8 The Error Surface

Because the network output is a function of its weights, the error is a function of $\mathbf{w}$. In general, $E(\mathbf{w})$ is a multidimensional function and impossible to visualize. If it could be plotted as a function of $\mathbf{w}$, however, E might look like a landscape with hills and valleys, high where E is high and low where E is low. Back-propagation, as an approximation to gradient descent, could then be viewed as placing a marble at some random point on the landscape and letting it roll downhill. If the surface were shaped like a smooth bowl, the marble (the weight state) would always roll to the lowest point; back-propagation would always find the best solution and local minima would never be a problem. Usually, of course, the surface is not so simple. Because the shape of the error surface has a fundamental effect on the learning process, it is useful to examine some of its properties. Many of the figures that follow are adapted from Hush, Horne, and Salas [183, 181].

8.1 Characteristic Features

Stair-Steps For classification problems the error surface often has a "stair-step" quality with flat regions separated by steep cliffs (figure 8.1a). The stair-step shape can arise because samples in finite training sets are sparse and because the classifier output changes sharply at a decision boundary in the input space. The decision boundary moves in the input space as the weights change but the error remains constant until the boundary crosses over a training sample and alters its classification. Either the reclassified sample is now classified correctly and the error drops a step, or the sample is now classified incorrectly and the error jumps a step. The $E(\mathbf{w})$ surface thus has flat areas where E doesn't change, separated by vertical steps where E changes discontinuously as the boundary crosses over a sample in the input space.

As the number of samples increases, the steps become more numerous and closer together (figure 8.1b); from a distance, the surface appears smoother. With continuous training data (samples available everywhere), many of the flat areas may disappear. The error can change continuously even if the node nonlinearity is a step function because the volume of positive and negative samples changes continuously as the boundary moves. Discontinuities may still occur though at points in the $E(\mathbf{w})$ space where the system decision boundary is parallel to and crosses a true boundary in the data.

The $E(\mathbf{w})$ surface also becomes smoother as the node nonlinearities of the classifier become smoother. With linear threshold units (step function nonlinearities) the plateaus of the error surface are truly flat and the steps between plateaus are truly discontinuous. When the step functions are replaced by smoother functions such as sigmoids, the steps are rounded and error surface is smoother. Figure 8.2 shows the smoothing effect of using a lower gain sigmoid. Indeed, one of the main reasons for using sigmoids rather than

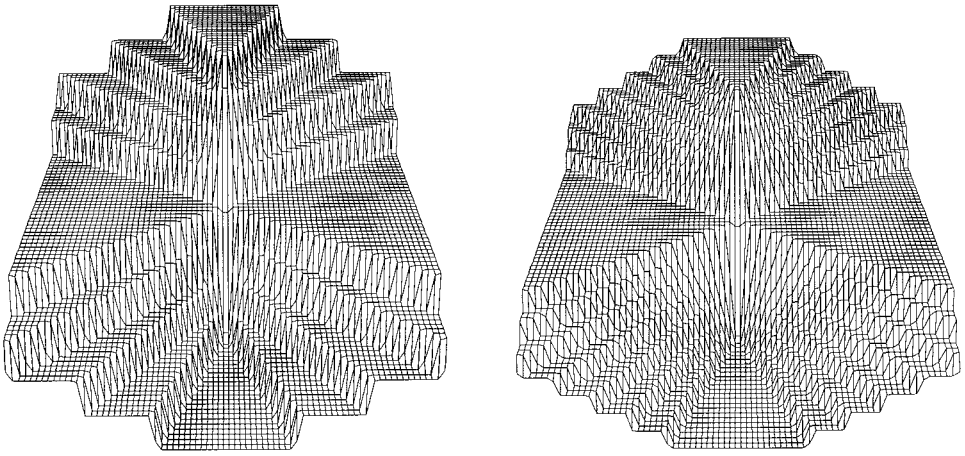


Figure 8.1
(a) The error surface of a classifier often has many flat plateaus separated by steep cliffs. (b) Increasing the number of samples creates more steps and moves them closer together (adapted from [181, 183]).

step functions is that the error surface becomes continuous so gradient based optimization methods can be used. Since gain scaling is equivalent to scaling all the node input weights by a constant factor (smaller weights correspond to smaller gains) this provides support for the heuristic of initializing with small weights.

