12 Variations on Backpropagation

Objectives

The backpropagation algorithm introduced in Chapter 11 was a major breakthrough in neural network research. However, the basic algorithm is too slow for most practical applications. In this chapter we present several variations of backpropagation that provide significant speedup and make the algorithm more practical.

We will begin by using a function approximation example to illustrate why the backpropagation algorithm is slow in converging. Then we will present several modifications to the algorithm. Recall that backpropagation is an approximate steepest descent algorithm. In Chapter 9 we saw that steepest descent is the simplest, and often the slowest, minimization method. The conjugate gradient algorithm and Newton's method generally provide faster convergence. In this chapter we will explain how these faster procedures can be used to speed up the convergence of backpropagation.

Theory and Examples

When the basic backpropagation algorithm is applied to a practical problem the training may take days or weeks of computer time. This has encouraged considerable research on methods to accelerate the convergence of the algorithm.

The research on faster algorithms falls roughly into two categories. The first category involves the development of heuristic techniques, which arise out of a study of the distinctive performance of the standard backpropagation algorithm. These heuristic techniques include such ideas as varying the learning rate, using momentum and rescaling variables (e.g., [VoMa88], [Jacob88], [Toll90] and [RiIr90]). In this chapter we will discuss the use of momentum and variable learning rates.

Another category of research has focused on standard numerical optimization techniques (e.g., [Shan90], [Barn92], [Batt92] and [Char92]). As we have discussed in Chapters 10 and 11, training feedforward neural networks to minimize squared error is simply a numerical optimization problem. Because numerical optimization has been an important research subject for 30 or 40 years (see Chapter 9), it seems reasonable to look for fast training algorithms in the large number of existing numerical optimization techniques. There is no need to "reinvent the wheel" unless absolutely necessary. In this chapter we will present two existing numerical optimization techniques that have been very successfully applied to the training of multilayer perceptrons: the conjugate gradient algorithm and the Levenberg-Marquardt algorithm (a variation of Newton's method).

We should emphasize that all of the algorithms that we will describe in this chapter use the backpropagation procedure, in which derivatives are processed from the last layer of the network to the first. For this reason they could all be called "backpropagation" algorithms. The differences between the algorithms occur in the way in which the resulting derivatives are used to update the weights. In some ways it is unfortunate that the algorithm we usually refer to as backpropagation is in fact a steepest descent algorithm. In order to clarify our discussion, for the remainder of this chapter we will refer to the basic backpropagation algorithm as steepest descent backpropagation (*SDBP*).

SDBP

In the next section we will use a simple example to explain why SDBP has problems with convergence. Then, in the following sections, we will present various procedures to improve the convergence of the algorithm.

Drawbacks of Backpropagation

Recall from Chapter 10 that the LMS algorithm is guaranteed to converge to a solution that minimizes the mean squared error, so long as the learning rate is not too large. This is true because the mean squared error for a single-layer linear network is a quadratic function. The quadratic function has only a single stationary point. In addition, the Hessian matrix of a quadratic function is constant, therefore the curvature of the function in a given direction does not change, and the function contours are elliptical.

SDBP is a generalization of the LMS algorithm. Like LMS, it is also an approximate steepest descent algorithm for minimizing the mean squared error. In fact, SDBP is equivalent to the LMS algorithm when used on a single-layer linear network. (See Problem P11.10.) When applied to multilayer networks, however, the characteristics of SDBP are quite different. This has to do with the differences between the mean squared error performance surfaces of single-layer linear networks and multilayer nonlinear networks. While the performance surface for a single-layer linear network has a single minimum point and constant curvature, the performance surface for a multilayer network may have many local minimum points, and the curvature can vary widely in different regions of the parameter space. This will become clear in the example that follows.

Performance Surface Example

To investigate the mean squared error performance surface for multilayer networks we will employ a simple function approximation example. We will use the 1-2-1 network shown in Figure 12.1, with log-sigmoid transfer functions in both layers.

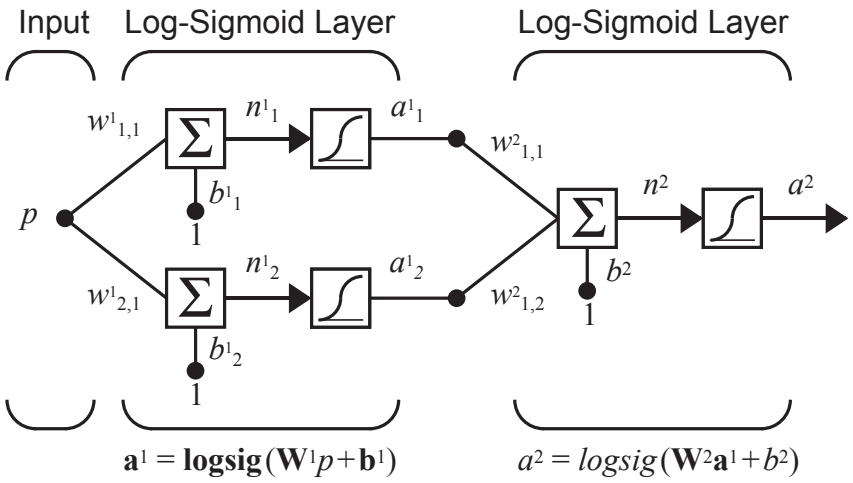


Figure 12.1 1-2-1 Function Approximation Network

In order to simplify our analysis, we will give the network a problem for which we know the optimal solution. The function we will approximate is

the response of the same 1-2-1 network, with the following values for the weights and biases:

$$w^1_{1,1} = 10,\ w^1_{2,1} = 10,\ b^1_1 = -5,\ b^1_2 = 5, \tag{12.1}$$

$$w^2_{1,1} = 1,\ w^2_{1,2} = 1,\ b^2 = -1. \tag{12.2}$$

The network response for these parameters is shown in Figure 12.2, which plots the network output a^2 as the input p is varied over the range $[-2, 2]$.

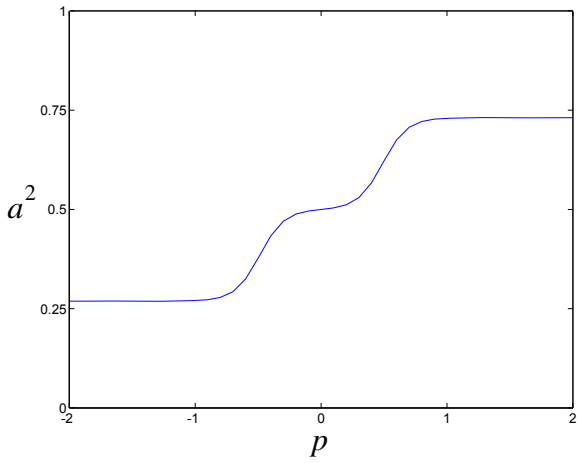


Figure 12.2 Nominal Function

We want to train the network of Figure 12.1 to approximate the function displayed in Figure 12.2. The approximation will be exact when the network parameters are set to the values given in Eq. (12.1) and Eq. (12.2). This is, of course, a very contrived problem, but it is simple and it illustrates some important concepts.

Let's now consider the performance index for our problem. We will assume that the function is sampled at the values

$$p = -2, -1.9, -1.8, \ldots, 1.9, 2, \tag{12.3}$$

and that each occurs with equal probability. The performance index will be the sum of the squared errors at these 41 points. (We won't bother to find the mean squared error, which just requires dividing by 41.)

In order to be able to graph the performance index, we will vary only two parameters at a time. Figure 12.3 illustrates the squared error when only $w^1_{1,1}$ and $w^2_{1,1}$ are being adjusted, while the other parameters are set to their optimal values given in Eq. (12.1) and Eq. (12.2). Note that the minimum error will be zero, and it will occur when $w^1_{1,1} = 10$ and $w^2_{1,1} = 1$, as indicated by the open blue circle in the figure.

There are several features to notice about this error surface. First, it is clearly not a quadratic function. The curvature varies drastically over the parameter space. For this reason it will be difficult to choose an appropriate learning rate for the steepest descent algorithm. In some regions the surface is very flat, which would allow a large learning rate, while in other regions the curvature is high, which would require a small learning rate. (Refer to discussions in Chapters 9 and 10 on the choice of learning rate for the steepest descent algorithm.)

It should be noted that the flat regions of the performance surface should not be unexpected, given the sigmoid transfer functions used by the network. The sigmoid is very flat for large inputs.

A second feature of this error surface is the existence of more than one local minimum point. The global minimum point is located at $w^1_{1,1} = 10$ and $w^2_{1,1} = 1$, along the valley that runs parallel to the $w^1_{1,1}$ axis. However, there is also a local minimum, which is located in the valley that runs parallel to the $w^2_{1,1}$ axis. (This local minimum is actually off the graph at $w^1_{1,1} = 0.88$, $w^2_{1,1} = 38.6$.) In the next section we will investigate the performance of backpropagation on this surface.

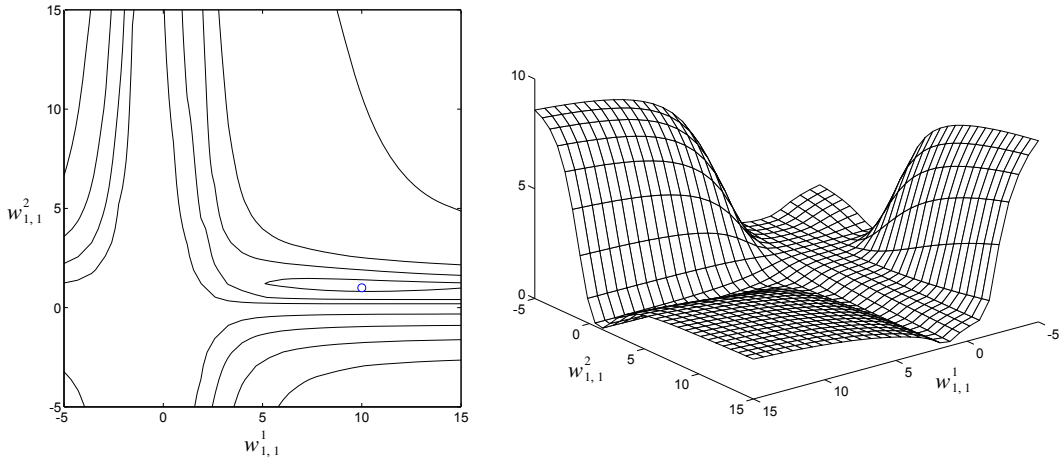


Figure 12.3 Squared Error Surface Versus $w^1_{1,1}$ and $w^2_{1,1}$

Figure 12.4 illustrates the squared error when $w^1_{1,1}$ and b^1_1 are being adjusted, while the other parameters are set to their optimal values. Note that the minimum error will be zero, and it will occur when $w^1_{1,1} = 10$ and $b^1_1 = -5$, as indicated by the open blue circle in the figure.

Again we find that the surface has a very contorted shape, steep in some regions and very flat in others. Surely the standard steepest descent algorithm will have some trouble with this surface. For example, if we have an initial guess of $w^1_{1,1} = 0$, $b^1_1 = -10$, the gradient will be very close to zero,

and the steepest descent algorithm would effectively stop, even though it is not close to a local minimum point.

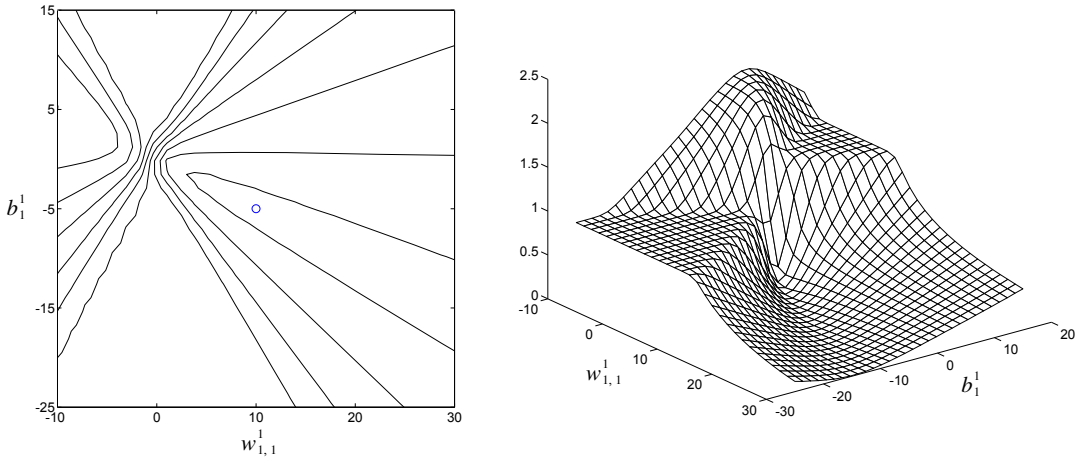


Figure 12.4 Squared Error Surface Versus $w_{1,1}^1$ and b_1^1

Figure 12.5 illustrates the squared error when b_1^1 and b_2^1 are being adjusted, while the other parameters are set to their optimal values. The minimum error is located at $b_1^1 = -5$ and $b_2^1 = 5$, as indicated by the open blue circle in the figure.

This surface illustrates an important property of multilayer networks: they have a symmetry to them. Here we see that there are two local minimum points and they both have the same value of squared error. The second solution corresponds to the same network being turned upside down (i.e., the top neuron in the first layer is exchanged with the bottom neuron). It is because of this characteristic of neural networks that we do not set the initial weights and biases to zero. The symmetry causes zero to be a saddle point of the performance surface.

This brief study of the performance surfaces for multilayer networks gives us some hints as to how to set the initial guess for the SDBP algorithm. First, we do not want to set the initial parameters to zero. This is because the origin of the parameter space tends to be a saddle point for the performance surface. Second, we do not want to set the initial parameters to large values. This is because the performance surface tends to have very flat regions as we move far away from the optimum point.

Typically we choose the initial weights and biases to be small random values. In this way we stay away from a possible saddle point at the origin without moving out to the very flat regions of the performance surface. (Another procedure for choosing the initial parameters is described in [NgWi90].) As we will see in the next section, it is also useful to try several different initial guesses, in order to be sure that the algorithm converges to a global minimum point.

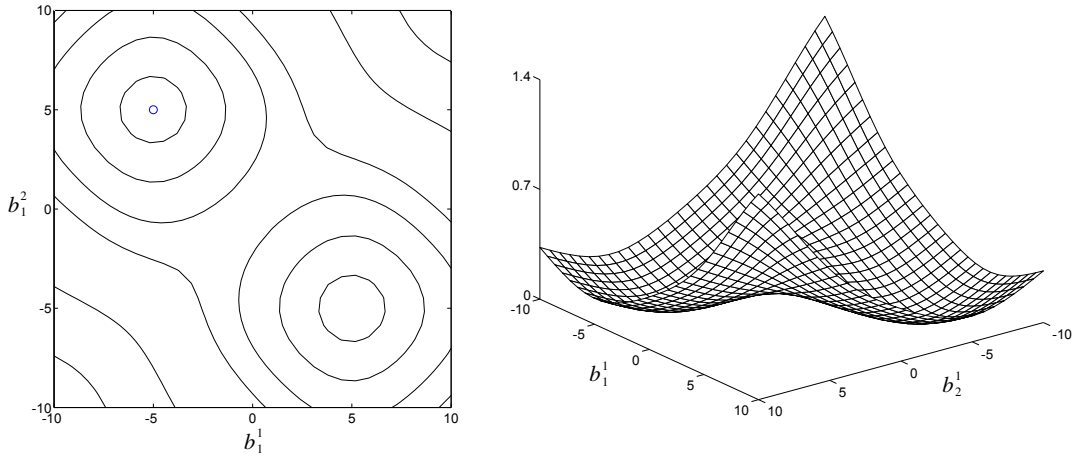


Figure 12.5 Squared Error Surface Versus b_1^1 and b_2^1

Convergence Example

Batching

Now that we have examined the performance surface, let's investigate the performance of SDBP. For this section we will use a variation of the standard algorithm, called *batching*, in which the parameters are updated only after the entire training set has been presented. The gradients calculated at each training example are averaged together to produce a more accurate estimate of the gradient. (If the training set is complete, i.e., covers all possible input/output pairs, then the gradient estimate will be exact.)

In Figure 12.6 we see two trajectories of SDBP (batch mode) when only two parameters, $w_{1,1}^1$ and $w_{1,1}^2$ are adjusted. For the initial condition labeled "a" the algorithm does eventually converge to the optimal solution, but the convergence is slow. The reason for the slow convergence is the change in curvature of the surface over the path of the trajectory. After an initial moderate slope, the trajectory passes over a very flat surface, until it falls into a very gently sloping valley. If we were to increase the learning rate, the algorithm would converge faster while passing over the initial flat surface, but would become unstable when falling into the valley, as we will see in a moment.

Trajectory "b" illustrates how the algorithm can converge to a local minimum point. The trajectory is trapped in a valley and diverges from the optimal solution. If allowed to continue the trajectory converges to $w_{1,1}^1 = 0.88$, $w_{1,1}^2 = 38.6$. The existence of multiple local minimum points is typical of the performance surface of multilayer networks. For this reason it is best to try several different initial guesses in order to ensure that a global minimum has been obtained. (Some of the local minimum points may have the same value of squared error, as we saw in Figure 12.5, so we would not expect the algorithm to converge to the same parameter values for each initial guess. We just want to be sure that the same minimum error is obtained.)

