7. Multilayer Perceptrons Neural Networks

7.1 Introduction

In this part, we are to study the multilayer feed-forward network, which is an important class of neural networks. Typically, the network consists of a set of inputs (source nodes) that constitute the input layer, one or more hidden layers of computational nodes, and an output layer of computational nodes. The input signals propagates through the network in a forward direction, on a layer-by-layer basis. These neural networks are commonly referred to as multilayer perceptrons (MLP).

A MLP neural network has three distinctive characteristics:

(1) The model of each neuron in the network includes a nonlinear activation function. The important point here is that the nonlinearity is smooth. Sigmoid function is often used.

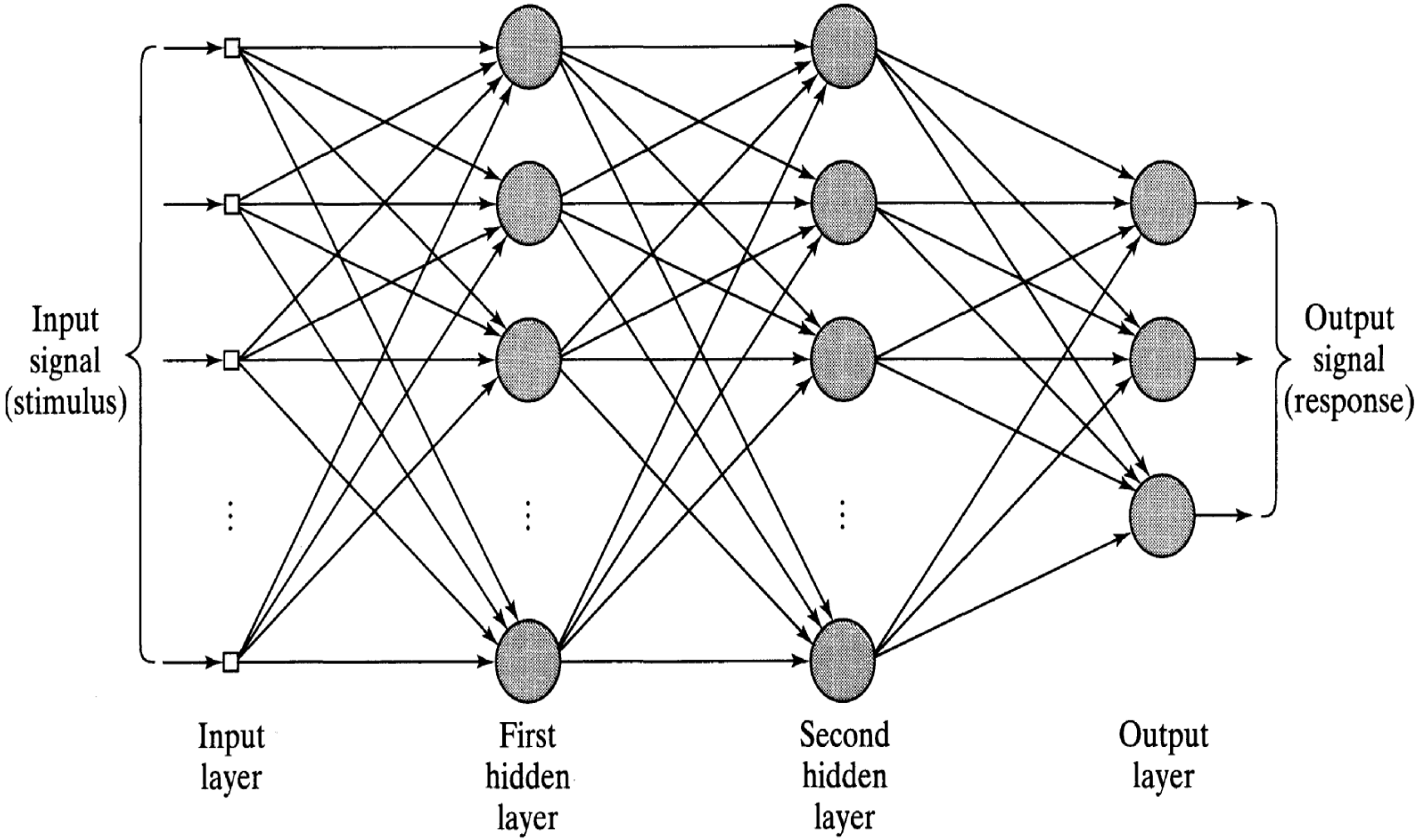
(2). The network contains one or more layers of hidden neurons that are not part of the input or output of the network. These hidden neurons enable the network to learn complex tasks.

(3) The network exhibits a high degree of connectivity, determined by the synapses of the network.

7.2 Some Preliminaries

The architecture of a MLP

The following diagram shows the architectural graph of a MLP neural network with two hidden layers and an output layer. Signal flowing through the network progresses in a forward direction, from left to the right, on a layer-by-layer basis.


Input signal (stimulus)
Output signal (response)
Input layer
First hidden layer
Second hidden layer
Output layer

1-D gradient descent method

Suppose we have a 1-D minimization problem:

$$J(x) = (a - bx)^2$$

a and *b* are parameters known. The goal of the optimization problem is to find a value of *x* so that *J*(x) is minimized.

