8 Back-Propagation of Errors

learning objectives:

- the differences between "back-propagation" and "counter-propagation"
- why the back-propagation algorithm (a scheme for correcting weights) is the most widely used method
- how two empirical factors have been introduced into the weight-correction equations to overcome some problems
- how a back-propagation net can be used to "model" the simple tennis game introduced in Chapter 7

8.1 General

Back-propagation of errors is not the name of a specific neural network architecture, but the name of a learning method, a strategy for the correction of weights. First introduced by Werbos and later on intensely popularized in connection with neural networks by Rumelhart and coworkers, back-propagation has become the most frequently used method in the field.

In fact, it has become so popular that for many authors the term "neural networks" simply means the back-propagation method. A recent study made by the authors (Reference 8-11) discovered that almost 90 percent of all publications using neural networks in chemistry had used the back-propagation method. Even more interestingly, a number of applications had used this method for clustering, as a lookup table, or as an associative memory – tasks for which other already described methods seem to be far more suitable.

The attractiveness of the back-propagation method comes from the well-defined and explicit set of equations for weight corrections.

These equations are applied throughout the layers, beginning with the correction of the weights in the last (output) layer, and then continuing backwards (hence the name!) towards the input layer (Figure 8-1).

The weight-correction procedure in the back-propagation algorithm does probably not resemble the real process of changing the weights (synaptic strengths) in the brain. However, it must be said that the back-propagation algorithm most closely follows the description of artificial neurons given in Chapter 2.

The back-propagation method, shown schematically in Figure 8-1, is a *supervised* learning method; therefore, it needs a set of **pairs** of objects, the inputs $\boldsymbol{X}_s$ and the targets $\boldsymbol{Y}_s$, ($\boldsymbol{X}_s$, $\boldsymbol{Y}_s$). Because the objects and targets can be represented by sets of real variables, $\boldsymbol{X}_s$ (x_{s1}, x_{s2}, ..., x_{sn}) and $\boldsymbol{Y}_s$ (y_{s1}, y_{s2}, ..., y_{sm}), the resulting network can be regarded as a *model* yielding an m-variate answer for each n-variable input (Figure 8-2).

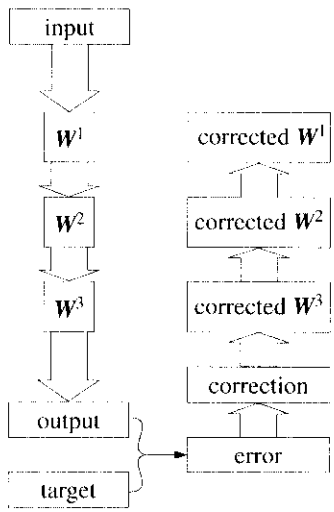

Figure 8-1: Schematic presentation of weight correction by back-propagation of errors.

Compared to standard statistical and pattern recognition methods for supervised learning, three things have to be stressed. First, almost all features known in standard model-generating techniques (choice of variables, representation of objects, experimental design, etc.) play an important role in the back-propagation procedure as well: the troublesome ones as well as the desirable ones.

Second, neural networks trained by back-propagation of errors have one very important advantage: there is no need to know the exact form of the analytical function on which the model should be built. This means that neither the functional type (polynomial, exponential, logarithmic, etc.) nor the number and positions of the parameters in the model-function need to be given.

In order to test the influence of weights on the final output, we would need to be able to determine the effect of **each input** variable on **each weight** separately. In a fully-connected multilayer network, however, **each** input influences **all** weights. Since the influence of a given input on any weight is virtually impossible to predict in advance, our only hope is to input as many objects as possible and try to observe how the weights act. (With neural networks having a million weights and more this is almost impossible.)

Nor can much information be obtained from the inspection of final weights. In any more or less complex back-propagation neural network, a **large** number of weights are trained to yield the correct answers. Therefore, it is very hard, if not impossible, to establish exactly what each weight is responsible for.

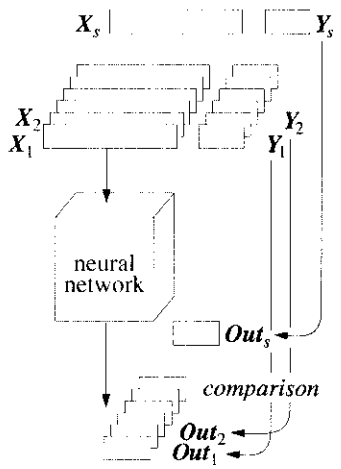

Figure 8-2: Supervised learning requires pairs of data: inputs $\boldsymbol{X}_s$ and targets $\boldsymbol{Y}_s$.

Thus, third, a back-propagation neural network acts as a black box, allowing no physical interpretation of its internal parameters.

8.2 Architecture

The architecture of the network is the main feature influencing the flexibility of the model it generates; that is, the number of layers, the number of neurons in each layer, and the way the neurons are connected.

Although the back-propagation algorithm was primarily designed for use in multilayer neural networks, it can also be applied to neural networks having only one layer. Such a network has, aside from the input units, only one layer of active neurons, the output layer whose weights are to be corrected via backward propagation.

The layers of neurons are usually fully connected. Figure 8-3 shows an architecture consisting of one input and three active layers of neurons (two hidden layers and the output layer). The connections to the biases and the bias weights are indicated by heavier lines and black squares, respectively.

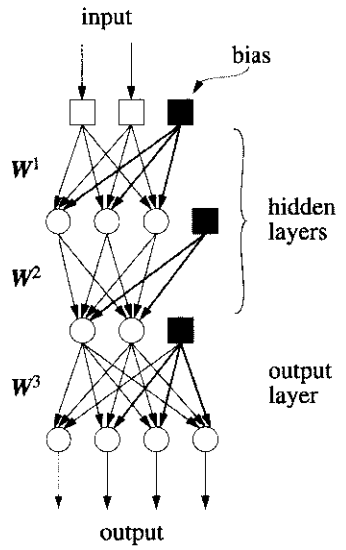

Figure 8-3: Architecture of a back-propagation network with three active layers.

The number of layers as well as the number of neurons in each layer depends on the application for which the neural network is set up, and is, as a rule, determined by trial and error. In the applications reported in the literature, as many as one million weights and as few as ten have been used.

In most cases, neural networks consisting of two active layers – one hidden and one output layer – are used. Only seldom have more than three active layers been linked together. However, quite a number of authors have used separate neural networks linked into a decision hierarchy rather than one large network making all decisions simultaneously (See Chapter 18).

Such designs are used either because of inadequate computer resources, or because not enough data are available to cover the entire variable space.

At any given time during learning by back-propagation of errors, a considerable number of interlayer calculations may occur, involving as many as **three different** layers of neurons. Great care must therefore be taken to make clear which layer is involved in a given operation. Therefore, we will need a comprehensive system of notation.

In order to make things more understandable, each specific item of data (input, output, weights, errors and corrections) will bear a superscript referring to the layer it belongs to.

As shown in Figure 8-4 and as already discussed in Chapter 3, the input to one layer is generally an output from the layer above, and a layer's output is an input to the layer of neurons below.

To avoid confusion, all signals will be labeled as outputs throughout this chapter; that is, the actual input signal that enters the neural network will be labeled as $\boldsymbol{Out}^0$, it will produce $\boldsymbol{Out}^1$ after exiting the first layer, will in turn produce $\boldsymbol{Out}^2$ from the second layer, and so on, until the final output is obtained and labeled $\boldsymbol{Out}^{last}$.

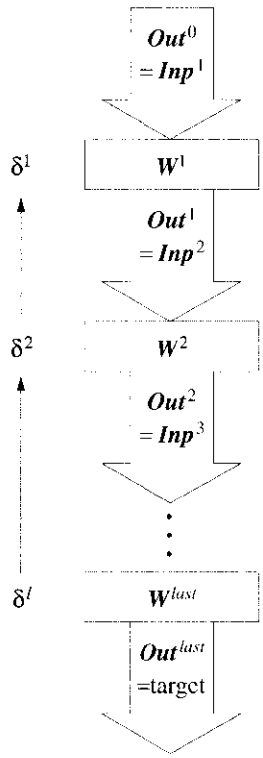

Figure 8-4: Notation used in discussing back-propagation networks: superscripts label the neural levels to which data refer.

8.3 Learning by Back-Propagation

The fact that the learning is supervised is the most important thing about this method, determining all of its other characteristics. Supervised learning means that the weights are corrected so as to produce prespecified ("correct") target values for as many inputs as possible.

The correction of weights, the most important step in the learning process, can be made **after each individual new input** (*immediate correction*), or **after all inputs have been tested** (*deferred correction*). In the first case, the correction is made immediately after the error is detected; in the second, the individual errors for all data pairs are accumulated, and then the accumulated error of the entire training set used for the correction.

