Chapter 10

Basics of Neural Network

OBJECTIVE OF THE CHAPTER:

In the last 9 chapters, you have been introduced to the concepts of machine learning in great details. You started with how to decide whether a problem can be solved with machine learning, and once you have made that decision, you learnt how to start with the modelling of the problem in the machine learning paradigm. In that context, you were introduced to three types of machine learning – supervised, unsupervised, and reinforcement. Then, you explored all different popular algorithms of supervised and unsupervised learning. Now, you have gained quite some background on machine learning basics, and it is time for you to get introduced to the concept of neural network.

Well, you have already seen right at the beginning of the book how the machine learning process maps with the human learning process. Now, it is time to see how the human nervous system has been mimicked in the computer world in the form of an artificial neural network or simply a neural network. This chapter gives a brief view of neural networks and how it helps in different forms of learning.

10.1 INTRODUCTION

In the previous chapters, we were slowly unveiling the mystery of machine learning, i.e. how a machine learns to perform tasks and improves with time from experience, by expert guidance or by itself. Machine learning, as we have seen, mimics the human

form of learning. On the other hand, human learning, or for that matter every action of a human being, is controlled by the nervous system. In any human being, the nervous system coordinates the different actions by transmitting signals to and from different parts of the body. The nervous system is constituted of a special type of cell, called **neuron** or **nerve cell**, which has special structures allowing it to receive or send signals to other neurons. Neurons connect with each other to transmit signals to or receive signals from other neurons. This structure essentially forms a network of neurons or a neural network.

By virtue of billions of networked neurons that it possesses, the biological neural network is a massively large and complex parallel computing network. It is because of this massive parallel computing network that the nervous system helps human beings to perform actions or take decisions at a speed and with such ease that the fastest supercomputer of the world will also be envy of. For example, let us think of the superb flying catches taken by the fielders in the cricket world cup. It is a combination of superior calculation based on past cricketing experience, understanding of local on-ground conditions, and anticipation of how hard the ball has been hit that the fielder takes the decision about when to jump, where to jump, and how much to jump. This is a highly complex task, and you may think that not every human being is skilled enough for such a magical action. In that case, let us think of something much simpler. Let us consider about a very simple, daily activity, namely swimming in the pool. Apparently, swimming may look very trivial, but think about the parallel actions and decisions that need to be taken to swim. It needs the right combination of body position, limb movement, breathe in/out, etc. to swim. To add to the challenge is the fact that water is few hundred times denser than air. So, to orient the body movement in that scale difference is a difficult proposition in itself. Coordinating the actions and taking the decisions for such a complex task, which may appear to be trivial, are possible because of the massive parallel complex network, i.e. the neural network.

The fascinating capability of the biological neural network has inspired the inception of artificial neural network (ANN). An ANN is made up of artificial neurons. In its most generic form, an ANN is a machine designed to model the functioning of the nervous system or, more specifically, the neurons. The only difference is that the biological form of neuron is replicated in the electronic or digital form of neuron. Digital neurons or artificial neurons form the smallest processing units of the ANNs. As we move on, let us first do a deep dive into the structure of the biological neuron and then try to see how that has been modelled in the artificial neuron.

10.2 UNDERSTANDING THE BIOLOGICAL NEURON

The human nervous system has two main parts –

- the central nervous system (CNS) consisting of the brain and spinal cord
- the peripheral nervous system consisting of nerves and ganglia outside the brain and spinal cord.

The CNS integrates all information, in the form of signals, from the different parts of the body. The peripheral nervous system, on the other hand, connects the CNS with

the limbs and organs. Neurons are basic structural units of the CNS. A neuron is able to receive, process, and transmit information in the form of chemical and electrical signals. Figure 10.1 presents the structure of a neuron. It has three main parts to carry out its primary functionality of receiving and transmitting information:

1. **Dendrites** – to receive signals from neighbouring neurons.
2. **Soma** – main body of the neuron which accumulates the signals coming from the different dendrites. It 'fires' when a sufficient amount of signal is accumulated.
3. **Axon** – last part of the neuron which receives signal from soma, once the neuron 'fires', and passes it on to the neighbouring neurons through the axon terminals (to the adjacent dendrite of the neighbouring neurons).

There is a very small gap between the axon terminal of one neuron and the adjacent dendrite of the neighbouring neuron. This small gap is known as **synapse**. The signals transmitted through synapse may be excitatory or inhibitory.

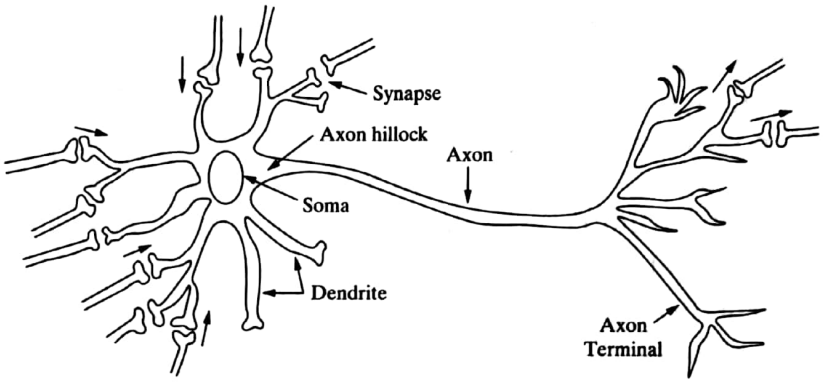


FIG. 10.1
Structure of biological neuron

Points to Ponder:

The adult human brain, which forms the main part of the central nervous system, is approximately 1.3 kg in weight and 1200 cm^3 in volume. It is estimated to contain about 100 billion (i.e. 10^{11}) neurons and 10 times more glial or glue cells. Glial cells act as support cells for the neurons. It is believed that neurons represent about 10% of all cells in the brain. On an average, each neuron is connected to 10^5 of other neurons, which means that altogether there are 10^{16} connections.

The axon, a human neuron, is 10–12 μm in diameter. Each synapse spans a gap of about a millionth of an inch wide.

■

10.3 EXPLORING THE ARTIFICIAL NEURON

The biological neural network has been modelled in the form of ANN with artificial neurons simulating the function of biological neurons. As depicted in Figure 10.2,

input signal x_i ($x_1, x_2, \ldots, x_n$) comes to an artificial neuron. Each neuron has three major components:

(i) A set of 'i' **synapses** having weight w_i. A signal x_i forms the input to the i-th synapse having weight w_i. The value of weight w_i may be positive or negative. A positive weight has an excitatory effect, while a negative weight has an inhibitory effect on the output of the summation junction, y_{sum}.

(ii) A **summation junction** for the input signals is weighted by the respective synaptic weight. Because it is a linear combiner or adder of the weighted input signals, the output of the summation junction, y_{sum}, can be expressed as follows:

$$y_{sum} = \sum_{i=1}^{n} w_i x_i$$

[Note: Typically, a neural network also includes a bias which adjusts the input of the activation function. However, for the sake of simplicity, we are ignoring bias for the time being. In the case of a bias 'b', the value of y_{sum} would have been as follows:

$$y_{sum} = b + \sum_{i=1}^{n} w_i x_i$$

(iii) A **threshold activation function** (or simply **activation function**, also called **squashing function**) results in an output signal only when an input signal exceeding a specific threshold value comes as an input. It is similar in behaviour to the biological neuron which transmits the signal only when the total input signal meets the firing threshold.

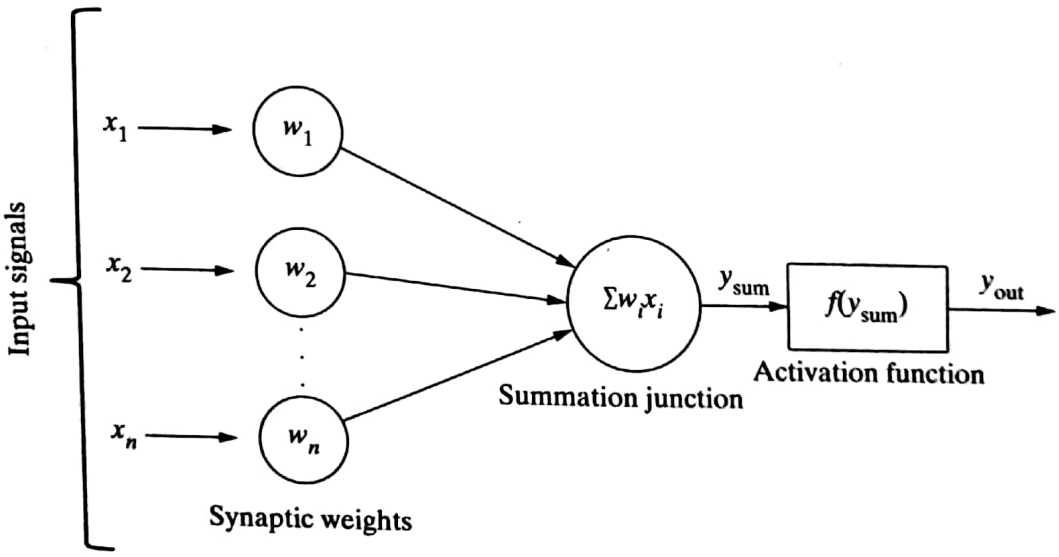


FIG. 10.2
Structure of an artificial neuron

Output of the activation function, y_{out}, can be expressed as follows:

$$y_{out} = f(y_{sum})$$

10.4 TYPES OF ACTIVATION FUNCTIONS

There are different types of activation functions. The most commonly used activation functions are highlighted below.

