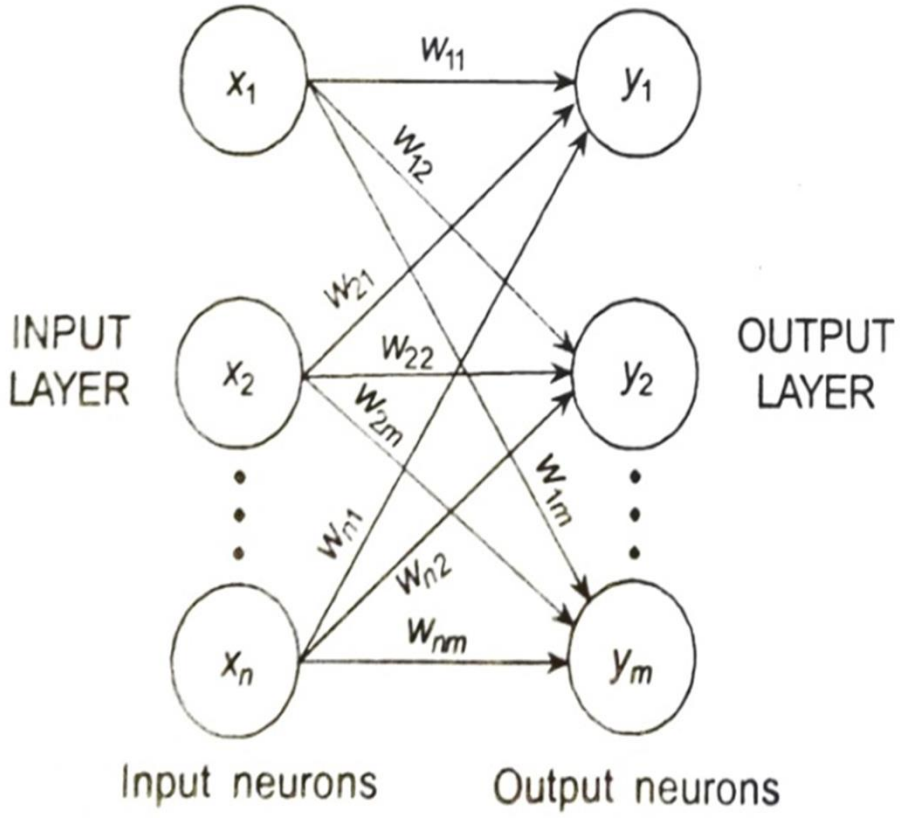
Application of Soft Computing

Unit 2

NETWORK ARCHITECTURES

1. Single-Layer Feedforward Networks

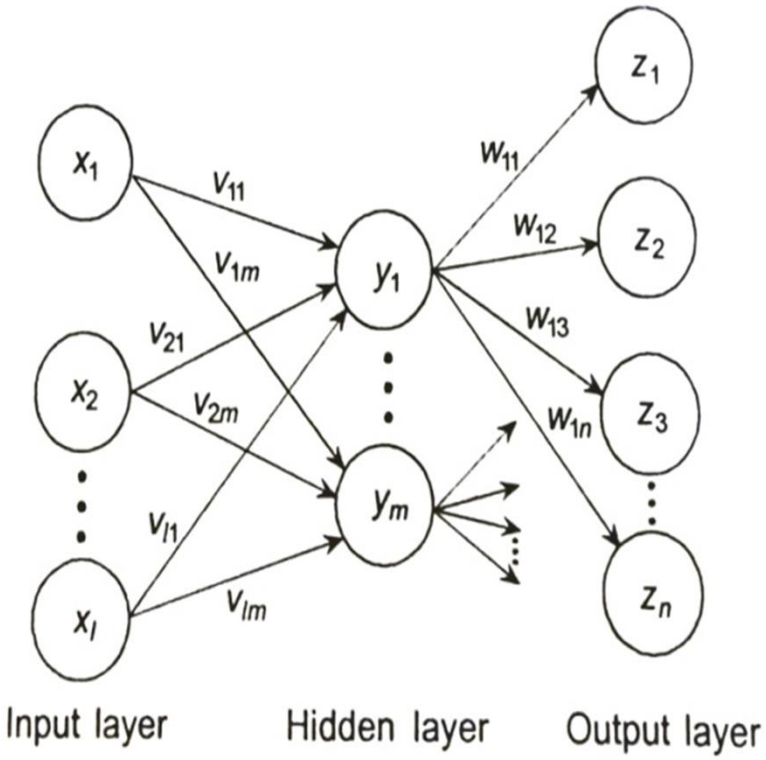


x_i: Input neurons

y_j: Output neurons

w_{ij}: Weights

MULTILAYER FEED FORWARD NETWORK



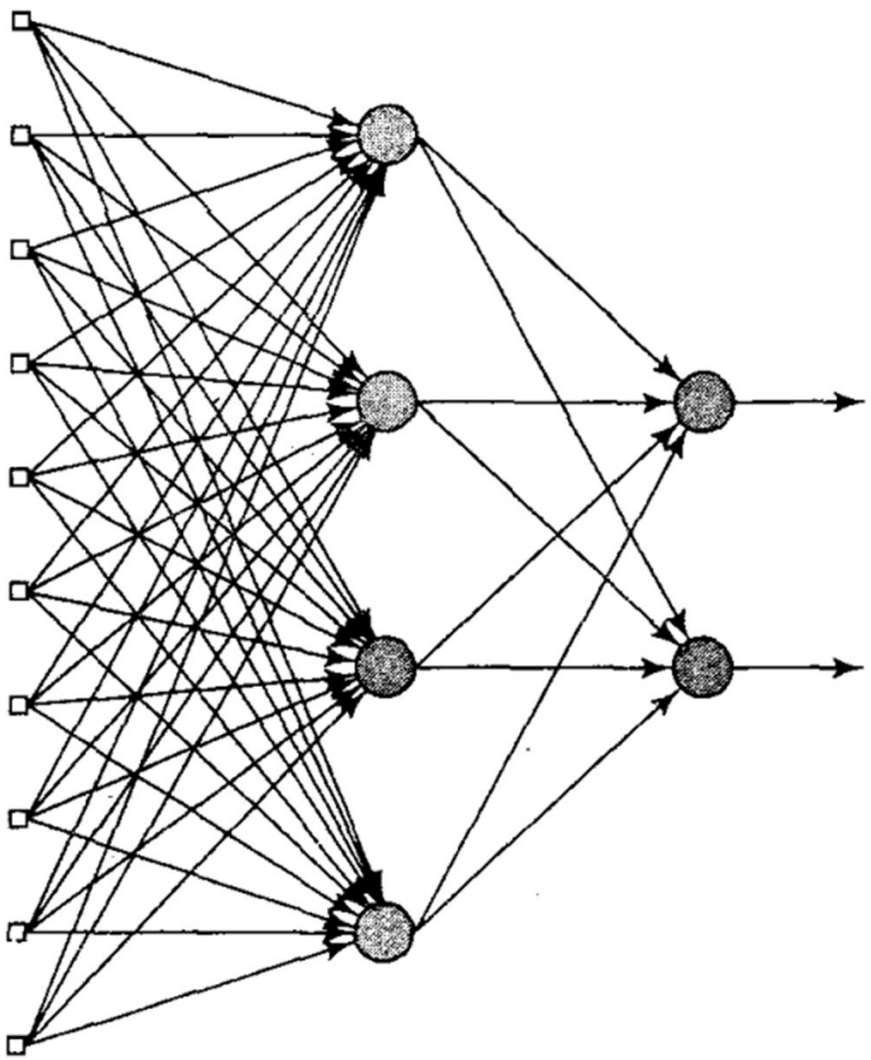
x_i : Input neurons

y_j : Hidden neurons

z_k : Output neurons

v_{ij} : Input hidden layer weights

w_{jk} : Output hidden layer weights



Input layer of source nodes

Layer of hidden neurons

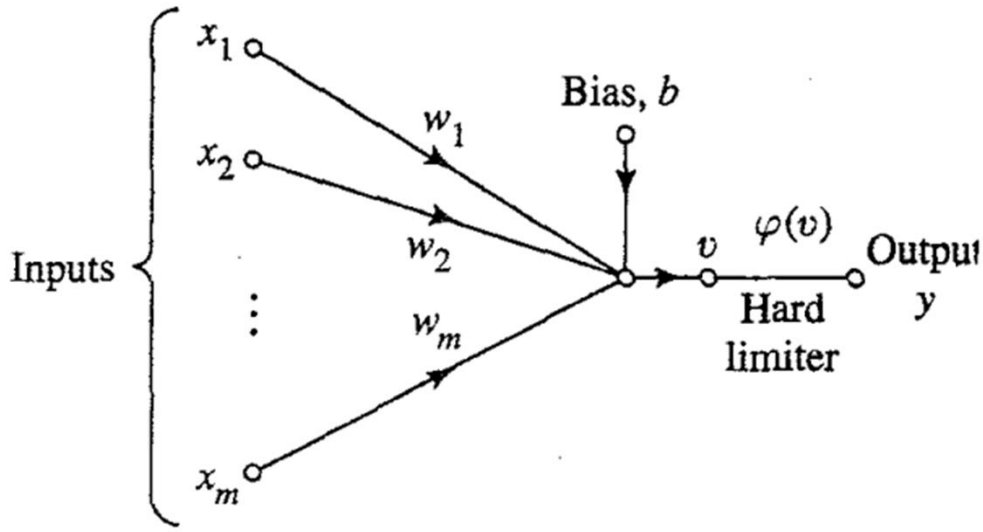
Layer of output neurons

Perceptron

The perceptron is the simplest form of a neural network used for the classification of patterns said to be *linearly separable* (i.e., patterns that lie on opposite sides of a hyperplane). Basically, it consists of a single neuron with adjustable synaptic weights and bias. The algorithm used to adjust the free parameters of this neural network first appeared in a learning procedure developed by Rosenblatt (1958, 1962) for his perceptron brain model.[1] Indeed, Rosenblatt proved that if the patterns (vectors) used to train the perceptron are drawn from two linearly separable classes, then the perceptron algorithm converges and positions the decision surface in the form of a