Most applications use immediate correction; deferred correction is more rare and does not offer any apparent advantage. Therefore, we will focus on the former.

During learning, the object $\boldsymbol{X}$ (input vector) is presented to the neural network and the output vector $\boldsymbol{Out}$ is immediately compared with the target vector $\boldsymbol{Y}$ ($y_1, y_2, ..., y_m$), which is the correct output for $\boldsymbol{X}$.

Once the actual error produced by the network is known, we have to figure out exactly how to use this to correct weights throughout the entire neural network. Before going into the details, here is the final result in condensed form:

$$\Delta w_{ji}^{l} = \eta \delta_j^l out_i^{l-1} + \mu \Delta w_{ji}^{l\,(previous)} \qquad (8.1)$$

Remember that l is the index of the current layer, j identifies the current neuron, and i is the index of the input source, i.e. the index of the neuron in the upper layer. In this equation, δ_j^l, the error introduced by the corresponding neuron, is calculated in two ways, depending on whether the *last* (output) layer or one of the hidden layers is under consideration. In the following two equations, (8.2) and (8.3), a sigmoidal transfer function is assumed (see also Table 8-1).

According to Equation (8.1), the correction of weights in the l-th layer is composed of two terms, which pull in opposite directions: the first one tends towards a fast "steepest-descent" convergence, while the second is a longer-range function that prevents the solution from getting trapped in shallow local minima. The constant η (the same one as $\eta(t)$ in Section 6.3 and Section 7.4, and serving the same purpose) is called the learning rate and μ is called the momentum constant. By taking into account the correction made on the previous cycle, μ can (to the degree you specify) prevent sudden changes in the direction in which corrections are made: this is particularly useful for damping oscillations. The magnitudes of these constants determine the relative influence of the two terms.

For the output layer ($l = last$) the error δ_j^l is expressed as:

$$\delta_j^{last} = \left(y_j - out_j^{last} \right) out_j^{last} \left(1 - out_j^{last} \right) \tag{8.2}$$

For all other layers l ($l = last - 1$ to 1) the error δ_j^l is calculated by:

$$\delta_j^l = \left(\sum_{k=1}^{r} \delta_k^{l+1} w_{kj}^{l+1} \right) out_j^l \left(1 - out_j^l \right) \tag{8.3}$$

Substituting (8.3) into (8.1) gives the full expression of the weight correction in a hidden layer:

$$\Delta w_{ji}^l = \eta \left(\sum_{k=1}^{r} \delta_k^{l+1} w_{kj}^{l+1} \right) out_j^l \left(1 - out_j^l \right) out_i^{l-1} + \mu \Delta w_{ji}^{l\,(previous)} \tag{8.4}$$

This equation shows that values from three layers influence the correction of weights in any one layer: values from the current layer, l, and the ones above ($l - 1$) and below ($l + 1$).

Section 8.4 below shows how the expressions (8.2) and (8.3) can be derived from the delta-rule (Section 2.3, Equation (2.12)), and how the gradient descent method is used to evaluate these terms. You may

want to skip this section the first time you read this book, but you should probably study it later in order to understand the essentials of this widely-used method.

8.4 The Generalized Delta-Rule

Learning by back-propagation sends the data through the network in one direction, and scans through it, changing weights, in the opposite direction. The correction of the i-th weight on the j-th neuron in the l-th layer of neurons is defined as:

$$\Delta w_{ji}^{l} = w_{ji}^{l\,(new)} - w_{ji}^{l\,(old)} \tag{8.5}$$

Within the j-th neuron in the l-th layer, the weight w_{ji}^{l} links the i-th **input** with the j-th **output** signal. These two links, one with the upper and one with the lower layer, (Figure 8-5), reflect the fact that the error originates partly on the input and partly on the output side.

Figure 8-5: The weight w_{ji}^{l} within the layer l is linked to the next-higher and -lower layer via the i-th input and j-th output.

A well known way to consider both influences was discussed in paragraph 2.3; it is called the delta-rule and is expressed as follows (cf. Equations (2.22) and (2.22a)):

$$\Delta parameter = \eta\ g\,(output\ error)\ f\,(input) \tag{8.6}$$

In its most general form, the delta-rule states that the change of any parameter in an adapting process should be proportional to the input signal **and** to the error on the output side. The proportionality constant η (learning rate) determines how fast the changes of this parameter should be implemented in the iteration cycles.

In order to give Equation (8.6) a more familiar look, it is rewritten by substituting into it the terms used in the neural network approach:

$$\Delta w_{ji}^{l} = \eta\ \delta_j^{l}\ out_i^{\,l-1} \tag{8.7}$$

Formally, Equations (8.6) and (8.7) are identical. The parameter which causes the error is the weight w_{ji}^{l}; its correction Δw_{ji}^{l} is proportional to the term δ_j^{l}, corresponding to the function g above (Equation (8.6)). Although the term out_i^{l-1} is labeled as the output of the $(l-1)$-st layer, it is at the same time the input to the l-th layer, which is consistent with Figure 8.4.

The function f is simply the input itself; therefore, the remaining problem is the estimation of the function δ_j^{l}.

In the back-propagation algorithm the change δ_j^l needed in the correction of the weights is obtained using the so-called *gradient descent* method, the essence of which is the observation that an error ε plotted against the parameter that causes it **must** show a minimum at some (initially) unknown value of this parameter. By observing the slope of this curve, we can decide how to change the parameter in order to come closer to the sought minimum. In Figure 8-6, the value of the parameter to be changed, i.e., the weight of the neuron, is to the right of the minimum; if the derivative $d\varepsilon/dw$ is **positive**, the new value of the parameter should be **smaller** than the old one and vice versa. In other words, we can write:

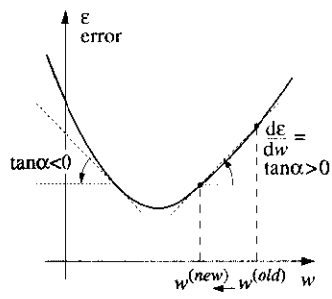

Figure 8-6: The error ε as a function of the weight.

$$\Delta w = w^{(new)} - w^{(old)} = -\kappa d\varepsilon/dw \tag{8.8}$$

where κ is just a positive numerical scaling factor; note the minus sign.

For a specific weight w_{ji}^l in the layer l, the corresponding equation is:

$$\Delta w_{ji}^l = -\kappa \partial \varepsilon^l / \partial w_{ji}^l \tag{8.9}$$

For the sake of clarity of presentation we defer the calculation of the error ε^l to a later part of this section and first discuss the dependence of the error on the weight.

This error function represents that part of the error caused by this particular weight at the output of layer l. Because the error function is a complicated and rather indirect function of the parameters w_{ji}^l, we can evaluate the derivative $\partial \varepsilon^l / \partial w_{ji}^l$ in a stepwise manner employing the chain rule:

The chain rule:

$$F = f(z(y(x)))$$

$$\frac{\partial F}{\partial x} = \left(\frac{\partial F}{\partial z}\right)\left(\frac{\partial z}{\partial y}\right)\left(\frac{\partial y}{\partial x}\right)$$

$$\Delta w_{ji}^l = -\kappa \frac{\partial \varepsilon^l}{\partial w_{ji}^l} = -\kappa \left(\frac{\partial \varepsilon^l}{\partial out_j^l}\right)\left(\frac{\partial out_j^l}{\partial Net_j^l}\right)\left(\frac{\partial Net_j^l}{\partial w_{ji}^l}\right) \tag{8.10}$$

The derivatives of the error function ε^l are calculated consecutively with respect to the values of out_j^l, Net_j^l and w_{ji}^l. Because all derivatives in Equation (8.10) play an important role in the back-propagation algorithm, we will take a closer look at each of them separately, starting with the last one.

