CHAPTER 12

Back-Propagation

12. 1 INTRODUCTION

As we have seen in the previous chapter, a single-layer network has severe restrictions: the class of tasks that can be accomplished is very limited. In this chapter we will focus on feed forward networks with layers of processing units.

Minsky and Papert showed in 1969 that a two layer feed-forward network can overcome many restrictions, but did not present a solution to the problem of how to adjust the weights from input to hidden units. An answer to this question was presented by Rumelhart, Hinton and Williams in 1986, and similar solutions appeared to have been published earlier (Parker, 1985; Cun, 1985).

The central idea behind this solution is that the errors for the units of the hidden layer are determined by back-propagating the errors of the units of the output layer. For this reason the method is often called the back-propagation learning rule. Back-propagation can also be considered as a generalization of the delta rule for non-linear activation functions and multilayer networks.

12.2 MULTI - LAYER FEED - FORWARD NETWORKS

A feed-forward network has a layered structure. Each layer consists of units, which receive their input from units from a layer directly below and send their output to units in a layer directly above the unit. There are no connections within a layer. The N_i inputs are fed into the first layer of $N_{h,\,1}$ hidden units. The input units are merely 'fan-out' units; no processing takes place in these units. The activation of a hidden unit is a function F_i of the weighted inputs plus a bias, as given in eq. (10.4). The output of the hidden units is distributed over the next layer of $N_{h,\,2}$ hidden units, until the last layer of hidden units, of which the outputs are fed into a layer of No output units (see Fig. 12.1).

Although back-propagation can be applied to networks with any number of layers, just as for networks with binary units (section 11.7) it has been shown (Cybenko, 1989; Funahashi, 1989; Hornik, Stinchcombe, & White, 1989; Hartman, Keeler, & Kowalski, 1990) that only one layer of hidden units suffices to approximate any function with finitely many discontinuities to arbitrary precision, provided the activation functions of the hidden units are non-linear (the universal approximation theorem).

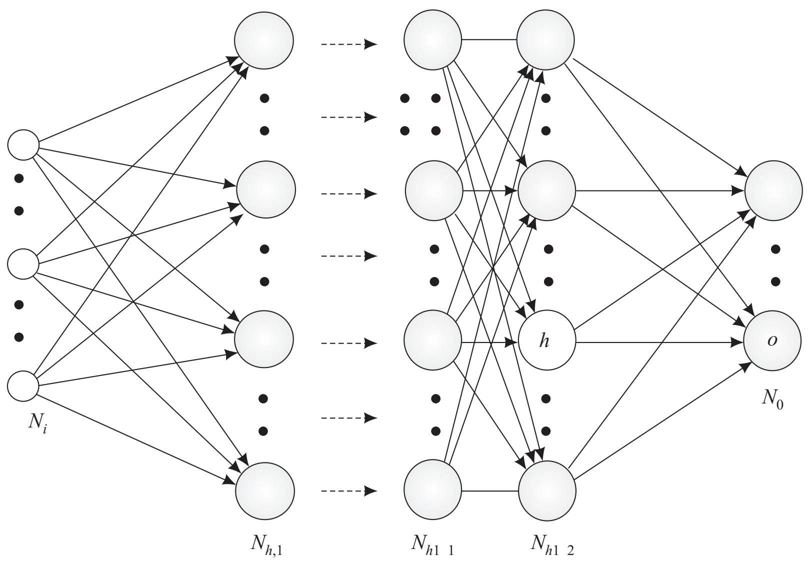

Fig. 12.1 A multi-layer network with layers of units.

In most applications a feed-forward network with a single layer of hidden units is used with a sigmoid activation function for the units.

12.3 THE GENERALISED DELTA RULE

Since we are now using units with nonlinear activation functions, we have to generalise the delta rule, which was presented in chapter 11 for linear functions to the set of non-linear activation functions. The activation is a differentiable function of the total input, given by

$$y_k^p = F(S_k^p) \qquad \text{... (12.1)}$$

in which

$$s_k^p = \sum_j w_{jk} y_k^p + \theta_k \qquad \text{...(12.2)}$$

To get the correct generalization of the delta rule as presented in the previous chapter, we must set

$$\Delta_p w_{jk} = -\gamma \frac{\partial E^p}{\partial w_{jk}} \qquad \text{...(12.3)}$$

The error E^p is defined as the total quadratic error for pattern p at the output units:

$$E^p = \frac{1}{2}\sum_{o=1}^{N_o}\left(d_o^p - y_o^p\right)^2 \qquad \text{...(12.4)}$$

where d_o^p is the desired output for unit 0 when pattern p is clamped. We further set $E = \sum_p E^p$ as the summed squared error. We can write

$$\frac{\partial E^p}{\partial w_{jk}} = \frac{\partial E^p}{\partial S_k^p} \frac{\partial S_k^p}{\partial w_{jk}} \quad \text{...(12.5)}$$

By equation (12.2) we see that the second factor is

$$\frac{\partial S_k^p}{\partial w_{jk}} = y_j^p \quad \text{...(12.6)}$$

When we define

$$\delta_k^p = \frac{\partial E^p}{\partial S_k^p} \quad \text{...(12.7)}$$

we will get an update rule which is equivalent to the delta rule as described in the previous chapter, resulting in a gradient descent on the error surface if we make the weight changes according to:

$$\Delta_p w_{jk} = \gamma \delta_k^p y_j^p \quad \text{...(12.8)}$$

The trick is to figure out what δ_k^p should be for each unit k in the network. The interesting result, which we now derive, is that there is a simple recursive computation of these δ's which can be implemented by propagating error signals backward through the network.

