# *ARTIFICIAL NEURAL NETWORKS*
# *UNIT 4*

## FEED FORWARD ANN

In multilayer feed forward networks, the network consists of a set of sensory units (source nodes) that constitute the input layer, one or more hidden layers of computation nodes, and an output layer of computation nodes. The input signal propagates through the network in a forward direction, on a layer-by-layer basis. These neural networks are commonly referred to as multilayer perceptron (MLPs).

A multilayer perceptron has three distinctive characteristics:

1. The model of each neuron in the network includes a nonlinear activation function. A commonly used form of nonlinearity that satisfies this requirement is a sigmoidal nonlinearity defined by the logistic function:

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

where $V_j$ is the induced local field (i.e., the weighted sum of all synaptic inputs plus the bias) of neuron j, and $Y_j$ is the output of the neuron. The presence of nonlinearities is important because otherwise the input-output relation of the network could be reduced to that of a single-layer perceptron.

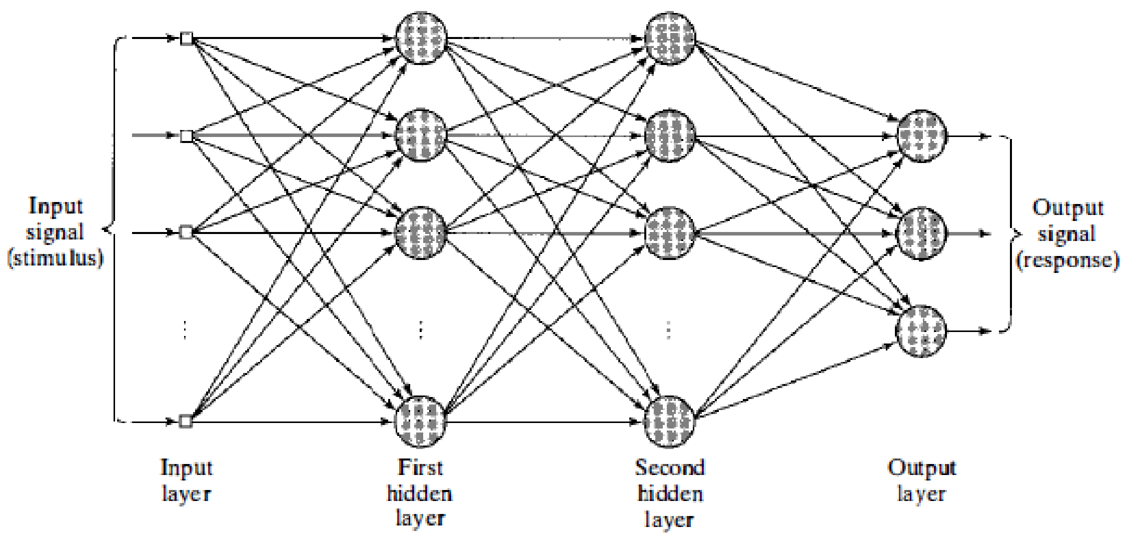
2. The network contains one or more layers of hidden neurons that are not part of the input or output of the network. These hidden neurons enable the network to learn complex tasks by extracting progressively more meaningful features from the input patterns (vectors).

3. The network exhibits high degrees of connectivity, determined by the synapses of the network. A change in the connectivity of the network requires a change in the population of synaptic connections or their weights.

## STRUCTURE OF MULTI-LAYER FEED FORWARD NETWORKS

A Multi-layer feed forward network consists of a layer of input units, one or more layer of hidden units and a layer of output units. The input signal propagates through network in a forward direction on a layer by layer basis. These are called feed forward because output of one layer of neurons feed forward onto next layer of neurons.

The following Figure shows the architectural graph of a multilayer perceptron with two hidden layers and an output layer. The network shown here is fully connected. This means that a neuron in any layer of the network is connected to all the nodes/neurons in the previous layer. Signal flow through the network progresses in a forward direction, from left to right and on a layer-by-layer basis.



Architectural graph of a multilayer perceptron with two hidden layers.

Two kinds of signals are identified in this network.

**1. Function Signals**. A function signal is an input signal (stimulus) that comes in at the input end of the network, propagates forward (neuron by neuron) through the network, and emerges at the output end of the network as an output signal. We refer to such a signal as a "function signal" for two reasons. First, it is presumed to perform a useful function at the output of the network. Second, at each neuron of the network through which a function signal passes, the signal is calculated as a function of the inputs and associated weights applied to that neuron.

The function signal is also referred to as the input signal.

**2. Error Signals.** An error signal originates at an output neuron of the network, and propagates backward (layer by layer) through the network. We refer to it as an "error signal" because its computation by every neuron of the network involves an error-dependent function in one form or another.

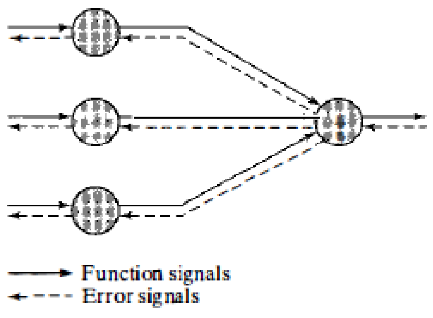


Illustration of the directions of two basic signal flows in a multilayer perceptron: forward propagation of function signals and back-propagation of error signals.