The orientation and placement of a node hyperplane depends only on the ratio of its weights (section 3.1). As all weights are scaled equivalently, the location of the hyperplane stays fixed but the steepness of the sigmoid transition varies (increasing with the magnitude of the weight vector). For the system as a whole, the input-output transfer function is defined by cells bounded by the node hyperplanes; as all weights are scaled equivalently, the cell boundaries remain fixed but the steepness of the boundary transitions change. For small scale factors (small weights), the sigmoids have small slopes and the boundary transition regions may extend across entire cells, effectively smoothing over the stair steps. As the scale factor becomes large (large weights), the boundary transition regions shrink, the cell interiors flatten, and the steps become sharper.

Radial Features For classification problems, the preceding means the $E(\mathbf{w})$ surface often has a radial or "star" topology because scaling all weights equivalently corresponds to moving along a radial line in weight space from the origin to infinity.

The surface is not truly "star-shaped" because the error can change nonmonotonically along a radial line in the region near the origin. Past a certain radius, however, the

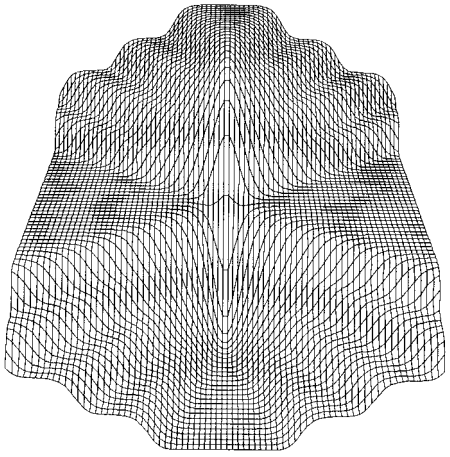


Figure 8.2
With a smaller tanh gain (0.1 in this case), the step transitions are smoother (cf. figure 8.1a). Gain scaling is equivalent to scaling all weights by a constant factor though so this does not change the basic shape of the error surface. In the figure, it corresponds to zooming in for a closer view of the origin.

classifications cease changing as the weights increase further. Once the scale factor is large enough, the classifications remain essentially constant and the error changes very little as the weight state moves along a line to infinity.

For {0, 1} training targets (or $\{-1, +1\}$ targets for tanh node functions), the minimum error on the line often occurs at infinity because the target values are reachable only by making the weights approach infinity. (This is generally true for single layer networks and linearly separable data; there may be exceptions for multilayer networks, data sets which are not linearly separable, or data sets for which the optimal outputs are not 0 and 1 even though the target values are.) The error surface therefore often has rays or troughs extending radially from the origin with minima (or maxima) at infinity. Because the sigmoid slopes are extremely small in the saturation region, the slope along the bottom of the trough is also very small. Although it is not visible in figure 8.1a, there is a trough along the center of the lowest plateau.

Replacing the {0, 1} targets with $\{0.1, 0.9\}$ values may move the minima in from infinity, but this may also introduce new minima in the form of small dips at the bottom of each cliff (figure 8.3); these are usually shallow and narrow, however. Consider how the error varies as a sigmoid is shifted sideways by varying the threshold. As the 0.9 part of the sigmoid passes over a 0.9 target, the error for that sample goes through zero. This creates