10.4.1 Identity function

Identity function is used as an activation function for the input layer. It is a linear function having the form

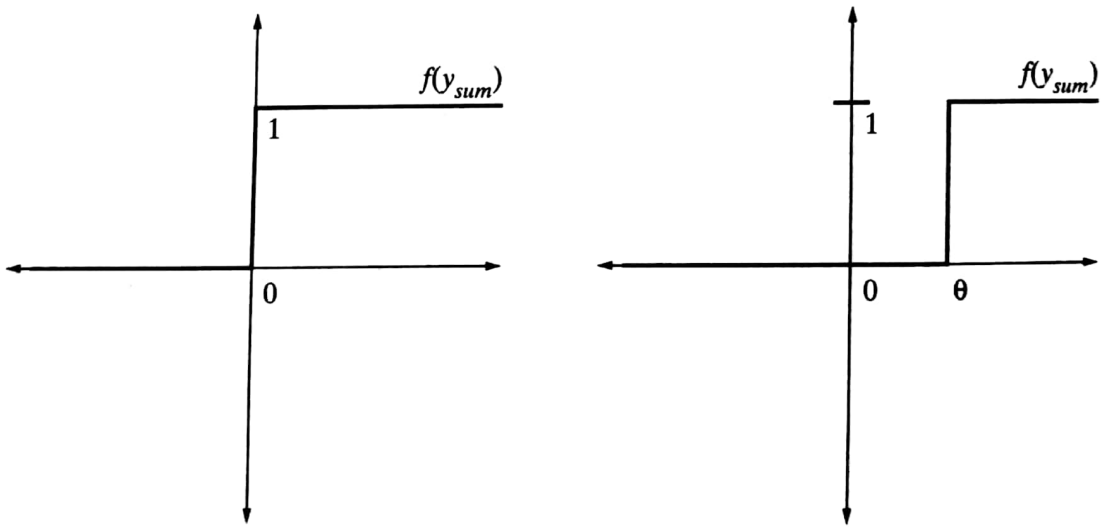
$$y_{out} = f(x) = x, \text{for all } x$$

As obvious, the output remains the same as the input.

10.4.2 Threshold/step function

Step/threshold function is a commonly used activation function. As depicted in Figure 10.3a, **step function** gives 1 as output if the input is either 0 or positive. If the input is negative, the step function gives 0 as output. Expressing mathematically,

$$y_{out} = f(y_{sum}) = \begin{cases} 1, & x \geq 0 \\ 0, & x < 0 \end{cases}$$



(a) Step function (b) Thershold function

FIG. 10.3
Step and threshold functions

The **threshold function** (depicted in Fig. 10.3b) is almost like the step function, with the only difference being the fact that θ is used as a threshold value instead of 0. Expressing mathematically,

$$y_{out} = f(y_{sum}) = \begin{cases} 1, & x \geq \theta \\ 0, & x < \theta \end{cases}$$

10.4.3 ReLU (Rectified Linear Unit) function

ReLU is the most popularly used activation function in the areas of convolutional neural networks and deep learning. It is of the form

$$f(x) = \begin{cases} x, & x \geq 0 \\ 0, & x < 0 \end{cases}$$

This means that $f(x)$ is zero when x is less than zero and $f(x)$ is equal to x when x is above or equal to zero. Figure 10.4 depicts the curve for a ReLU activation function.

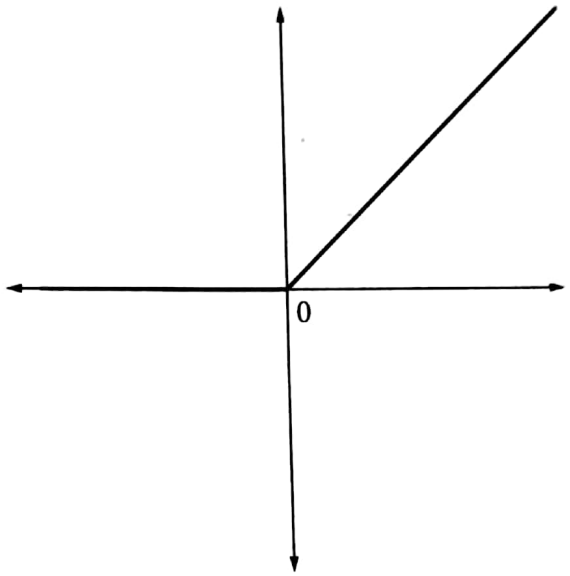


FIG. 10.4
ReLU function

This function is differentiable, except at a single point x = 0. In that sense, the derivative of a ReLU is actually a sub-derivative.

10.4.4 Sigmoid function

Sigmoid function, depicted in Figure 10.5, is by far the most commonly used activation function in neural networks. The need for sigmoid function stems from the fact that many learning algorithms require the activation function to be differentiable and hence continuous. Step function is not suitable in those situations as it is not continuous. There are two types of sigmoid function:

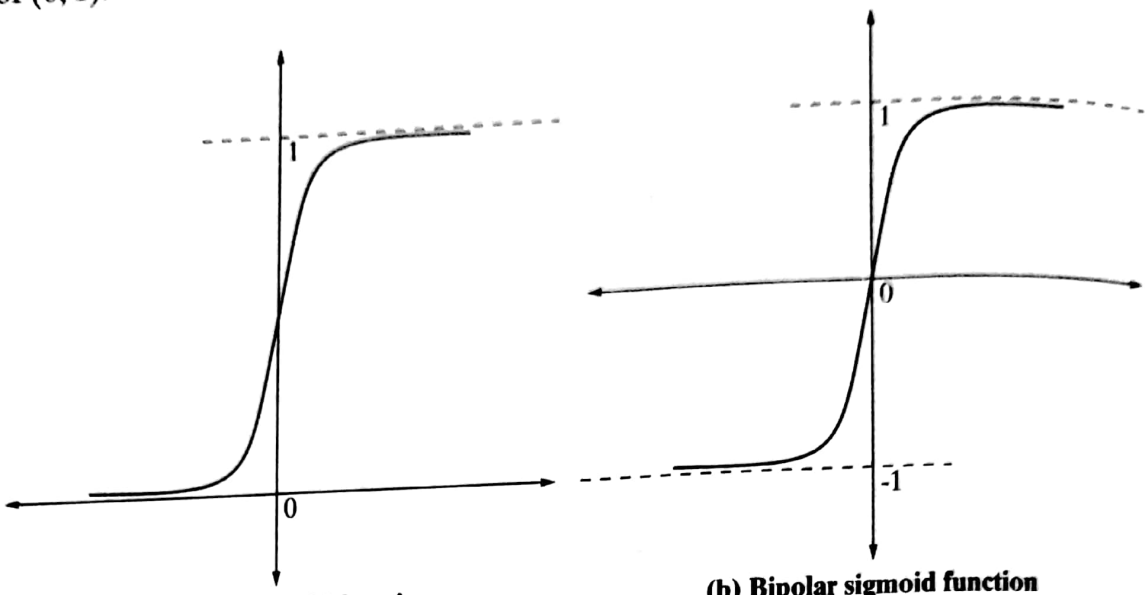
1. Binary sigmoid function
2. Bipolar sigmoid function

10.4.4.1 Binary sigmoid function

A binary sigmoid function, depicted in Figure 10.5a, is of the form

$$y_{out} = f(x) = \frac{1}{1 + e^{-kx}}$$

where k = steepness or slope parameter of the sigmoid function. By varying the value of k, sigmoid functions with different slopes can be obtained. It has range of (0, 1).



(a) Binary sigmoid function

(b) Bipolar sigmoid function

FIG. 10.5
Sigmoid function

The slope at origin is $k/4$. As the value of k becomes very large, the sigmoid function becomes a threshold function.

10.4.4.2 Bipolar sigmoid function

A bipolar sigmoid function, depicted in Figure 10.5b, is of the form

$$y_{out} = f(x) = \frac{1 - e^{-kx}}{1 + e^{-kx}}$$

The range of values of sigmoid functions can be varied depending on the application. However, the range of (−1, +1) is most commonly adopted.

10.4.5 Hyperbolic tangent function

Hyperbolic tangent function is another continuous activation function, which is bipolar in nature. It is a widely adopted activation function for a special type of neural network known as backpropagation network (discussed elaborately in Section 10.8). The hyperbolic tangent function is of the form

$$y_{out} = f(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

This function is similar to the bipolar sigmoid function.

Note that all the activation functions defined in Sections 10.4.2 and 10.4.4 have values ranging between 0 and 1. However, in some cases, it is desirable to have values

ranging from −1 to +1. In that case, there will be a need to reframe the activation function. For example, in the case of step function, the revised definition would be as follows:

$$y_{out} = f(y_{sum}) = \begin{cases} 1, & x > 0 \\ 0, & x = 0 \\ -1, & x < 0 \end{cases}$$

Did you know?