The output neurons (computational nodes) constitute the output layers of the network. The remaining neurons (computational nodes) constitute hidden layers of the network. Thus the hidden units are not part of the output or input of the network hence their designation as "hidden." The first hidden layer is fed from the input layer made up of sensory units (source nodes); the resulting outputs of the first hidden layer are in turn applied to the next hidden layer; and so on for the rest of the network.

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.

**BACK-PROPAGATION ALGORITHM**

Multilayer perceptron's have been applied successfully to solve some difficult and diverse problems by training them in a supervised manner with a highly popular algorithm known as the error back-propagation algorithm. This algorithm is based on the error correction learning rule.

The summary of the notations used in the algorithm derivation are as follows.

- The indices $i, j$, and $k$ refer to different neurons in the network; with signals propagating through the network from left to right, neuron $j$ lies in a layer to the right of neuron $i$, and neuron $k$ lies in a layer to the right of neuron $j$ when neuron $j$ is a hidden unit.
- In iteration (time step) $n$, the $n$th training pattern (example) is presented to the network.
- The symbol $\mathscr{E}(n)$ refers to the instantaneous sum of error squares or error energy at iteration $n$. The average of $\mathscr{E}(n)$ over all values of $n$ (i.e., the entire training set) yields the average error energy $\mathscr{E}_{av}$.
- The symbol $e_j(n)$ refers to the error signal at the output of neuron $j$ for iteration $n$.
- The symbol $d_j(n)$ refers to the desired response for neuron $j$ and is used to compute $e_j(n)$.
- The symbol $y_j(n)$ refers to the function signal appearing at the output of neuron $j$ at iteration $n$.
- The symbol $w_{ji}(n)$ denotes the synaptic weight connecting the output of neuron $i$ to the input of neuron $j$ at iteration $n$. The correction applied to this weight at iteration $n$ is denoted by $\Delta w_{ji}(n)$.

- The induced local field (i.e., weighted sum of all synaptic inputs plus bias) of neuron $j$ at iteration $n$ is denoted by $v_j(n)$; it constitutes the signal applied to the activation function associated with neuron $j$.
- The activation function describing the input–output functional relationship of the nonlinearity associated with neuron $j$ is denoted by $\varphi_j(\cdot)$.
- The bias applied to neuron $j$ is denoted by $b_j$; its effect is represented by a synapse of weight $w_{j0} = b_j$ connected to a fixed input equal to $+1$.
- The $i$th element of the input vector (pattern) is denoted by $x_i(n)$.
- The $k$th element of the overall output vector (pattern) is denoted by $o_k(n)$.
- The learning-rate parameter is denoted by $\eta$.
- The symbol $m_l$ denotes the size (i.e., number of nodes) in layer $l$ of the multilayer perceptron; $l = 0, 1, \ldots, L$, where $L$ is the "depth" of the network. Thus $m_0$ denotes the size of the input layer, $m_1$ denotes the size of the first hidden layer, and $m_L$ denotes the size of the output layer. The notation $m_L = M$ is also used.

The error signal at the output of neuron j at iteration n is defined by

$$e_j(n) = d_j(n) - y_j(n), \text{ ------- (1)} \qquad \text{neuron j is an output node}$$

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

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

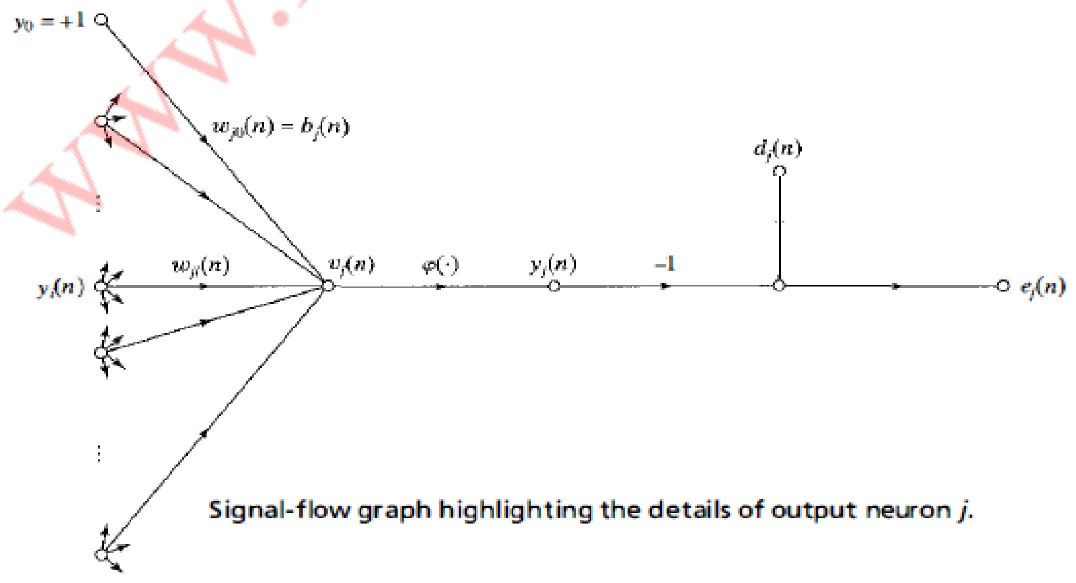
where the set C includes all the neurons in the output layer of the network. Let N denote the total number of patterns contained in the training set. The average squared error energy is obtained by summing $\mathscr{E}(n)$ over all n and then normalizing with respect to the set size N, as shown by

$$\mathscr{E}_{av} = \frac{1}{N}\sum_{n=1}^{N} \mathscr{E}(n) \text{ ------- (3)}$$

The instantaneous error energy $\mathscr{E}(n)$, and therefore the average error energy $\mathscr{E}_{av}$ is a function of all the free parameters (i.e., synaptic weights and bias levels) of the network. For a given training set, $\mathscr{E}_{av}$ represents the cost function as a measure of learning performance. The objective of the learning process is to adjust the free parameters of the network to minimize $\mathscr{E}_{av}$ .