hyperplane between the two classes. The proof of convergence of the algorithm is known as the *perceptron convergence theorem*. The perceptron built around a *single neuron* is limited to performing pattern classification with only two classes (hypotheses).

Signal-flow graph of the perceptron

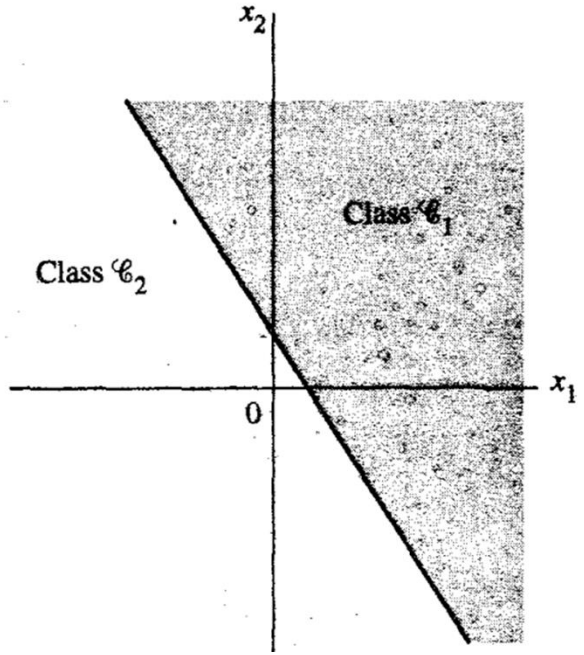
$$v = \sum_{i=1}^{m} w_i x_i + b$$



The goal of the perceptron is to correctly classify the set of externally applied stimuli $x_1, x_2, \ldots, x_m$ into one of two classes, $\mathscr{C}_1$ or $\mathscr{C}_2$. The decision rule for the classification is to assign the point represented by the inputs $x_1, x_2, \ldots, x_m$ to class $\mathscr{C}_1$ if the perceptron output y is $+1$ and to class $\mathscr{C}_2$ if it is -1.

The synaptic weights $w_1, w_2, \ldots, w_m$ of the perceptron can be adapted on an iteration-by-iteration basis. For the adaptation we may use an error-correction rule known as the perceptron convergence algorithm.

For the perceptron to function properly, the two classes $\mathscr{C}_1$ and $\mathscr{C}_2$ must be *linearly separable*. This, in turn, means that the patterns to be classified must be sufficiently separated from each other to ensure that the decision surface consists of a hyperplane.