Derivative $\partial Net_j^l / \partial w_{ji}^l$. In Equation (3.5) of Section 3.7:

$\partial Net_j^l / \partial w_{ji}^l$

$$Net_j^l = \sum_{i=1}^{m} w_{ji}^l x_i^l \tag{8.11}$$

we have an exact description of the dependence of the net input Net_j^l of the neuron j on the corresponding set of weights. The x_i^l are the components of the vector $\boldsymbol{X}^l$ that is input to the level l. According to our convention, all inputs are written as outputs from the layer above; therefore:

$$x_i^l = out_i^{l-1} \tag{8.12}$$

If Equation (8.11) is written as a sum of products, the derivative of Net_j^l with respect to the particular weight can clearly be seen:

$$\frac{\partial Net_j^l}{\partial w_{ji}^l} = \frac{\partial\left(w_{j1}^l out_1^{l-1} + \ldots + w_{ji}^l out_i^{l-1} + \ldots + w_{jm}^l out_m^{l-1} \right)}{\partial w_{ji}^l} = out_i^{l-1} \tag{8.13}$$

By inserting Equation (8.13) into Equation (8.10) for the corrections of weights, we obtain:

$$\Delta w_{ji}^l = -\kappa \left(\frac{\partial \varepsilon^l}{\partial out_j^l} \right) \left(\frac{\partial out_j^l}{\partial Net_j^l} \right) out_i^{l-1} \tag{8.14}$$

Now we see a correspondence of terms between Equations (8.14) and (8.7) representing the delta-rule correction in its expanded form:

$$\Delta w_{ji}^l = -\kappa \left(\frac{\partial \varepsilon^l}{\partial out_j^l} \right) \left(\frac{\partial out_j^l}{\partial Net_j^l} \right) out_i^{l-1} \tag{8.15}$$

$$\Delta w_{ji}^l = \eta \quad \delta_j^l \quad out_i^{l-1}$$

The above comparison leads directly to the delta-term:

$$\delta_j^l = -\left(\frac{\partial \varepsilon^l}{\partial out_j^l}\right)\left(\frac{\partial out_j^l}{\partial Net_j^l}\right) \tag{8.16}$$

This result is of major importance in the back-propagation model.

$\partial out_j^l / \partial Net_j^l$

Derivative $\partial out_j^l / \partial Net_j^l$. Because the form of the transfer or squashing function (see Section 2.4, Figure 8-7) of the neurons is usually known explicitly, the derivative $\partial out_j^l / \partial Net_j^l$ is not difficult to obtain.

The relationship between out_j^l and Net_j^l was discussed extensively in Section 2.5, where we mentioned the hard-limiter and threshold functions. Although very convenient for evaluation, these two squashing functions are nevertheless often passed over in favor of the more complex sigmoidal transfer function:

$$out_j^l = \frac{1}{1 + exp\left(-Net_j^l\right)} \tag{8.17}$$

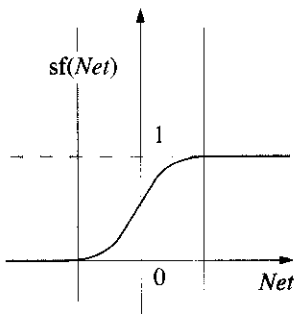

Figure 8-7: Squashing function.

The main reason why (8.17) is used instead of simpler ones is because its derivative can be obtained analytically. As shown in Section 2.5 (Equation (2.32)), function (8.17) can not only be differentiated easily, but its derivative can be expressed in terms of the function itself:

$$\frac{\partial out_j^l}{\partial Net_j^l} = out_j^l\left(1 - out_j^l\right) \tag{8.18}$$

This property is very convenient for use on computers: the derivative is, so to speak, bundled free with the function. The sigmoidal function (8.17) is not the only one whose derivative can be expressed in terms of the function itself; some other squashing functions having this particular property are discussed in Section 8.5.

$\partial \varepsilon^l / \partial out_j^l$

Derivative $\partial \varepsilon^l / \partial out_j^l$. This is the last derivative remaining from Equation (8.10). For this derivative, we have to distinguish two cases, depending on whether or not ε^l is explicitly known; in other words, whether the correction is calculated for:

- the last (output) layer, or
- the hidden layers.

The first case: correction on the last layer. This case is much simpler to handle; since the back-propagation method is a supervised learning method, the error ε^l at the very last (output) level l is always known:

> The error ε^l in the output is the difference between the expected output (target), $\boldsymbol{Y}$ $(y_1, y_2, ..., y_m)$, and the actual output $\boldsymbol{Out}^l$ $(out_1^l, out_2^l, ..., out_m^l)$.

Obviously, the error ε^l can be expressed by subtracting the output out_j^l of each neuron j from the corresponding component y_j of the target vector $\boldsymbol{Y}$:

$$\varepsilon^l = \sum_{j=1}^{n} \left(y_j - out_j^l \right)^2 \tag{8.19}$$

Then, the derivative $\partial\varepsilon^l/\partial out_j^l$ can be obtained easily:

$$\frac{\partial \varepsilon^l}{\partial out_j^l} = \frac{\partial \left(y_1 - out_1^l \right)^2}{\partial out_j^l} + ... + \frac{\partial \left(y_j - out_j^l \right)^2}{\partial out_j^l} + ... = \tag{8.20}$$

$$= -2\left(y_j - out_j^l \right)$$

As only the j-th component in this expansion is dependent on out_j^l, only this component gives a nonzero value in the derivation.

The final expression for the evaluation of the weight corrections in the **last** layer of a neural network is obtained by collecting all three derivatives, $\partial Net_j^l/\partial w_{ji}^l$, $\partial out_j^l/\partial Net_j^l$, and $\partial\varepsilon^l/\partial out_j^l$ in the above procedure from Equations (8.13), (8.18) and (8.20), and inserting them into the expression for the delta-rule, (8.10).

Because the only error we know exactly (Figure 8-8) comes from the last layer, ε^{last}, the superscript l indicating the neuron's layer must be changed from l to *last*: Furthermore, we substitute η for 2κ.

$$\frac{\partial \varepsilon^{last}}{\partial out_j^{last}} = -2\left(y_j - out_j^{last} \right)$$

$$\frac{\partial out_j^{last}}{\partial Net_j^{last}} = out_j^{last}\left(1 - out_j^{last} \right)$$

$$\frac{\partial Net_j^{last}}{\partial w_{ji}^{last}} = out_i^{last-1}$$

resulting in:

$$w_{ji}^{last} = \eta\left(y_j - out_j^{last}\right) out_j^{last}\left(1 - out_j^{last}\right) out_i^{last-1} \qquad (8.21)$$

The second case: corrections on the hidden layers. The expression we have not yet evaluated is the case where the explicit relationship between the error function ε^l and the output out_j^l is not known, which is the case in the **hidden** layers l.

In a hidden layer l, the actual output error ε^l cannot be calculated directly, because the "true" values of their outputs are not known (even in supervised learning). Therefore, the derivative $\partial\varepsilon^l/\partial out_j^l$ can be calculated only if we make some assumptions.

One simple, defensible assumption is that the error ε^l produced by the forward process at a given layer l has been distributed **evenly** over all neurons r in the lower layer $l+1$:

$$\varepsilon^l = \sum_{k=1}^{r} \varepsilon_k^{l+1} \qquad (8.22)$$

The summations run over all r neurons in level $(l+1)$ (Figure 8-9). Therefore, the error at a level l (which is needed to calculate the corrections of weights on the same level) can be obtained by collecting the errors from the level $l+1$ below it.

Assuming hypothesis (8.22), the derivative $\partial\varepsilon^l/\partial out_j^l$ is not hard to figure out; by application of the chain rule and use of Equation (8.22), it follows that:

$$\frac{\partial \varepsilon^l}{\partial out_j^l} = \sum_{k=1}^{r}\left(\frac{\partial \varepsilon_k^{l+1}}{\partial Net_k^{l+1}}\right)\left(\frac{\partial Net_k^{l+1}}{\partial out_j^l}\right) \qquad (8.23)$$

The rightmost derivative $\partial Net_k^{l+1}/\partial out_j^l$ is obtained similarly to the derivative described by Equations (8.11) and (8.13). The net input Net_k^{l+1} is written as in Equation (3.5):

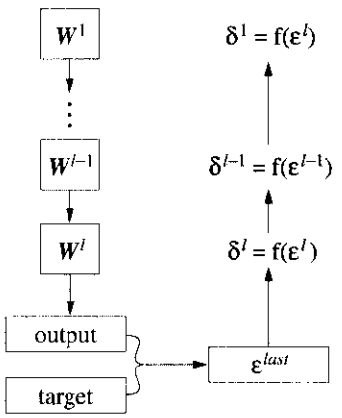

Figure 8-8: For each layer in the network ε^l and $\partial\varepsilon^l/\partial out_j^l$ have to be evaluated. Only the error in the last layer, ε^{last}, is known explicitly.

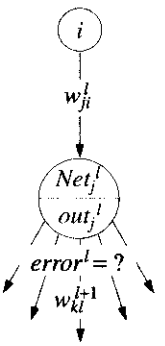

Figure 8-9: Diffusion of error over all lower weights.

$$Net_k^{l+1} = \sum_{j=1}^{m} w_{kj}^{l+1} x_j^{l+1} = \sum_{j=1}^{m} w_{kj}^{l+1} out_j^l = \\ = w_{k1}^{l+1} out_1^l + \ldots + w_{kj}^{l+1} out_j^l + \ldots + w_{km}^{l+1} out_m^l \tag{8.24}$$

Thus, it follows that:

$$\frac{\partial Net_k^{l+1}}{\partial out_j^l} = w_{kj}^{l+1} \tag{8.25}$$

This result is substituted into Equation (8.23):

$$\frac{\partial \varepsilon^l}{\partial out_j^l} = \sum_{k=1}^{r} \left(\frac{\partial \varepsilon_k^{l+1}}{\partial Net_k^{l+1}} \right) w_{kj}^{l+1} \tag{8.26}$$

Due to the differentiation over out_j^l, all terms of the expanded sum (8.24) have vanished except for the j-th term.