Works related to neural network dates long back to the 1940s. Warren McCulloch and Walter Pitts in 1943 created a computational network inspired by the biological processes in the brain. This model paved the way for neural networks.

The next notable work in the area of neural network was in the late 1940s by the Canadian psychologist Donald Olding Hebb. He proposed a learning hypothesis based on the neurons and synaptic connection between neurons. This became popular as Hebbian learning.

In 1958, the American psychologist Frank Rosenblatt refined the Hebbian concept and evolved the concept of perceptron. It was almost at the same time, in 1960, that Professor Bernard Widrow of Stanford University came up with an early single-layer ANN named ADALINE (Adaptive Linear Neuron or later Adaptive Linear Element). Marvin Lee Minsky and Seymour Aubrey Papert stated two key issues of perceptron in their research paper in 1969. The first issue stated was the inability of perceptron of processing the XOR (exclusive-OR) circuit. The other issue was lack of processing power of computers to effectively handle the large neural networks.

From here till the middle of the 1970s, there was a slowdown in the research work related to neural networks. There was a renewed interest generated in neural networks and learning with Werbos' backpropagation algorithm in 1975. Slowly, as the computing ability of the computers increased drastically, a lot of research on neural network has been conducted in the later decades. Neural network evolved further as deep neural networks and started to be implemented in solving large-scale learning problems such as image recognition, thereby getting a more popular name as deep learning.

10.5 EARLY IMPLEMENTATIONS OF ANN

10.5.1 McCulloch–Pitts model of neuron

The McCulloch–Pitts neural model (depicted in Fig. 10.6), which was the earliest ANN model, has only two types of inputs – excitatory and inhibitory. The excitatory inputs have weights of positive magnitude and the inhibitory weights have weights of negative magnitude. The inputs of the McCulloch–Pitts neuron could be either 0 or 1. It has a threshold function as activation function. So, the output signal y_{out} is 1 if the input y_{sum} is greater than or equal to a given threshold value, else 0.

Simple McCulloch–Pitts neurons can be used to design logical operations. For that purpose, the connection weights need to be correctly decided along with the threshold

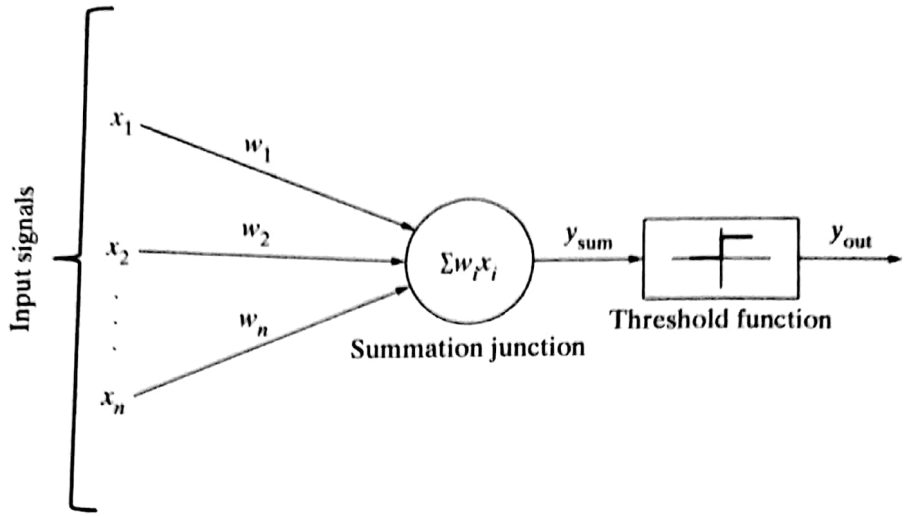


FIG. 10.6
McCulloch–Pitts neuron

function (rather the threshold value of the activation function). Let us take a small example.

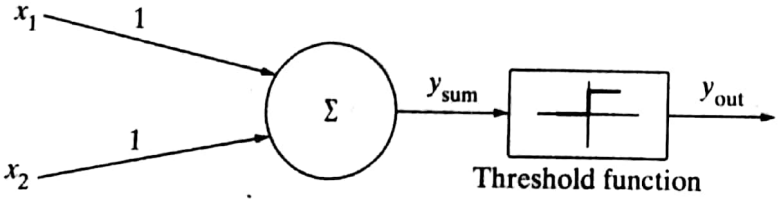
John carries an umbrella if it is sunny or if it is raining. There are four given situations. We need to decide when John will carry the umbrella. The situations are as follows:

- Situation 1 – It is not raining nor is it sunny.
- Situation 2 – It is not raining, but it is sunny.
- Situation 3 – It is raining, and it is not sunny.
- Situation 4 – Wow, it is so strange! It is raining as well as it is sunny.

To analyse the situations using the McCulloch–Pitts neural model, we can consider the input signals as follows:

- $x_1 \rightarrow$ Is it raining?
- $x_2 \rightarrow$ Is it sunny?

So, the value of both x_1 and x_2 can be either 0 or 1. We can use the value of both weights x_1 and x_2 as 1 and a threshold value of the activation function as 1. So, the neural model will look as Figure 10.7a.



(a) McCulloch–Pitts neural model

Situation	x_1	x_2	y_{sum}	y_{out}
1	0	0	0	0
2	0	1	1	1
3	1	0	1	1
4	1	1	2	1

(b) Truth table

FIG. 10.7
McCulloch–Pitts neural model (illustration)

Formally, we can say,

$$y_{\text{sum}} = \sum_{i=1}^{2} w_i x_i$$

$$y_{\text{out}} = f(y_{\text{sum}}) = \begin{cases} 1, & x \geq 1 \\ 0, & x < 1 \end{cases}$$

The truth table built with respect to the problem is depicted in Figure 10.7b. From the truth table, we can conclude that in the situations where the value of y_{out} is 1, John needs to carry an umbrella. Hence, he will need to carry an umbrella in situations 2, 3, and 4. Surprised, as it looks like a typical logic problem related to 'OR' function? Do not worry, it is really an implementation of logical OR using the McCulloch–Pitts neural model.

10.5.2 Rosenblatt's perceptron

Rosenblatt's perceptron is built around the McCulloch–Pitts neural model. The perceptron, as depicted in Figure 10.7, receives a set of input $x_1, x_2, \ldots, x_n$. The linear combiner or the adder node computes the linear combination of the inputs applied to the synapses with synaptic weights being $w_1, w_2, \ldots, w_n$. Then, the hard limiter checks whether the resulting sum is positive or negative. If the input of the hard limiter node is positive, the output is +1, and if the input is negative, the output is –1. Mathematically, the hard limiter input is

$$v = \sum_{i=1}^{n} w_i x_i$$

However, perceptron includes an adjustable value or bias as an additional weight w_0. This additional weight w_0 is attached to a dummy input x_0, which is always assigned a value of 1. This consideration modifies the above equation to

$$v = \sum_{i=0}^{n} w_i x_i$$

The output is decided by the expression

$$y_{\text{out}} = f(v) = \begin{cases} +1, v > 0 \\ -1, v < 0 \end{cases}$$

The objective of perceptron is to classify a set of inputs into two classes, c_1 and c_2. This can be done using a very simple decision rule – assign the inputs $x_0, x_1, x_2, \ldots, x_n$ to c_1 if the output of the perceptron, i.e. y_{out}, is +1 and c_2 if y_{out} is –1. So, for an n-dimensional signal space, i.e. a space for 'n' input signals $x_0, x_1, x_2, \ldots, x_n$, the simplest form of perceptron will have two decision regions, resembling two classes, separated by a hyperplane defined by

$$\sum_{i=0}^{n} w_i x_i = 0$$

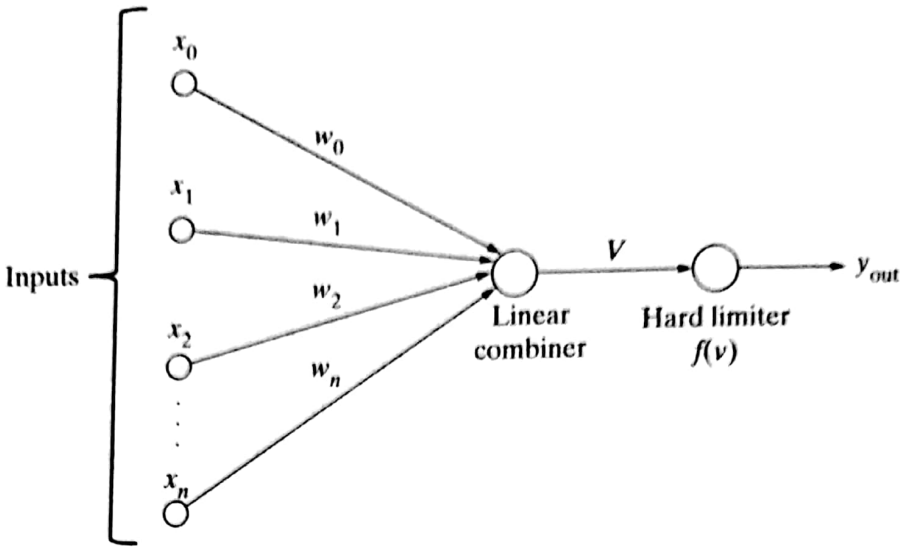


FIG. 10.8
Rosenblatt's perceptron

Therefore, for two input signals denoted by variables x_1 and x_2, the decision boundary is a straight line of the form

$$w_0x_0 + w_1x_1 + w_2x_2 = 0$$
$$\text{or, } w_0 + w_1x_1 + w_2x_2 = 0[\because x_0 = 1]$$

So, for a perceptron having the values of synaptic weights w_0, w_1, and w_2 as -2, ½, and ¼, respectively, the linear decision boundary will be of the form

$$-2 + ½x_1 + ¼\,x_2 = 0$$
$$\text{or, } 2x_1 + x_2 = 8$$

So, any point (x_1, x_2) which lies above the decision boundary, as depicted by Figure 10.9, will be assigned to class c_1 and the points which lie below the boundary are assigned to class c_2.