To compute δ_k^p we apply the chain rule to write this partial derivative as the product of two factors, one factor reflecting the change in error as a function of the output of the unit and one reflecting the change in the output as a function of changes in the input. Thus, we have

$$\delta_k^p = \frac{\partial E^p}{\partial S_k^p} = \frac{\partial E^p}{\partial y_k^p} \frac{\partial y_k^p}{\partial S_k^p} \quad \text{...(12.9)}$$

Let us compute the second factor. By equation (12.1) we see that

$$\frac{\partial y_k^p}{\partial S_k^p} = F(S_k^p) \quad \text{...(12.10)}$$

which is simply the derivative of the squashing function F for the kth unit, evaluated at the net input S_k^p to that unit. To compute the first factor of equation (12.9), we consider two cases. First, assume that unit k is an output unit $k = o$ of the network. In this case, it follows from the definition of E^p that

$$\frac{\partial E^p}{\partial y_o^p} = -(d_o^p - y_o^p) \quad \text{... (12.11)}$$

which is the same result as we obtained with the standard delta rule. Substituting this and equation (12.10) in equation (12.9), we get

$$\delta_o^p = (d_o^p - y_o^p)\ F_o^{'}\ (S_o^p) \qquad \text{...(12.12)}$$

for any output unit o. Secondly, if k is not an output unit but a hidden unit $k = h$, we do not readily know the contribution of the unit to the output error of the network. However, the error measure can be written as a function of the net inputs from hidden to output layer $E^p = E^p\ (s_1^p, s_2^p, ..., s_j^p, ...)$ and we use the chain rule to write

$$\frac{\partial E^p}{\partial y_h^p} = \sum_{o=1}^{N_o} \frac{\partial E^p}{\partial S_o^p} \frac{\partial S_o^p}{\partial S_h^p} = \sum_{o=1}^{N_o} \frac{\partial E^p}{\partial S_o^p} \frac{\partial}{\partial y_h^p} \sum_{j=1}^{N_o} w_{ko} y_j^p = \sum_{j=1}^{N_o} \frac{\partial E^p}{\partial S_o^p} w_{ho} = \sum_{j=1}^{N_o} \delta_o^p w_{ho} \qquad \text{... (12.13)}$$

Substituting this in equation (12.9) yields

$$\delta_h^p = F(S_h^p) \sum_{j=1}^{N_o} \delta_o^p w_{ho} \qquad \text{... (12.14)}$$

Equations (12.12) and (12.14) give a recursive procedure for computing the δ'*s* for all units in the network, which are then used to compute the weight changes according to equation (12.8). This procedure constitutes the generalized delta rule for a feed-forward network of non-linear units.

12.3.1 Understanding Back-Propagation

The equations derived in the previous section may be mathematically correct, but what do they actually mean? Is there a way of understanding back-propagation other than reciting the necessary equations?

The answer is, of course, yes. In fact, the whole back-propagation process is intuitively very clear. What happens in the above equations is the following. When a learning pattern is clamped, the activation values are propagated to the output units, and the actual network output is compared with the desired output values, we usually end up with an error in each of the output units. Let's call this error e_o for a particular output unit o. We have to bring e_o to zero.

The simplest method to do this is the greedy method: we strive to change the connections in the neural network in such a way that, next time around, the error e_o will be zero for this particular pattern. We know from the delta rule that, in order to reduce an error, we have to adapt its incoming weights according to

$$\Delta w_{ho} = (d_\circ - y_\circ)\, y_h \qquad \text{...(12.15)}$$

That is step one. But it alone is not enough: when we only apply this rule, the weights from input to hidden units are never changed, and we do not have the full representational power of the feed-forward network as promised by the universal approximation theorem.

In order to adapt the weights from input to hidden units, we again want to apply the delta rule. In this case, however, we do not have a value for δ for the hidden units. This is solved by the chain rule which does the following: distribute the error of an output unit o to all the hidden units that is it connected to, weighted by this connection. Differently put, a hidden unit h receives a delta from each output unit o equal to the delta of that output unit weighted with (= multiplied by) the weight of the connection between those units. In symbols: $\delta_h = \sum_0 \delta_0 w_{ho}$ Well, not exactly: we forgot the activation function of the hidden unit; F' has to be applied to the delta, before the back-propagation process can continue.

12.4 WORKING WITH BACK-PROPAGATION

The application of the generalised delta rule thus involves two phases: During the first phase the input x is presented and propagated forward through the network to compute the output values y_o^p for each output unit. This output is compared with its desired value d_o, resulting in an error signal δ_o^p for each output unit.

The second phase involves a backward pass through the network during which the error signal is passed to each unit in the network and appropriate weight changes are calculated.

12.4.1 Weight Adjustments with Sigmoid Activation Function

The results from the previous section can be summarised in three equations:

- The weight of a connection is adjusted by an amount proportional to the product of an error signal δ, on the unit k receiving the input and the output of the unit j sending this signal along the connection:

$$\Delta_p w_{kj} = \gamma \delta_k^p y_j^p \quad ...(12.16)$$

- If the unit is an output unit, the error signal is given by

$$\delta_o^p = (d_o^p - y_o^p)\ F_o'(S_o^p) \quad ...(12.17)$$

Take as the activation function F the 'sigmoid' function as defined in chapter 2:

$$y^p = F(S^p) = \frac{1}{1+e^{s^p}} \quad ...(12.18)$$

In this case the derivative is equal to

$$F'(S^p) = \frac{\partial}{\partial S^p} \frac{1}{1+e^{s^p}} = \frac{1}{\left(1+e^{s^e}\right)^2}\left(-e^{s^p}\right) = -\frac{1}{\left(1+e^{s^p}\right)^2} \frac{\left(-e^{s^p}\right)}{\left(1+e^{s^p}\right)} = y^p(1-y^p) \quad ...(12.19)$$

such that the error signal for an output unit can be written as:

$$\delta_o^p = (d_o^p - y_o^p)\ y_{o\,o}^p\,(1 - y_o^p) \quad ...(12.20)$$

- The error signal for a hidden unit is determined recursively in terms of error signals of the units to which it directly connects and the weights of those connections. For the sigmoid activation function:

$$\delta_h^p = F'(S_h^p) \sum_{j=1}^{N_o} d_o^p w_{ho} = y_h^p(1 - y_h^p) \sum_{j=1}^{N_o} d_o^p w_{ho} \qquad ...(12.21)$$

12.4.2 Learning Rate And Momentum

The learning procedure requires that the change in weight is proportional to $\frac{\partial E^p}{\partial w}$. True gradient descent requires that infinitesimal steps are taken. The constant of proportionality is the learning rate γ. For practical purposes we choose a learning rate that is as large as possible without leading to oscillation. One way to avoid oscillation at large, is to make the change in weight dependent of the past weight change by adding a momentum term:

$$\Delta w_{jk}(t+1) = \gamma \delta_k^p y_j^p + \alpha \Delta w_{jk}(t) \qquad ...(12.22)$$

where t indexes the presentation number and α is a constant which determines the effect of the previous weight change.

