8

Fast Learning Algorithms

8.1 Introduction – classical backpropagation

Artificial neural networks attracted renewed interest over the last decade, mainly because new learning methods capable of dealing with large scale learning problems were developed. After the pioneering work of Rosenblatt and others, no efficient learning algorithm for multilayer or arbitrary feed-forward neural networks was known. This led some to the premature conclusion that the whole field had reached a dead-end. The rediscovery of the backpropagation algorithm in the 1980s, together with the development of alternative network topologies, led to the intense outburst of activity which put neural computing back into the mainstream of computer science.

Much has changed since the original proposals of the PDP group. There is now no lack of alternative fast learning algorithms for neural networks. Each new conference and each new journal issue features some kind of novel learning method offering better and faster convergence than the tried and trusted standard backpropagation method. The reason for this combinatorial explosion of new algorithms is twofold: on the one hand, backpropagation itself is a rather *slow* learning algorithm, which through malicious selection of parameters can be made even slower. By using any of the well-known techniques of nonlinear optimization, it is possible to accelerate the training method with practically no effort. Since authors usually compare their new algorithms with standard backpropagation, they always report a substantial improvement [351]. On the other hand, since the learning problem for artificial neural networks is *NP*-complete (see Chap. 10) in the worst case, the computational effort involved in computing the network parameters grows exponentially with the number of unknowns. This leaves room for alternative proposals which could deal with some learning tasks in a more efficient manner. However, it is always possible to fool the best learning method with a suitable learning task and it is always possible to make it perform incomparably better than all its competitors. It is a rather surprising fact that standard on-line backpropagation performs better than many fast learning algorithms as soon as the learning task achieves a

realistic level of complexity and when the size of the training set goes beyond a critical threshold [391].

In this chapter we try to introduce some order into the burgeoning field of fast learning algorithms for neural networks. We show the essential characteristics of the proposed methods, the fundamental alternatives open to anyone wishing to improve traditional backpropagation and the similarities and differences between the different techniques. Our analysis will be restricted to those algorithms which deal with a fixed network topology. One of the lessons learned over the past years is that significant improvements in the approximation capabilities of neural networks will only be obtained through the use of *modularized networks*. In the future, more complex learning algorithms will deal not only with the problem of determining the network parameters, but also with the problem of adapting the network topology. Algorithms of this type already in existence fall beyond the scope of this chapter.

8.1.1 Backpropagation with momentum

Before reviewing some of the variations and improvements which have been proposed to accelerate the learning process in neural networks, we briefly discuss the problems involved in trying to minimize the error function using gradient descent. Therefore, we first describe a simple modification of backpropagation called *backpropagation with momentum*, and then look at the form of the iteration path in weight space.

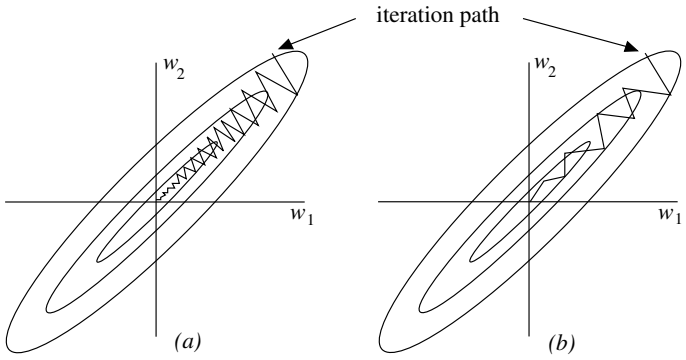

Fig. 8.1. Backpropagation without (a) or with (b) momentum term

When the minimum of the error function for a given learning task lies in a narrow "valley", following the gradient direction can lead to wide oscillations of the search process. Figure 8.1 shows an example for a network with just two weights w_1 and w_2. The best strategy in this case is to orient the search towards the center of the valley, but the form of the error function is such that the gradient does not point in this direction. A simple solution is to

introduce a *momentum* term. The gradient of the error function is computed for each new combination of weights, but instead of just following the negative gradient direction a *weighted average* of the current gradient and the previous correction direction is computed at each step. Theoretically, this approach should provide the search process with a kind of *inertia* and could help to avoid excessive oscillations in narrow valleys of the error function.

As explained in the previous chapter, in standard backpropagation the input-output patterns are fed into the network and the error function E is determined at the output. When using backpropagation with momentum in a network with n different weights $w_1, w_2, \ldots, w_n$, the i-th correction for weight w_k is given by

$$\Delta w_k(i) = -\gamma \frac{\partial E}{\partial w_k} + \alpha \Delta w_k(i-1),$$

where γ and α are the learning and momentum rate respectively. Normally, we are interested in accelerating convergence to a minimum of the error function, and this can be done by increasing the learning rate up to an optimal value. Several fast learning algorithms for neural networks work by trying to find the best value of γ which still guarantees convergence. Introduction of the momentum rate allows the attenuation of oscillations in the iteration process.

The adjustment of both learning parameters to obtain the best possible convergence is normally done by trial and error or even by some kind of random search [389]. Since the optimal parameters are highly dependent on the learning task, no general strategy has been developed to deal with this problem. Therefore, in the following we show the trade-offs involved in choosing a specific learning and momentum rate, and the kind of oscillating behavior which can be observed with the backpropagation feedback rule and large momentum rates. We show that they are necessary when the optimal size of the learning step is unknown and the form of the error function is highly degenerate.

The linear associator

Let us first consider the case of a *linear associator*, that is, a single computing element with associated weights $w_1, w_2, \ldots, w_n$ and which for the input $x_1, x_2, \ldots, x_n$ produces $w_1 x_1 + \cdots + w_n x_n$ as output, as shown in Figure 8.2.

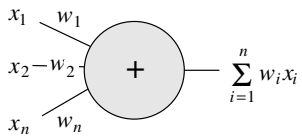

Fig. 8.2. Linear associator

The input-output patterns in the training set are the p ordered pairs $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_p, y_p)$, whereby the input patterns are row vectors of dimension n and the output patterns are scalars. The weights of the linear associator can be ordered in an n-dimensional column vector $\mathbf{w}$ and the learning task consists of finding the $\mathbf{w}$ that minimizes the quadratic error

$$E = \sum_{i=1}^{n} \|\mathbf{x}_i \cdot \mathbf{w} - y_i\|^2.$$

By defining a $p \times m$ matrix $\mathbf{X}$ whose rows are the vectors $\mathbf{x}_1, \ldots, \mathbf{x}_p$ and a column vector $\mathbf{y}$ whose elements are the scalars $y_1, \ldots, y_p$, the learning task reduces to the minimization of

$$\begin{aligned} E &= \|\mathbf{X}\mathbf{w} - \mathbf{y}\|^2 \\ &= (\mathbf{X}\mathbf{w} - \mathbf{y})^{\mathrm{T}}(\mathbf{X}\mathbf{w} - \mathbf{y}) \\ &= \mathbf{w}^{\mathrm{T}}(\mathbf{X}^{\mathrm{T}}\mathbf{X})\mathbf{w} - 2\mathbf{y}^{\mathrm{T}}\mathbf{X}\mathbf{w} + \mathbf{y}^{\mathrm{T}}\mathbf{y}. \end{aligned}$$

Since this is a quadratic function, the minimum can be found using gradient descent.

The quadratic function E can be thought of as a paraboloid in m-dimensional space. The lengths of its principal axes are determined by the magnitude of the eigenvalues of the correlation matrix $\mathbf{X}^{\mathrm{T}}\mathbf{X}$. Gradient descent is most effective when the principal axes of the quadratic form are all of the same length. In this case the gradient vector points directly towards the minimum of the error function. When the axes of the paraboloid are of very different sizes, the gradient direction can lead to oscillations in the iteration process as shown in Figure 8.1.

Let us consider the simple case of the quadratic function $ax^2 + by^2$. Gradient descent with momentum yields the iteration rule

$$\Delta x(i) = -2\gamma a x + \alpha \Delta x(i-1)$$

in the x direction and

$$\Delta y(i) = -2\gamma b x + \alpha \Delta y(i-1)$$

in the y direction. An optimal parameter combination in the x direction is $\gamma = 1/2a$ and $\alpha = 0$. In the y direction the optimal combination is $\gamma = 1/2b$ and $\alpha = 0$. Since iteration proceeds with a single γ value, we have to find a compromise between these two options. Intuitively, if the momentum term is zero, an intermediate γ should do best. Figure 8.3 shows the number of iterations needed to find the minimum of the error function to a given precision as a function of γ, when $a = 0.9$ and $b = 0.5$. The optimal value for γ is the one found at the intersection of the two curves and is $\gamma = 0.7$. The global optimal γ is larger than the optimal γ in the x direction and smaller than the optimal γ in the y direction. This means that there will be some oscillations

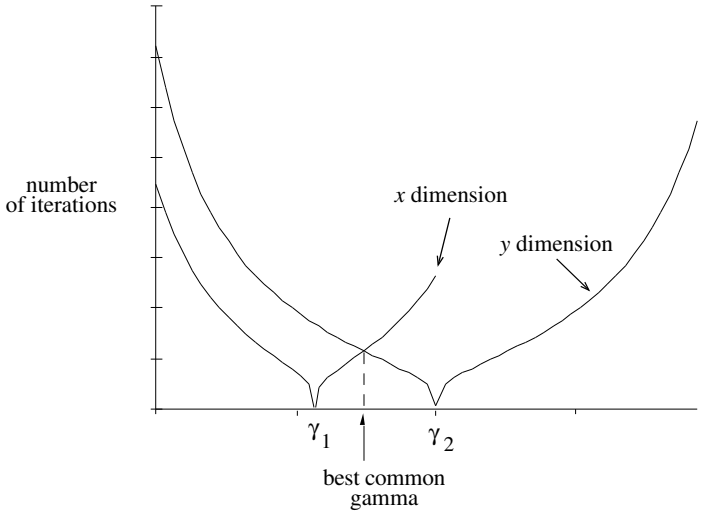

Fig. 8.3. Optimal γ in the two-dimensional case

in the y direction and slow convergence in the x direction, but this is the best possible compromise. It is obvious that in the n-dimensional case we could have oscillations in some of the principal directions and slow convergence in others. A simple strategy used by some fast learning algorithms to avoid these problems consists of using a different learning rate for each weight, that is, a different γ for each direction in weight space [217].

Minimizing oscillations

Since the lengths of the principal axes of the error function are given by the eigenvalues of the correlation matrix $\mathbf{X}^{\mathrm{T}}\mathbf{X}$, and since one of these eigenvalues could be much larger than the others, the range of possible values for γ reduces accordingly. Nevertheless, a very small γ and the oscillations it produces can be neutralized by increasing the momentum rate. We proceed to a detailed discussion of the one-dimensional case, which provides us with the necessary insight for the more complex multidimensional case.

In the one-dimensional case, that is, when minimizing functions of type kx^2, the optimal learning rate is given by $\gamma = 1/2k$. The rate $\gamma = 1/k$ produces an oscillation between the initial point x_0 and $-x_0$. Any γ greater than $2/k$ leads to an "explosion" of the iteration process. Figure 8.4 shows the main regions for parameter combinations of γ and the momentum rate α. These regions were determined by iterating in the one-dimensional case and *integrating* the length of the iteration path. Parameter combinations in the divergence region lead to divergence of the iteration process. Parameter combinations in the boundary between regions lead to stable oscillations.

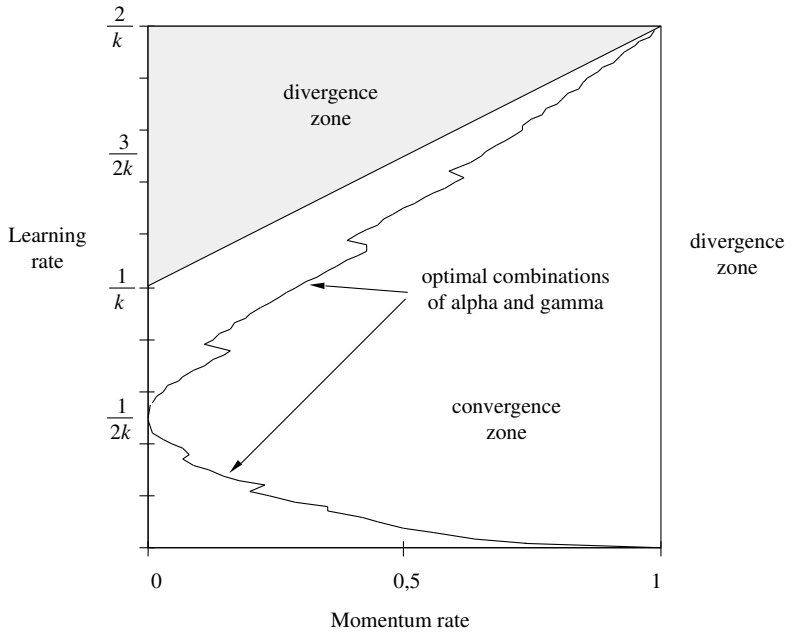

Fig. 8.4. Convergence zone for combinations of γ and α

Figure 8.4 shows some interesting facts. Any value of γ greater than four times the constant $1/2k$ cannot be balanced with any value of α. Values of α greater than 1 are prohibited since they lead to a geometric explosion of the iteration process. Any value of γ between the explosion threshold $1/k$ and $2/k$ can lead to convergence if a large enough α is selected. For any given γ between $1/k$ and $2/k$ there are two points at which the iteration process falls in a stable oscillation, namely at the boundaries between regions. For values of γ under the optimal value $1/2k$, the convergence speed is optimal for a unique α. The optimal combinations of α and γ are the ones represented by the jagged line in the diagram.