The last step in this long story is to apply the chain rule again. This time it should be applied to the remaining derivative $\partial \varepsilon^{l+1} / \partial Net_k^{l+1}$:

$$\frac{\partial \varepsilon^{l+1}}{\partial Net_k^{l+1}} = \left(\frac{\partial \varepsilon^{l+1}}{\partial out_k^{l+1}} \right) \left(\frac{\partial out_k^{l+1}}{\partial Net_k^{l+1}} \right) \tag{8.27}$$

By comparing the right-hand side of (8.27) with the middle parts of the right-hand sides in expressions (8.15), it is easy to deduce that the chained derivative $(\partial \varepsilon^{l+1} / \partial out_k^{l+1})$ $(\partial out_k^{l+1} / \partial Net_k^{l+1})$ is equal to the corrections δ_k^{l+1} on level $l + 1$. Hence:

$$\frac{\partial \varepsilon^{l+1}}{\partial Net_k^{l+1}} = \delta_k^{l+1} \tag{8.28}$$

By inserting the derivative $\partial \varepsilon^{l+1} / \partial Net_k^{l+1}$ into Equation (8.23), the following expression is obtained:

$$\frac{\partial \varepsilon^l}{\partial out_j^l} = \sum_{k=1}^{r} \delta_k^{l+1} w_{kj}^{l+1} \tag{8.29}$$

As when correcting the weights in the output layer, the three derivatives $\partial Net_j^l / \partial w_{ji}^l$, $\partial out_j^l / \partial Net_j^l$ and $\partial \varepsilon^l / \partial out_j^l$ are collected from Equations (8.13), (8.18) and (8.29), and inserted into Equation (8.10) for the delta-rule:

$$\frac{\partial Net_j^l}{\partial w_{ji}^l} = out_i^{l-1}$$

$$\frac{\partial out_j^l}{\partial Net_j^l} = out_j^l \left(1 - out_j^l\right)$$

$$\frac{\partial \varepsilon^l}{\partial out_j^l} = \sum_{k=1}^{r} \delta_k^{l+1} w_{kj}^{l+1}$$

resulting in:

$$\Delta w_{ji}^l = \eta \left(\sum_{k=1}^{r} \delta_k^{l+1} w_{kj}^{l+1} \right) out_j^l \left(1 - out_j^l\right) out_i^{l-1} \qquad (8.30)$$

In Equation (8.30) the learning parameter η has the same meaning and the same function as the parameter η mentioned in Chapters 6 and 7 in Equations (6.5) - (6.7) and (7.3), respectively. In the error back-propagation algorithm η is mostly held constant, however, it can be linearly diminishing during learning according to Equation (6.5), already explained in Chapter 6:

$$\eta(t) = (a_{max} - a_{min}) \frac{t_{max} - t}{t_{max} - 1} + a_{min} \qquad (6.5)$$

The result obtained in Equation (8.30) distinctly shows how values from **three** different layers are involved in the calculation of the weight correction in the hidden layer l:

- the output out_i^{l-1} of the layer above acting as the input i to the l-th layer,
- the out_j^l of the j-th neuron on the current layer l,
- the correction δ_k^{l+1} of the weight w_{kj}^{l+1} from layer $l + 1$.

8.5 Learning Algorithm

Now that we are familiar with the equations used in back-propagation learning, the procedure for the weight correction will be described algorithmically. (For understanding this method, the flow of data in the network and the timing of weight corrections are just as important as the equations.)
The learning procedure involves the following steps:

- input an object $\boldsymbol{X}$ $(x_1, x_2, ..., x_m)$
- label the components x_i of the input object $\boldsymbol{X}$ as out_i^0 and add a component 1 for bias; the input vector thus becomes: $\boldsymbol{Out}^0$ $(out_1^0, out_2^0, ..., out_m^0, 1)$
- propagate $\boldsymbol{Out}^0$ through the network's layers by consecutively evaluating the output vectors $\boldsymbol{Out}^l$; for this, we use the weights w_{ji}^l of the l-th layer and the output out_i^{l-1} from the previous layer (which acts as input to layer l):

$$out_j^l = f\left(\sum_{i=1}^{m} w_{ji}^l out_i^{l-1} \right)$$

  where f is the chosen transfer function, e.g. the sigmoidal function.
- calculate the correction factor for all weights in the output layer δ_j^{last}, by using its output vector $\boldsymbol{Out}^{last}$ and the target vector $\boldsymbol{Y}$:

$$\delta_j^{last} = \left(y_j - out_j^{last} \right) out_j^{last} \left(1 - out_j^{last} \right)$$

- correct all weights w_{ji}^{last} on the last layer:

$$\Delta w_{ji}^{last} = \eta \delta_j^{last} out_i^{last-1} + \mu \Delta w_{ji}^{last\,(previous)}$$

- calculate consecutively layer by layer the correction factors δ_j^l for the hidden layers from $l = last - 1$ to $l = 1$:

$$\delta_j^l = \left(\sum_{k=1}^{r} \delta_k^{l+1} w_{kj}^{l+1} \right) out_j^l \left(1 - out_j^l \right)$$

– correct all weights w_{ji}^{l} on the layer l:

$$\Delta w_{ji}^{l} = \eta \delta_{j}^{l} out_{i}^{l-1} + \mu \Delta w_{ji}^{l(previous)}$$

– repeat the procedure with a new input-target pair ($\boldsymbol{X}$, $\boldsymbol{Y}$).

Due to the widespread use of back-propagation learning, it is important to comment on some of the steps listed in the learning algorithm above, and to point out some problems which may arise when applying it.

Before the actual learning begins, three things have to be done:

– initial choice of the neural network architecture

– randomization of initial weights, and

– selection of the learning rate η and momentum constant μ.

The initial architecture of the neural network (the number of layers, neurons and weights) is only a starting guess. You may want to modify the architecture after you see how the network performs during the learning or testing phase; this will be discussed later on in this Chapter.

The weights are initialized by setting them to small random numbers. Be sure that not all weights are equal to zero. Usually this is done automatically by the program package, but you will probably be asked to specify the interval within which the weights should be randomized. A typical choice for a layer l is the interval between $\{-1/n, 1/n\}$, n being the number of all weights in that layer. Because the weights will be changed anyway, the exact starting values have no particular significance.

The most important of these choices, apart from the neural network architecture, is the *learning rate constant* η (Equations (8.1) – (8.3)), which determines the speed at which the weights change; if they change too quickly, the procedure may end up in a local minimum (the steepest way downhill does not necessarily lead to the global minimum). The gradient descent method is justified for continuous functions and requires infinitesimally small corrections of weights. This is not possible in the back-propagation approach. The trick is to find a reasonable tradeoff between fast learning and converging to the lowest minimum.

The learning rate constant η is generally obtained by trial and error; good starting values are between 0.3 and 0.6.

The *momentum constant* μ determining the size of the momentum term has a close connection with η. Figure 8-10 shows its influence. In a real multivariate system, of course, the situation is much more complex than shown in this one-dimensional picture; the paths among the local minima are very hard to follow or predict accurately.

As can be seen from Equation (8.1), the momentum term takes into account the **most recent** correction of weights; this is how μ gets its power to prevent sudden changes in the direction in which the solution is being moved.

Let's assume for simplicity that $\mu = \eta$; if, further, the two terms of Equation (8.1) are equal, then:

$$\delta_j^l out_i^{l-1} = -\Delta w_{ji}^{l\,(previous)} \tag{8.31}$$

This would cause the weight w_{ji}^l **not to be changed at all**, even though the current cycle taken by itself recommends a change equal to $\eta \delta_j^l \, out_i^{l-1}$.

Hence, a value of μ larger than η tends to suppress oscillations, but possibly at the price of overlooking some narrow ways to the global minimum. The learning rate η and the momentum constant μ may need to be systematically changed during the learning process. Then, they are usually decreased during the iteration process.

The inclusion of the momentum term considerably increases the need for computer memory, since in addition to the current values of the weights, we must store their values on the previous cycle as well.