The role of the momentum term is shown in Fig. 12.2. When no momentum term is used, it takes a long time before the minimum has been reached with a low learning rate, whereas for high learning rates the minimum is never reached because of the oscillations. When adding the momentum term, the minimum will be reached faster.

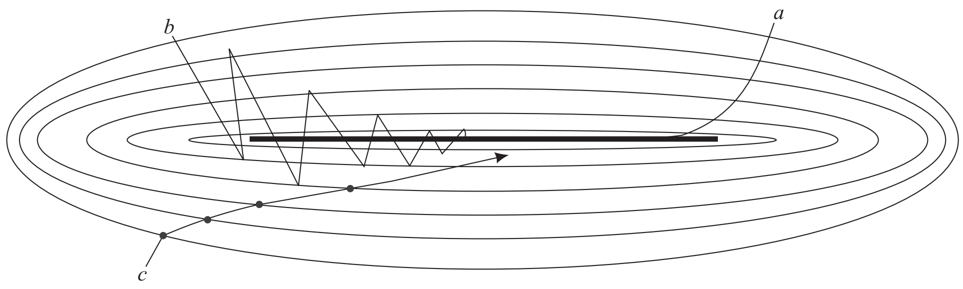

Fig. 12.2 The descent in weight space. (a) for small learning rate; (b) for large learning rate: note the oscillations, and (c) with large learning rate and momentum term added.

12.4.3 Learning Per Pattern

Although, theoretically, the back-propagation algorithm performs gradient descent on the total error only if the weights are adjusted after the full set of learning patterns has been presented, more often than not the learning rule is applied to each pattern separately, i.e., a pattern p is applied, E^p is calculated, and the weights are adapted ($p = 1, 2, \ldots, P$). There exists empirical indication that this results in faster convergence. Care has to be taken, however, with the order in which the patterns are taught. For example, when using the same sequence over and over again the network may become focused on the first few patterns. This problem can be overcome by using a permuted training method.

Example 12.1: A feed-forward network can be used to approximate a function from examples. Suppose we have a system (for example a chemical process or a financial market) of which we want to know the characteristics. The input of the system is given by the two-dimensional vector *x* and the output is given by the one-dimensional vector *d*. We want to estimate the relationship $d = f(x)$ from 80 examples $\{x^p, d^p\}$ as depicted in Fig. 12.3 (top left). A feed-forward network was programmed with two inputs, 10 hidden units with sigmoid activation function and an output unit with a linear activation function. Check for yourself how equation (4.20) should be adapted for the linear instead of sigmoid activation function. The network weights are initialized to small values and the network is trained for 5,000 learning iterations with the back-propagation training rule, described in the previous section. The relationship between *x* and *d* as represented by the network is shown in Fig. 12.3 (top right), while the function which generated the learning samples is given in Fig. 12.3 (bottom left). The approximation error is depicted in Fig. 12.3 (bottom right). We see that the error is higher at the edges of the region within which the learning samples were generated. The network is considerably better at interpolation than extrapolation.