The more interesting message we get from Figure 8.4 is the following: in the case where in some direction in weight space the principal axis of the error function is very small compared to another axis, we should try to achieve a compromise by adjusting the momentum rate in such a way that the directions in which the algorithm oscillates become less oscillating and the directions with slow convergence improve their convergence speed. Obviously when dealing with n axes in weight space, this compromise could be dominated by a single direction.

Critical parameter combinations

Backpropagation is used in those cases in which we do not have an analytic expression of the error function to be optimized. A learning rate γ has to be chosen without any previous knowledge of the correlation matrix of the input. In on-line learning the training patterns are not always defined in advance, and are generated one by one. A conservative approach is to minimize the risk by choosing a very small learning rate. In this case backpropagation can be trapped in a local minimum of a nonlinear error function. The learning rate should then be increased.

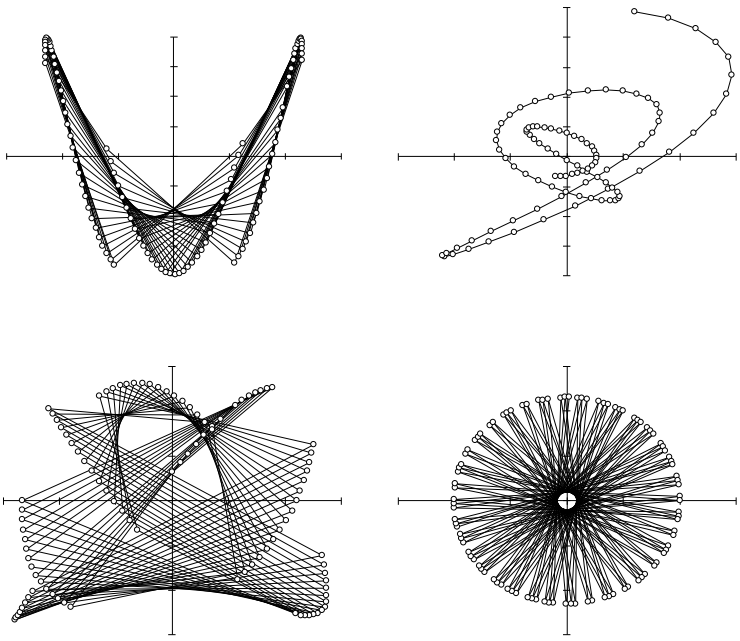

Fig. 8.5. Paths in weight space for backpropagation learning (linear associators). The minimum of the error function is located at the origin.

In the case of a correlation matrix $\mathbf{X}^{\mathrm{T}}\mathbf{X}$ with some very large eigenvalues, a given choice of γ could lead to divergence in the associated direction in weight space (assuming for simplicity that the principal directions of the quadratic form are aligned with the coordinate axis). Let us assume that the selected γ is near to the explosion point $2/k$ found in the one-dimensional case and shown in Figure 8.4. In this case only values of the momentum rate near to one can guarantee convergence, but oscillations in some of the directions in weight space can become synchronized. The result is oscillating paths in weight space, reminiscent of Lissajou's figures. Figure 8.5 shows some paths in a two-dimensional weight space for several linear associators trained with

momentum rates close to one and different γ values. In some cases the trajectories lead to convergence after several thousand iterations. In others a momentum rate equal to one precludes convergence of the iteration process.

Adjustment of the learning and momentum rate in the nonlinear case is even more difficult than in the linear case, because there is no fast explosion of the iteration process. In the quadratic case, whenever the learning rate is excessively large, the iteration process leads rapidly to an overflow which alerts the programmer to the fact that the step size should be reduced. In the nonlinear case the output of the network and the error function are bounded and no overflow occurs. In regions far from local minima the gradient of the error function becomes almost zero as do the weight adjustments. The divergence regions of the quadratic case can now become oscillatory regions. In this case, even as step sizes increase, the iteration returns to the convex part of the error function.

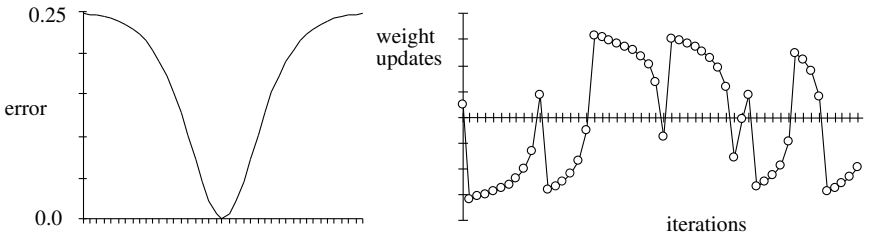

Fig. 8.6. Bounded nonlinear error function and the result of several iterations

Figure 8.6 shows the possible shape of the error function for a linear associator with sigmoidal output and the associated oscillation process for this kind of error function in the one-dimensional case. The jagged form of the iteration curve is reminiscent of the kind of learning curves shown in many papers about learning in nonlinear neural networks. Although in the quadratic case only large momentum rates lead to oscillations, in the nonlinear case an excessively large γ can also produce oscillations even when no momentum rate is present.

Researchers in the field of neural networks should be concerned not only with the possibility of getting stuck in local minima of the error function when learning rates are too small, but also with the possibility of falling into the oscillatory traps of backpropagation when the learning rate is too big. Learning algorithms should try to balance the speedup they are attempting to obtain with the risk of divergence involved in doing so. Two different kinds of remedy are available: a) adaptive learning rates and b) statistical preprocessing of the learning set which is done to decorrelate the input patterns, thus avoiding the negative effect of excessively large eigenvalues of the correlation matrix.

8.1.2 The fractal geometry of backpropagation

It is empirically known that standard backpropagation is very sensitive to the initial learning rate chosen for a given learning task. In this section we examine the shape of the iteration path for the training of a linear associator using on-line backpropagation. We show that the path is a fractal in weight space. The specific form depends on the learning rate chosen, but there is a threshold value for which the attractor of the iteration path is dense in a region of weight space around a local minimum of the error function. This result also yields a deeper insight into the mechanics of the iteration process in the nonlinear case.

The Gauss–Jacobi and Gauss–Seidel methods and backpropagation

Backpropagation can be performed in batch or on-line mode, that is, by updating the network weights once after each presentation of the complete training set or immediately after each pattern presentation. In general, on-line backpropagation does not converge to a single point in weight space, but oscillates around the minimum of the error function. The expected value of the deviation from the minimum depends on the size of the learning step being used. In this section we show that although the iteration process can fail to converge for some choices of learning rate, the iteration path for on-line learning is not just random noise, but possesses some structure and is indeed a fractal. This is rather surprising, because if the training patterns are selected randomly, one would expect a random iteration path. As we will see in what follows, it is easy to show that on-line backpropagation, in the case of linear associators, defines an *Iterated Function System* of the same type as the ones popularized by Barnsley [42]. This is sufficient proof that the iteration path has a fractal structure. To illustrate this fact we provide some computer-generated graphics.

First of all we need a visualization of the way off-line and on-line backpropagation approach the minimum of the error function. To simplify the discussion consider a linear associator with just two input lines (and thus two weights w_1 and w_2). Assume that three input-output patterns are given so that the equations to be satisfied are:

$$x_1^1 w_1 + x_2^1 w_2 = y_1 \tag{8.1}$$

$$x_1^2 w_1 + x_2^2 w_2 = y_2 \tag{8.2}$$

$$x_1^3 w_1 + x_2^3 w_2 = y_3 \tag{8.3}$$

These three equations define three lines in weight space and we look for the combination of w_1 and w_2 which satisfies all three simultaneously. If the three lines intersect at the same point, we can compute the solution using Gauss elimination. If the three lines do not intersect, no exact solution exists but we can ask for the combination of w_1 and w_2 which minimizes the quadratic

error. This is the point inside the triangle shown in Figure 8.7, which has the minimal cumulative quadratic distance to the three sides of the triangle.

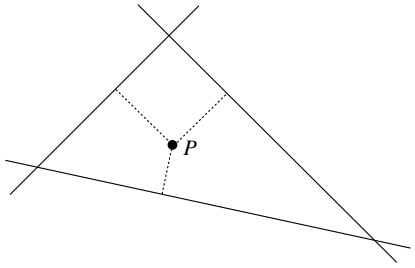

Fig. 8.7. Three linear constraints in weight space

Now for the interesting part: the point of intersection of linear equations can be found by linear algebraic methods such as the Gauss–Jacobi or the Gauss–Seidel method. Figure 8.8 shows how they work. If we are looking for the intersection of two lines, the Gauss–Jacobi method starts at some point in search space and projects this point in the directions of the axes on to the two lines considered in the example. The x-coordinate of the horizontal projection and the y-coordinate of the vertical projection define the next iteration point. It is not difficult to see that in the example this method converges to the point of intersection of the two lines. The Gauss–Seidel method deals with each linear equation individually. It first projects in the x direction, then in the y direction. Each projection is the new iteration point. This algorithm usually converges faster than the Gauss–Jacobi method [444].

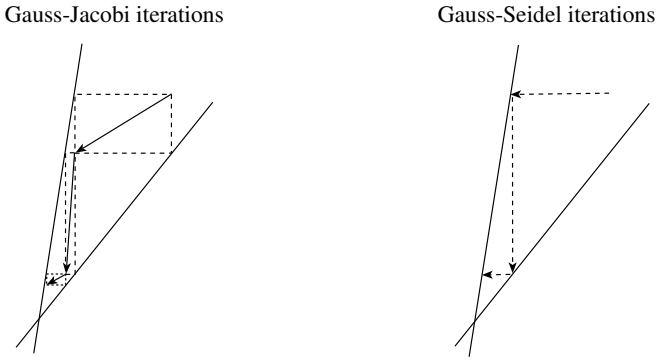

Fig. 8.8. The Gauss–Jacobi and Gauss–Seidel methods

Off-line and on-line backpropagation are in a certain sense equivalent to the Gauss–Jacobi and Gauss–Seidel methods. When on-line backpropagation

is used, the negative gradient of the error function is followed and this corresponds to following a line perpendicular to each one of the equations (8.1) to (8.3). In the case of the first input-output pattern, the error function is

$$\frac{1}{2}(x_1^1 w_1 + x_2^1 w_2 - y_1)^2$$

and the gradient (with respect to w_1 and w_2) is the vector

$$(x_1, x_2)$$

which is normal to the line defined by equation (8.1). By randomly alternating the selection of each pattern, on-line backpropagation iterates, always moving in the direction normal to the linear constraints. Figure 8.9 shows that this method can be used to find the solution to linear equations. If the directions of the axis and the linear constraints coincide, on-line backpropagation is the Gauss–Seidel method with a learning constant.

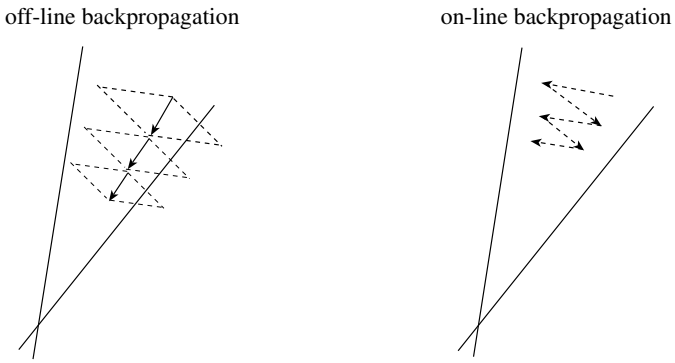

Fig. 8.9. Off-line and on-line backpropagation

Off-line backpropagation iterates by adding the corrections for each pattern. This means that the corrections in the direction normal to each linear constraint are calculated and the new iteration point is obtained by combining them. Figure 8.9 shows that this method is very similar to the Gauss–Jacobi method of linear algebra if we are looking for the solution to linear equations. Note that in both cases the size of the learning constant determines whether the iteration stops short of reaching the linear constraint or goes beyond it. This also depends on the curvature of the error function for each linear constraint (not shown in Figure 8.9).