Augmenting the input vectors by including the bias is generally an automatic program feature. Normalization of input vectors, a desirable if not always mandatory procedure, is included in standard packages for neural network calculations by means of algorithms from linear scaling in the required interval, to a statistic auto-scaling that ensures that each variable will have a mean value of zero and a standard deviation of one.

It should be noted that the expression $out_j^l \, (1 - out_j^l)$ (Equations (8.2) and (8.3)) results from the derivative $\partial out_j^l / \partial Net_j^l$ given by Equation (8.18). In general, the squashing function does not always need to have the form of (8.17); if a different squashing function is used, the derivative $\partial out_j^l / \partial Net_j^l$ will have a different form from the one given by Equation (8.18) (see Table 8-1). Therefore, in Equations (8.2) and (8.3), the expression $out_j^l (1 - out_j^l)$ can be different. Sometimes it is advisable to write the derivative $\partial out_j^l / \partial Net_j^l$ in a shorter form:

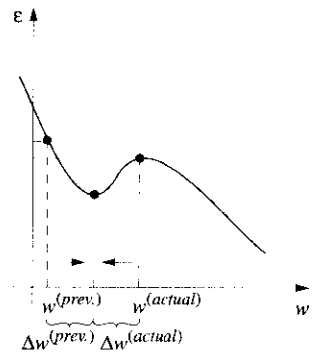

Figure 8-10: Escape from a local minimum is possible if the momentum term is large enough to "push" the weight over the barrier.

$$\frac{\partial out_j^l}{\partial Net_j^l} = f'\left(Net_j^l\right) \tag{8.32}$$

Table 8-1 gives a few examples of functions that can be used in the back-propagation evaluation of weight corrections.

As can be seen from Figure 8-11, the second function in Table 8-1, the hyperbolic tangent, $\tanh(x)$, is especially attractive. It has its inflection point at the origin and two asymptotes at ± 1. This makes it useful in applications where the input data lie between +1 and −1.

$f(x)$	$f'(x)$	$f'(f(x))$
$\frac{1}{1+e^{-x}}$	$\frac{-e^{-x}}{\left(1+e^{-x}\right)^2}$	$f(x)(1-f(x))$
$\frac{1-e^{-x}}{1+e^{-x}}$	$\frac{2e^{-x}}{\left(1+e^{-x}\right)^2}$	$1-(f(x))^2$
$\frac{x}{1+x}$	$\frac{1}{(1+x)^2}$	$(1-f(x))^2$
$1-e^{-x}$	e^{-x}	$1-f(x)$

Table 8-1: Functions that can be used as transfer functions in artificial neurons; these have the property that the first derivative is expressible in terms of the function itself.

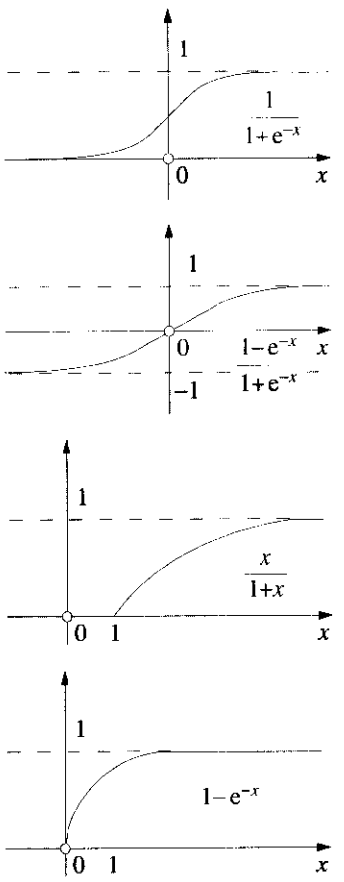

Figure 8-11: Functions for which the derivative can be expressed in terms of the function itself. The hyperbolic tangent, $\tanh(x)$, is especially attractive because it is antisymmetric with respect to the coordinate origin.

The last two functions are used only for argument values larger than or equal to 1; for argument values smaller than 1 they are regarded as equal to zero.

In the back-propagation of errors learning scheme, one pass of all objects through the network is called one iteration *cycle* or one *epoch*. As a rule, many hundreds or even thousands of cycles are necessary to achieve convergence in this learning scheme. Because the number of weights is large even for a medium sized net, the convergence can be quite a lengthy procedure, and may not even be achieved at all.

8.6 Example: Tennis Match

There are many different applications where the back-propagation method can be useful in neural networks. Since back-propagation is usually a supervised learning method, training and test sets with

known answers (targets) have to be selected and run through the network until it:

- **recognizes** the training data, and
- **predicts** a proper association for new input data.

Unlike the counter-propagation method (Chapter 7), which is used to create lookup tables, the back-propagation method is used to develop models. This means that for each different input vector, a different (though possibly very similar) output vector is obtained. In order to compare these two neural networks (both of which are supervised learning methods), we will use the same example here as in Chapter 7 (learning tennis).

When developing a neural network application, careful selection of data is utterly crucial. We will see from this example that data selection is even more critical in back-propagation than in other neural network methods.

See Section 7.5 for the details of the tennis problem. On the basis of two variables provided by the trainer, the trainee (the neural network) should be able to return the stroke by choosing her own two parameters correctly.

The input variables are the position, x, of the trainer on his service line and the angle β at which the ball moves towards the trainee.

The desired output values (targets) that the trainee is trying to learn are the position, y, where the ball will cross her service line, and the angle γ at which she should place her racket so that the ball will bounce off towards the point z on the trainer's service line (Figure 8-12).

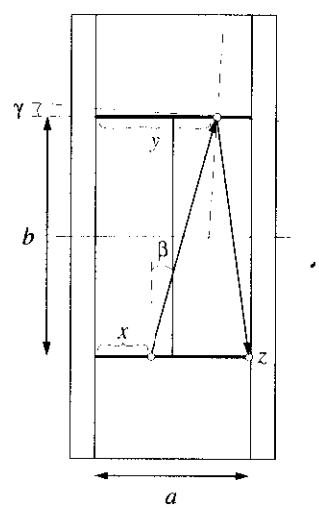

Figure 8-12: Tennis example. Input values are pairs of (x, β), while outputs are pairs (y, γ).

8.6.1 Choice of Data

Before beginning the training, we must construct a set of data that will be representative of all possible cases; here, we will produce sixteen such training pairs, using trigonometry rather than resorting to an actual tennis court.

Theoretically, the pairs of input data and desired outputs (targets), can be expressed as follows (Figure 8-12):

$$y = x + b\tan\beta$$
$$\gamma = 0.5\left(\beta - \arctan\left(\frac{z-y}{b}\right)\right) \tag{8.33}$$

The tangent function of small angles can be approximated with good precision by the angles themselves; hence, the outputs y and γ can be approximated by:

$$y = x + b\beta$$
$$\gamma = 0.5\left(\beta - \frac{z-y}{b}\right) \tag{8.34}$$

Regardless of whether the data are simulated or real, the question we are immediately faced with in any type of modeling is: how many data (pairs of input and target variables) do we need to obtain a good model? Ten, a hundred, a thousand?

In standard modeling techniques where an analytical function (say, a quadratic polynomial of two variables) is to be used as a model, the answer is not complicated: there must be at least one more data point than there are parameters in the model (Figure 8-13). In the case of neural networks, however, where there is no analytical form to fall back on, the answer to this question is much harder to come by.

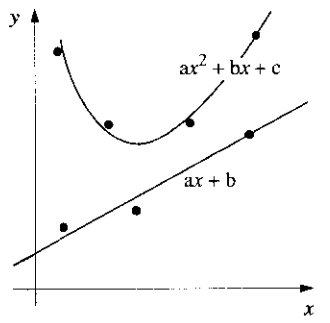

Figure 8-13: Standard models, such as the straight line ($ax + b$) or parabola ($ax^2 + bx + c$), need at least one more data point than there are parameters to be determined.

Because the output variables do not seem to be very complicated, let us start by taking sixteen data points evenly distributed over the entire variable space (the tennis court). Table 8-2 lists all sixteen data points (two inputs and two targets each) chosen carefully to represent all of the conditions that can occur.

The angles are given in radians; positive and negative angles indicate clockwise and counterclockwise directions with respect to the perpendicular axis.

8.6.2 Architecture and Parameters of the Network

Once we have the data, we then must decide on the network architecture to be used: how many layers, how many neurons, how many weights?

Because the numbers of input and output neurons are known (two inputs and two outputs – see Figure 8-14), the input and output layers are defined; but the "inner" architecture of the hidden layers is still entirely up to us.