Suppose then that the input variables of the perceptron originate from two linearly separable classes. Let $\mathscr{X}_1$ be the subset of training vectors $\mathbf{x}_1(1), \mathbf{x}_1(2), \ldots$ that belong to class $\mathscr{C}_1$, and let $\mathscr{X}_2$ be the subset of training vectors $\mathbf{x}_2(1), \mathbf{x}_2(2), \ldots$ that belong to class $\mathscr{C}_2$. The union of $\mathscr{X}_1$ and $\mathscr{X}_2$ is the complete training set $\mathscr{X}$. Given the sets of vectors $\mathscr{X}_1$ and $\mathscr{X}_2$ to train the classifier, the training process involves the adjustment of the weight vector $\mathbf{w}$ in such a way that the two classes $\mathscr{C}_1$ and $\mathscr{C}_2$ are linearly separable. That is, there exists a weight vector $\mathbf{w}$ such that we may state

$$\mathbf{w}^T\mathbf{x} > 0 \text{ for every input vector } \mathbf{x} \text{ belonging to class } \mathscr{C}_1$$

$$\mathbf{w}^T\mathbf{x} \leq 0 \text{ for every input vector } \mathbf{x} \text{ belonging to class } \mathscr{C}_2$$

The algorithm for adapting the weight vector of the elementary perceptron may now be formulated as follows:

1. If the nth member of the training set, $\mathbf{x}(n)$, is correctly classified by the weight vector $\mathbf{w}(n)$ computed at the nth iteration of the algorithm, no correction is made to the weight vector of the perceptron in accordance with the rule:

$$\mathbf{w}(n+1) = \mathbf{w}(n) \quad \text{if } \mathbf{w}^T\mathbf{x}(n) > 0 \text{ and } \mathbf{x}(n) \text{ belongs to class } \mathscr{C}_1$$

$$\mathbf{w}(n+1) = \mathbf{w}(n) \quad \text{if } \mathbf{w}^T\mathbf{x}(n) \leq 0 \text{ and } \mathbf{x}(n) \text{ belongs to class } \mathscr{C}_2$$

2. Otherwise, the weight vector of the perceptron is updated in accordance with the rule

$$\begin{aligned} \mathbf{w}(n+1) &= \mathbf{w}(n) - \eta(n)\mathbf{x}(n) \quad \text{if } \mathbf{w}^T(n)\mathbf{x}(n) > 0 \text{ and } \mathbf{x}(n) \text{ belongs to class } \mathscr{C}_2 \\ \mathbf{w}(n+1) &= \mathbf{w}(n) + \eta(n)\mathbf{x}(n) \quad \text{if } \mathbf{w}^T(n)\mathbf{x}(n) \leq 0 \text{ and } \mathbf{x}(n) \text{ belongs to class } \mathscr{C}_1 \end{aligned} \tag{3.55}$$

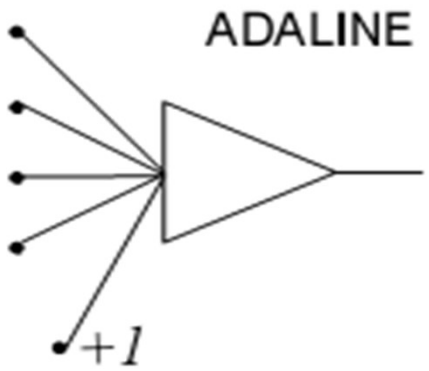
where the *learning-rate parameter* $\eta(n)$ controls the adjustment applied to the weight vector at iteration n.

Suppose that we are going to work on AND Gate problem using perceptron. The gate returns true value if and only if both inputs are true.
We are going to set weights randomly. Let's say that $w_1 = 0.9$ and $w_2 = 0.9$. Learning rate=0.5, bias = 0.5

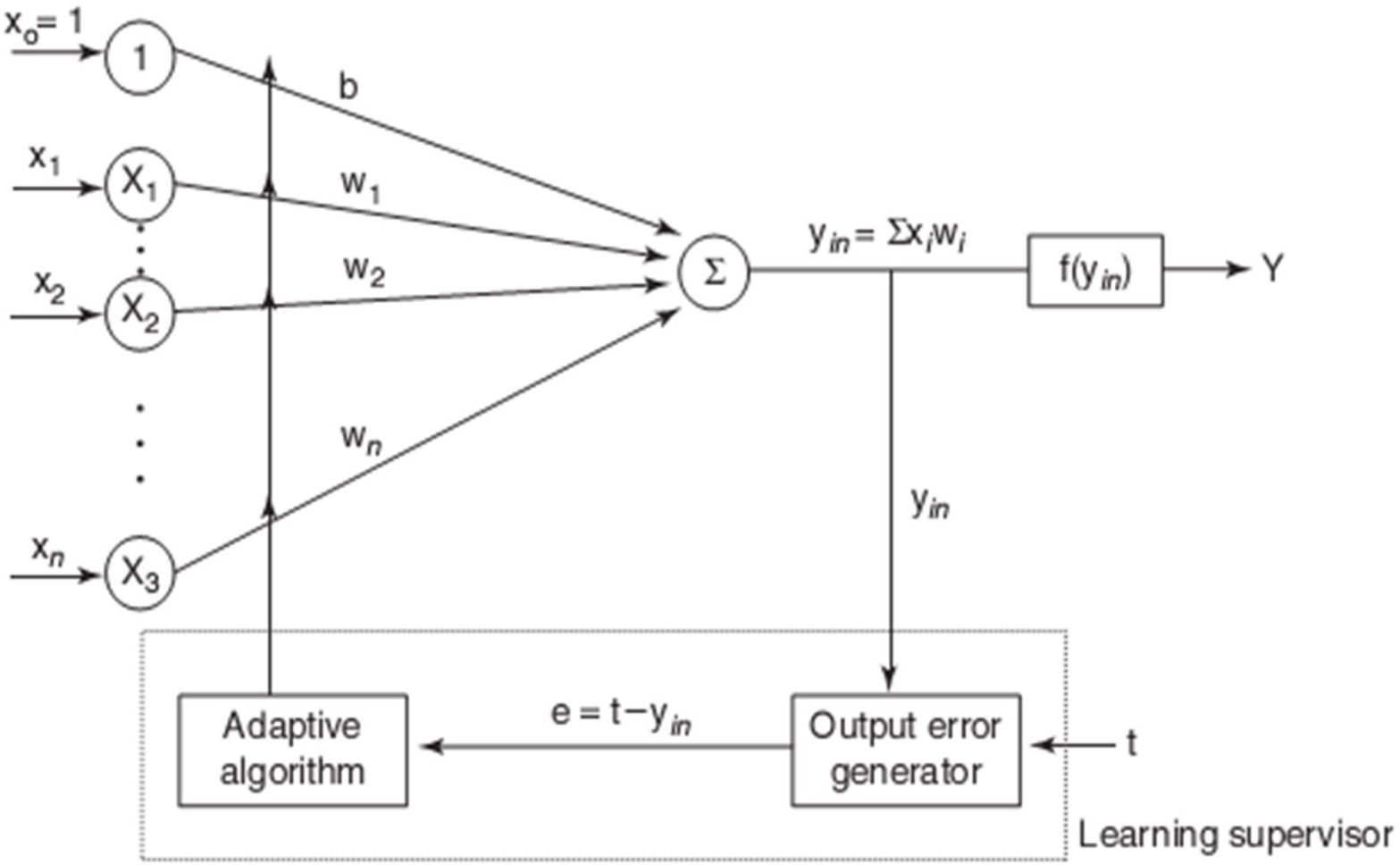
X_1	X_2	Y
0	0	0
0	1	0
1	0	0
1	1	1

ADAPTIVE LINEAR NEURON (ADALINE)

In 1959, Bernard Widrow and Marcian Hoff of Stanford developed models they called ADALINE (Adaptive Linear Neuron) and MADALINE (Multilayer ADALINE). These models were named for their use of Multiple ADAptive LINear Elements. MADALINE was the first neural network to be applied to a real world problem. It is an adaptive filter which eliminates echoes on phone lines.