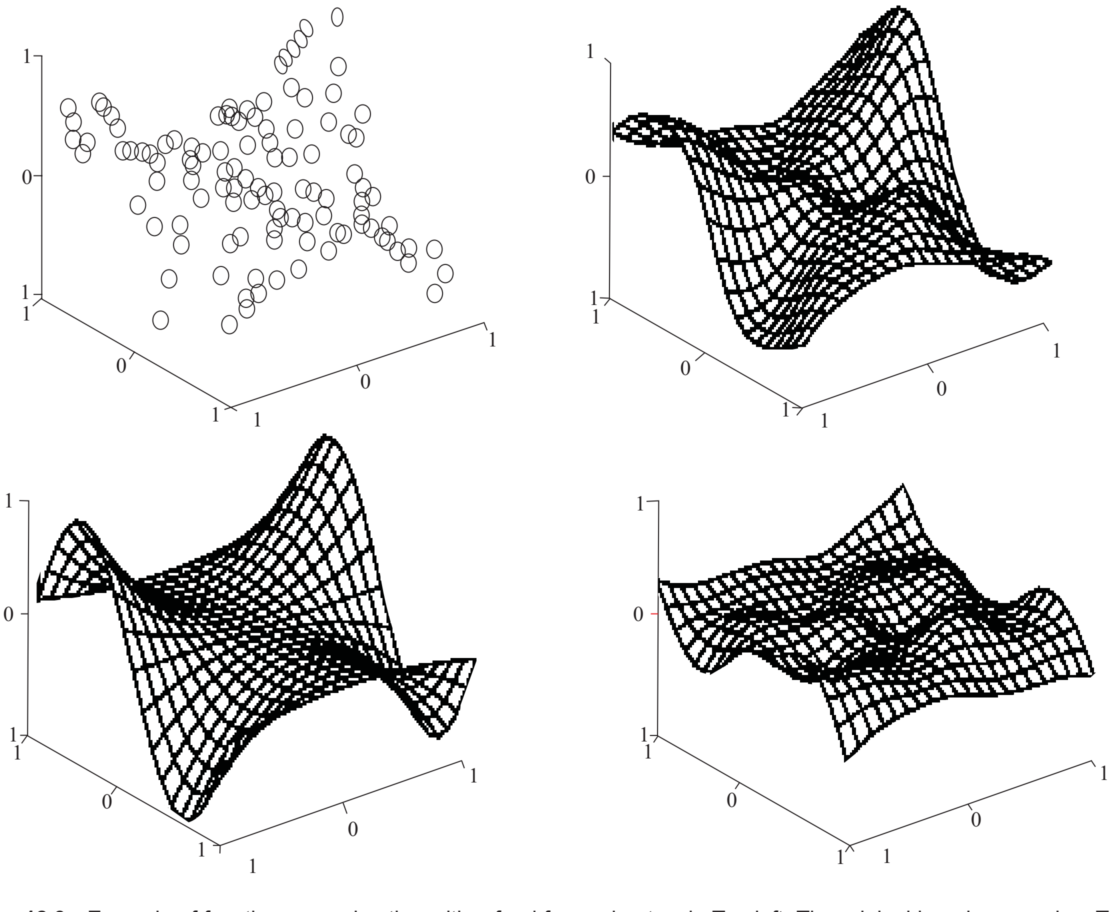

Fig. 12.3 Example of function approximation with a feed forward network. Top left: The original learning samples; Top right: The approximation with the network; Bottom left: The function which generated the learning samples; Bottom right: The error in the approximation.

12.5 OTHER ACTIVATION FUNCTIONS

Although sigmoid functions are quite often used as activation functions, other functions can be used as well. In some cases this leads to a formula, which is known from traditional function approximation theories.

For example, from Fourier analysis it is known that any periodic function can be written as a infinite sum of sine and cosine terms (Fourier series):

$$f(x) = \sum_{n=0}^{\infty}(a_n \cos nx + b_n \sin nx) \qquad \text{...(12.23)}$$

We can rewrite this as a summation of sine terms

$$f(x) = a_0 + \sum_{n=1}^{\infty} c_n \sin (nx + \theta_n) \qquad \text{...(12.24)}$$

with $c_n = \sqrt{a_n^2 + b_n^2}$ and $\theta_n = \arctan (b/a)$. This can be seen as a feed-forward network with a single input unit for *x;* a single output unit for $f(x)$ and hidden units with an activation function $F = \sin (s)$. The factor a_0 corresponds with the bias of the output unit, the factors c_n correspond with the weighs from hidden to output unit; the phase factor θ_n corresponds with the bias term of the hidden units and the factor n corresponds with the weights between the input and hidden layer.

The basic difference between the Fourier approach and the back-propagation approach is that the in the Fourier approach the 'weights' between the input and the hidden units (these are the factors n) are fixed integer numbers which are analytically determined, whereas in the back-propagation approach these weights can take any value and are typically learning using a learning heuristic.

To illustrate the use of other activation functions we have trained a feed-forward network with one output unit, four hidden units, and one input with ten patterns drawn from the function $f(x) = \sin(2x) \sin(x)$. The result is depicted in Fig. 12.4. The same function (albeit with other learning points) is learned with a network with eight sigmoid hidden units (see Figure 12.5). From the figures it is clear that it pays off to use as much knowledge of the problem at hand as possible.

12.6 DEFICIENCIES OF BACK-PROPAGATION

Despite the apparent success of the back-propagation learning algorithm, there are some aspects, which make the algorithm not guaranteed to be universally useful. Most troublesome is the long training process. This can be a result of a non-optimum learning rate and momentum. A lot of advanced algorithms based on back-propagation learning have some optimized method to adapt this learning rate, as will be discussed in the next section. Outright training failures generally arise from two sources: network paralysis and local minima.

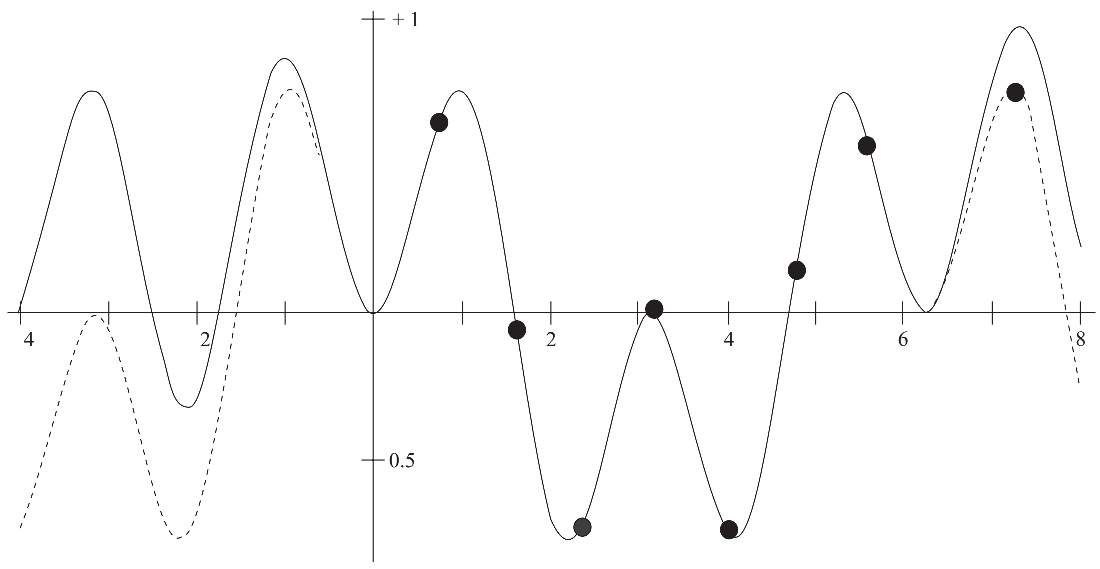

Fig. 12.4 The periodic function *f*(*x*) = sin (2*x*) sin (*x*) approximated with sine activation functions.