In the nonlinear case, when a sigmoid is added as the activation function to the computing unit, the same kind of iteration process is used, but the sigmoid weights each one of the correction directions. On-line backpropagation always moves in the direction normal to the constraints, but the length of the search step is multiplied by the derivative of the sigmoid. Figure 8.10 shows the path of some iterations in the case of three input-output patterns and two weights.

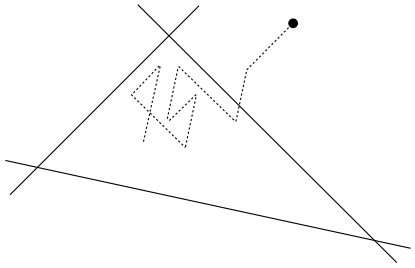

Fig. 8.10. On-line backpropagation iterations for sigmoidal units

Iterated Function Systems

Barnsley has shown that a set of affine transformations in a metric space can produce fractal structures when applied repetitively to a compact subset of points and its subsequent images. More specifically: an *Iterated Function System* (IFS) consists of a space of points X, a metric d defined in this space, and a set of affine contraction mappings $h_i : X \to X$, $i = 1, \ldots, N$. Given a nonvoid compact subset A_0 of points of X, new image subsets are computed successively according to the recursive formula

$$A_{n+1} = \bigcup_{j=1}^{N} h_j(A_n), \text{ for } n = 1, 2, \ldots.$$

A theorem guarantees that the sequence $\{A_n\}$ converges to a fixed point, which is called the *attractor* of the IFS. Moreover, the *Collage Theorem* guarantees that given any nonvoid compact subset of X we can always find an IFS whose associated attractor can arbitrarily approximate the given subset under a suitable metric.

For the case of an n-dimensional space X, an affine transformation applied to a point $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ is defined by a matrix $\mathbf{M}$ and a vector $\mathbf{t}$. The transformation is given by

$$\mathbf{x} \to \mathbf{M}\mathbf{x} + \mathbf{t}.$$

and is contractive if the determinant of $\mathbf{M}$ is smaller than one.

It is easy to show that the attractor of the IFS can be approximated by taking any point a_o which belongs to the attractor and computing the sequence a_n, whereby

$$a_{n+1} = h_k(a_n),$$

and the affine transformation h_k is selected randomly from the IFS. The infinite sequence $\{a_n\}$ is a *dense* subset of the attractor. This means that we can produce a good graphical approximation of the attractor of the IFS with this simple randomized method. Since it is easy to approximate a point belonging to the attractor (by starting from the origin and iterating with the IFS a fixed number of times), this provides us with the necessary initial point.

We now proceed to show that on-line backpropagation, in the case of a linear associator, defines a set of affine transformations which are applied in the course of the learning process either randomly or in a fixed sequence. The iteration path of the initial point is thus a fractal.

On-line Backpropagation and IFS

Consider a linear associator and the p n-dimensional patterns $\mathbf{x}^1, \mathbf{x}^2, \ldots, \mathbf{x}^{\mathrm{p}}$. The symbol x_i^j denotes the i-th coordinate of the j-th pattern. The targets for training are the p real values $y^1, y^2, \ldots, y^p$. We denote by $w_1, w_2, \ldots, w_n$ the weights of the linear associator. In on-line backpropagation the error function is determined for each individual pattern. For pattern j the error is

$$E_j = \frac{1}{2}(w_1 x_1^j + w_2 x_2^j + \cdots + w_n x_n^j - y^j)^2.$$

The correction for each weight $w_i, i = 1, \ldots, n$, is given by

$$w_i \rightarrow w_i - \gamma x_i^j (w_1 x_1^j + w_2 x_2^j + \cdots + w_n x_n^j - y^j),$$

where γ is the learning constant being used. This can be also written as

$$w_i \rightarrow w_i - \gamma (w_1 x_i^j x_1^j + w_2 x_i^j x_2^j + \cdots + w_n x_i^j x_n^j) - \gamma x_i^j y^j \,. \tag{8.4}$$

Let M_j be the matrix with elements $m_{ik} = x_i^j x_k^j$ for $i, k = 1, \ldots, n$. Equation (8.4) can be rewritten in matrix form (considering all values of i) as

$$\mathbf{w} \rightarrow \mathbf{I}\mathbf{w} - \gamma \mathbf{M}_j \mathbf{w} - \mathbf{t}_j,$$

where $\mathbf{t}_j$ is the column vector with components $x_i^j y^j$ for $i = 1, \ldots, n$ and $\mathbf{I}$ is the identity matrix. We can rewrite the above expression as

$$\mathbf{w} \rightarrow (\mathbf{I} - \gamma \mathbf{M}_j)\mathbf{w} - \mathbf{t}_j.$$

This is an affine transformation, which maps the current point in weight space into a new one. Note that each pattern in the training set defines a different affine transformation of the current point $(w_1, w_2, \ldots, w_n)$.

In on-line backpropagation with a random selection of input patterns, the initial point in weight space is successively transformed in just the way prescribed by a randomized IFS algorithm. This means that the sequence of updated weights approximates the attractor of the IFS defined by the input patterns, i.e., the iteration path in weight space is a fractal.

This result can be visualized in the following manner: given a point $(w'_1, w'_2, \ldots, w'_n)$ in weight space, the square of the distance ℓ to the hyperplane defined by the equation

$$w_1 x_1^j + w_2 x_2^j + \cdots + w_n x_n^j = y^j$$

Fig. 8.11. Iteration paths in weight space for different learning rates

is given by

$$\ell^2 = \frac{(w'_1 x_1^j + w'_2 x_2^j + \cdots + w'_n x_n^j - y^j)^2}{(x_1^j)^2 + \cdots + (x_n^j)^2}.$$

Comparing this expression with the updating step performed by on-line backpropagation, we see that each backpropagation step amounts to displacing the current point in weight space in the direction normal to the hyperplane defined by the input and target patterns. A learning rate γ with value

$$\gamma = \frac{1}{(x_1^j)^2 + \cdots + (x_n^j)^2} \tag{8.5}$$

produces exact learning of the j-th pattern, that is, the point in weight space is brought up to the hyperplane defined by the pattern. Any value of γ below this threshold displaces the iteration path just a fraction of the distance to the hyperplane. The iteration path thus remains trapped in a certain region of weight space near to the optimum.

Figure 8.11 shows a simple example in which three input-output patterns define three lines in a two-dimensional plane. Backpropagation for a linear associator will find the point in the middle as the solution with the minimal error for this task. Some combinations of learning rate, however, keep the iteration path a significant distance away from this point. The fractal structure of the iteration path is visible in the first three graphics. With $\gamma = 0.25$ the fractal structure is obliterated by a dense approximation to the whole triangular region. For values of γ under 0.25 the iteration path repeatedly comes arbitrarily near to the local minimum of the error function.

It is clear that on-line backpropagation with random selection of the input patterns yields iteration dynamics equivalent to those of IFS. When the learning constant is such that the affine transformations overlap, the iteration path densely covers a certain region of weight space around the local minimum of the error function. Practitioners know that they have to systematically reduce the size of the learning step, because otherwise the error function begins oscillating. In the case of linear associators and on-line backpropagation this means that the iteration path has gone fractal. Chaotic behavior of recurrent neural networks has been observed before [281], but in our case we are dealing with very simple feed-forward networks which fall into a similar dynamic.

In the case of sigmoid units at the output, the error correction step is no longer equivalent to an affine transformation, but the updating step is very similar. It can be shown that the error function for sigmoid units can approximate a quadratic function for suitable parameter combinations. In this case the iteration dynamics will not differ appreciably from those discussed in this chapter. Nonlinear updating steps should also produce fractal iteration paths of a more complex nature.

8.2 Some simple improvements to backpropagation

Since learning in neural networks is an *NP*-complete problem and since traditional gradient descent methods are rather slow, many alternatives have been tried in order to accelerate convergence. Some of the proposed methods are mutually compatible and a combination of them normally works better than each method alone [340]. But apart from the learning algorithm, the basic point to consider before training a neural network is *where* to start the iterative learning process. This has led to an analysis of the best possible weight initialization strategies and their effect on the convergence speed of learning [328].

8.2.1 Initial weight selection

A well-known initialization heuristic for a feed-forward network with sigmoidal units is to select its weights with uniform probability from an interval $[-\alpha, \alpha]$. The zero mean of the weights leads to an expected zero value of the total input to each node in the network. Since the derivative of the sigmoid at each node reaches its maximum value of $1/4$ with exactly this input, this should lead in principle to a larger backpropagated error and to more significant weight updates when training starts. However, if the weights in the networks are very small (or all zero) the backpropagated error from the output to the hidden layer is also very small (or zero) and practically no weight adjustment takes place between input and hidden layer. Very small values of α paralyze learning, whereas very large values can lead to saturation of the nodes in the network and to flat zones of the error function in which, again, learning is very

slow. Learning then stops at a suboptimal local minimum [277]. Therefore it is natural to ask what is the best range of values for α in terms of the learning speed of the network.

Some authors have conducted empirical comparisons of the values for α and have found a range in which convergence is best [445]. The main problem with these results is that they were obtained from a limited set of examples and the relation between learning step and weight initialization was not considered. Others have studied the percentage of nodes in a network which become paralyzed during training and have sought to minimize it with the "best" α [113]. Their empirical results show, nevertheless, that there is not a single α which works best, but a very broad range of values with basically the same convergence efficiency.

Maximizing the derivatives at the nodes

Let us first consider the case of an output node. If n different edges with associated weights $w_1, w_2, \ldots, w_n$ point to this node, then after selecting weights with uniform probability from the interval $[-\alpha, \alpha]$, the expected total input to the node is

$$\left\langle \sum_{i=1}^{n} w_i x_i \right\rangle = 0$$

where $x_1, x_2, \ldots, x_n$ are the input values transported through each edge. We have assumed that these inputs and the initial weights are not correlated. By the law of large numbers we can also assume that the total input to the node has a Gaussian distribution Numerical integration shows that the expected value of the derivative is a decreasing function of the standard deviation σ. The expected value falls slowly with an increase of the variance. For $\sigma = 0$ the expected value is 0.25 and for $\sigma = 4$ it is still 0.12, that is, almost half as big.

The variance of the total input to a node is

$$\sigma^2 = E((\sum_{i=1}^{n} w_i x_i)^2) - E((\sum_{i=1}^{n} w_i x_i))^2 = \sum_{i=1}^{n} E(w_i^2) E(x_i^2),$$

since inputs and weights are uncorrelated. For binary vectors we have $E(x_i^2) = 1/3$ and the above equation simplifies to

$$\sigma = \frac{1}{3}\sqrt{n}\,\alpha.$$

If $n = 100$, selecting weights randomly from the interval $[-1.2, 1.2]$ leads to a variance of 4 at the input of a node with 100 connections and to an expected value of the derivative equal to 0.12.

Therefore in small networks, in which the maximum input to each node comes from fewer than 100 edges, the expected value of the derivative is not very sensitive to the width α of the random interval, when α is small enough.

Maximizing the backpropagated error

In order to make corrections to the weights in the first block of weights (those between input and hidden layer) easier, the backpropagated error should be as large as possible. Very small weights between hidden and output nodes lead to a very small backpropagated error, and this in turn to insufficient corrections to the weights. In a network with m nodes in the hidden layer and k nodes in the output layer, each hidden node h receives a backpropagated input δ_h from the k output nodes, equal to

$$\delta_h = \sum_{i=1}^{k} w_{hi} s'_i \delta_i^0,$$

where the weights $w_{hi}, i = 1, \ldots, k$, are the ones associated with the edges from hidden node h to output node i, s'_i is the sigmoid's derivative at the output node i, and δ_i^0 is the difference between output and target also at this node.

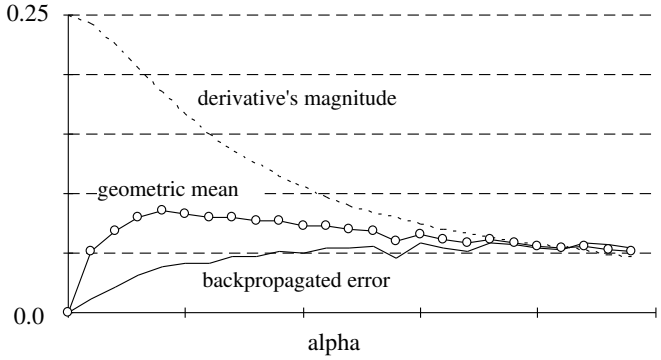

Fig. 8.12. Expected values of the backpropagated error and the sigmoid's derivative

After initialization of the network's weights the expected value of δ_h is zero. In the first phase of learning we are interested in *breaking the symmetry* of the hidden nodes. They should specialize in the recognition of different features of the input. By making the variance of the backpropagated error larger, each hidden node gets a greater chance of pulling apart from its neighbors. The above equation shows that by making the initialization interval $[-\alpha, \alpha]$ wider, two contradictory forces come into play. On the one hand, the variance of the weights becomes larger, but on the other hand, the expected value of the derivative s'_k becomes lower. We would like to make δ_h as large as possible, but without making s'_i too low, since weight corrections in the second block of weights are proportional to s'_i. Figure 8.12 shows the expected values of the derivative at the output nodes, the expected value of the backpropagated

error for the hidden nodes as a function of α, and the geometric mean of both values. The data in the figure was obtained from Monte Carlo trials, assuming a constant expected value of δ_i^0. Forty hidden and one output unit were used.