To minimize the cost function *J*(x), we take the partial derivative:

$$\frac{\partial J(x)}{\partial x} = -2b(a - bx)$$

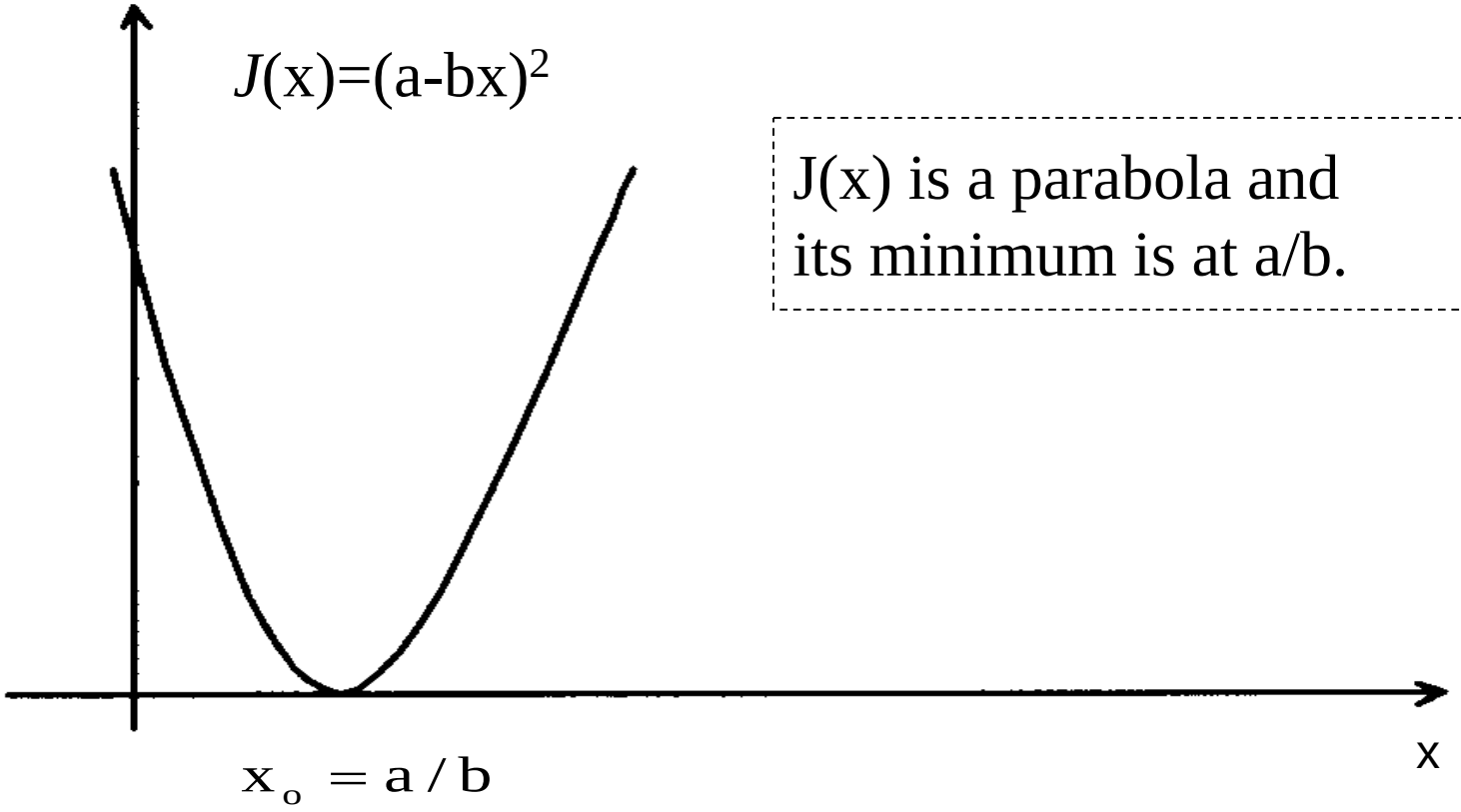
Solving the equation:

$$\frac{\partial J(x)}{\partial x} = 0$$

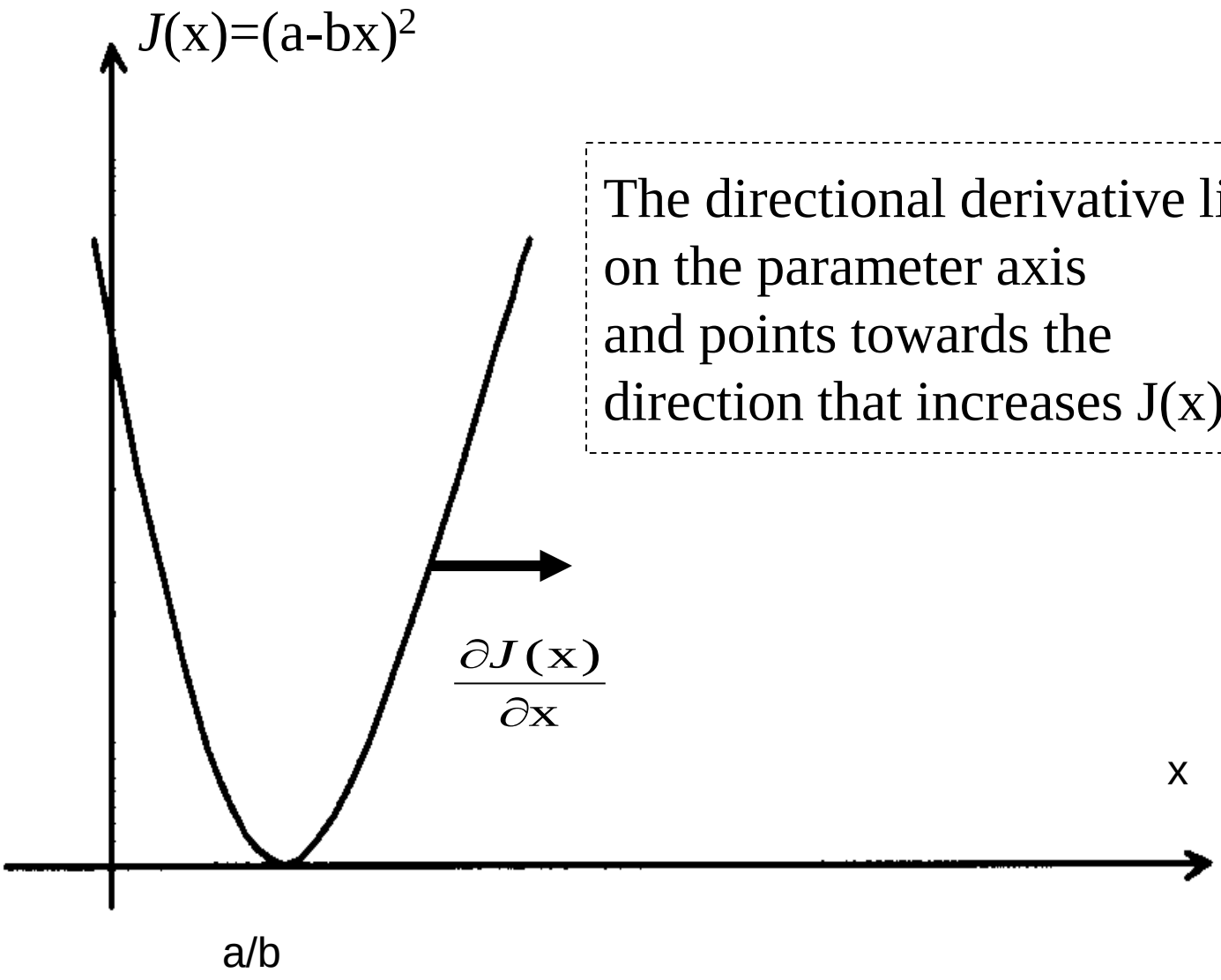
We obtain the optimal solution:

$$x_0 = a / b$$

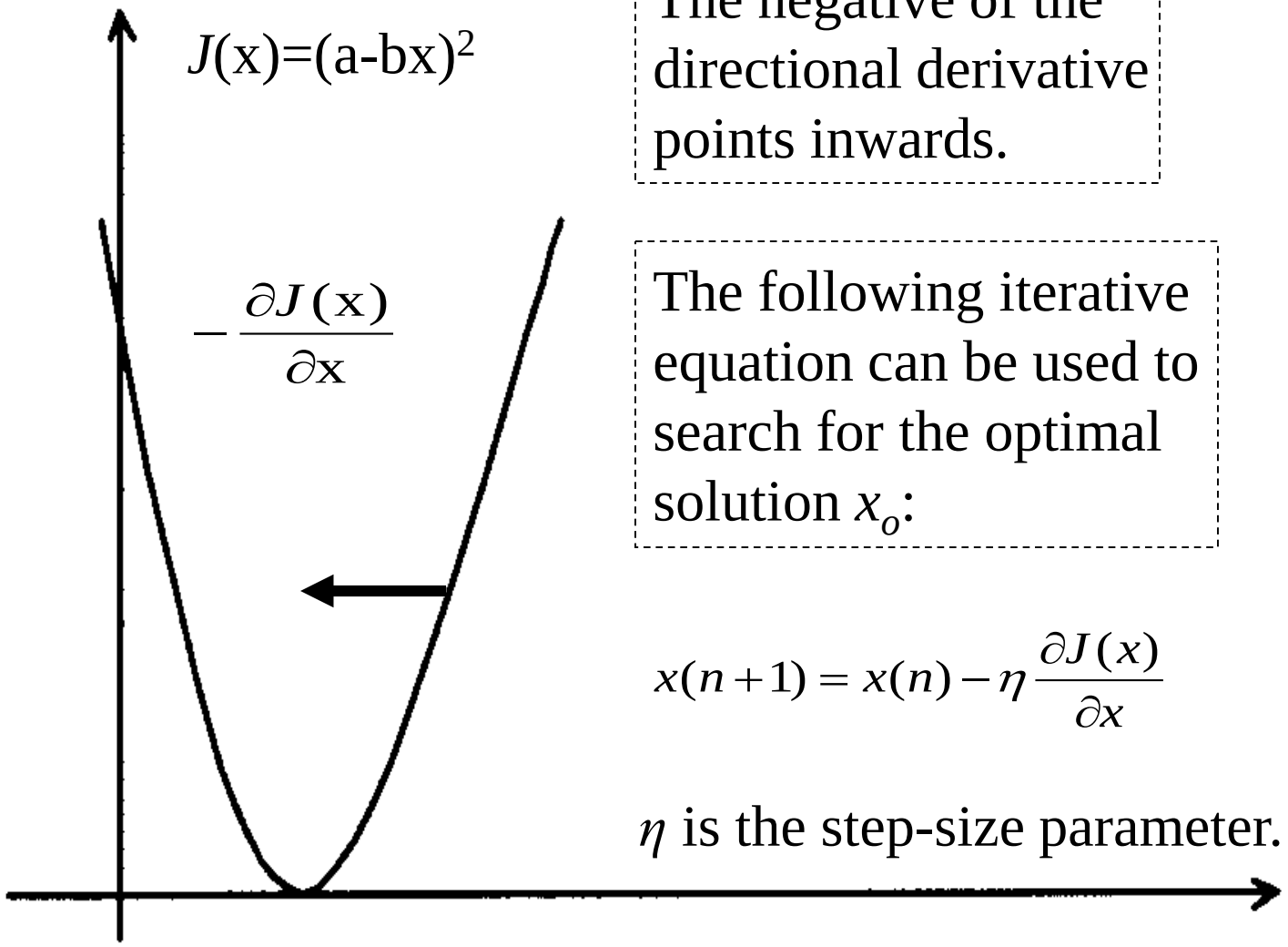
The following is the graphical illustration:



J(x) is a parabola and its minimum is at a/b.



The directional derivative lies on the parameter axis and points towards the direction that increases J(x)



The negative of the directional derivative points inwards.

The following iterative equation can be used to search for the optimal solution x_o:

$$x(n+1) = x(n) - \eta \frac{\partial J(x)}{\partial x}$$

η is the step-size parameter.

The object of MLP neural network learning

Given a set of training (learning) samples:

$$\{\mathbf{x}(1), \mathbf{d}(1)\}, \{\mathbf{x}(2), \mathbf{d}(2)\}, \cdots, \{\mathbf{x}(N), \mathbf{d}(N)\}$$

where

$$\mathbf{x}(i) = [x_1(i), x_2(i), \cdots, x_{n1}(i)]$$

$$\mathbf{d}(i) = [d_1(i), d_2(i), \cdots, d_{n2}(i)]$$

The objective of learning is to construct a MLP neural network so that the network response **f**[**x**(i)] approximates the desired response **d**(i) well:

$$\mathbf{f}[\mathbf{x}(i)] \rightarrow \mathbf{d}(i)$$

7.3 Back propagation algorithm for MLP training

Two types of signals used in the back propagation algorithm

(1) Function signals

A function signal is an input signal that comes in at the input layer of the network, propagates forward through the network and emerges at the output layer as output signal. We refer to such a signal as a function signal.