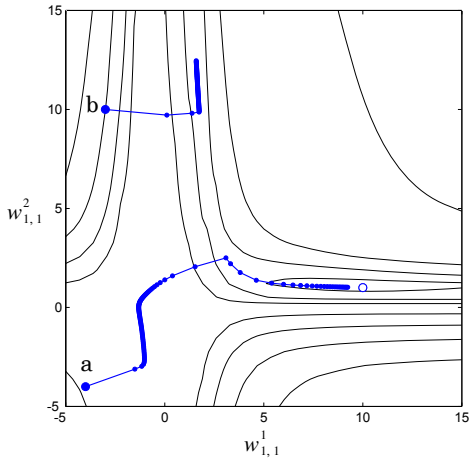


Figure 12.6 Two SDBP (Batch Mode) Trajectories

The progress of the algorithm can also be seen in Figure 12.7, which shows the squared error versus the iteration number. The curve on the left corresponds to trajectory “a” and the curve on the right corresponds to trajectory “b.” These curves are typical of SDBP, with long periods of little progress and then short periods of rapid advance.

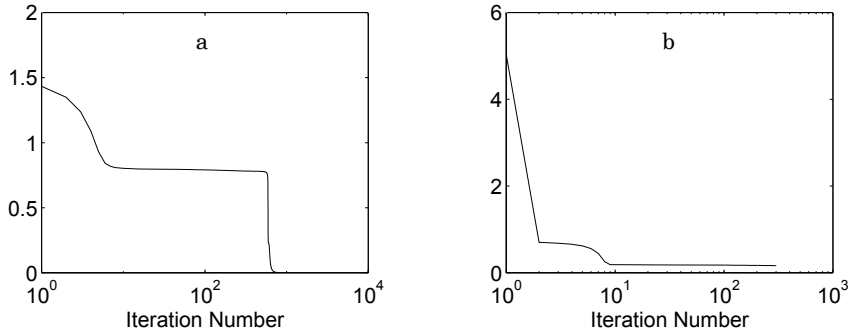


Figure 12.7 Squared Error Convergence Patterns

We can see that the flat sections in Figure 12.7 correspond to times when the algorithm is traversing a flat section of the performance surface, as shown in Figure 12.6. During these periods we would like to increase the learning rate, in order to speed up convergence. However, if we increase the learning rate the algorithm will become unstable when it reaches steeper portions of the performance surface.

This effect is illustrated in Figure 12.8. The trajectory shown here corresponds to trajectory “a” in Figure 12.6, except that a larger learning rate was used. The algorithm converges faster at first, but when the trajectory reaches the narrow valley that contains the minimum point the algorithm begins to diverge. This suggests that it would be useful to vary the learning rate. We could increase the learning rate on flat surfaces and then decrease the learning rate as the slope increased. The question is: “How will the al-

gorithm know when it is on a flat surface?" We will discuss this in a later section.

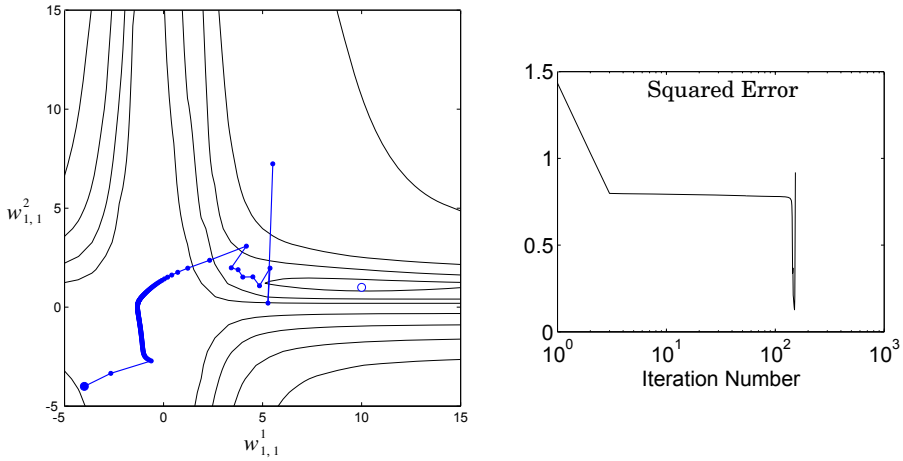
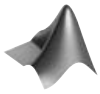


Figure 12.8 Trajectory with Learning Rate Too Large

Another way to improve convergence would be to smooth out the trajectory. Note in Figure 12.8 that when the algorithm begins to diverge it is oscillating back and forth across a narrow valley. If we could filter the trajectory, by averaging the updates to the parameters, this might smooth out the oscillations and produce a stable trajectory. We will discuss this procedure in the next section.



To experiment with this backpropagation example, use the MATLAB® Neural Network Design Demonstration Steepest Descent Backpropagation (**nnd12sd**).

Heuristic Modifications of Backpropagation

Now that we have investigated some of the drawbacks of backpropagation (steepest descent), let's consider some procedures for improving the algorithm. In this section we will discuss two heuristic methods. In a later section we will present two methods based on standard numerical optimization algorithms.

Momentum

The first method we will discuss is the use of momentum. This is a modification based on our observation in the last section that convergence might be improved if we could smooth out the oscillations in the trajectory. We can do this with a low-pass filter.

Before we apply momentum to a neural network application, let's investigate a simple example to illustrate the smoothing effect. Consider the following first-order filter:

$$y(k) = \gamma y(k-1) + (1-\gamma)w(k), \tag{12.4}$$

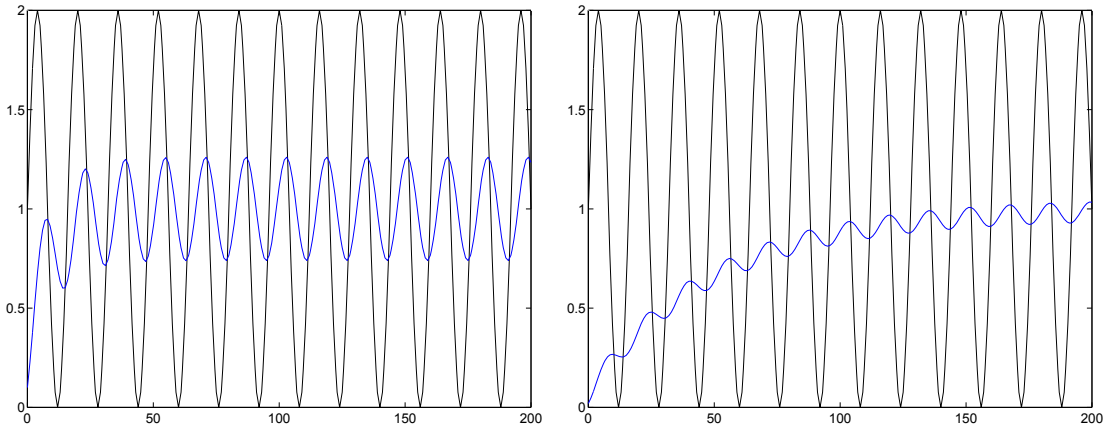
where $w(k)$ is the input to the filter, $y(k)$ is the output of the filter and γ is the momentum coefficient that must satisfy

$$0 \le \gamma < 1. \tag{12.5}$$

The effect of this filter is shown in Figure 12.9. For these examples the input to the filter was taken to be the sine wave:

$$w(k) = 1 + \sin\left(\frac{2\pi k}{16}\right), \tag{12.6}$$

and the momentum coefficient was set to $\gamma = 0.9$ (left graph) and $\gamma = 0.98$ (right graph). Here we can see that the oscillation of the filter output is less than the oscillation in the filter input (as we would expect for a low-pass filter). In addition, as γ is increased the oscillation in the filter output is reduced. Notice also that the average filter output is the same as the average filter input, although as γ is increased the filter output is slower to respond.



a) $\gamma = 0.9$ b) $\gamma = 0.98$

Figure 12.9 Smoothing Effect of Momentum

To summarize, the filter tends to reduce the amount of oscillation, while still tracking the average value. Now let's see how this works on the neural network problem. First, recall that the parameter updates for SDBP (Eq. (11.46) and Eq. (11.47)) are

$$\Delta \mathbf{W}^m(k) = -\alpha \mathbf{s}^m (\mathbf{a}^{m-1})^T, \tag{12.7}$$

$$\Delta \mathbf{b}^m(k) = -\alpha \mathbf{s}^m. \tag{12.8}$$

Momentum

MOBP

When the *momentum* filter is added to the parameter changes, we obtain the following equations for the momentum modification to backpropagation (*MOBP*):

$$\Delta \mathbf{W}^m(k) = \gamma \Delta \mathbf{W}^m(k-1) - (1-\gamma)\alpha \mathbf{s}^m (\mathbf{a}^{m-1})^T, \tag{12.9}$$

$$\Delta \mathbf{b}^m(k) = \gamma \Delta \mathbf{b}^m(k-1) - (1-\gamma)\alpha \mathbf{s}^m. \tag{12.10}$$

If we now apply these modified equations to the example in the preceding section, we obtain the results shown in Figure 12.10. (For this example we have used a batching form of MOBP, in which the parameters are updated only after the entire training set has been presented. The gradients calculated at each training example are averaged together to produce a more accurate estimate of the gradient.) This trajectory corresponds to the same initial condition and learning rate shown in Figure 12.8, but with a momentum coefficient of $\gamma = 0.8$. We can see that the algorithm is now stable. By the use of momentum we have been able to use a larger learning rate, while maintaining the stability of the algorithm. Another feature of momentum is that it tends to accelerate convergence when the trajectory is moving in a consistent direction.

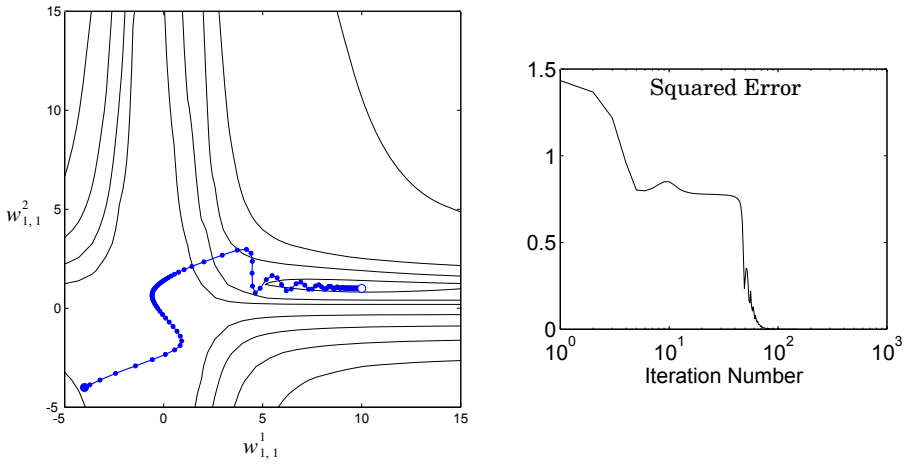


Figure 12.10 Trajectory with Momentum

If you look carefully at the trajectory in Figure 12.10, you can see why the procedure is given the name *momentum*. It tends to make the trajectory continue in the same direction. The larger the value of γ, the more "momentum" the trajectory has.

To experiment with momentum, use the MATLAB® Neural Network Design Demonstration Momentum Backpropagation (`nnd12mo`).

Variable Learning Rate

We suggested earlier in this chapter that we might be able to speed up convergence if we increase the learning rate on flat surfaces and then decrease the learning rate when the slope increases. In this section we want to explore this concept.

Recall that the mean squared error performance surface for single-layer linear networks is always a quadratic function, and the Hessian matrix is therefore constant. The maximum stable learning rate for the steepest descent algorithm is two divided by the maximum eigenvalue of the Hessian matrix. (See Eq. (9.25).)

As we have seen, the error surface for the multilayer network is not a quadratic function. The shape of the surface can be very different in different regions of the parameter space. Perhaps we can speed up convergence by adjusting the learning rate during the course of training. The trick will be to determine when to change the learning rate and by how much.

Variable Learning Rate
VLBP

There are many different approaches for varying the learning rate. We will describe a very straightforward batching procedure [VoMa88], where the learning rate is varied according to the performance of the algorithm. The rules of the *variable learning rate* backpropagation algorithm (*VLBP*) are:

1. If the squared error (over the entire training set) increases by more than some set percentage ζ (typically one to five percent) after a weight update, then the weight update is discarded, the learning rate is multiplied by some factor $0 < \rho < 1$, and the momentum coefficient γ (if it is used) is set to zero.

2. If the squared error decreases after a weight update, then the weight update is accepted and the learning rate is multiplied by some factor $\eta > 1$. If γ has been previously set to zero, it is reset to its original value.

3. If the squared error increases by less than ζ, then the weight update is accepted but the learning rate is unchanged. If γ has been previously set to zero, it is reset to its original value.

(See Problem P12.3 for a numerical example of VLBP.)

To illustrate VLBP, let's apply it to the function approximation problem of the previous section. Figure 12.11 displays the trajectory for the algorithm using the same initial guess, initial learning rate and momentum coefficient as was used in Figure 12.10. The new parameters were assigned the values

$$\eta = 1.05, \ \rho = 0.7 \text{ and } \zeta = 4\%. \tag{12.11}$$

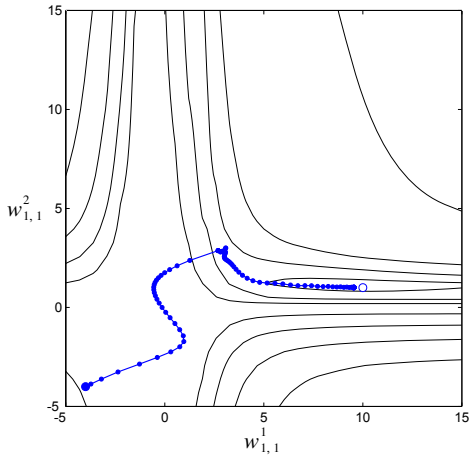


Figure 12.11 Variable Learning Rate Trajectory

Notice how the learning rate, and therefore the step size, tends to increase when the trajectory is traveling in a straight line with constantly decreasing error. This effect can also be seen in Figure 12.12, which shows the squared error and the learning rate versus iteration number.

When the trajectory reaches a narrow valley, the learning rate is rapidly decreased. Otherwise the trajectory would have become oscillatory, and the error would have increased dramatically. For each potential step where the error would have increased by more than 4% the learning rate is reduced and the momentum is eliminated, which allows the trajectory to make the quick turn to follow the valley toward the minimum point. The learning rate then increases again, which accelerates the convergence. The learning rate is reduced again when the trajectory overshoots the minimum point when the algorithm has almost converged. This process is typical of a VLBP trajectory.

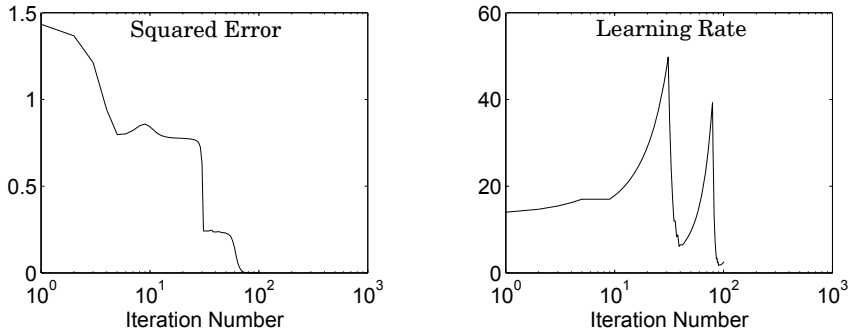


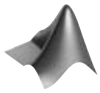
Figure 12.12 Convergence Characteristics of Variable Learning Rate

There are many variations on this variable learning rate algorithm. Jacobs [Jaco88] proposed the *delta-bar-delta* learning rule, in which each network parameter (weight or bias) has its own learning rate. The algorithm increases the learning rate for a network parameter if the parameter change

has been in the same direction for several iterations. If the direction of the parameter change alternates, then the learning rate is reduced. The *SuperSAB* algorithm of Tollenaere [Toll90] is similar to the delta-bar-delta rule, but it has more complex rules for adjusting the learning rates.

Another heuristic modification to SDBP is the Quickprop algorithm of Fahlman [Fahl88]. It assumes that the error surface is parabolic and concave upward around the minimum point and that the effect of each weight can be considered independently.

The heuristic modifications to SDBP can often provide much faster convergence for some problems. However, there are two main drawbacks to these methods. The first is that the modifications require that several parameters be set (e.g., ζ, ρ and γ), while the only parameter required for SDBP is the learning rate. Some of the more complex heuristic modifications can have five or six parameters to be selected. Often the performance of the algorithm is sensitive to changes in these parameters. The choice of parameters is also problem dependent. The second drawback to these modifications to SDBP is that they can sometimes fail to converge on problems for which SDBP will eventually find a solution. Both of these drawbacks tend to occur more often when using the more complex algorithms.



To experiment with VLBP, use the MATLAB® Neural Network Design Demonstration Variable Learning Rate Backpropagation (`nnd12vl`).