The figure shows, again, that the expected value of the sigmoid's derivative falls slowly with an increasing α, but the value of the backpropagated error is sensitive to small values of α. In the case shown in the figure, any possible choice of α between 0.5 and 1.5 should lead to virtually the same performance. This explains the flat region of possible values for α found in the experiments published in [113] and [445]. Consequently, the best values for α depend on the exact number of input, hidden, and output units, but the learning algorithm should not be very sensitive to the exact α chosen from a certain range of values.

8.2.2 Clipped derivatives and offset term

One of the factors which leads to slow convergence of gradient descent methods is the small absolute value of the partial derivatives of the error function computed by backpropagation. The derivatives of the sigmoid stored at the computing units can easily approach values near to zero and since several of them can be multiplied in the backpropagation step, the length of the gradient can become too small. A solution to this problem is clipping the derivative of the sigmoid, so that it can never become smaller than a predefined value. We could demand, for example, that $s'(x) \geq 0.01$. In this case the "derivatives" stored in the computing units do not correspond exactly to the actual derivative of the sigmoid (except in the regions where the derivative is not too small). However, the partial derivatives have the correct sign and the gradient direction is not significantly affected.

It is also possible to add a small constant ε to the derivative and use $s'(x)+\varepsilon$ for the backpropagation step. The net effect of an offset value for the sigmoid's derivative is to pull the iteration process out of flat regions of the error function. Once this has happened, backpropagation continues iterating with the exact gradient direction. It has been shown in many different learning experiments that this kind of heuristic, proposed by Fahlman [130], among others, can contribute significantly to accelerate several different variants of the standard backpropagation method [340]. Note that the offset term can be implemented very easily when the sigmoid is not computed at the nodes but only read from a table of function values. The table of derivatives can combine clipping of the sigmoid values with the addition of an offset term, to enhance the values used in the backpropagation step.

8.2.3 Reducing the number of floating-point operations

Backpropagation is an expensive algorithm because a straightforward implementation is based on floating-point operations. Since all values between 0 and 1 are used, problems of precision and stability arise which are normally

avoided by using 32-bit or 64-bit floating-point arithmetic. There are several possibilities to reduce the number of floating-point operations needed.

Avoiding the computation of the squashing function

If the nonlinear function used at each unit is a sigmoid or the hyperbolic tangent, then an exponential function has to be computed and this requires a sequence of floating-point operations. However, computation of the nonlinearity can be avoided by using tables stored at each unit, in which for an interval $[x_i, x_{i+1}]$ in the real line the corresponding approximation to the sigmoid is stored. A piecewise linear approximation can be used as shown in Figure 8.13, so that the output of the unit is $y = a_i + a_{i+1}(x - x_i)$ when $x_i \leq x < x_{i+1}$ and where $a_1 = s(x_i)$ and $a_{i+1} = s(x_{i+1})$. Computation of the nonlinear function is reduced in this way to a comparison, a table lookup, and an interpolation. Another table holding some values of the sigmoid's derivative can be stored at each unit for the backpropagation step. A piecewise linear approximation can also be used in this case. This strategy is used in chips for neural computation in order to avoid using many logic gates.

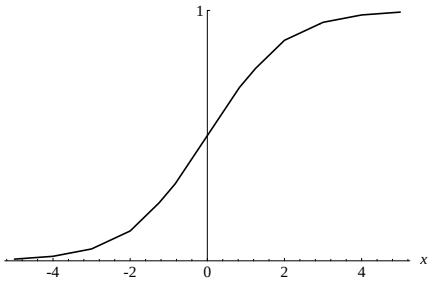

Fig. 8.13. Piecewise linear approximation to the sigmoid

Avoiding the nonlinear function at the output

In some cases the sigmoid at the output of the network can be eliminated. If the output vectors in the training set are m-dimensional vectors of the form $(t_1, \ldots, t_m)$ with $0 < t_i < 1$ for $i = 1, \ldots, m$, then a new training set can be defined with the same input vectors and output vectors of the form $(s^{-1}(y_1), \ldots, s^{-1}(y_m))$, where the function s^{-1} is the inverse of the sigmoid. The sigmoid is eliminated from the output units and the network is trained with standard backpropagation. This strategy will save some operations but its applicability depends on the problem, since the sigmoid is equivalent to a kind of weighting of the output error. The inputs 100 or 1000 produce almost the same sigmoid output, but the two numbers are very different when compared directly. Only more knowledge about the specific application can help

to decide if the nonlinearity at the output can be eliminated (see Sect. 9.1.3 on logistic regression).

Fixed-point arithmetic

Since integer operations can be executed in many processors faster than floating-point operations, and since the outputs of the computing units are values in the interval $(0, 1)$ or $(-1, 1)$, it is possible to adopt a fixed-point representation and perform all necessary operations with integers. By convention we can define the last 12 bits of a number as its fractional part and the three previous bits as its integer part. Using a sign bit it is possible to represent numbers in the interval $(-8, 8)$ with a precision of $2^{-12} \approx 0.00025$. Care has to be taken to re-encode the input and output vectors, to define the tables for the sigmoid and its derivatives and to implement the correct arithmetic (which requires a shift after multiplications and tests to avoid overflow). Most of the more popular neural chips implement some form of fixed-point arithmetic (see Chap. 16).

Some experiments show that in many applications it is enough to reserve 16 bits for the weights and 8 for the coding of the outputs, without affecting the convergence of the learning algorithm [31]. Holt and Baker compared the results produced by networks with floating-point and fixed-point parameters using the Carnegie-Mellon benchmarks [197]. Their results confirmed that a combination of 16-bit and 8-bit coding produces good results. In four of five benchmarks the result of the comparison was "excellent" for fixed-point arithmetic and in the other case "good". Based on these results, groups developing neural computers like the CNAPS of Adaptive Solutions [175] and SPERT in Berkeley [32] decided to stick to 16-bit and 8-bit representations.

Reyneri and Filippi [364] did more extensive experimentation on this problem and arrived at the conclusion that the necessary word length of the representation depends on the learning constant and *the kind of learning algorithm used*. This was essentially confirmed by the experiments done by Pfister on a fixed-point neurocomputer [341]. Standard backpropagation can diverge in some cases when the fixed-point representation includes less than 16 bits. However, modifying the learning algorithm and adapting the learning constant reduced the necessary word length to 14 bits. With the modified algorithm 16 bits were more than enough.

8.2.4 Data decorrelation

We have already mentioned that if the principal axes of the quadratic approximation of the error function are too dissimilar, gradient descent can be slowed down arbitrarily. The solution lies in decorrelating the data set and there is now ample evidence that this preconditioning step is beneficial for the learning algorithm [404, 340].

One simple decorrelation strategy consists in using bipolar vectors. We showed in Chap. 6 that the solution regions defined by bipolar data for perceptrons are more symmetrically distributed than when binary vectors are used. The same holds for multilayer networks. Pfister showed that convergence of the standard backpropagation algorithm can be improved and that a speedup between 1.91 and 3.53 can be achieved when training networks for several small benchmarks (parity and clustering problems) [341]. The exception to this general result are encode-decode problems in which the data consists of sparse vectors (n-dimensional vectors with a single 1 in a component). In this case binary coding helps to focus on the corrections needed for the relevant weights. But if the data consists of non-sparse vectors, bipolar coding usually works better. If the input data consists of N real vectors $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N$ it is usually helpful to center the data around the origin by subtracting the centroid $\hat{\mathbf{x}}$ of the distribution, in such a way that the new input data consists of the vectors $\mathbf{x}_i - \hat{\mathbf{x}}$.

PCA and adaptive decorrelation

Centering the input and output data is just the first step in more sophisticated preconditioning methods. One of them is principal component analysis, already discussed in Chap. 5. Using PCA it is possible to reduce the number of vector components (when there is redundancy) and to order the remaining ones according to their importance.

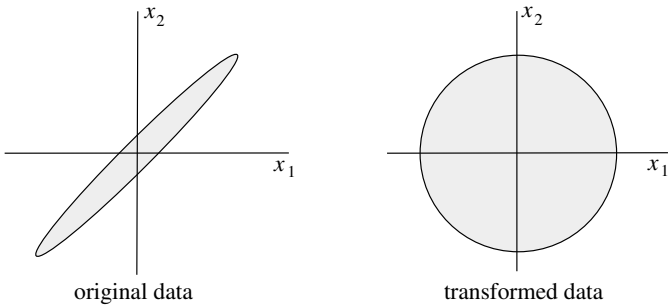

Fig. 8.14. Data distribution before and after preconditioning

Assume that the data distribution is the one shown on the left side of Figure 8.14. Any of the points in the ellipsoid can be a future input vector to the network. Two vectors selected randomly from the distribution have a large probability of not being orthogonal (since their components are correlated). After a rotation of the axes and a scaling step the data distribution becomes the one shown to the right in Figure 8.14. Note that the transformation is one-to-one and therefore invertible. But now, any two vectors selected randomly

from the new distribution (a sphere in n-dimensional space) have a greater probability of being orthogonal. The transformation of the data is linear (a rotation followed by scaling) and can be applied to new data as it is provided.

Note that we do not know the exact form of the data distribution. All we have is a training set (the dots in Figure 8.14) from which we can compute the optimal transformation. Principal component analysis is performed on the available data. This gives us a new encoding for the data set and also a linear transformation for new data. The scaling factors are the reciprocal of the variances of each component (see Exercise 2). PCA preconditioning speeds up backpropagation in most cases, except when the data consists of sparse vectors.

Silva and Almeida have proposed another data decorrelation technique, called Adaptive Data Decorrelation, which they use to precondition data [404]. A linear layer is used to transform the input vectors, and another to transform the outputs of the hidden units. The linear layer consists of as many output as input units, that is, it only applies a linear transformation to the vectors. Consider an n-dimensional input vector $\mathbf{x} = (x_1, x_2, \ldots, x_n)$. Denote the output of the linear layer by $(y_1, y_2, \ldots, y_n)$. The objective is to make the expected value of the correlation coefficient r_{ij} of the i-th and j-th output units equal to Kronecker's delta δ_{ij}, i.e.,

$$r_{ij} =< y_i y_j >= \delta_{ij}.$$

The expected value is computed over all vectors in the data set. The algorithm proposed by Silva and Almeida is very simple: it consists in pulling the weight vector of the i-th unit away from the weight vector of the j-th unit, whenever $r_{ij} > 0$ and in the direction of w_j when $r_{ij} < 0$. The precise formula is

$$\mathbf{w}_i^{k+1} = \mathbf{w}_i^k - \beta \sum_{j \neq i}^{n} r_{ij} \mathbf{w}_j^k,$$

where $\mathbf{w}_m^p$ denotes the weight vector of the m-th unit at the p-th iteration and β is a constant. Data normalization is achieved by including a positive or negative term according to whether r_{ii} is smaller or greater than 1:

$$\mathbf{w}_i^{k+1} = (1+\beta)\mathbf{w}_i^k - \beta \sum_{j=1}^{n} r_{ij} \mathbf{w}_j^k.$$

The proof that the algorithm converges (under certain assumptions) can be found in [404]. Adaptive Decorrelation can accelerate the convergence of backpropagation almost as well as principal component analysis, but is somewhat sensitive to the choice of the constant β and the data selection process [341].

Active data selection

For large training sets, redundancy in the data can make prohibitive the use of off-line backpropagation techniques. On-line backpropagation can still perform well under these conditions if the training pairs are selected randomly. Since some of the fastest variations of backpropagation work off-line, there is a strong motivation to explore methods of reducing the size of the effective training set.

Some authors have proposed training a network with a subset of the training set, adding iteratively the rest of the training pairs [261]. This can be done by testing the untrained pairs. If the error exceeds a certain threshold (for example if it is one standard deviation larger than the average error on the effective training set), the pair is added to the effective training set and the network is retrained when more than a certain number of pairs have been recruited [369].

8.3 Adaptive step algorithms

The class of adaptive step algorithms uses a very similar basic strategy: the step size is increased whenever the algorithm proceeds down the error function over several iterations. The step size is decreased when the algorithm jumps over a valley of the error function. The algorithms differ according to the kind of information used to modify the step size.