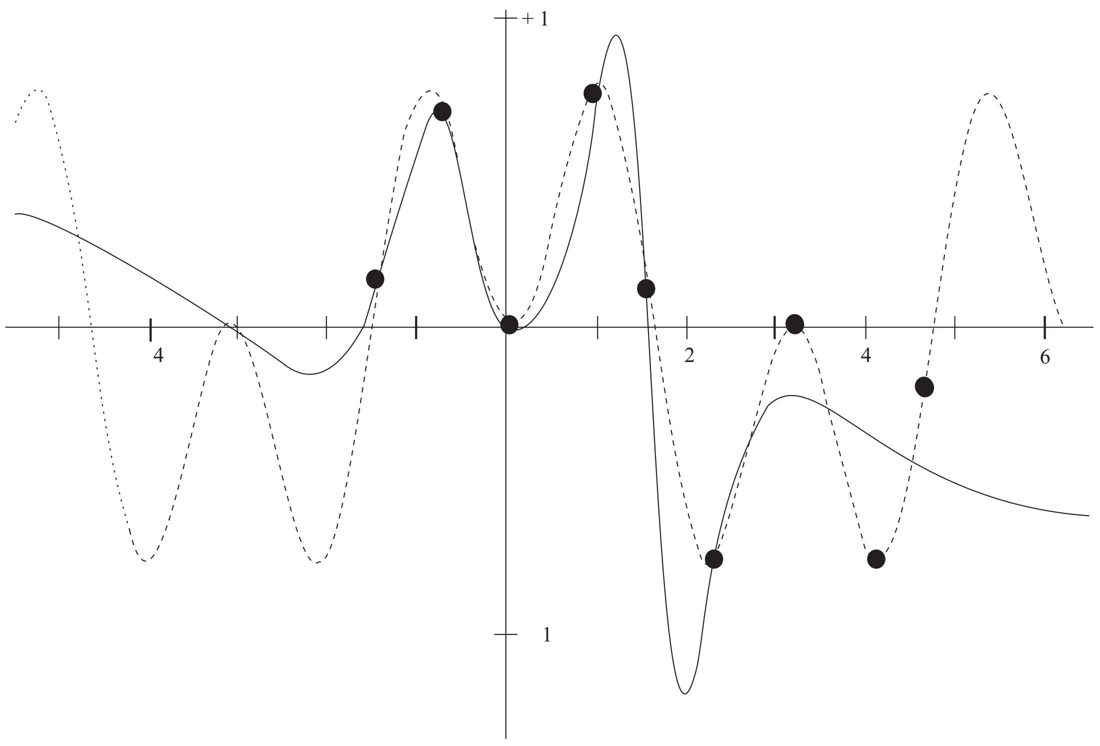

Fig. 12.5 The periodic function *f*(*x*) = sin (2*x*) sin (*x*) approximated with sigmoid activation functions.

12.6.1 Network Paralysis

As the network trains, the weights can be adjusted to very large values. The total input of a hidden unit or output unit can therefore reach very high (either positive or negative) values, and because of the sigmoid activation function the unit will have an activation very close to zero or very close to one. As is clear from equations (12.20) and (12.21), the weight adjustments which are proportional to $y_k^p(1 - y_k^p)$ will be close to zero, and the training process can come to a virtual standstill.

12.6.2 Local Minima

The error surface of a complex network is full of hills and valleys. Because of the gradient descent, the network can get trapped in a local minimum when there is a much deeper minimum nearby. Probabilistic methods can help to avoid this trap, but they tend to be slow. Another suggested possibility is to increase the number of hidden units. Although this will work because of the higher dimensionality of the error space, and the chance to get trapped is smaller, it appears that there is some upper limit of the number of hidden units which, when exceeded, again results in the system being trapped in local minima.

12.7 ADVANCED ALGORITHMS

Many researchers have devised improvements of and extensions to the basic back-propagation algorithm described above. It is too early for a full evaluation: some of these techniques may prove to be fundamental, others may simply fade away. A few methods are discussed in this section.

May be the most obvious improvement is to replace the rather primitive steepest descent method with a direction set minimization method, e.g., conjugate gradient minimization. Note that minimization along a direction *u* brings the function *f* at a place where its gradient is perpendicular to *u* (otherwise minimization along *u* is not complete).

Instead of following the gradient at every step, a set of *n* directions is constructed which are all conjugate to each other such that minimization along one of these directions u_j does not spoil the minimization along one of the earlier directions u_i, i.e., the directions are non-interfering. Thus one minimization in the direction of u_i suffices, such that *n* minimizations in a system with n degrees of freedom bring this system to a minimum (provided the system is quadratic). This is different from gradient descent, which directly minimizes in the direction of the steepest descent (Press, Flannery, Teukolsky, & Vetterling, 1986).

Suppose the function to be minimized is approximated by its Taylor series

$$f(x) = f(p) + \sum_i \left.\frac{\partial f}{\partial x_i}\right|_p x_i + \frac{1}{2}\sum_{i,j} \left.\frac{\partial^2 f}{\partial x_i \partial x_j}\right|_p x_i x_j + \ldots \approx \frac{1}{2} x^T A x - b^T x + c \qquad \ldots(12.25)$$

where *T* denotes transpose, and

$$c \equiv f(p)$$

$$b \equiv -\nabla f\big|_p$$

$$[A]_{ij} = \left.\frac{\partial^2 f}{\partial x_i \partial x_j}\right|_p \qquad \text{...(12.26)}$$

A is a symmetric positive definite $n \times n$ matrix, the Hessian of f at p. The gradient of f is

$$\nabla f = A\mathrm{x} - b \qquad \text{...(12.27)}$$

such that a change of x results in a change of the gradient as

$$\delta(\nabla f) = A(\delta x) \qquad \text{...(12.28)}$$

Now suppose f was minimized along a direction ui to a point where the gradient $-g_{i+1}$ of f is perpendicular to u_i, i.e.,

$$u_i^T \mathrm{g}_{i+1} = 0 \qquad \text{...(12.29)}$$

and a new direction u_{i+1} is sought. In order to make sure that moving along u_{i+1} does not spoil minimization along u_i we require that the gradient of f remain perpendicular to u_i, i.e.,

$$u_i^T g_{i+2} = 0 \qquad \text{...(12.30)}$$

otherwise we would once more have to minimise in a direction which has a component of u_i.

Combining (12.29) and (12.30), we get

$$0 = u_i^T(g_{i+1} - g_{i+2}) = u_i^T \delta(\nabla f) = u_i^T A u_{i+1} \qquad \text{...(12.31)}$$

When eq. (12.31) holds for two vectors u_i and u_{i+1} they are said to be conjugate.