(2) Error signal

An error signal originates at the output neurons of the network and propagates backward, layer by layer, through the network. We refer to such a signal as an error signal because its computation by every neuron of the network involves an error-dependent function.

The error signal of the output neuron *j* at iteration *n* (i.e. when the *n*-th training sample is presented to the network) is defined by:

$$e_j(n) = d_j(n) - y_j(n) \tag{7.1}$$

Where $d_j(n)$ and $y_j(n)$ denote desired response and actual response of neuron *j* at *n*-th iteration.

Define the instantaneous value of the error energy for neuron *j* as:

$$\frac{1}{2} e_j^2(n) \tag{7.2}$$

Then the instantaneous value of the total error energy is obtained by summing the error energy of all neurons in the output layer:

$$E(n) = \frac{1}{2} \sum_{j \in C} e_j^2(n) \tag{7.3}$$

C is the set of output neurons.

Assume there are *N* training samples, the average squared error energy is obtained by summing *E(n)* over all *n* and then normalising with respect to the number of samples *N*:

$$E_{ave} = \frac{1}{N} \sum_{n=1}^{N} E(n) \qquad (7.4)$$

The instantaneous error energy *E(n)* and therefore the average error energy E_{ave} is a function of all free parameters, i.e. synaptic weights of the network. For a given training data set, E_{ave} represents the cost function as a measure of learning performance.

The objective of learning is to adjust the parameters (i.e. the weights) of the network to minimize E_{ave}. To perform this minimization, we consider a simple method of training, in which the weights are updated on a sample-by-sample basis until one complete presentation of the entire training samples have been used.

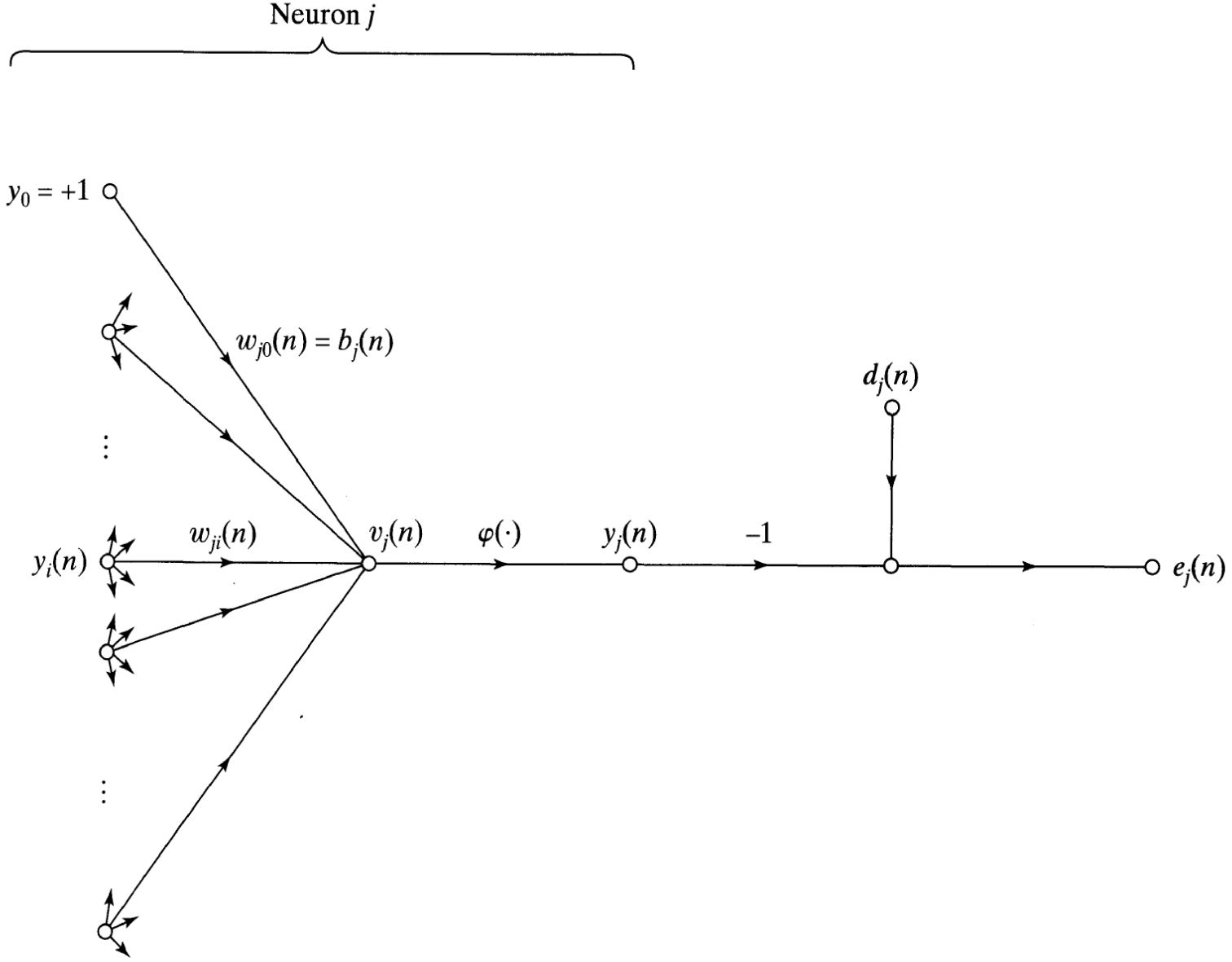
The average of these individual weight changes over the training set is therefore an estimate of the true change that would result from modifying the weights based on minimizing the cost function E_{ave} over the entire training samples.