ADALINE MODEL



ADALINE LEARNING RULE

Adaline network uses Delta Learning Rule. This rule is also called as Widrow Learning Rule or Least Mean Square Rule. The delta rule for adjusting the weights is given as (i = 1 to n):

$$\Delta w_i = \alpha(t - y_{in})x_i$$

Δw_i = weight change

α = learning rate

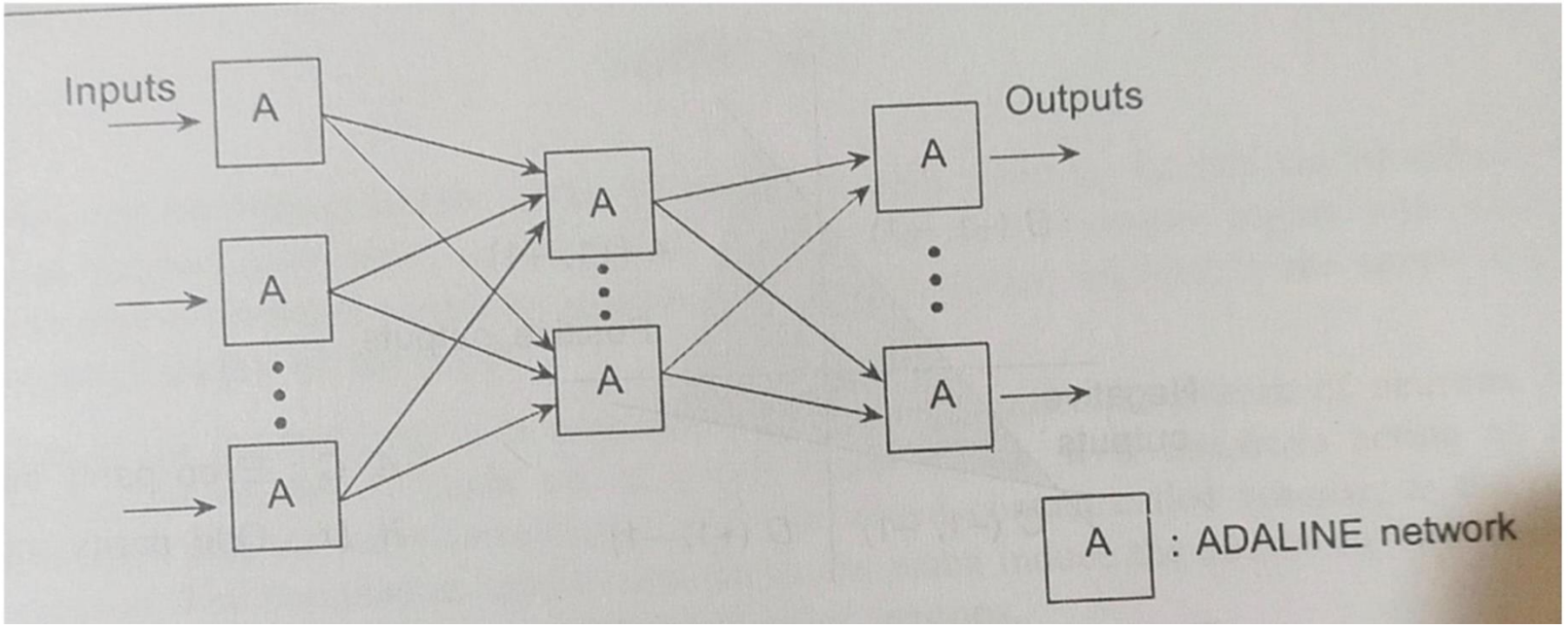
x = vector of activation of input unit

y_{in} = net input to output unit, $i.e.$, $Y = \sum_{i=1}^{n} x_i w_i$

t = target output

MADALINE Network

- A MADALINE (Many ADALINE) network is created by combining a number of
  The network of ADALINES can span many layers.
- The learning rule adopted by MADALINE network is termed as MADALINE A
  Rule' (MR) and is a form of supervised learning.
- In this method, the objective is to adjust the weights such that the error is mi
  the current training pattern, but with as little damage to the learning acquire
  previous training patterns.
- It solves the problem of non-linear separatibility.



Multilayer Perceptron Model

A multilayer perceptron has three distinctive characteristics:

1. Non-linear Activation Function

The model of each neuron in the network includes a *nonlinear activation function*. The important point to emphasize here is that the nonlinearity is *smooth* (i.e., differentiable everywhere), as opposed to the hard-limiting used in Rosenblatt's perceptron. A commonly used form of nonlinearity that satisfies this requirement is a *sigmoidal nonlinearity*[1] defined by the *logistic function:*

$$y_j = \frac{1}{1 + \exp(-v_j)}$$

2. One or more hidden layers: To learn Complex task

3. High Degree of Connectivity

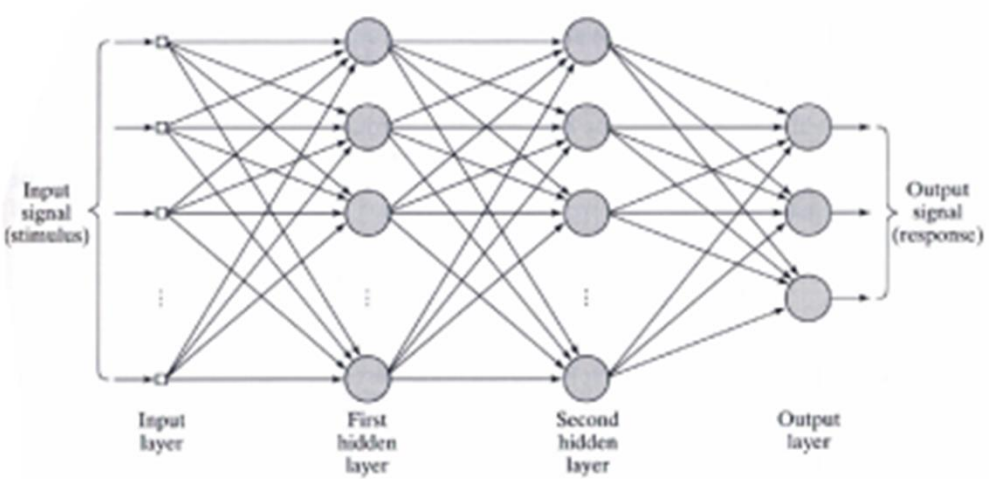


FIGURE 4.1 Architectural graph of a multilayer perceptron with two hidden layers.

