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 electrochemical 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.

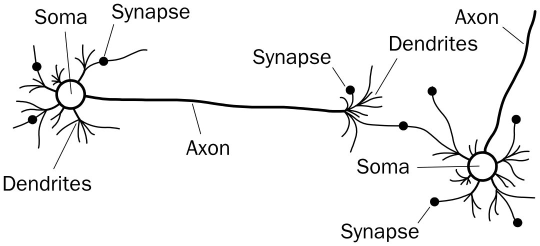


Figure 6.1 Biological neural network

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.

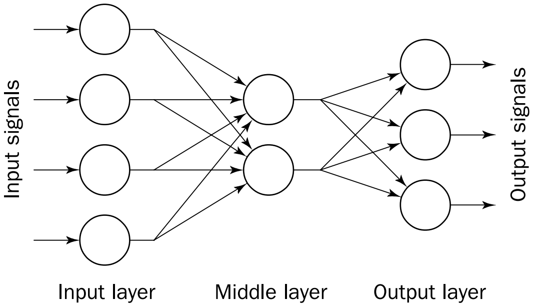


Figure 6.2 Architecture of a typical artificial neural network

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.

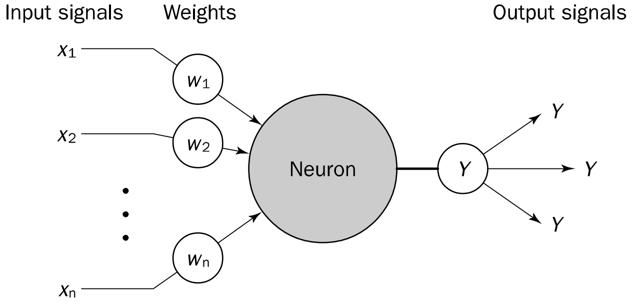


Figure 6.3 Diagram of a neuron

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.

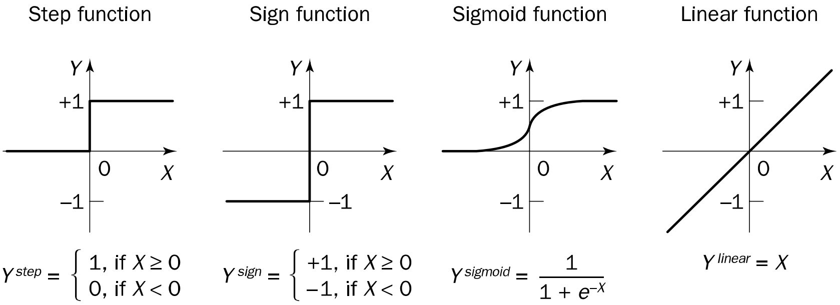


Figure 6.4 Activation functions of a neuron

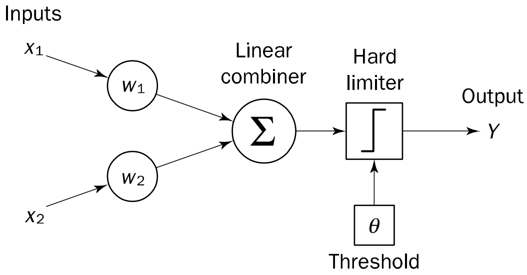


Figure 6.5 Single-layer two-input perceptron

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.

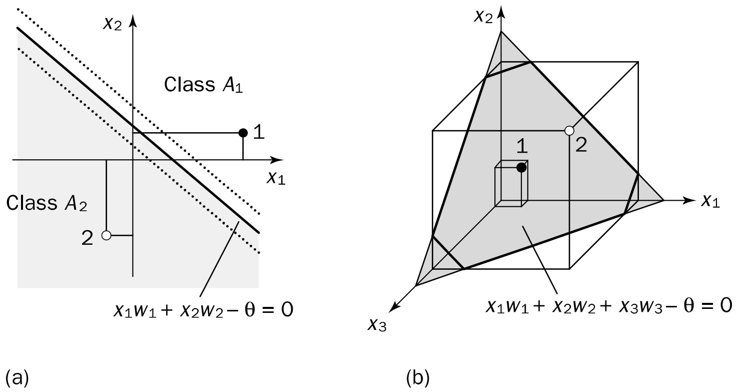

x_2
Class A_1
1
x_1
Class A_2
2
$x_1w_1 + x_2w_2 - \theta = 0$
(a)
x_2
1
2
x_1
x_3
$x_1w_1 + x_2w_2 + x_3w_3 - \theta = 0$
(b)

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

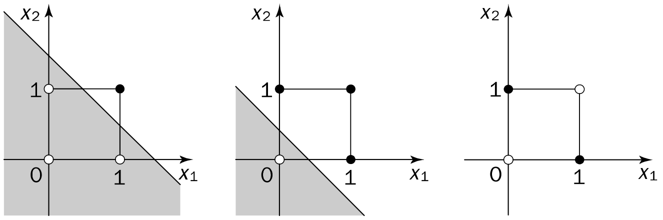
Table 6.3 Example of perceptron learning: the logical operation AND

	Inputs		Desired output	Initial weights		Actual output	Error	Final weights	
Epoch	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.

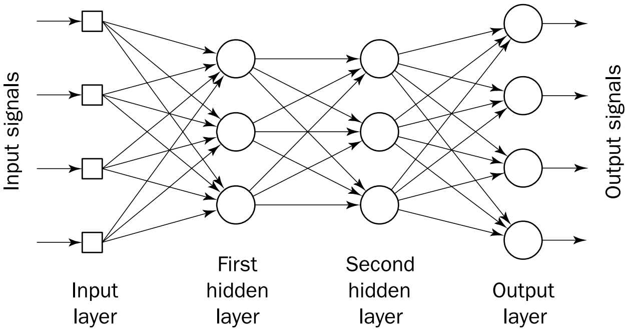


Figure 6.8 Multilayer perceptron with two hidden layers

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.