Numerical Optimization Techniques

Now that we have investigated some of the heuristic modifications to SDBP, let's consider those methods that are based on standard numerical optimization techniques. We will investigate two techniques: conjugate gradient and Levenberg-Marquardt. The conjugate gradient algorithm for quadratic functions was presented in Chapter 9. We need to add two procedures to this algorithm in order to apply it to more general functions.

The second numerical optimization method we will discuss in this chapter is the Levenberg-Marquardt algorithm, which is a modification to Newton's method that is well-suited to neural network training.

Conjugate Gradient

In Chapter 9 we presented three numerical optimization techniques: steepest descent, conjugate gradient and Newton's method. Steepest descent is the simplest algorithm, but is often slow in converging. Newton's method is much faster, but requires that the Hessian matrix and its inverse be calculated. The conjugate gradient algorithm is something of a compromise; it does not require the calculation of second derivatives, and yet it still has the quadratic convergence property. (It converges to the minimum of a quadratic function in a finite number of iterations.) In this section we will describe how the conjugate gradient algorithm can be used to train

multilayer networks. We will call this algorithm *conjugate gradient backpropagation* (*CGBP*).

CGBP

Let's begin by reviewing the conjugate gradient algorithm. For ease of reference, we will repeat the algorithm steps from Chapter 9 (page 9-18):

1. Select the first search direction $\mathbf{p}_0$ to be the negative of the gradient, as in Eq. (9.59):

$$\mathbf{p}_0 = -\mathbf{g}_0, \tag{12.12}$$

where

$$\mathbf{g}_k \equiv \nabla F(\mathbf{x})\big|_{\mathbf{x}=\mathbf{x}_k}. \tag{12.13}$$

2. Take a step according to Eq. (9.57), selecting the learning rate α_k to minimize the function along the search direction:

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k. \tag{12.14}$$

3. Select the next search direction according to Eq. (9.60), using Eq. (9.61), Eq. (9.62), or Eq. (9.63) to calculate β_k:

$$\mathbf{p}_k = -\mathbf{g}_k + \beta_k \mathbf{p}_{k-1}, \tag{12.15}$$

with

$$\beta_k = \frac{\Delta\mathbf{g}_{k-1}^T \mathbf{g}_k}{\Delta\mathbf{g}_{k-1}^T \mathbf{p}_{k-1}} \text{ or } \beta_k = \frac{\mathbf{g}_k^T \mathbf{g}_k}{\mathbf{g}_{k-1}^T \mathbf{g}_{k-1}} \text{ or } \beta_k = \frac{\Delta\mathbf{g}_{k-1}^T \mathbf{g}_k}{\mathbf{g}_{k-1}^T \mathbf{g}_{k-1}}. \tag{12.16}$$

4. If the algorithm has not converged, continue from step 2.

This conjugate gradient algorithm cannot be applied directly to the neural network training task, because the performance index is not quadratic. This affects the algorithm in two ways. First, we will not be able to use Eq. (9.31) to minimize the function along a line, as required in step 2. Second, the exact minimum will not normally be reached in a finite number of steps, and therefore the algorithm will need to be reset after some set number of iterations.

Let's address the linear search first. We need to have a general procedure for locating the minimum of a function in a specified direction. This will involve two steps: interval location and interval reduction. The purpose of the interval location step is to find some initial interval that contains a local minimum. The interval reduction step then reduces the size of the initial interval until the minimum is located to the desired accuracy.

Interval Location

We will use a function comparison method [Scal85] to perform the *interval location* step. This procedure is illustrated in Figure 12.13. We begin by evaluating the performance index at an initial point, represented by a_1 in the figure. This point corresponds to the current values of the network weights and biases. In other words, we are evaluating

$$F(\mathbf{x}_0)\,. \tag{12.17}$$

The next step is to evaluate the function at a second point, represented by b_1 in the figure, which is a distance ε from the initial point, along the first search direction $\mathbf{p}_0$. In other words, we are evaluating

$$F(\mathbf{x}_0 + \varepsilon\mathbf{p}_0)\,. \tag{12.18}$$

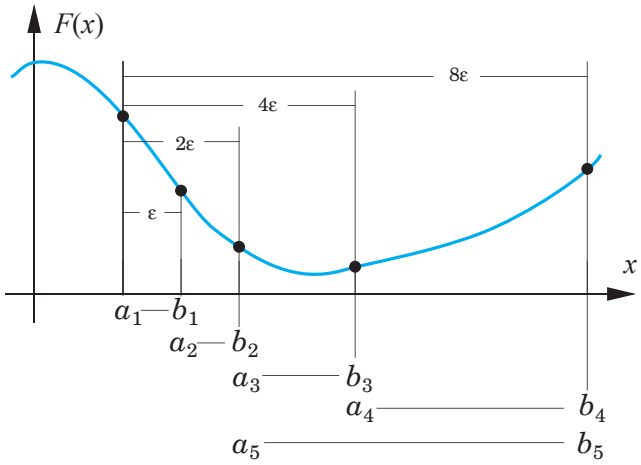


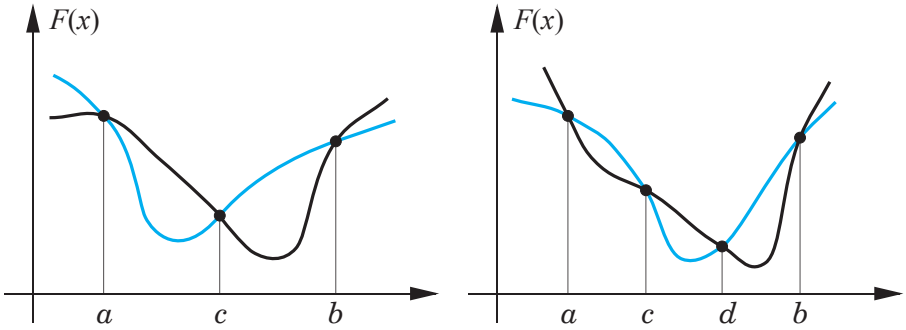
Figure 12.13 Interval Location

We then continue to evaluate the performance index at new points b_i, successively doubling the distance between points. This process stops when the function increases between two consecutive evaluations. In Figure 12.13 this is represented by b_3 to b_4. At this point we know that the minimum is bracketed by the two points a_5 and b_5. We cannot narrow the interval any further, because the minimum may occur either in the interval $[a_4, b_4]$ or in the interval $[a_3, b_3]$. These two possibilities are illustrated in Figure 12.14 (a).

Interval Reduction

Now that we have located an interval containing the minimum, the next step in the linear search is *interval reduction*. This will involve evaluating the function at points inside the interval $[a_5, b_5]$, which was selected in the interval location step. From Figure 12.14 we can see that we will need to evaluate the function at two internal points (at least) in order to reduce the size of the interval of uncertainty. Figure 12.14 (a) shows that one internal function evaluation does not provide us with any information on the location of the minimum. However, if we evaluate the function at two points c and d, as in Figure 12.14 (b), we can reduce the interval of uncertainty. If

$F(c) > F(d)$, as shown in Figure 12.14 (b), then the minimum must occur in the interval $[c, b]$. Conversely, if $F(c) < F(d)$, then the minimum must occur in the interval $[a, d]$. (Note that we are assuming that there is a single minimum located in the initial interval. More about that later.)



(a) Interval is not reduced.

(b) Minimum must occur between c and b.

Figure 12.14 Reducing the Size of the Interval of Uncertainty

Golden Section Search

The procedure described above suggests a method for reducing the size of the interval of uncertainty. We now need to decide how to determine the locations of the internal points c and d. There are several ways to do this (see [Scal85]). We will use a method called the *Golden Section search*, which is designed to reduce the number of function evaluations required. At each iteration one new function evaluation is required. For example, in the case illustrated in Figure 12.14 (b), point a would be discarded and point c would become the new a. Then point d would become the new point c, and a new d would be placed between the original points d and b. The trick is to place the new point so that the interval of uncertainty will be reduced as quickly as possible.

The algorithm for the Golden Section search is as follows [Scal85]:

$\tau = 0.618$

Set $c_1 = a_1 + (1-\tau)(b_1 - a_1)$, $F_c = F(c_1)$.

$d_1 = b_1 - (1-\tau)(b_1 - a_1)$, $F_d = F(d_1)$.

For $k = 1, 2, \ldots$ repeat

If $F_c < F_d$ then

Set $a_{k+1} = a_k$; $b_{k+1} = d_k$; $d_{k+1} = c_k$

$c_{k+1} = a_{k+1} + (1-\tau)(b_{k+1} - a_{k+1})$

$F_d = F_c$; $F_c = F(c_{k+1})$

else

Set $a_{k+1} = c_k;\ b_{k+1} = b_k;\ c_{k+1} = d_k$

$d_{k+1} = b_{k+1} - (1-\tau)(b_{k+1} - a_{k+1})$

$F_c = F_d;\ F_d = F(d_{k+1})$

end

end until $b_{k+1} - a_{k+1} < tol$

Where tol is the accuracy tolerance set by the user.

(See Problem P12.4 for a numerical example of the interval location and interval reduction procedures.)

There is one more modification to the conjugate gradient algorithm that needs to be made before we apply it to neural network training. For quadratic functions the algorithm will converge to the minimum in at most n iterations, where n is the number of parameters being optimized. The mean squared error performance index for multilayer networks is not quadratic, therefore the algorithm would not normally converge in n iterations. The development of the conjugate gradient algorithm does not indicate what search direction to use once a cycle of n iterations has been completed. There have been many procedures suggested, but the simplest method is to reset the search direction to the steepest descent direction (negative of the gradient) after n iterations [Scal85]. We will use this method.

Let's now apply the conjugate gradient algorithm to the function approximation example that we have been using to demonstrate the other neural network training algorithms. We will use the backpropagation algorithm to compute the gradient (using Eq. (11.23) and Eq. (11.24)) and the conjugate gradient algorithm to determine the weight updates. This is a batch mode algorithm, as the gradient is computed after the entire training set has been presented to the network.

Figure 12.15 shows the intermediate steps of the CGBP algorithm for the first three iterations. The interval location process is illustrated by the open blue circles; each one represents one evaluation of the function. The final interval is indicated by the larger open black circles. The black dots in Figure 12.15 indicate the location of the new interior points during the Golden Section search, one for each iteration of the procedure. The final point is indicated by a blue dot.

Figure 12.16 shows the total trajectory to convergence. Notice that the CGBP algorithm converges in many fewer iterations than the other algorithms that we have tested. This is a little deceiving, since each iteration of CGBP requires more computations than the other methods; there are many function evaluations involved in each iteration of CGBP. Even so, CGBP has been shown to be one of the fastest batch training algorithms for multilayer networks [Char92].

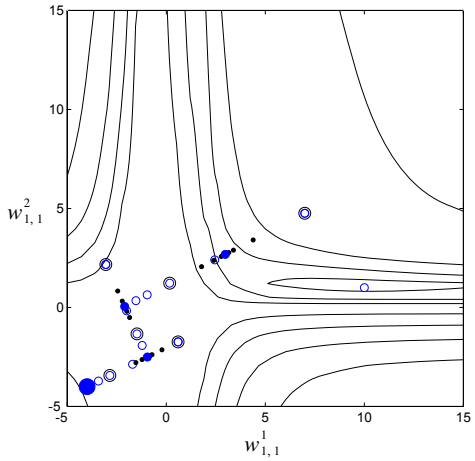


Figure 12.15 Intermediate Steps of CGBP

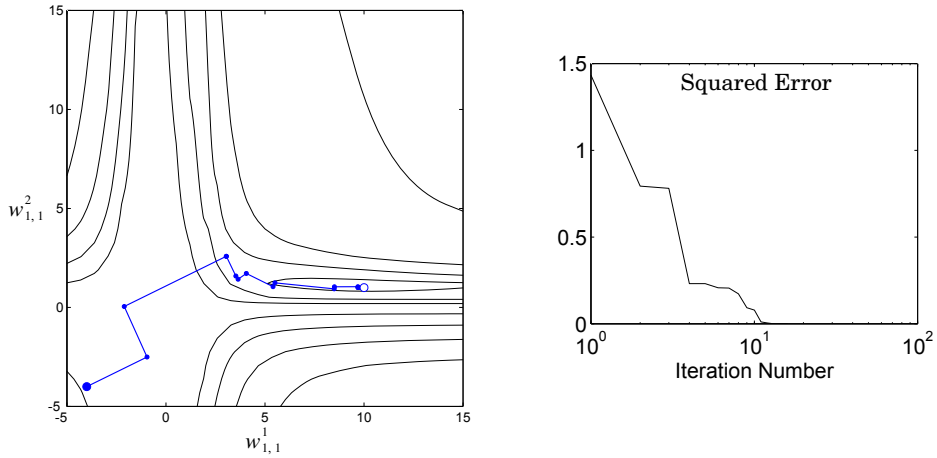
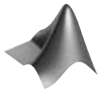


Figure 12.16 Conjugate Gradient Trajectory



To experiment with CGBP, use the MATLAB® Neural Network Design Demonstrations Conjugate Gradient Line Search (`nnd12ls`) *and Conjugate Gradient Backpropagation* (`nnd12cg`).

Levenberg-Marquardt Algorithm

The Levenberg-Marquardt algorithm is a variation of Newton's method that was designed for minimizing functions that are sums of squares of other nonlinear functions. This is very well suited to neural network training where the performance index is the mean squared error.

Basic Algorithm

Let's begin by considering the form of Newton's method where the performance index is a sum of squares. Recall from Chapter 9 that Newton's method for optimizing a performance index $F(\mathbf{x})$ is

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \mathbf{A}_k^{-1}\mathbf{g}_k, \tag{12.19}$$

$$\text{where } \mathbf{A}_k \equiv \nabla^2 F(\mathbf{x})\Big|_{\mathbf{x}=\mathbf{x}_k} \text{ and } \mathbf{g}_k \equiv \nabla F(\mathbf{x})\Big|_{\mathbf{x}=\mathbf{x}_k}.$$

If we assume that $F(\mathbf{x})$ is a sum of squares function:

$$F(\mathbf{x}) = \sum_{i=1}^{N} v_i^2(\mathbf{x}) = \mathbf{v}^T(\mathbf{x})\mathbf{v}(\mathbf{x}), \tag{12.20}$$

then the jth element of the gradient would be

$$[\nabla F(\mathbf{x})]_j = \frac{\partial F(\mathbf{x})}{\partial x_j} = 2\sum_{i=1}^{N} v_i(\mathbf{x})\frac{\partial v_i(\mathbf{x})}{\partial x_j}. \tag{12.21}$$

The gradient can therefore be written in matrix form:

$$\nabla F(\mathbf{x}) = 2\mathbf{J}^T(\mathbf{x})\mathbf{v}(\mathbf{x}), \tag{12.22}$$

where

$$\mathbf{J}(\mathbf{x}) = \begin{bmatrix} \frac{\partial v_1(\mathbf{x})}{\partial x_1} & \frac{\partial v_1(\mathbf{x})}{\partial x_2} & \cdots & \frac{\partial v_1(\mathbf{x})}{\partial x_n} \\ \frac{\partial v_2(\mathbf{x})}{\partial x_1} & \frac{\partial v_2(\mathbf{x})}{\partial x_2} & \cdots & \frac{\partial v_2(\mathbf{x})}{\partial x_n} \\ \vdots & \vdots & & \vdots \\ \frac{\partial v_N(\mathbf{x})}{\partial x_1} & \frac{\partial v_N(\mathbf{x})}{\partial x_2} & \cdots & \frac{\partial v_N(\mathbf{x})}{\partial x_n} \end{bmatrix}. \tag{12.23}$$

Jacobian Matrix

is the *Jacobian matrix*.

Next we want to find the Hessian matrix. The k, j element of the Hessian matrix would be

$$[\nabla^2 F(\mathbf{x})]_{k,j} = \frac{\partial^2 F(\mathbf{x})}{\partial x_k \partial x_j} = 2\sum_{i=1}^{N}\left\{\frac{\partial v_i(\mathbf{x})}{\partial x_k}\frac{\partial v_i(\mathbf{x})}{\partial x_j} + v_i(\mathbf{x})\frac{\partial^2 v_i(\mathbf{x})}{\partial x_k \partial x_j}\right\}. \tag{12.24}$$

The Hessian matrix can then be expressed in matrix form:

$$\nabla^2 F(\mathbf{x}) = 2\mathbf{J}^T(\mathbf{x})\mathbf{J}(\mathbf{x}) + 2\mathbf{S}(\mathbf{x}), \tag{12.25}$$

where

$$\mathbf{S}(\mathbf{x}) = \sum_{i=1}^{N} v_i(\mathbf{x})\nabla^2 v_i(\mathbf{x}) . \tag{12.26}$$

If we assume that $\mathbf{S}(\mathbf{x})$ is small, we can approximate the Hessian matrix as

$$\nabla^2 F(\mathbf{x}) \cong 2\mathbf{J}^T(\mathbf{x})\mathbf{J}(\mathbf{x}) . \tag{12.27}$$

Gauss-Newton

If we then substitute Eq. (12.27) and Eq. (12.22) into Eq. (12.19), we obtain the *Gauss-Newton* method:

$$\begin{aligned}\mathbf{x}_{k+1} &= \mathbf{x}_k - [2\mathbf{J}^T(\mathbf{x}_k)\mathbf{J}(\mathbf{x}_k)]^{-1} 2\mathbf{J}^T(\mathbf{x}_k)\mathbf{v}(\mathbf{x}_k) \\ &= \mathbf{x}_k - [\mathbf{J}^T(\mathbf{x}_k)\mathbf{J}(\mathbf{x}_k)]^{-1}\mathbf{J}^T(\mathbf{x}_k)\mathbf{v}(\mathbf{x}_k) .\end{aligned} \tag{12.28}$$

Note that the advantage of Gauss-Newton over the standard Newton's method is that it does not require calculation of second derivatives.

