha \times x_2 \times \delta_3 = 0.1 \times 1 \times 0.0381 = 0.0038$$

$$\Delta \theta_3 = \alpha \times (-1) \times \delta_3 = 0.1 \times (-1) \times 0.0381 = -0.0038$$

$$\Delta w_{14} = \alpha \times x_1 \times \delta_4 = 0.1 \times 1 \times (-0.0147) = -0.0015$$

$$\Delta w_{24} = \alpha \times x_2 \times \delta_4 = 0.1 \times 1 \times (-0.0147) = -0.0015$$

$$\Delta \theta_4 = \alpha \times (-1) \times \delta_4 = 0.1 \times (-1) \times (-0.0147) = 0.0015$$

At last, we update all weights and threshold levels in our network:

$$w_{13} = w_{13} + \Delta w_{13} = 0.5 + 0.0038 = 0.5038$$

$$w_{14} = w_{14} + \Delta w_{14} = 0.9 - 0.0015 = 0.8985$$

$$w_{23} = w_{23} + \Delta w_{23} = 0.4 + 0.0038 = 0.4038$$

$$w_{24} = w_{24} + \Delta w_{24} = 1.0 - 0.0015 = 0.9985$$

$$w_{35} = w_{35} + \Delta w_{35} = -1.2 - 0.0067 = -1.2067$$

$$w_{45} = w_{45} + \Delta w_{45} = 1.1 - 0.0112 = 1.0888$$

$$\theta_3 = \theta_3 + \Delta \theta_3 = 0.8 - 0.0038 = 0.7962$$

$$\theta_4 = \theta_4 + \Delta \theta_4 = -0.1 + 0.0015 = -0.0985$$

$$\theta_5 = \theta_5 + \Delta \theta_5 = 0.3 + 0.0127 = 0.3127$$

The training process is repeated until the sum of squared errors is less than 0.001.

Why do we need to sum the squared errors?

The **sum of the squared errors** is a useful indicator of the network's performance. The back-propagation training algorithm attempts to minimise this criterion. When the value of the sum of squared errors in an entire pass through all

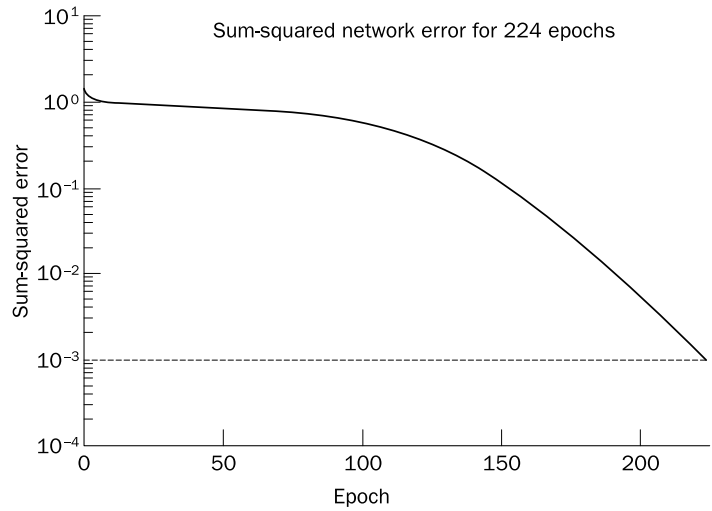


Figure 6.11 Learning curve for operation Exclusive-OR

training sets, or epoch, is **sufficiently small**, a network is considered to have **converged**. In our example, the sufficiently small sum of squared errors is defined as less than 0.001. Figure 6.11 represents a learning curve: the sum of squared errors plotted versus the number of epochs used in training. The learning curve shows how fast a network is learning.

It took 224 epochs or 896 iterations to train our network to perform the Exclusive-OR operation. The following set of final weights and threshold levels satisfied the chosen error criterion:

$w_{13} = 4.7621$, $w_{14} = 6.3917$, $w_{23} = 4.7618$, $w_{24} = 6.3917$, $w_{35} = -10.3788$, $w_{45} = 9.7691$, $\theta_3 = 7.3061$, $\theta_4 = 2.8441$ and $\theta_5 = 4.5589$.

The network has solved the problem! We may now test our network by presenting all training sets and calculating the network's output. The results are shown in Table 6.4.

Table 6.4 Final results of three-layer network learning: the logical operation Exclusive-OR

Inputs		Desired output	Actual output	Error	Sum of squared errors
x_1	x_2	y_d	y_5	e	
1	1	0	0.0155	−0.0155	0.0010
0	1	1	0.9849	0.0151	
1	0	1	0.9849	0.0151	
0	0	0	0.0175	−0.0175	

The initial weights and thresholds are set randomly. Does this mean that the same network may find different solutions?

The network obtains different weights and threshold values when it starts from different initial conditions. However, we will always solve the problem, although using a different number of iterations. For instance, when the network was trained again, we obtained the following solution:

$w_{13} = -6.3041$, $w_{14} = -5.7896$, $w_{23} = 6.2288$, $w_{24} = 6.0088$, $w_{35} = 9.6657$, $w_{45} = -9.4242$, $\theta_3 = 3.3858$, $\theta_4 = -2.8976$ and $\theta_5 = -4.4859$.

Can we now draw decision boundaries constructed by the multilayer network for operation Exclusive-OR?

It may be rather difficult to draw decision boundaries constructed by neurons with a sigmoid activation function. However, we can represent each neuron in the hidden and output layers by a McCulloch and Pitts model, using a sign function. The network in Figure 6.12 is also trained to perform the Exclusive-OR operation (Touretzky and Pomerlean, 1989; Haykin, 1999).

The positions of the decision boundaries constructed by neurons 3 and 4 in the hidden layer are shown in Figure 6.13(a) and (b), respectively. Neuron 5 in the output layer performs a linear combination of the decision boundaries formed by the two hidden neurons, as shown in Figure 6.13(c). The network in Figure 6.12 does indeed separate black and white dots and thus solves the Exclusive-OR problem.

Is back-propagation learning a good method for machine learning?

Although widely used, back-propagation learning is not immune from problems. For example, the back-propagation learning algorithm does not seem to function in the biological world (Stork, 1989). Biological neurons do not work backward to adjust the strengths of their interconnections, synapses, and thus back-propagation learning cannot be viewed as a process that emulates brain-like learning.

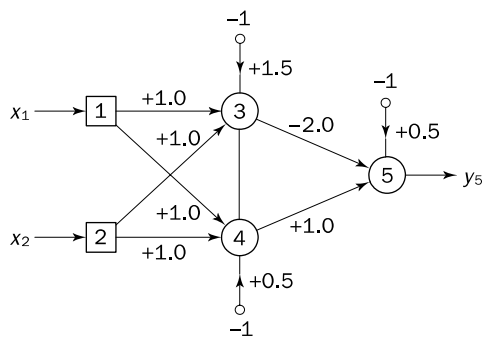


Figure 6.12 Network represented by McCulloch–Pitts model for solving the Exclusive-OR operation.

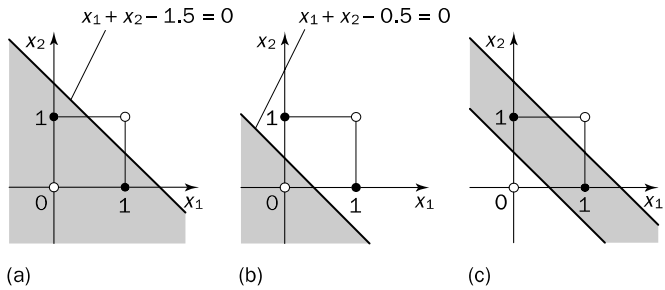


Figure 6.13 (a) Decision boundary constructed by hidden neuron 3 of the network in Figure 6.12; (b) decision boundary constructed by hidden neuron 4; (c) decision boundaries constructed by the complete three-layer network

Another apparent problem is that the calculations are extensive and, as a result, training is slow. In fact, a pure back-propagation algorithm is rarely used in practical applications.

There are several possible ways to improve the computational efficiency of the back-propagation algorithm (Caudill, 1991; Jacobs, 1988; Stubbs, 1990). Some of them are discussed below.

6.5 Accelerated learning in multilayer neural networks

A multilayer network, in general, learns much faster when the sigmoidal activation function is represented by a **hyperbolic tangent**,

$$Y^{tanh} = \frac{2a}{1 + e^{-bX}} - a, \tag{6.16}$$

where a and b are constants.

Suitable values for a and b are: $a = 1.716$ and $b = 0.667$ (Guyon, 1991).

We also can accelerate training by including a **momentum term** in the delta rule of Eq. (6.12) (Rumelhart *et al.*, 1986):

$$\Delta w_{jk}(p) = \beta \times \Delta w_{jk}(p-1) + \alpha \times y_j(p) \times \delta_k(p), \tag{6.17}$$

where β is a positive number $(0 \leqslant \beta < 1)$ called the momentum constant. Typically, the momentum constant is set to 0.95.

Equation (6.17) is called the **generalised delta rule**. In a special case, when $\beta = 0$, we obtain the delta rule of Eq. (6.12).

Why do we need the momentum constant?

According to the observations made in Watrous (1987) and Jacobs (1988), the inclusion of momentum in the back-propagation algorithm has a **stabilising effect** on training. In other words, the inclusion of momentum tends to

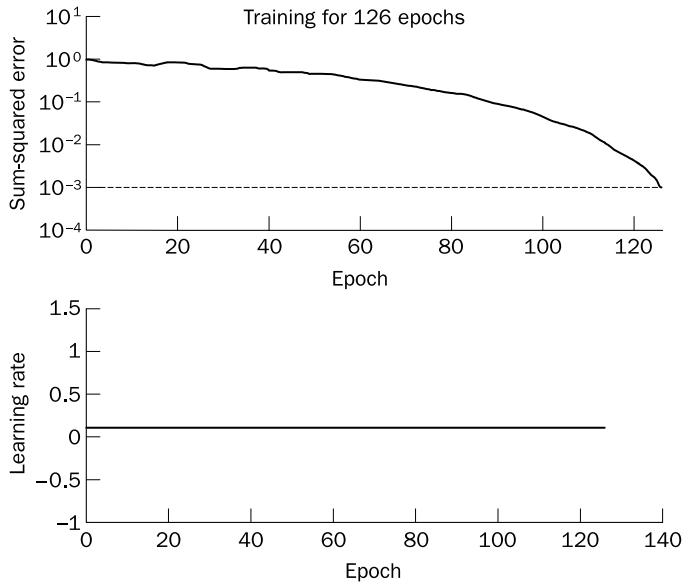


Figure 6.14 Learning with momentum

accelerate descent in the steady downhill direction, and to slow down the process when the learning surface exhibits peaks and valleys.

