CHAPTER VI

Learning in Feedforward Neural Networks

The method of storing and recalling information in brain is not fully understood. However, experimental research has enabled some understanding of how neurons appear to gradually modify their characteristics because of exposure to particular stimuli. The most obvious changes have been observed to occur in the electrical and chemical properties of the synaptic junctions. For example the quantity of chemical transmitter released into the synaptic cleft is increased or reduced, or the response of the post-synaptic neuron to receive transmitter molecules is altered. The overall effect is to modify the significance of nerve impulses reaching that synaptic junction on determining whether the accumulated inputs to post-synaptic neuron will exceed the threshold value and cause it to fire. Thus learning appears to effectively modify the weighting that a particular input has with respect to other inputs to a neuron.

In this chapter, learning in feedforward networks will be considered. Research interest in multilayer feedforward networks dates back to the pioneering work of Rosenblatt (1962) on perceptrons and that of Widrow on Madalines [Widrow 62]. Madalines were constructed with many Adaline elements in the first layer, and with various logic devices such as AND, OR and majority vote-taker elements in the second layer. Madalines of the 1960`s had adaptive first layers and fixed threshold functions in the second (output) layers [Widrow and Lehr 90]. However the tool that was missing in those early days of multilayer feedforward networks was what we now call backpropagation learning.

Usage of the term ***backpropagation*** appears to have evolved in 1985. However, the basic idea of back-propagation was first described by Werbos in his Ph.D. Thesis [Werbos 74], in the context of a more general network. Subsequently, it was rediscovered by

Rumelhart, Hinton and Williams (1986b), and popularized through the publication of the seminal book entitled Parallel and Distributed Processing [Rumelhart and McClelland 86]. A similar generalization of the algorithm was derived independently by Parker, 1985, and interestingly enough, a roughly similar learning algorithm was also studied by LeCun (1985).

.

6.1. Perceptron Convergence Procedure

Perceptron was introduced by Frank Rosenblatt in the late 1950's (Rosenblatt, 1958) with a learning algorithm on it. Perceptron may have continuous valued inputs. It works in the same way as the formal artificial neuron defined previously. Its activation is determined by equation:

$$a=\mathbf{w}^{T}\mathbf{u} + \theta \quad (6.1.1)$$

Moreover, its output function is:

$$f(a) = \begin{cases} +1 & for\ 0 \le a \\ -1 & for\ a < 0 \end{cases} \quad (6.1.2)$$

having value either +1 or -1.

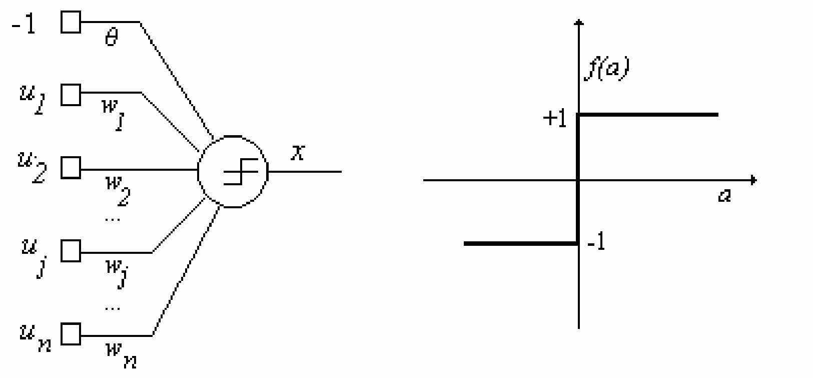

Figure 6.1. Perceptron

Now, consider such a perceptron in *N* dimensional space (Figure 6.1), the equation

$$\mathbf{w}^T\mathbf{u} + \theta = 0 \qquad (6.1.3)$$

that is

$$w_1u_1+w_2u_2+\ldots+w_N u_N + \theta = 0 \qquad (6.1.4)$$

defines a hyperplane. This hyperplane divides th e input space into two parts such that at one side, the perceptron has output value +1, and in the other side, it is -1.

A perceptron can be used to decide whether an input vector belongs to one of the two classes, say classes A and B. The decision ru le may be set as to respond as class A if the output is +1 and as class B if the output is -1. The perceptron forms two decision regions separated by the hyperplane. The equation of the boundary hyperplane depends on the connection weights and threshold.

Example 6.1: When the input space is two-dimensional then the equation

$$w_1u_1+w_2u_2 + \theta = 0 \qquad (6.1.5)$$

defines a line as shown in the Figure 6.2.

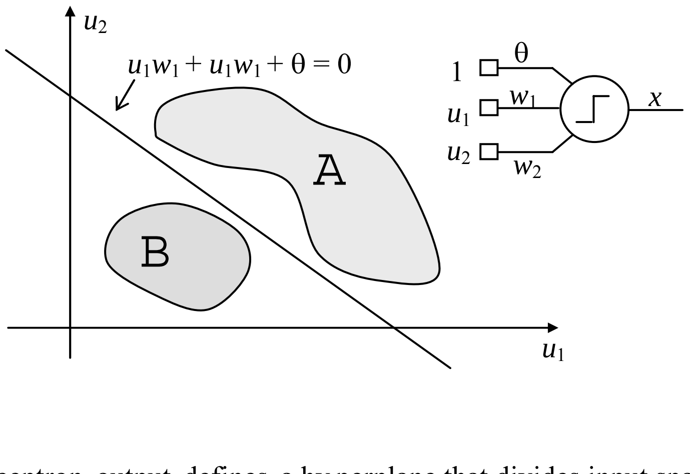

Figure 6.2. Perceptron output defines a hy perplane that divides input space into two separate subspaces