One problem with the Gauss-Newton method is that the matrix $\mathbf{H} = \mathbf{J}^T\mathbf{J}$ may not be invertible. This can be overcome by using the following modification to the approximate Hessian matrix:

$$\mathbf{G} = \mathbf{H} + \mu\mathbf{I} . \tag{12.29}$$

To see how this matrix can be made invertible, suppose that the eigenvalues and eigenvectors of $\mathbf{H}$ are $\{\lambda_1, \lambda_2, \ldots, \lambda_n\}$ and $\{\mathbf{z}_1, \mathbf{z}_2, \ldots, \mathbf{z}_n\}$. Then

$$\mathbf{G}\mathbf{z}_i = [\mathbf{H} + \mu\mathbf{I}]\mathbf{z}_i = \mathbf{H}\mathbf{z}_i + \mu\mathbf{z}_i = \lambda_i\mathbf{z}_i + \mu\mathbf{z}_i = (\lambda_i + \mu)\mathbf{z}_i . \tag{12.30}$$

Therefore the eigenvectors of $\mathbf{G}$ are the same as the eigenvectors of $\mathbf{H}$, and the eigenvalues of $\mathbf{G}$ are $(\lambda_i + \mu)$. $\mathbf{G}$ can be made positive definite by increasing μ until $(\lambda_i + \mu) > 0$ for all i, and therefore the matrix will be invertible.

Levenberg-Marquardt

This leads to the *Levenberg-Marquardt* algorithm [Scal85]:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - [\mathbf{J}^T(\mathbf{x}_k)\mathbf{J}(\mathbf{x}_k) + \mu_k\mathbf{I}]^{-1}\mathbf{J}^T(\mathbf{x}_k)\mathbf{v}(\mathbf{x}_k) . \tag{12.31}$$

or

$$\Delta\mathbf{x}_k = -[\mathbf{J}^T(\mathbf{x}_k)\mathbf{J}(\mathbf{x}_k) + \mu_k\mathbf{I}]^{-1}\mathbf{J}^T(\mathbf{x}_k)\mathbf{v}(\mathbf{x}_k) . \tag{12.32}$$

This algorithm has the very useful feature that as μ_k is increased it approaches the steepest descent algorithm with small learning rate:

$$\mathbf{x}_{k+1} \cong \mathbf{x}_k - \frac{1}{\mu_k}\mathbf{J}^T(\mathbf{x}_k)\mathbf{v}(\mathbf{x}_k) = \mathbf{x}_k - \frac{1}{2\mu_k}\nabla F(\mathbf{x}), \text{ for large } \mu_k, \tag{12.33}$$

while as μ_k is decreased to zero the algorithm becomes Gauss-Newton.

The algorithm begins with μ_k set to some small value (e.g., $\mu_k = 0.01$). If a step does not yield a smaller value for $F(\mathbf{x})$, then the step is repeated with μ_k multiplied by some factor $\vartheta > 1$ (e.g., $\vartheta = 10$). Eventually $F(\mathbf{x})$ should decrease, since we would be taking a small step in the direction of steepest descent. If a step does produce a smaller value for $F(\mathbf{x})$, then μ_k is divided by ϑ for the next step, so that the algorithm will approach Gauss-Newton, which should provide faster convergence. The algorithm provides a nice compromise between the speed of Newton's method and the guaranteed convergence of steepest descent.

Now let's see how we can apply the Levenberg-Marquardt algorithm to the multilayer network training problem. The performance index for multilayer network training is the mean squared error (see Eq. (11.11)). If each target occurs with equal probability, the mean squared error is proportional to the sum of squared errors over the Q targets in the training set:

$$\begin{aligned} F(\mathbf{x}) &= \sum_{q=1}^{Q} (\mathbf{t}_q - \mathbf{a}_q)^T(\mathbf{t}_q - \mathbf{a}_q) \\ &= \sum_{q=1}^{Q} \mathbf{e}_q^T\mathbf{e}_q = \sum_{q=1}^{Q}\sum_{j=1}^{S^M} (e_{j,q})^2 = \sum_{i=1}^{N} (v_i)^2, \end{aligned} \tag{12.34}$$

where $e_{j,q}$ is the j th element of the error for the q th input/target pair.

Eq. (12.34) is equivalent to the performance index, Eq. (12.20), for which Levenberg-Marquardt was designed. Therefore it should be a straightforward matter to adapt the algorithm for network training. It turns out that this is true in concept, but it does require some care in working out the details.

Jacobian Calculation

The key step in the Levenberg-Marquardt algorithm is the computation of the Jacobian matrix. To perform this computation we will use a variation of the backpropagation algorithm. Recall that in the standard backpropagation procedure we compute the derivatives of the squared errors, with respect to the weights and biases of the network. To create the Jacobian matrix we need to compute the derivatives of the errors, instead of the derivatives of the squared errors.

It is a simple matter conceptually to modify the backpropagation algorithm to compute the elements of the Jacobian matrix. Unfortunately, although

the basic concept is simple, the details of the implementation can be a little tricky. For that reason you may want to skim through the rest of this section on your first reading, in order to obtain an overview of the general flow of the presentation, and return later to pick up the details. It may also be helpful to review the development of the backpropagation algorithm in Chapter 11 before proceeding.

Before we present the procedure for computing the Jacobian, let's take a closer look at its form (Eq. (12.23)). Note that the error vector is

$$\mathbf{v}^T = \begin{bmatrix} v_1 & v_2 & \cdots & v_N \end{bmatrix} = \begin{bmatrix} e_{1,1} & e_{2,1} & \cdots & e_{S^M,1} & e_{1,2} & \cdots & e_{S^M,Q} \end{bmatrix}, \tag{12.35}$$

the parameter vector is

$$\mathbf{x}^T = \begin{bmatrix} x_1 & x_2 & \cdots & x_n \end{bmatrix} = \begin{bmatrix} w^1_{1,1} & w^1_{1,2} & \cdots & w^1_{S^1,R} & b^1_1 & \cdots & b^1_{S^1} & w^2_{1,1} & \cdots & b^M_{S^M} \end{bmatrix}, \tag{12.36}$$

$N = Q \times S^M$ and $n = S^1(R+1) + S^2(S^1+1) + \cdots + S^M(S^{M-1}+1)$.

Therefore, if we make these substitutions into Eq. (12.23), the Jacobian matrix for multilayer network training can be written

$$\mathbf{J}(\mathbf{x}) = \begin{bmatrix} \frac{\partial e_{1,1}}{\partial w^1_{1,1}} & \frac{\partial e_{1,1}}{\partial w^1_{1,2}} & \cdots & \frac{\partial e_{1,1}}{\partial w^1_{S^1,R}} & \frac{\partial e_{1,1}}{\partial b^1_1} & \cdots \\ \frac{\partial e_{2,1}}{\partial w^1_{1,1}} & \frac{\partial e_{2,1}}{\partial w^1_{1,2}} & \cdots & \frac{\partial e_{2,1}}{\partial w^1_{S^1,R}} & \frac{\partial e_{2,1}}{\partial b^1_1} & \cdots \\ \vdots & \vdots & & \vdots & \vdots & \\ \frac{\partial e_{S^M,1}}{\partial w^1_{1,1}} & \frac{\partial e_{S^M,1}}{\partial w^1_{1,2}} & \cdots & \frac{\partial e_{S^M,1}}{\partial w^1_{S^1,R}} & \frac{\partial e_{S^M,1}}{\partial b^1_1} & \cdots \\ \frac{\partial e_{1,2}}{\partial w^1_{1,1}} & \frac{\partial e_{1,2}}{\partial w^1_{1,2}} & \cdots & \frac{\partial e_{1,2}}{\partial w^1_{S^1,R}} & \frac{\partial e_{1,2}}{\partial b^1_1} & \cdots \\ \vdots & \vdots & & \vdots & \vdots & \end{bmatrix}. \tag{12.37}$$

The terms in this Jacobian matrix can be computed by a simple modification to the backpropagation algorithm.

Standard backpropagation calculates terms like

$$\frac{\partial \hat{F}(\mathbf{x})}{\partial x_l} = \frac{\partial \mathbf{e}_q{}^T \mathbf{e}_q}{\partial x_l}. \tag{12.38}$$

For the elements of the Jacobian matrix that are needed for the Levenberg-Marquardt algorithm we need to calculate terms like

$$[\mathbf{J}]_{h,l} = \frac{\partial v_h}{\partial x_l} = \frac{\partial e_{k,q}}{\partial x_l}. \tag{12.39}$$

Recall from Eq. (11.18) in our derivation of backpropagation that

$$\frac{\partial \hat{F}}{\partial w^m_{i,j}} = \frac{\partial \hat{F}}{\partial n^m_i} \times \frac{\partial n^m_i}{\partial w^m_{i,j}}, \tag{12.40}$$

where the first term on the right-hand side was defined as the sensitivity:

$$s^m_i \equiv \frac{\partial \hat{F}}{\partial n^m_i}. \tag{12.41}$$

The backpropagation process computed the sensitivities through a recurrence relationship from the last layer backward to the first layer. We can use the same concept to compute the terms needed for the Jacobian matrix (Eq. (12.37)) if we define a new *Marquardt sensitivity*:

Marquardt Sensitivity

$$\tilde{s}^m_{i,h} \equiv \frac{\partial v_h}{\partial n^m_{i,q}} = \frac{\partial e_{k,q}}{\partial n^m_{i,q}}, \tag{12.42}$$

where, from Eq. (12.35), $h = (q-1)S^M + k$.

Now we can compute elements of the Jacobian by

$$[\mathbf{J}]_{h,l} = \frac{\partial v_h}{\partial x_l} = \frac{\partial e_{k,q}}{\partial w^m_{i,j}} = \frac{\partial e_{k,q}}{\partial n^m_{i,q}} \times \frac{\partial n^m_{i,q}}{\partial w^m_{i,j}} = \tilde{s}^m_{i,h} \times \frac{\partial n^m_{i,q}}{\partial w^m_{i,j}} = \tilde{s}^m_{i,h} \times a^{m-1}_{j,q}, \tag{12.43}$$

or if x_l is a bias,

$$[\mathbf{J}]_{h,l} = \frac{\partial v_h}{\partial x_l} = \frac{\partial e_{k,q}}{\partial b^m_i} = \frac{\partial e_{k,q}}{\partial n^m_{i,q}} \times \frac{\partial n^m_{i,q}}{\partial b^m_i} = \tilde{s}^m_{i,h} \times \frac{\partial n^m_{i,q}}{\partial b^m_i} = \tilde{s}^m_{i,h}. \tag{12.44}$$

The Marquardt sensitivities can be computed through the same recurrence relations as the standard sensitivities (Eq. (11.35)) with one modification at the final layer, which for standard backpropagation is computed with Eq. (11.40). For the Marquardt sensitivities at the final layer we have

$$\tilde{s}^M_{i,h} = \frac{\partial v_h}{\partial n^M_{i,q}} = \frac{\partial e_{k,q}}{\partial n^M_{i,q}} = \frac{\partial (t_{k,q} - a^M_{k,q})}{\partial n^M_{i,q}} = -\frac{\partial a^M_{k,q}}{\partial n^M_{i,q}}$$

$$= \begin{cases} -\dot{f}^M(n^M_{i,q}) & \text{for } i = k \\ 0 & \text{for } i \neq k \end{cases} . \tag{12.45}$$

Therefore when the input $\mathbf{p}_q$ has been applied to the network and the corresponding network output $\mathbf{a}^M_q$ has been computed, the Levenberg-Marquardt backpropagation is initialized with

$$\tilde{\mathbf{S}}^M_q = -\dot{\mathbf{F}}^M(\mathbf{n}^M_q) , \tag{12.46}$$

where $\dot{\mathbf{F}}^M(\mathbf{n}^M)$ is defined in Eq. (11.34). Each column of the matrix $\tilde{\mathbf{S}}^M_q$ must be backpropagated through the network using Eq. (11.35) to produce one row of the Jacobian matrix. The columns can also be backpropagated together using

$$\tilde{\mathbf{S}}^m_q = \dot{\mathbf{F}}^m(\mathbf{n}^m_q)(\mathbf{W}^{m+1})^T \tilde{\mathbf{S}}^{m+1}_q . \tag{12.47}$$

The total Marquardt sensitivity matrices for each layer are then created by augmenting the matrices computed for each input:

$$\tilde{\mathbf{S}}^m = \left[\tilde{\mathbf{S}}^m_1 \middle| \tilde{\mathbf{S}}^m_2 \middle| \cdots \middle| \tilde{\mathbf{S}}^m_Q \right] . \tag{12.48}$$

Note that for each input that is presented to the network we will backpropagate S^M sensitivity vectors. This is because we are computing the derivatives of each individual error, rather than the derivative of the sum of squares of the errors. For every input applied to the network there will be S^M errors (one for each element of the network output). For each error there will be one row of the Jacobian matrix.

After the sensitivities have been backpropagated, the Jacobian matrix is computed using Eq. (12.43) and Eq. (12.44). See Problem P12.5 for a numerical illustration of the Jacobian computation.

LMBP

The iterations of the Levenberg-Marquardt backpropagation algorithm (*LMBP*) can be summarized as follows:

1. Present all inputs to the network and compute the corresponding network outputs (using Eq. (11.41) and Eq. (11.42)) and the errors $\mathbf{e}_q = \mathbf{t}_q - \mathbf{a}^M_q$. Compute the sum of squared errors over all inputs, $F(\mathbf{x})$,

using Eq. (12.34).

2. Compute the Jacobian matrix, Eq. (12.37). Calculate the sensitivities with the recurrence relations Eq. (12.47), after initializing with Eq. (12.46). Augment the individual matrices into the Marquardt sensitivities using Eq. (12.48). Compute the elements of the Jacobian matrix with Eq. (12.43) and Eq. (12.44).

3. Solve Eq. (12.32) to obtain $\Delta \mathbf{x}_k$.

4. Recompute the sum of squared errors using $\mathbf{x}_k + \Delta \mathbf{x}_k$. If this new sum of squares is smaller than that computed in step 1, then divide μ by ϑ, let $\mathbf{x}_{k+1} = \mathbf{x}_k + \Delta \mathbf{x}_k$ and go back to step 1. If the sum of squares is not reduced, then multiply μ by ϑ and go back to step 3.

The algorithm is assumed to have converged when the norm of the gradient, Eq. (12.22), is less than some predetermined value, or when the sum of squares has been reduced to some error goal.

To illustrate LMBP, let's apply it to the function approximation problem introduced at the beginning of this chapter. We will begin by looking at the basic Levenberg-Marquardt step. Figure 12.17 illustrates the possible steps the LMBP algorithm could take on the first iteration.

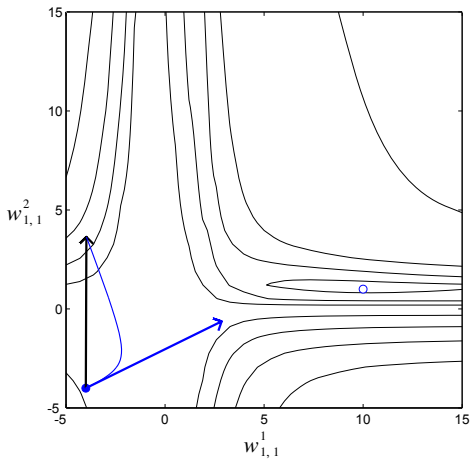


Figure 12.17 Levenberg-Marquardt Step

The black arrow represents the direction taken for small μ_k, which corresponds to the Gauss-Newton direction. The blue arrow represents the direction taken for large μ_k, which corresponds to the steepest descent direction. (This was the initial direction taken by all of the previous algorithms discussed.) The blue curve represents the Levenberg-Marquardt step for all intermediate values of μ_k. Note that as μ_k is increased the algorithm moves toward a small step in the direction of steepest descent. This guarantees that the algorithm will always be able to reduce the sum of squares at each iteration.

Figure 12.18 shows the path of the LMBP trajectory to convergence, with $\mu_0 = 0.01$ and $\vartheta = 5$. Note that the algorithm converges in fewer iterations than any of the methods we have discussed so far. Of course this algorithm also requires more computation per iteration than any of the other algorithms, since it involves a matrix inversion. Even given the large number of computations, however, the LMBP algorithm appears to be the fastest neural network training algorithm for moderate numbers of network parameters [HaMe94].