Consider the following Figure which depicts neuron j being fed by a set of function signals produced by a layer of neurons to its left. The induced local field $v_j(n)$ produced at the input of the activation function associated with neuron j is therefore

$$v_j(n) = \sum_{i=0}^{m} w_{ji}(n)y_i(n) \text{ ------- (4)}$$



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

where m is the total number of inputs. The synaptic weight Wj0 (corresponding to the fixed input Yo = + 1) equals the bias bj applied to neuron).

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

$$y_j(n) = \varphi_j(v_j(n)) \quad \text{------- (5)}$$

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:

$$\frac{\partial\mathscr{E}(n)}{\partial w_{ji}(n)} = \frac{\partial\mathscr{E}(n)}{\partial e_j(n)}\frac{\partial e_j(n)}{\partial y_j(n)}\frac{\partial y_j(n)}{\partial v_j(n)}\frac{\partial v_j(n)}{\partial w_{ji}(n)} \quad \text{------- (6)}$$

The partial derivative $\partial\mathscr{E}(n)/\partial w_{ji}(n)$represents a sensitivity factor, determining the direction of search in weight space for the synaptic weight Wji.

Differentiating both sides of Eq. (2) with respect to ej(n). we get

$$\frac{\partial\mathscr{E}(n)}{\partial e_j(n)} = e_j(n) \quad \text{------- (7)}$$

Differentiating both sides of Eq. (1) with respect to yj(n), we get

$$\frac{\partial e_j(n)}{\partial y_j(n)} = -1 \quad \text{------- (8)}$$

Next, differentiating Eq. (5) with respect to vj(n), we get

$$\frac{\partial y_j(n)}{\partial v_j(n)} = \varphi_j'(v_j(n)) \quad \text{------- (9)}$$

where the use of prime signifies differentiation with respect to the argument.

Finally, differentiating Eq. (4) with respect to wj(n) yields

$$\frac{\partial v_j(n)}{\partial w_{ji}(n)} = y_i(n) \quad \text{------- (10)}$$

The use of Eqs. (7) to (10) in (6) yields

$$\frac{\partial\mathscr{E}(n)}{\partial w_{ji}(n)} = -e_j(n)\varphi_j'(v_j(n))y_i(n) \quad \text{------- (11)}$$

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

$$\Delta w_{ji}(n) = -\eta\frac{\partial\mathscr{E}(n)}{\partial w_{ji}(n)} \quad \text{------- (12)}$$

where *n* is the learning-rate parameter of the back-propagation algorithm. The use of the minus sign in Eq. (12) accounts for gradient descent in weight space.

Accordingly, the use of Eq. (11) in (12) yields

$$\Delta w_{ji}(n) = \eta\delta_j(n)y_i(n) \quad \text{------- (13)}$$

where the local gradient $\delta_j(n)$ is defined by

$$\begin{aligned}\delta_j(n) &= -\frac{\partial \mathscr{E}(n)}{\partial v_j(n)} \\ &= -\frac{\partial \mathscr{E}(n)}{\partial e_j(n)}\frac{\partial e_j(n)}{\partial y_j(n)}\frac{\partial y_j(n)}{\partial v_j(n)} \\ &= e_j(n)\varphi_j'(v_j(n))\end{aligned} \qquad \text{------- (14)}$$

The local gradient points to required changes in synaptic weights. According to Eq. (14), the local gradient $\delta_j(n)$ for output neuron j is equal to the product of the corresponding error signal ej(n) for that neuron and the derivative $\varphi_j'(v_j(n))$ of the associated activation function.

From Eqs. (13) and (14) note that a key factor involved in the calculation of the weight adjustment $\Delta w_{ji}(n)$ is the error signal ej(n) at the output of neuron j. In this context we may identify two distinct cases depending on where in the network neuron j is located.

Case 1 Neuron j Is an Output Node

Case 2 Neuron j Is a Hidden Node

**Case 1 Neuron j Is an Output Node**

When neuron j is located in the output layer of the network, it is supplied with a desired response of its own. We may use Eq. (1) to compute the error signal ej(n) associated with this neuron; see Fig. Having determined ej(n), it is a straightforward matter to compute the local gradient $\delta_j(n)$ using Eq. (14).