This line divides the space of input variables u_1 and u_2, which is a plane, into to two separate parts. In the given figure the elements of the classes A and B lies on the different sides of the line. ♦

Connection weights and the threshold in a pe rceptron can be fixed or adapted by using a number of different algorithm s. The original perceptron convergence procedure developed by [Rosenblatt, 1959] for adjusting weights is provided in the following:

THE PERCEPTRON CONVERGENCE PROCEDURE

Step 1: Initialize weights and threshold

Set each $w_j(0)$, for $j=0,1,2,..,N$, in $\mathbf{w}(0)$ to sm all random values. Here $\mathbf{w}=\mathbf{w}(t)$ is the weight vector at iteration tim e t and the com ponent $w0=\theta$ corresponds to the threshold.

Step 2. Present New Input and Desired output:

Present a new continuous valued input vector $\mathbf{u}^k$ to the along with the desired output y^k, such that:

$$y^k = \begin{cases} +1 & \mathit{for}\ \mathrm{u}^k \in A \\ -1 & \mathit{for}\ \mathrm{u}^k \in \mathrm{B} \end{cases}$$

Step 3. Calculate actual output

$$x^k=f(\mathbf{w}^\mathbf{T}\mathbf{u}^k)$$

Step 4. Adapt weights

$$\mathbf{w}(t+1)=\mathbf{w}(t)+\eta(y^k-x^k(t))\,\mathbf{u}k$$

where η is a positive constant less than 1.

Step 5. Repeat steps 2-4 until no error occurs

Initially connection weights and bias values are set to sm all random non-zero values. Then, a new input vector **u** with N continuous valued elements is applied to the input and the output value is calculated in Step 2 by usi ng the Eqs. (6.1.1) and (6.1.2). Notice that the connection weights are adapted only when an error occurs in step 4 that is when the calculated and the desired values are differe nt. W eights rem ain unchanged if a correct decision is m ade by the perceptron. The we ight update equation given in this step includes a gain term η that ranges from 0.0 to 1.0 and controls the learning rate. If η is not sm all enough, then oscillation m ay occur during weight adaptation. On the other hand, if η is too small then adaptation rate is very slow.

Example 6.2: Figure 6.3 dem onstrates how the line defined by the perceptrons parameters is shifted in tim e as the wei ghts are updated. Although it is not able to separate the classes A and B with the initial weights assigned at tim e *t*=0, it m anages to separate them at the end.

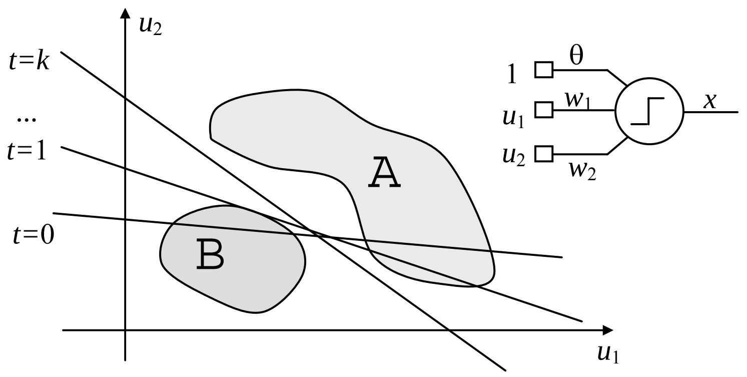

Figure 6.3: Perceptron convergence

In [Rosenblatt, 1959] it is proved that if th e inputs presented from the two classes are separable, that is if they fall on the opposite sides of som e hyperplane, then the perceptron convergence procedure always converge in tim e. Furthermore, it positions the final decision hyperplane such that it separates the samples of class A from those of class B.

One problem with the perceptron convergence procedure is that the decision boundary may oscillate continuously when the distributi ons overlap or the classes are not linearly separable (Figure 6.4).

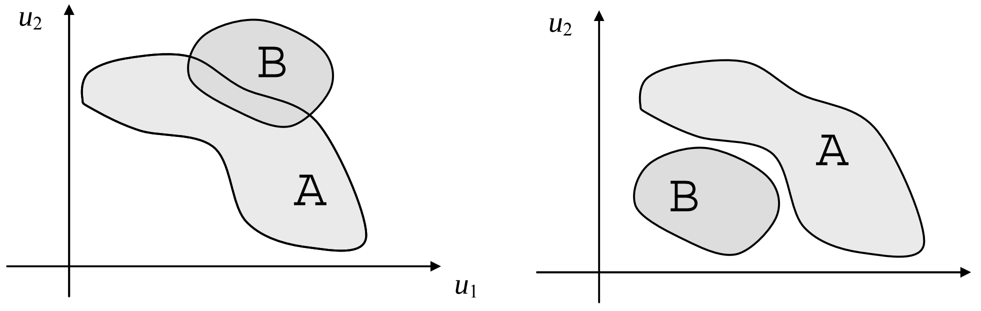

Figure 6.4. (a) Overlapping distributions (b) non linearly separable distribution

The types of decision regions that can be formed by single and m ultilayer perceptrons with one and two layers of hidden layers are given in the Figure 6.5 [Lipmann 87].