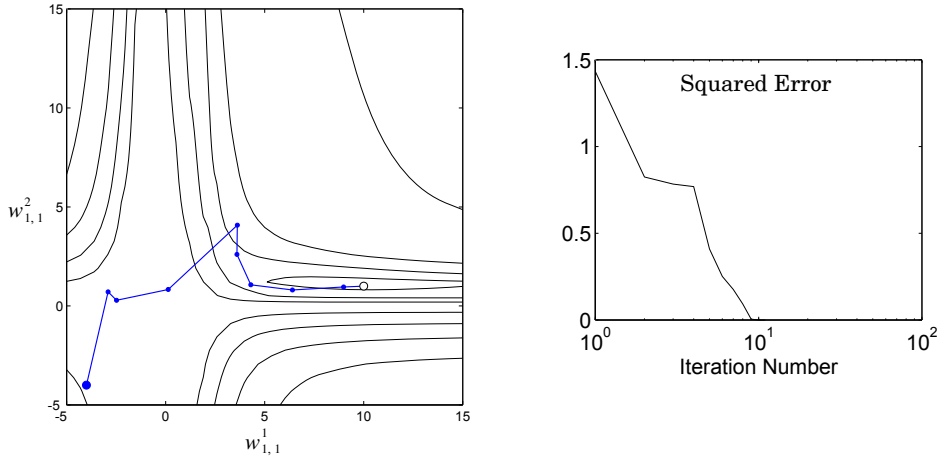
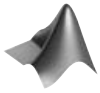


Figure 12.18 LMBP Trajectory



To experiment with the LMBP algorithm, use the MATLAB® Neural Network Design Demonstrations Marquardt Step (`nnd12ms`) *and Marquardt Backpropagation* (`nnd12m`).

The key drawback of the LMBP algorithm is the storage requirement. The algorithm must store the approximate Hessian matrix $\mathbf{J}^T\mathbf{J}$. This is an $n \times n$ matrix, where n is the number of parameters (weights and biases) in the network. Recall that the other methods discussed need only store the gradient, which is an n-dimensional vector. When the number of parameters is very large, it may be impractical to use the Levenberg-Marquardt algorithm. (What constitutes "very large" depends on the available memory on your computer, but typically a few thousand parameters is an upper limit.)

Summary of Results

Heuristic Variations of Backpropagation

Batching

The parameters are updated only after the entire training set has been presented. The gradients calculated for each training example are averaged together to produce a more accurate estimate of the gradient. (If the training set is complete, i.e., covers all possible input/output pairs, then the gradient estimate will be exact.)

Backpropagation with Momentum (MOBP)

$$\Delta \mathbf{W}^m(k) = \gamma \Delta \mathbf{W}^m(k-1) - (1-\gamma)\alpha \mathbf{s}^m (\mathbf{a}^{m-1})^T$$

$$\Delta \mathbf{b}^m(k) = \gamma \Delta \mathbf{b}^m(k-1) - (1-\gamma)\alpha \mathbf{s}^m$$

Variable Learning Rate Backpropagation (VLBP)

1. If the squared error (over the entire training set) increases by more than some set percentage ζ (typically one to five percent) after a weight update, then the weight update is discarded, the learning rate is multiplied by some factor $\rho < 1$, and the momentum coefficient γ (if it is used) is set to zero.

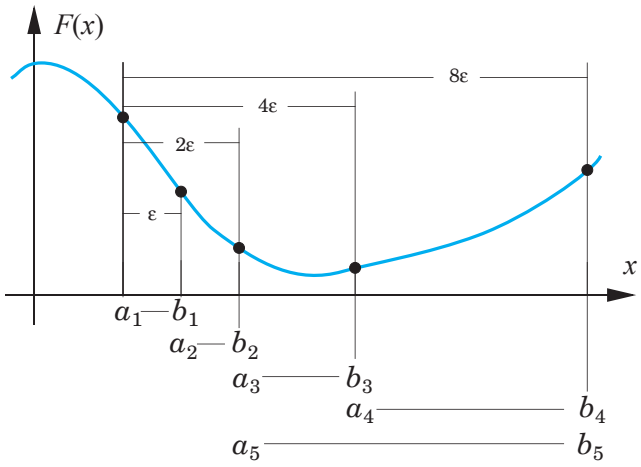
2. If the squared error decreases after a weight update, then the weight update is accepted and the learning rate is multiplied by some factor $\eta > 1$. If γ has been previously set to zero, it is reset to its original value.

3. If the squared error increases by less than ζ, then the weight update is accepted but the learning rate and the momentum coefficient are unchanged.

Numerical Optimization Techniques

Conjugate Gradient

Interval Location



Interval Reduction (Golden Section Search)

$\tau = 0.618$

Set $c_1 = a_1 + (1-\tau)(b_1 - a_1)$, $F_c = F(c_1)$.

$d_1 = b_1 - (1-\tau)(b_1 - a_1)$, $F_d = F(d_1)$.

For $k = 1, 2, \ldots$ repeat

If $F_c < F_d$ then

Set $a_{k+1} = a_k$; $b_{k+1} = d_k$; $d_{k+1} = c_k$

$c_{k+1} = a_{k+1} + (1-\tau)(b_{k+1} - a_{k+1})$

$F_d = F_c$; $F_c = F(c_{k+1})$

else

Set $a_{k+1} = c_k$; $b_{k+1} = b_k$; $c_{k+1} = d_k$

$d_{k+1} = b_{k+1} - (1-\tau)(b_{k+1} - a_{k+1})$

$F_c = F_d$; $F_d = F(d_{k+1})$

end

end until $b_{k+1} - a_{k+1} < tol$

Levenberg-Marquardt Backpropagation (LMBP)

$$\Delta \mathbf{x}_k = -[\mathbf{J}^T(\mathbf{x}_k)\mathbf{J}(\mathbf{x}_k) + \mu_k \mathbf{I}]^{-1}\mathbf{J}^T(\mathbf{x}_k)\mathbf{v}(\mathbf{x}_k)$$

$$\mathbf{v}^T = \begin{bmatrix} v_1 & v_2 & \dots & v_N \end{bmatrix} = \begin{bmatrix} e_{1,1} & e_{2,1} & \dots & e_{S^M,1} & e_{1,2} & \dots & e_{S^M,Q} \end{bmatrix}$$

$$\mathbf{x}^T = \begin{bmatrix} x_1 & x_2 & \dots & x_n \end{bmatrix} = \begin{bmatrix} w^1_{1,1} & w^1_{1,2} & \dots & w^1_{S^1,R} & b^1_1 & \dots & b^1_{S^1} & w^2_{1,1} & \dots & b^M_{S^M} \end{bmatrix}$$

$$N = Q \times S^M \text{ and } n = S^1(R+1) + S^2(S^1+1) + \dots + S^M(S^{M-1}+1)$$

$$\mathbf{J}(\mathbf{x}) = \begin{bmatrix} \frac{\partial e_{1,1}}{\partial w^1_{1,1}} & \frac{\partial e_{1,1}}{\partial w^1_{1,2}} & \dots & \frac{\partial e_{1,1}}{\partial w^1_{S^1,R}} & \frac{\partial e_{1,1}}{\partial b^1_1} & \dots \\ \frac{\partial e_{2,1}}{\partial w^1_{1,1}} & \frac{\partial e_{2,1}}{\partial w^1_{1,2}} & \dots & \frac{\partial e_{2,1}}{\partial w^1_{S^1,R}} & \frac{\partial e_{2,1}}{\partial b^1_1} & \dots \\ \vdots & \vdots & & \vdots & \vdots & \\ \frac{\partial e_{S^M,1}}{\partial w^1_{1,1}} & \frac{\partial e_{S^M,1}}{\partial w^1_{1,2}} & \dots & \frac{\partial e_{S^M,1}}{\partial w^1_{S^1,R}} & \frac{\partial e_{S^M,1}}{\partial b^1_1} & \dots \\ \frac{\partial e_{1,2}}{\partial w^1_{1,1}} & \frac{\partial e_{1,2}}{\partial w^1_{1,2}} & \dots & \frac{\partial e_{1,2}}{\partial w^1_{S^1,R}} & \frac{\partial e_{1,2}}{\partial b^1_1} & \dots \\ \vdots & \vdots & & \vdots & \vdots & \end{bmatrix}$$

$$[\mathbf{J}]_{h,l} = \frac{\partial v_h}{\partial x_l} = \frac{\partial e_{k,q}}{\partial w^m_{i,j}} = \frac{\partial e_{k,q}}{\partial n^m_{i,q}} \times \frac{\partial n^m_{i,q}}{\partial w^m_{i,j}} = \tilde{s}^m_{i,h} \times \frac{\partial n^m_{i,q}}{\partial w^m_{i,j}} = \tilde{s}^m_{i,h} \times a^{m-1}_{j,q} \text{ for weight } x_l$$

$$[\mathbf{J}]_{h,l} = \frac{\partial v_h}{\partial x_l} = \frac{\partial e_{k,q}}{\partial b^m_i} = \frac{\partial e_{k,q}}{\partial n^m_{i,q}} \times \frac{\partial n^m_{i,q}}{\partial b^m_i} = \tilde{s}^m_{i,h} \times \frac{\partial n^m_{i,q}}{\partial b^m_i} = \tilde{s}^m_{i,h} \text{ for bias } x_l$$

$$\tilde{s}^m_{i,h} \equiv \frac{\partial v_h}{\partial n^m_{i,q}} = \frac{\partial e_{k,q}}{\partial n^m_{i,q}} \text{ (Marquardt Sensitivity) where } h = (q-1)S^M + k$$

$$\tilde{\mathbf{S}}^M_q = -\dot{\mathbf{F}}^M(\mathbf{n}^M_q)$$

$$\tilde{\mathbf{S}}^m_q = \dot{\mathbf{F}}^m(\mathbf{n}^m_q)(\mathbf{W}^{m+1})^T\tilde{\mathbf{S}}^{m+1}_q$$

$$\tilde{\mathbf{S}}^m = \begin{bmatrix} \tilde{\mathbf{S}}^m_1 \mid \tilde{\mathbf{S}}^m_2 \mid \dots \mid \tilde{\mathbf{S}}^m_Q \end{bmatrix}$$

Levenberg-Marquardt Iterations

1. Present all inputs to the network and compute the corresponding network outputs (using Eq. (11.41) and Eq. (11.42)) and the errors $\mathbf{e}_q = \mathbf{t}_q - \mathbf{a}_q^M$. Compute the sum of squared errors over all inputs, $F(\mathbf{x})$, using Eq. (12.34).

2. Compute the Jacobian matrix, Eq. (12.37). Calculate the sensitivities with the recurrence relations Eq. (12.47), after initializing with Eq. (12.46). Augment the individual matrices into the Marquardt sensitivities using Eq. (12.48). Compute the elements of the Jacobian matrix with Eq. (12.43) and Eq. (12.44).

3. Solve Eq. (12.32) to obtain $\Delta\mathbf{x}_k$.

4. Recompute the sum of squared errors using $\mathbf{x}_k + \Delta\mathbf{x}_k$. If this new sum of squares is smaller than that computed in step 1, then divide μ by ϑ, let $\mathbf{x}_{k+1} = \mathbf{x}_k + \Delta\mathbf{x}_k$ and go back to step 1. If the sum of squares is not reduced, then multiply μ by ϑ and go back to step 3.

Solved Problems

P12.1 We want to train the network shown in Figure P12.1 on the training set

$$\left\{(\mathbf{p}_1 = \begin{bmatrix}-3\end{bmatrix}), (\mathbf{t}_1 = \begin{bmatrix}0.5\end{bmatrix})\right\}, \left\{(\mathbf{p}_2 = \begin{bmatrix}2\end{bmatrix}), (\mathbf{t}_2 = \begin{bmatrix}1\end{bmatrix})\right\},$$

starting from the initial guess

$$w(0) = 0.4\,,\; b(0) = 0.15\,.$$

Demonstrate the effect of batching by computing the direction of the initial step for SDBP with and without batching.

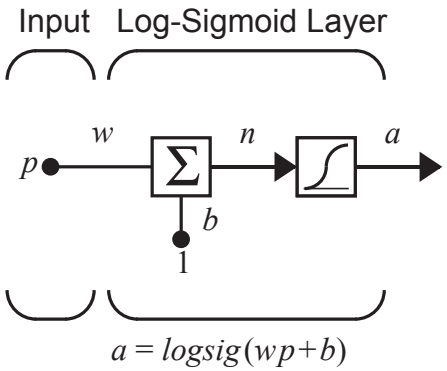


Figure P12.1 Network for Problem P12.1

Let's begin by computing the direction of the initial step if batching is not used. In this case the first step is computed from the first input/target pair. The forward and backpropagation steps are

$$a = logsig(wp + b) = \frac{1}{1 + \exp(-(0.4(-3) + 0.15))} = 0.2592$$

$$e = t - a = 0.5 - 0.2592 = 0.2408$$

$$s = -2\dot{f}(n)e = -2a(1 - a)e = -2(0.2592)(1 - 0.2592)0.2408 = -0.0925\,.$$

The direction of the initial step is the negative of the gradient. For the weight this will be

$$-sp = -(-0.0925)(-3) = -0.2774\,.$$

For the bias we have

$$-s = -(-0.0925) = 0.0925\,.$$

Therefore the direction of the initial step in the (w, b) plane would be

$$\begin{bmatrix} -0.2774 \\ 0.0925 \end{bmatrix}.$$

Now let's consider the initial direction for the batch mode algorithm. In this case the gradient is found by adding together the individual gradients found from the two sets of input/target pairs. For this we need to apply the second input to the network and perform the forward and backpropagation steps:

$$a = logsig(wp + b) = \frac{1}{1 + \exp(-(0.4(2) + 0.15))} = 0.7211$$

$$e = t - a = 1 - 0.7211 = 0.2789$$

$$s = -2\dot{f}(n)e = -2a(1 - a)e = -2(0.7211)(1 - 0.7211)0.2789 = -0.1122 .$$

The direction of the step is the negative of the gradient. For the weight this will be

$$-sp = -(-0.1122)(2) = 0.2243 .$$

For the bias we have

$$-s = -(-0.1122) = 0.1122 .$$

The partial gradient for the second input/target pair is therefore

$$\begin{bmatrix} 0.2243 \\ 0.1122 \end{bmatrix}.$$

If we now add the results from the two input/target pairs we find the direction of the first step of the batch mode SDBP to be

$$\frac{1}{2}\left(\begin{bmatrix} -0.2774 \\ 0.0925 \end{bmatrix} + \begin{bmatrix} 0.2243 \\ 0.1122 \end{bmatrix}\right) = \frac{1}{2}\begin{bmatrix} -0.0531 \\ 0.2047 \end{bmatrix} = \begin{bmatrix} -0.0265 \\ 0.1023 \end{bmatrix}.$$

The results are illustrated in Figure P12.2. The blue circle indicates the initial guess. The two blue arrows represent the directions of the partial gradients for each of the two input/target pairs, and the black arrow represents the direction of the total gradient. The function that is plotted is the sum of squared errors for the entire training set. Note that the individual partial gradients can point in quite different directions than the true gradient. However, on the average, over several iterations, the path will generally follow the steepest descent trajectory.

The relative effectiveness of the batch mode over the incremental approach depends very much on the particular problem. The incremental approach requires less storage, and, if the inputs are presented randomly to the network, the trajectory is stochastic, which makes the algorithm somewhat less likely to be trapped in a local minimum. It may also take longer to converge than the batch mode algorithm.

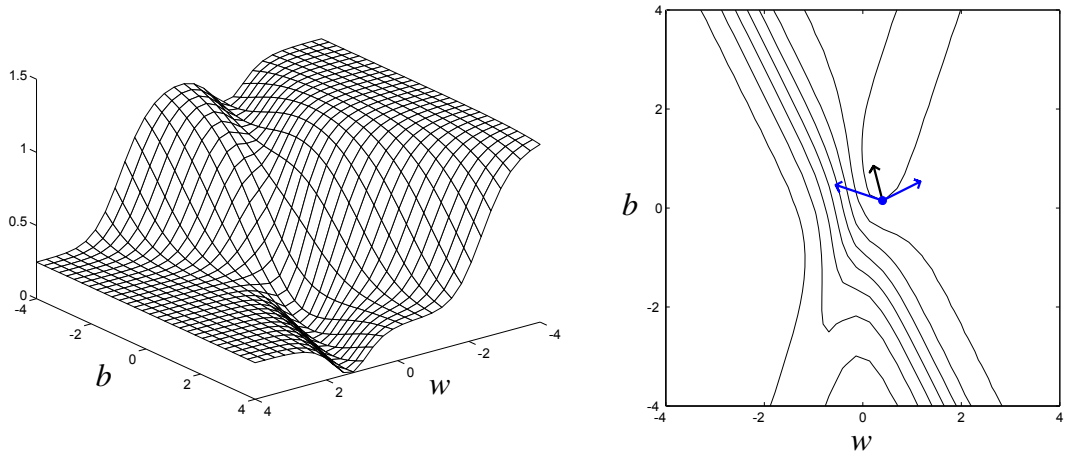


Figure P12.2 Effect of Batching in Problem P12.1

P12.2 In Chapter 9 we proved that the steepest descent algorithm, when applied to a quadratic function, would be stable if the learning rate was less than 2 divided by the maximum eigenvalue of the Hessian matrix. Show that if a momentum term is added to the steepest descent algorithm there will always be a momentum coefficient that will make the algorithm stable, regardless of the learning rate. Follow the format of the proof on page 9-6.