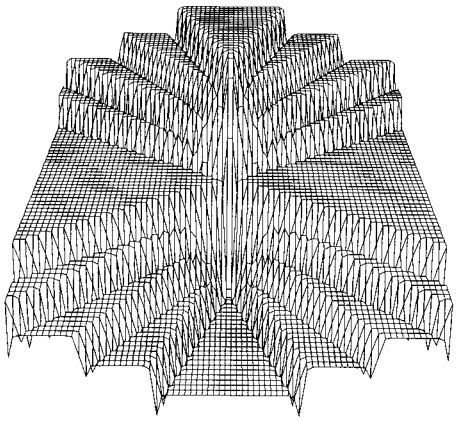


Figure 8.3
Replacing the $\{-1, 1\}$ targets with $\{-0.9, 0.9\}$ values produces a small dip at the bottom of each cliff. For illustration purposes, the tanh gain was reduced to $1/2$ to make the dip wider and smaller targets were used to make it deeper.

a local minimum (if other samples are sufficiently far away) because the error increases as the sigmoid is shifted to either side around this point. In the two-dimensional plots, this appears as a small gutter or trough along the bottom of each step. Similar effects can also occur at step tops in networks with hidden layers. One way to suppress the gutter is to change the error function so that outputs greater than 0.9 (for target values $t = 1$) and less than 0.1 (for $t = 0$ and sigmoid nodes) do not contribute to the error [181, 183]. (In section 8.5 this is called the LMS-threshold error function.) This could introduce truly flat plateau regions, however, causing problems for gradient-based training methods.

Troughs and Ridges More significant troughs and ridges occur when the classifier cannot completely separate the training samples. Figure 8.4 shows the error surface for the two-weight classifier given a training set that is not linearly separable. The input data is one-dimensional (points on a line). As the threshold weight varies, the sigmoid shifts along the input axis and the error increases and then decreases again as the decision boundary crosses individual samples. This occurs for all values of the gain weight so the result is a trough in the error surface. A gradient based optimization method could easily get stuck in one of these troughs and so converge to a poor solution.

It is interesting to note that, in figure 8.4 at least, the troughs come together at the origin. This supports the idea of initializing with small weights, that is, near the origin, where all troughs (including the main basin) are reachable in just a few steps. Although true gradient following methods would not be able to escape from a poor trough, ap-

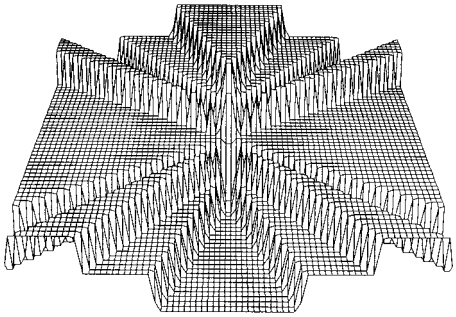


Figure 8.4
When the samples are not linearly separable, the error surface has radial troughs and ridges. A gradient based optimization method could easily get stuck in one of the troughs corresponding to a poor solution (adapted from [181, 183]).

proximations such as on-line or batch back-propagation with a noninfinitesimal step size would have an appreciable chance if better alternatives are sufficiently close. Of course, we should not jump to conclusions based on this one example; in many problems the origin is a local minimum and for these it may be better to initialize at some intermediate distance.

8.2 The Gradient is the Sum of Single-Pattern Gradients

With an SSE or MSE cost function, the $E(w)$ surface is the sum (or average) of the individual surfaces for each pattern and the total gradient is the sum (or average) of the single-pattern gradients. In other words, the error is shaped by the interaction of the weights with each of the individual training patterns. Figure 8.5 shows single-pattern gradients for a simple two-weight problem. These are the vectors that would be used for weight updates in on-line learning. On a "hillside" (a), most of the vectors point in a dominant direction. On a "ridge" (b) or at the bottom of a "valley" (c), there are often two bundles of vectors pointing in opposite directions across the valley. In on-line learning, the weights are updated from just one pattern and thus tend to oscillate across the valley. At a local minima (d) the vectors sum to zero; they may be large and distributed in all directions, or they may all go to zero. If they simply cancel without going to zero, the minimum will be unstable with on-line learning—the weight vector will move off the minimum if placed there. Point (e) shows a relatively "flat spot." These examples aren't universal since similar $E(w)$ features could be created in many ways, but they are common. Other cost functions may yield different behavior.