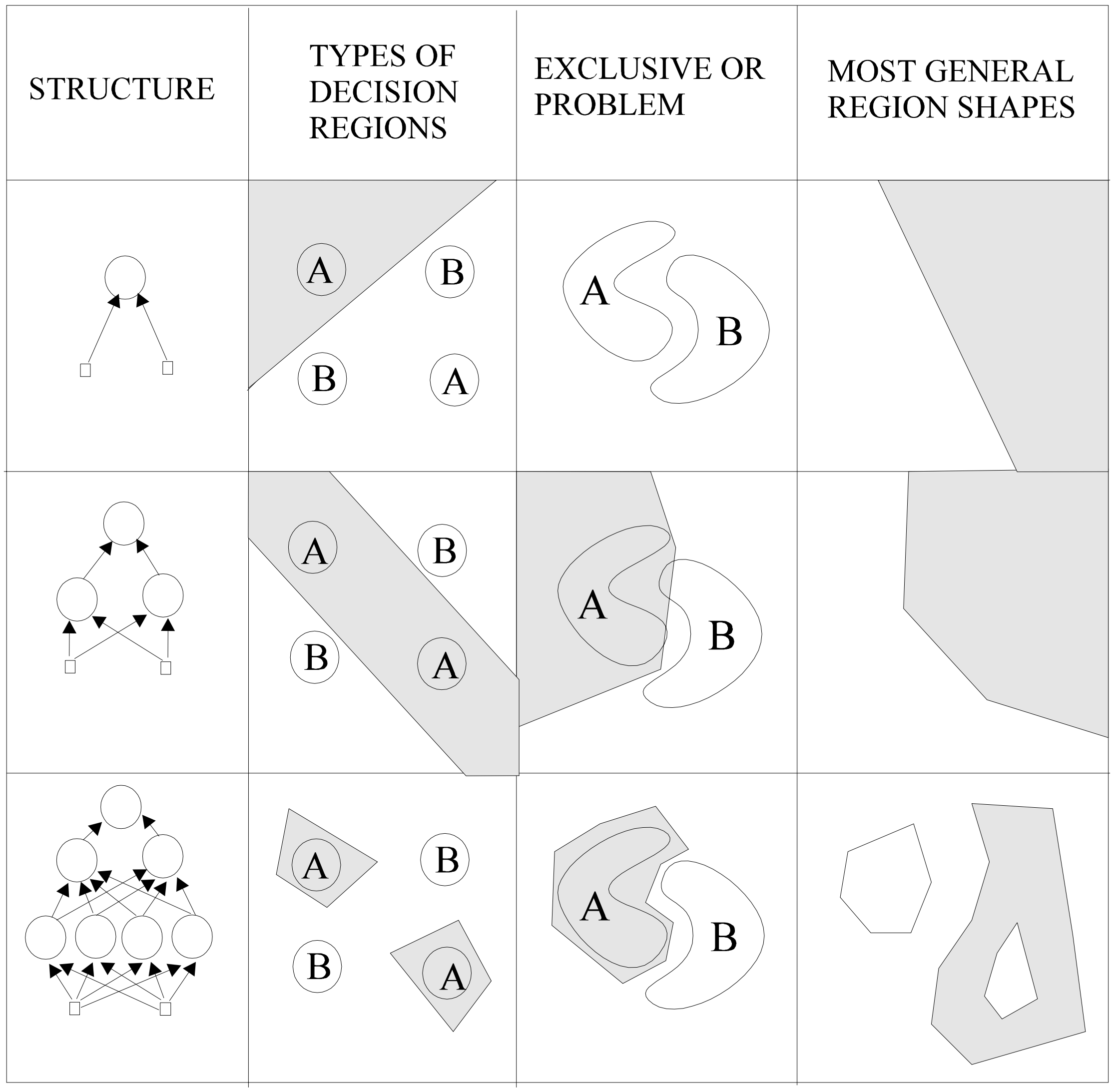

Figure 6.5. Types of regions that can be formed by single and multi-layer perceptrons (Adapted from Lippmann 87)

6.2 LMS Learning Rule

A modification to the perceptron convergence procedure forms the Least Mean Square (LMS) solution for the case that the classes are not separable. This solution minimizes the mean square error between the desired output and the actual output of the processing element. The LMS algorithm was first proposed for Adaline (Adaptive Linear Element) in [Widrow and Hoff 60]. The structure of Adaline is shown in the Figure 6.6. The part of the Adaline that executes the summation is called Adaptive Linear Combiner.

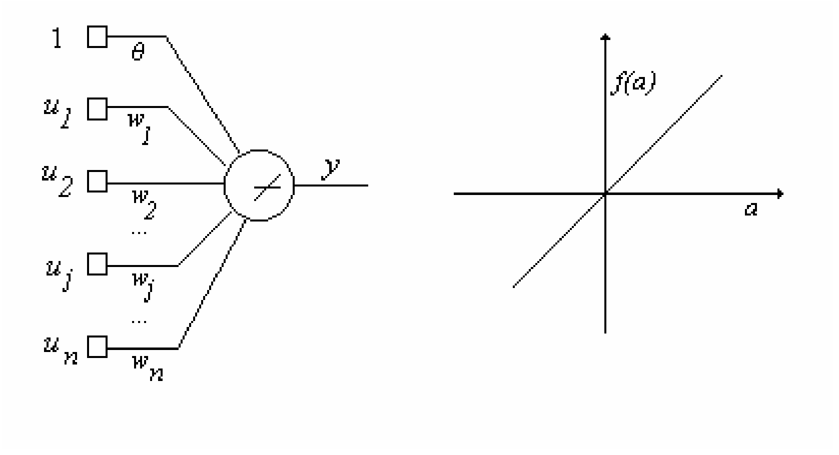

Figure 6.6 Adaline

The output function of the Adaline can be represented by the identity function as:

$$f(a)=a, \tag{6.2.1}$$

So the output can be written in terms of input and weights as:

$$x = f(a) = \sum_{j=0}^{N} w_j u_j \tag{6.2.2}$$

where the bias is implemented via a connection to a constant input u_0, which means that the input vector and the weight vector are of space $R^{(N+1)}$ instead of R^N.