The standard steepest descent algorithm is

$$\Delta \mathbf{x}_k = -\alpha \nabla F(\mathbf{x}_k) = -\alpha \mathbf{g}_k ,$$

If we add momentum this becomes

$$\Delta \mathbf{x}_k = \gamma \Delta \mathbf{x}_{k-1} - (1-\gamma)\alpha \mathbf{g}_k .$$

Recall from Chapter 8 that the quadratic function has the form

$$F(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T \mathbf{A} \mathbf{x} + \mathbf{d}^T \mathbf{x} + c ,$$

and the gradient of the quadratic function is

$$\nabla F(\mathbf{x}) = \mathbf{A}\mathbf{x} + \mathbf{d} .$$

If we now insert this expression into our expression for the steepest descent algorithm with momentum we obtain

$$\Delta \mathbf{x}_k = \gamma \Delta \mathbf{x}_{k-1} - (1-\gamma)\alpha(\mathbf{A}\mathbf{x}_k + \mathbf{d}) .$$

Using the definition $\Delta \mathbf{x}_k = \mathbf{x}_{k+1} - \mathbf{x}_k$ this can be rewritten

$$\mathbf{x}_{k+1} - \mathbf{x}_k = \gamma(\mathbf{x}_k - \mathbf{x}_{k-1}) - (1-\gamma)\alpha(\mathbf{A}\mathbf{x}_k + \mathbf{d})$$

or

$$\mathbf{x}_{k+1} = [(1+\gamma)\mathbf{I} - (1-\gamma)\alpha\mathbf{A}]\mathbf{x}_k - \gamma\mathbf{x}_{k-1} - (1-\gamma)\alpha\mathbf{d} .$$

Now define a new vector

$$\tilde{\mathbf{x}}_k = \begin{bmatrix} \mathbf{x}_{k-1} \\ \mathbf{x}_k \end{bmatrix} .$$

The momentum variation of steepest descent can then be written

$$\tilde{\mathbf{x}}_{k+1} = \begin{bmatrix} \mathbf{0} & \mathbf{I} \\ -\gamma\mathbf{I} & [(1+\gamma)\mathbf{I} - (1-\gamma)\alpha\mathbf{A}] \end{bmatrix} \tilde{\mathbf{x}}_k + \begin{bmatrix} \mathbf{0} \\ -(1-\gamma)\alpha\mathbf{d} \end{bmatrix} = \mathbf{W}\tilde{\mathbf{x}}_k + \mathbf{v} .$$

This is a linear dynamic system that will be stable if the eigenvalues of $\mathbf{W}$ are less than one in magnitude. We will find the eigenvalues of $\mathbf{W}$ in stages. First, rewrite $\mathbf{W}$ as

$$\mathbf{W} = \begin{bmatrix} \mathbf{0} & \mathbf{I} \\ -\gamma\mathbf{I} & \mathbf{T} \end{bmatrix} \text{ where } \mathbf{T} = [(1+\gamma)\mathbf{I} - (1-\gamma)\alpha\mathbf{A}] .$$

The eigenvalues and eigenvectors of $\mathbf{W}$ should satisfy

$$\mathbf{W}\mathbf{z}^w = \lambda^w \mathbf{z}^w , \text{ or } \begin{bmatrix} \mathbf{0} & \mathbf{I} \\ -\gamma\mathbf{I} & \mathbf{T} \end{bmatrix} \begin{bmatrix} \mathbf{z}_1^w \\ \mathbf{z}_2^w \end{bmatrix} = \lambda^w \begin{bmatrix} \mathbf{z}_1^w \\ \mathbf{z}_2^w \end{bmatrix} .$$

This means that

$$\mathbf{z}_2^w = \lambda^w \mathbf{z}_1^w \text{ and } -\gamma\mathbf{z}_1^w + \mathbf{T}\mathbf{z}_2^w = \lambda^w \mathbf{z}_2^w .$$

At this point we will choose $\mathbf{z}_2^w$ to be an eigenvector of the matrix $\mathbf{T}$, with corresponding eigenvalue λ^t. (If this choice is not appropriate it will lead to a contradiction.) Therefore the previous equations become

$$\mathbf{z}_2^w = \lambda^w \mathbf{z}_1^w \text{ and } -\gamma \mathbf{z}_1^w + \lambda^t \mathbf{z}_2^w = \lambda^w \mathbf{z}_2^w .$$

If we substitute the first equation into the second equation we find

$$-\frac{\gamma}{\lambda^w}\mathbf{z}_2^w + \lambda^t \mathbf{z}_2^w = \lambda^w \mathbf{z}_2^w \text{ or } [(\lambda^w)^2 - \lambda^t(\lambda^w) + \gamma]\mathbf{z}_2^w = 0 .$$

Therefore for each eigenvalue λ^t of $\mathbf{T}$ there will be two eigenvalues λ^w of $\mathbf{W}$ that are roots of the quadratic equation

$$(\lambda^w)^2 - \lambda^t(\lambda^w) + \gamma = 0 .$$

From the quadratic formula we have

$$\lambda^w = \frac{\lambda^t \pm \sqrt{(\lambda^t)^2 - 4\gamma}}{2} .$$

For the algorithm to be stable the magnitude of each eigenvalue must be less than 1. We will show that there always exists some range of γ for which this is true.

Note that if the eigenvalues λ^w are complex then their magnitude will be $\sqrt{\gamma}$:

$$|\lambda^w| = \sqrt{\frac{(\lambda^t)^2}{4} + \frac{4\gamma - (\lambda^t)^2}{4}} = \sqrt{\gamma} .$$

(This is true only for real λ^t. We will show later that λ^t is real.) Since γ is between 0 and 1, the magnitude of the eigenvalue must be less than 1. It remains to show that there exists some range of γ for which all of the eigenvalues are complex.

In order for λ^w to be complex we must have

$$(\lambda^t)^2 - 4\gamma < 0 \text{ or } |\lambda^t| < 2\sqrt{\gamma} .$$

Let's now consider the eigenvalues λ^t of $\mathbf{T}$. These eigenvalues can be expressed in terms of the eigenvalues of $\mathbf{A}$. Let $\{\lambda_1, \lambda_2, \ldots, \lambda_n\}$ and $\{\mathbf{z}_1, \mathbf{z}_2, \ldots, \mathbf{z}_n\}$ be the eigenvalues and eigenvectors of the Hessian matrix. Then

$$\mathbf{T}\mathbf{z}_i = [(1+\gamma)\mathbf{I} - (1-\gamma)\alpha\mathbf{A}]\mathbf{z}_i = (1+\gamma)\mathbf{z}_i - (1-\gamma)\alpha\mathbf{A}\mathbf{z}_i$$

$$= (1+\gamma)\mathbf{z}_i - (1-\gamma)\alpha\lambda_i\mathbf{z}_i = \{(1+\gamma) - (1-\gamma)\alpha\lambda_i\}\mathbf{z}_i = \lambda_i^t \mathbf{z}_i .$$

Therefore the eigenvectors of $\mathbf{T}$ are the same as the eigenvectors of $\mathbf{A}$, and the eigenvalues of $\mathbf{T}$ are

$$\lambda_i^t = \{(1+\gamma) - (1-\gamma)\alpha\lambda_i\} .$$

(Note that λ_i^t is real, since γ, α and λ_i for symmetric $\mathbf{A}$ are real.) Therefore, in order for λ^w to be complex we must have

$$|\lambda^t| < 2\sqrt{\gamma} \text{ or } |(1+\gamma) - (1-\gamma)\alpha\lambda_i| < 2\sqrt{\gamma} .$$

For $\gamma = 1$ both sides of the inequality will equal 2. The function on the right of the inequality, as a function of γ, has a slope of 1 at $\gamma = 1$. The function on the left of the inequality has a slope of $1 + \alpha\lambda_i$. Since the eigenvalues of the Hessian will be positive real numbers if the function has a strong minimum, and the learning rate is a positive number, this slope must be greater than 1. This shows that the inequality will always hold for γ close enough to 1.

To summarize the results, we have shown that if a momentum term is added to the steepest descent algorithm on a quadratic function, then there will always be a momentum coefficient that will make the algorithm stable, regardless of the learning rate. In addition we have shown that if γ is close enough to 1, then the magnitudes of the eigenvalues of $\mathbf{W}$ will be $\sqrt{\gamma}$. It can be shown (see [Brog91]) that the magnitudes of the eigenvalues determine how fast the algorithm will converge. The smaller the magnitude, the faster the convergence. As the magnitude approaches 1, the convergence time increases.

We can demonstrate these results using the example on page 9-7. There we showed that the steepest descent algorithm, when applied to the function $F(\mathbf{x}) = x_1^2 + 25x_2^2$, was unstable for a learning rate $\alpha \geq 0.4$. In Figure P12.3 we see the steepest descent trajectory (with momentum) with $\alpha = 0.041$ and $\gamma = 0.2$. Compare this trajectory with Figure 9.3, which uses the same learning rate but no momentum.

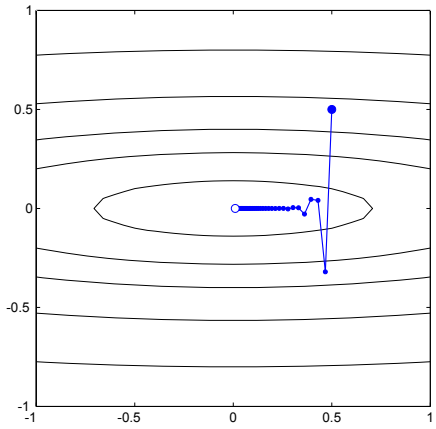


Figure P12.3 Trajectory for $\alpha = 0.041$ and $\gamma = 0.2$

P12.3 Execute three iterations of the variable learning rate algorithm on the following function (from the Chapter 9 example on page 9-7):

$$F(\mathbf{x}) = x_1^2 + 25x_2^2,$$

starting from the initial guess

$$\mathbf{x}_0 = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix},$$

and use the following values for the algorithm parameters:

$$\alpha = 0.05,\ \gamma = 0.2,\ \eta = 1.5,\ \rho = 0.5,\ \zeta = 5\%.$$

The first step is to evaluate the function at the initial guess:

$$F(\mathbf{x}_0) = \frac{1}{2}\mathbf{x}_0^T \begin{bmatrix} 2 & 0 \\ 0 & 50 \end{bmatrix} \mathbf{x}_0 = \frac{1}{2}\begin{bmatrix} 0.5 & 0.5 \end{bmatrix} \begin{bmatrix} 2 & 0 \\ 0 & 50 \end{bmatrix} \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix} = 6.5.$$

The next step is to find the gradient:

$$\nabla F(\mathbf{x}) = \begin{bmatrix} \frac{\partial}{\partial x_1} F(\mathbf{x}) \\ \frac{\partial}{\partial x_2} F(\mathbf{x}) \end{bmatrix} = \begin{bmatrix} 2x_1 \\ 50x_2 \end{bmatrix}.$$

If we evaluate the gradient at the initial guess we find:

$$\mathbf{g}_0 = \nabla F(\mathbf{x})\big|_{\mathbf{x}=\mathbf{x}_0} = \begin{bmatrix} 1 \\ 25 \end{bmatrix}.$$

With the initial learning rate of $\alpha = 0.05$, the tentative first step of the algorithm is

$$\Delta\mathbf{x}_0 = \gamma\Delta\mathbf{x}_{-1} - (1-\gamma)\alpha\mathbf{g}_0 = 0.2\begin{bmatrix} 0 \\ 0 \end{bmatrix} - 0.8(0.05)\begin{bmatrix} 1 \\ 25 \end{bmatrix} = \begin{bmatrix} -0.04 \\ -1 \end{bmatrix}$$

$$\mathbf{x}_1^t = \mathbf{x}_0 + \Delta\mathbf{x}_0 = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix} + \begin{bmatrix} -0.04 \\ -1 \end{bmatrix} = \begin{bmatrix} 0.46 \\ -0.5 \end{bmatrix}.$$

To verify that this is a valid step we must test the value of the function at this new point:

$$F(\mathbf{x}_1^t) = \frac{1}{2}(\mathbf{x}_1^t)^T\begin{bmatrix} 2 & 0 \\ 0 & 50 \end{bmatrix}\mathbf{x}_1^t = \frac{1}{2}\begin{bmatrix} 0.46 & -0.5 \end{bmatrix}\begin{bmatrix} 2 & 0 \\ 0 & 50 \end{bmatrix}\begin{bmatrix} 0.46 \\ -0.5 \end{bmatrix} = 6.4616.$$

This is less than $F(\mathbf{x}_0)$. Therefore this tentative step is accepted and the learning rate is increased:

$$\mathbf{x}_1 = \mathbf{x}_1^t = \begin{bmatrix} 0.46 \\ -0.5 \end{bmatrix},\ F(\mathbf{x}_1) = 6.4616 \text{ and } \alpha = \eta\alpha = 1.5(0.05) = 0.075.$$

The tentative second step of the algorithm is

$$\Delta\mathbf{x}_1 = \gamma\Delta\mathbf{x}_0 - (1-\gamma)\alpha\mathbf{g}_1 = 0.2\begin{bmatrix} -0.04 \\ -1 \end{bmatrix} - 0.8(0.075)\begin{bmatrix} 0.92 \\ -25 \end{bmatrix} = \begin{bmatrix} -0.0632 \\ 1.3 \end{bmatrix}$$

$$\mathbf{x}_2^t = \mathbf{x}_1 + \Delta\mathbf{x}_1 = \begin{bmatrix} 0.46 \\ -0.5 \end{bmatrix} + \begin{bmatrix} -0.0632 \\ 1.3 \end{bmatrix} = \begin{bmatrix} 0.3968 \\ 0.8 \end{bmatrix}.$$

We evaluate the function at this point:

$$F(\mathbf{x}_2^t) = \frac{1}{2}(\mathbf{x}_2^t)^T\begin{bmatrix} 2 & 0 \\ 0 & 50 \end{bmatrix}\mathbf{x}_2^t = \frac{1}{2}\begin{bmatrix} 0.3968 & 0.8 \end{bmatrix}\begin{bmatrix} 2 & 0 \\ 0 & 50 \end{bmatrix}\begin{bmatrix} 0.3968 \\ 0.8 \end{bmatrix} = 16.157.$$

Since this is more than 5% larger than $F(\mathbf{x}_1)$, we reject this step, reduce the learning rate and set the momentum coefficient to zero.

$$\mathbf{x}_2 = \mathbf{x}_1,\ F(\mathbf{x}_2) = F(\mathbf{x}_1) = 6.4616,\ \alpha = \rho\alpha = 0.5(0.075) = 0.0375,\ \gamma = 0$$

Now a new tentative step is computed (momentum is zero).

$$\Delta \mathbf{x}_2 = -\alpha \mathbf{g}_2 = -(0.0375)\begin{bmatrix} 0.92 \\ -25 \end{bmatrix} = \begin{bmatrix} -0.0345 \\ 0.9375 \end{bmatrix}$$

$$\mathbf{x}_3^t = \mathbf{x}_2 + \Delta \mathbf{x}_2 = \begin{bmatrix} 0.46 \\ -0.5 \end{bmatrix} + \begin{bmatrix} -0.0345 \\ 0.9375 \end{bmatrix} = \begin{bmatrix} 0.4255 \\ 0.4375 \end{bmatrix}$$

$$F(\mathbf{x}_3^t) = \frac{1}{2}(\mathbf{x}_3^t)^T \begin{bmatrix} 2 & 0 \\ 0 & 50 \end{bmatrix} \mathbf{x}_3^t = \frac{1}{2}\begin{bmatrix} 0.4255 & 0.4375 \end{bmatrix} \begin{bmatrix} 2 & 0 \\ 0 & 50 \end{bmatrix} \begin{bmatrix} 0.4255 \\ 0.4375 \end{bmatrix} = 4.966$$

This is less than $F(\mathbf{x}_2)$. Therefore this step is accepted, the momentum is reset to its original value, and the learning rate is increased.

$$\mathbf{x}_3 = \mathbf{x}_3^t, \gamma = 0.2, \alpha = \eta\alpha = 1.5(0.0375) = 0.05625$$

This completes the third iteration.

P12.4 Recall the example from Chapter 9 that we used to demonstrate the conjugate gradient algorithm (page 9-18):

$$F(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} \mathbf{x},$$

with initial guess

$$\mathbf{x}_0 = \begin{bmatrix} 0.8 \\ -0.25 \end{bmatrix}.$$

Perform one iteration of the conjugate gradient algorithm. For the linear minimization use interval location by function evaluation and interval reduction by the Golden Section search.