**Case 2 Neuron j Is a Hidden Node**

When neuron j is located in a hidden layer of the network there is no specified desired response for that neuron. Accordingly, the error signal for a hidden neuron would have to be determined recursively in terms of the error signals of all the neurons to which that hidden neuron is directly connected; this is where the development of the back propagation algorithm gets complicated. Consider the situation depicted in following Figure which depicts neuron j as a hidden node of the network. According to Eq. (14), we may redefine the local gradient $\delta_j(n)$ for hidden neuron j as

$$\begin{aligned}\delta_j(n) &= -\frac{\partial \mathscr{E}(n)}{\partial y_j(n)}\frac{\partial y_j(n)}{\partial v_j(n)} \\ &= -\frac{\partial \mathscr{E}(n)}{\partial y_j(n)}\varphi_j'(v_j(n)), \qquad \text{neuron } j \text{ is hidden}\end{aligned} \qquad \text{------- (15)}$$

where in the second line we have used Eq. (9). To calculate the partial derivative $\partial \mathscr{E}(n)/\partial w_{ji}(n)$ we may proceed as follows. From Fig. 4.4 we see that

$$\mathscr{E}(n) = \frac{1}{2}\sum_{k\in C} e_k^2(n), \qquad \text{neuron } k \text{ is an output node} \qquad \text{------- (16)}$$

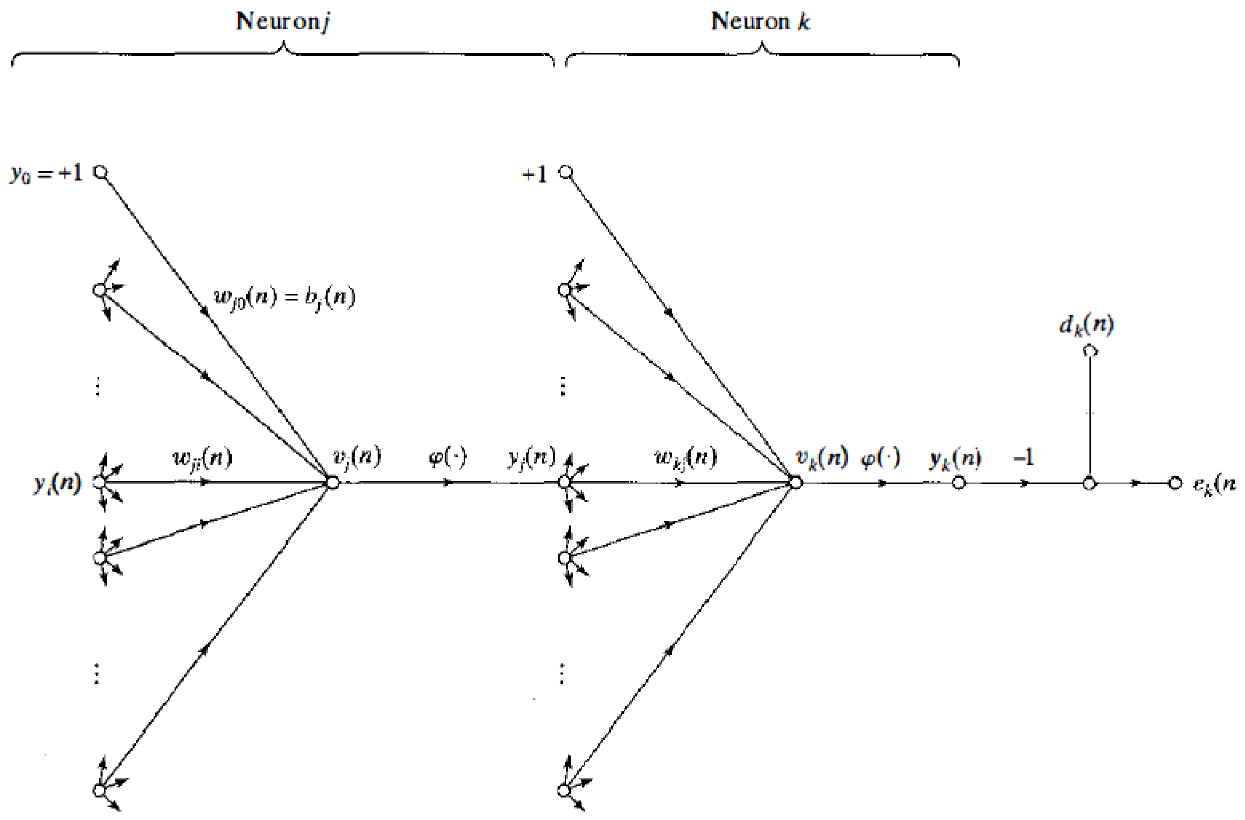


**FIGURE 4.4** Signal-flow graph highlighting the details of output neuron $k$ connected to hidde neuron $j$.

Differentiating Eq. (16) with respect to the function signal $y_j(n)$, we get

$$\frac{\partial \mathscr{E}(n)}{\partial y_j(n)} = \sum_k e_k \frac{\partial e_k(n)}{\partial y_j(n)} \quad \text{------- (17)}$$

Next we use the chain rule for the partial derivative $\partial \mathscr{E}(n)/\partial w_{ji}(n)$. and rewrite Eq. (17) in the equivalent form

$$\frac{\partial \mathscr{E}(n)}{\partial y_j(n)} = \sum_k e_k(n) \frac{\partial e_k(n)}{\partial v_k(n)} \frac{\partial v_k(n)}{\partial y_j(n)} \quad \text{------- (18)}$$

However. from above Fig.. we note that

$$\begin{aligned} e_k(n) &= d_k(n) - y_k(n) \\ &= d_k(n) - \varphi_k(v_k(n)), \end{aligned} \quad \text{------- (19)}$$

Hence

$$\frac{\partial e_k(n)}{\partial v_k(n)} = -\varphi_k'(v_k(n)) \quad \text{------- (20)}$$

We also note from above Figure that for neuron k the induced local field is

$$v_k(n) = \sum_{j=0}^{m} w_{kj}(n) y_j(n) \quad \text{------- (21)}$$

where m is the total number of inputs applied to neuron k.