The output equation of Adaline can be written as:

$$x=\mathbf{w}^{\mathsf{T}}\mathbf{u} \qquad (6.2.3)$$

where **w** and **u** are weight and input vectors respectively having dimension $N+1$.

Suppose that we have a set of input vectors $\mathbf{u}^k$, $k=1..K$, each having its own desired output value y^k.

The performance of the Adaline for a given input value $\mathbf{u}^k$ can be defined by considering the difference between the desired output y^k and the actual output x^k, which is called error and denoted as ε. Therefore, the error for the input $\mathbf{u}^k$ is as follows:

$$\varepsilon^k=y^k-x^k=y^k-\mathbf{w}^{\mathsf{T}}\mathbf{u}^k \quad (6.2.4)$$

The aim of the LMS learning is to adju st the weights through a training set $\{(\mathbf{u}^k,y^k)\}$, k=1..K, such that the mean of the square of the errors is minimum. The mean square error is defined as:

$$<(\varepsilon^k)^2>=\lim_{k\to\infty} \tfrac{1}{K}\sum_{k=1}^{K}(\varepsilon^k)^2 \quad (6.2.5)$$

where the notation <.> denotes the mean value.

The mean square error can be rewritten as:

$$\begin{aligned}<(\varepsilon^k)^2>&=<(y^k-\mathbf{w}^{\mathsf{T}}\mathbf{u}^k)^2>\\ &=<(y^k)^2>+\mathbf{w}^{\mathsf{T}}<\mathbf{u}^k\times\mathbf{u}^k>\mathbf{w}-2<y^k\mathbf{u}^{k^{\mathsf{T}}}>\mathbf{w} \quad (6.2.6)\end{aligned}$$

where $^{\mathsf{T}}$ denotes transpose and x is the outer vector product.

Defining input correlation matrix **R** [Widrow 85, Freeman 91]

$$\mathbf{R}=<\mathbf{u}^k\times\mathbf{u}^k>=<\mathbf{u}^k\mathbf{u}^{k^{\mathsf{T}}}> \quad (6.2.7)$$

and a vector **P** as

$$\mathbf{P} = < y^k \mathbf{u}^k > \quad (6.2.8)$$

results in:

$$e(\mathbf{w}) = < (\varepsilon^k)^2 > = < (y^k)^2 > + \mathbf{w}^{\mathsf{T}} \mathbf{R}\, \mathbf{w} - 2\mathbf{P}^{\mathsf{T}} \mathbf{w} \quad (6.2.9)$$

The optimum value **w*** for the weight vector correspondi ng to the minimum of the mean squared error can be obtained by evaluating the gradient of $e(\mathbf{w})$. The point which makes the gradient zero gives us the value of **w***. That is:

$$\nabla e(\mathbf{w})\Big|_{\mathbf{w}=\mathbf{w}^*} = \frac{\partial e\ (\mathbf{w})}{\partial \mathbf{w}}\Bigg|_{\mathbf{w}=\mathbf{w}^*} = 2\mathbf{R}\mathbf{w}^* - 2\mathbf{P} = 0 \quad (6.2.10)$$

Here, the gradient is

$$\nabla e(\mathbf{w}) = \left[{}^{\partial e}\!/\!_{\partial w_1} \quad {}^{\partial e}\!/\!_{\partial w_2} \quad \cdots \quad {}^{\partial e}\!/\!_{\partial w_n} \right]^{\mathsf{T}} \quad (6.2.11)$$

and it is a vector extending in the direction of the greatest rate of change. The gradient of a function evaluated at som e point is zero if the function has a m aximum or minimum at that point. The error function is of the s econd degree so it is a paraboloid and it has a single minimum at point **w***.

When we set the gradient of the mean square error to zero, this implies that

$$\mathbf{Rw^{*}=P} \qquad (6.2.12)$$

and then

$$\mathbf{w^{*}=R^{-1}P} \qquad (6.2.13)$$

6.3 Steepest Descent Algorithm.

The analytical calculation of the optim um weight vector for a problem is rather difficult in general. Not only does the m atrix m anipulation get cum bersome for the large dimensions, but also each com ponent of **R** and **P** itself is an expectation value. Thus, explicit calculations of **R** and **P** require knowledge of the statistics of the input signal [Freeman 91]. A better approach would be to let the Adaline Linear Combiner to find the optimum weights by itself through a search ove r the error surface. Instead of having a purely random search, som e intelligence is adde d to the procedure such that the weight vector is changed by considering the gradient of e(**w**) iteratively [Widrow 60], according to formula known as *delta rule*:

$$\mathbf{w}(t+1)=\mathbf{w}(t)+\Delta\mathbf{w}(t) \quad (6.3.1)$$

where

$$\Delta\mathbf{w}(t)=-\eta\nabla e(\mathbf{w}(t)) \quad (6.3.2)$$

In the above formula η is a small positive constant, determining the learning rate.