The gradient of this function is

$$\nabla F(\mathbf{x}) = \begin{bmatrix} 2x_1 + x_2 \\ x_1 + 2x_2 \end{bmatrix}.$$

As with steepest descent, the first search direction for the conjugate gradient algorithm is the negative of the gradient:

$$\mathbf{p}_0 = -\mathbf{g}_0 = -\nabla F(\mathbf{x})^T\Big|_{\mathbf{x}=\mathbf{x}_0} = \begin{bmatrix} -1.35 \\ -0.3 \end{bmatrix}.$$

For the first iteration we need to minimize $F(\mathbf{x})$ along the line

$$\mathbf{x}_1 = \mathbf{x}_0 + \alpha_0 \mathbf{p}_0 = \begin{bmatrix} 0.8 \\ -0.25 \end{bmatrix} + \alpha_0 \begin{bmatrix} -1.35 \\ -0.3 \end{bmatrix}.$$

The first step is interval location. Assume that the initial step size is $\varepsilon = 0.075$. Then the interval location would proceed as follows:

$$F(a_1) = F\left(\begin{bmatrix} 0.8 \\ -0.25 \end{bmatrix}\right) = 0.5025,$$

$$b_1 = \varepsilon = 0.075,\ F(b_1) = F\left(\begin{bmatrix} 0.8 \\ -0.25 \end{bmatrix} + 0.075\begin{bmatrix} -1.35 \\ -0.3 \end{bmatrix}\right) = 0.3721$$

$$b_2 = 2\varepsilon = 0.15,\ F(b_2) = F\left(\begin{bmatrix} 0.8 \\ -0.25 \end{bmatrix} + 0.15\begin{bmatrix} -1.35 \\ -0.3 \end{bmatrix}\right) = 0.2678$$

$$b_3 = 4\varepsilon = 0.3,\ F(b_3) = F\left(\begin{bmatrix} 0.8 \\ -0.25 \end{bmatrix} + 0.3\begin{bmatrix} -1.35 \\ -0.3 \end{bmatrix}\right) = 0.1373$$

$$b_4 = 8\varepsilon = 0.6,\ F(b_4) = F\left(\begin{bmatrix} 0.8 \\ -0.25 \end{bmatrix} + 0.6\begin{bmatrix} -1.35 \\ -0.3 \end{bmatrix}\right) = 0.1893.$$

Since the function increases between two consecutive evaluations we know that the minimum must occur in the interval $[0.15, 0.6]$. This process is illustrated by the open blue circles in Figure P12.4, and the final interval is indicated by the large open black circles.

The next step in the linear minimization is interval reduction using the Golden Section search. This proceeds as follows:

$$c_1 = a_1 + (1-\tau)(b_1 - a_1) = 0.15 + (0.382)(0.6 - 0.15) = 0.3219,$$

$$d_1 = b_1 - (1-\tau)(b_1 - a_1) = 0.6 - (0.382)(0.6 - 0.15) = 0.4281,$$

$$F_a = 0.2678,\ F_b = 0.1893,\ F_c = 0.1270,\ F_d = 0.1085.$$

Since $F_c > F_d$, we have

$$a_2 = c_1 = 0.3219,\ b_2 = b_1 = 0.6,\ c_2 = d_1 = 0.4281$$

$$d_2 = b_2 - (1-\tau)(b_2 - a_2) = 0.6 - (0.382)(0.6 - 0.3219) = 0.4938,$$

$$F_a = F_c = 0.1270,\ F_c = F_d = 0.1085,\ F_d = F(d_2) = 0.1232.$$

This time $F_c < F_d$, therefore

$$a_3 = a_2 = 0.3219,\ b_3 = d_2 = 0.4938,\ d_3 = c_2 = 0.4281,$$

$$c_3 = a_3 + (1-\tau)(b_3 - a_3) = 0.3219 + (0.382)(0.4938 - 0.3219) = 0.3876,$$

$$F_b = F_d = 0.1232,\ F_d = F_c = 0.1085,\ F_c = F(c_3) = 0.1094.$$

This routine continues until $b_{k+1} - a_{k+1} < tol$. The black dots in Figure P12.4 indicate the location of the new interior points, one for each iteration of the procedure. The final point is indicated by a blue dot. Compare this result with the first iteration shown in Figure 9.10.

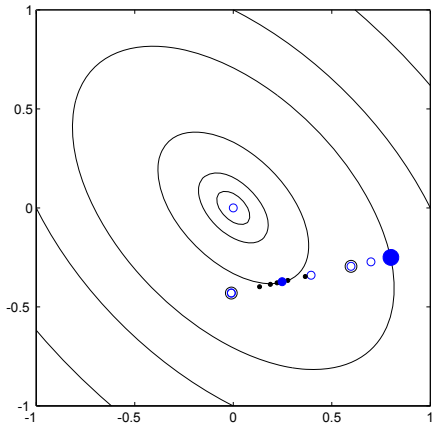


Figure P12.4 Linear Minimization Example

P12.5 To illustrate the computation of the Jacobian matrix for the Levenberg-Marquardt method, consider using the network of Figure P12.5 for function approximation. The network transfer functions are chosen to be

$$f^1(n) = (n)^2,\ f^2(n) = n.$$

Therefore their derivatives are

$$\dot{f}^1(n) = 2n,\ \dot{f}^2(n) = 1.$$

Assume that the training set consists of

$$\{(\mathbf{p}_1 = \begin{bmatrix}1\end{bmatrix}),(\mathbf{t}_1 = \begin{bmatrix}1\end{bmatrix})\} \ , \ \{(\mathbf{p}_2 = \begin{bmatrix}2\end{bmatrix}),(\mathbf{t}_2 = \begin{bmatrix}2\end{bmatrix})\} \ ,$$

and that the parameters are initialized to

$$\mathbf{W}^1 = \begin{bmatrix}1\end{bmatrix} , \ \mathbf{b}^1 = \begin{bmatrix}0\end{bmatrix} , \ \mathbf{W}^2 = \begin{bmatrix}2\end{bmatrix} , \ \mathbf{b}^1 = \begin{bmatrix}1\end{bmatrix} .$$

Find the Jacobian matrix for the first step of the Levenberg-Marquardt method.

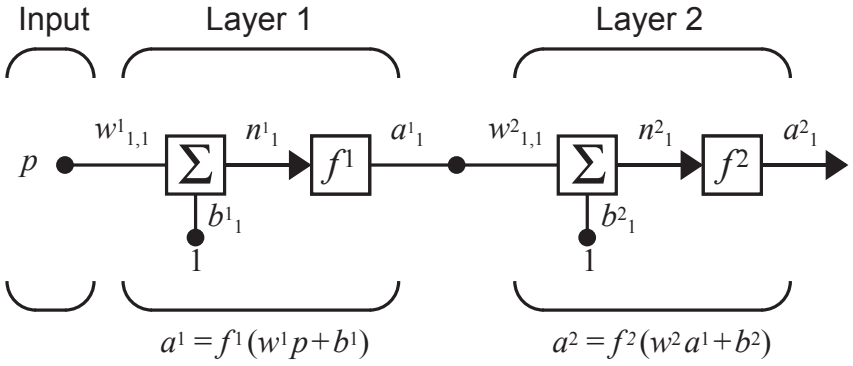


Figure P12.5 Two-Layer Network for LMBP Demonstration

The first step is to propagate the inputs through the network and compute the errors.

$$\mathbf{a}_1^0 = \mathbf{p}_1 = \begin{bmatrix}1\end{bmatrix}$$

$$\mathbf{n}_1^1 = \mathbf{W}^1\mathbf{a}_1^0 + \mathbf{b}^1 = \begin{bmatrix}1\end{bmatrix}\begin{bmatrix}1\end{bmatrix} + \begin{bmatrix}0\end{bmatrix} = \begin{bmatrix}1\end{bmatrix} , \ \mathbf{a}_1^1 = \mathbf{f}^1(\mathbf{n}_1^1) = (\begin{bmatrix}1\end{bmatrix})^2 = \begin{bmatrix}1\end{bmatrix}$$

$$\mathbf{n}_1^2 = \mathbf{W}^2\mathbf{a}_1^1 + \mathbf{b}^2 = (\begin{bmatrix}2\end{bmatrix}\begin{bmatrix}1\end{bmatrix} + \begin{bmatrix}1\end{bmatrix}) = \begin{bmatrix}3\end{bmatrix} , \ \mathbf{a}_1^2 = \mathbf{f}^2(\mathbf{n}_1^2) = (\begin{bmatrix}3\end{bmatrix}) = \begin{bmatrix}3\end{bmatrix}$$

$$\mathbf{e}_1 = (\mathbf{t}_1 - \mathbf{a}_1^2) = (\begin{bmatrix}1\end{bmatrix} - \begin{bmatrix}3\end{bmatrix}) = \begin{bmatrix}-2\end{bmatrix}$$

$$\mathbf{a}_2^0 = \mathbf{p}_2 = \begin{bmatrix}2\end{bmatrix}$$

$$\mathbf{n}_2^1 = \mathbf{W}^1\mathbf{a}_2^0 + \mathbf{b}^1 = \begin{bmatrix}1\end{bmatrix}\begin{bmatrix}2\end{bmatrix} + \begin{bmatrix}0\end{bmatrix} = \begin{bmatrix}2\end{bmatrix} , \ \mathbf{a}_2^1 = \mathbf{f}^1(\mathbf{n}_2^1) = (\begin{bmatrix}2\end{bmatrix})^2 = \begin{bmatrix}4\end{bmatrix}$$

$$\mathbf{n}_2^2 = \mathbf{W}^2\mathbf{a}_2^1 + \mathbf{b}^2 = (\begin{bmatrix}2\end{bmatrix}\begin{bmatrix}4\end{bmatrix} + \begin{bmatrix}1\end{bmatrix}) = \begin{bmatrix}9\end{bmatrix} , \ \mathbf{a}_2^2 = \mathbf{f}^2(\mathbf{n}_2^2) = (\begin{bmatrix}9\end{bmatrix}) = \begin{bmatrix}9\end{bmatrix}$$

$$\mathbf{e}_2 = (\mathbf{t}_2 - \mathbf{a}_2^2) = (\begin{bmatrix}2\end{bmatrix} - \begin{bmatrix}9\end{bmatrix}) = \begin{bmatrix}-7\end{bmatrix}$$

The next step is to initialize and backpropagate the Marquardt sensitivities using Eq. (12.46) and Eq. (12.47).

$$\tilde{\mathbf{S}}_1^2 = -\dot{\mathbf{F}}^2(\mathbf{n}_1^2) = -\begin{bmatrix}1\end{bmatrix}$$

$$\tilde{\mathbf{S}}_1^1 = \dot{\mathbf{F}}^1(\mathbf{n}_1^1)(\mathbf{W}^2)^T\tilde{\mathbf{S}}_1^2 = \begin{bmatrix}2n_{1,1}^1\end{bmatrix}\begin{bmatrix}2\end{bmatrix}\begin{bmatrix}-1\end{bmatrix} = \begin{bmatrix}2(1)\end{bmatrix}\begin{bmatrix}2\end{bmatrix}\begin{bmatrix}-1\end{bmatrix} = \begin{bmatrix}-4\end{bmatrix}$$

$$\tilde{\mathbf{S}}_2^2 = -\dot{\mathbf{F}}^2(\mathbf{n}_2^2) = -\begin{bmatrix}1\end{bmatrix}$$

$$\tilde{\mathbf{S}}_2^1 = \dot{\mathbf{F}}^1(\mathbf{n}_2^1)(\mathbf{W}^2)^T\tilde{\mathbf{S}}_2^2 = \begin{bmatrix}2n_{1,2}^2\end{bmatrix}\begin{bmatrix}2\end{bmatrix}\begin{bmatrix}-1\end{bmatrix} = \begin{bmatrix}2(2)\end{bmatrix}\begin{bmatrix}2\end{bmatrix}\begin{bmatrix}-1\end{bmatrix} = \begin{bmatrix}-8\end{bmatrix}$$

$$\mathbf{S}^1 = \begin{bmatrix}\tilde{\mathbf{S}}_1^1 \middle| \tilde{\mathbf{S}}_2^1\end{bmatrix} = \begin{bmatrix}-4 & -8\end{bmatrix}, \mathbf{S}^2 = \begin{bmatrix}\tilde{\mathbf{S}}_1^2 \middle| \tilde{\mathbf{S}}_2^2\end{bmatrix} = \begin{bmatrix}-1 & -1\end{bmatrix}$$

We can now compute the Jacobian matrix using Eq. (12.43), Eq. (12.44) and Eq. (12.37).

$$\mathbf{J}(\mathbf{x}) = \begin{bmatrix}\frac{\partial v_1}{\partial x_1} & \frac{\partial v_1}{\partial x_2} & \frac{\partial v_1}{\partial x_3} & \frac{\partial v_1}{\partial x_4} \\ \frac{\partial v_2}{\partial x_1} & \frac{\partial v_2}{\partial x_2} & \frac{\partial v_2}{\partial x_3} & \frac{\partial v_2}{\partial x_4}\end{bmatrix} = \begin{bmatrix}\frac{\partial e_{1,1}}{\partial w_{1,1}^1} & \frac{\partial e_{1,1}}{\partial b_1^1} & \frac{\partial e_{1,1}}{\partial w_{1,1}^2} & \frac{\partial e_{1,1}}{\partial b_1^2} \\ \frac{\partial e_{1,2}}{\partial w_{1,1}^1} & \frac{\partial e_{1,2}}{\partial b_1^1} & \frac{\partial e_{1,2}}{\partial w_{1,1}^2} & \frac{\partial e_{1,2}}{\partial b_1^2}\end{bmatrix}$$

$$[\mathbf{J}]_{1,1} = \frac{\partial v_1}{\partial x_1} = \frac{\partial e_{1,1}}{\partial w_{1,1}^1} = \frac{\partial e_{1,1}}{\partial n_{1,1}^1} \times \frac{\partial n_{1,1}^1}{\partial w_{1,1}^1} = \tilde{s}_{1,1}^1 \times \frac{\partial n_{1,1}^1}{\partial w_{1,1}^1} = \tilde{s}_{1,1}^1 \times a_{1,1}^0$$

$$= (-4)(1) = -4$$

$$[\mathbf{J}]_{1,2} = \frac{\partial v_1}{\partial x_2} = \frac{\partial e_{1,1}}{\partial b_1^1} = \frac{\partial e_{1,1}}{\partial n_{1,1}^1} \times \frac{\partial n_{1,1}^1}{\partial b_1^1} = \tilde{s}_{1,1}^1 \times \frac{\partial n_{1,1}^1}{\partial b_1^1} = \tilde{s}_{1,1}^1 = -4$$

$$[\mathbf{J}]_{1,3} = \frac{\partial v_1}{\partial x_3} = \frac{\partial e_{1,1}}{\partial n_{1,1}^2} \times \frac{\partial n_{1,1}^2}{\partial w_{1,1}^2} = \tilde{s}_{1,1}^2 \times \frac{\partial n_{1,1}^2}{\partial w_{1,1}^2} = \tilde{s}_{1,1}^2 \times a_{1,1}^1 = (-1)(1) = -1$$

$$[\mathbf{J}]_{1,4} = \frac{\partial v_1}{\partial x_4} = \frac{\partial e_{1,1}}{\partial n^2_{1,1}} \times \frac{\partial n^1_{1,1}}{\partial b^2_1} = \tilde{s}^2_{1,1} \times \frac{\partial n^2_{1,1}}{\partial b^2_1} = \tilde{s}^2_{1,1} = -1$$

$$[\mathbf{J}]_{2,1} = \frac{\partial v_2}{\partial x_1} = \frac{\partial e_{1,2}}{\partial n^1_{1,2}} \times \frac{\partial n^1_{1,2}}{\partial w^1_{1,1}} = \tilde{s}^1_{1,2} \times \frac{\partial n^1_{1,2}}{\partial w^1_{1,1}} = \tilde{s}^1_{1,2} \times a^0_{1,2} = (-8)(2) = -16$$

$$[\mathbf{J}]_{2,2} = \frac{\partial v_2}{\partial x_2} = \frac{\partial e_{1,2}}{\partial b^1_1} = \frac{\partial e_{1,2}}{\partial n^1_{1,2}} \times \frac{\partial n^1_{1,2}}{\partial b^1_1} = \tilde{s}^1_{1,2} \times \frac{\partial n^1_{1,2}}{\partial b^1_1} = \tilde{s}^1_{1,2} = -8$$

$$[\mathbf{J}]_{2,3} = \frac{\partial v_2}{\partial x_3} = \frac{\partial e_{1,2}}{\partial n^2_{1,2}} \times \frac{\partial n^2_{1,2}}{\partial w^2_{1,1}} = \tilde{s}^2_{1,2} \times \frac{\partial n^2_{1,2}}{\partial w^2_{1,1}} = \tilde{s}^2_{1,2} \times a^1_{1,2} = (-1)(4) = -4$$

$$[\mathbf{J}]_{2,4} = \frac{\partial v_2}{\partial x_4} = \frac{\partial e_{1,2}}{\partial b^2_1} = \frac{\partial e_{1,2}}{\partial n^2_{1,2}} \times \frac{\partial n^2_{1,2}}{\partial b^2_1} = \tilde{s}^2_{1,2} \times \frac{\partial n^2_{1,2}}{\partial b^2_1} = \tilde{s}^2_{1,2} = -1$$

Therefore the Jacobian matrix is

$$\mathbf{J}(\mathbf{x}) = \begin{bmatrix} -4 & -4 & -1 & -1 \\ -16 & -8 & -4 & -1 \end{bmatrix}.$$

Epilogue

One of the major problems with the basic backpropagation algorithm (steepest descent backpropagation — SDBP) has been the long training times. It is not feasible to use SDBP on practical problems, because it can take weeks to train a network, even on a large computer. Since backpropagation was first popularized, there has been considerable work on methods to accelerate the convergence of the algorithm. In this chapter we have discussed the reasons for the slow convergence of SDBP and have presented several techniques for improving the performance of the algorithm.