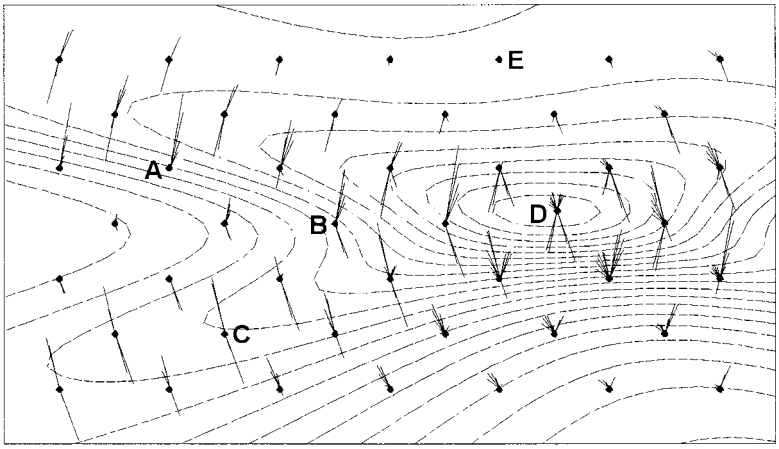


Figure 8.5
Single-pattern weight update vectors. With an SSE cost function, the $E(w)$ surface is the sum of individual surfaces for each pattern and the total gradient is the sum of the single-pattern gradients. On a "hillside" (a), most of the vectors point in a dominant direction. On a "ridge" (b) or at the bottom of a "valley" (c), there are often two bundles of vectors pointing in opposite directions. At a local minima (d) the vectors sum to zero; they may be large and evenly distributed in all directions, or they may all go to zero. Point (e) shows a relatively flat spot.

8.3 Weight-Space Symmetries

Consider a network with two or more nodes in a hidden layer. The network output is unchanged when all weights into and out of two hidden nodes, i and j, are swapped; node i computes what node j used to and vice versa so the effect on the rest of the net is unchanged. Equivalently, the node indexes could just be swapped or the locations in the layer could be exchanged. There are $H!$ permutations for the position of H nodes in a hidden layer, so this gives $H!$ different weight vectors that produce equivalent input-output network functions. An immediate consequence of this is that the error surface will not have a single global minimum (unless it is the zero vector); there will be many points with equally small errors.

Another symmetry results because the tanh function is odd, $f(-x) = -f(x)$. An equivalent response can be obtained by changing the sign of all the weights into and out of a hidden node since changing the sign of the input weights simply changes the sign of the node output and the effect on the following layer can be compensated by also changing the sign of the output weights. Similar symmetries exist for networks of sigmoid nodes if the biases of nodes in the following layer are adjusted when the signs are flipped because

$$\text{sigmoid}(x) = 1 - \text{sigmoid}(-x).$$

Any combination of the hidden nodes may have their signs flipped, so there are 2^H possibilities.

Together, these give $M = 2^H H!$ different weight vectors that produce identical input-output functions. For every weight vector that produces a particular input-output function, there are at least $2^H H! - 1$ "twins" that produce equivalent responses. H does not have to be large for this to be a huge number, e.g., for $H = 10$, $M = 3.7$ billion.

For networks with more than one hidden layer, the number of symmetries is a product of similar terms for each layer [78]

$$M = \prod_{\ell} 2^{H_\ell} H_\ell! \tag{8.1}$$

where ℓ indexes the hidden layers and H_ℓ is the number of nodes in layer ℓ.

Hecht-Nielsen [162] asked whether these symmetries exhaust the possibilities. Sussmann [362] showed that, aside from these symmetries, the weights of a feedforward single-hidden-layer network with tanh nodes are uniquely determined by the input-output map, provided that the network is irreducible (i.e., that no nodes can be removed without affecting the output). The results have been extended to reducible networks with more general node nonlinearities [232].