Figure 6.14 represents learning with momentum for operation Exclusive-OR. A comparison with a pure back-propagation algorithm shows that we reduced the number of epochs from 224 to 126.

In the delta and generalised delta rules, we use a constant and rather small value for the learning rate parameter, α. Can we increase this value to speed up training?

One of the most effective means to accelerate the convergence of back-propagation learning is to adjust the learning rate parameter during training. The small learning rate parameter, α, causes small changes to the weights in the network from one iteration to the next, and thus leads to the smooth learning curve. On the other hand, if the learning rate parameter, α, is made larger to speed up the training process, the resulting larger changes in the weights may cause instability and, as a result, the network may become oscillatory.

To accelerate the convergence and yet avoid the danger of instability, we can apply two heuristics (Jacobs, 1988):

- **Heuristic 1**. If the change of the sum of squared errors has the same algebraic sign for several consequent epochs, then the learning rate parameter, α, should be increased.
- **Heuristic 2**. If the algebraic sign of the change of the sum of squared errors alternates for several consequent epochs, then the learning rate parameter, α, should be decreased.

Adapting the learning rate requires some changes in the back-propagation algorithm. First, the network outputs and errors are calculated from the initial learning rate parameter. If the sum of squared errors at the current epoch exceeds the previous value by more than a predefined ratio (typically 1.04), the learning rate parameter is decreased (typically by multiplying by 0.7) and new weights and thresholds are calculated. However, if the error is less than the previous one, the learning rate is increased (typically by multiplying by 1.05).

Figure 6.15 represents an example of back-propagation training with adaptive learning rate. It demonstrates that adapting the learning rate can indeed decrease the number of iterations.

Learning rate adaptation can be used together with learning with momentum. Figure 6.16 shows the benefits of applying simultaneously both techniques.

The use of momentum and adaptive learning rate significantly improves the performance of a multilayer back-propagation neural network and minimises the chance that the network can become oscillatory.

Neural networks were designed on an analogy with the brain. The brain's memory, however, works by association. For example, we can recognise a familiar face even in an unfamiliar environment within 100–200 ms. We can also recall a complete sensory experience, including sounds and scenes, when we hear only a few bars of music. The brain routinely associates one thing with another.

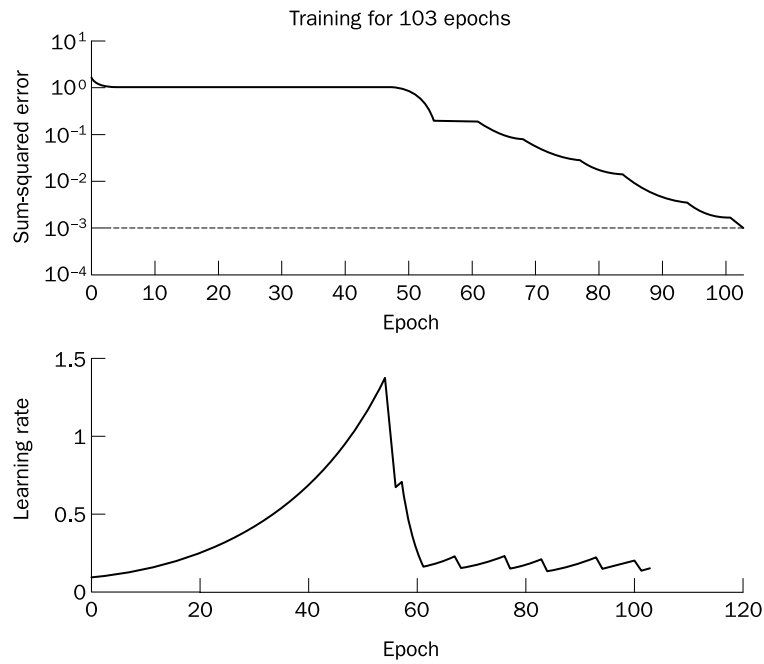


Figure 6.15 Learning with adaptive learning rate

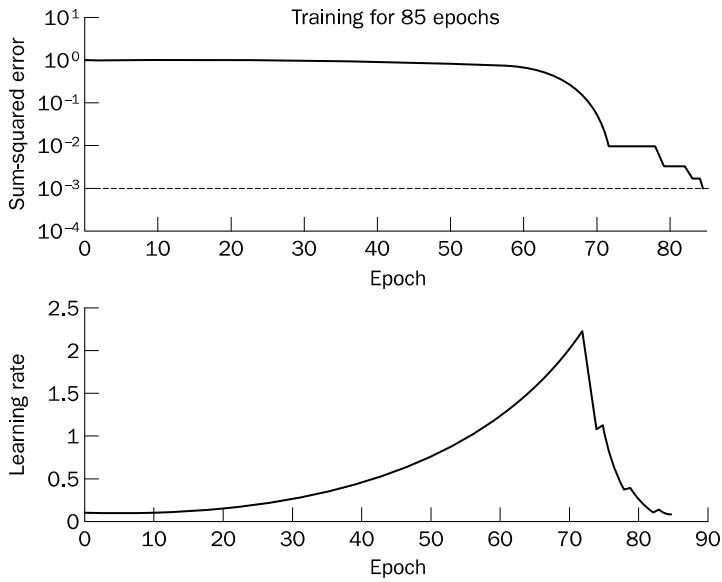


Figure 6.16 Learning with momentum and adaptive learning rate

Can a neural network simulate associative characteristics of the human memory?

Multilayer neural networks trained with the back-propagation algorithm are used for pattern recognition problems. But, as we noted, such networks are not intrinsically intelligent. To emulate the human memory's associative characteristics we need a different type of network: a **recurrent neural network**.

6.6 The Hopfield network

A recurrent neural network has feedback loops from its outputs to its inputs. The presence of such loops has a profound impact on the learning capability of the network.

How does the recurrent network learn?

After applying a new input, the network output is calculated and fed back to adjust the input. Then the output is calculated again, and the process is repeated until the output becomes constant.

Does the output always become constant?

Successive iterations do not always produce smaller and smaller output changes, but on the contrary may lead to chaotic behaviour. In such a case, the network output never becomes constant, and the network is said to be **unstable**.

The stability of recurrent networks intrigued several researchers in the 1960s and 1970s. However, none was able to predict which network would be stable,

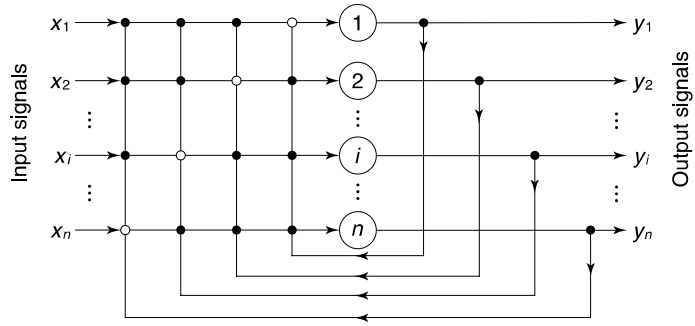


Figure 6.17 Single-layer *n*-neuron Hopfield network

and some researchers were pessimistic about finding a solution at all. The problem was solved only in 1982, when John Hopfield formulated the physical principle of storing information in a dynamically stable network (Hopfield, 1982).

Figure 6.17 shows a single-layer Hopfield network consisting of n neurons. The output of each neuron is fed back to the inputs of all other neurons (there is no self-feedback in the Hopfield network).

The Hopfield network usually uses McCulloch and Pitts neurons with the **sign activation function** as its computing element.

How does this function work here?

It works in a similar way to the sign function represented in Figure 6.4. If the neuron's weighted input is less than zero, the output is -1; if the input is greater than zero, the output is $+1$. However, if the neuron's weighted input is exactly zero, its output remains unchanged – in other words, a neuron remains in its previous state, regardless of whether it is $+1$ or -1.

$$Y^{sign} = \begin{cases} +1, & \text{if } X > 0 \\ -1, & \text{if } X < 0 \\ Y, & \text{if } X = 0 \end{cases} \tag{6.18}$$

The sign activation function may be replaced with a **saturated linear function**, which acts as a pure linear function within the region $[-1, 1]$ and as a sign function outside this region. The saturated linear function is shown in Figure 6.18.

The current state of the network is determined by the current outputs of all neurons, $y_1, y_2, \ldots, y_n$. Thus, for a single-layer n-neuron network, the state can be defined by the **state vector** as

$$\mathbf{Y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} \tag{6.19}$$

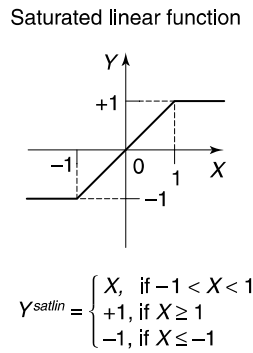


Figure 6.18 The saturated linear activation function

In the Hopfield network, synaptic weights between neurons are usually represented in matrix form as follows:

$$\mathbf{W} = \sum_{m=1}^{M} \mathbf{Y}_m \mathbf{Y}_m^T - M\mathbf{I}, \tag{6.20}$$

where M is the number of states to be memorised by the network, $\mathbf{Y}_m$ is the n-dimensional binary vector, $\mathbf{I}$ is $n \times n$ identity matrix, and superscript T denotes a matrix transposition.

An operation of the Hopfield network can be represented geometrically. Figure 6.19 shows a three-neuron network represented as a cube in the three-dimensional space. In general, a network with n neurons has 2^n possible states and is associated with an n-dimensional hypercube. In Figure 6.19, each state is represented by a vertex. When a new input vector is applied, the network moves from one state-vertex to another until it becomes stable.

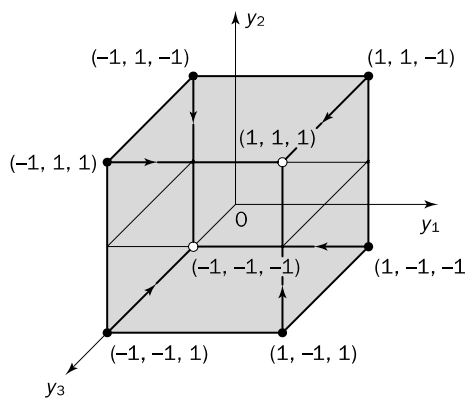


Figure 6.19 Cube representation of the possible states for the three-neuron Hopfield network

What determines a stable state-vertex?

The stable state-vertex is determined by the weight matrix **W**, the current input vector **X**, and the threshold matrix θ. If the input vector is partially incorrect or incomplete, the initial state will converge into the stable state-vertex after a few iterations.