For the real valued scalar function $e(\mathbf{w})$ on a vector space $\mathbf{w} \in R^N$, the gradient $\nabla e(\mathbf{w})$ gives the direction of the steepest upward slope , so the negative of the gradient is the direction of the steepest descent . This fact is demonstrated in Figure 6.7 for a parabolic error surface on two dimensions.

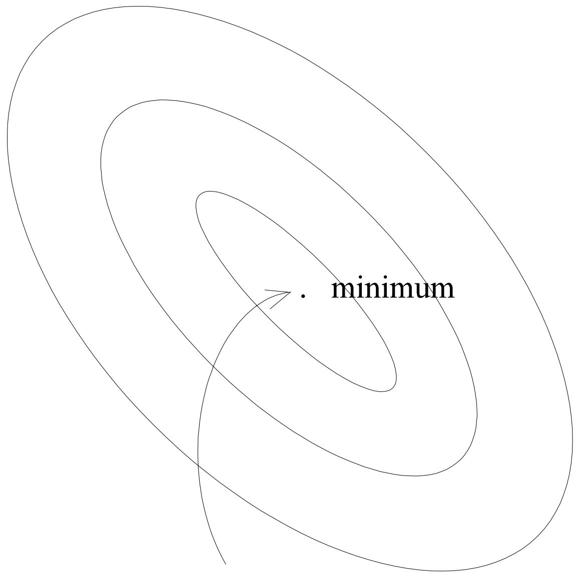

Figure 6.7 Direction of the steepest gradient descent on the paraboloid error surface on two-dimensional weight space. Only the equpotential curves of the error surface is shown instead of the 3D-error surface.

In Section 6.2 we have considered the lin ear output function in the derivation of the optimum weight **w*** for the m inimum error. However in the general case, we should consider any nonlinearity *f*(.) at the output of the neuron. It should be noted that in such a case the error surface is no more a paraboloid, so it may have several local minima.

For an input $\mathbf{u}^k$ applied at tim e *t*, $(\varepsilon^k(t))^2$ can be used as an approxim ation to $<(\varepsilon^k)^2>$, where

$$\varepsilon^k(t)=y^k-f(a^k)=y^k-f(\mathbf{w}(t)^\mathsf{T}\mathbf{u}^k) \quad (6.3.3)$$

Therefore, we obtain:

$$\nabla<(\varepsilon^k)^2> \cong \nabla(\varepsilon^k(t))^2=\nabla(y^k-f(a^k))^2 \quad (6.3.4)$$

With a differentiable function *f*(.) having derivative *f'*(.) it becomes

$$\nabla(y^k-f(a^k))^2=-2\varepsilon^k(t)f'(a^k)\nabla a^k \quad (6.3.5)$$

Since

$$\nabla a^k=\nabla\mathbf{w}(t)^\mathsf{T}\mathbf{u}^k=\mathbf{u}^k \qquad (6.3.6)$$

Then the weight update formula becomes:

$$\mathbf{w}(t+1)=\mathbf{w}(t)+2\eta\varepsilon^k(t)f'(a^k)\mathbf{u}^k. \qquad (6.3.7)$$

Notice that for Adaline's linear output function:

$$f'(a)=1 \quad (6.3.8)$$

For sigmoid function it is:

$$f'(a)=\frac{\partial}{\partial a}\left(\frac{1}{1+e^{-a/T}}\right)=\frac{1}{T}f(a)(1-f(a)) \quad (6.3.9)$$

The steepest descent algorithm based on least mean square error is summarized in the following:

STEEPEST DESCENT ALGORITHM

Step 1: Apply an input vector $\mathbf{u}^k$ with an desired output value y^k to the neuron's inputs
Step 2: By considering $\mathbf{u}^k$ and using the current value of the weight vector determine the value of the activation a^k:

$$a^k=\mathbf{w}(t)^\mathsf{T}\mathbf{u}^k$$

Step 3: Determine the value of the derivative of the output function using the current value of activation a^k, that is:

$$f'(a^k)=\left.\frac{\partial f(a)}{\partial a}\right|_{a=a^k}$$

Step 4: Determine the value of error $\varepsilon k(t)$ as: $\varepsilon^k(t)= y^k-f(a^k)$

Step 5: Update the weight vector using the following update formula

$$\mathbf{w}(t+1)=\mathbf{w}(t)+2\eta f'(a^k)\varepsilon^k(t)\mathbf{u}^k$$

Step 6: Repeat steps 1-5 until $<\varepsilon^k(t)^2>$ reduces to an acceptable level.

The parameter η in the algorithm determines the stability and the speed of convergence of the weight vector towards the minimum error value. The value of η should be tuned well. If it is chosen too small this effects considerably the convergence time. On the other hand, if changes are too large, the weight vector may wander around the minimum as shown in the Figure 6.8, without being able to reach it.