Now, starting at some point p_0, the first minimization direction u_0 is taken equal to $g_0 = -\nabla f\,(p_0)$, resulting in a new point p_1. For $i \geq 0$, calculate the directions

$$u_{i+1} \quad \mathrm{g}_{i+1} + \gamma_i u_i \qquad \text{...(12.32)}$$

where γ_i is chosen to make $u_i^T A u_i \;\; 1$ and the successive gradients perpendicular, i.e.,

$$\gamma_i = \frac{g_{i+1}^T \, g_{i+1}}{g_i^T \, g_i} \text{ with } g_k = -\nabla f|_{pk} \text{ for all } k \geq 0 \qquad \text{...(12.33)}$$

Next, calculate $p_{i+2} = p_{i+1} + \lambda_{i+1}\, u_{i+1}$ where λ_{i+1} is chosen so as to minimize $f(P_{i+2})^3$.

It can be shown that the u's thus constructed are all mutually conjugate (e.g., see (Stoer & Bulirsch, 1980)). The process described above is known as the Fletcher-Reeves method, but there are many variants, which work more or less the same (Hestenes & Stiefel, 1952; Polak, 1971; Powell, 1977).

Although only n iterations are needed for a quadratic system with n degrees of freedom, due to the fact that we are not minimizing quadratic systems, as well as a result of round-off errors, the n directions have to be followed several times (see Fig. 12.6). Powell introduced some improvements to correct for behaviour in non-quadratic systems. The resulting cost is $O(n)$ which is significantly better than the linear convergence 4 of steepest descent.

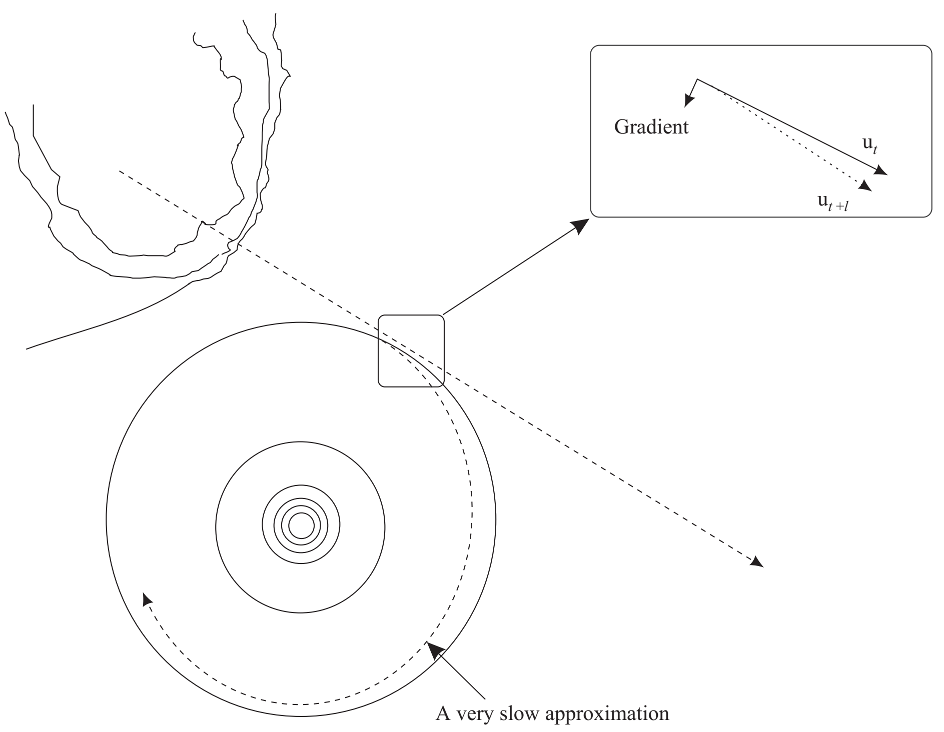

Fig. 12.6 Slow decrease with conjugate gradient in non-quadratic systems. [The hills on the left are very steep, resulting in a large search vector u_i. When the quadratic portion is entered the new search direction is constructed from the previous direction and the gradient, resulting in a spiraling minimization. This problem can be overcome by detecting such spiraling minimizations and restarting the algorithm with $u_0 = \nabla f$].

Some improvements on back-propagation have been presented based on an independent adaptive arning rate parameter for each weight.

Van den Boomgaard and Smeulders (Boomgaard & Smeulders, 1989) show that for a feed-forward network without hidden units an incremental procedure to find the optimal weight matrix W needs an adjustment of the weights with

$$\Delta w(t+1) = \gamma(t+1)\,[d(t+1) - w(t) \times (t+1)] \times (t+1) \qquad \text{...(12.34)}$$

in which γ is not a constant but an variable $(N_i + 1) \times (N_i + 1)$ matrix which depends on the input vector. By using a priori knowledge about the input signal, the storage requirements for can be reduced.

Silva and Almeida (Silva & Almeida, 1990) also show the advantages of an independent step size for each weight in the network. In their algorithm the learning rate is adapted after every learning pattern:

$$\gamma_{jk}(t+1) = \begin{cases} u\gamma_{jk}(t) & \text{if } \dfrac{\partial E(t+1)}{\partial w_{jk}} \text{ and } \dfrac{\partial E(t)}{\partial w_{jk}} \text{ have the same signs} \\ d\gamma_{jk}(t) & \text{if } \dfrac{\partial E(t+1)}{\partial w_{jk}} \text{ and } \dfrac{\partial E(t)}{\partial w_{jk}} \text{ have the opposite signs} \end{cases} \qquad \text{...(12.35)}$$

where u and d are positive constants with values slightly above and below unity, respectively. The idea is to decrease the learning rate in case of oscillations.

12.8 HOW GOOD ARE MULTI-LAYER FEED-FORWARD NETWORKS?

From the example shown in Fig. 12.3 is clear that the approximation of the network is not perfect. The resulting approximation error is influenced by:

1. The learning algorithm and number of iterations. This determines how good the error on the training set is minimized.
2. The number of learning samples. This determines how good the training samples represent the actual function.
3. The number of hidden units. This determines the 'expressive power' of the network. For 'smooth' functions only a few number of hidden units are needed, for wildly fluctuating functions more hidden units will be needed.