Suppose, for instance, that our network is required to memorise two opposite states, $(1, 1, 1)$ and $(-1, -1, -1)$. Thus,

$$\mathbf{Y}_1 = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \text{ and } \mathbf{Y}_2 = \begin{bmatrix} -1 \\ -1 \\ -1 \end{bmatrix},$$

where $\mathbf{Y}_1$ and $\mathbf{Y}_2$ are the three-dimensional vectors.

We also can represent these vectors in the row, or *transposed*, form

$$\mathbf{Y}_1^T = [1 \quad 1 \quad 1] \text{ and } \mathbf{Y}_2^T = [-1 \quad -1 \quad -1]$$

The 3×3 identity matrix **I** is

$$\mathbf{I} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Thus, we can now determine the weight matrix as follows:

$$\mathbf{W} = \mathbf{Y}_1\mathbf{Y}_1^T + \mathbf{Y}_2\mathbf{Y}_2^T - 2\mathbf{I}$$

or

$$\mathbf{W} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} [1 \quad 1 \quad 1] + \begin{bmatrix} -1 \\ -1 \\ -1 \end{bmatrix} [-1 \quad -1 \quad -1] - 2 \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 2 & 2 \\ 2 & 0 & 2 \\ 2 & 2 & 0 \end{bmatrix}$$

Next, the network is tested by the sequence of input vectors, $\mathbf{X}_1$ and $\mathbf{X}_2$, which are equal to the output (or target) vectors $\mathbf{Y}_1$ and $\mathbf{Y}_2$, respectively. We want to see whether our network is capable of recognising familiar patterns.

How is the Hopfield network tested?

First, we activate it by applying the input vector **X**. Then, we calculate the actual output vector **Y**, and finally, we compare the result with the initial input vector **X**.

$$\mathbf{Y}_m = sign\,(\mathbf{W}\,\mathbf{X}_m - \boldsymbol{\theta}), \qquad m = 1, 2, \ldots, M \tag{6.21}$$

where $\boldsymbol{\theta}$ is the threshold matrix.

In our example, we may assume all thresholds to be zero. Thus,

$$\mathbf{Y}_1 = sign\left\{ \begin{bmatrix} 0 & 2 & 2 \\ 2 & 0 & 2 \\ 2 & 2 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \right\} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

and

$$\mathbf{Y}_2 = sign\left\{ \begin{bmatrix} 0 & 2 & 2 \\ 2 & 0 & 2 \\ 2 & 2 & 0 \end{bmatrix} \begin{bmatrix} -1 \\ -1 \\ -1 \end{bmatrix} - \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \right\} = \begin{bmatrix} -1 \\ -1 \\ -1 \end{bmatrix}$$

As we see, $\mathbf{Y}_1 = \mathbf{X}_1$ and $\mathbf{Y}_2 = \mathbf{X}_2$. Thus, both states, $(1, 1, 1)$ and $(-1, -1, -1)$, are said to be *stable*.

How about other states?

With three neurons in the network, there are eight possible states. The remaining six states are all unstable. However, stable states (also called **fundamental memories**) are capable of attracting states that are close to them. As shown in Table 6.5, the fundamental memory $(1, 1, 1)$ attracts unstable states $(-1, 1, 1)$, $(1, -1, 1)$ and $(1, 1, -1)$. Each of these unstable states represents a single error, compared to the fundamental memory $(1, 1, 1)$. On the other hand, the

Table 6.5 Operation of the three-neuron Hopfield network

Possible state	Iteration	Inputs			Outputs			Fundamental memory
		x_1	x_2	x_3	y_1	y_2	y_3	
1 1 1	0	1	1	1	1	1	1	1 1 1
−1 1 1	0	−1	1	1	1	1	1	
	1	1	1	1	1	1	1	1 1 1
1 −1 1	0	1	−1	1	1	1	1	
	1	1	1	1	1	1	1	1 1 1
1 1 −1	0	1	1	−1	1	1	1	
	1	1	1	1	1	1	1	1 1 1
−1 −1 −1	0	−1	−1	−1	−1	−1	−1	−1 −1 −1
−1 −1 1	0	−1	−1	1	−1	−1	−1	
	1	−1	−1	−1	−1	−1	−1	−1 −1 −1
−1 1 −1	0	−1	1	−1	−1	−1	−1	
	1	−1	−1	−1	−1	−1	−1	−1 −1 −1
1 −1 −1	0	1	−1	−1	−1	−1	−1	
	1	−1	−1	−1	−1	−1	−1	−1 −1 −1

fundamental memory $(-1,-1,-1)$ attracts unstable states $(-1,-1,1)$, $(-1,1,-1)$ and $(1,-1,-1)$. Here again, each of the unstable states represents a single error, compared to the fundamental memory. Thus, the Hopfield network can indeed act as an **error correction network**. Let us now summarise the Hopfield network training algorithm.

Step 1: ***Storage***

The n-neuron Hopfield network is required to store a set of M fundamental memories, $\mathbf{Y}_1, \mathbf{Y}_2, \ldots, \mathbf{Y}_M$. The synaptic weight from neuron i to neuron j is calculated as

$$w_{ij} = \begin{cases} \sum_{m=1}^{M} y_{m,i}\, y_{m,j}, & i \neq j \\ 0, & i = j \end{cases}, \tag{6.22}$$

where $y_{m,i}$ and $y_{m,j}$ are the ith and the jth elements of the fundamental memory $\mathbf{Y}_m$, respectively. In matrix form, the synaptic weights between neurons are represented as

$$\mathbf{W} = \sum_{m=1}^{M} \mathbf{Y}_m \mathbf{Y}_m^T - M\mathbf{I}$$

The Hopfield network can store a set of fundamental memories if the weight matrix is symmetrical, with zeros in its main diagonal (Cohen and Grossberg, 1983). That is,

$$\mathbf{W} = \begin{bmatrix} 0 & w_{12} & \cdots & w_{1i} & \cdots & w_{1n} \\ w_{21} & 0 & \cdots & w_{2i} & \cdots & w_{2n} \\ \vdots & \vdots & & \vdots & & \vdots \\ w_{i1} & w_{i2} & \cdots & 0 & \cdots & w_{in} \\ \vdots & \vdots & & \vdots & & \vdots \\ w_{n1} & w_{n2} & \cdots & w_{ni} & \cdots & 0 \end{bmatrix}, \tag{6.23}$$

where $w_{ij} = w_{ji}$.

Once the weights are calculated, they remain fixed.

Step 2: ***Testing***

We need to confirm that the Hopfield network is capable of recalling all fundamental memories. In other words, the network must recall any fundamental memory $\mathbf{Y}_m$ when presented with it as an input. That is,

$$x_{m,i} = y_{m,i}, \qquad i = 1, 2, \ldots, n; \qquad m = 1, 2, \ldots, M$$

$$y_{m,i} = sign\left(\sum_{j=1}^{n} w_{ij}\, x_{m,j} - \theta_i \right),$$

where $y_{m,i}$ is the ith element of the actual output vector $\mathbf{Y}_m$, and $x_{m,j}$ is the jth element of the input vector $\mathbf{X}_m$. In matrix form,

$$\mathbf{X}_m = \mathbf{Y}_m, \qquad m = 1, 2, \ldots, M$$
$$\mathbf{Y}_m = sign\,(\mathbf{W}\mathbf{X}_m - \boldsymbol{\theta})$$

If all fundamental memories are recalled perfectly we may proceed to the next step.

Step 3: ***Retrieval***

Present an unknown n-dimensional vector (probe), $\mathbf{X}$, to the network and retrieve a stable state. Typically, the probe represents a corrupted or incomplete version of the fundamental memory, that is,

$$\mathbf{X} \neq \mathbf{Y}_m, \qquad m = 1, 2, \ldots, M$$

(a) Initialise the retrieval algorithm of the Hopfield network by setting

$$x_j(0) = x_j \qquad j = 1, 2, \ldots, n$$

and calculate the initial state for each neuron

$$y_i(0) = sign\left(\sum_{j=1}^{n} w_{ij}\, x_j(0) - \theta_i\right), \qquad i = 1, 2, \ldots, n$$

where $x_j(0)$ is the jth element of the probe vector $\mathbf{X}$ at iteration $p = 0$, and $y_i(0)$ is the state of neuron i at iteration $p = 0$.

In matrix form, the state vector at iteration $p = 0$ is presented as

$$\mathbf{Y}(0) = sign\,[\mathbf{W}\mathbf{X}(0) - \boldsymbol{\theta}]$$

(b) Update the elements of the state vector, $\mathbf{Y}(p)$, according to the following rule:

$$y_i(p+1) = sign\left(\sum_{j=1}^{n} w_{ij}\, x_j(p) - \theta_i\right)$$

Neurons for updating are selected **asynchronously**, that is, randomly and one at a time.

Repeat the iteration until the state vector becomes unchanged, or in other words, a stable state is achieved. The condition for stability can be defined as:

$$y_i(p+1) = sign\left(\sum_{j=1}^{n} w_{ij}\, y_j(p) - \theta_i\right), \qquad i = 1, 2, \ldots, n \tag{6.24}$$

or, in matrix form,

$$\mathbf{Y}(p+1) = sign\,[\mathbf{W}\mathbf{Y}(p) - \boldsymbol{\theta}] \tag{6.25}$$

The Hopfield network will always converge to a stable state if the retrieval is done asynchronously (Haykin, 1999). However, this stable state does not necessarily represent one of the fundamental memories, and if it is a fundamental memory it is not necessarily the closest one.

Suppose, for example, we wish to store three fundamental memories in the five-neuron Hopfield network:

$$\mathbf{X}_1 = (+1, +1, +1, +1, +1)$$
$$\mathbf{X}_2 = (+1, -1, +1, -1, +1)$$
$$\mathbf{X}_3 = (-1, +1, -1, +1, -1)$$

The weight matrix is constructed from Eq. (6.20),

$$\mathbf{W} = \begin{bmatrix} 0 & -1 & 3 & -1 & 3 \\ -1 & 0 & -1 & 3 & -1 \\ 3 & -1 & 0 & -1 & 3 \\ -1 & 3 & -1 & 0 & -1 \\ 3 & -1 & 3 & -1 & 0 \end{bmatrix}$$

Assume now that the probe vector is represented by

$$\mathbf{X} = (+1, +1, -1, +1, +1)$$

If we compare this probe with the fundamental memory $\mathbf{X}_1$, we find that these two vectors differ only in a single bit. Thus, we may expect that the probe $\mathbf{X}$ will converge to the fundamental memory $\mathbf{X}_1$. However, when we apply the Hopfield network training algorithm described above, we obtain a different result. The pattern produced by the network recalls the memory $\mathbf{X}_3$, a false memory.

This example reveals one of the problems inherent to the Hopfield network.