The techniques for speeding up convergence have fallen into two main categories: heuristic methods and standard numerical optimization methods. We have discussed two heuristic methods: momentum (MOBP) and variable learning rate (VLBP). MOBP is simple to implement, can be used in batch mode or incremental mode and is significantly faster than SDBP. It does require the selection of the momentum coefficient, but γ is limited to the range $[0, 1]$ and the algorithm is not extremely sensitive to this choice.

The VLBP algorithm is faster than MOBP but must be used in batch mode. For this reason it requires more storage. VLBP also requires the selection of a total of five parameters. The algorithm is reasonably robust, but the choice of the parameters can affect the convergence speed and is problem dependent.

We also presented two standard numerical optimization techniques: conjugate gradient (CGBP) and Levenberg-Marquardt (LMBP). CGBP is generally faster than VLBP. It is a batch mode algorithm, which requires a linear search at each iteration, but its storage requirements are not significantly different than VLBP. There are many variations of the conjugate gradient algorithm proposed for neural network applications. We have presented only one.

The LMBP algorithm is the fastest algorithm that we have tested for training multilayer networks of moderate size, even though it requires a matrix inversion at each iteration. It requires that two parameters be selected, but the algorithm does not appear to be sensitive to this selection. The main drawback of LMBP is the storage requirement. The $\mathbf{J}^T\mathbf{J}$ matrix, which must be inverted, is $n \times n$, where n is the total number of weights and biases in the network. If the network has more than a few thousand parameters, the LMBP algorithm becomes impractical on current machines.

There are many other variations on backpropagation that have not been discussed in this chapter. Some references to other techniques are given in Chapter 19.

Further Reading

[Barn92] E. Barnard, "Optimization for training neural nets," *IEEE Trans. on Neural Networks*, vol. 3, no. 2, pp. 232–240, 1992.

A number of optimization algorithms that have promise for neural network training are discussed in this paper.

[Batt92] R. Battiti, "First- and second-order methods for learning: Between steepest descent and Newton's method," *Neural Computation*, vol. 4, no. 2, pp. 141–166, 1992.

This paper is an excellent survey of the current optimization algorithms that are suitable for neural network training.

[Char92] C. Charalambous, "Conjugate gradient algorithm for efficient training of artificial neural networks," *IEE Proceedings*, vol. 139, no. 3, pp. 301–310, 1992.

This paper explains how the conjugate gradient algorithm can be used to train multilayer networks. Comparisons are made to other training algorithms.

[Fahl88] S. E. Fahlman, "Faster-learning variations on back-propagation: An empirical study," In D. Touretsky, G. Hinton & T. Sejnowski, eds., *Proceedings of the 1988 Connectionist Models Summer School*, San Mateo, CA: Morgan Kaufmann, pp. 38–51, 1988.

The QuickProp algorithm, which is described in this paper, is one of the more popular heuristic modifications to backpropagation. It assumes that the error curve can be approximated by a parabola, and that the effect of each weight can be considered independently. QuickProp provides significant speedup over standard backpropagation on many problems.

[HaMe94] M. T. Hagan and M. Menhaj, "Training feedforward networks with the Marquardt algorithm," *IEEE Transactions on Neural Networks*, vol. 5, no. 6, 1994.

This paper describes the use of the Levenberg-Marquardt algorithm for training multilayer networks and compares the performance of the algorithm with variable learning rate backpropagation and conjugate gradient. The Levenberg-Marquardt algorithm is faster, but requires more storage.

[Jaco88] R. A. Jacobs, “Increased rates of convergence through learning rate adaptation,” *Neural Networks*, vol. 1, no. 4, pp. 295–308, 1988.

This is another early paper discussing the use of variable learning rate backpropagation. The procedure described here is called the delta-bar-delta learning rule, in which each network parameter has its own learning rate that varies at each iteration.

[NgWi90] D. Nguyen and B. Widrow, “Improving the learning speed of 2-layer neural networks by choosing initial values of the adaptive weights,” *Proceedings of the IJCNN*, vol. 3, pp. 21–26, July 1990.

This paper describes a procedure for setting the initial weights and biases for the backpropagation algorithm. It uses the shape of the sigmoid transfer function and the range of the input variables to determine how large the weights should be, and then uses the biases to center the sigmoids in the operating region. The convergence of backpropagation is improved significantly by this procedure.

[RiIr90] A. K. Rigler, J. M. Irvine and T. P. Vogl, “Rescaling of variables in back propagation learning,” *Neural Networks*, vol. 4, no. 2, pp. 225–230, 1991.

This paper notes that the derivative of a sigmoid function is very small on the tails. This means that the elements of the gradient associated with the first few layers will generally be smaller that those associated with the last layer. The terms in the gradient are then scaled to equalize them.

[Scal85] L. E. Scales, *Introduction to Non-Linear Optimization*. New York: Springer-Verlag, 1985.

Scales has written a very readable text describing the major optimization algorithms. The book emphasizes methods of optimization rather than existence theorems and proofs of convergence. Algorithms are presented with intuitive explanations, along with illustrative figures and examples. Pseudocode is presented for most algorithms.

[Shan90] D. F. Shanno, "Recent advances in numerical techniques for large-scale optimization," *Neural Networks for Control*, Miller, Sutton and Werbos, eds., Cambridge MA: MIT Press, 1990.

This paper discusses some conjugate gradient and quasi-Newton optimization algorithms that could be used for neural network training.

[Toll90] T. Tollenaere, "SuperSAB: Fast adaptive back propagation with good scaling properties," *Neural Networks*, vol. 3, no. 5, pp. 561–573, 1990.

This paper presents a variable learning rate backpropagation algorithm in which different learning rates are used for each weight.

[VoMa88] T. P. Vogl, J. K. Mangis, A. K. Zigler, W. T. Zink and D. L. Alkon, "Accelerating the convergence of the backpropagation method," *Biological Cybernetics.*, vol. 59, pp. 256–264, Sept. 1988.

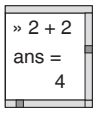
This was one of the first papers to introduce several heuristic techniques for accelerating the convergence of backpropagation. It included batching, momentum and variable learning rate.

Exercises

E12.1 We want to train the network shown in Figure E12.1 on the training set

$$\left\{(\mathbf{p}_1 = \begin{bmatrix}-2\end{bmatrix}), (\mathbf{t}_1 = \begin{bmatrix}0.8\end{bmatrix})\right\}, \left\{(\mathbf{p}_2 = \begin{bmatrix}2\end{bmatrix}), (\mathbf{t}_2 = \begin{bmatrix}1\end{bmatrix})\right\},$$

where each pair is equally likely to occur.



Write a MATLAB M-file to create a contour plot for the mean squared error performance index.

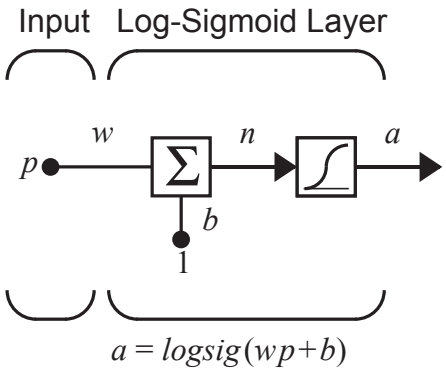


Figure E12.1 Network for Exercise E12.1

E12.2 Demonstrate the effect of batching by computing the direction of the initial step for SDBP with and without batching for the problem described in Exercise E12.1, starting from the initial guess

$$w(0) = 0, \; b(0) = 0.5.$$

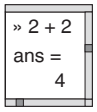
E12.3 Recall the quadratic function used in Problem P9.1:

$$F(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T \begin{bmatrix}10 & -6\\ -6 & 10\end{bmatrix}\mathbf{x} + \begin{bmatrix}4 & 4\end{bmatrix}\mathbf{x}.$$

We want to use the steepest descent algorithm with momentum to minimize this function.

i. Suppose that the learning rate is $\alpha = 0.2$. Find a value for the momentum coefficient γ for which the algorithm will be stable. Use the ideas presented in Problem P12.2.

ii. Suppose that the learning rate is $\alpha = 20$. Find a value for the momentum coefficient γ for which the algorithm will be stable.



iii. Write a MATLAB program to plot the trajectories of the algorithm for the α and γ values of both part (i) and part (ii) on the contour plot of $F(\mathbf{x})$, starting from the initial guess

$$\mathbf{x}_0 = \begin{bmatrix} -1 \\ -2.5 \end{bmatrix}.$$

E12.4 Consider the following quadratic function.

$$F(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T \begin{bmatrix} 3 & -1 \\ -1 & 3 \end{bmatrix} \mathbf{x} + \begin{bmatrix} 4 & -4 \end{bmatrix} \mathbf{x}.$$

We want to use the steepest descent algorithm with momentum to minimize this function.

i. Perform two iterations (finding $\mathbf{x}_1$ and $\mathbf{x}_2$) of steepest descent with momentum, starting from the initial condition $\mathbf{x}_0 = \begin{bmatrix} 0 & 0 \end{bmatrix}^T$. Use a learning rate of $\alpha = 1$ and a momentum coefficient of $\gamma = 0.75$.

ii. Is the algorithm stable with this learning rate and this momentum? Use the ideas presented in Problem P12.2.

iii. Would the algorithm be stable with this learning rate, if the momentum were zero?

E12.5 Consider the following quadratic function.

$$F(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T \begin{bmatrix} 3 & 1 \\ 1 & 3 \end{bmatrix} \mathbf{x} + \begin{bmatrix} 1 & 2 \end{bmatrix} \mathbf{x} + 2.$$

We want to use the steepest descent algorithm with momentum to minimize this function.

i. Suppose the learning rate is $\alpha = 1$. Is the algorithm stable, if the momentum coefficient is $\gamma = 0$? Use the ideas presented in Problem P12.2.

ii. Suppose the learning rate is $\alpha = 1$. Is the algorithm stable, if the momentum coefficient is $\gamma = 0.6$?

E12.6 Consider the following quadratic function.

$$F(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\begin{bmatrix}2 & 1\\ 1 & 2\end{bmatrix}\mathbf{x} + \begin{bmatrix}1 & 2\end{bmatrix}\mathbf{x} + 2 .$$

We want to use the steepest descent algorithm with momentum to minimize this function. Suppose the learning rate is $\alpha = 1$. Find a value for the momentum coefficient γ so that the algorithm will be stable. Use the ideas presented in Problem Eq. P12.2.

E12.7 For the function of Exercise E12.3, perform three iterations of the variable learning rate algorithm, with initial guess

$$\mathbf{x}_0 = \begin{bmatrix}-1\\ -2.5\end{bmatrix}.$$

Plot the algorithm trajectory on a contour plot of $F(\mathbf{x})$. Use the algorithm parameters

$$\alpha = 0.4 , \gamma = 0.1 , \eta = 1.5 , \rho = 0.5 , \zeta = 5\% .$$

E12.8 Consider the following quadratic function:

$$F(\mathbf{x}) = x_1^2 + 2x_2^2 .$$

Perform three iterations of the variable learning rate algorithm, with initial guess

$$\mathbf{x}_0 = \begin{bmatrix}0\\ -1\end{bmatrix}.$$

Use the algorithm parameters

$$\alpha = 1 , \gamma = 0.2 , \eta = 1.5 , \rho = 0.5 , \zeta = 5\% .$$

(Count an iteration each time the function is evaluated after the initial guess.)

E12.9 For the function of Exercise E12.3, perform one iteration of the conjugate gradient algorithm, with initial guess

$$\mathbf{x}_0 = \begin{bmatrix}-1\\ -2.5\end{bmatrix}.$$

For the linear minimization use interval location by function evaluation and interval reduction by the Golden Section search. Plot the path of the search on a contour plot of $F(\mathbf{x})$.

E12.10 Consider the following quadratic function.

$$F(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\begin{bmatrix}4 & 0\\ 0 & 2\end{bmatrix}\mathbf{x} + \begin{bmatrix}-2 & -1\end{bmatrix}\mathbf{x}.$$

We want to minimize this function along the line

$$\mathbf{x} = \begin{bmatrix}0\\ 0\end{bmatrix} + \alpha\begin{bmatrix}1\\ 1\end{bmatrix}.$$

i. Sketch this line in the x_1, x_2 plane.

ii. The learning rate α must fall somewhere between 0 and 3. Perform one iteration of the golden section search. You should find a_2, b_2, c_2 and d_2, and indicate these points along the line that you drew in part i.

E12.11 Consider the following quadratic function.

$$F(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\begin{bmatrix}1 & 1\\ 1 & 1\end{bmatrix}\mathbf{x} + \begin{bmatrix}1 & 1\end{bmatrix}\mathbf{x}.$$

We want to minimize this function along the line

$$\mathbf{x} = \begin{bmatrix}0\\ 0\end{bmatrix} + \alpha\begin{bmatrix}-1\\ 0\end{bmatrix}.$$

i. Use the method described on page 12-16 to determine an initial interval containing the minimum. Use $\varepsilon = 0.5$.

ii. Take one iteration of the golden section search to reduce the interval you obtained in part i.

E12.12 Consider the following quadratic function.

$$F(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\begin{bmatrix}1 & 0\\ 0 & 2\end{bmatrix}\mathbf{x}.$$

We want to minimize this function along the line

$$\mathbf{x} = \begin{bmatrix} 1 \\ 1 \end{bmatrix} + \alpha \begin{bmatrix} 1 \\ -1 \end{bmatrix}.$$

Perform two iterations of the Golden Section search ($k = 1, 2$) to find the interval $[a_3, b_3]$. Assume that the initial interval is defined by $a_1 = 0$ and $b_1 = 1$. Make a rough sketch of the contour plot of $F(\mathbf{x})$, draw the search line in the same figure and indicate your search points (points where you evaluated $F(\mathbf{x})$) on the line.

E12.13 Consider the following quadratic function.

$$F(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T \begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix} \mathbf{x}.$$

We want to minimize this function along the line

$$\mathbf{x} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} + \alpha \begin{bmatrix} 1 \\ -1 \end{bmatrix}.$$

Perform two iterations of the Golden Section search ($k = 1, 2$) to find the interval $[a_3, b_3]$. Assume that the initial interval is defined by $a_1 = 0$ and $b_1 = 1$. Make a rough sketch of the contour plot of $F(\mathbf{x})$, draw the search line in the same figure and indicate your search points (points where you evaluated $F(\mathbf{x})$) on the line.

E12.14 We want to use the network of Figure E12.2 to approximate the function

$$g(p) = 1 + \sin\left(\frac{\pi}{4}p\right) \text{ for } -2 \le p \le 2.$$

The initial network parameters are chosen to be

$$\mathbf{w}^1(0) = \begin{bmatrix} -0.27 \\ -0.41 \end{bmatrix}, \mathbf{b}^1(0) = \begin{bmatrix} -0.48 \\ -0.13 \end{bmatrix}, \mathbf{w}^2(0) = \begin{bmatrix} 0.09 & -0.17 \end{bmatrix}, \mathbf{b}^2(0) = \begin{bmatrix} 0.48 \end{bmatrix}.$$

To create the training set we sample the function $g(p)$ at the points $p = 1$ and $p = 0$. Find the Jacobian matrix for the first step of the LMBP algorithm. (Some of the information you will need has been computed in the example starting on page 11-14.)

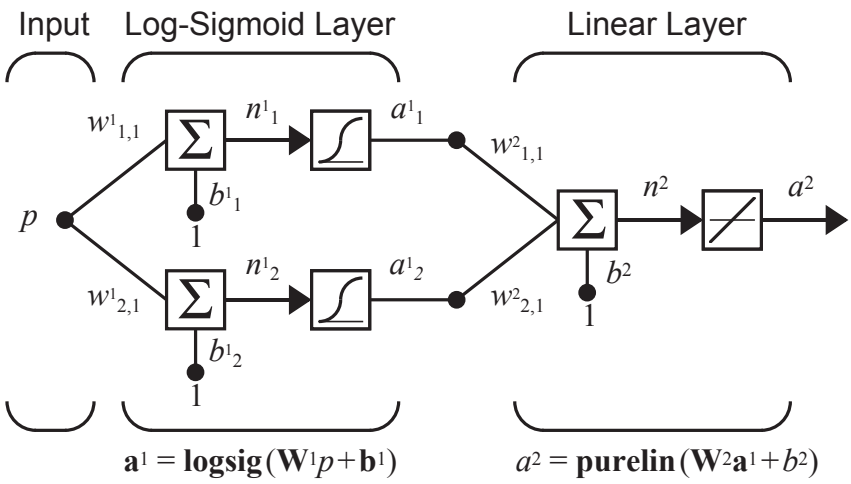


Figure E12.2 Network for Exercise E12.14

E12.15 Show that for a linear network the LMBP algorithm will converge to an optimum solution in one iteration if $\mu = 0$.

E12.16 In Exercise E11.25 you wrote a MATLAB program to implement the SDBP algorithm for a $1 - S^1 - 1$ network, and trained the network to approximate the function

$$g(p) = 1 + \sin\left(\frac{\pi}{4}p\right) \text{ for } -2 \le p \le 2 .$$

```
» 2 + 2
ans =
     4
```

Repeat this exercise, modifying your program to use the training procedures discussed in this chapter: batch mode SDBP, MOBP, VLBP, CGBP and LMBP. Compare the convergence results of the various methods.