In the previous sections we discussed the learning rules such as back-propagation and the other gradient based learning algorithms, and the problem of finding the minimum error. In this section we particularly address the effect of the number of learning samples and the effect of the number of hidden units.

We first have to define an adequate error measure. All neural network training algorithms try to minimize the error of the set of learning samples which are available for training the network. The average error per learning sample is defined as the learning error rate error rate:

$$E_{\text{learning}} = \frac{1}{P_{\text{learning}}} \sum_{p=1}^{P_{\text{learning}}} E^p \qquad \text{...(12.36)}$$

in which E^p is the difference between the desired output value and the actual network output for the learning samples:

$$E^p = \frac{1}{2} \sum_{0=1}^{N_o} (d_o^p - y_o^p) \qquad \text{...(12.37)}$$

This is the error, which is measurable during the training process.

It is obvious that the actual error of the network will differ from the error at the locations of the training samples. The difference between the desired output value and the actual network output should be integrated over the entire input domain to give a more realistic error measure. This integral can be estimated if we have a large set of samples.

We now define the test error rate as the average error of the test set:

$$E_{\text{test}} = \frac{1}{P_{\text{test}}} \sum_{p=1}^{P_{\text{test}}} E^p \qquad \text{...(12.38)}$$

In the following subsections we will see how these error measures depend on learning set size and number of hidden units.

12.8.1 The Effect Of the Number of Learning Samples

A simple problem is used as example: a function $y = f(x)$ has to be approximated with a feed-forward neural network. A neural network is created with an input, 5 hidden units with sigmoid activation function and a linear output unit. Suppose we have only a small number of learning samples (e.g., 4) and the networks is trained with these samples. Training is stopped when the error does not decrease anymore.

The original (desired) function is shown in Fig. 4.7A as a dashed line. The learning samples and the approximation of the network are shown in the same figure. We see that in this case E_{learning} is small (the network output goes perfectly through the learning samples) but E_{test} is large: the test error of the network is large. The approximation obtained from 20 learning samples is shown in Fig. 12.7B. The E_{learning} is larger than in the case of 5 learning samples, but the E_{test} is smaller.

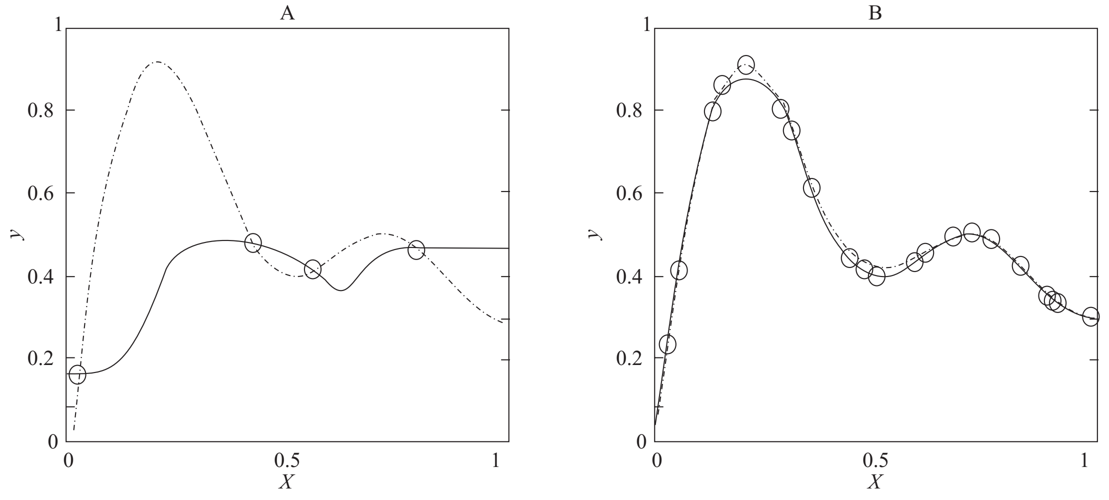

Fig. 12.7 Effect of the learning set size on the generalization. The dashed line gives the desired function, the learning samples are depicted as circles and the approximation by the network is shown by the drawn line. 5 hidden units are used. a) 4 learning samples. b) 20 learning samples.

This experiment was carried out with other learning set sizes, where for each learning set size the experiment was repeated 10 times. The average learning and test error rates as a function of the learning set size are given in Fig. 12.8. Note that the learning error increases with an increasing learning set size, and the test error decreases with increasing learning set size.

A low learning error on the (small) learning set is no guarantee for a good network performance! With increasing number of learning samples the two error rates converge to the same value. This value depends on the representational power of the network: given the optimal weights, how good is the approximation. This error depends on the number of hidden units and the activation function. If the learning error rate does not converge to the test error rate the learning procedure has not found a global minimum.

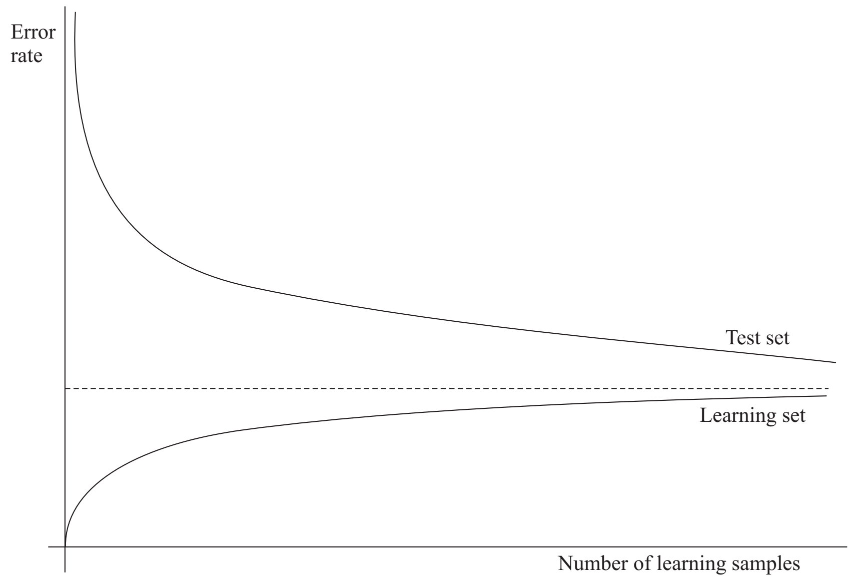

Fig. 12.8 Effect of the learning set size on the error rate. The average error rate and the average test error rate are as a function of the number of learning samples.