Another problem is the **storage capacity**, or the largest number of fundamental memories that can be stored and retrieved correctly. Hopfield showed experimentally (Hopfield, 1982) that the maximum number of fundamental memories M_{max} that can be stored in the n-neuron recurrent network is limited by

$$M_{max} = 0.15n \tag{6.26}$$

We also may define the storage capacity of a Hopfield network on the basis that **most** of the fundamental memories are to be retrieved perfectly (Amit, 1989):

$$M_{max} = \frac{n}{2 \ln n} \tag{6.27}$$

What if we want all the fundamental memories to be retrieved perfectly?
It can be shown that to retrieve **all** the fundamental memories perfectly, their number must be halved (Amit, 1989):

$$M_{max} = \frac{n}{4 \ln n} \tag{6.28}$$

As we can see now, the storage capacity of a Hopfield network has to be kept rather small for the fundamental memories to be retrievable. This is a major limitation of the Hopfield network.

Strictly speaking, a Hopfield network represents an **auto-associative** type of memory. In other words, a Hopfield network can retrieve a corrupted or incomplete memory but cannot associate it with another different memory.

In contrast, human memory is essentially associative. One thing may remind us of another, and that of another, and so on. We use a chain of mental associations to recover a lost memory. If we, for example, forget where we left an umbrella, we try to recall where we last had it, what we were doing, and who we were talking to. Thus, we attempt to establish a chain of associations, and thereby to restore a lost memory.

Why can't a Hopfield network do this job?
The Hopfield network is a single-layer network, and thus the output pattern appears on the same set of neurons to which the input pattern was applied. To associate one memory with another, we need a recurrent neural network capable of accepting an input pattern on one set of neurons and producing a related, but different, output pattern on another set of neurons. In fact, we need a two-layer recurrent network, the **bidirectional associative memory**.

6.7 Bidirectional associative memory

Bidirectional associative memory (BAM), first proposed by Bart Kosko, is a heteroassociative network (Kosko, 1987, 1988). It associates patterns from one set, set A, to patterns from another set, set B, and vice versa. Like a Hopfield network, the BAM can generalise and also produce correct outputs despite corrupted or incomplete inputs. The basic BAM architecture is shown in Figure 6.20. It consists of two fully connected layers: an input layer and an output layer.

How does the BAM work?
The input vector $\mathbf{X}(p)$ is applied to the transpose of weight matrix $\mathbf{W}^T$ to produce an output vector $\mathbf{Y}(p)$, as illustrated in Figure 6.20(a). Then, the output vector $\mathbf{Y}(p)$ is applied to the weight matrix $\mathbf{W}$ to produce a new input vector $\mathbf{X}(p+1)$, as in Figure 6.20(b). This process is repeated until input and output vectors become unchanged, or in other words, the BAM reaches a stable state.

The basic idea behind the BAM is to store pattern pairs so that when n-dimensional vector $\mathbf{X}$ from set A is presented as input, the BAM recalls

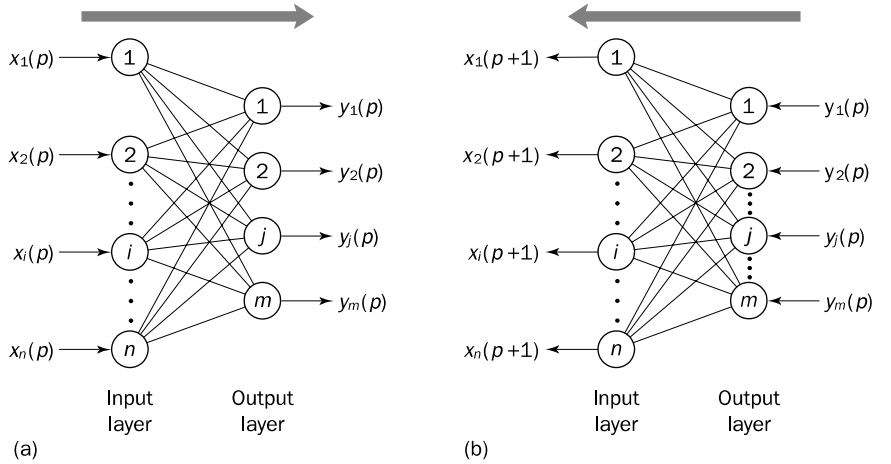


Figure 6.20 BAM operation: (a) forward direction; (b) backward direction

m-dimensional vector $\mathbf{Y}$ from set B, but when $\mathbf{Y}$ is presented as input, the BAM recalls $\mathbf{X}$.

To develop the BAM, we need to create a correlation matrix for each pattern pair we want to store. The correlation matrix is the matrix product of the input vector $\mathbf{X}$, and the transpose of the output vector $\mathbf{Y}^T$. The BAM weight matrix is the sum of all correlation matrices, that is,

$$\mathbf{W} = \sum_{m=1}^{M} \mathbf{X}_m \mathbf{Y}_m^T, \tag{6.29}$$

where M is the number of pattern pairs to be stored in the BAM.

Like a Hopfield network, the BAM usually uses McCulloch and Pitts neurons with the sign activation function.

The BAM training algorithm can be presented as follows.

Step 1: ***Storage***

The BAM is required to store M pairs of patterns. For example, we may wish to store four pairs:

$$\text{Set } A\text{: } \mathbf{X}_1 = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \quad \mathbf{X}_2 = \begin{bmatrix} -1 \\ -1 \\ -1 \\ -1 \\ -1 \\ -1 \end{bmatrix} \quad \mathbf{X}_3 = \begin{bmatrix} 1 \\ 1 \\ -1 \\ -1 \\ 1 \\ 1 \end{bmatrix} \quad \mathbf{X}_4 = \begin{bmatrix} -1 \\ -1 \\ 1 \\ 1 \\ -1 \\ -1 \end{bmatrix}$$

$$\text{Set } B\text{: } \mathbf{Y}_1 = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \quad \mathbf{Y}_2 = \begin{bmatrix} -1 \\ -1 \\ -1 \end{bmatrix} \quad \mathbf{Y}_3 = \begin{bmatrix} 1 \\ -1 \\ 1 \end{bmatrix} \quad \mathbf{Y}_4 = \begin{bmatrix} -1 \\ 1 \\ -1 \end{bmatrix}$$

In this case, the BAM input layer must have six neurons and the output layer three neurons.

The weight matrix is determined as

$$\mathbf{W} = \sum_{m=1}^{4} \mathbf{X}_m \mathbf{Y}_m^T$$

or

$$\mathbf{W} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} [1 \quad 1 \quad 1] + \begin{bmatrix} -1 \\ -1 \\ -1 \\ -1 \\ -1 \\ -1 \end{bmatrix} [-1 \quad -1 \quad -1] + \begin{bmatrix} 1 \\ 1 \\ -1 \\ -1 \\ 1 \\ 1 \end{bmatrix} [1 \quad -1 \quad 1]$$
$$+ \begin{bmatrix} -1 \\ -1 \\ 1 \\ 1 \\ -1 \\ -1 \end{bmatrix} [-1 \quad 1 \quad -1] = \begin{bmatrix} 4 & 0 & 4 \\ 4 & 0 & 4 \\ 0 & 4 & 0 \\ 0 & 4 & 0 \\ 4 & 0 & 4 \\ 4 & 0 & 4 \end{bmatrix}$$

Step 2: ***Testing***

The BAM should be able to receive any vector from set A and retrieve the associated vector from set B, and receive any vector from set B and retrieve the associated vector from set A. Thus, first we need to confirm that the BAM is able to recall $\mathbf{Y}_m$ when presented with $\mathbf{X}_m$. That is,

$$\mathbf{Y}_m = sign\,(\mathbf{W}^T \mathbf{X}_m), \qquad m = 1, 2, \ldots, M \tag{6.30}$$

For instance,

$$\mathbf{Y}_1 = sign\,(\mathbf{W}^T \mathbf{X}_1) = sign\left\{ \begin{bmatrix} 4 & 4 & 0 & 0 & 4 & 4 \\ 0 & 0 & 4 & 4 & 0 & 0 \\ 4 & 4 & 0 & 0 & 4 & 4 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \right\} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

Then, we confirm that the BAM recalls $\mathbf{X}_m$ when presented with $\mathbf{Y}_m$. That is,

$$\mathbf{X}_m = sign\,(\mathbf{W}\,\mathbf{Y}_m), \qquad m = 1, 2, \ldots, M \tag{6.31}$$

For instance,

$$\mathbf{X}_3 = sign\,(\mathbf{W}\,\mathbf{Y}_3) = sign\left\{\begin{bmatrix} 4 & 0 & 4 \\ 4 & 0 & 4 \\ 0 & 4 & 0 \\ 0 & 4 & 0 \\ 4 & 0 & 4 \\ 4 & 0 & 4 \end{bmatrix}\begin{bmatrix} 1 \\ -1 \\ 1 \end{bmatrix}\right\} = \begin{bmatrix} 1 \\ 1 \\ -1 \\ -1 \\ 1 \\ 1 \end{bmatrix}$$

In our example, all four pairs are recalled perfectly, and we can proceed to the next step.

Step 3: ***Retrieval***

Present an unknown vector (probe) $\mathbf{X}$ to the BAM and retrieve a stored association. The probe may present a corrupted or incomplete version of a pattern from set A (or from set B) stored in the BAM. That is,

$$\mathbf{X} \neq \mathbf{X}_m, \qquad m = 1, 2, \ldots, M$$

(a) Initialise the BAM retrieval algorithm by setting

$$\mathbf{X}(0) = \mathbf{X}, \qquad p = 0$$

and calculate the BAM output at iteration p

$$\mathbf{Y}(p) = sign\,[\mathbf{W}^T\,\mathbf{X}(p)]$$

(b) Update the input vector $\mathbf{X}(p)$:

$$\mathbf{X}(p+1) = sign\,[\mathbf{W}\,\mathbf{Y}(p)]$$

and repeat the iteration until equilibrium, when input and output vectors remain unchanged with further iterations. The input and output patterns will then represent an associated pair.

The BAM is unconditionally stable (Kosko, 1992). This means that any set of associations can be learned without risk of instability. This important quality arises from the BAM using the transpose relationship between weight matrices in forward and backward directions.

Let us now return to our example. Suppose we use vector $\mathbf{X}$ as a probe. It represents a single error compared with the pattern $\mathbf{X}_1$ from set A:

$$\mathbf{X} = (-1, +1, +1, +1, +1, +1)$$

This probe applied as the BAM input produces the output vector $\mathbf{Y}_1$ from set B. The vector $\mathbf{Y}_1$ is then used as input to retrieve the vector $\mathbf{X}_1$ from set A. Thus, the BAM is indeed capable of error correction.