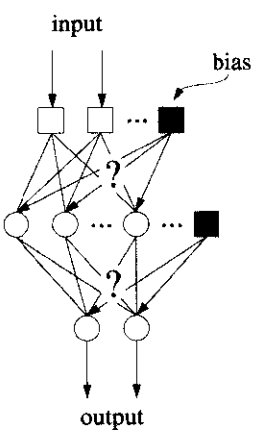

Figure 8-14: The architecture of the network is not yet defined.

no.	inputs		targets	
	x	β	y	γ
1	0.000	0.000	0.000	–0.232
2	0.333	–0.165	0.000	–0.314
3	0.667	–0.322	0.000	–0.393
4	1.000	–0.465	0.000	–0.464
5	0.000	0.165	0.333	–0.078
6	0.333	0.000	0.333	–0.161
7	0.667	–0.167	0.333	–0.243
8	1.000	–0.322	0.333	–0.322
9	0.000	0.322	0.667	0.078
10	0.333	0.165	0.667	0.000
11	0.667	0.000	0.667	–0.083
12	1.000	–0.165	0.667	–0.165
13	0.000	0.465	1.000	0.232
14	0.333	0.322	1.000	0.161
15	0.667	0.165	1.000	0.083
16	1.000	0.000	1.000	1.000

Table 8-2: The sixteen calculated input-target pairs for back-propagation learning in the tennis problem.

In a back-propagation network, the output is obtained directly from the neurons in the output layer. Therefore, it is advisable to scale each component of the target to lie between 0 and 1. Due to the nonlinear character of the transfer function, it is better to scale the entire output to lie between 0.1 and 0.9 or even between 0.2 and 0.8 (Figure 8-15). Scaling confers three advantages:

- easier comparison of the output and target data,
- proper calculation of RMS (root-mean-square) error,
- later recalculation of the correct answer from the output neuron.

First, we will try a network having one hidden layer with six neurons. Later on we will investigate different numbers of hidden neurons. Each of the six neurons has three weights to accommodate two inputs and one bias (Figure 8-16); the seven weights on both output neurons receive the output from each of those six neurons plus a bias.

The last thing we do before running the example is to choose the learning rate η and momentum μ; our first choice will be 0.5 and 0.9, arbitrarily. We shall soon see that this choice is not a very good one.

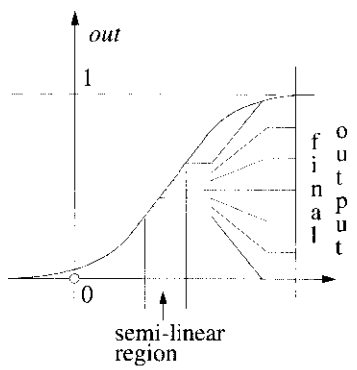

Figure 8-15: Scaling of the output into the linear response region.

Now we can run the first learning experiment with sixteen data points. These are randomly mixed and then input to the above (2 x 6 x 2) – or with biases, (3 x 7 x 2), – network for 1000, 2000, 3000 and 4000 epochs.

Recognition (Recall)

After each input (not after each epoch), all the weights are changed according to Equations (8.2) to (8.4). At the end of each thousand epochs of learning, the RMS error (Equation (7.6)), is calculated. The results of learning the training patterns – what we call the **recognition** or **recall** of the objects – are very good at first glance. The RMS error is 0.008, or slightly less than 1%. Table 8-3 explicitly shows the recall results of the sixteen target pairs. Table 8-3 also indicates what an error of less than 1% actually looks like, in terms of actual data.

no.	*y*			*γ*		
	target	output	error	target	output	error
1	0.000	0.003	0.003	–0.232	–0.236	0.004
2	0.000	0.001	0.001	–0.314	–0.307	0.007
3	0.000	0.001	0.001	–0.393	–0.392	0.001
4	0.000	0.001	0.001	–0.464	–0.461	0.003
5	0.333	0.320	0.013	–0.078	–0.073	0.005
6	0.333	0.326	0.007	–0.161	–0.156	0.004
7	0.333	0.315	0.018	–0.243	–0.235	0.008
8	0.333	0.312	0.021	–0.322	–0.325	0.003
9	0.667	0.665	0.002	0.078	0.091	0.013
10	0.667	0.659	0.008	0.000	–0.006	0.006
11	0.667	0.651	0.016	–0.083	–0.082	0.001
12	0.667	0.661	0.006	–0.165	–0.158	0.007
13	1.000	0.998	0.002	0.232	0.223	0.009
14	1.000	0.999	0.001	0.161	0.168	0.007
15	1.000	0.998	0.002	0.083	0.078	0.005
16	1.000	0.996	0.004	0.000	0.009	0.009

Table 8-3: The output from sixteen training data after 4000 epochs of learning on (2 x 6 x 2) back-propagation network. The error column contains the absolute differences between the targets and the outputs.

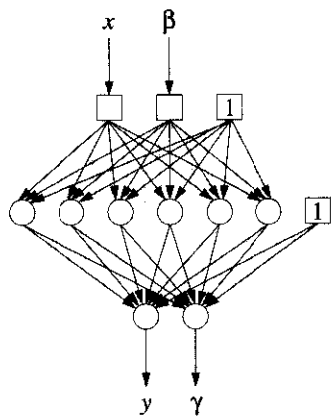

Figure 8-16: The first network used for the application of back-propagation learning to the tennis problem.

It can be seen clearly that the average absolute error is about 0.006 and that it can be as small as 0.001 or as large as 0.021. If the relative errors are important for a given application, one has to be aware that the results at lower values always have **larger** relative errors

compared to those which are close to 1. Therefore, the training set should contain proportionally more objects giving low outputs in order to achieve the same relative error over the entire interval.

The next question is: can the learning rate and momentum significantly influence the results of learning? To answer this question the network was retrained several times with the same sixteen objects, each time with a different choice of the learning rate η and momentum parameter μ.

The result (Figure 8-17) is quite interesting. Each *iso*-RMS error line on the two-dimensional η vs. μ chart represents a constant value of the RMS error. The full circles represent the RMS values obtained for the learning procedures actually performed as described above.

The *iso*-RMS lines in Figure 8-17 tell us that reasonable RMS errors can be achieved in our example with higher learning rates, even if the momentum is zero. Using the information obtained from these preliminary runs, the best combination of the learning rate and momentum constant can be selected.

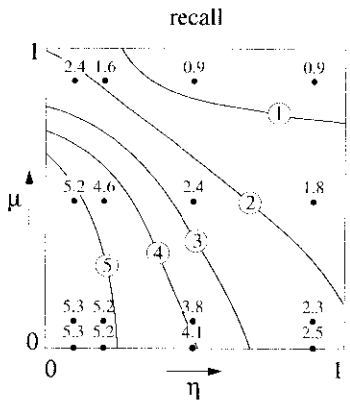

Figure 8-17: Plot of RMS error (in %) in training versus learning rate (η) and momentum parameter (μ), showing lines of constant error.

Prediction

After testing the recall or recognition ability of the network comes the fun part: checking the network's ability to predict output from new inputs.

To do this, we took 1000 random (x, β) pairs and calculated the correct answer pairs, (y, γ), using the "tennis equations" (Equations (8.33)). The calculated values were then compared to the values output by the neural network. See Table 8-4 for a small sample of these results.

Table 8-4 gives a completely different impression from Table 8-3; the errors can be larger by almost an order of magnitude than when simply recalling the training inputs. In the worst case (no. 2 in Table 8-4), the correct position for the player is 0.1 units from the side line, while the program puts her/him almost at the line, 0.008. And the errors in the angle, although at first glance smaller than the errors in position, are quite large, especially since the error in the racket angle is doubled when we consider both the trajectories towards the racket and away from it (after the stroke).

> Always evaluate errors in view of your actual application and needs.

Now let's examine the influence of the learning constant η and the momentum parameter μ on the RMS error. Again, the *iso*-RMS-error lines are plotted against η and μ. The same network as for Figure 8-17 is used for Figure 8-18.

Figure 8-18 shows that the high values of η and μ that yield the network having the **best recall** (recognition) are associated with the **worst prediction** ability.

From both Figures 8-17 and 8-18, we can conclude that for the tennis problem the best choice of these parameters lies in the shaded area of Figure 8-19.

The sum of η and μ should be more or less equal to one:

$$\eta + \mu \approx 1 \qquad (8.35)$$

It has to be stressed that this relationship will certainly depend on the kind of data being investigated and therefore might not necessarily be the best one in all cases.