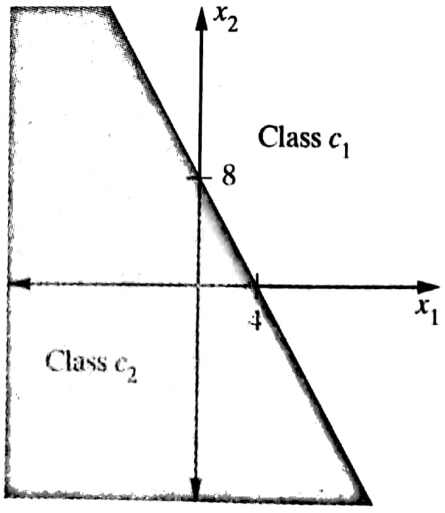


FIG. 10.9
Perceptron decision boundary

Let us examine if this perceptron is able to classify a set of points given below:

$p_1 = (5, 2)$ and $p_2 = (-1, 12)$ belonging to c_1

$p_3 = (3, -5)$ and $p_4 = (-2, -1)$ belonging to c_2

As depicted in Figure 10.10, we can see that on the basis of activation function output, only points p_1 and p_2 generate an output of 1. Hence, they are assigned to class c_1 as expected. On the other hand, p_3 and p_4 points having activation function output as negative generate an output of 0. Hence, they are assigned to class c_2, again as expected.

point	$v = \sum w_i x_i$	$y_{out} = f(v)$	Class
p_1	$-2 + (½)*5 + (¼)*2 = 1$	1	c_1
p_2	$-2 + (½)*(-1) + (¼)*12 = 0.5$	1	c_1
p_3	$-2 + (½)*3 + (¼)*(-5) = -1.75$	0	c_2
p_4	$-2 + (½)*(-2) + (¼)*(-1) = -3.25$	0	c_2

FIG. 10.10
Class assignment through perceptron

The same classification is obtained by mapping the points in the input space, as shown in Figure 10.11.

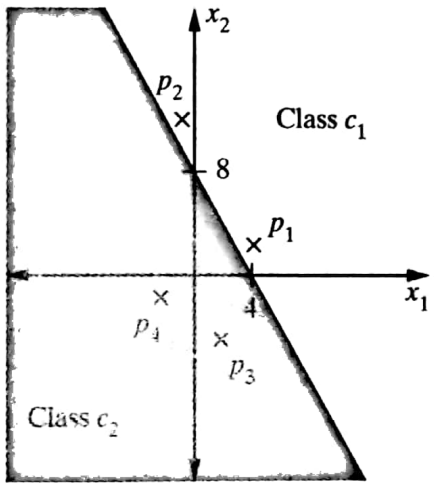
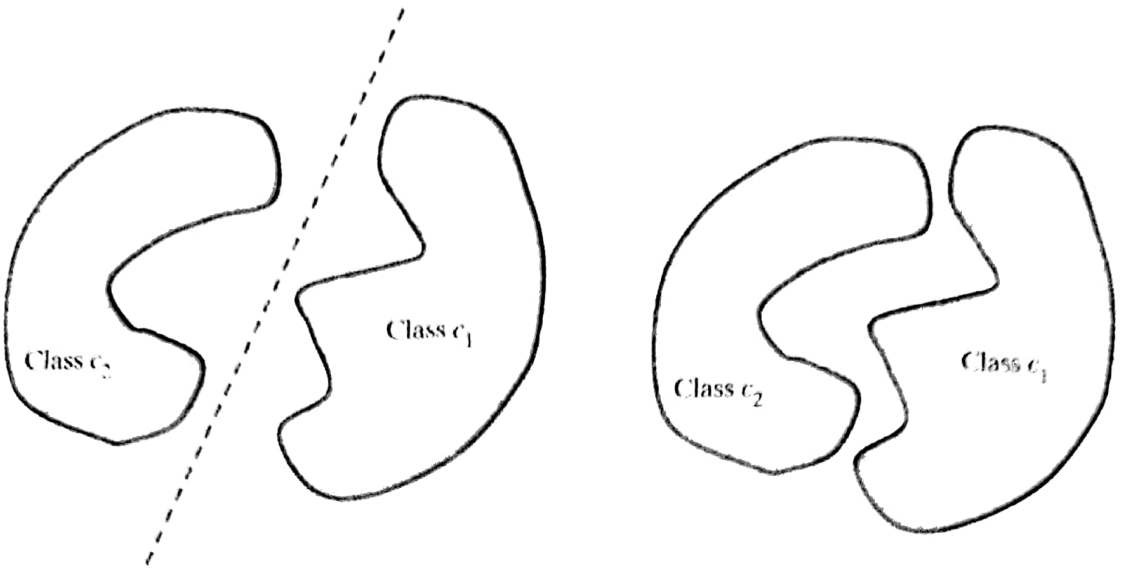


FIG. 10.11
Classification by decision boundary

Thus, we can see that for a data set with linearly separable classes, perceptrons can always be employed to solve classification problems using decision lines (for two-dimensional space), decision planes (for three-dimensional space), or decision hyperplanes (for n-dimensional space).

Appropriate values of the synaptic weights $w_0, w_1, w_2, \ldots, w_n$ can be obtained by training a perceptron. However, one assumption for perceptron to work properly is that the two classes should be linearly separable (as depicted in Figure 10.12a), i.e. the classes should be sufficiently separated from each other. Otherwise, if the classes are



(a) Linearly separable classes (b) Non-linearly separable classes

FIG. 10.12
Class separability

non-linearly separable (as depicted in Figure 10.12b), then the classification problem cannot be solved by perceptron.

10.5.2.1 Multi-layer perceptron

A basic perceptron works very successfully for data sets which possess linearly separable patterns. However, in practical situation, that is an ideal situation to have. This was exactly the point driven by Minsky and Papert in their work (1969). They showed that a basic perceptron is not able to learn to compute even a simple 2-bit XOR. Why is that so? Let us try to understand.

Figure 10.13 is the truth table highlighting output of a 2-bit XOR function.

x_1	x_2	x_1 XOR x_2	Class
1	1	0	c_2
1	0	1	c_1
0	1	1	c_1
0	0	0	c_2

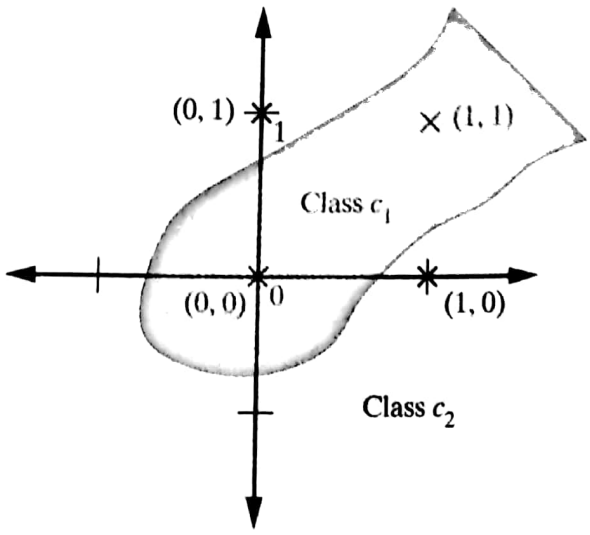


FIG. 10.13
Class separability of XOR function output

As we can see in Figure 10.13, the data is not linearly separable. Only a curved decision boundary can separate the classes properly.

To address this issue, the other option is to use two decision lines in place of one. Figure 10.14 shows how a linear decision boundary with two decision lines can clearly partition the data.