Differentiating Eq. (21) with respect to Yj(n) yields

$$\frac{\partial v_k(n)}{\partial y_j(n)} = w_{kj}(n) \qquad \text{-------}\ (22)$$

By using Eqs. (20) and (22) in (18) we get the desired partial derivative:

$$\begin{aligned}\frac{\partial \mathscr{E}(n)}{\partial y_j(n)} &= -\sum_k e_k(n)\varphi_k'(v_k(n))\,w_{kj}(n)\\ &= -\sum_k \delta_k(n) w_{kj}(n) \qquad \text{-------}\ (23)\end{aligned}$$

Where in the second line we have used the definition of the local gradient $\delta_j(n)$ given in Eq. (14) with the index k substituted for j.

$$\delta_j(n) = \varphi_j'(v_j(n))\sum_k \delta_k(n) w_{kj}(n), \qquad \text{-------}\ (24)$$

The factor $\varphi_j'(v_j(n))$ involved in the computation of the local gradient $\delta_j(n)$ in Eq. (24) depends solely on the activation function associated with hidden neuron j.

We now summarize the relations that we have derived for the back-propagation algorithm.

1. The correction $\Delta w_{ji}(n)$ applied to the synaptic weight connecting neuron i to neuron j is defined by the delta rule:

$$\begin{pmatrix}\textit{Weight}\\ \textit{correction}\\ \Delta w_{ji}(n)\end{pmatrix} = \begin{pmatrix}\textit{learning-}\\ \textit{rate parameter}\\ \eta\end{pmatrix}\cdot\begin{pmatrix}\textit{local}\\ \textit{gradient}\\ \delta_j(n)\end{pmatrix}\cdot\begin{pmatrix}\textit{input signal}\\ \textit{of neuron } j\\ y_i(n)\end{pmatrix} \qquad \text{-------}\ (25)$$

2. The local gradient $\delta_j(n)$ depends on whether neuron j is an output node or a hidden node:

- 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 ej(n), both of which are associated with neuron j; see Eq.(14).
- If neuronj 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 os computed for the neurons in the next hidden or output layer that are connected to neuron j; see Eq. (24).

**The Two Passes of Computation**

In the application of the back-propagation algorithm, two distinct passes of computation are distinguished. The first pass is referred to as the forward pass, and the second is referred to as the backward pass

- In the forward pass the synaptic weights remain unaltered throughout the network, and the function signals of the network are computed on a neuron-by-neuron basis. The function signal appearing at the output of neuron j is computed as

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

where vj(n) is the induced local field of neuron j, defined by

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

where m is the total number of inputs applied to neuron j, and Wji (n) is the synaptic weight connecting neuron i to neuron j, and yi(n) is the input signal of neuron j or equivalently, the function signal appearing at the output of neuron i.

→The forward phase of computation begins at the first hidden layer by presenting $i_{th}$ with the input vector, and terminates at the output layer by computing the error signal for each neuron of this layer.

- The backward pass starts at the output layer by passing the error signals leftward through the network, layer by layer, and recursively computing the 0 (i.e., the local gradient) for each neuron. This recursive process permits the synaptic weights of the network to undergo changes in accordance with the delta rule of Eq.(25).

**Activation Function**

The computation of the $\delta$ for each neuron of the multilayer perceptron requires knowledge of the derivative of the activation function $\varphi(\cdot)$ associated with that neuron. For this derivative to exist, we require the function $\varphi(\cdot)$ to be continuous. The commonly used nonlinear activation function in multilayer perceptrons is sigmoidal nonlinearity;

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

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

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 $0 \leq y_i \leq 1$ .

Differentiating Eq. (1) with respect to $v_j(n)$ , we get

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

With $y_i(n) = \varphi_i(v_i(n))$,we may eliminate the exponential term $\exp(-av_j(n))$ from Eq. (2), and so express the derivative $\varphi_j'(v_j(n))$as

$$\varphi_j'(v_j(n)) = ay_j(n)[1 - y_j(n)] \quad \text{-----------(3)}$$

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

$$\begin{aligned}\delta_j(n) &= e_j(n)\varphi_j'(v_j(n)) \\ &= a[d_j(n) - o_j(n)]o_j(n)[1 - o_j(n)], \qquad \text{neuron } j \text{ is an output node} \quad \text{-----------(4)}\end{aligned}$$

where $o_j(n)$ is the function signal at the output of neuron j, and $d_j(n)$ is the desired response for it. On the other hand, for an arbitrary hidden neuron j, we may express the local gradient as

$$\delta_j(n) = \varphi_j'(v_j(n)) \sum_k \delta_k(n) w_{kj}(n)$$

$$= a y_j(n)[1 - y_j(n)] \sum_k \delta_k(n) w_{kj}(n), \qquad \text{neuron } j \text{ is hidden} \quad \text{-----------} \ (5)$$

Note from Eq. (3) that the derivative $\varphi_j'(v_j(n))$ attains its maximum value at $y_j(n)$ = 0.5, and its minimum value (zero) at $y_j(n)$ = 0, or $y_j(n)$ = 1.0. Since the amount of change in a synaptic weight of the network is proportional to the derivative $\varphi_j'(v_j(n))$.