Table 8-4 gave us a qualitative indication that our system for modeling tennis is not very good; Figure 8-19 confirms this: with the present training data set and present size of the neural network (2 x 6 x 2) we cannot expect better results than about 4% RMS error. How can we make our "tennis brain" smarter?

no.	y			γ		
	calc.	output	error	calc.	output	error
1	0.074	0.020	0.054	–0.207	–0.208	0.001
2	0.097	0.008	0.089	–0.414	0.439	0.025
3	0.348	0.337	0.011	–0.301	–0.296	0.005
4	0.412	0.385	0.073	–0.048	–0.044	0.004
5	0.671	0.665	0.006	–0.011	–0.017	0.006
6	0.755	0.833	0.078	0.108	0.125	0.013
7	0.902	0.985	0.083	0.110	0.111	0.001
8	0.937	0.993	0.056	0.187	0.201	0.014

Table 8-4: Output from new test data, presented after training: 4000 epochs in the (2 x 6 x 2) back-propagation network with $\eta = 0.5$ and $\mu = 0.9$. The error column contains absolute differences between the targets and the outputs.

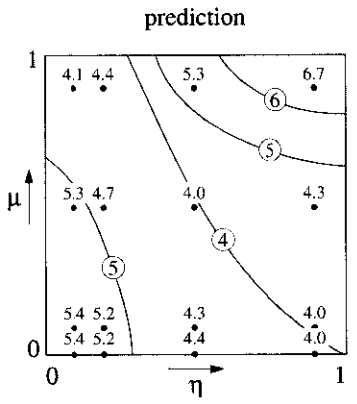

Figure 8-18: Plot of RMS error (in %) in prediction output versus learning rate and momentum, showing lines of constant error.

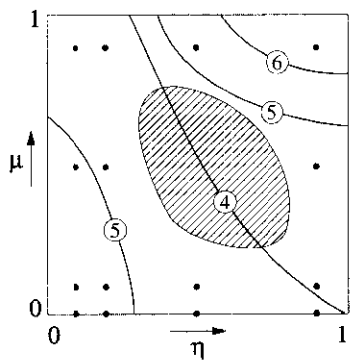

Figure 8-19: The optimum region for the learning rate η and momentum μ for the tennis problem.

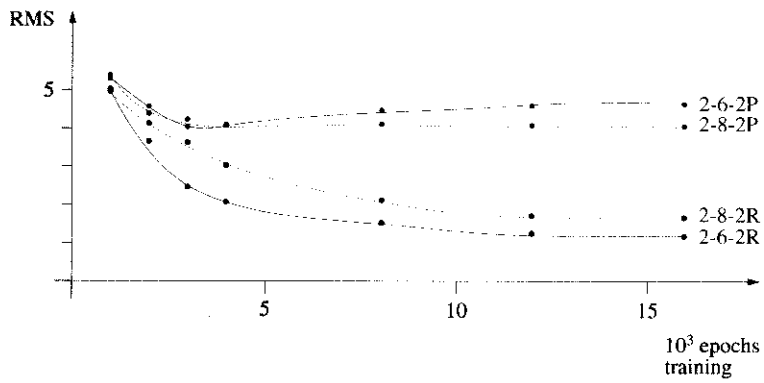

Figure 8-20: Recall (R) and prediction (P) error vs. number of training epochs.

8.6.3 Number of Neurons in the Hidden Layer: Overtraining

There are three things that can still be tried to improve the results:

- enlarge the number of learning steps (epochs), i.e. train the network longer,
- change the network design,
- expand the training data set.

(In our further attempts, we will hold η and μ at the values 0.5 and 0.6.)

First, let's increase the number of training steps from 4000 to 8000, 12000, and 16000 epochs. The resulting RMS errors for recall and prediction are given in Figure 8-20.

The recall performance of the original network (bottom curve) improves monotonically with longer training (though the rate of improvements slows down considerably after 4000 epochs); but, surprisingly, its ability to handle new data (top curve) actually gets worse!

This is known as the *overtraining effect*. Overtraining can be explained as a consequence of parameter redundancy; that is, the system has more parameters than are needed for the solution of the problem. In curve-fitting, we might see this in a polynomial with too many terms: it can make a "better" fit to a set of data by adapting to, rather than smoothing out, the "wiggles" caused by noise.

For example, suppose we are fitting three points in the xy-plane to a quadratic function $ax^2 + bx + c$ instead of to a straight line $ax + b$ (Figure 8-21). In spite of the fact that the straight line cannot go

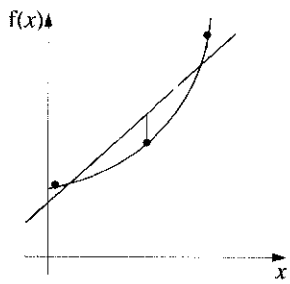

Figure 8-21: In constructing a model, more adjustable parameters are **not** necessarily better. In this example, a three-parameter curve fits the data better than a two-parameter (linear) one – but it may be fitting noise. The "worse" fit may make better predictions.

through all three points (full circles) exactly, as the quadratic function can, it is possible that the prediction of other points (not taken into account before; open circles) will be much better if based on the straight line than based on the quadratic function.

Based on the overtraining data, it appears that learning with about 4000 epochs should be optimum in our case.

The next thing to consider is changing the neural network design and/or the training data set. While the network design can be changed at any time, getting more data is sometimes not as easy as it may seem. In the tennis example, it is easy to calculate as many as we like; but when the data come from an expensive experiment, it's a different story!

> If your problem just does not involve enough different data, you might do well to consider some other, more standard approaches and forget about neural networks.

If, in spite of having a small data collection, you still insist on using the neural network approach as a solution to your problem, then your best chance is to find the **smallest** adequate design!

In the tennis problem, we will try both things: changing the design of the network, and training the network with larger numbers of objects.

For the first, we select four designs: (2 x 4 x 2), (2 x 6 x 2) (the original), (2 x 8 x 2), and (2 x 10 x 2) (Figure 8-22); for the second, we select three training sets containing 16, 36, and 100 objects at specific locations in the variable space, and three more containing 100, 200, and 300 randomly chosen ones. These six data sets are used to test all four networks to see which combination gives the best results.

The RMS errors for recall and predictions for these twenty-four cases are very informative. First of all, increasing the number of neurons neither improves the recall nor the prediction when only sixteen training data are used. The same behavior can be observed in all other cases too, although to a lesser extent; when there are 300 training data, the situation is not so clear, because with more data in the training set we are allowed to have more weights to adjust (Table 8-5 and Figures 8-23 and 8-24).

Nevertheless, it can be seen from Figure 8-24 that the best prediction ability is achieved when the number of weights is

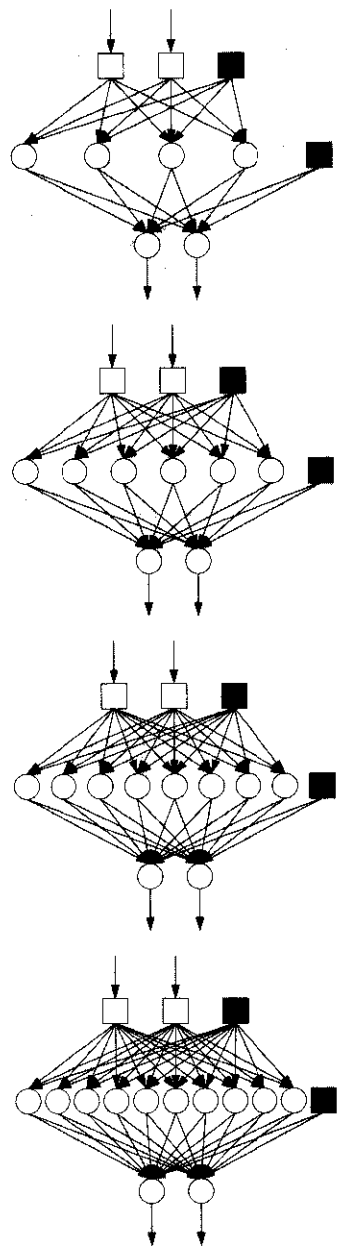

Figure 8-22: Four networks (with 22, 32, 42, and 52 weights, resp.) used for back-propagation learning in the tennis example.

o \ *h* / *w*	4 22	6 32	8 42	10 52	
16	2.91	2.10	3.05	3.25	R
	4.66	4.09	4.11	4.08	P
36	3.26	1.79	1.93	2.19	R
	4.06	3.45	3.42	3.51	P
100	2.07	1.60	1.25	1.34	R
	3.50	3.37	3.24	3.51	P
100r	1.60	1.11	0.95	0.93	R
	3.66	3.36	3.29	3.29	P
200r	1.49	1.12	0.80	0.70	R
	3.46	3.31	3.23	3.23	P
300r	1.44	0.77	0.70	0.70	R
	3.41	3.17	3.17	3.18	P

Table 8-5: RMS error of the recall (R) and prediction (P), as a function of the size of the network (h = number of hidden neurons; w = total number of weights) and the size of the training set (o = number of objects; r = randomly selected). In all cases, the training period was 4000 epochs.