Consider the following diagram, which depicts neuron *j* being fed by a set of function signals produced by a layer of neurons to its left. The activation of the neuron *j* is therefore:

$$v_j(n) = \sum_{i=0}^{m} w_{ji}(n) y_i(n) \qquad (7.5)$$

Where *m+1* is the total number of inputs applied to neuron *j*. Hence, the function signal $y_j(n)$, *i.e.* the output of neuron *j* at iteration *n* is:

$$y_j(n) = \varphi_j[v_j(n)] \qquad (7.6)$$


Neuron j
$y_0 = +1$
$w_{j0}(n) = b_j(n)$
$d_j(n)$
$w_{ji}(n)$
$y_i(n)$
$v_j(n)$
$\varphi(\cdot)$
$y_j(n)$
-1
$e_j(n)$

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 of $E(n)$ with respect to $w_{ji}(n)$.

According to the chain rule of calculus, we may express this derivative (gradient) as:

$$\frac{\partial E(n)}{\partial w_{ji}(n)} = \frac{\partial E(n)}{\partial e_j(n)} \frac{\partial e_j(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)} \frac{\partial v_j(n)}{\partial w_{ji}(n)} \qquad (7.7)$$

The partial derivative represents a sensitivity factor that determines the direction of search in weight space for the synaptic weight $w_{ji}(n)$.

Differentiating both sides of Eqn (7.3) with respect to $e_j(n)$, we get:

$$\frac{\partial E(n)}{\partial e_j(n)} = e_j(n) \qquad (7.8)$$

Differentiating both sides of Eqn (7.1) with respect to $y_j(n)$, we get:

$$\frac{\partial e_j(n)}{\partial y_j(n)} = -1 \tag{7.9}$$

Differentiating Eqn (7.6) with respect to $v_j(n)$, we obtain:

$$\frac{\partial y_j(n)}{\partial v_j(n)} = \varphi_j'[v_j(n)] \tag{7.10}$$

Differentiating Eqn (7.5) with respect to $w_{ji}(n)$, we obtain:

$$\frac{\partial v_j(n)}{\partial w_{ji}(n)} = y_i(n) \tag{7.11}$$

Substituting Eqns (7.8)-(7.11) into Eqn (7.7), yields:

$$\frac{\partial E(n)}{\partial w_{ji}(n)} = -e_j(n)\varphi_j'[v_j(n)]y_i(n) \tag{7.12}$$

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

$$\Delta w_{ji}(n) = -\eta \frac{\partial E(n)}{\partial w_{ji}(n)} = \eta \delta_j(n) y_i(n) \tag{7.13}$$

Where η is the learning-rate parameter of the back-propagation algorithm, and $\delta_j(n)$ is the local gradient defined by:

$$\begin{aligned} \delta_j(n) &= -\frac{\partial E(n)}{\partial v_j(n)} \\ &= -\frac{\partial E(n)}{\partial e_j(n)} \frac{\partial e_j(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)} \\ &= e_j(n)\varphi_j'[v_j(n)] \end{aligned} \tag{7.14}$$

The above equation shows that the local gradient points to the required changes in synaptic weights. The local gradient is equal to the product of the corresponding error signal for that neuron and the derivative of the associated activation function.

In the above, we note that a key factor involved in the calculation of the weight adjustment $\Delta w_{ji}(n)$ is the error signal $e_j(n)$ at the output 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 at the output layer

When neuron j is located in the output layer of the network, it is supplied with a desired response of its own. Thus, it is straightforward to compute the error signal $e_j(n)$.

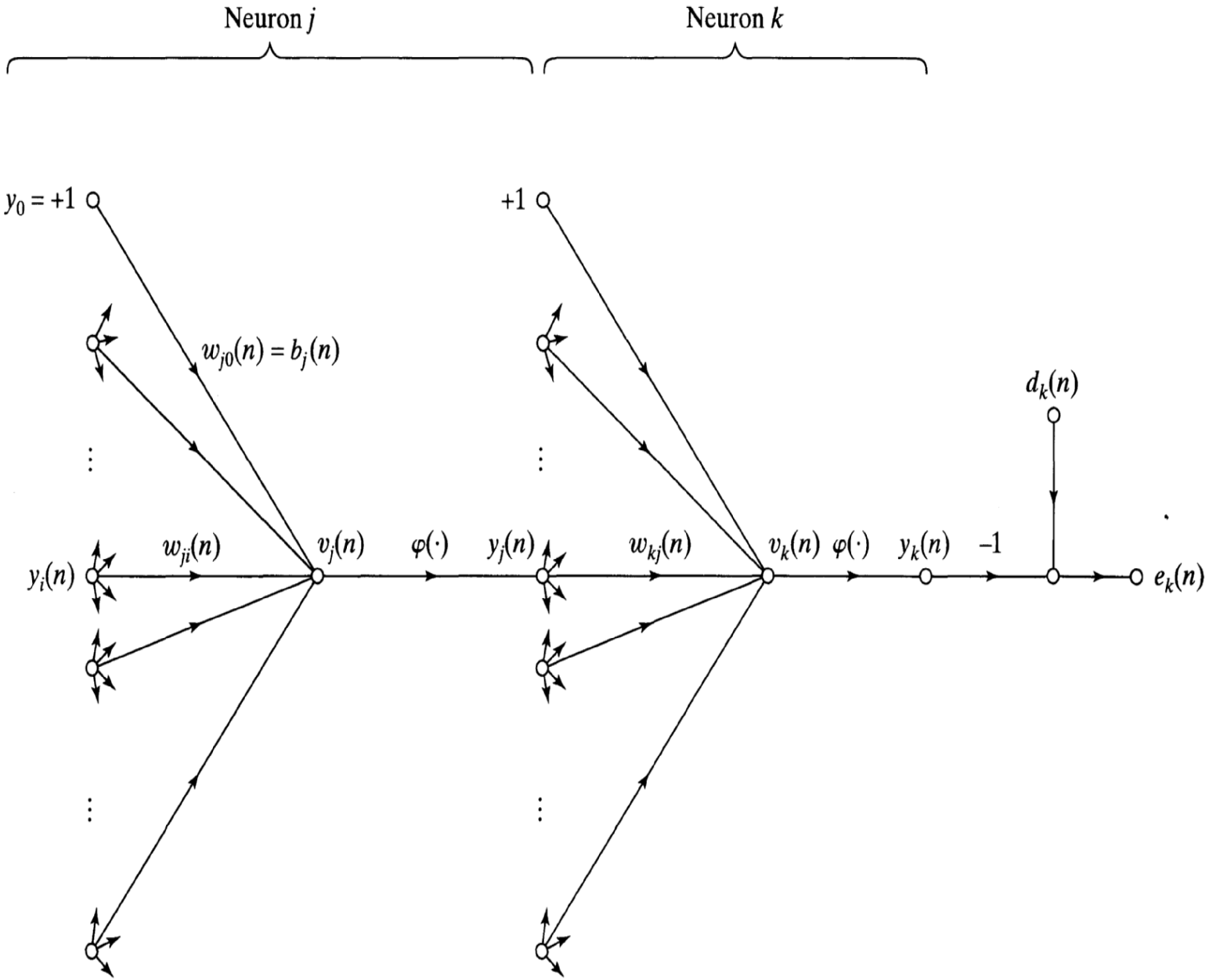
Case 2: neuron *j* is at the hidden layer