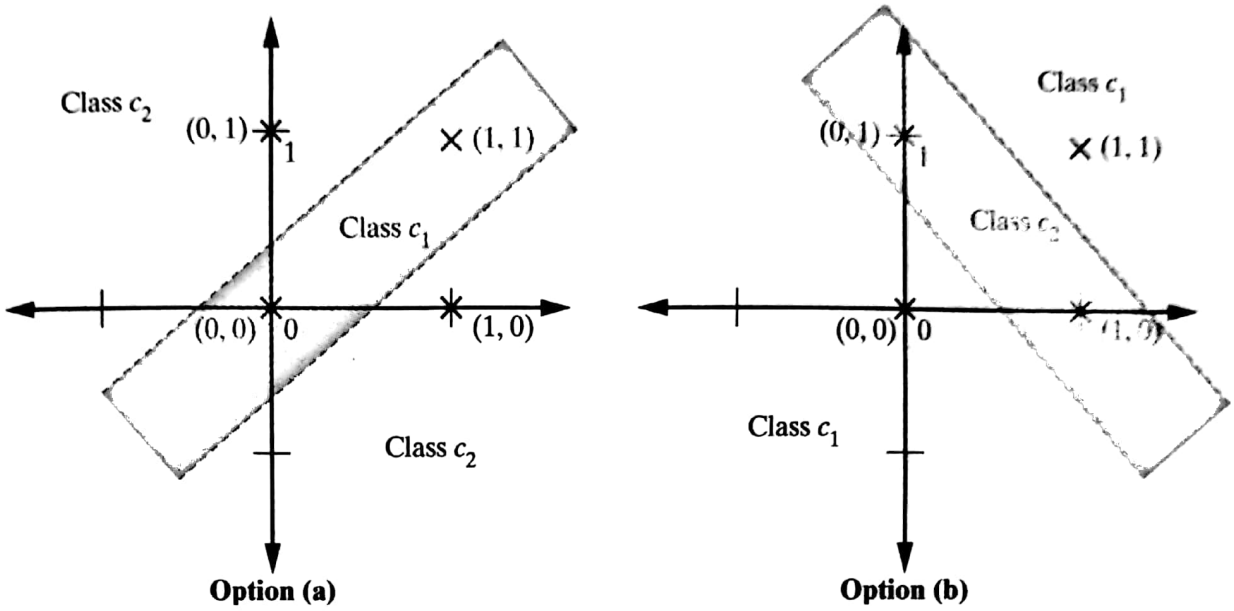


FIG. 10.14
Classification with two decision lines in XOR function output

This is the philosophy used to design the multi-layer perceptron model. The major highlights of this model are as follows:

- The neural network contains one or more intermediate layers between the input and the output nodes, which are hidden from both input and output nodes.
- Each neuron in the network includes a non-linear activation function that is differentiable.
- The neurons in each layer are connected with some or all the neurons in the previous layer.

The diagram in Figure 10.15 resembles a fully connected multi-layer perceptron with multiple hidden layers between the input and output layers. It is called fully connected because any neuron in any layer of the perceptron is connected with all neurons (or input nodes in the case of the first hidden layer) in the previous layer. The signals flow from one layer to another layer from left to right.

10.5.3 ADALINE network model

Adaptive Linear Neural Element (ADALINE) is an early single-layer ANN developed by Professor Bernard Widrow of Stanford University. As depicted in Figure 10.16, it has only output neuron. The output value can be +1 or −1. A bias input x_0 (where

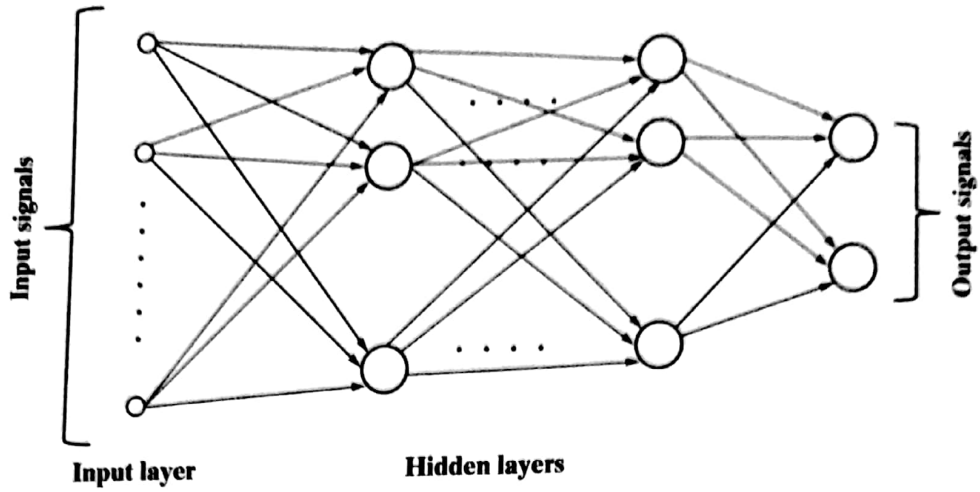


FIG. 10.15
Multi-layer perceptron

$x_0 = 1$) having a weight w_0 is added. The activation function is such that if the weighted sum is positive or 0, then the output is 1, else it is –1. Formally, we can say,

$$y_{\text{sum}} = \sum_{i=1}^{n} w_i x_i + b, \text{ where } b = w_0$$

$$y_{\text{out}} = f(y_{\text{sum}}) = \begin{cases} 1, & x \geq 1 \\ -1, & x < 1 \end{cases}$$

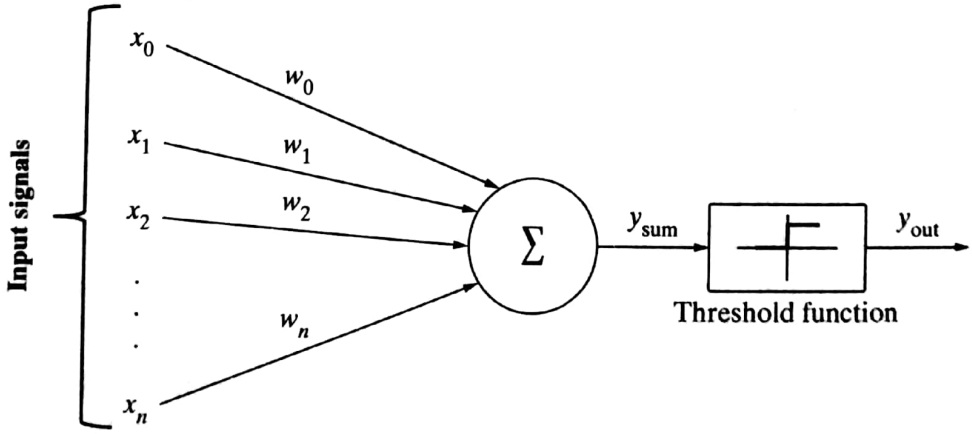


FIG. 10.16
ADALINE network

The supervised learning algorithm adopted by the ADALINE network is known as Least Mean Square (LMS) or Delta rule.

A network combining a number of ADALINEs is termed as MADALINE (many ADALINE). MADALINE networks can be used to solve problems related to non-linear separability.

Note:

Both perceptron and ADALINE are neural network models. Both of them are classifiers for binary classification. They have linear decision boundary and use a threshold activation function.

10.6 ARCHITECTURES OF NEURAL NETWORK

As we have seen till now, ANN is a computational system consisting of a large number of interconnected units called artificial neurons. The connection between artificial neurons can transmit signal from one neuron to another. There are multiple possibilities for connecting the neurons based on which architecture we are going to adopt for a specific solution. Some of the choices are listed below:

- There may be just two layers of neuron in the network – the input and output layer.
- Other than the input and output layers, there may be one or more intermediate 'hidden' layers of neuron.
- The neurons may be connected with one or more of the neurons in the next layer.
- The neurons may be connected with all neurons in the next layer.
- There may be single or multiple output signals. If there are multiple output signals, they might be connected with each other.
- The output from one layer may become input to neurons in the same or preceding layer.

10.6.1 Single-layer feed forward network

Single-layer feed forward is the simplest and most basic architecture of ANNs. It consists of only two layers as depicted in Figure 10.17 – the input layer and the output layer. The input layer consists of a set of 'm' input neurons $X_1, X_2, \ldots, X_m$ connected to each of the 'n' output neurons $Y_1, Y_2, \ldots, Y_n$. The connections carry weights w_{11}, $w_{12}, \ldots, w_{mn}$. The input layer of neurons does not conduct any processing – they pass the input signals to the output neurons. The computations are performed only by the neurons in the output layer. So, though it has two layers of neurons, only one layer is performing the computation. This is the reason why the network is known as single layer in spite of having two layers of neurons. Also, the signals always flow from the input layer to the output layer. Hence, this network is known as feed forward.

The net signal input to the output neurons is given by

$$y_{\text{in\_}k} + x_1 w_{1k} + x_2 w_{2k} + \ldots + x_m w_{mk} = \sum_{i=1}^{m} x_i w_{ik},$$

for the k-th output neuron. The signal output from each output neuron will depend on the activation function used.