Hecht-Neilsen [163] showed that these symmetries give the weight space a structure of cone or wedge-shaped regions that differ only by symmetry. The cones are otherwise identical so each contains weight vectors for every input-output function the network can implement. In principle, a training system could restrict search to a single cone and still cover all possible input-output functions. Because M can be very large, this could reduce the size of the search space by a huge amount. Unfortunately, the remaining space is still huge. There might be some benefit for nonlocal methods such as the genetic algorithm as this would limit redundancy in the search. (An empirical test using a simulated annealing method on the 2-input XOR problem showed a reduction of search time by about 1/2 [198].) For local search (e.g., gradient) techniques, however, there is no good reason to stay inside a single cone because, after all, the cones are identical. It might also seem counterproductive because the introduction of the cone boundary as a hard constraint could give rise to additional poor local minima at the boundary. The cone boundaries are natural divisions, however, because of symmetry so pure gradient descent naturally stays in its starting cone [164] and there is no need for special measures to restrict the weight vector to a single cone.

8.4 Remarks

The efficiency of any optimization method depends on having a good fit between the basic assumptions of the algorithm and the actual characteristics of the function being minimized. Many advanced optimization methods assume the error surface is locally quadratic, for example, and may not do well on the "cliffs-and-plateaus" surface common in neural network classifiers. In this case, the quadratic assumption is not reasonable on a large scale so these optimizers may be no more efficient than simpler methods in finding a good approximate solution. The assumption, however, *is* usually reasonable near a minimum, in which case these methods may be very efficient for final tuning of a near-solution found by other methods.

For back-propagation, a large learning rate is needed to make progress across the large flat regions. But near the "cliffs" where $\|\partial E/\partial \mathbf{w}\|$ is large, a small learning rate is necessary to prevent huge weight changes in essentially random directions. If a fixed learning rate is used, the value will have to be a compromise. One of the advantages of the common technique of initializing with small random weights is that the system starts in the area near the origin where the error surface is smoother and it has a better chance of finding the right trough.

Before ending this discussion, it should be noted that the error surfaces illustrated in the figures are for classification problems with small training sets. The error surfaces may be very different for regression problems with large sample sizes. Based on figure 8.1, it is reasonable to expect it to be smoother.

For regression problems where the target is a continuous function of its inputs, smooth input-output functions are usually preferred. If there is sufficient data that the system cannot fit every point exactly, then it must approximate multiple points by fitting a surface "close" to them. For many cost functions, this surface can be thought of as the local average of nearby points and will tend to be smooth because of the smoothing effects of averaging. Because smoother functions generally correspond to smaller weights, the good minima will usually be in the interior of the weight space rather than at infinity. Similarly, in classification problems with many samples in overlapping clusters, it may be better to form gradual transitions between classes in regions where they overlap. This again corresponds to smaller weights and moves the minima in from infinity.

Of course, if there is so little data and the network is so powerful that it can fit every point exactly, then there is no reason to expect it to form a smooth function, and minima at infinity may survive. Even when a smooth function would be preferable, local minima at infinity may survive corresponding to fitting a few of the points exactly while ignoring the rest; these are likely to be shallow and narrow, however, and will probably be shadowed by better

minima closer to the origin. Although plateaus and cliffs will be apparent at large distances from the origin, this may be irrelevant because those regions will never be investigated by the learning algorithm.

The stair-step shape may survive in very underconstrained networks that can essentially classify each of the training points internally (e.g., by assigning a hidden "grandmother node" to each training sample). In this case, the global minima of the training set error would be at infinity and low generalization-error areas corresponding to smooth functions are unlikely to be minima.

8.5 Local Minima

Like all local search techniques, back-propagation may converge to a local minimum of the error rather than the global optimum. This is a fundamental characteristic of local search methods, not a defect of the implementation.