Notice that, the iterative weight update by the delta rule is derived by assuming constant $\mathbf{u}^k$. Therefore, it tends to minimize the error with respect to applied $\mathbf{u}^k$. In fact, we require the average error, that is:

$$e=<(\varepsilon^k)^2>=\tfrac{1}{K}\sum_{k=1}^{K}(\varepsilon^k)^2 \quad (6.3.10)$$

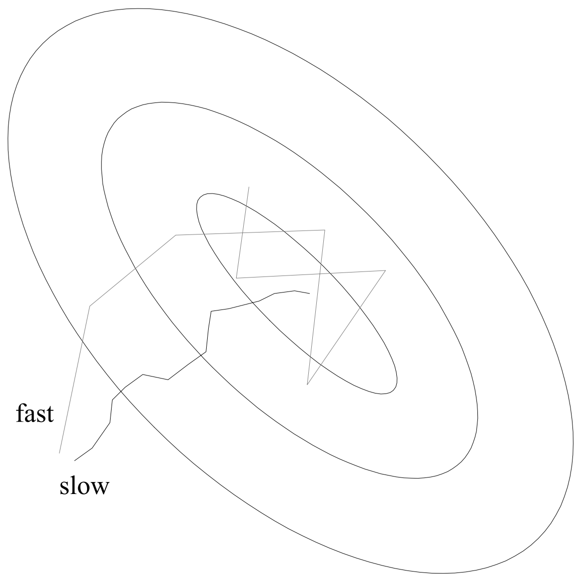

Figure 6.8. Inappropriate value of learning rate η may cause oscillations in the weight values without convergence

to be minimum, and this implies that

$$\frac{\partial e}{\partial w_j} = \tfrac{1}{K}\sum_{k=1}^{K}\frac{\partial(\varepsilon^k)^2}{w_j} = \tfrac{1}{K}\sum_{k=1}^{K}\frac{2\varepsilon^k\partial\varepsilon^k}{w_j} \tag{6.3.11}$$

Therefore, the net change in w_j af ter one com plete cycle of pattern presentation is expected to be:

$$w_j(t+K) = w_j(t) - \eta\frac{1}{K}\sum_{k=1}^{K}\frac{2\varepsilon^k\partial\varepsilon^k}{\partial wj} \tag{6.3.12}$$

However, this would be true that if the we ights are not updated along a cycle, but only at the end. By changing the weights as each pattern is presented, we depart to som e extend from the gradient descent in e. Nevertheless, provided the learning rate is suf ficiently small, this departure w ill be negligible a nd the delta rule w ill im plement a very close approximation to gradient descent in mean squared error [Freeman 91].

6.4. The Backpropagation Algorithm

6.4.1. Learning Single Layer Network

Consider a single layer multiple output network as shown in the Figure 6.9. Here, we still have N inputs denoted u_j, $j=1..N$, but M processing elem ents whose activations and outputs are denoted as a_i and x_i , $i=1..M$ respectively. Here w_{ji} is used to denote the strength of the connection from the jth input to the ith processing elem ent. In vector notation w_{ji} is the jth component of weight vector $\mathbf{w}_i$, while u_j is the jth component of the input vector **u.** Let $\mathbf{u}^k$ and $\mathbf{y}^k$ to represent the kth input sam ple and the corresponding desired output vector respectively.

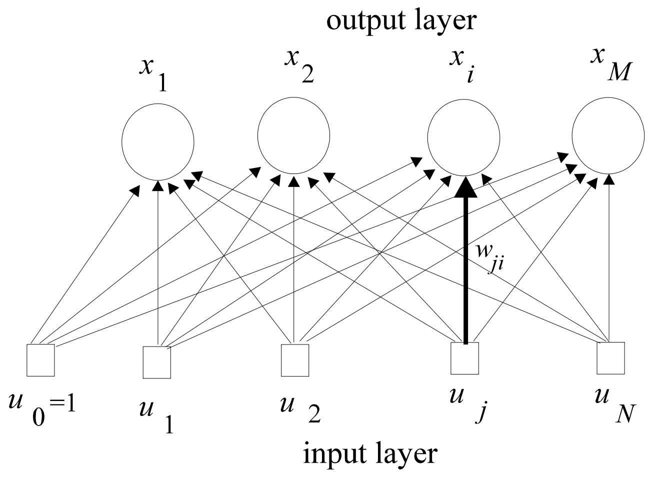

Figure 6.9. Multiple output network

Let the error observed at the output i be

$$\varepsilon_i^k = y_i^k - x_i^k \quad (6.4.1)$$

when $\mathbf{u}^k$ is applied at the input. If the error is to be written in terms of the input vector $\mathbf{u}^k$ and the weights $\mathbf{w}_i$, we obtain

$$\varepsilon_i^k = y_i^k - f(\mathbf{w}_i^{\mathsf{T}}\mathbf{u}^k) \quad (6.4.2)$$

If we take partial derivative with respect to w_{ji} by applying the chain rule

$$\frac{\partial \varepsilon_i^k}{\partial w_{ji}} = \frac{\partial \varepsilon_i^k}{\partial x_i^k}\frac{\partial x_i^k}{\partial w_{ji}} \quad (6.4.3)$$

where

$$\frac{\partial \varepsilon_i^k}{\partial x_i^k} = -1 \quad (6.4.4)$$

and

$$\frac{\partial x_i^k}{\partial w_{ji}} = f'(a_i^k)u_j^k \qquad (6.4.5)$$

we obtain

$$\frac{\partial \varepsilon^k}{\partial w_{ij}} = -f'(a^k)u_j^k \quad (6.4.6)$$

If we def ine the total output error f or input $\mathbf{u}^k$ as the sum of the square of the errors at each neuron output, that is:

$$e^k = \frac{1}{2}\sum_{i=1}^{m}(\varepsilon_i^k)^2 \quad (6.4.7)$$

then partial derivative of the total error with respect to w_{ji} when $\mathbf{u}^k$ is applied at the input can be written as:

$$\frac{\partial e^k}{\partial w_{ji}} = \frac{\partial e^k}{\partial \varepsilon_i^k}\frac{\partial \varepsilon_i^k}{\partial w_{ji}} \quad (6.4.8)$$

which is

$$\frac{\partial e^k}{\partial w_{ji}} = -\varepsilon_i^k f'(a^k)u_j \quad (6.4.9)$$