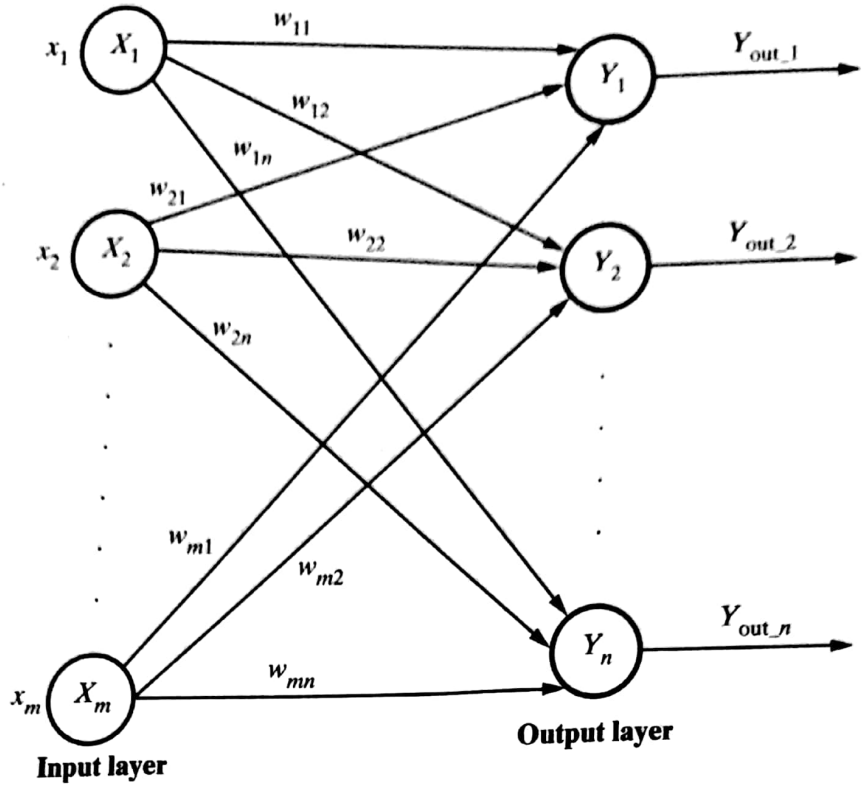


FIG. 10.17
Single-layer feed forward

10.6.2 Multi-layer feed forward ANNs

The multi-layer feed forward network is quite similar to the single-layer feed forward network, except for the fact that there are one or more intermediate layers of neurons between the input and the output layers. Hence, the network is termed as multi-layer. The structure of this network is depicted in Figure 10.18.

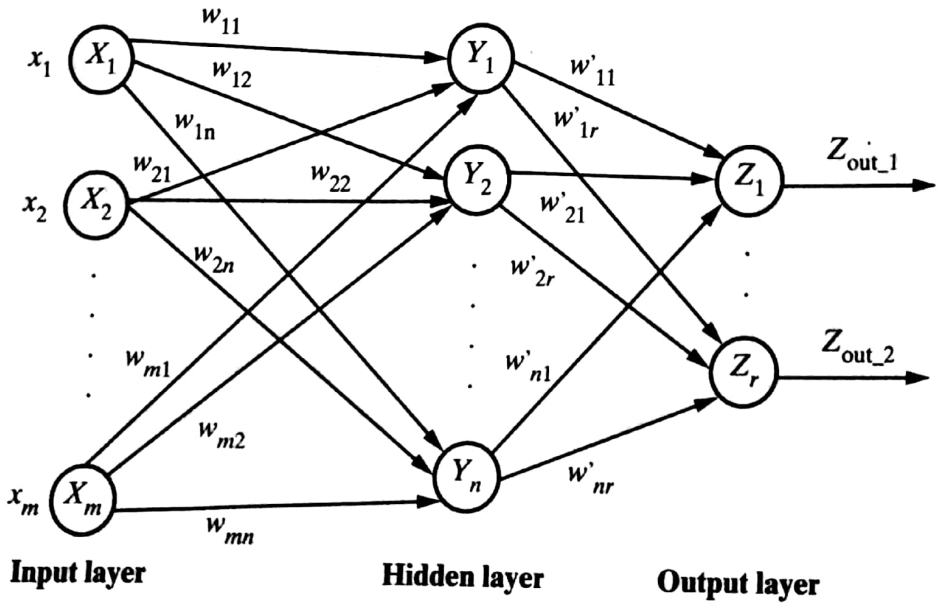


FIG. 10.18
Multi-layer feed forward

Each of the layers may have varying number of neurons. For example, the one shown in Figure 10.18 has 'm' neurons in the input layer and 'r' neurons in the output layer, and there is only one hidden layer with 'n' neurons.

The net signal input to the neuron in the hidden layer is given by

$$y_{in\_k} = x_1 w_{1k} + x_2 w_{2k} + \ldots + x_m w_{mk} = \sum_{i=1}^{m} x_i w_{ik},$$

for the k-th hidden layer neuron. The net signal input to the neuron in the output layer is given by

$$z_{in\_k} = y_{out\_1} w'_{1k} + y_{out\_2} w'_{2k} + \ldots + y_{out\_n} w'_{nk} = \sum_{j=1}^{n} y_{out\_j} w'_{jk}$$

for the k-th output layer neuron.

10.6.3 Competitive network

The competitive network is almost the same in structure as the single-layer feed forward network. The only difference is that the output neurons are connected with each other (either partially or fully). Figure 10.19 depicts a fully connected competitive network.

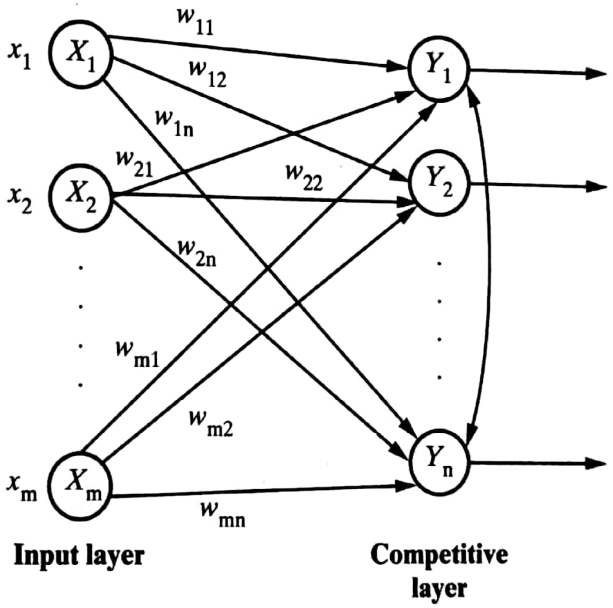


FIG. 10.19
Competitive network

In competitive networks, for a given input, the output neurons compete amongst themselves to represent the input. It represents a form of unsupervised learning algorithm in ANN that is suitable to find clusters in a data set.

10.6.4 Recurrent network

We have seen that in feed forward networks, signals always flow from the input layer towards the output layer (through the hidden layers in the case of multi-layer feed forward networks), i.e. in one direction. In the case of recurrent neural networks,

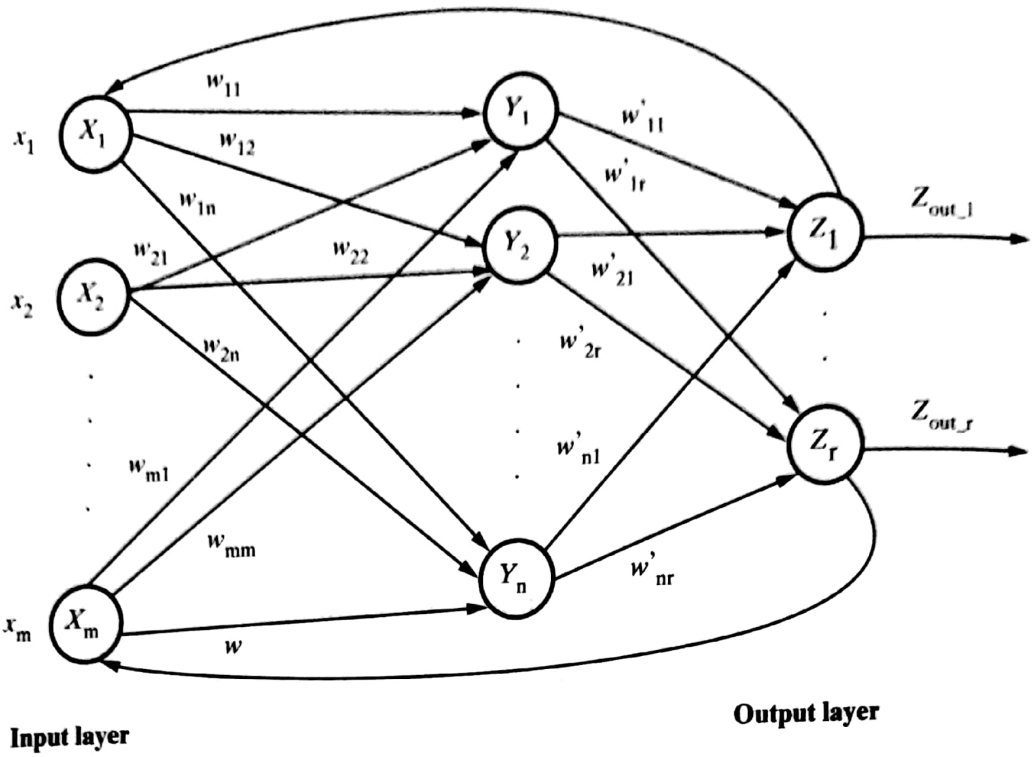


FIG. 10.20
Recurrent neural network

there is a small deviation. There is a feedback loop, as depicted in Figure 10.20, from the neurons in the output layer to the input layer neurons. There may also be self-loops.