Random Restarts One of the simplest ways to deal with local minima is to train many different networks with different initial weights. Training times can be long, so it may be useful to train a number of networks for a small number of iterations and then choose a fraction of the best for more extensive training. Although there are no guarantees, this tends to weed out bad starting points, favoring those already close to good solutions. Because many of the initial networks may converge toward equivalent solutions, a clustering procedure may be useful to choose a nonredundant subset.

If training is viewed as a dynamic process, each minimum (local or global) can be considered an attractor with a basin of attraction consisting of all the starting points that converge to it during training. With gradient descent, an initial weight vector will basically "roll downhill" until it reaches a minima; the minimum is the attractor and all points that flow into it form the basin of attraction. In figure 8.6, the dotted line separates basins of attraction of the local and global minima. Because the probability that a randomly selected starting point converges to a particular minimum is equal to the probability that it starts in its basin of attraction, the success of the random restart approach will depend on the relative sizes of the basins of attraction.

In theory, if we knew the relative sizes of the basins of attraction (and the distribution of the initial weights), we would be able to calculate how many random restarts would be needed to achieve a certain probability of landing in the basin of a global minima. Easy problems would have large global basins of attraction and hard problems would have small basins. In general, however, it is not practical to map the basins of attraction because evaluation of each point requires training a network to convergence. These ideas

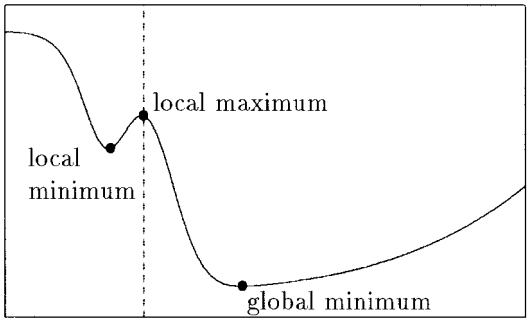


Figure 8.6
Local and global minima. A global minimum of a function can be defined as its lowest point, the input that gives the lowest possible output. A local minimum is a point that is lower than all its neighbors, but higher than the global minimum.

are, therefore, more useful as an explanation of why a problem is hard than as a tool to improve training.

A further complication is that the basins of attraction depend on the weight-change method as well as the static $E(\mathbf{w})$ function so different learning algorithms may have different attractors for the same $E(\mathbf{w})$ function. With gradient descent, for example, several local minima may have large regions of attraction. In principle, these shrink to infinitesimal points with simulated annealing and almost all points belong to the basin of attraction of the global minimum. Simple parameter changes in things like the learning rate and momentum also affect the boundaries. In gradient descent with a small step size, the basins are generally contiguous and have smooth boundaries. When the learning rate is too high, however, the process can become chaotic and the basins of attraction may become disjoint with very complex, possibly fractal, boundaries [216, 217].

With stochastic algorithms the discrete basins of attraction are replaced by probability density functions. For every ending point z there is a density function over x measuring the probability that starting point x ends at z. Loosely speaking, the regions of high probability can be thought of as the basin of attraction of z.

The standard solution to the problem of local minima is to improve the optimization algorithm. In the random restart approach, the optimization routine is augmented with an outer loop searching over initial starting points —simple random search in this case. This will not be effective if the global attractor basins are too small to find by random sampling. It is possible to consider more sophisticated outer search techniques, but use of such techniques is not widespread. Another approach is to use stochastic search methods by introducing randomness at a lower algorithmic level. Simple examples include online

training, training with added input noise (jitter), or adding noise to the weights during updates. If these do not work, more sophisticated global search techniques such as simulated annealing or genetic algorithms may be needed.

Instead of improving the optimization algorithm, an alternative approach is to avoid creating local minima in the first place. If we knew more about their causes, we might be able to design networks and training algorithms without poor local minima. Unfortunately, relatively little is known. The following paragraphs outline a few results.