12.8.2 The Effect of the Number of Hidden Units

The same function as in the previous subsection is used, but now the number of hidden units is varied. The original (desired) function, learning samples and network approximation is shown in Fig. 4.9A for 5 hidden units and in Fig. 4.9B for 20 hidden units. The effect visible in Fig. 4.9B is called over training. The network fits exactly with the learning samples, but because of the large number of hidden units the function which is actually represented by the network is far more wild than the original one. Particularly in case of learning samples which contain a certain amount of noise (which all real-world data have), the network will 'fit the noise' of the learning samples instead of making a smooth approximation.

This example shows that a large number of hidden units leads to a small error on the training set but not necessarily leads to a small error on the test set. Adding hidden units will always lead to a reduction of the E_{learning}. However, adding hidden units will first lead to a reduction of the E_{test}, but then lead to an increase of E_{test}. This effect is called the peaking effect. The average learning and test error rates as a function of the learning set size are given in Fig. 12.10.

12.9 APPLICATIONS

Back-propagation has been applied to a wide variety of research applications.

- Sejnowski and Rosenberg (1986) produced a spectacular success with NETtalk, a system that converts printed English text into highly intelligible speech.
- A feed-forward network with one layer of hidden units has been described by Gorman and Sejnowski (1988) as a classification machine for sonar signals.

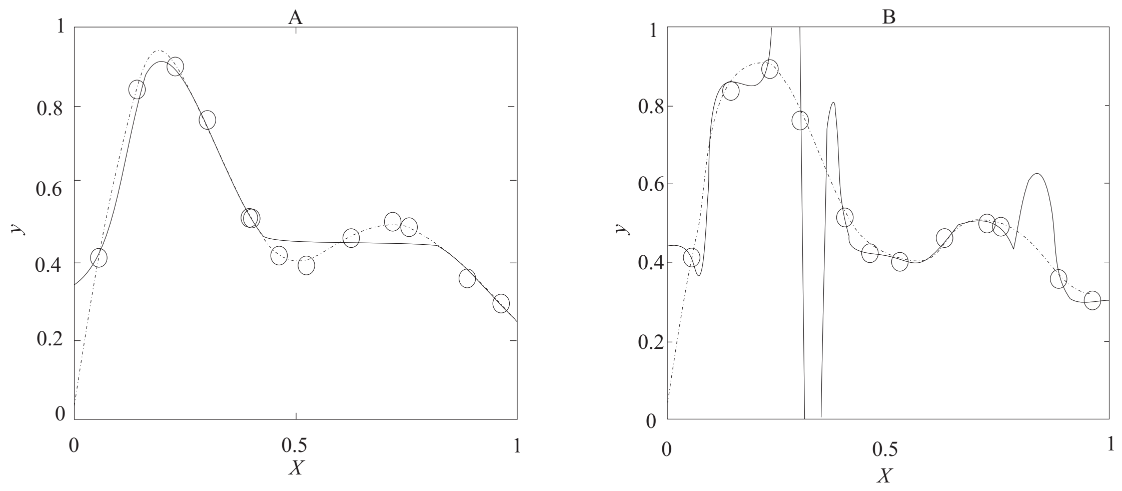

Fig. 12.9 Effect of the number of hidden units on the network performance. The dashed line gives the desired function, the circles denote the learning samples and the drawn line gives the approximation by the network. 12 learning samples are used. a) 5 hidden units. b) 20 hidden units.

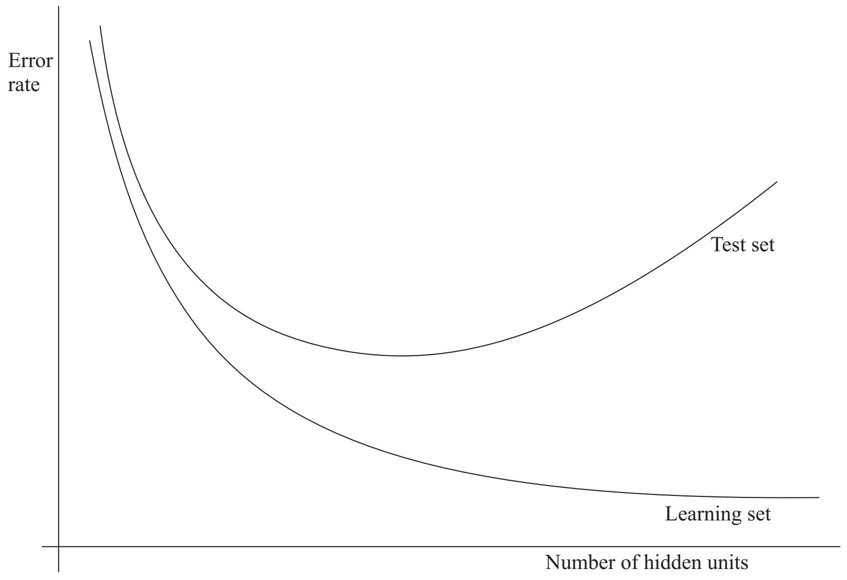

Fig. 12.10 The average learning error rate and the average test error rate as a function of the number of hidden units.

· A multi-layer feed-forward network with a back-propagation training algorithm is used to learn an unknown function between input and output signals from the presentation of examples. It is hoped that the network is able to generalize correctly, so that input values which are not presented as learning patterns will result in correct output values. An example is the work of Josin (1988), who used a two-layer feed-forward network with back-propagation learning to perform the inverse kinematic transform which is needed by a robot arm controller.