There is also a close relationship between the BAM and the Hopfield network. If the BAM weight matrix is square and symmetrical, then $\mathbf{W} = \mathbf{W}^T$. In this case, input and output layers are of the same size, and the BAM can be reduced to the autoassociative Hopfield network. Thus, the Hopfield network can be considered as a BAM special case.

The constraints imposed on the storage capacity of the Hopfield network can also be extended to the BAM. In general, the maximum number of associations to be stored in the BAM should not exceed the number of neurons in the smaller layer. Another, even more serious problem, is incorrect convergence. The BAM may not always produce the closest association. In fact, a stable association may be only slightly related to the initial input vector.

The BAM still remains the subject of intensive research. However, despite all its current problems and limitations, the BAM promises to become one of the most useful artificial neural networks.

Can a neural network learn without a 'teacher'?

The main property of a neural network is an ability to learn from its environment, and to improve its performance through learning. So far we have considered **supervised** or **active learning** – learning with an external 'teacher' or a supervisor who presents a training set to the network. But another type of learning also exists: **unsupervised learning**.

In contrast to supervised learning, unsupervised or **self-organised learning** does not require an external teacher. During the training session, the neural network receives a number of different input patterns, discovers significant features in these patterns and learns how to classify input data into appropriate categories. Unsupervised learning tends to follow the neuro-biological organisation of the brain.

Unsupervised learning algorithms aim to learn rapidly. In fact, self-organising neural networks learn much faster than back-propagation networks, and thus can be used in real time.

6.8 Self-organising neural networks

Self-organising neural networks are effective in dealing with unexpected and changing conditions. In this section, we consider Hebbian and competitive learning, which are based on self-organising networks.

6.8.1 Hebbian learning

In 1949, neuropsychologist Donald Hebb proposed one of the key ideas in biological learning, commonly known as Hebb's Law (Hebb, 1949). Hebb's Law states that if neuron *i* is near enough to excite neuron *j* and repeatedly participates in its activation, the synaptic connection between these two neurons is strengthened and neuron *j* becomes more sensitive to stimuli from neuron *i*.

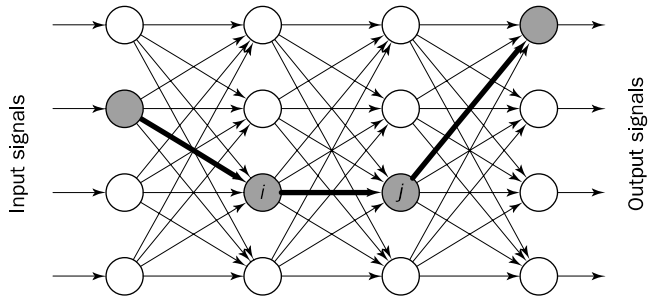


Figure 6.21 Hebbian learning in a neural network

We can represent Hebb's Law in the form of two rules as follows (Stent, 1973):

1. If two neurons on either side of a connection are activated synchronously, then the weight of that connection is increased.
2. If two neurons on either side of a connection are activated asynchronously, then the weight of that connection is decreased.

Hebb's Law provides the basis for learning without a teacher. Learning here is a local phenomenon occurring without feedback from the environment. Figure 6.21 shows Hebbian learning in a neural network.

Using Hebb's Law we can express the adjustment applied to the weight w_{ij} at iteration p in the following form:

$$\Delta w_{ij}(p) = F[y_j(p), x_i(p)], \tag{6.32}$$

where $F[y_j(p), x_i(p)]$ is a function of both postsynaptic and presynaptic activities.

As a special case, we can represent Hebb's Law as follows (Haykin, 1999):

$$\Delta w_{ij}(p) = \alpha\, y_j(p)\, x_i(p), \tag{6.33}$$

where α is the **learning rate** parameter.

This equation is referred to as the **activity product rule**. It shows how a change in the weight of the synaptic connection between a pair of neurons is related to a product of the incoming and outgoing signals.

Hebbian learning implies that weights can only increase. In other words, Hebb's Law allows the strength of a connection to increase, but it does not provide a means to decrease the strength. Thus, repeated application of the input signal may drive the weight w_{ij} into saturation. To resolve this problem, we might impose a limit on the growth of synaptic weights. It can be done by introducing a non-linear **forgetting factor** into Hebb's Law in Eq. (6.33) as follows (Kohonen, 1989):

$$\Delta w_{ij}(p) = \alpha\, y_j(p)\, x_i(p) - \phi\, y_j(p)\, w_{ij}(p) \tag{6.34}$$

where ϕ is the forgetting factor.

What does a forgetting factor mean?

Forgetting factor ϕ specifies the weight decay in a single learning cycle. It usually falls in the interval between 0 and 1. If the forgetting factor is 0, the neural network is capable only of strengthening its synaptic weights, and as a result, these weights grow towards infinity. On the other hand, if the forgetting factor is close to 1, the network remembers very little of what it learns. Therefore, a rather small forgetting factor should be chosen, typically between 0.01 and 0.1, to allow only a little 'forgetting' while limiting the weight growth.

Equation (6.34) may also be written in the form referred to as a **generalised activity product rule**

$$\Delta w_{ij}(p) = \phi\, y_j(p)[\lambda\, x_i(p) - w_{ij}(p)], \tag{6.35}$$

where $\lambda = \alpha/\phi$.

The generalised activity product rule implies that, if the presynaptic activity (input of neuron i) at iteration p, $x_i(p)$, is less than $w_{ij}(p)/\lambda$, then the modified synaptic weight at iteration $(p+1)$, $w_{ij}(p+1)$, will decrease by an amount proportional to the postsynaptic activity (output of neuron j) at iteration p, $y_j(p)$. On the other hand, if $x_i(p)$ is greater than $w_{ij}(p)/\lambda$, then the modified synaptic weight at iteration $(p+1)$, $w_{ij}(p+1)$, will increase also in proportion to the output of neuron j, $y_j(p)$. In other words, we can determine the **activity balance point** for modifying the synaptic weight as a variable equal to $w_{ij}(p)/\lambda$. This approach solves the problem of an infinite increase of the synaptic weights.

Let us now derive the generalised Hebbian learning algorithm.

Step 1: ***Initialisation***

Set initial synaptic weights and thresholds to small random values, say in an interval $[0, 1]$. Also assign small positive values to the learning rate parameter α and forgetting factor ϕ.

Step 2: ***Activation***

Compute the neuron output at iteration p

$$y_j(p) = \sum_{i=1}^{n} x_i(p)\, w_{ij}(p) - \theta_j,$$

where n is the number of neuron inputs, and θ_j is the threshold value of neuron j.

Step 3: ***Learning***

Update the weights in the network:

$$w_{ij}(p+1) = w_{ij}(p) + \Delta w_{ij}(p),$$

where $\Delta w_{ij}(p)$ is the weight correction at iteration p.

The weight correction is determined by the generalised activity product rule:

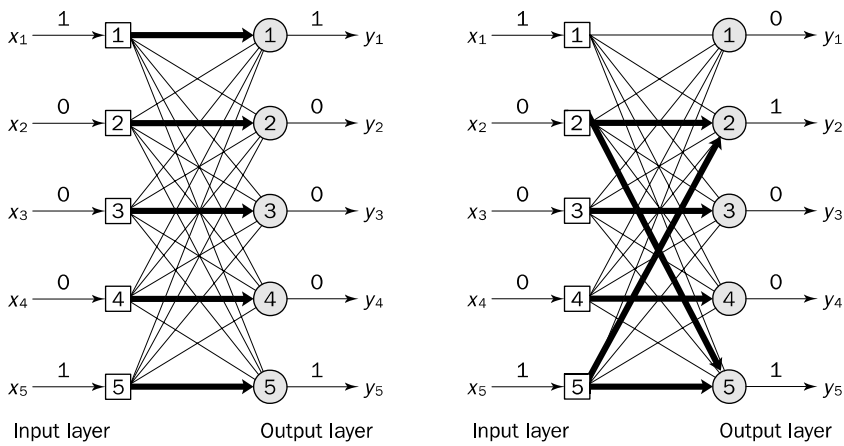
$$\Delta w_{ij}(p) = \phi\, y_j(p)[\lambda\, x_i(p) - w_{ij}(p)]$$

Step 4: ***Iteration***

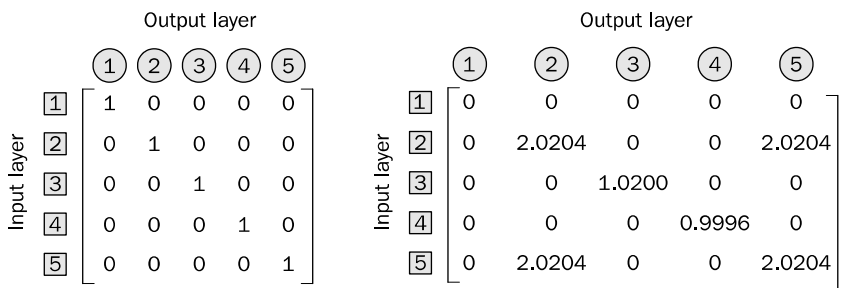
Increase iteration p by one, go back to Step 2 and continue until the synaptic weights reach their steady-state values.

To illustrate Hebbian learning, consider a fully connected feedforward network with a single layer of five computation neurons, as shown in Figure 6.22(a). Each neuron is represented by a McCulloch and Pitts model with the sign activation function. The network is trained with the generalised activity product rule on the following set of input vectors:

$$\mathbf{X}_1 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad \mathbf{X}_2 = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 1 \end{bmatrix} \quad \mathbf{X}_3 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \quad \mathbf{X}_4 = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \quad \mathbf{X}_5 = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$



(a)



Input layer \ Output layer	1	2	3	4	5
1	1	0	0	0	0
2	0	1	0	0	0
3	0	0	1	0	0
4	0	0	0	1	0
5	0	0	0	0	1

Input layer \ Output layer	1	2	3	4	5
1	0	0	0	0	0
2	0	2.0204	0	0	2.0204
3	0	0	1.0200	0	0
4	0	0	0	0.9996	0
5	0	2.0204	0	0	2.0204

(b)

Figure 6.22 Unsupervised Hebbian learning in a single-layer network: (a) initial and final states of the network; (b) initial and final weight matrices

Here, the input vector $\mathbf{X}_1$ is the null vector. As you may also notice, input signals x_4 (in the vector $\mathbf{X}_3$) and x_3 (in the vector $\mathbf{X}_4$) are the only unity components in the corresponding vectors, while unity signals x_2 and x_5 always come together, as seen in the vectors $\mathbf{X}_2$ and $\mathbf{X}_5$.