8.5.1 Single-Layer Nets Can Have Local Minima

Single-layer networks with linear outputs implement linear functions and minimum-MSE linear regression has a quadratic error surface with a single minimum so one might guess that single-layer networks do not have local minima. This ignores effects of the node nonlinearity.

Sontag and Sussmann [352, 353] (see also [408]) give conditions where this intuition is true. They show that if (1) the cost function does not penalize overclassification and (2) the data are linearly separable, then there are no local minima that are not global minima and gradient descent converges (to within a tolerance) globally from any starting point in a finite number of steps. Condition 1 means that the error is taken to be zero when the output "goes beyond what the target asks." With tanh nodes, for example, the error is taken to be zero when $y \geq 0.9$ for $t = 0.9$ or when $y \leq -0.9$ for $t = -0.9$, for target t and output y. Similar relations apply for sigmoids and other nonlinearities. This has been called the *LMS-threshold cost function*. Use of unobtainable targets, for example, 0 and 1 for the sigmoid, might be considered a special case.

There can be local minima when either of the two conditions fail. Dips like those in figure 8.3 can occur when condition 1 is not satisfied and troughs as in figure 8.4 may exist because the data is not linearly separable.

Auer, Herbster, and Warmuth [13] show that local minima can exist in a network consisting of a single neuron whenever the composition of the loss function (error function) with the node activation function is continuous and bounded. This result favors the entropic error function because the sigmoid and MSE error function meet the criterion and the sigmoid and entropic error function do not. Reference [13] also shows that artificial data sets can be constructed in which the number of minima grows exponentially with the input dimension.

It has been pointed out [58, 59] that one can find linearly separable data sets for which the MSE cost function has minima at weight vectors that do not separate all patterns correctly. The perceptron algorithm would classify these data correctly whereas back-propagation would not. This does not necessarily mean the MSE function has local minima though, it just shows that it differs from the function that counts the number of classification errors.

The results just discussed show that gradient descent will separate linearly separable data sets if condition 1 is satisfied (i.e., if overclassification is not penalized).

8.5.2 No Local Minima for Linearly Separable Data

Similar results for one-hidden-layer networks have been shown in [145, 127]. If the data are linearly separable, the network has a pyramidal structure after the inputs and a full-rank weight matrix, and the LMS-threshold cost function is used then no local minima exist and back-propagation will separate the data. A distinction is made between spurious local minima, which can be eliminated by the use of the LMS-threshold error function, and more serious structural local minima.

Together, these results show that there need not be local minima for linearly separable data sets. That is, there may be local minima in some set-ups, but steps can be taken to eliminate them. An important condition appears to be the use of the LMS-threshold error or the use of unobtainable targets, for example, 0 and 1 for the sigmoid, which eliminate the spurious local minima.

Of course, it is well-known that the perceptron learning algorithm will always find a set of weights that correctly classifies a linearly-separable data set. Optimal convergence of on-line back-propagation for linearly separable data, akin to the perceptron convergence property, is shown in [144] for a single-hidden-layer network using the LMS-threshold error function. The demonstration is similar to the perceptron convergence theorem. A side point of the paper is that on-line learning can be qualitatively different from batch-mode learning when the learning rate is not small and it is not necessarily a crude approximation of gradient descent.

These results are intriguing but, unfortunately, most interesting problems are not linearly separable. It is worth noting that a data set can sometimes be made linearly separable by changing the way variables are represented, but there are many other issues to consider.

Auer, Herbster, and Warmuth [13] show that a single-layer net has no poor local minima when the transfer function and loss function are monotonic and the data are realizable ($E(\mathbf{w}) = 0$ for some weights $\mathbf{w}$). This is not restricted to binary targets; linearly separable data sets with binary targets are realizable, but there are also realizable data sets with continuous targets.