In learning algorithms with a *global* learning rate, this is used to update all weights in the network. The learning rate is made bigger or smaller according to the iterations made so far.

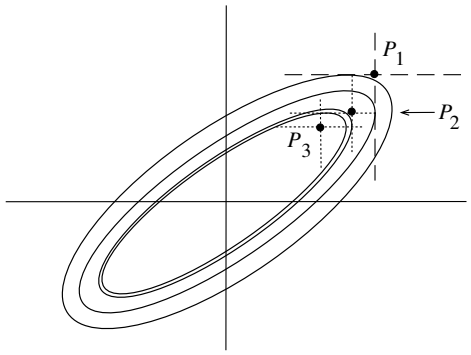

Fig. 8.15. Optimization of the error function with updates in the directions of the axes

In algorithms with a *local* learning constant a different constant is used for each weight. Whereas in standard backpropagation a single constant γ is used

to compute the weight corrections, in four of the algorithms considered below each weight w_i has an associated learning constant γ_i so that the updates are given by

$$\Delta w_i = -\gamma_i \frac{\partial E}{\partial w_i}.$$

The motivation behind the use of different learning constants for each weight is to try to "correct" the direction of the negative gradient to make it point directly to the minimum of the error function. In the case of a degenerate error function, the gradient direction can lead to many oscillations. An adequate scaling of the gradient components can help to reach the minimum in fewer steps.

Figure 8.15 shows how the optimization of the error function proceeds when only one-dimensional optimization steps are used (from point P_1 to P_2, and then to P_3). If the lengths of the principal axes of the quadratic approximation are very different, the algorithm can be slowed by an arbitrary factor.

8.3.1 Silva and Almeida's algorithm

The method proposed by Silva and Almeida works with different learning rates for each weight in a network [403]. Assume that the network consists of n weights $w_1, w_2, \ldots, w_n$ and denote the individual learning rates by $\gamma_1, \gamma_2, \ldots, \gamma_n$. We can better understand how the algorithm works by looking at Figure 8.16. The left side of the figure shows the level curves of a quadratic approximation to the error function. Starting the iteration process as described before and as illustrated with Figure 8.15, we can see that the algorithm performs several optimization steps in the horizontal direction. Since horizontal cuts to a quadratic function are quadratic, what we are trying to minimize at each step is one of the several parabolas shown on the right side of Figure 8.16. A quadratic function of n variable weights has the general form

$$c_1^2 w_1^2 + c_2^2 w_2^2 + \cdots + c_n^2 w_n^2 + \sum_{i \neq j} d_{ij} w_i w_j + C,$$

where $c_1, \ldots, c_n$, the d_{ij} and C are constants. If the i-th direction is chosen for a one-dimensional minimization step the function to be minimized has the form

$$c_i^2 w_i^2 + k_1 w_i + k_2,$$

where k_1 and k_2 are constants which depend on the values of the 'frozen' variables at the current iteration point. This equation defines a family of parabolas. Since the curvature of each parabola is given by c_i^2, they differ just by a translation in the plane, as shown in Figure 8.16. Consequently, it makes sense to try to find the optimal learning rate for this direction, which is equal to $1/2c_i^2$, as discussed in Sect. 8.1.1. If we have a different learning rate, optimization proceeds as shown on the left of Figure 8.16: the first parabola

is considered and the negative gradient direction is followed. The iteration steps in the other dimensions change the family of parabolas which we have to consider in the next step, but in this case the negative gradient direction is also followed. The family of parabolas changes again and so on. With this quadratic approximation in mind the question then becomes, what are the optimal values for γ_1 to γ_n? We arrive at an additional optimization problem!

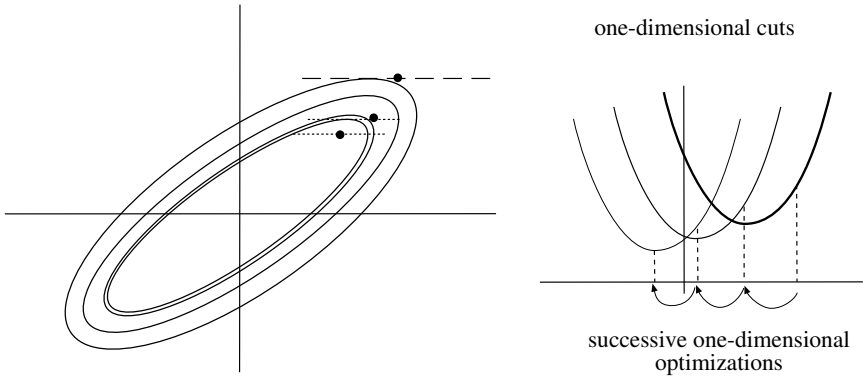

Fig. 8.16. One-dimensional cuts: family of parabolas

The heuristic proposed by Almeida is very simple: accelerate if, in two successive iterations, the sign of the partial derivative has not changed, and decelerate if the sign changes. Let $\nabla_i E^{(k)}$ denote the partial derivative of the error function with respect to weight w_i at the k-th iteration. The initial learning rates $\gamma_i^{(0)}$ for $i = 1, \ldots, n$ are initialized to a small positive value. At the k-th iteration the value of the learning constant for the next step is recomputed for each weight according to

$$\gamma_i^{(k+1)} = \begin{cases} \gamma_i^{(k)} u \text{ if } \nabla_i E^{(k)} \cdot \nabla_i E^{(k-1)} \geq 0 \\ \gamma_i^{(k)} d \text{ if } \nabla_i E^{(k)} \cdot \nabla_i E^{(k-1)} < 0 \end{cases}$$

The constants u (up) and d (down) are set by hand with $u > 1$ and $d < 1$. The weight updates are made according to

$$\Delta^{(k)} w_i = -\gamma_i^{(k)} \nabla_i E^{(k)} .$$

Note that due to the constants u and d the learning rates grow and decrease *exponentially*. This can become a problem if too many acceleration steps are performed successively. Obviously this algorithm does not follow the gradient direction exactly. If the level curves of the quadratic approximation of the error function are perfect circles, successive one-dimensional optimizations lead to the minimum after n steps. If the quadratic approximation has semi-axes of

very different lengths, the iteration process can be arbitrarily slowed. To avoid this, the updates can include a momentum term with rate α. Nevertheless, both kinds of corrections together are somewhat contradictory: the individual learning rates can only be optimized if the optimization updates are strictly one-dimensional. Tuning the constant α can therefore become quite problem-specific. The alternative is to preprocess the original data to achieve a more regular error function. This can have a dramatic effect on the convergence speed of algorithms which perform one-dimensional optimization steps.

8.3.2 Delta-bar-delta

The algorithm proposed by Jacobs is similar to Silva and Almeida's, the main difference being that acceleration of the learning rates is made with more caution than deceleration. The algorithm is started with individual learning rates $\gamma_1, \ldots, \gamma_n$ all set to a small value, and at the k-th iteration the new learning rates are set to

$$\gamma_i^{(k+1)} = \begin{cases} \gamma_i^{(k)} + u & \text{if } \nabla_i E^{(k)} \cdot \delta_i^{(k-1)} > 0 \\ \gamma_i^{(k)} d & \text{if } \nabla_i E^{(k)} \cdot \delta_i^{(k-1)} < 0 \\ \gamma_i^{(k)} & \text{otherwise,} \end{cases}$$

where u and d are constants and $\delta_i^{(k)}$ is an exponentially averaged partial derivative in the direction of weight w_i:

$$\delta_i^{(k)} = (1-\phi)\nabla_i E^{(k)} + \phi \delta_i^{(k-1)}.$$

The constant ϕ determines what weight is given to the last averaged term. The weight updates are performed without a momentum term:

$$\Delta^{(k)} w_i = -\gamma_i^{(k)} \nabla_i E^{(k)}.$$

The motivation for using an averaged gradient is to avoid excessive oscillations of the basic algorithm. The problem with this approach, however, is that a new constant has to be set again by the user and its value can also be highly problem-dependent. If the error function has regions which allow a good quadratic approximation, then $\phi = 0$ is optimal and we are essentially back to Silva and Almeida's algorithm.

8.3.3 Rprop

A variant of Silva and Almeida's algorithm is Rprop, first proposed by Riedmiller and Braun [366]. The main idea of the algorithm is to update the network weights using just the learning rate and the sign of the partial derivative of the error function with respect to each weight. This accelerates learning mainly in flat regions of the error function as well as when the iteration has

arrived close to a local minimum. To avoid accelerating or decelerating too much, a minimum value for the learning rates γ_{min} and a maximum value γ_{max} is enforced. The algorithm covers all of weight space with an n-dimensional grid of side γ_{min} and an n-dimensional grid of side length γ_{max}. The individual one-dimensional optimization steps can move on all possible intermediate grids. The learning rates are updated in the k-th iteration according to

$$\gamma_i^{(k+1)} = \begin{cases} \min(\gamma_i^{(k)} u, \gamma_{max}) & \text{if } \nabla_i E^{(k)} \cdot \nabla_i E^{(k-1)} > 0 \\ \max(\gamma_i^{(k)} d, \gamma_{min}) & \text{if } \nabla_i E^{(k)} \cdot \nabla_i E^{(k-1)} < 0 \\ \gamma_i^{(k)} & \text{otherwise,} \end{cases}$$

where the constants u and d satisfy $u > 1$ and $d < 1$, as usual. When $\nabla_i E^{(k)} \cdot \nabla_i E^{(k-1)} \geq 0$ the weight updates are given by

$$\Delta^{(k)} w_i = -\gamma_i^{(k)} \text{sgn}(\nabla_i E^{(k)}),$$

otherwise $\Delta^{(k)} w_i$ and $\nabla_i E^{(k)}$ are set to zero. In the above equation $\text{sgn}(\cdot)$ denotes the sign function with the peculiarity that $\text{sgn}(0) = 0$.

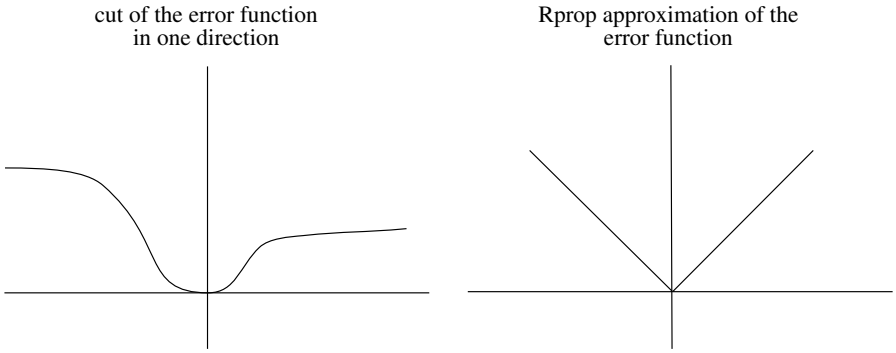

Fig. 8.17. Local approximation of Rprop

Figure 8.17 shows the kind of one-dimensional approximation of the error function used by Rprop. It may seem a very imprecise approach, but it works very well in the flat regions of the error function. Schiffmann, Joost, and Werner tested several algorithms using a medical data set and Rprop produced the best results, being surpassed only by the constructive algorithm called *cascade correlation* [391] (see Chap. 14).

Table 8.1 shows the results obtained by Pfister on some of the Carnegie Mellon Benchmarks [341]. The training was done on a CNAPS neurocomputer. Backpropagation was coded using some optimizations (bipolar vectors, offset term, etc.). The table shows that BP did well for most benchmarks but failed to converge in two of them. Rprop converged almost always (98% of

Table 8.1. Comparison of Rprop and batch backpropagation

benchmark	generations		time	
	BP	Rprop	BP	Rprop
sonar signals	109.7	82.0	8.6 s	6.9 s
vowels	–	1532.9	–	593.6 s
vowels (decorrelated)	331.4	319.1	127.8 s	123.6 s
NETtalk (200 words)	268.9	108.7	961.5 s	389.6 s
protein structure	347.5	139.2	670.1 s	269.1 s
digits	–	159.5	–	344.7 s

the time) and was faster by up to a factor of about 2.5 with respect to batch backpropagation.

It should be pointed out that the vowels recognition task was presented in two versions, one without and one with decorrelated inputs. In the latter case, backpropagation did converge and was almost as efficient as Rprop. This shows how important the preprocessing of the input data can be. Note that the overall speedup obtained is limited because the version of backpropagation used for the comparison was rather efficient.

8.3.4 The Dynamic Adaption algorithm

We close our discussion of adaptive step methods with an algorithm based on a global learning rate [386]. The idea of the method is to use the negative gradient direction to generate two new points instead of one. The point with the lowest error is used for the next iteration. If it is the farthest away the algorithm accelerates, by making the learning constant bigger. If it is the nearest one, the learning constant γ is reduced.