In our example, the initial weight matrix is represented by the 5×5 identity matrix $\mathbf{I}$. Thus, in the initial state, each of the neurons in the input layer is connected to the neuron in the same position in the output layer with a synaptic weight of 1, and to the other neurons with weights of 0. The thresholds are set to random numbers in the interval between 0 and 1. The learning rate parameter α and forgetting factor ϕ are taken as 0.1 and 0.02, respectively.

After training, as can be seen from Figure 6.22(b), the weight matrix becomes different from the initial identity matrix $\mathbf{I}$. The weights between neuron 2 in the input layer and neuron 5 in the output layer, and neuron 5 in the input layer and neuron 2 in the output layer have increased from 0 to 2.0204. Our network has learned new associations. At the same time, the weight between neuron 1 in the input layer and neuron 1 in the output layer has become 0. The network has forgotten this association.

Let us now test our network. A test input vector, or probe, is defined as

$$\mathbf{X} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

When this probe is presented to the network, we obtain

$$\mathbf{Y} = sign\,(\mathbf{W}\,\mathbf{X} - \boldsymbol{\theta})$$

$$\mathbf{Y} = sign\left\{ \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 2.0204 & 0 & 0 & 2.0204 \\ 0 & 0 & 1.0200 & 0 & 0 \\ 0 & 0 & 0 & 0.9996 & 0 \\ 0 & 2.0204 & 0 & 0 & 2.0204 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} - \begin{bmatrix} 0.4940 \\ 0.2661 \\ 0.0907 \\ 0.9478 \\ 0.0737 \end{bmatrix} \right\} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

Sure enough, the network has associated input x_5 with outputs y_2 and y_5 because inputs x_2 and x_5 were coupled during training. But the network cannot associate input x_1 with output y_1 any more because unity input x_1 did not appear during training and our network has lost the ability to recognise it.

Thus, a neural network really can learn to associate stimuli commonly presented together, and most important, the network can learn without a 'teacher'.

6.8.2 Competitive learning

Another popular type of unsupervised learning is **competitive learning**. In competitive learning, neurons compete among themselves to be activated. While in Hebbian learning, several output neurons can be activated simultaneously, in competitive learning only a single output neuron is active at any time. The output neuron that wins the 'competition' is called the **winner-takes-all** neuron.

The basic idea of competitive learning was introduced in the early 1970s (Grossberg, 1972; von der Malsburg, 1973; Fukushima, 1975). However, competitive learning did not attract much interest until the late 1980s, when Teuvo Kohonen introduced a special class of artificial neural networks called **self-organising feature maps** (Kohonen, 1989). These maps are based on competitive learning.

What is a self-organising feature map?

Our brain is dominated by the cerebral cortex, a very complex structure of billions of neurons and hundreds of billions of synapses. The cortex is neither uniform nor homogeneous. It includes areas, identified by the thickness of their layers and the types of neurons within them, that are responsible for different human activities (motor, visual, auditory, somatosensory, etc.), and thus associated with different sensory inputs. We can say that each sensory input is mapped into a corresponding area of the cerebral cortex; in other words, the cortex is a self-organising computational map in the human brain.

Can we model the self-organising map?

Kohonen formulated the **principle of topographic map formation** (Kohonen, 1990). This principle states that the spatial location of an output neuron in the topographic map corresponds to a particular feature of the input pattern. Kohonen also proposed the feature-mapping model shown in Figure 6.23 (Kohonen, 1982). This model captures the main features of self-organising maps in the brain and yet can be easily represented in a computer.

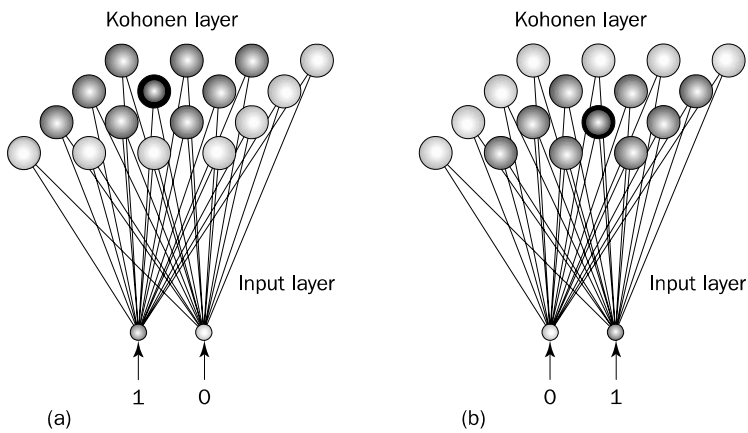


Figure 6.23 Feature-mapping Kohonen model

The Kohonen model provides a topological mapping, placing a fixed number of input patterns from the input layer into a higher-dimensional output or Kohonen layer. In Figure 6.23, the Kohonen layer consists of a two-dimensional lattice made up of 4-by-4 neurons, with each neuron having two inputs. The winning neuron is shown in black and its neighbours in grey. Here, the winner's neighbours are neurons in close physical proximity to the winner.

How close is 'close physical proximity'?

How close physical proximity is, is determined by the network designer. The winner's neighbourhood may include neurons within one, two or even three positions on either side. For example, Figure 6.23 depicts the winner's neighbourhood of size one. Generally, training in the Kohonen network begins with the winner's neighbourhood of a fairly large size. Then, as training proceeds, the neighbourhood size gradually decreases.

The Kohonen network consists of a single layer of computation neurons, but it has two different types of connections. There are **forward connections** from the neurons in the input layer to the neurons in the output layer, and also **lateral connections** between neurons in the output layer, as shown in Figure 6.24. The lateral connections are used to create a competition between neurons. The neuron with the largest activation level among all neurons in the output layer becomes the winner (the winner-takes-all neuron). This neuron is the only neuron that produces an output signal. The activity of all other neurons is suppressed in the competition.

When an input pattern is presented to the network, each neuron in the Kohonen layer receives a full copy of the input pattern, modified by its path through the weights of the synaptic connections between the input layer and the Kohonen layer. The lateral feedback connections produce excitatory or inhibitory effects, depending on the distance from the winning neuron. This is achieved by the use of a **Mexican hat function** which describes synaptic weights between neurons in the Kohonen layer.

What is the Mexican hat function?

The Mexican hat function shown in Figure 6.25 represents the relationship between the distance from the winner-takes-all neuron and the strength of the

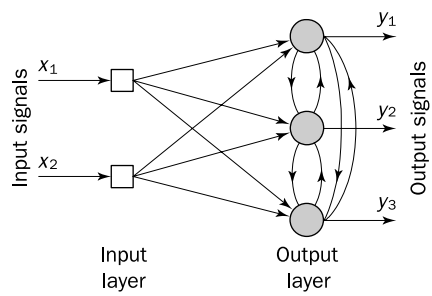


Figure 6.24 Architecture of the Kohonen network

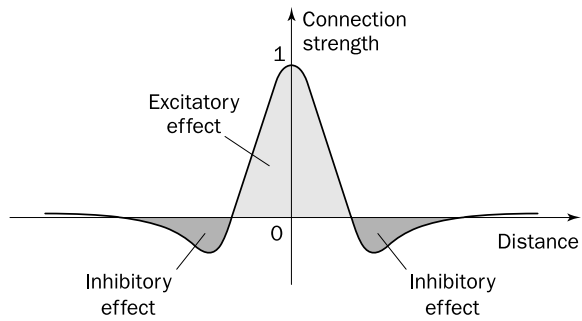


Figure 6.25 The Mexican hat function of lateral connection

connections within the Kohonen layer. According to this function, the near neighbourhood (a short-range lateral excitation area) has a strong excitatory effect, remote neighbourhood (an inhibitory penumbra) has a mild inhibitory effect and very remote neighbourhood (an area surrounding the inhibitory penumbra) has a weak excitatory effect, which is usually neglected.

In the Kohonen network, a neuron learns by shifting its weights from inactive connections to active ones. Only the winning neuron and its neighbourhood are allowed to learn. If a neuron does not respond to a given input pattern, then learning cannot occur in that particular neuron.

The output signal, y_j, of the winner-takes-all neuron j is set equal to one and the output signals of all the other neurons (the neurons that lose the competition) are set to zero.

The **standard competitive learning rule** (Haykin, 1999) defines the change Δw_{ij} applied to synaptic weight w_{ij} as

$$\Delta w_{ij} = \begin{cases} \alpha(x_i - w_{ij}), & \text{if neuron } j \text{ wins the competition} \\ 0, & \text{if neuron } j \text{ loses the competition} \end{cases} \tag{6.36}$$

where x_i is the input signal and α is the **learning rate parameter**. The learning rate parameter lies in the range between 0 and 1.

The overall effect of the competitive learning rule resides in moving the synaptic weight vector $\mathbf{W}_j$ of the winning neuron j towards the input pattern $\mathbf{X}$. The matching criterion is equivalent to the minimum **Euclidean distance** between vectors.

What is the Euclidean distance?

The Euclidean distance between a pair of n-by-1 vectors $\mathbf{X}$ and $\mathbf{W}_j$ is defined by

$$d = \|\mathbf{X} - \mathbf{W}_j\| = \left[\sum_{i=1}^{n}(x_i - w_{ij})^2\right]^{1/2}, \tag{6.37}$$

where x_i and w_{ij} are the ith elements of the vectors $\mathbf{X}$ and $\mathbf{W}_j$, respectively.

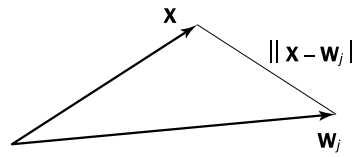


Figure 6.26 Euclidean distance as a measure of similarity between vectors **X** and $\mathbf{W}_j$

The similarity between the vectors **X** and $\mathbf{W}_j$ is determined as the reciprocal of the Euclidean distance d. In Figure 6.26, the Euclidean distance between the vectors **X** and $\mathbf{W}_j$ is presented as the length of the line joining the tips of those vectors. Figure 6.26 clearly demonstrates that the smaller the Euclidean distance is, the greater will be the similarity between the vectors **X** and $\mathbf{W}_j$.

To identify the winning neuron, $j_{\mathbf{X}}$, that best matches the input vector **X**, we may apply the following condition (Haykin, 1999):

$$j_{\mathbf{X}} = \min_j \|\mathbf{X} - \mathbf{W}_j\|, \qquad j = 1, 2, \ldots, m \tag{6.38}$$

where m is the number of neurons in the Kohonen layer.