10.7 LEARNING PROCESS IN ANN

Now that we have a clear idea about neurons, how they form networks using different architectures, what is activation function in a neuron, and what are the different choices of activation functions, it is time to relate all these to our main focus, i.e. learning. First, we need to understand what is learning in the context of ANNs? There are four major aspects which need to be decided:

(a) The number of layers in the network
(b) The direction of signal flow
(c) The number of nodes in each layer
(d) The value of weights attached with each interconnection between neurons

10.7.1 Number of layers

As we have seen earlier, a neural network may have a single layer or multi-layer. In the case of a single layer, a set of neurons in the input layer receives signal, i.e. a single feature per neuron, from the data set. The value of the feature is transformed

by the activation function of the input neuron. The signals processed by the neurons in the input layer are then forwarded to the neurons in the output layer. The neurons in the output layer use their own activation function to generate the final prediction.

More complex networks may be designed with multiple hidden layers between the input layer and the output layer. Most of the multi-layer networks are fully connected.

10.7.2 Direction of signal flow

In certain networks, termed as feed forward networks, signal is always fed in one direction, i.e. from the input layer towards the output layer through the hidden layers, if there is any. However, certain networks, such as the recurrent network, also allow signals to travel from the output layer to the input layer.

This is also an important consideration for choosing the correct learning model.

10.7.3 Number of nodes in layers

In the case of a multi-layer network, the number of nodes in each layer can be varied. However, the number of nodes or neurons in the input layer is equal to the number of features of the input data set. Similarly, the number of output nodes will depend on possible outcomes, e.g. number of classes in the case of supervised learning. So, the number of nodes in each of the hidden layers is to be chosen by the user. A larger number of nodes in the hidden layer help in improving the performance. However, too many nodes may result in overfitting as well as an increased computational expense.

10.7.4 Weight of interconnection between neurons

Deciding the value of weights attached with each interconnection between neurons so that a specific learning problem can be solved correctly is quite a difficult problem by itself. Let us try to understand it in the context of a problem. Let us take a step back and look at the problem in Section 10.6.2.

We have a set of points with known labels as given below. We have to train an ANN model using this data, so that it can classify a new test data, say p_5 (3, –2).

- $p_1 = (5, 2)$ and $p_2 = (-1, 12)$ belonging to c_1
- $p_3 = (3, -5)$ and $p_4 = (-2, -1)$ belonging to c_2

When we were discussing the problem in Section 10.6.2, we assumed the values of the synaptic weights w_0, w_1, and w_2 as –2, ½, and ¼, respectively. But where on earth did we get those values from? Will we get these weight values for every learning problem that we will attempt to solve using ANN? The answer is a big NO.

For solving a learning problem using ANN, we can start with a set of values for the synaptic weights and keep doing changes to those values in multiple iterations. In the case of supervised learning, the objective to be pursued is to reduce the number of misclassifications. Ideally, the iterations for making changes in weight values should be continued till there is no misclassification. However, in practice, such a stopping criterion may not be possible to achieve. Practical stopping criteria may be the rate of misclassification less than a specific threshold value, say 1%, or the maximum number

of iterations reaches a threshold, say 25, etc. There may be other practical challenges to deal with, such as the rate of misclassification is not reducing progressively. This may become a bigger problem when the number of interconnections and hence the number of weights keeps increasing. There are ways to deal with those challenges, which we will see in more details in the next section.

So, to summarize, learning process using ANN is a combination of multiple aspects – which include deciding the number of hidden layers, number of nodes in each of the hidden layers, direction of signal flow, and last but not the least, deciding the connection weights.

Multi-layer feed forward network is a commonly adopted architecture. It has been observed that a neural network with even one hidden layer can be used to reasonably approximate any continuous function. The learning method adopted to train a multi-layer feed forward network is termed as backpropagation, which we will study in the next section.

10.8 BACKPROPAGATION

We have already seen that one of the most critical activities of training an ANN is to assign the inter-neuron connection weights. It can be a very intense work, more so for the neural networks having a high number of hidden layers or a high number of nodes in a layer. In 1986, an efficient method of training an ANN was discovered. In this method, errors, i.e. difference in output values of the output layer and the expected values, are propagated back from the output layer to the preceding layers. Hence, the algorithm implementing this method is known as backpropagation, i.e. propagating the errors backward to the preceding layers.

The backpropagation algorithm is applicable for multi-layer feed forward networks. It is a supervised learning algorithm which continues adjusting the weights of the connected neurons with an objective to reduce the deviation of the output signal from the target output. This algorithm consists of multiple iterations, also known as **epochs**. Each epoch consists of two phases –

- A **forward phase** in which the signals flow from the neurons in the input layer to the neurons in the output layer through the hidden layers. The weights of the interconnections and activation functions are used during the flow. In the output layer, the output signals are generated.
- A **backward phase** in which the output signal is compared with the expected value. The computed errors are propagated backwards from the output to the preceding layers. The errors propagated back are used to adjust the interconnection weights between the layers.

The iterations continue till a stopping criterion is reached. Figure 10.21 depicts a reasonably simplified version of the backpropagation algorithm.

One main part of the algorithm is adjusting the interconnection weights. This is done using a technique termed as **gradient descent**. In simple terms, the algorithm calculates the partial derivative of the activation function by each interconnection weight to identify the 'gradient' or extent of change of the weight required to minimize the

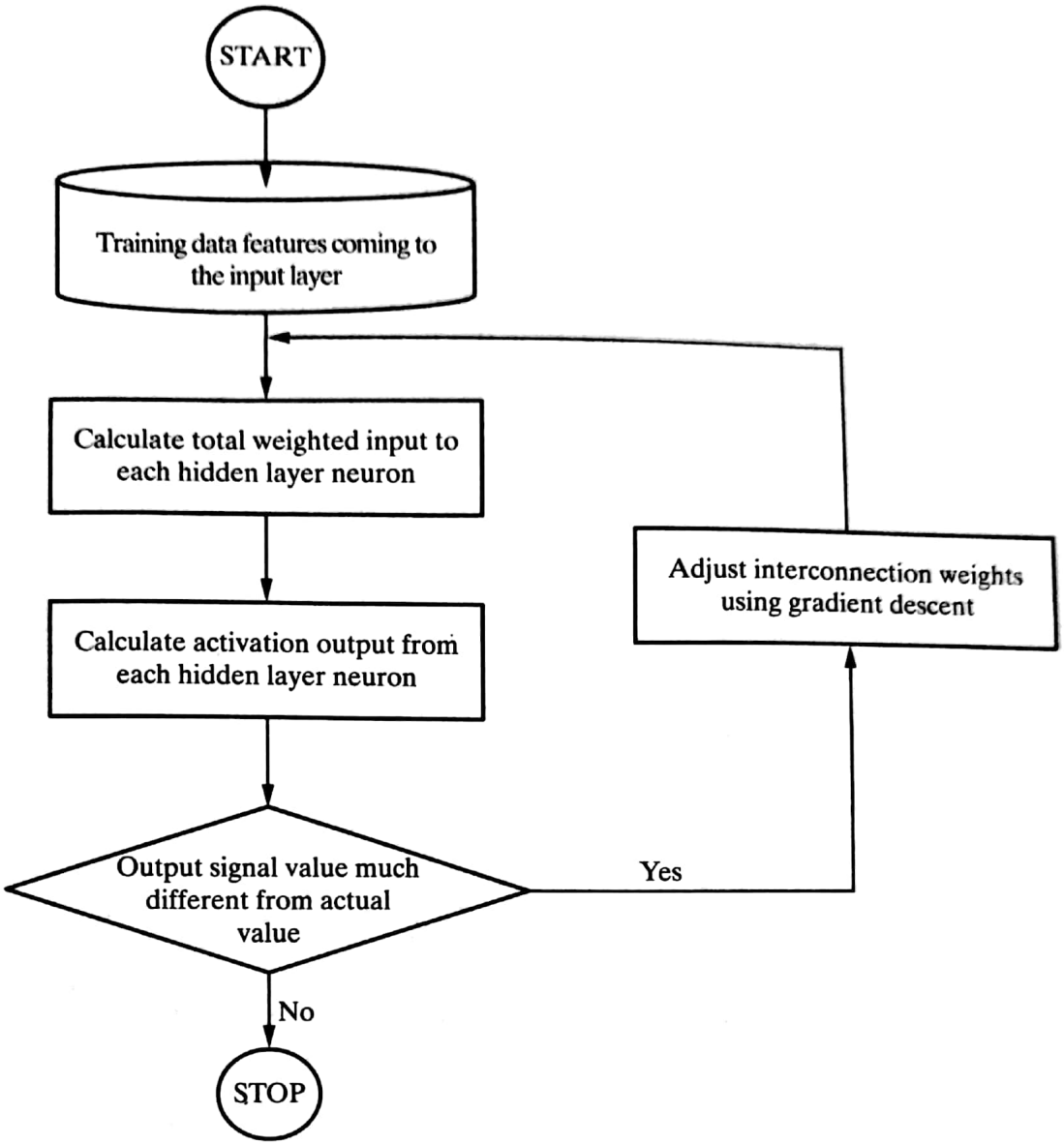


FIG. 10.21
Backpropagation algorithm

cost function. Quite understandably, therefore, the activation function needs to be differentiable. Let us try to understand this in a bit more details.

Points to Ponder:

A real-world simile for the gradient descent algorithm is a blind person trying to come down from a hill top without anyone to assist. The person is not able to see. So, for the person who is not able to see the path of descent, the only option is to check in which direction the land slope feels to be downward. One challenge of this approach arises when the person reaches a point which feels to be the lowest, as all the points surrounding it is higher in slope, but in reality it is not so. Such a point, which is local minima and not global minima, may be deceiving and stalls the algorithm before reaching the real global minima.