The k-th iteration of the algorithm consists of the following three steps:

- Compute

$$\mathbf{w}^{(k_1)} = \mathbf{w}^{(k)} - \nabla E(\mathbf{w}^{(k)})\gamma^{(k-1)} \cdot \xi$$
$$\mathbf{w}^{(k_2)} = \mathbf{w}^{(k)} - \nabla E(\mathbf{w}^{(k)})\gamma^{(k-1)} / \xi$$

  where ξ is a small constant (for example $\xi = 1.7$).
- Update the learning rate:

$$\gamma^{(k)} = \begin{cases} \gamma^{(k-1)} \cdot \xi \text{ if } E(\mathbf{w}^{(k_1)}) \leq E(\mathbf{w}^{(k_2)}) \\ \gamma^{(k-1)} / \xi \quad \text{otherwise.} \end{cases}$$

- Update the weights:

$$\mathbf{w}^{(k+1)} = \begin{cases} \mathbf{w}^{(k_1)} \text{ if } E(\mathbf{w}^{(k_1)}) \leq E(\mathbf{w}^{(k_2)}) \\ \mathbf{w}^{(k_2)} \quad \text{otherwise.} \end{cases}$$

The algorithm is not as good as the adaptive step methods with a local learning constant, but is very easy to implement. The overhead involved is an extra feed-forward step.

8.4 Second-order algorithms

The family of second-order algorithms considers more information about the shape of the error function than the mere value of the gradient. A better iteration can be performed if the curvature of the error function is also considered at each step. In second-order methods a quadratic approximation of the error function is used [43]. Denote all weights of a network by the vector $\mathbf{w}$. Denote the error function by $E(\mathbf{w})$. The truncated Taylor series which approximates the error function E is given by

$$E(\mathbf{w}+\mathbf{h}) \approx E(\mathbf{w}) + \nabla E(\mathbf{w})^{\mathrm{T}}\mathbf{h} + \frac{1}{2}\mathbf{h}^{\mathrm{T}}\nabla^2 E(\mathbf{w})\mathbf{h}, \tag{8.6}$$

where $\nabla^2 E(\mathbf{w})$ is the $n \times n$ Hessian matrix of second-order partial derivatives:

$$\nabla^2 E(\mathbf{w}) = \begin{pmatrix} \frac{\partial^2 E(\mathbf{w})}{\partial w_1^2} & \frac{\partial^2 E(\mathbf{w})}{\partial w_1 \partial w_2} & \cdots & \frac{\partial^2 E(\mathbf{w})}{\partial w_1 \partial w_n} \\ \frac{\partial^2 E(\mathbf{w})}{\partial w_2 \partial w_1} & \frac{\partial^2 E(\mathbf{w})}{\partial w_2^2} & \cdots & \frac{\partial^2 E(\mathbf{w})}{\partial w_2 \partial w_n} \\ \vdots & & \ddots & \vdots \\ \frac{\partial^2 E(\mathbf{w})}{\partial w_n \partial w_1} & \frac{\partial^2 E(\mathbf{w})}{\partial w_n \partial w_2} & \cdots & \frac{\partial^2 E(\mathbf{w})}{\partial w_n^2} \end{pmatrix}.$$

The gradient of the error function can be computed by differentiating (8.6):

$$\nabla E(\mathbf{w}+\mathbf{h})^{\mathrm{T}} \approx \nabla E(\mathbf{w})^{\mathrm{T}} + \mathbf{h}^{\mathrm{T}}\nabla^2 E(\mathbf{w}).$$

Equating to zero (since we are looking for the minimum of E) and solving, we get

$$\mathbf{h} = -(\nabla^2 E(\mathbf{w}))^{-1}\nabla E(\mathbf{w}), \tag{8.7}$$

that is, the minimization problem can be solved in a single step if we have previously computed the Hessian matrix and the gradient, of course under the assumption of a quadratic error function.

Newton's method works by using equation (8.7) iteratively. If we denote now the weight vector at the k-th iteration by $\mathbf{w}^{(k)}$, the new weight vector $\mathbf{w}^{(k+1)}$ is given by

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - (\nabla^2 E(\mathbf{w}))^{-1}\nabla E(\mathbf{w}). \tag{8.8}$$

Under the quadratic approximation, this will be a position where the gradient has reduced its magnitude. Iterating several times we can get to the minimum of the error function.

However, computing the Hessian can become quite a difficult task. Moreover, what is needed is the *inverse* of the Hessian. In neural networks we have to repeat this computation on each new iteration. Consequently, many techniques have been proposed to approximate the second-order information contained in the Hessian using certain heuristics.

Pseudo-Newton methods are variants of Newton's method that work with a simplified form of the Hessian matrix [48]. The non-diagonal elements are all set to zero and only the diagonal elements are computed, that is, the second derivatives of the form $\partial^2 E(\mathbf{w})/\partial w_i^2$. In that case equation (8.8) simplifies (for each component of the weight vector) to

$$w_i^{(k+1)} = w_i^{(k)} - \frac{\nabla_i E(\mathbf{w})}{\partial^2 E(\mathbf{w})/\partial w_i^2}. \tag{8.9}$$

No matrix inversion is necessary and the computational effort involved in finding the required second partial derivatives is limited. In Sect. 8.4.3 we show how to perform this computation efficiently.

Pseudo-Newton methods work well when the error function has a nice quadratic form, otherwise care should be exercised with the corrections, since a small second-order partial derivative can lead to extremely large corrections.

8.4.1 Quickprop

In this section we consider an algorithm which tries to take second-order information into account but follows a rather simple approach: only one-dimensional minimization steps are taken and information about the curvature of the error function in the update directions is obtained from the current and the last partial derivative of the error function in this direction.

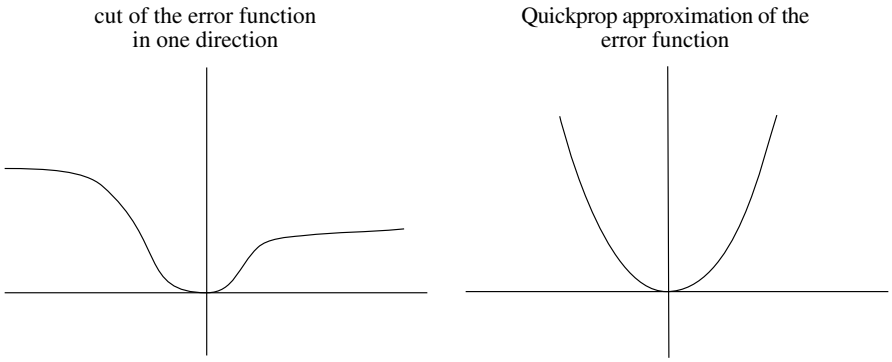

Fig. 8.18. Local approximation of Quickprop

Quickprop is based on independent optimization steps for each weight. A quadratic one-dimensional approximation of the error function is used. The

update term for each weight at the k-th step is given by

$$\Delta^{(k)} w_i = \Delta^{(k-1)} w_i \left(\frac{\nabla_i E^{(k)}}{\nabla_i E^{(k-1)} - \nabla_i E^{(k)}} \right), \tag{8.10}$$

where it is assumed that the error function has been computed at steps $(k-1)$ and k using the weight difference $\Delta^{(k-1)} w_i$, obtained from a previous Quickprop or an standard gradient descent step.

Note that if we rewrite (8.10) as

$$\Delta^{(k)} w_i = -\frac{\nabla_i E^{(k-1)}}{(\nabla_i E^{(k)} - \nabla_i E^{(k)})/\Delta^{(k-1)} w_i} \tag{8.11}$$

then the weight update in (8.11) is of the same form as the weight update in (8.9). The denominator is just a discrete approximation to the second-order derivative $\partial^2 E(\mathbf{w})/\partial w_i^2$. Quickprop is therefore a discrete pseudo-Newton method that uses so-called *secant steps*.

According to the value of the derivatives, Quickprop updates may become very large. This is avoided by limiting $\Delta^{(k)} w_i$ to a constant times $\Delta^{(k-1)}$. See [130] for more details on the algorithm and the handling of different problematic situations. Since the assumptions on which Quickprop is based are more far-fetched than the assumptions used by, for example, Rprop, it is not surprising that Quickprop has some convergence problems with certain tasks and requires careful handling of the weight updates [341].

8.4.2 QRprop

Pfister and Rojas proposed an algorithm that adaptively switches between the Manhattan method used by Rprop and local one-dimensional secant steps like those used by Quickprop [340, 341]. Since the resulting algorithm is a hybrid of Rprop and Quickprop it was called QRprop.

QRprop uses the individual learning rate strategy of Rprop if two consecutive error function gradient components $\nabla_i E^{(k)}$ and $\nabla_i E^{(k-1)}$ have the same sign or one of these components equals zero. This produces a fast approach to a region of minimum error. If the sign of the gradient changes, we know that we have overshot a local minimum in this specific weight direction, so now a second-order step (a Quickprop step) is taken. If we assume that in this direction the error function is independent from all the other weights, a step based on a quadratic approximation will be far more accurate than just stepping half way back as it is (indirectly) done by Rprop. Since the error function depends on all weights and since the quadratic approximation will be better the closer the two investigated points lie together, QRprop constrains the size of the secant step to avoid large oscillations of the weights. In summary:

i) As long as $\nabla_i E^{(k)} \cdot \nabla_i E^{(k-1)} > 0$ holds, Rprop steps are performed because we assume that a local minimum lies ahead.

ii) If $\nabla_i E^{(k)} \cdot \nabla_i E^{(k-1)} < 0$, which suggests that a local minimum has been overshot, then, unlike Rprop, neither the individual learning rate γ_i nor the weight w_i are changed. A "marker" is defined by setting $\nabla_i E^{(k)} := 0$ and the secant step is performed in the subsequent iteration.

iii) If $\nabla_i E^{(k)} \cdot \nabla_i E^{(k-1)} = 0$, this means that either a marker was set in the previous step, or one of the gradient components is zero because a local minimum has been directly hit. In both cases we are near a local minimum and the algorithm performs a second-order step. The secant approximation is done using the gradient information provided by $\nabla_i E^{(k)}$ and $\nabla_i E^{(k-2)}$. The second-order approximation is used even when $\nabla_i E^{(k)} \cdot \nabla_i E^{(k-2)} > 0$. Since we know that we are near a local minimum (and very likely we have already overshot it in the previous step), the second-order approximation is still a better choice than just stepping halfway back.

iv) In the secant step the quadratic approximation

$$q_i := |\nabla_i E^{(k)})/(\nabla_i E^{(k)} - \nabla_i E^{(k-2)})|$$

is constrained to a certain interval to avoid very large or very small updates.

Therefore, the k-th iteration of the algorithm consists of the following steps:

Step 1: Update the individual learning rates

if $(\nabla_i E^{(k)} \cdot \nabla_i E^{(k-1)} = 0)$ then
  if $(\nabla_i E^{(k)} \neq (\nabla_i E^{(k-2)})$ then
    $q_i = \max\left(d, \min\left(1/u, \left|\frac{\nabla_i E^{(k)}}{\nabla_i E^{(k)} - \nabla_i E^{(k-2)}}\right|\right)\right)$
  else
    $q_i = 1/u$
  endif
endif

$$\gamma_i^{(k)} = \begin{cases} \min(u \cdot \gamma_i^{(k-1)},\ \gamma_{max}) & \nabla_i E^{(k)} \cdot \nabla_i E^{(k-1)} > 0 \\ \gamma_i^{(k-1)} & \text{if } \nabla_i E^{(k)} \cdot \nabla_i E^{(k-1)} < 0 \\ \max(q_i \cdot \gamma_i^{(k-1)},\ \gamma_{min}) & \nabla_i E^{(k)} \cdot \nabla_i E^{(k-1)} = 0 \end{cases}$$

Step 2: Update the weights

$$w_i^{(k+1)} = \begin{cases} w_i^{(k)} - \gamma_i^{(k)} \cdot \text{sgn}(\nabla_i E^{(k)}) & \text{if } \nabla_i E^{(k)} \cdot \nabla_i E^{(k-1)} \geq 0 \\ w_i^{(k)} & \text{otherwise} \end{cases}$$

If $(\nabla_i E^{(k)} \cdot \nabla_i E^{(k-1)} < 0)$ set $\nabla_i E^{(k)} := 0$.

The constants d, u, γ_{min}, and γ_{max} must be chosen in advance as for other adaptive steps methods. QRprop has shown to be an efficient algorithm that can outperform both of its original algorithmic components. Table 8.2 shows the speedup obtained with QRprop relative to Rprop for the Carnegie Mellon benchmarks [341].