**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

$$\varphi_j(v_j(n)) = a \tanh(b v_j(n)), \qquad (a,b) > 0$$

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 vj(n) is given by

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

For a neuron j located in the output layer, the local gradient is

$$\begin{aligned}\delta_j(n) &= e_j(n)\varphi_j'(v_j(n)) \\ &= \frac{b}{a}[d_j(n) - o_j(n)][a - o_j(n)][a + o_j(n)]\end{aligned}$$

For a neuronj in a hidden layer, we have

$$\begin{aligned}\delta_j(n) &= \varphi_j'(v_j(n)) \sum_k \delta_k(n) w_{kj}(n) \\ &= \frac{b}{a}[a - y_j(n)][a + y_j(n)] \sum_k \delta_k(n) w_{kj}(n), \qquad \text{neuron } j \text{ is hidden}\end{aligned}$$

## BACK PROPAGATION TRAINING AND CONVERGENCE

For a given training set, back -propagation learning may thus proceed in one of two basic ways:

**1. Sequential Mode.**

The sequential mode of back-propagation learning is also referred to as on-line, pattern, or stochastic mode. In this mode of operation weight updating is performed after the presentation of each training example.

One complete presentation of the entire training set during the learning process is called an epoch. The learning process is maintained on an epoch-by-epoch basis until the synaptic weights and bias levels of the network stabilize and the average squared error over the entire training set converges to some minimum value.

To be specific, consider an epoch consisting of N training examples (patterns) arranged in the order (x(l), d(l)) . . . . , (x(N), d(N)). The first example pair (x(1), d(l)) in the epoch is presented to the network, and the sequence of forward and backward computations described previously is performed, resulting in certain adjustments to the synaptic weights and bias levels of the network. Then the second example pair (x(2), d(2)) in the epoch is presented, and the sequence of torward and backward computations is repeated, resulting in further adjustments to the synaptic weights and bias levels. This process is continued until the last example pair (x(N), d(N)) in the epoch is accounted for.

**2. Batch Mode.**

In the batch mode of back-propagation learning, weight updating is performed after the presentation of all the training examples that constitute an epoch. For a particular epoch, we define the cost function as the average squared error

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

.

where the error signal $e_j(n)$ pertains to output neuron j for training example n. The error $e_j(n)$ equals the difference between $d_j(n)$ and $y_j(n)$, which represents the $j_{th}$ element of the desired response vector $d_j(n)$ and the corresponding value of the network output, respectively.

Here the inner summation with respect to j is performed over all the neurons in the output layer of the network, whereas the outer summation with respect to n is performed over the entire training set in the epoch at hand.

For a learning-rate parameter 11, the adjustment applied to synaptic weight $w_{ji}$ connecting neuron i to neuron j, is defined by the delta rule

$$\begin{aligned}\Delta w_{ji} &= -\eta\frac{\partial \mathscr{E}_{\text{av}}}{\partial w_{ji}}\\ &= -\frac{\eta}{N}\sum_{n=1}^{N} e_j(n)\frac{\partial e_j(n)}{\partial w_{ji}}\end{aligned}$$

To calculate the partial derivative $\partial e_j(n)/\partial w_{ji}$ we proceed in the same way as before. According to above Eq. in the batch mode the weight adjustment $\Delta w_{ji}(n)$ is made only after the entire training set has been presented to the network.

In general, the back -propagation algorithm cannot be shown to converge, and there are no well-defined criteria for stopping its operation. But there are some reasonable criteriae used to terminate the weight adjustments.

To formulate such a criterion, it is logical to think in terms of the unique properties of a local or global minimum of the error surface•

→Let the weight vector w* denote a minimum, be it local or global. A necessary condition for w* to be a minimum is that the gradient vector g(w) of the error surface with respect to the weight vector w be zero at w = w*.

***The back-propagation algorithm is considered to have converged when the Euclidean norm of the gradient vector reaches a sufficiently small gradient threshold.***

The drawback of this convergence criterion is learning times may be long. Also, it requires the computation of the gradient vector g( w).

→Another unique property of a minimum that we can use is the fact that the cost function or error measur $\mathcal{E}_{av}$ (w) is stationary at the point w = w*. We may therefore suggest a different criterion of convergence:

***The back-propagation algorithm is considered to have converged when the absolute rate of change in the average squared error per epoch is sufficiently small.***

The rate of change in the average squared error is typically considered to be small enough if it lies in the range of 0.1 to 1 percent per epoch. Sometimes a value as small as 0.01 percent per epoch is used. Unfortunately, this criterion may result in a premature termination of the learning process

## APPROXIMATIONS OF FUNCTIONS

A multilayer perceptron trained with the back-propagation algorithm may be viewed as a practical vehicle for performing a nonlinear input-output mapping of a general nature. To be specific, let $m_o$ denote the number of input (source) nodes of a multilayer perceptron, and let M = $m_L$ denote the number of neurons in the output layer of the network. The input-output relationship of the network defines a mapping from an $m_0$ dimensional Euclidean input space to an M-dimensional Euclidean output space. In assessing the capability of the multilayer perceptron from this viewpoint of input-output mapping, the following fundamental question arises:

*what is the minimum number of hidden layers in a multilayer perceptron with an input-output mapping that provides an approximate realization of any continuous mapping?*