Suppose, for instance, that the two-dimensional input vector **X** is presented to the three-neuron Kohonen network,

$$\mathbf{X} = \begin{bmatrix} 0.52 \\ 0.12 \end{bmatrix}$$

The initial weight vectors, $\mathbf{W}_j$, are given by

$$\mathbf{W}_1 = \begin{bmatrix} 0.27 \\ 0.81 \end{bmatrix} \quad \mathbf{W}_2 = \begin{bmatrix} 0.42 \\ 0.70 \end{bmatrix} \quad \mathbf{W}_3 = \begin{bmatrix} 0.43 \\ 0.21 \end{bmatrix}$$

We find the winning (best-matching) neuron $j_{\mathbf{X}}$ using the minimum-distance Euclidean criterion:

$$d_1 = \sqrt{(x_1 - w_{11})^2 + (x_2 - w_{21})^2} = \sqrt{(0.52 - 0.27)^2 + (0.12 - 0.81)^2} = 0.73$$

$$d_2 = \sqrt{(x_1 - w_{12})^2 + (x_2 - w_{22})^2} = \sqrt{(0.52 - 0.42)^2 + (0.12 - 0.70)^2} = 0.59$$

$$d_3 = \sqrt{(x_1 - w_{13})^2 + (x_2 - w_{23})^2} = \sqrt{(0.52 - 0.43)^2 + (0.12 - 0.21)^2} = 0.13$$

Thus, neuron 3 is the winner and its weight vector $\mathbf{W}_3$ is to be updated according to the competitive learning rule described in Eq. (6.36). Assuming that the learning rate parameter α is equal to 0.1, we obtain

$$\Delta w_{13} = \alpha(x_1 - w_{13}) = 0.1(0.52 - 0.43) = 0.01$$

$$\Delta w_{23} = \alpha(x_2 - w_{23}) = 0.1(0.12 - 0.21) = -0.01$$

The updated weight vector $\mathbf{W}_3$ at iteration $(p+1)$ is determined as:

$$\mathbf{W}_3(p+1) = \mathbf{W}_3(p) + \Delta\mathbf{W}_3(p) = \begin{bmatrix} 0.43 \\ 0.21 \end{bmatrix} + \begin{bmatrix} 0.01 \\ -0.01 \end{bmatrix} = \begin{bmatrix} 0.44 \\ 0.20 \end{bmatrix}$$

The weight vector $\mathbf{W}_3$ of the winning neuron 3 becomes closer to the input vector $\mathbf{X}$ with each iteration.

Let us now summarise the competitive learning algorithm as follows (Kohonen, 1989):

Step 1: ***Initialisation***

Set initial synaptic weights to small random values, say in an interval $[0, 1]$, and assign a small positive value to the learning rate parameter α.

Step 2: ***Activation and similarity matching***

Activate the Kohonen network by applying the input vector $\mathbf{X}$, and find the winner-takes-all (best matching) neuron $j_{\mathbf{X}}$ at iteration p, using the minimum-distance Euclidean criterion

$$j_{\mathbf{X}}(p) = \min_j \|\mathbf{X} - \mathbf{W}_j(p)\| = \left\{ \sum_{i=1}^{n} [x_i - w_{ij}(p)]^2 \right\}^{1/2}, \quad j = 1, 2, \ldots, m$$

where n is the number of neurons in the input layer, and m is the number of neurons in the output or Kohonen layer.

Step 3: ***Learning***

Update the synaptic weights

$$w_{ij}(p+1) = w_{ij}(p) + \Delta w_{ij}(p),$$

where $\Delta w_{ij}(p)$ is the weight correction at iteration p.

The weight correction is determined by the competitive learning rule

$$\Delta w_{ij}(p) = \begin{cases} \alpha[x_i - w_{ij}(p)], & j \in \Lambda_j(p) \\ 0, & j \notin \Lambda_j(p) \end{cases}, \tag{6.39}$$

where α is the learning rate parameter, and $\Lambda_j(p)$ is the neighbourhood function centred around the winner-takes-all neuron $j_{\mathbf{X}}$ at iteration p.

The neighbourhood function Λ_j usually has a constant amplitude. It implies that all the neurons located inside the topological neighbourhood are activated simultaneously, and the relationship among those neurons is independent of their distance from the winner-takes-all neuron $j_{\mathbf{X}}$. This simple form of a neighbourhood function is shown in Figure 6.27.

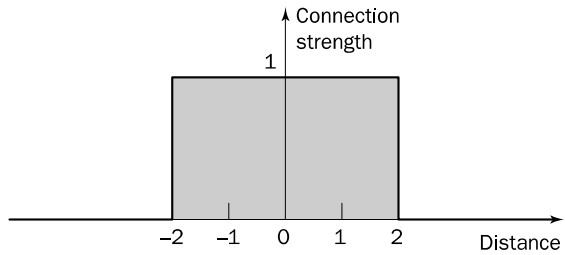


Figure 6.27 Rectangular neighbourhood function

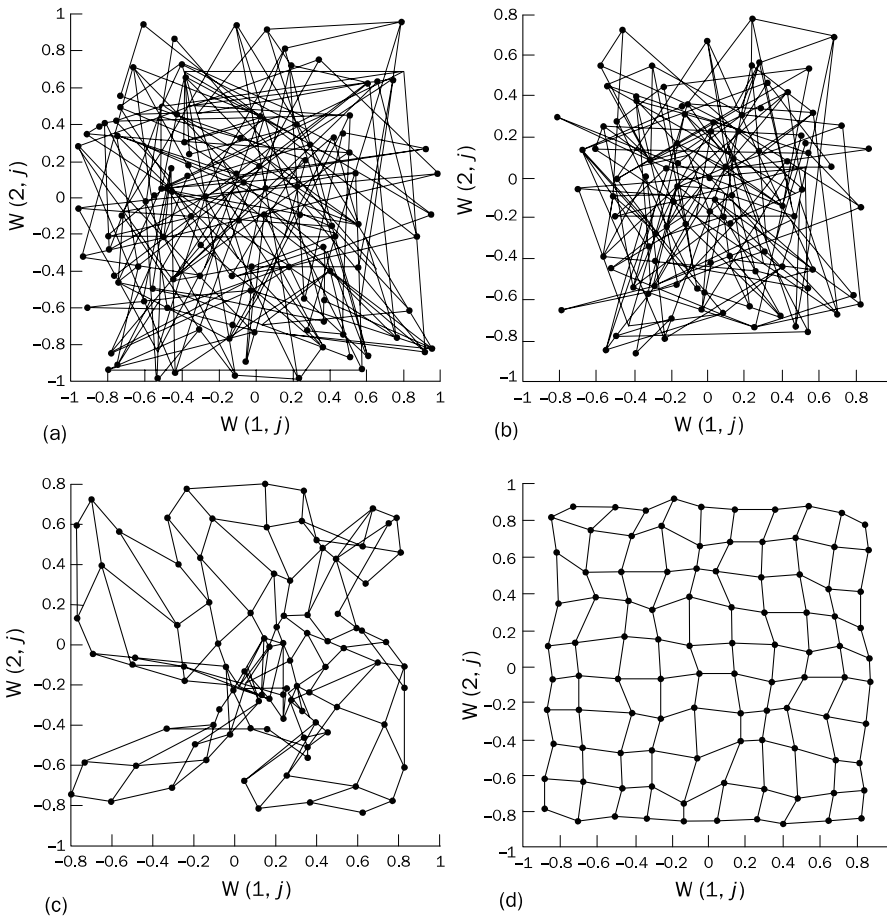


Figure 6.28 Competitive learning in the Kohonen network: (a) initial random weights; (b) network after 100 iterations; (c) network after 1000 iterations; (d) network after 10,000 iterations

The rectangular neighbourhood function Λ_j takes on a binary character. Thus, identifying the neuron outputs, we may write

$$y_j = \begin{cases} 1, & j \in \Lambda_j(p) \\ 0, & j \notin \Lambda_j(p) \end{cases} \tag{6.40}$$

Step 4: ***Iteration***

Increase iteration p by one, go back to Step 2 and continue until the minimum-distance Euclidean criterion is satisfied, or no noticeable changes occur in the feature map.

To illustrate competitive learning, consider the Kohonen network with 100 neurons arranged in the form of a two-dimensional lattice with 10 rows and 10 columns. The network is required to classify two-dimensional input vectors. In other words, each neuron in the network should respond only to the input vectors occurring in its region.

The network is trained with 1000 two-dimensional input vectors generated randomly in a square region in the interval between -1 and $+1$. Initial synaptic weights are also set to random values in the interval between -1 and $+1$, and the learning rate parameter α is equal to 0.1.

Figure 6.28 demonstrates different stages in the process of network learning. Each neuron is represented by a black dot at the location of its two weights, w_{1j} and w_{2j}. Figure 6.28(a) shows the initial synaptic weights randomly distributed in the square region. Figures 6.28(b), (c) and (d) present the weight vectors in the input space after 100, 1000 and 10,000 iterations, respectively.

The results shown in Figure 6.28 demonstrate the self-organisation of the Kohonen network that characterises unsupervised learning. At the end of the learning process, the neurons are mapped in the correct order and the map itself spreads out to fill the input space. Each neuron now is able to identify input vectors in its own input space.

To see how neurons respond, let us test our network by applying the following input vectors:

$$\mathbf{X}_1 = \begin{bmatrix} 0.2 \\ 0.9 \end{bmatrix} \quad \mathbf{X}_2 = \begin{bmatrix} 0.6 \\ -0.2 \end{bmatrix} \quad \mathbf{X}_3 = \begin{bmatrix} -0.7 \\ -0.8 \end{bmatrix}$$

As illustrated in Figure 6.29, neuron 6 responds to the input vector $\mathbf{X}_1$, neuron 69 responds to the input vector $\mathbf{X}_2$ and neuron 92 to the input vector $\mathbf{X}_3$. Thus, the feature map displayed in the input space in Figure 6.29 is topologically ordered and the spatial location of a neuron in the lattice corresponds to a particular feature of input patterns.

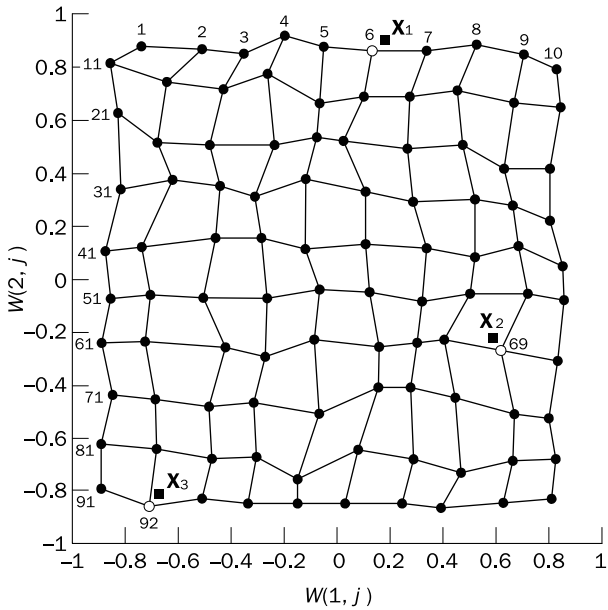


Figure 6.29 Topologically ordered feature map displayed in the input space

6.9 Summary

In this chapter, we introduced artificial neural networks and discussed the basic ideas behind machine learning. We presented the concept of a perceptron as a simple computing element and considered the perceptron learning rule. We explored multilayer neural networks and discussed how to improve the computational efficiency of the back-propagation learning algorithm. Then we introduced recurrent neural networks, considered the Hopfield network training algorithm and bidirectional associative memory (BAM). Finally, we presented self-organising neural networks and explored Hebbian and competitive learning.