By defining

$$\delta_i^k = \varepsilon_i^k f'(a^k) \quad (6.4.10)$$

it can be reformulated as

$$\frac{\partial e^k}{\partial w_{ji}} = -\delta_i^k u_j^k \quad (6.4.11)$$

For the error to be m inimum, the gradient of the total error with respect to the weights should be

$$\nabla e^k = \mathbf{0} \quad (6.4.12)$$

where $\mathbf{0}$ is the vector having $N.M$ entries each having value zero. In other words, it should be satisfied:

$$\frac{\partial e^k}{\partial w_{ji}} = 0 \quad for\ j = 1..N,\ i = 1..M \quad (6.4.13)$$

In order to reach the m inimum of the total e rror, without solving the above equation, we apply the delta rule in the same way explained for the steepest descent algorithm:

$$\mathbf{w}(t+1)=\mathbf{w}(t)-\eta\tilde{}\ e^k \quad (6.4.14)$$

in which

$$w_{ji}(t+1) = w_{ji}(t) - \eta \frac{\partial e^k}{\partial w_{ji}} \quad for\ j = 1..N,\ i = 1..M \quad (6.4.15)$$

that is

$$w_{ji}(t+1) = w_{ji}(t) + \eta \delta_i^k u_j^k \quad for\ j = 1..N,\ i = 1..M \quad (6.4.16)$$

6.4.2. Multilayer Network

Now assume that another layer of neurons is connected to the input side of the output layer. Therefore we have the input, hidden and the output layers as shown in Figure 6.10. In order to discriminate between the elements of the hidden and output layers we will use the subscripts *L* and *o* respectively. Furthermore, we will use *h* as the index on the hidden layer elements, while still using index *j* and *i* for the input and output layers.

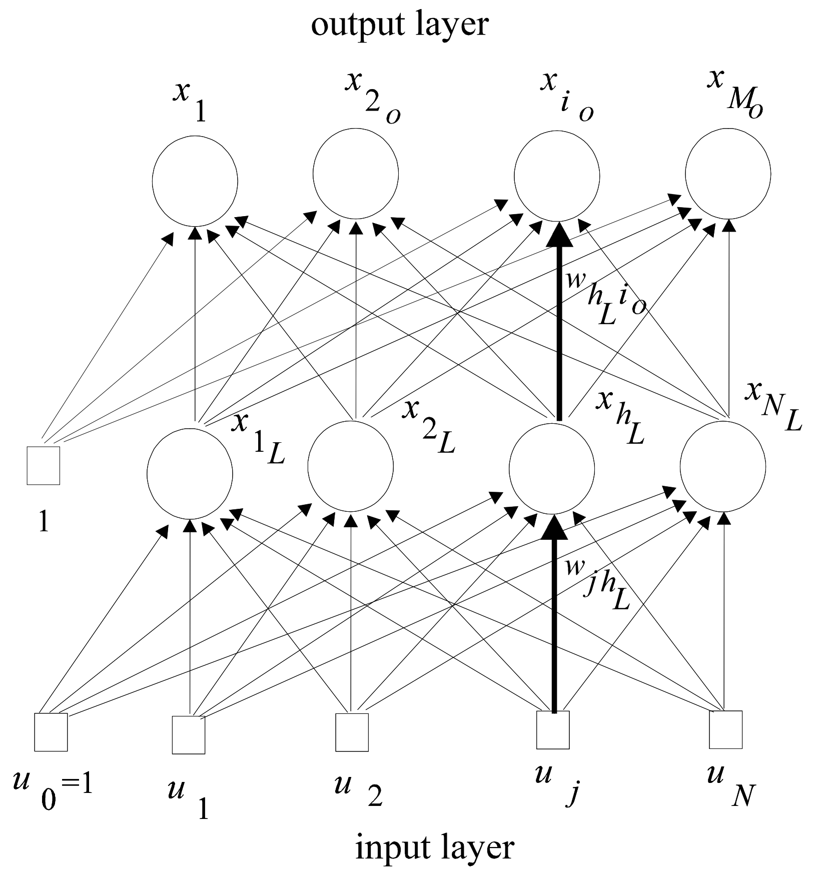

Figure 6.10 Multilayer network

In such a network, the output value of *i*th neuron of output layer can be written as:

$$x_{i_o}^k = f_o(\mathbf{w}_{i_o}{}^{\mathsf{T}}\mathbf{x}_L^k) \quad (6.4.17)$$

where $\mathbf{x}_L^k$ being the vector of output values at hidden layer that is connected as input to the output layer. The value of the *h*th element in $\mathbf{x}_L{}^k$ is determined by the equation:

$$x_{h_L}^k = f_L(\mathbf{w}_{h_L}{}^{\mathrm{T}}\mathbf{u}^k) \quad (6.4.18)$$

Since

$$\mathbf{w}_{h_L}{}^{\mathrm{T}}\mathbf{u}^k = \sum_{j=1}^{N} w_{jh_L} u_j^k \quad (6.4.19)$$

the partial derivative of the output of a neuron i_o of output layer with respect to hidden layer weight w_{jh_L} can be determined by applying the chain rule

$$\frac{\partial x_{i_o}^k}{\partial w_{jh_L}} = \frac{\partial x_{i_o}^k}{\partial x_{h_L}^k} \frac{\partial x_{h_L}^k}{\partial w_{ij_L}} \quad (6.4.20)$$