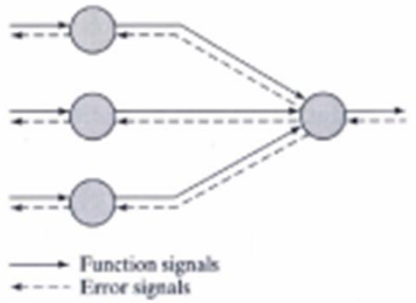


FIGURE 4.2 Illustration of the directions of two basic signal flows in a multilayer perceptron: forward propagation of function signals and back-propagation of error signals.

- Networks typically consisting of input, hidden, and output layers.
- Commonly referred to as *Multilayer perceptrons*.
- Popular learning algorithm is the *error backpropagation algorithm* (backpropagation, or backprop, for short), which is a generalization of the LMS rule.
  - Forward pass: activate the network, layer by layer
  - Backward pass: error signal backpropagates from output to hidden and hidden to input, based on which weights are updated.

Backpropagation Algorithm

Backpropagation is a supervised learning algorithm, for training Multi-layer Perceptron (Artificial Neural Networks).

Each hidden or output neuron of a multilayer perceptron is designed to perform two computations:

1. The computation of the function signal appearing at the output of a neuron, which is expressed as a continuous nonlinear function of the input signal and synaptic weights associated with that neuron.
2. The computation of an estimate of the gradient vector (i.e., the gradients of the error surface with respect to the weights connected to the inputs of a neuron), which is needed for the backward pass through the network.

To calculate gradient vector or gradient of error surface, derivative of activation function will be required. Hence before getting into the computation of back propagation algorithm, Let us go through some preliminary work to find out the derivatives of commonly used activation functions.

Activation Function

1. *Logistic Function.* This form of sigmoidal nonlinearity in its general form is defined by

$$\varphi_j(v_j(n)) = \frac{1}{1 + \exp(-av_j(\mathrm{n}))} \qquad a > 0 \text{ and } -\infty < v_j(n) < \infty \tag{4.30}$$

where $v_j(n)$ is the induced local field of neuron j. According to this nonlinearity, the amplitude of the output lies inside the range $0 \leq y_j \leq 1$. Differentiating Eq. (4.30) with respect to $v_j(n)$, we get

$$\varphi_j'(v_j(n)) = \frac{a\exp(-av_j(n))}{[1 + \exp(-av_j(n))]^2} \tag{4.31}$$

With $y_j(n) = \varphi_j(v_j(n))$, we may eliminate the exponential term $\exp(-av_j(n))$ from Eq. (4.31), and so express the derivative $\varphi_j'(v_j(n))$ as

$$\varphi_j'(v_j(n)) = ay_j(n)[1 - y_j(n)] \tag{4.32}$$

For a neuron j located in the output layer, $y_j(n) = o_j(n)$. Hence, we may express the local gradient for neuron j as

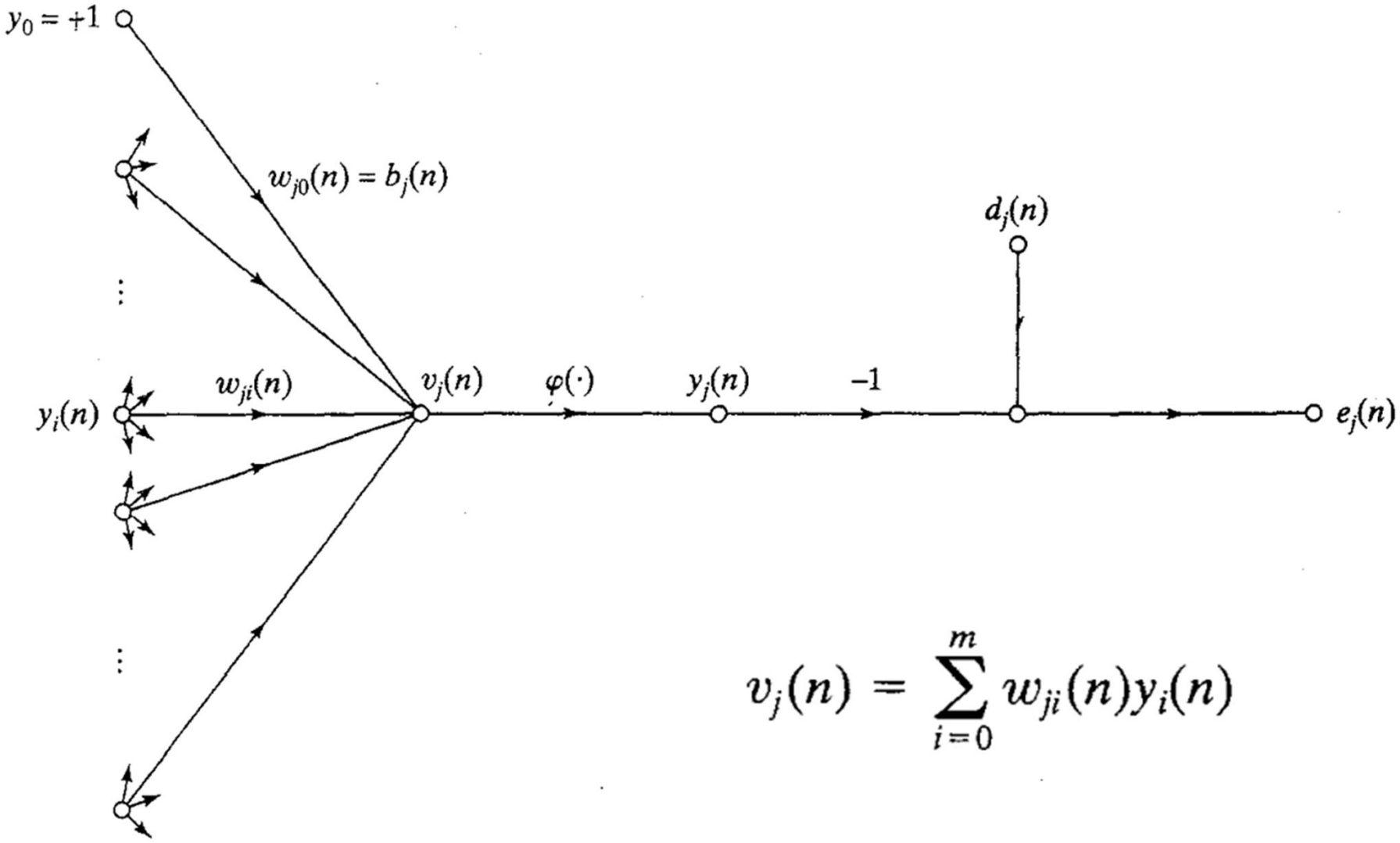
2. *Hyperbolic tangent function.* Another commonly used form of sigmoidal nonlinearity is the hyperbolic tangent function, which in its most general form is defined by

$$\varphi_j(v_j(n)) = a \tanh(bv_j(n)), \qquad (a,b) > 0 \tag{4.35}$$

where a and b are constants. In reality, the hyperbolic tangent function is just the logistic function rescaled and biased. Its derivative with respect to $v_j(n)$ is given by

$$\begin{aligned} \varphi_j'(v_j(n)) &= ab \operatorname{sech}^2(bv_j(n)) \\ &= ab\left(1 - \tanh^2(bv_j(n))\right) \\ &= \frac{b}{a}[a - y_j(n)][a + y_j(n)] \end{aligned} \tag{4.36}$$