The most important lessons learned in this chapter are:

- Machine learning involves adaptive mechanisms that enable computers to learn from experience, learn by example and learn by analogy. Learning capabilities can improve the performance of an intelligent system over time. One of the most popular approaches to machine learning is artificial neural networks.
- An artificial neural network consists of a number of very simple and highly interconnected processors, called neurons, which are analogous to the biological neurons in the brain. The neurons are connected by weighted links that pass signals from one neuron to another. Each link has a numerical weight associated with it. Weights are the basic means of long-term memory in ANNs. They express the strength, or importance, of each neuron input. A neural network 'learns' through repeated adjustments of these weights.

- In the 1940s, Warren McCulloch and Walter Pitts proposed a simple neuron model that is still the basis for most artificial neural networks. The neuron computes the weighted sum of the input signals and compares the result with a threshold value. If the net input is less than the threshold, the neuron output is −1. But if the net input is greater than or equal to the threshold, the neuron becomes activated and its output attains a value +1.
- Frank Rosenblatt suggested the simplest form of a neural network, which he called a perceptron. The operation of the perceptron is based on the McCulloch and Pitts neuron model. It consists of a single neuron with adjustable synaptic weights and a hard limiter. The perceptron learns its task by making small adjustments in the weights to reduce the difference between the actual and desired outputs. The initial weights are randomly assigned and then updated to obtain the output consistent with the training examples.
- A perceptron can learn only linearly separable functions and cannot make global generalisations on the basis of examples learned locally. The limitations of Rosenblatt's perceptron can be overcome by advanced forms of neural networks, such as multilayer perceptrons trained with the back-propagation algorithm.
- A multilayer perceptron is a feedforward neural network with an input layer of source neurons, at least one middle or hidden layer of computational neurons, and an output layer of computational neurons. The input layer accepts input signals from the outside world and redistributes these signals to all neurons in the hidden layer. The hidden layer detects the feature. The weights of the neurons in the hidden layer represent the features in the input patterns. The output layer establishes the output pattern of the entire network.
- Learning in a multilayer network proceeds in the same way as in a perceptron. The learning algorithm has two phases. First, a training input pattern is presented to the network input layer. The network propagates the input pattern from layer to layer until the output pattern is generated by the output layer. If it is different from the desired output, an error is calculated and then propagated backwards through the network from the output layer to the input layer. The weights are modified as the error is propagated.
- Although widely used, back-propagation learning is not without problems. Because the calculations are extensive and, as a result, training is slow, a pure back-propagation algorithm is rarely used in practical applications. There are several possible ways to improve computational efficiency. A multilayer network learns much faster when the sigmoidal activation function is represented by a hyperbolic tangent. The use of momentum and adaptive learning rate also significantly improves the performance of a multilayer back-propagation neural network.
- While multilayer back-propagation neural networks are used for pattern recognition problems, the associative memory of humans is emulated by a different type of network called recurrent: a recurrent network, which has feedback loops from its outputs to its inputs. John Hopfield formulated the

physical principle of storing information in a dynamically stable network, and also proposed a single-layer recurrent network using McCulloch and Pitts neurons with the sign activation function.

- The Hopfield network training algorithm has two basic phases: storage and retrieval. In the first phase, the network is required to store a set of states, or fundamental memories, determined by the current outputs of all neurons. This is achieved by calculating the network's weight matrix. Once the weights are calculated, they remain fixed. In the second phase, an unknown corrupted or incomplete version of the fundamental memory is presented to the network. The network output is calculated and fed back to adjust the input. This process is repeated until the output becomes constant. For the fundamental memories to be retrievable, the storage capacity of the Hopfield network has to be kept small.
- The Hopfield network represents an autoassociative type of memory. It can retrieve a corrupted or incomplete memory but cannot associate one memory with another. To overcome this limitation, Bart Kosko proposed the bidirectional associative memory (BAM). BAM is a heteroassociative network. It associates patterns from one set to patterns from another set and vice versa. As with a Hopfield network, the BAM can generalise and produce correct outputs despite corrupted or incomplete inputs. The basic BAM architecture consists of two fully connected layers – an input layer and an output layer.
- The idea behind the BAM is to store pattern pairs so that when n-dimensional vector $\mathbf{X}$ from set A is presented as input, the BAM recalls m-dimensional vector $\mathbf{Y}$ from set B, but when $\mathbf{Y}$ is presented as input, the BAM recalls $\mathbf{X}$. The constraints on the storage capacity of the Hopfield network can also be extended to the BAM. The number of associations to be stored in the BAM should not exceed the number of neurons in the smaller layer. Another problem is incorrect convergence, that is, the BAM may not always produce the closest association.
- In contrast to supervised learning, or learning with an external 'teacher' who presents a training set to the network, unsupervised or self-organised learning does not require a teacher. During a training session, the neural network receives a number of different input patterns, discovers significant features in these patterns and learns how to classify input.
- Hebb's Law, introduced by Donald Hebb in the late 1940s, states that if neuron i is near enough to excite neuron j and repeatedly participates in its activation, the synaptic connection between these two neurons is strengthened and neuron j becomes more sensitive to stimuli from neuron i. This law provides the basis for learning without a teacher. Learning here is a local phenomenon occurring without feedback from the environment.
- Another popular type of unsupervised learning is competitive learning. In competitive learning, neurons compete among themselves to become active. The output neuron that wins the 'competition' is called the winner-takes-all

neuron. Although competitive learning was proposed in the early 1970s, it was largely ignored until the late 1980s, when Teuvo Kohonen introduced a special class of artificial neural networks called self-organising feature maps. He also formulated the principle of topographic map formation which states that the spatial location of an output neuron in the topographic map corresponds to a particular feature of the input pattern.

- The Kohonen network consists of a single layer of computation neurons, but it has two different types of connections. There are forward connections from the neurons in the input layer to the neurons in the output layer, and lateral connections between neurons in the output layer. The lateral connections are used to create a competition between neurons. In the Kohonen network, a neuron learns by shifting its weights from inactive connections to active ones. Only the winning neuron and its neighbourhood are allowed to learn. If a neuron does not respond to a given input pattern, then learning does not occur in that neuron.

Questions for review

1 How does an artificial neural network model the brain? Describe two major classes of learning paradigms: supervised learning and unsupervised (self-organised) learning. What are the features that distinguish these two paradigms from each other?

2 What are the problems with using a perceptron as a biological model? How does the perceptron learn? Demonstrate perceptron learning of the binary logic function OR. Why can the perceptron learn only linearly separable functions?

3 What is a fully connected multilayer perceptron? Construct a multilayer perceptron with an input layer of six neurons, a hidden layer of four neurons and an output layer of two neurons. What is a hidden layer for, and what does it hide?

4 How does a multilayer neural network learn? Derive the back-propagation training algorithm. Demonstrate multilayer network learning of the binary logic function Exclusive-OR.

5 What are the main problems with the back-propagation learning algorithm? How can learning be accelerated in multilayer neural networks? Define the generalised delta rule.

6 What is a recurrent neural network? How does it learn? Construct a single six-neuron Hopfield network and explain its operation. What is a fundamental memory?

7 Derive the Hopfield network training algorithm. Demonstrate how to store three fundamental memories in the six-neuron Hopfield network.

8 The delta rule and Hebb's rule represent two different methods of learning in neural networks. Explain the differences between these two rules.

9 What is the difference between autoassociative and heteroassociative types of memory? What is the bidirectional associative memory (BAM)? How does the BAM work?

10 Derive the BAM training algorithm. What constraints are imposed on the storage capacity of the BAM? Compare the BAM storage capacity with the storage capacity of the Hopfield network.

11 What does Hebb's Law represent? Derive the activity product rule and the generalised activity product rule. What is the meaning of the forgetting factor? Derive the generalised Hebbian learning algorithm.

12 What is competitive learning? What are the differences between Hebbian and competitive learning paradigms? Describe the feature-mapping Kohonen model. Derive the competitive learning algorithm.

References

Amit, D.J. (1989). *Modelling Brain Functions: The World of Attractor Neural Networks*. Cambridge University Press, New York.

Bryson, A.E. and Ho, Y.C. (1969). *Applied Optimal Control*. Blaisdell, New York.

Caudill, M. (1991). Neural network training tips and techniques, *AI Expert*, January, 56–61.

Cohen, M.H. and Grossberg, S. (1983). Absolute stability of global pattern formation and parallel memory storage by competitive networks, *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-13, 815–826.

Grossberg, S. (1972). Neural expectation: cerebellar and retinal analogs of cells fired by learnable or unlearned pattern classes, *Kybernetik*, 10, 49–57.

Guyon, I.P. (1991). Applications of neural networks to character recognition, *International Journal of Pattern Recognition and Artificial Intelligence*, 5, 353–382.

Fu, L.M. (1994). *Neural Networks in Computer Intelligence*. McGraw-Hill Book, Inc., Singapore.

Fukushima, K. (1975). Cognition: a self-organizing multilayered neural network, *Biological Cybernetics*, 20, 121–136.

Haykin, S. (1999). *Neural Networks: A Comprehensive Foundation,* 2nd edn. Prentice Hall, Englewood Cliffs, NJ.

Hebb, D.O. (1949). *The Organisation of Behaviour: A Neuropsychological Theory*. John Wiley, New York.

Hopfield, J.J. (1982). Neural networks and physical systems with emergent collective computational abilities, *Proceedings of the National Academy of Sciences of the USA*, 79, 2554–2558.

Jacobs, R.A. (1988). Increased rates of convergence through learning rate adaptation, *Neural Networks*, 1, 295–307.

Kohonen, T. (1982). Self-organized formation of topologically correct feature maps, *Biological Cybernetics*, 43, 59–69.

Kohonen, T. (1989). *Self-Organization and Associative Memory*, 3rd edn. Springer-Verlag, Berlin, Heidelberg.

Kohonen, T. (1990). The self-organizing map, *Proceedings of the IEEE*, 78, 1464–1480.

Kosko, B. (1987). Adaptive bidirectional associative memories, *Applied Optics*, 26(23), 4947–4960.

Kosko, B. (1988). Bidirectional associative memories, *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-18, 49–60.