## Universal Approximation Theorem

Let $\varphi(\cdot)$ be a non-constant, bounded, and monotone-increasing continuous function. Let $I_{m_0}$. denote the $m_0$-dimensional unit hypercube $[0,1]^{m_0}$. The space of continuous functions on $I_{m_0}$ is denoted by $C(I_{m_0})$. Then, given any function $f \ni C(I_{m_0})$ *and* $\epsilon > 0$, there exist an integer M and sets of real constants $\alpha_i, b_i$ and $w_{ij}$> where $i = 1, \ldots, m_1$ *and* $j = 1, \ldots, m_0$ m() such that we may define

$$F(x_1, \ldots, x_{m_0}) = \sum_{i=1}^{m_1} \alpha_i \varphi\left(\sum_{j=1}^{m_0} w_{ij} x_j + b_i\right)$$

as an approximate realization of the function f( . ); that is,

$$|F(x_1, \ldots, x_{m_0}) - f(x_1, \ldots, x_{m_0})| < \epsilon$$

for all $x_1, x_2, \ldots, x_{m_0}$ that lie in the input space.
The universal approximation theorem is directly applicable to multilayer perceptrons. We first note that the logistic function 1/[1 + exp( -v)] used as the nonlinearity in a neuronal model for the construction of a multilayer perceptron is indeed a nonconstant, bounded, and monotone-increasing function; The above equation represents the output of a multilayer perceptron described as follows:
1. The network has m D input nodes and a single hidden layer consisting of $m_1$ neurons the inputs are denoted by $x_1, x_2, \ldots, x_{m_0}$
2. Hidden neuron i has synaptic weights $w_{i_1}, \ldots, w_{m_0}$ and bias $b_i$
3. The network output is a linear combination of the outputs of the hidden neurons, with $\alpha_1, \ldots, \alpha_{m_1}$, defining the synaptic weights of the output layer.
The universal approximation theorem is an existence theorem in the sense that it
provides the mathematical justification for the approximation of an arbitrary continuous function as opposed to exact representation.

**Bounds on Approximation Errors**

Barron (1993) has established the approximation properties of a multilayer perceptron, assuming that the network has a single layer of hidden neurons using sigmoid functions and a linear output neuron. The network is trained using the back-propagation algorithm and then tested with new data. During training, the network learns specific points of a target function f in accordance with the training data.

A smoothness property of the target function f is expressed in terms of its Fourier representation. In particular, the average of the norm of the frequency vector weighted by the Fourier magnitude distribution is used as a measure for the extent to which the function f oscillates. Let $\tilde{f}(\omega)$ denote the multidimensional Fourier transform of the function $f(\mathbf{x}), \mathbf{x} \in \mathbb{R}^{m_0}$ the $m_0$-by-1 vector $\boldsymbol{\omega}$ is the frequency vector. The function f(x) is defined in terms of its Fourier transform $\tilde{f}(\boldsymbol{\omega})$ by the inverse formula:

$$f(x) = \int_{\mathbb{R}^{m_0}} \tilde{f}(\boldsymbol{\omega}) \exp(j\boldsymbol{\omega}^T \mathbf{x})\, d\boldsymbol{\omega}$$

where $j = \sqrt{-1}$. For the complex-valued function $\tilde{f}(\boldsymbol{\omega})$ for which $\boldsymbol{\omega}\tilde{f}(\boldsymbol{\omega})$ is integrable, we define the first absolute moment of the Fourier magnitude distribution of the function f as:

$$C_f = \int_{\mathbb{R}^{m_0}} \tilde{f}(\boldsymbol{\omega})| \times \|\boldsymbol{\omega}\|^{1/2}\, d\boldsymbol{\omega}$$

$\tilde{f}(\boldsymbol{\omega})$. The first absolute moment $C_f$ quantifies the smoothness or regularity of the function f.

The first absolute moment $C_f$ provides the basis for a bound on the error that results from the use of a multilayer perceptron represented by the input-output mapping function F(x) to approximate f(x). The approximation error is measuredby the integrated squared error with respect to an arbitrary probability measure $\mu$ on the ball $B_r = \{\mathbf{x}: \|\mathbf{x}\| \le r\}$ of radius r > 0. On this basis we may state the following proposition for a bound on the approximation error due to Barron (1993):

For every continuous function f(x) with first moment $C_f$ finite, and every m1 2: 1, there exists a linear combination of sigmoid functions F(x) of the form defined in Eq. (4.86), such that

$$\int_{B_r} (f(\mathbf{x}) - F(\mathbf{x}))^2 \mu(d\mathbf{x}) \le \frac{C_f'}{m_1}$$

where $C_f' = (2\, r\, C_f)^2$.

When the function f(x) is observed at a set of values of the input vector x denoted by $\{\mathbf{x}_i\}_{i=1}^{N}$ that are restricted to lie inside the ban B" the result provides the following $B_r$ bound on the empirical risk:

$$R = \frac{1}{N} \sum_{i=1}^{N} (f(\mathbf{x}_i) - F(\mathbf{x}_i))^2 \le \frac{C_f'}{m_1}$$

The bound on the risk R resulting from the use of a multilayer perceptron with rno input nodes and $m_1$ hidden neurons as follows:

$$R \le O\left(\frac{C_f^2}{m_1}\right) + O\left(\frac{m_0 m_1}{N} \log N\right)$$

The two terms in the bound on the risk R express the tradeoff between two conflicting requirements on the size of the hidden layer:

1. Accuracy of best approximation. For this requirement to be satisfied, $m_1$, the size of the hidden layer, must be large in accordance with the universal approximation theorem.
2. Accuracy of empirical fit to the approximation. To satisfy this second requirement, we must use a small $m_1/N$ ratio md N. For a fixed size of training sample, N, the size of the hidden layer, m" should be kept small, which is in conflict with the first requirement.