■

We have already seen that multi-layer neural networks have multiple hidden layers. During the learning phase, the interconnection weights are adjusted on the basis of the errors generated by the network, i.e. difference in the output signal of the network vis-à-vis the expected value. These errors generated at the output layer are propagated back to the preceding layers. Because of the backward propagation of errors which happens during the learning phase, these networks are also called back-propagation networks or simply backpropagation nets. One such backpropagation net with one hidden layer is depicted in Figure 10.22. In this network, X_0 is the bias input to the hidden layer and Y_0 is the bias input to the output layer.

The net signal input to the hidden layer neurons is given by

$$y_{in\_k} = x_0 w_{0k} + x_1 w_{1k} + x_2 w_{2k} + \ldots + x_m w_{mk} = w_{0k} + \sum_{i=1}^{m} x_i w_{ik},$$

for the k-th neuron in the hidden layer. If f_y is the activation function of the hidden layer, then

$$y_{out\_k} = f_y(y_{in\_k})$$

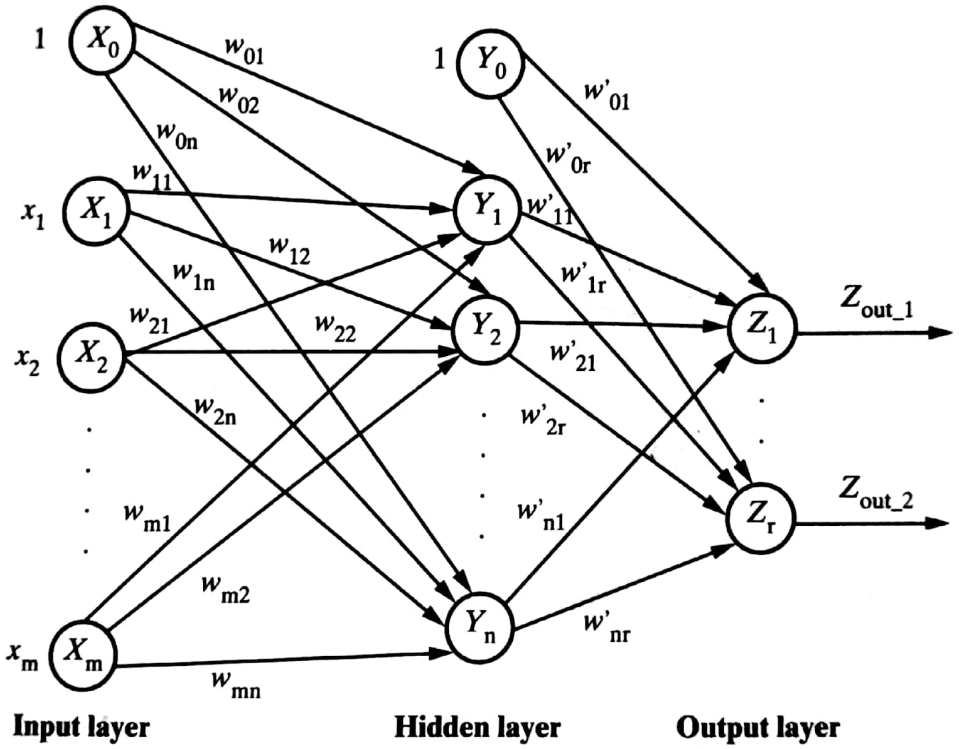


FIG. 10.22
Backpropagation net

The net signal input to the output layer neurons is given by

$$z_{in\_k} = y_0 w'_{0k} + y_{out\_1} w'_{1k} + y_{out\_2} w'_{2k} + \ldots + y_{out\_n} w'_{nk} = w'_{0k} + \sum_{i=1}^{n} y_{out\_i} w'_{ik},$$

for the k-th neuron in the output layer. Note that the input signals to X_0 and Y_0 are assumed as 1. If f_z is the activation function of the hidden layer, then

$$z_{out\_k} = f_z(z_{in\_k})$$

If t_k is the target output of the k-th output neuron, then the cost function defined as the squared error of the output layer is given by

$$E = \frac{1}{2}\sum_{k=1}^{n}(t_k - z_{\text{out\_}k})^2$$

$$= \frac{1}{2}\sum_{k=1}^{n}(t_k - f_z(z_{\text{in\_}k}))^2$$

Note:

There are two types of gradient descent algorithms. When a full data set is used in one shot to compute the gradient, it is known as **full batch gradient** descent. In the case of **stochastic gradient descent**, which is also known as **incremental gradient descent**, smaller samples of the data are taken iteratively and used in the gradient computation. Thus, stochastic gradient descent tries to identify the global minima.

So, as a part of the gradient descent algorithm, partial derivative of the cost function E has to be done with respect to each of the interconnection weights $w'_{01}, w'_{02}, \ldots, w'_{nr}$. Mathematically, it can be represented as follows:

$$\frac{\partial E}{\partial w'_{jk}} = \frac{\partial}{\partial w'_{jk}}\left\{\frac{1}{2}\sum_{k=1}^{n}(t_k - f_z(z_{\text{in\_}k}))^2\right\},$$

for the interconnection weight between the j-th neuron in the hidden layer and the k-th neuron in the output layer. This expression can be deduced to

$$\frac{\partial E}{\partial w'_{jk}} = -(t_k - z_{\text{out\_}k}) \cdot f'_z(z_{\text{in\_}k}) \cdot \frac{\partial}{\partial w'_{jk}}\left(\sum_{i=0}^{n} y_{\text{out\_}i} \cdot w_{ik}\right),$$

where $f'_z(z_{\text{in\_}k}) = \frac{\partial}{\partial w'_{jk}}(f_z(z_{\text{in\_}k}))$

$$\text{or, } \frac{\partial E}{\partial w'_{jk}} = -(t_k - z_{\text{out\_}k}) \cdot f'_z(z_{\text{in\_}k}) \cdot y_{\text{out\_}i}$$

If we assume $\delta w_k = -(t_k - z_{\text{out\_}k}).f'_z(z_{\text{in\_}k})$ as a component of the weight adjustment needed for weight w_{jk} corresponding to the k-th output neuron, then

$$\frac{\partial E}{\partial w'_{jk}} = \delta w'_k \cdot y_{\text{out\_}i}$$

On the basis of this, the weights and bias need to be updated as follows:

For weights: $\Delta w_{jk} = -\alpha.\frac{\partial E}{\partial w'_{jk}} = -\alpha.\,\delta w'_k \cdot y_{\text{out\_}i}$

Hence, $w'_{jk}(\text{new}) = w'_{jk}(\text{old}) + \Delta w'_{jk}$

For bias: $\Delta w_{0k} = -\alpha.\,\delta w'_k$

Hence, $w'_{0k}(\text{new}) = w'_{0k}(\text{old}) + \Delta w'_{0k}$

Note that 'α' is the learning rate of the neural network.

In the same way, we can perform the calculations for the interconnection weights between the input and hidden layers. The weights and bias for the interconnection between the input and hidden layers need to be updated as follows:

$$\text{For weights: } \Delta w_{ij} = -\alpha \cdot \frac{\partial E}{\partial w_{ij}} = -\alpha \cdot \delta w_j \cdot x_{out\_i}$$

$$\text{Hence, } w_{ij}\,(\text{new}) = w_{ij}\,(\text{old}) + \Delta w_{ij}$$

$$\text{For bias: } \Delta w_{0j} = -\alpha \cdot \delta w_j$$

$$\text{Hence, } w_{0j}\,(\text{new}) = w_{0j}\,(\text{old}) + \Delta w_{0j}$$

Note:

Learning rate is a user parameter which increases or decreases the speed with which the interconnection weights of a neural network is to be adjusted. If the learning rate is too high, the adjustment done as a part of the gradient descent process may miss the minima. Then, the training may not converge, and it may even diverge. On the other hand, if the learning rate is too low, the optimization may consume more time because of the small steps towards the minima.

A balanced approach is to start the training with a relatively large learning rate (say 0.1), because in the beginning, random weights are far from optimal. Then, the learning rate should be decreased during training (say exponentially lower values, e.g. 0.01, 0.001, etc.) to allow more fine-grained weight updates.

10.9 DEEP LEARNING

Neural networks, as we have already seen in this chapter, are a class of machine learning algorithms. As we have also seen, there are multiple choices of architectures for neural networks, multi-layer neural network being one of the most adopted ones. However, in a multi-layer neural network, as we keep increasing the number of hidden layers, the computation becomes very expensive. Going beyond two to three layers becomes quite difficult computationally. The only way to handle such intense computation is by using graphics processing unit (GPU) computing.

When we have less number of hidden layers – at the maximum two to three layers, it is a normal neural network, which is sometimes given the fancy name 'shallow neural network'. However, when the number of layers increases, it is termed as deep neural network. One of the earliest deep neural networks had three hidden layers. Deep learning is a more contemporary branding of deep neural networks, i.e. multi-layer neural networks having more than three layers. More detailed understanding of deep learning is beyond the scope of this book.