Table 8.2. Speedup of QRprop relative to Rprop

benchmark	speedup
sonar signals	1.01
vowels	1.33
vowels (decorrelated)	1.30
NETtalk (200 words)	1.02
protein structure	1.14
digits	1.29
average	1.18

8.4.3 Second-order backpropagation

In this section we introduce *second-order backpropagation*, a method to efficiently compute the Hessian of a linear network of one-dimensional functions. This technique can be used to get explicit symbolic expressions or numerical approximations of the Hessian and could be used in parallel computers to improve second-order learning algorithms for neural networks. Methods for the determination of the Hessian matrix in the case of multilayered networks have been studied recently [58].

We show how to efficiently compute the elements of the Hessian matrix using a graphical approach, which reduces the whole problem to a computation by inspection. Our method is more general than the one developed in [58] because arbitrary topologies can be handled. The only restriction we impose on the network is that it should contain no cycles, i.e., it should be of the feed-forward type. The method is of interest when we do not want to derive analytically the Hessian matrix each time the network topology changes.

Second-order derivatives

We investigate the case of second-order derivatives, that is, expressions of the form $\partial^2 F/\partial w_i \partial w_j$, where F is the network function as before and w_i and w_j are network's weights. We can think of each weight as a small potentiometer and we want to find out what happens to the network function when the resistance of both potentiometers is varied.

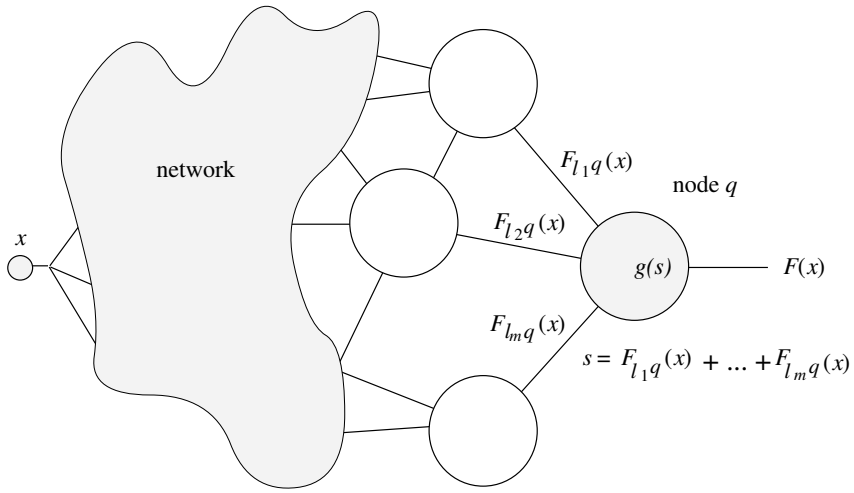

Fig. 8.19. Second-order computation

Figure 8.19 shows the general case. Let us assume, without loss of generality, that the input to the network is the one-dimensional value x. The network function F is computed at the output node with label q (shown shaded) for the given input value. We can also think of the inputs to the output node as network functions computed by subnetworks of the original network. Let us call these functions $F_{\ell_1 q}, F_{\ell_2 q}, \ldots, F_{\ell_m q}$. If the one-dimensional function at the output node is g, the network function is the composition

$$F(x) = g(F_{\ell_1 q}(x) + F_{\ell_2 q}(x) + \ldots + F_{\ell_m q}(x)).$$

We are interested in computing $\partial^2 F(x)/\partial w_i \partial w_j$ for two given network weights w_i and w_j. Simple differential calculus tells us that

$$\frac{\partial^2 F(x)}{\partial w_i \partial w_j} = g''(s)\frac{\partial s}{\partial w_i}\frac{\partial s}{\partial w_j} + g'(s)\left(\frac{\partial^2 F_{\ell_1 q}(x)}{\partial w_i \partial w_j} + \cdots + \frac{\partial^2 F_{\ell_m q}(x)}{\partial w_i \partial w_j}\right),$$

where $s = F_{\ell_1 q}(x) + F_{\ell_2 q}(x) + \ldots + F_{\ell_m q}(x)$. This means that the desired second-order partial derivative consists of two terms: the first is the second derivative of g evaluated at its input multiplied by the partial derivatives of the sum of the m subnetwork functions $F_{\ell_1 q}, \ldots, F_{\ell_m q}$, once with respect to w_i and once with respect to w_j. The second term is the first derivative of g multiplied by the sum of the second-order partial derivatives of each subnetwork function with respect to w_i and w_j. We call this term the *second-order correction*. The recursive structure of the problem is immediately obvious. We already have an algorithm to compute the first partial derivatives of any network function with

respect to a weight. We just need to use the above expression in a recursive manner to obtain the second-order derivatives we want.

We thus extend the feed-forward labeling phase of the backpropagation algorithm in the following manner: at each node which computes a one-dimensional function f we will store *three* values: $f(x)$, $f'(x)$ and $f''(x)$, where x represents the input to this node. When looking for the second-order derivatives we apply the recursive strategy given above. Figure 8.20 shows the main idea:

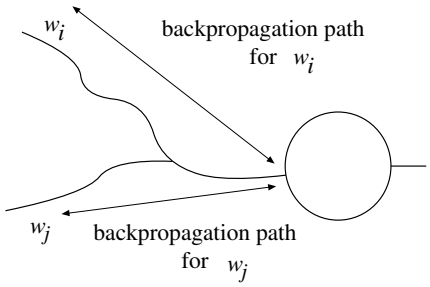

Fig. 8.20. Intersecting paths to a node

- Perform the feed-forward labeling step in the usual manner, but store additionally at each node the second derivative of the node's function evaluated at its input
- Select two weights w_i and w_j and an output node whose associated network function we want to derive. The second-order partial derivative of the network function with respect to these weights is the product of the stored g'' value with the backpropagation path value from the output node up to weight w_i and with the backpropagation path value from the output node up to weight w_j. If the backpropagation paths for w_i and w_j intersect, a second-order correction is needed which is equal to the stored value of g' multiplied by the sum of the second-order derivative with respect to w_i and w_j of all subnetwork function inputs to the node which belong to intersecting paths.

This looks like an intricate rule, but it is again the chain rule for second-order derivatives expressed in a recursive manner. Consider the multilayer perceptron shown in Figure 8.21. A weight w_{ih} in the first layer of weights and a weight w_{jm} in the second layer can only interact at the output node m. The second derivative of F_m with respect to w_{ih} and w_{jm} is just the stored value f'' multiplied by the stored output of the hidden unit j and the backpropagation path up to w_{ih}, that is, $w_{hm}h'x_i$. Since the backpropagation paths for w_{ih} and w_{jm} do not intersect, this is the required expression. This is also the expression found analytically by Bishop [1993].

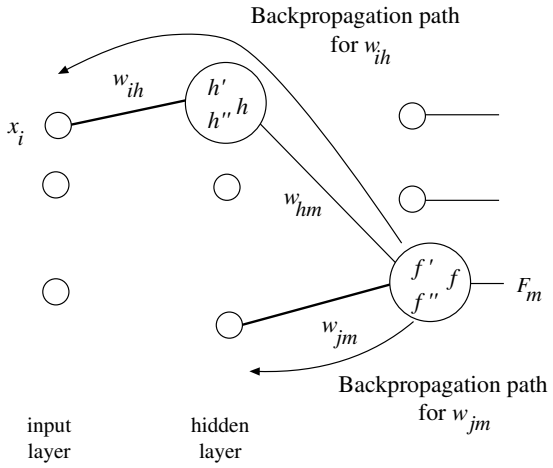

Fig. 8.21. Multilayer perceptron

In the case where one weight lies in the backpropagation path of another, a simple adjustment has to be made. Let us assume that weight w_{ik} lies in the backpropagation path of weight w_j. The second-order backpropagation algorithm is performed as usual and the backward computation proceeds up to the point where weight w_{ik} transports an input to a node k for which a second-order correction is needed. Figure 8.22 shows the situation. The information transported through the edge with weight w_{ik} is the subnetwork function F_{ik}. The second-order correction for the node with primitive function g is

$$g' \frac{\partial^2 F_{ik}}{\partial w_{ik} \partial wj} = g' \frac{\partial^2 w_{ik} F_i}{\partial w_{ik} \partial wj},$$

but this is simply

$$g' \frac{\partial F_i}{\partial w_j},$$

since the subnetwork function F_i does not depend on w_{ik}. Thus, the second-order backpropagation method must be complemented by the following rule:

- If the second-order correction to a node k with activation function g involves a weight w_{ik} (that is, a weight directly affecting node k) and a node w_j, the second-order correction is just g' multiplied by the backpropagation path value of the subnetwork function F_i with respect to w_j.

Explicit calculation of the Hessian

For the benefit of the reader, we put together all the pieces of what we call second-order backpropagation in this section. We consider the case of a single input pattern into the network, since the more general case is easy to handle.

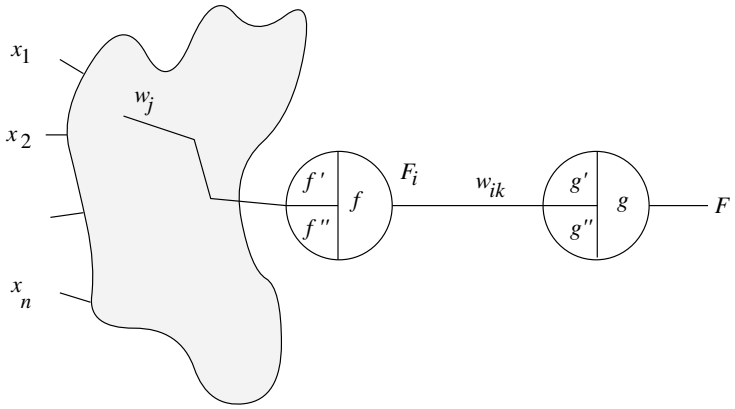

Fig. 8.22. The special case

Algorithm 8.4.1 *Second-order backpropagation*

i) Extend the neural network by adding nodes which compute the squared difference of each component of the output and the expected target values. Collect all these differences at a single node whose output is the error function of the network. The activation function of this node is the identity.

ii) Label all nodes in the feed-forward phase with the result of computing $f(x)$, $f'(x)$, and $f''(x)$, where x represents the global input to each node and f its associated activation function.

iii) Starting from the error function node in the extended network, compute the second-order derivative of E with respect to two weights w_i and w_j, by proceeding recursively in the following way:

iii.1) The second-order derivative of the output of a node G with activation function g with respect to two weights w_i and w_j is the product of the stored g'' value with the backpropagation path values between w_i and the node G and between w_j and the node G. A second-order correction is needed if both backpropagation paths intersect.

iii.2) The second-order correction is equal to the product of the stored g' value with the sum of the second-order derivative (with respect to w_i and w_j) of each node whose output goes directly to G and which belongs to the intersection of the backpropagation paths of w_i and w_j.

iii.3) In the special case that one of the weights, for example, w_i, connects node h directly to node G, the second-order correction is just g' multiplied by the backpropagation path value of the subnetwork function F_h with respect to w_j.

Example of second-order backpropagation

Consider the network shown in Figure 8.23, commonly used to compute the XOR function. The left node is labeled 1, the right node 2. The input values x and y are kept fixed and we are interested in the second-order partial derivative of the network function $F_2(x, y)$ with respect to the weights w_1 and w_2.

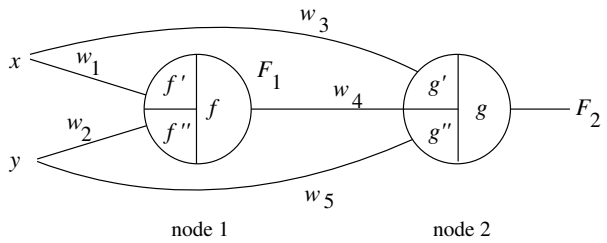

Fig. 8.23. A two unit network

By mere inspection and using the recursive method mentioned above, we see that the first term of $\partial^2 F_2/\partial w_1 \partial w_2$ is the expression

$$g''(w_3 x + w_5 y + w_4 f(w_1 x + w_2 y))(w_4 f'(w_1 x + w_2 y)x)(w_4 f'(w_1 x + w_2 y)y).$$

In this expression $(w_4 f'(w_1 x + w_2 y)x)$ is the backpropagation path value from the output of the node which computes the function f, including multiplication by the weight w_4 (that is the subnetwork function $w_4 F_1$), up to the weight w_1. The term $(w_4 f'(w_1 x + w_2 y)y)$ is the result of backpropagation for $w_4 F_1$ up to w_2. The second-order correction needed for computation of $\partial^2 F_2/\partial w_1 \partial w_2$ is

$$g'(w_3 x + w_5 y + w_4 f(w_1 x + w_2 y))\frac{\partial^2 w_4 F_1}{\partial w_1 \partial w_2}.$$

Since it is obvious that

$$\frac{\partial^2 w_4 F_1}{\partial w_1 \partial w_2} = w_4 \frac{\partial^2 F_1}{\partial w_1 \partial w_2} = w_4 f''(w_1 x + w_2 y)xy$$

we finally get

$$\begin{aligned}\frac{\partial^2 F_2}{\partial w_1 \partial w_2} &= g''(w_3 x + w_5 y + w_4 f(w_1 x + w_2 y)) \\ &\quad \times (w_4 f'(w_1 x + w_2 y)x)(w_4 f'(w_1 x + w_2 y)y) \\ &\quad + g'(w_3 x + w_5 y + w_4 f(w_1 x + w_2 y))w_4 f''(w_1 x + w_2 y)xy.\end{aligned}$$

The reader can *visually* check the following expression:

$$\frac{\partial^2 F_2}{\partial w_1 \partial w_5} = g''(w_3 x + w_5 y + w_4 f(w_1 x + w_2 y))(w_4 f'(w_1 x + w_2 y)x)y.$$

In this case no second-order correction is needed, since the backpropagation paths up to w_1 and w_5 do not intersect.