**Curse of Dimensionality**

When the size of the hidden layer is optimized (i.e., the risk R is minimized with respect to N) by setting

$$m_1 \simeq C_f\left(\frac{N}{m_0 \log N}\right)^{1/2}$$

then the risk R is bounded by $O(C_f\sqrt{m_0(\log N/N)})$.

A surprising aspect of this result is that in terms of the first-order behavior of the risk R, the rate of convergence expressed as a function of the training sample size N is of order $(1/N)^{2s/(2s+m_0)}$.

Let s denote a measure of smoothness, defined as the number of continuous derivatives of a function of interest. Then, for traditional smooth functions we find that the minimax rate of convergence of the total risk R is of order $(1/N)^{2s/(2s+m_0)}$. The dependence of this rate on the dimensionality of the input space, mo, is a curse of dimensionality, which severely restricts the practical application of these functions. The use of a multilayer perceptron for function approximation appears to offer an advantage over traditional smooth functions; this advantage is, however, subject to the condition that the first absolute moment $C_f$ remains finite; this is a smoothness constraint.

The curse of dimensionality was introduced by Richard Bellman in his studies of adaptive control processes (Bellman, 1961). For a geometric interpretation of this notion, let x denote an ma·dimensional input vector, and $\{(\mathbf{x}_i, d_i)\}, i = 1, 2, \ldots, N$, denote the training sample. The sampling density is proportional to $N^{1/m0}$. Let a function f(x) represent a surface lying in the ma-dimensional input space, which passes near the data points $\{(\mathbf{x}_i, d_i)\}_{i=1}^{N}$,. Now, if the function f(x) is arbitrarily complex and completely unknown, we need dense sample (data) points to learn it well.

A function defined in high-dimensional space is likely to be much more complex than a function defined in a lower-dimensional space, and those complications are harder to discern.
The only practical way to beat the curse of dimensionality is to incorporate prior knowledge about the function over and above the training data, which is known to be correct.

## <u>PRACTICAL AND DESIGN ISSUES OF BACK PROPAGATION LEARNING</u>

The universal approximation theorem is important from a theoretical viewpoint, because it provides the necessary mathematical tool for the viability of feed forward networks with a single hidden layer as a class of approximate solutions. Without such theorem, we could conceivably be searching for a solution that cannot exist. However, the theorem is not constructive, that is, it does not actually specify how to determine a multilayer perceptron with the stated approximation properties.
The universal approximation theorem assumes that the continuous function to be approximated is given and that a hidden layer of unlimited size is available for the approximation. Both of these assumptions are violated in most practical applications of multilayer perceptrons. The problem with multilayer perceptrons using a single hidden layer is that the neurons there in tend to interact with each other globally. In complex situations this interaction makes it difficult to improve the approximation at one point without worsening it at some other point.
In particular, we may proceed as follows:
1. Local features are extracted in the first hidden layer. Specifically, some neurons in the first hidden layer are used to partition the input space into regions, and other neurons in that layer learn the local features characterizing those regions.
2. Global features are extracted in the second hidden layer. Specifically, a neuron in the second hidden layer combines the outputs of neurons in the first hidden layer operating on a particular region of the input space, and thereby learns the global features for that region and outputs zero elsewhere.
Sontag (1992) provides further justification for the use of two hidden layers in the context of inverse problems. Specifically, the fOllowing inverse problem is considered: Given a continuous vector-valued function $\mathbf{f}: \mathbb{R}^m \rightarrow \mathbb{R}^M$, a compact subset $\mathscr{C} \subseteq \mathbb{R}^M$ that is included in the image of f, and an $\epsilon > 0$, find a vector -valued function $\varphi: \mathbb{R}^M \rightarrow \mathbb{R}^m$ such that the following condition is satisfied:

$$\|\Phi(\mathbf{x}(n)) - \mathbf{u}\| < \epsilon \qquad \text{for } \mathbf{u} \in \mathscr{C}$$

This problem arises in inverse kinematics (dynamics), where the observed state x(n) of a system is a function of current actions $\mathbf{u}(n)$ and the previous state x(n - 1) of the system, as shown by

$$\mathbf{x}(n) = \mathbf{f}(\mathbf{x}(n-1), \mathbf{u}(n))$$

It is assumed that C is invertible, so that we may solve for $\mathbf{u}(n)$as a function of x(n) for any x(n - 1). The function C represents the direct kinematics, whereas the function φ represents the inverse kinematics. In practical terms, the motivation is to find a function φ that is computable by a multilayer perceptron. In general, discontinuous functions φ are needed to solve the inverse kinematics problem. It is interesting that even if the use of neuronal models with discontinuous activation functions is permitted, one hidden layer is not enough to guarantee the solution of all such inverse problems, whereas multilayer perceptrons with two hidden layers are sufficient for every possible $\mathbf{f}$, $\mathscr{C}$, and $\epsilon$

$\mathscr{E}(n)$ $\mathscr{E}_{\text{av}}$ $v_j(n)$ $y_j(n)$ $\partial\mathscr{E}(n)/\partial w_{ji}(n)$ $\delta_j(n)$ $\Delta w_{ji}(n)$ $w_{ji}(n)$ $\varphi_j'(v_j(n))$ $\delta$ $\varphi(\cdot)$