Backpropagation Algorithm



$$v_j(n) = \sum_{i=0}^{m} w_{ji}(n) y_i(n)$$

Signal-flow graph highlighting the details of output neuron *j*.

Hence the function signal $y_j(n)$ appearing at the output of neuron j at iteration n is

$$y_j(n) = \varphi_j(v_j(n))$$

In a manner similar to the LMS algorithm, the back-propagation algorithm applies a correction $\Delta w_{ji}(n)$ to the synaptic weight $w_{ji}(n)$, which is proportional to the partial derivative $\partial \mathscr{E}(n)/\partial w_{ji}(n)$. According to the *chain rule* of calculus, we may express this gradient as:

$$\frac{\partial \mathscr{E}(n)}{\partial w_{ji}(n)} = \frac{\partial \mathscr{E}(n)}{\partial e_j(n)} \frac{\partial e_j(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)} \frac{\partial v_j(n)}{\partial w_{ji}(n)}$$

We define the instantaneous value of the error energy for neuron j as $\frac{1}{2}e_j^2(n)$. Correspondingly, the instantaneous value $\mathscr{E}(n)$ of the total error energy is obtained by summing $\frac{1}{2}e_j^2(n)$ over *all neurons in the output layer;* these are the only "visible" neurons for which error signals can be calculated directly. We may thus write

$$\mathscr{E}(n) = \frac{1}{2} \sum_{j \in C} e_j^2(n)$$

$$\frac{\partial \mathscr{E}(n)}{\partial e_j(n)} = e_j(n)$$

$$\frac{\partial e_j(n)}{\partial y_j(n)} = -1$$

$$\frac{\partial y_j(n)}{\partial v_j(n)} = \varphi_j'(v_j(n))$$

$$\frac{\partial v_j(n)}{\partial w_{ji}(n)} = y_i(n)$$

(4.14)

$$\frac{\partial \mathscr{E}(n)}{\partial w_{ji}(n)} = -e_j(n)\varphi_j'(v_j(n))y_i(n)$$

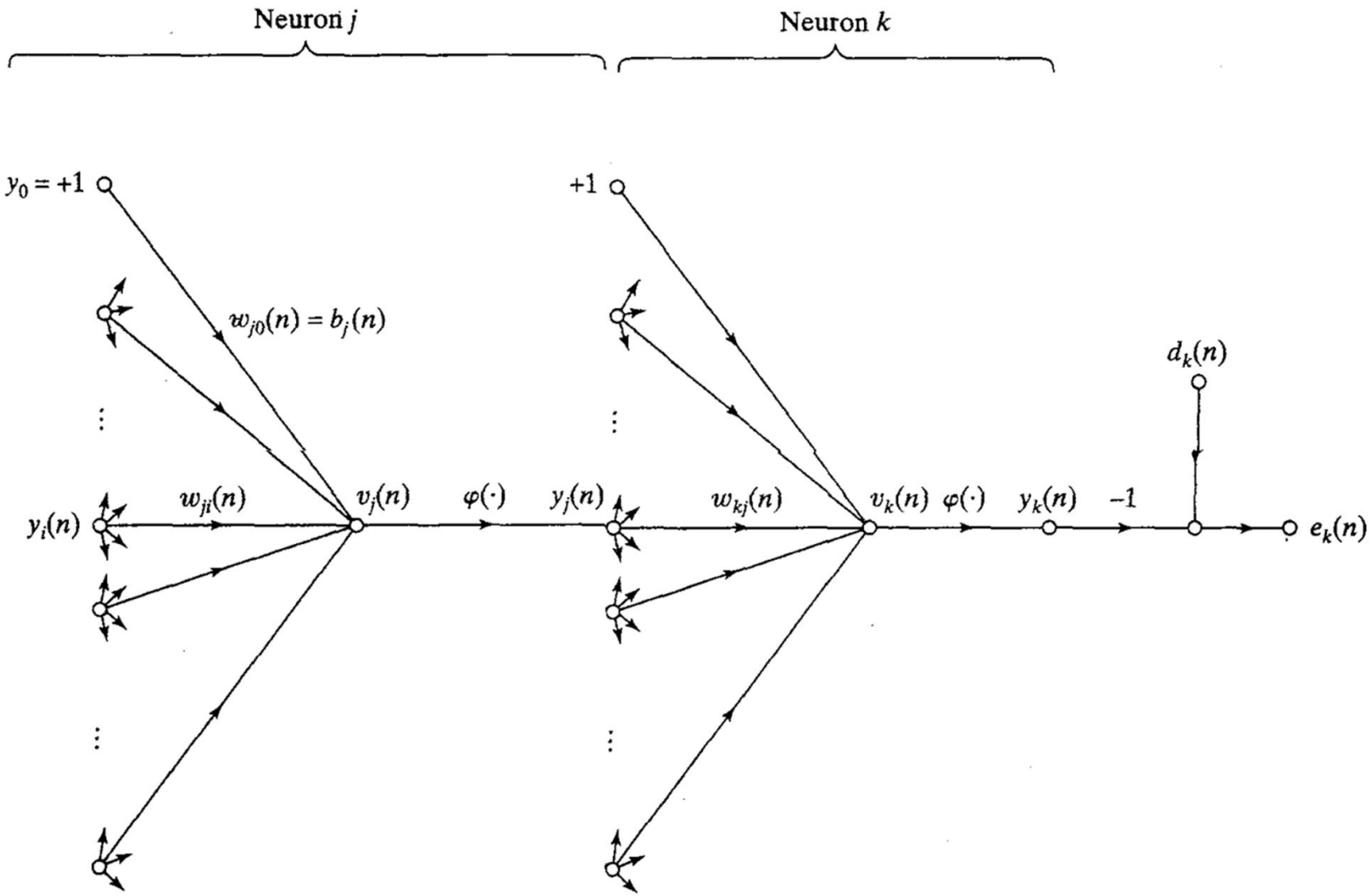
The correction $\Delta w_{ji}(n)$ applied to $w_{ji}(n)$ is defined by the *delta rule*:

$$\Delta w_{ji}(n) = -\eta \frac{\partial \mathscr{E}(n)}{\partial w_{ji}(n)}$$

$$\Delta w_{ji}(n) = \eta \delta_j(n) y_i(n)$$

$$\delta_j(n) = e_j(n)\varphi_j'(v_j(n))$$

Case 2 Neuron *j* Is a Hidden Node



$$\frac{\partial \mathcal{E}(n)}{\partial y_j(n)} = \sum_k e_k(n) \frac{\partial e_k(n)}{\partial v_k(n)} \frac{\partial v_k(n)}{\partial y_j(n)}$$

$$e_k(n) = d_k(n) - y_k(n)$$

$$= d_k(n) - \varphi_k(v_k(n)), \qquad \text{neuron } k \text{ is an output node}$$

$$\frac{\partial e_k(n)}{\partial v_k(n)} = -\varphi_k'(v_k(n))$$

$$v_k(n) = \sum_{j=0}^{m} w_{kj}(n) y_j(n) \qquad \frac{\partial v_k(n)}{\partial y_j(n)} = w_{kj}(n)$$

$$\frac{\partial \mathcal{E}(n)}{\partial y_j(n)} = -\sum_k e_k(n) \varphi_k'(v_k(n)) w_{kj}(n)$$

$$= -\sum_k \delta_k(n) w_{kj}(n)$$

$$\delta_j(n) = \varphi_j'(v_j(n)) \sum_k \delta_k(n) w_{kj}(n), \qquad \text{neuron } j \text{ is hidden} \tag{4.24}$$

$$\begin{pmatrix} \textit{Weight} \\ \textit{correction} \\ \Delta w_{ji}(n) \end{pmatrix} = \begin{pmatrix} \textit{learning-} \\ \textit{rate parameter} \\ \eta \end{pmatrix} \cdot \begin{pmatrix} \textit{local} \\ \textit{gradient} \\ \delta_j(n) \end{pmatrix} \cdot \begin{pmatrix} \textit{input signal} \\ \textit{of neuron } j \\ y_i(n) \end{pmatrix}$$

Second, the local gradient $\delta_j(n)$ depends on whether neuron j is an output node or a hidden node:

1. If neuron j is an output node, $\delta_j(n)$ equals the product of the derivative $\varphi_j'(v_j(n))$ and the error signal $e_j(n)$, both of which are associated with neuron j; see Eq.(4.14).
2. If neuron j is a hidden node, $\delta_j(n)$ equals the product of the associated derivative $\varphi_j'(v_j(n))$ and the weighted sum of the δs computed for the neurons in the next hidden or output layer that are connected to neuron i: see Eq. (4.24).

Role of Learning Rate

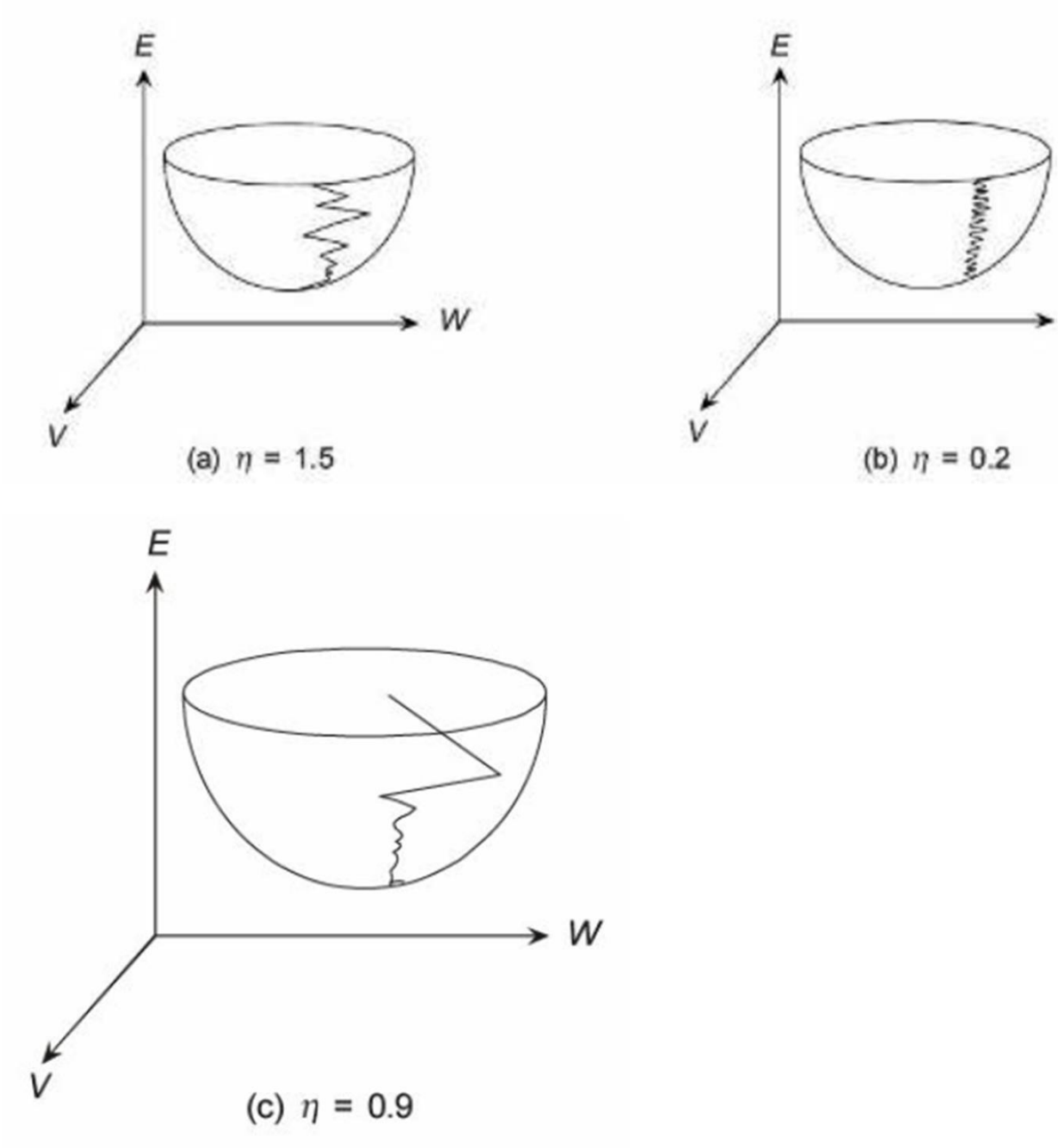
The effectiveness and convergence of Error backpropagation are based on the value of learning constant or learning rate η. The amount by which weights of network are updated is directly proportional to the learning rate η and hence it plays the important role in the convergence of training.