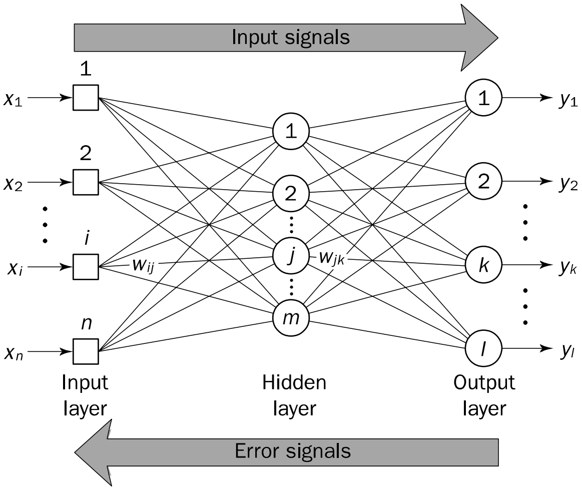


Figure 6.9 Three-layer back-propagation neural network

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:

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

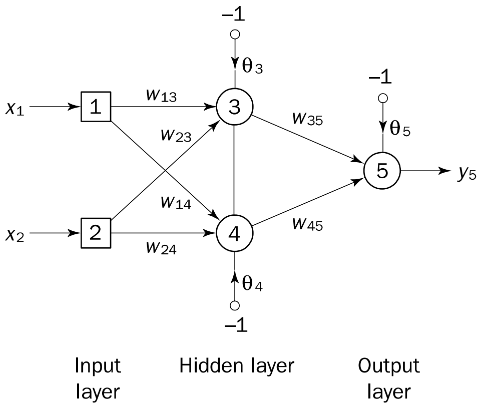


Figure 6.10 Three-layer network for solving the Exclusive-OR operation

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:

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

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

$$y_3 = sigmoid\,(x_1 w_{13} + x_2 w_{23} - \theta_3) = 1/[1 + e^{-(1\times0.5+1\times0.4-1\times0.8)}] = 0.5250$$
$$y_4 = sigmoid\,(x_1 w_{14} + x_2 w_{24} - \theta_4) = 1/[1 + e^{-(1\times0.9+1\times1.0+1\times0.1)}] = 0.8808$$

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

$$y_5 = sigmoid\,(y_3 w_{35} + y_4 w_{45} - \theta_5) = 1/[1 + e^{-(-0.5250\times1.2+0.8808\times1.1-1\times0.3)}] = 0.5097$$

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

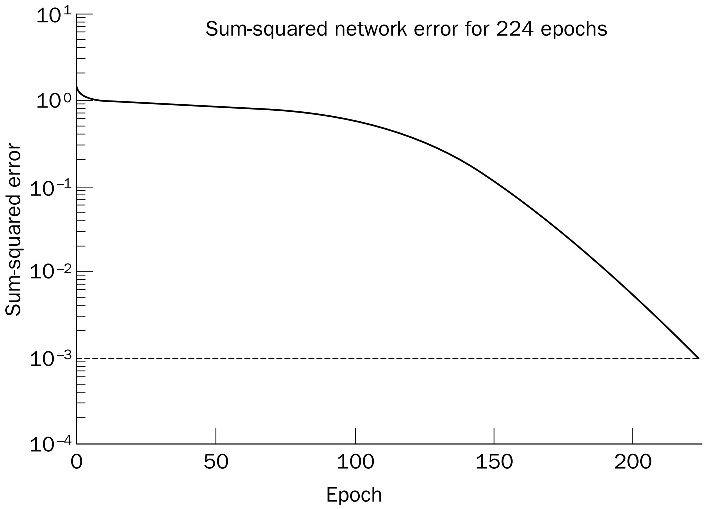


Figure 6.11 Learning curve for operation Exclusive-OR

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:

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

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:

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

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.

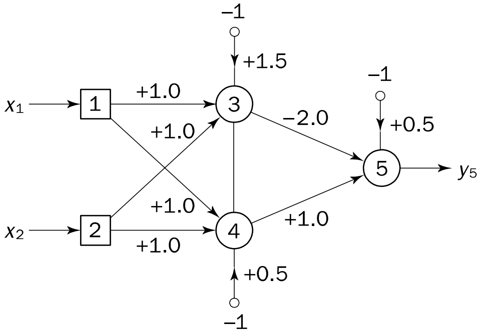


Figure 6.12 Network represented by McCulloch–Pitts model for solving the Exclusive-OR operation.

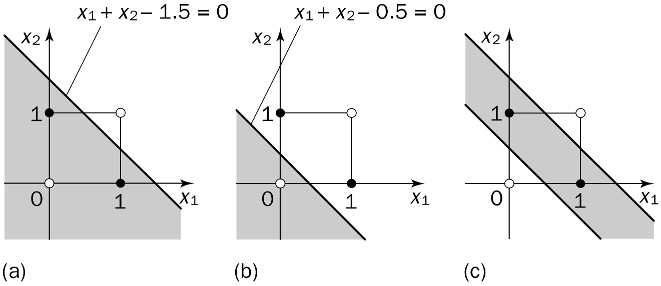


Figure 6.13 (a) Decision boundary constructed by hidden neuron 3 of the network in Figure 6.12; (b) decision boundary constructed by hidden neuron 4; (c) decision boundaries constructed by the complete three-layer network

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**,

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

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