When neuron *j* is located in a hidden layer of the network, there is no supplied desired response for the neuron. Thus, the error signal for a hidden layer neuron has to be determined recursively in terms of the error signals of all neurons to which that hidden layer neuron is directly connected. This makes the back-propagation algorithm very complicated.

The local gradient for a hidden neuron *j* is defined as:

$$\delta_j(n) = -\frac{\partial E(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)}$$

$$= -\frac{\partial E(n)}{\partial y_j(n)} \varphi_j'[v_j(n)] \qquad (7.15)$$


Neuron j
Neuron k
$y_0 = +1$
$w_{j0}(n) = b_j(n)$
$y_i(n)$
$w_{ji}(n)$
$v_j(n)$
$\varphi(\cdot)$
$y_j(n)$
+1
$w_{kj}(n)$
$v_k(n)$
$\varphi(\cdot)$
$y_k(n)$
$d_k(n)$
−1
$e_k(n)$

To calculate the partial derivative $\partial E(n)/\partial y_j(n)$, we may proceed as follows:

$$E(n) = \frac{1}{2}\sum_{k \in C} e_k^2(n) \tag{7.16}$$

Where *C* is the set of output neurons.

Differentiating Eqn (7.16) with respect to $y_j(n)$, we get:

$$\frac{\partial E(n)}{\partial y_j(n)} = \sum_k e_k \frac{\partial e_k(n)}{\partial y_j(n)} \tag{7.17}$$

Next, we use the chain rule for the partial derivative $\partial E(n)/\partial y_j(n)$, and rewrite Eqn (7.17) as:

$$\frac{\partial E(n)}{\partial y_j(n)} = \sum_k e_k \frac{\partial e_k(n)}{\partial v_k(n)} \frac{\partial v_k(n)}{\partial y_j(n)} \tag{7.18}$$

We note that:

$$\begin{aligned} e_k(n) &= d_k(n) - y_k(n) \\ &= d_k(n) - \varphi_k[v_k(n)] \end{aligned} \qquad (7.19)$$

Hence, we have:

$$\frac{\partial e_k(n)}{\partial v_k(n)} = -\varphi_k'[v_k(n)] \qquad (7.20)$$

We also note from the above diagram that for neuron k, the activation is:

$$v_k(n) = \sum_{j=0}^{m} w_{kj}(n) y_j(n) \qquad (7.21)$$

Differentiating (7.21) with respect to $y_j(n)$, yields:

$$\frac{\partial v_k(n)}{\partial y_j(n)} = w_{kj}(n) \qquad (7.22)$$

Substituting Eqns (7.20) and (7.22) into Eqn (7.18), yields:

$$\frac{\partial E(n)}{\partial y_j(n)} = -\sum_k e_k(n)\varphi_k'(v_k(n))w_{kj}(n)$$
$$= -\sum_k \delta_k(n)w_{kj}(n) \qquad (7.23)$$

Substituting Eqn (7.23) into Eqn (7.15), we get the back-propagation formula for the local gradient of hidden neuron *j*:

$$\delta_j(n) = \varphi_j'(v_j(n))\sum_k \delta_k(n)w_{kj}(n) \qquad (7.24)$$

The factor $\varphi'_j[v_j(n)]$ involved in the computation of the local gradient in the above equation depends solely on the activation functions associated with hidden neuron *j*. The remaining factor $\delta_k(n)$ requires knowledge of the error signal $e_k(n)$, for all neurons that lie in the layer to the immediate right of hidden neuron *j*, and $w_{kj}(n)$ are synaptic weights associated.

Now, we summarize the relations that we have derived for the back-propagation algorithm.

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

$$\begin{pmatrix} weight \\ correction \\ \Delta w_{ji}(k) \end{pmatrix} = \begin{pmatrix} learning \\ rate \\ \eta \end{pmatrix} \begin{pmatrix} local \\ gradient \\ \delta_j(n) \end{pmatrix} \begin{pmatrix} input \\ signal \\ y_i(n) \end{pmatrix}$$

(2) Second, the local gradient depends on whether neuron *j* is an output or hidden neuron:

(i) If neuron *j* is hidden, the gradient is computed using Eqn (7.24)

(ii) If neuron *j* is an output neuron, the gradient is computed using Eqn (7.14)

Two passes of the computation

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

(1) Forward pass

In the forward pass, the synaptic weights remain unchanged throughout the network, and the function signals of the network are computed on a neuron-by-neuron, layer-by-layer basis.

(2) Backward pass

The backward pass starts from the output layer, passes error signals leftward through the network, layer-by-layer, and recursively computes the local gradient for each neuron. This recursive process permits the weights undergo changes, in accordance with the delta rule.