When we use a larger value of η, our network takes wider steps to reach global minima of error plot. Due to a larger value of η, there is a chance of missing global minima if error plot yields shorter global minima. Similarly if we use a smaller value of η, our network takes shorter steps to reach global minima of error plot but in this case, there is a chance of stuck in local minima of error plot.

A simple method of increasing the learning rate yet avoiding the danger of instability is to modify the delta rule of weight updation as:

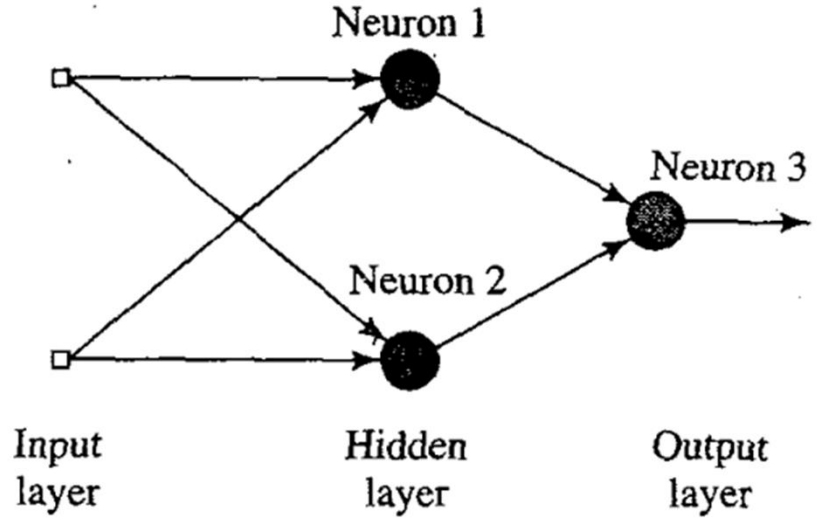
$$\Delta w_{ji}(n) = \alpha \Delta w_{ji}(n-1) + \eta \delta_j(n) y_i(n)$$

Where α is the momentum term whose value is taken between 0.5 and 0.9

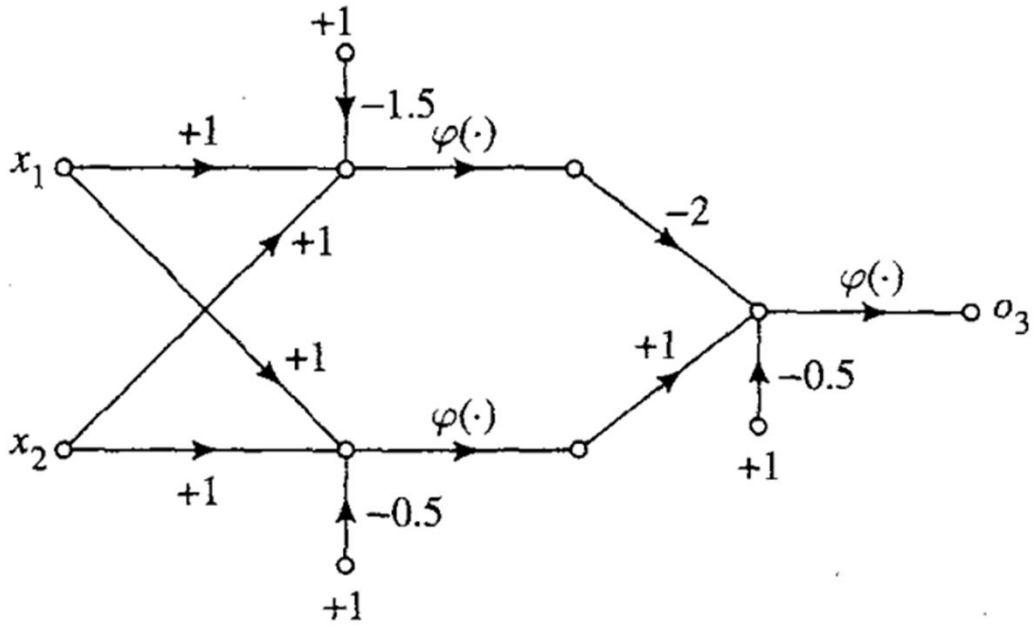


Convergence paths for different learning coefficients.

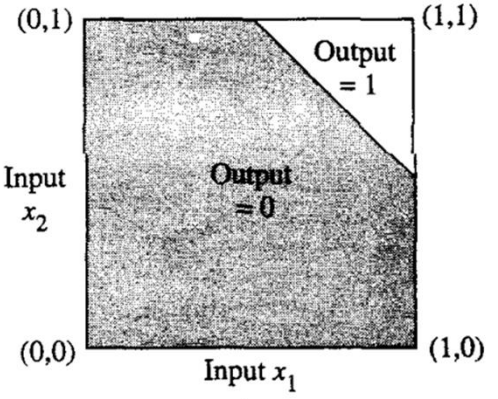
XOR PROBLEM



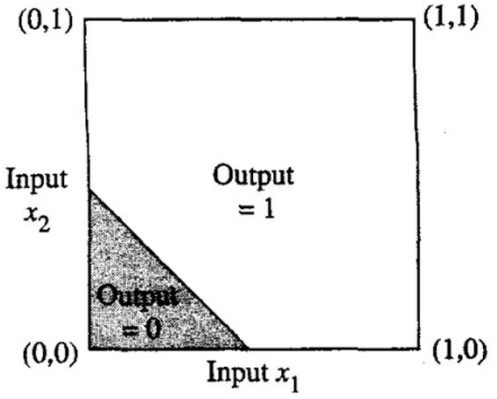
(a)



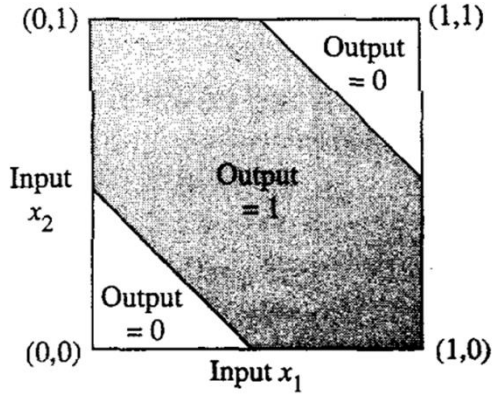
(a) Decision boundary constructed by hidden neuron 1 of the network in Fig. 4.8.
(b) Decision boundary constructed by hidden neuron 2 of the network.
(c) Decision boundaries constructed by the complete network.



(a)



(b)



(c)

Train the network using error back propagation learning algorithm