By using Eq. (6.4.17) and (6.4.19) this can be written as

$$\frac{\partial x_{i_o}^k}{\partial w_{jh_L}} = (f_o'(a_{i_o}^k) w_{h_L i_o})(f_L'(a_{h_L}^k) u_j^k) \quad (6.4.21)$$

Then the partial derivative of the total error

$$e^k = \tfrac{1}{2}\sum_{i_o=1}^{M} (\varepsilon_{i_o}^k)^2 = \tfrac{1}{2}\sum_{i_o=1}^{M} (y_{i_o}^k - x_{i_o}^k)^2 \quad (6.4.22)$$

with respect to the hidden layer weight w_{jh_L} can be written as

$$\frac{\partial e^k}{\partial w_{jh_L}} = -\sum_{i_o=1}^{M} \varepsilon_{i_o}^k f_o'(a_{i_o}^k) w_{h_L i_o} f_L'(a_{h_L}^k) u_j^k \quad (6.4.23)$$

It can be reformulated as

$$\frac{\partial e^k}{\partial w_{jh_L}} = -\sum_{i_o=1}^{M} \delta_{i_o}^k w_{h_L i_o} f_L'(a_{h_L}^k) u_j^k . \quad (6.4.24)$$

When defined

$$\delta_{h_L}^{k} = f_L'(a_{h_L}^{k}) \sum_{i_o=1}^{M} \delta_{i_o}^{k} w_{h_L i_o} \quad (6.4.25)$$

it becomes

$$\frac{\partial e^k}{\partial w_{j h_L}} = -\delta_{h_L}^{k} u_j \quad (6.4.26).$$

Therefore, the weight update rule for the hidden layer

$$w_{j h_L}(t+1) = w_{j h_L}(t) - \eta \frac{\partial e^k}{\partial w_{j h_L}} \quad (6.4.27)$$

can be reformulated in analogy with the weight update rule of the output layer, as

$$w_{j h_L}(t+1) = w_{j h_L}(t) + \eta \delta_{h_L}^{k} u_j \quad (6.4.28)$$

This weight update rule m ay be generalized for the networks having several hidden layers as:

$$w_{j_{(L-1)} h_L}(t+1) = w_{j_{(L-1)} h_L}(t) + \eta \delta_{h_L}^{k} x_{j_{(L-1)}} . \quad (6.4.29)$$

where *L* and (*L*-1) are used to denote any hidden layer and its previous layer respectively.

Furthermore,

$$\delta_{j_{(L-1)}}^{k} = f_{L-1}'(a_{j_{(L-1)}}^{k}) \sum_{h_L=1}^{N_L} \delta_{h_L}^{k} w_{j_{(L-1)} h_L} \quad (6.4.30)$$

where N_L is the number of neurons at layer *L*.

The backpropagation algorithm f or m ulti-layered network is sum marized in the following.

BACKPROPAGATION ALGORITHM FOR MULTILAYERED FEEDFORWARD NEURAL NETWORK

Step 0. Initialize weights: to small random values;

Step 1. Apply a sample: apply to the input a sample vector $\mathbf{u}^k$ having desired output vector $\mathbf{y}^k$;

Step 2. Forward Phase:

Starting from the first hidden layer and propagating towards the output layer:

2.1. Calculate the activation values for the units at layer *L* as:

2.1.1. If *L*-1 is the input layer

$$a_{h_L}^k = \sum_{j=0}^{N} w_{jh_L} u_j^k$$

2.1.2. If *L*-1 is a hidden layer

$$a_{h_L}^k = \sum_{j_{L-1}=0}^{N_{L-1}} w_{j_{(L-1)}h_L} x_{j_{(L-1)}}^k$$

2.2. Calculate the output values for the units at layer *L* as:

$$x_{h_L}^k = f_L(a_{h_L}^k)$$

in which use i_o instead of h_L if it is an output layer

Step 4. Output errors: Calculate the error terms at the output layer as:

$$\delta_{i_o}^k = (y_{i_o}^k - x_{i_o}^k) f_o'(a_{i_o}^k)$$

Step 5. Backward Phase Propagate error backward to the input layer through each layer *L* using the error term

$$\delta_{h_L}^k = f_L'(a_{h_L}^k) \sum_{i_{L+1}=1}^{N_{L+1}} \delta_{i_{(L+1)}}^k w_{h_L i_{(L+1)}}^k$$

in which, use i_o instead of $i_{(L+1)}$ if L+1 is an output layer;

Step 6. Weight update: Update weights according to the formula

$$w_{j_{(L-1)}h_L}(t+1) = w_{j_{(L-1)}h_L}(t) + \eta \delta_{h_L}^k x_{j_{(L-1)}}^k$$

Step7. Repeat steps 1-6 until the stop criterion is satisfied, which may be chosen as the mean of the total error

$$< e^k > = < \tfrac{1}{2} \sum_{i_o=1}^{M} (y_{i_o}^k - x_{i_o}^k)^2 >$$

is sufficiently small.