Activation function

The computation of the local gradient and hence weight correction of each neuron of the MLP neural network requires knowledge of the derivative of the activation function. For this derivative to exist, we require the activation function to be continuous and hence differentiable. An example of a continuously differentiable nonlinear activation function commonly used in MLP neural network is the sigmoidal nonlinear function, which has two typical forms:

(1) Logistic function.

This form of sigmoidal nonlinearity in its general form is defined by:

$$\varphi_j[v_j(n)] = \frac{1}{1+\exp[-av_j(n)]} \qquad (7.25)$$

Where $a>0$, and $v_j(n)$ is the activation signal of neuron j.

According to this nonlinearity, the amplitude of the output lies in the range of:

$$0 \le y_j(n) \le 1$$

Differentiating Eqn (7.25) with respect to $v_j(n)$, we get:

$$\varphi_j'[v_j(n)] = \frac{a \exp[-a v_j(n)]}{[1+\exp[-a v_j(n)]]^2} \qquad (7.26)$$

Considering $y_j(n)=\varphi_j[v_j(n)]$, we may express Eqn (7-26) into:

$$\varphi_j'[v_j(n)] = a y_j(n)[1-y_j(n)]$$

For a neuron *j* located in the output layer, $y_j(n)=o_j(n)$. Hence, the local gradient maybe expressed as:

$$\begin{aligned}\delta_j(n) &= e_j(n)\varphi_j'[v_j(n)] \\ &= a[d_j(n)-o_j(n)]o_j(n)[1-o_j(n)]\end{aligned} \qquad (7.27)$$

One the other hand, for an arbitrary hidden neuron *j*, we may express the local gradient as:

$$\begin{aligned}\delta_j(n) &= \varphi_j'[v_j(n)]\sum_k \delta_k(n) w_{kj}(n) \\ &= a y_j(n)[1 - y_j(n)]\sum_k \delta_k(n) w_{kj}(n)\end{aligned} \qquad (7.28)$$

(2) Hyperbolic tangent function

Hyperbolic tangent function is another commonly used form of sigmoidal nonlinearity, and its general form is defined by:

$$\varphi_j[v_j(n)] = a\tanh[bv_j(n)] = a\frac{\exp[bv_j(n)] - \exp[-bv_j(n)]}{\exp[bv_j(n)] + \exp[-bv_j(n)]} \qquad (7.29)$$

Suitable values for *a* and *b* are:

$$a = 1.7159$$
$$b = 2/3$$

The derivative of the hyperbolic tangent function with respect to $v_j(n)$ is given by:

$$\begin{aligned}\varphi_j'[v_j(n)] &= ab\,\mathrm{sech}^2[bv_j(n)] \\ &= ab\left(1-\tanh^2[bv_j(n)]\right) \\ &= \frac{b}{a}[a-y_j(n)][a+y_j(n)]\end{aligned} \qquad (7.30)$$

Where *sech* denotes hyperbolic secant:

$$\mathrm{sec}\,h(z) = \frac{2}{\exp(z)+\exp(-z)}$$

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

$$\begin{aligned}\delta_j(n) &= e_j(n)\varphi_j'[v_j(n)] \\ &= \frac{b}{a}[d_j(n)-o_j(n)][a-o_j(n)][a+o_j(n)]\end{aligned} \qquad (7.31)$$

For a neuron *j* located in a hidden layer, we have

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

$$= \frac{b}{a}[a - y_j(n)][a + y_j(n)] \sum_k \delta_k(n) w_{kj}(n) \qquad (7.32)$$

Learn-rate parameter *η*

The smaller we make the learning rate parameter η, the smaller the changes to synaptic weights in the network will be from one iteration to the next. This improvement, however, is attained at the cost of a slower rate of learning. If on the other hand, we make the learning-rate parameter too large, in order to speed up the rate of learning, the resulting large changes in the synaptic weights might make the network become unstable.

A simple method of increasing the rate of learning, yet avoiding the danger of instability, is to modify the delta rule by including a momentum term:

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

Where *α* is usually a positive number called the momentum constant. It controls the feedback loop acting around $\Delta w_{ji}(n)$. For the convergence of the weight estimation, the momentum constant must be restricted to the range:

$$0 \leq \alpha < 1$$

Modes of training

In a practical application of the back-propagation algorithm, learning results from the many presentations of a prescribed set of training samples to the MLP. 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 of the network stabilize.

For a give set of training samples, back-propagation learning may proceed in one of the following two basic ways.

(i) Sequential mode

The sequential mode of the back-propagation learning is also referred to as on-line learning. In this mode of operation, weight updating is performed after the presentation of each training sample. To be specific, consider an epoch consisting of *N* training samples (pairs):

$$\{\mathbf{x}(1), \mathbf{d}(1)\}, \{\mathbf{x}(2), \mathbf{d}(2)\}, \cdots, \{\mathbf{x}(N), \mathbf{d}(N)\}$$

The first training sample pair ***{x(1),d(1)}*** 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 of the network. Then the second training sample pair ***{x(2),d(2)}*** in the epoch is presented, and the forward and backward computations are repeated, resulting further adjustments to the weights. This process is continued until the last training sample pair ***{x(N),d(N)}*** in the epoch is presented.

(2) Batch mode

In the batch model, the weight updating is performed when all the samples in the epoch are presented to the network. Details are not described here, since the sequential mode is the commonly used method.