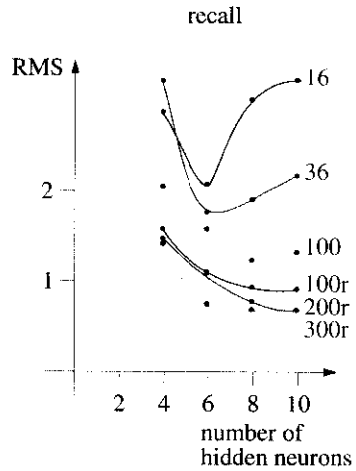

Figure 8-23: The RMS error of recall as a function of the number of neurons in the hidden layer for different sizes of the training sets. The numbers labeled with r (e.g. 100r) refer to randomly chosen training sets.

approximately equal to or slightly smaller than the number of different objects in the training set. When the number of weights is larger than the number of **different** objects in the training set, overtraining appears.

After all this testing, we are finally in a position to select what seems to be the appropriate training set and the best suited network for our problem.

All training sets with more than 100 objects generate networks that differ only slightly in the RMS error of the predictions, regardless of whether the objects are randomly selected or are sampled equidistantly over the variable space. More than 200 objects would prolong the learning time unnecessarily; thus, a choice of 200 objects seems optimum.

Concerning architecture, the (2 x 8 x 2) design with 52 weights seems adequate: it shows a remarkable stability in the RMS error for predictions, even over long periods of training, and gives the best RMS error of all networks we tested (Figure 8-24).

(The learning rate and the momentum were 0.5 and 0.6. Any other values within the ±0.1 interval would do equally well).

The network obtained after 4000 epochs is shown in Figure 8-25.

The predictive results of the network shown at the right are not overwhelmingly good (RMS error of 3.1). Nevertheless, they show very well what can be expected from back-propagation learning: what the problems are and how to tackle some of them.

Additional examples, with practical hints and descriptions of problems, are given in Part IV.

Modeling is not the most favorable example of back-propagation learning in neural networks; for example, sorting multivariate objects into classes might be more interesting or even more exciting, certainly more "successful"; but this example better illustrates the method's limitations.

In classification problems, the output layer consists of as many neurons as there are classes of objects, and the RMS error at each output neuron need not be as small as in the tennis example. For applications producing binary/bipolar outputs, it is adequate for the "1" output to lie between 1.0 and 0.6, and "0", at 0.4 to 0.0. Modeling a system with continuous output values, on the other hand, requires a precision of at least a few percent.

A number of authors claim that their back-propagation networks are able to **generalize** the solution of the problem; by this, they mean that the network yields reliable answers even outside the area of the variable space from which the objects for the training are taken. Unless it is shown beyond a reasonable doubt that the network has not been overtrained (i.e., that it does not fit the training data so closely that its predictive ability is compromised), such claims are, to put it mildly, exaggerated.

> The most important aspect of designing a network for back-propagation learning is to ensure that it will not become overtrained.

See Section 16.5 for a detailed comparison of the back-propagation and counter-propagation learning techniques.

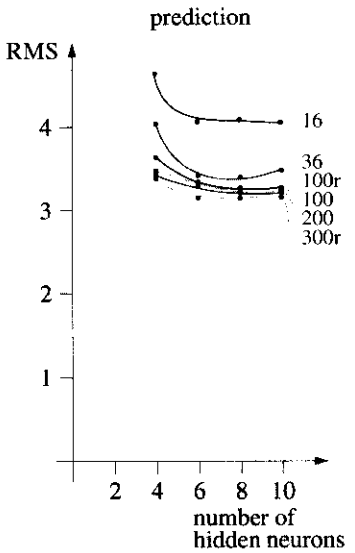

Figure 8-24: The RMS error of predictions as a function of the number of neurons in the hidden layer and the sizes of the training sets.

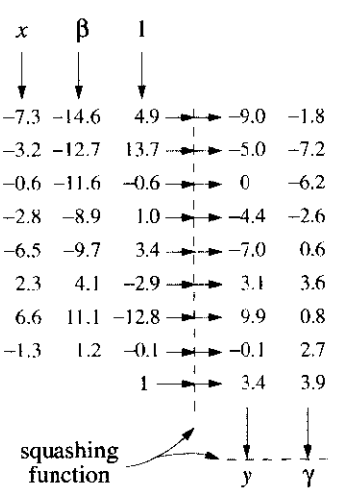

Figure 8-25: The (2 x 8 x 2) neural network obtained after 4000 epochs.

8.7 Essentials

- the back-propagation of errors enables the correction of all weights in a multilayer network
- this procedure uses a supervised learning method, i.e. it requires the answers (targets) to the inputs to be known in advance
- the delta-rule and the gradient descent method are the basis for the correction of weights
- an empirical learning rate constant, η, determines the speed of learning in an iterative procedure
- the inclusion of the momentum term, μ, into the corrective equations is necessary to avoid being trapped in small local minima
- back-propagation nets are used as models, especially when the analytic form of the model relationship is not known
- finding the proper network design (number of layers, number of neurons, and number of weights) is usually a trial and error procedure

- **total weight change:**

$$\Delta w_{ji}^{l} = \eta \delta_j^{l} out_i^{l-1} + \mu \Delta w_{ji}^{l(previous)} \tag{8.1}$$

error in the last layer

$$\delta_j^{last} = \left(y_j - out_j^{last} \right) out_j^{last} \left(1 - out_j^{last} \right) \tag{8.2}$$

error in the hidden layer

$$\delta_j^{l} = \left(\sum_{k=1}^{r} \delta_k^{l+1} w_{kj}^{l+1} \right) out_j^{l} \left(1 - out_j^{l} \right) \tag{8.3}$$

- **required weight change:**

in the last layer

$$\Delta w_{ji}^{last} = \eta \left(y_j - out_j^{last} \right) out_j^{last} \left(1 - out_j^{last} \right) out_i^{last-1} \tag{8.21}$$

in the hidden layer

$$\Delta w_{ji}^{l} = \eta \left(\sum_{k=1}^{r} \delta_k^{l+1} w_{kj}^{l+1} \right) out_j^{l} \left(1 - out_j^{l} \right) out_i^{l-1} \tag{8.30}$$

if changing of learning rate is required:

$$\eta(t) = (a_{max} - a_{min}) \frac{t_{max} - t}{t_{max} - 1} + a_{min} \tag{6.5}$$

8.8 References and Suggested Readings

8-1. P. Werbos, "Applications of Advances in Nonlinear Sensitivity Analysis", in *System Modeling and Optimization: Proc. of the Int. Federation for Information Processes*, Eds.: R. Drenick, F. Kozin, Springer Verlag, New York, USA, 1982, pp. 762 – 770.

8-2. D. E. Rumelhart, G. E. Hinton and R. J. Williams, "Learning Internal Representations by Error Propagation", in *Parallel Distributed Processing: Explorations in the Microstructures of Cognition*, Eds.: D. E. Rumelhart, J. L. McClelland, Vol. *1*, MIT Press, Cambridge, MA, USA, 1986, pp. 318 – 362.

8-3. W. P. Jones and J. Hoskins, "Back-Propagation, A Generalized Delta-Learning Rule", *Byte*, October 1987, 155 – 162.

8-4. L. B. Elliot, "Neural Networks – Conference Update and Overview", *IEEE Expert*, Winter 1987, 12 – 13.

8-5. R. P. Lippmann, "An Introduction to Computing with Neural Nets", *IEEE ASSP Magazine*, April 1987, 4 – 22; P. D. Isserman and T. Schwartz, "Neural Networks, Part 2: What are They and Why is Everybody so Interested in Them Now?", *IEEE Expert*, Spring 1988, 10 – 15.

8-6. T. Kohonen, "An Introduction to Neural Computing", *Neural Networks* **1** (1988) 3 – 16.

8-7. D. S. Touretzky, D. A. Pomerleau, "What's Hidden in the Hidden Layers?", *Byte*, August 1989, 227 – 233.

8-8. W. Kirchner, "Fehlerkorrektur im Rückwärtsgang, Neuronales Backpropagation-Netz zum Selbermachen", *c't*, Heft 11 (1990) 248 – 257.

8-9. J. Dayhoff, *Neural Network Architectures, An Introduction*, Van Nostrand Reinhold, New York, USA, 1990.

8-10. H. Ritter, T. Martinetz and K. Schulten, *Neuronale Netze, Eine Einführung in die Neuroinformatik selbstorganisierender Netzwerke*, Addison-Wesley, Bonn, FRG, 1990.

8-11. J. Zupan and J. Gasteiger, "Neural Networks: A New Method for Solving Chemical Problems or Just A Passing Phase?", *Anal. Chim. Acta.* **248** (1991) 1 – 30.