A final example is the calculation of the whole Hessian matrix for the network shown above (Figure 8.23). We omit the error function expansion and compute the Hessian of the network function F_2 with respect to the network's five weights. The labelings of the nodes are f, f', and f'' computed over the input $w_1x + w_2y$, and g, g', g'' computed over the input $w_4f(w_1x + w_2y) + w_3x + w_5y$. Under these assumptions the components of the upper triangular part of the Hessian are the following:

$$
\begin{aligned}
H_{11} &= g''w_4^2f'^2x^2 + g'w_4f''x^2 \\
H_{22} &= g''w_4^2f'^2y^2 + g'w_4f''y^2 \\
H_{33} &= g''x^2 \\
H_{44} &= g''f^2 \\
H_{55} &= g''y^2 \\
H_{12} &= g''w_4^2f'^2xy + g'w_4f''xy \\
H_{13} &= g''w_4f'x^2 \\
H_{14} &= g''w_4f'xf + g'f'x \\
H_{15} &= g''w_4f'xy \\
H_{23} &= g''w_4f'yx \\
H_{24} &= g''w_4f'yf + g'f'y \\
H_{25} &= g''w_4f'y^2 \\
H_{34} &= g''xf \\
H_{35} &= g''xy \\
H_{45} &= g''yf
\end{aligned}
$$

All these results were obtained by simple inspection of the network shown in Figure 8.23. Note that the method is totally general in the sense that each node can compute a different activation function.

Some conclusions

With some experience it is easy to compute the Hessian matrix even for convoluted feed-forward topologies. This can be done either symbolically or numerically. The importance of this result is that once the recursive strategy has been defined it is easy to implement in a computer. It is the same kind of difference as the one existing between the chain rule and the backpropagation algorithm. The first one gives us the same result as the second, but backpropagation tries to organize the data in such a way that redundant computations are avoided. This can also be done for the method described here. Calculation of the Hessian matrix involves repeated computation of the same terms. In

this case the network itself provides us with a data structure in which we can store partial results and with which we can organize the computation. This explains why standard and second-order backpropagation are also of interest for computer algebra systems. It is not very difficult to program the method described here in a way that minimizes the number of arithmetic operations needed. The key observation is that the backpropagation path values can be stored to be used repetitively and that the nodes in which the backpropagation paths of different weights intersect need to be calculated only once. It is then possible to optimize computation of the Hessian using graph traversing algorithms.

A final observation is that computing the diagonal of the Hessian matrix involves only local communication in a neural network. Since the backpropagation path to a weight intersects itself in its whole length, computation of the second partial derivative of the associated network function of an output unit with respect to a given weight can be organized as a recursive backward computation over this path. Pseudo-Newton methods [48] can profit from this computational locality.

8.5 Relaxation methods

The class of relaxation methods includes all those techniques in which the network weights are perturbed and the new network error is compared directly to the previous one. Depending on their relative magnitudes a decision is taken regarding the subsequent iteration steps.

8.5.1 Weight and node perturbation

Weight perturbation is a learning strategy which has been proposed to avoid calculating the gradient of the error function at each step. A discrete approximation to the gradient is made at each iteration by taking the initial weight vector $\mathbf{w}$ in weight space and the value $E(w)$ of the error function for this combination of parameters. A small perturbation β is added to the weight w_i. The error $E(\mathbf{w}')$ at the new point $\mathbf{w}'$ in weight space is computed and the weight w_i is updated using the increment

$$\Delta w_i = -\gamma \frac{E(w') - E(w)}{\beta}.$$

This step is repeated iteratively, randomly selecting the weight to be updated. The discrete approximation to the gradient is especially important for VLSI chips in which the learning algorithm is implemented with minimal hardware additional to that needed for the feed-forward phase [216].

Another alternative which can provide faster convergence is perturbing not a weight, but the output o_i of the i-th node by Δo_i. The difference $E - E'$ in

the error function is computed and if it is positive, the new error E' could be achieved with the output $o_i + \Delta o_i$ for the i-th node. If the activation function is a sigmoid, the desired weighted input to node i is $\sum_{k=1}^{m} w'_k x_k = s^{-1}(o_i + \Delta o_i)$. If the previous weighted input was $\sum_{k=1}^{m} w_k x_k$, then the new weights are given by

$$w'_k = w_k \frac{s^{-1}(o_i + \Delta o_i)}{\sum_{k=1}^{m} w_k x_k} \qquad \text{for } k = 1, \ldots, m.$$

The weights are updated in proportion to their relative size. To avoid always keeping the same proportions, a stochastic factor can be introduced or a node perturbation step can be alternated with a weight perturbation step.

8.5.2 Symmetric and asymmetric relaxation

According to the analysis we made in Sect. 7.3.3 of two layered networks trained with backpropagation, the backpropagated error up to the output layer can be written as

$$\delta^{(2)} = \mathbf{D}_2 \mathbf{e},$$

where $\mathbf{e}$ is the column vector whose components are the derivatives of the corresponding components of the quadratic error, and $\mathbf{D}_2$ is a diagonal matrix as defined in Chap. 7. We can try to reduce the magnitude of the error $\mathbf{e}$ to zero by adjusting the matrix $\mathbf{W}_2$ in a single step. Since the desired target vector is $\mathbf{t}$, the necessary weighted input at the output nodes is $s^{-1}(\mathbf{t})$. This means that we want the equation

$$\mathbf{o}^{(1)} \mathbf{W}_2 = s^{-1}(\mathbf{t})$$

to hold for all the p possible input patterns. If we arrange all vectors $\mathbf{o}^{(1)}$ as the rows of a $p \times k$ matrix $\mathbf{O}_1$ and all targets as the rows of a $p \times m$ matrix $\mathbf{T}$, we are looking for the matrix $\mathbf{W}_2$ for which

$$\mathbf{O}_1 \mathbf{W}_2 = s^{-1}(\mathbf{T}) \tag{8.12}$$

holds. In general this matrix equation may have no solution for $\mathbf{W}_2$, but we can compute the matrix which minimizes the quadratic error for this equality. We show in Sects. 9.2.4 and **??** that if $\mathbf{O}_1^+$ is the so-called *pseudoinverse* of $\mathbf{O}_1$, then $\mathbf{W}_2$ is given by

$$\mathbf{W}_2 = \mathbf{O}_1^+ s^{-1}(\mathbf{T}).$$

After computing $\mathbf{W}_2$ we can ask what is the matrix $\mathbf{O}_1$ which minimizes the quadratic deviation from equality in (8.12). We compute intermediate targets for the hidden units, which are given by the rows of the matrix

$$\mathbf{O}'_1 = s^{(-1)}(\mathbf{T}) \mathbf{W}_2^+.$$

The new intermediate targets can now be used to obtain an update for the matrix $\mathbf{W}_1$ of weights between input sites and hidden units. The pseudoinverse can be computed with the method discussed in Sect. 9.2.4.

Note that this method is computationally intensive for every "pseudoinverse" step. The weight corrections are certainly much more accurate than in other algorithms but these high-powered iterations demand many computations for each of the two matrices. If the error function can be approximated nicely with a quadratic function, the algorithm converges very fast. If the input data is highly redundant, then the pseudoinverse step can be very inefficient when compared to on-line backpropagation, for example.

The algorithm described is an example of *symmetric* relaxation, since both the targets **T** and the outputs of the hidden units are determined in a back and forth kind of approach.

8.5.3 A final thought on taxonomy

The contents of this chapter can be summarized using Figure 8.24. The fast variations of backpropagation have been divided into two columns: gradient descent and relaxation methods. Algorithms in the first column use information about the error function's partial derivatives. Algorithms in the second column try to adjust the weights to fit the problem in a stochastic way or solving a linear subproblem.

The three rows in Figure 8.24 show the kinds of derivative used. First-order algorithms work with the first partial derivatives, second-order algorithms with the second partial derivatives. In between we have adaptive first-order methods that from first-order information extract an approximation to parts of the Hessian matrix.

Standard backpropagation is a first-order gradient descent method. However, since on-line backpropagation does not exactly follow the gradient's direction, it also partially qualifies as a relaxation method. The adaptive step methods (DBD, Rprop, etc.) are also a combination of gradient descent and relaxation. Quickprop is an algorithm which approximates second-order information but which updates the weights separately using a kind of relaxation approach.

The adaptive first-order methods can be also divided in two groups: in the first a single global learning rate is used, in the second there is a learning rate for each weight. The conjugate gradient methods of numerical analysis rely also on a first-order approximation of second-order features of the error function.

8.6 Historical and bibliographical remarks

This survey of fast learning algorithms is by no means complete. We have disregarded constructive algorithms of the type reviewed in Chap. 14. We have only given a quick overview of some of the main ideas that have been developed in the last decade. There is a large amount of literature on nonlinear optimization that should be reviewed by anyone wishing to improve current

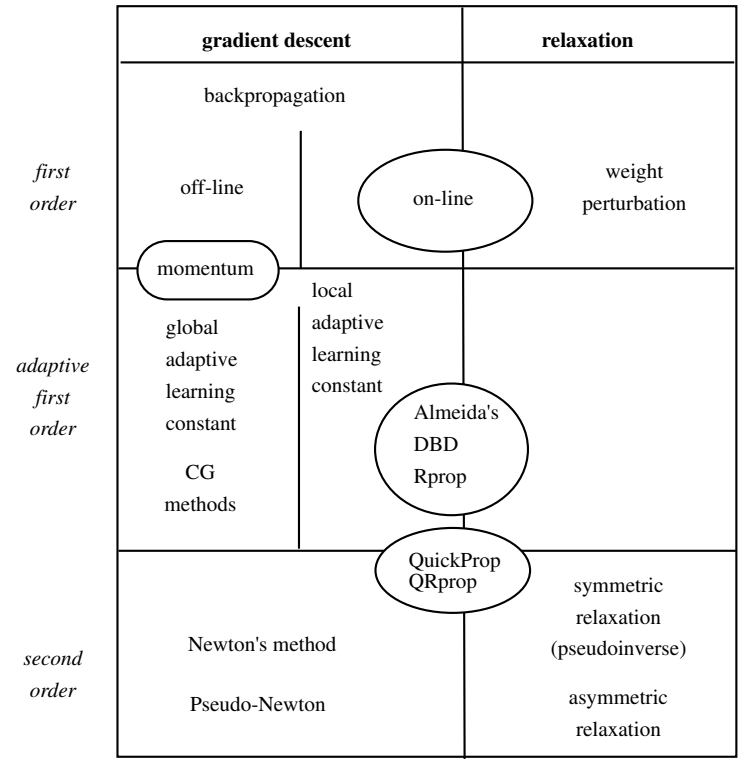

Fig. 8.24. Taxonomy of learning algorithms

learning methods for multilayer networks. Classical CG algorithms and some variants developed specially for neural networks are interesting in this respect [315].

A different approach has been followed by Karayiannis and Venetsanopoulos who use a type of what is called a continuation method [235]. The network is trained to minimize a given measure of error, which is iteratively changed during the computation. One can start solving a linear regression problem and introduce the nonlinearities slowly. The method could possibly be combined with some of the other fast learning methods.

A factor which has traditionally hampered direct comparisons of learning algorithms is the wide variety of benchmarks used. Only in a few cases have large learning problems taken from public domain databases been used. Some efforts to build a more comprehensive set of benchmarks have been announced.

Exercises

1. Prove that the value of the learning constant given by equation (8.5) in fact produces exact learning of the j input pattern to a linear associator. Derive the expression for ℓ^2.
2. Show how to construct the linear transformation that maps an ellipsoidal data distribution to a sphere. Assume that the training set consists of N points (Sect. 8.2.4).
3. Compare Rprop and Quickprop using a small benchmark (for example the 8-bit parity problem).
4. Train a neural network using: a) Backpropagation and a piecewise linear approximation of the sigmoid; b) A table of values of the sigmoid and its derivative.