Summary of the back-propagation algorithm

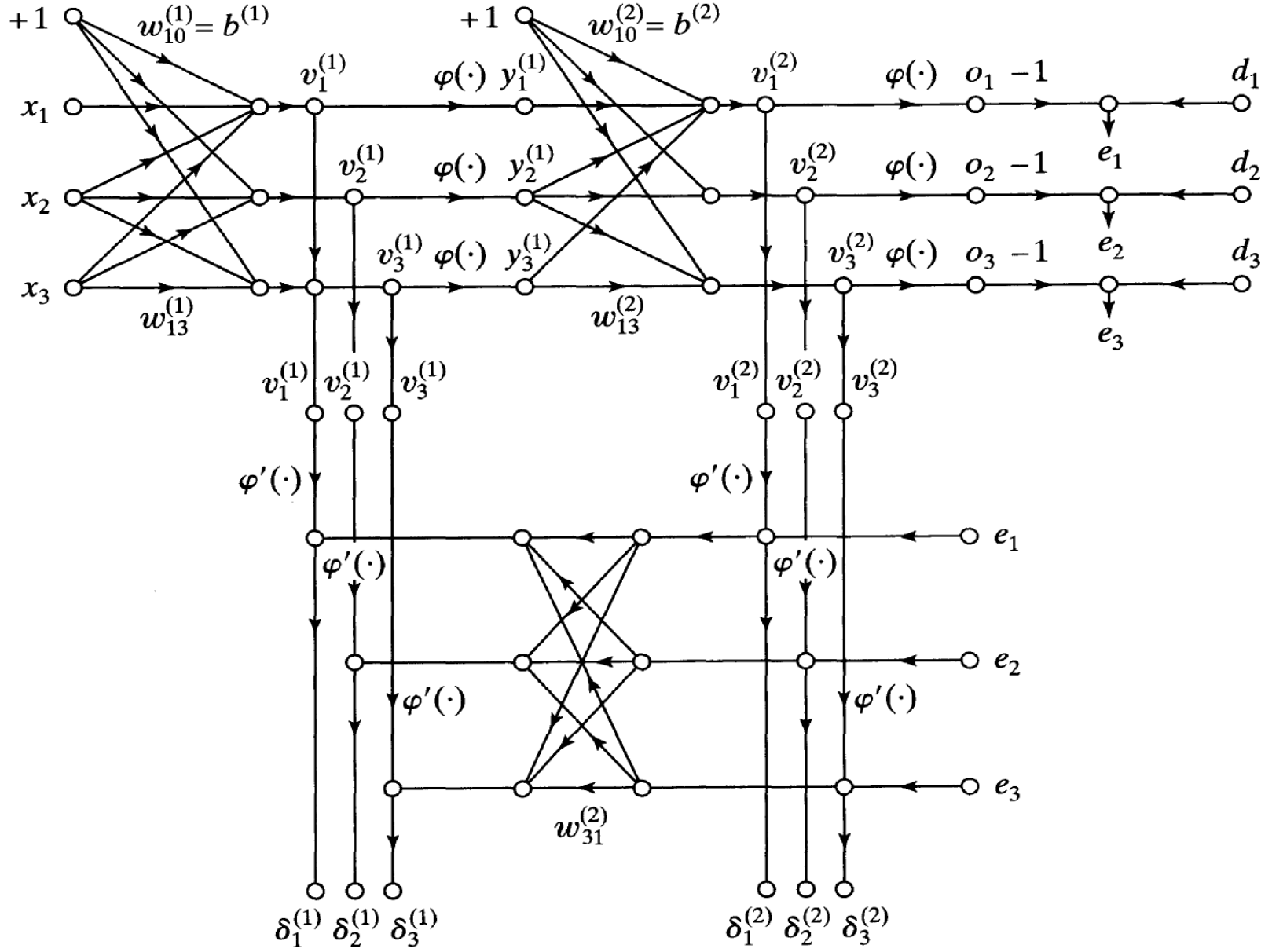
The signal flow graph for back-propagation learning, incorporating both the forward and backward phases of the computations involved in the learning process, is shown in the following diagram:

This graph is for the case of 1 hidden layer. The top part of the signal flow graph accounts for the forward pass, while the lower part of the graph accounts for the backward pass.

Often, the sequential learning of weights is the preferred method for on-line implementation of the back-propagation algorithm. For this mode of operation, the algorithm cycles as follows:

Assume there are *N* the training samples:

$$\{\mathbf{x}(1), \mathbf{d}(1)\}, \{\mathbf{x}(2), \mathbf{d}(2)\}, \cdots, \{\mathbf{x}(N), \mathbf{d}(N)\}$$


+1
$w_{10}^{(1)} = b^{(1)}$
x_1
x_2
x_3
$w_{13}^{(1)}$
$v_1^{(1)}$
$v_2^{(1)}$
$v_3^{(1)}$
$\varphi(\cdot)$
$y_1^{(1)}$
$y_2^{(1)}$
$y_3^{(1)}$
+1
$w_{10}^{(2)} = b^{(2)}$
$w_{13}^{(2)}$
$v_1^{(2)}$
$v_2^{(2)}$
$v_3^{(2)}$
o_1
o_2
o_3
-1
d_1
d_2
d_3
e_1
e_2
e_3
$\varphi'(\cdot)$
$w_{31}^{(2)}$
$\delta_1^{(1)}$ $\delta_2^{(1)}$ $\delta_3^{(1)}$
$\delta_1^{(2)}$ $\delta_2^{(2)}$ $\delta_3^{(2)}$

(1) Initialization.

Assuming that no prior information is available, pick the synaptic weights from a uniform distribution whose mean is zero and whose variance is chosen to make the standard deviation of the activation signals lie at the transition between linear and saturated parts of the sigmoidal activation function.

(2) Presentation of training samples

Present the network with an epoch of training samples. For each sample in the set, perform the sequence of forward and backward computations.

(3) Forward computation

Let a training sample in the epoch be denoted by $\{\mathbf{x}(n), \mathbf{d}(n)\}$, with the $\mathbf{x}(n)$ applied to the input layer and the desired response $\mathbf{d}(n)$ presented to the output layer. Compute the activation signals and function signals of the network by proceeding through the network, layer by layer.

(4) Backward computation

Compute the local gradient of the network, and adjust the synaptic weights of network.

(5) Iteration

Iterate the forward and backward computations in steps (3)-(4) by representing new epochs of training samples to the network until the stopping criterion is met.

Note, the order of presentation of training samples should be randomized from epoch to epoch.

The stopping criterion can be the number of iterations, or the rate of change of the average error small enough.