8.5.3 Local Minima Really Do Exist

It has been suggested that (nearly) flat areas in error surface are sometimes mistaken for minima and that true local minima might be relatively rare. That is, the error may decrease so slowly as the weight vector creeps across a flat spot that it appears as if learning has stopped because the system has reached a minimum. Internally, it might be possible

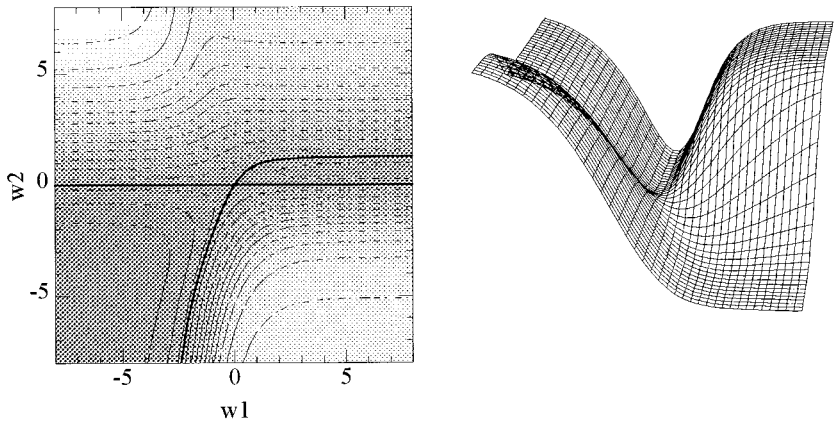
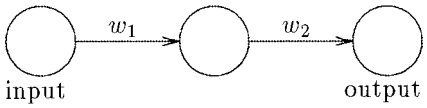


Figure 8.7
The error surface for a 1/1/1 network trained on the identity mapping illustrates that local minima do in fact exist. This surface has a good minima in the lower left quadrant, a poor minimum in the upper right quadrant, and a saddle point at the origin: (a) the contour plot, and (b) a view looking across the origin down the axis of the poor minimum.

to observe the weights moving in a consistent direction but from the outside it may be impossible to tell the difference. The fact that these are not minima is sometimes shown because the error resumes its decrease (sometimes sharply) if training is continued long enough for the system to cross the flat area. (Figures 6.4 and 6.12 show examples of this.)

Example It is easy to illustrate local minima in small networks. Figure 8.4 shows one case. An example of a very simple network having local minima is described by McClelland and Rumelhart [261]. The task is to do an identity mapping with a 1/1/1 network. The network has one input, one hidden unit, and one output. No bias weights are used so there are only two weights. The nonlinearities are sigmoids.



Given 0 or 1 as an input, the net should reproduce the value at the output. Figure 8.7 shows the error as a function of the weights w_1 and w_2. There is a global minimum in the lower left quadrant, a poor local minimum in the upper right quadrant, and a saddle point at the origin separating the two basins of attraction. The saddle point is visible in figure 8.7b. Although the poor minimum appears to be narrower in the contour plot, both have basins

of attraction of approximately equal size and a random weight vector has a roughly equal chance of landing in either.

This example is contrived, of course. When adaptable bias weights are added, the local minimum disappears leaving two global minima separated by a saddle point. Use of tanh nodes and $\{-1, +1\}$ inputs would also remove the asymmetry causing the problem.

The existence of real local minima in nontrivial networks was demonstrated by McInerney et al. [263, 262]. (These are an abstract and an unpublished technical report. Hecht-Nielsen [162] reports some of the details.) After extensive simulations using closed-form expressions for the gradients and second derivatives, they were able to find a point on the error surface where the error was higher than at other points, where all gradients were zero, and where the Hessian was strongly positive-definite.

8.5.4 The Effect of Network Size

Aside from the fact that the architecture is poorly matched to the problem, the 1/1/1 network in the previous example is very small. The common wisdom is that if there are enough weights and/or hidden units then local minima do not exist or are not a major problem. Indeed, when bias weights are added to the simple network above, the local minimum disappears. McClelland and Rumelhart [261: 132] state, "[i]n problems with many hidden units, local minima seem quite rare." The extra degrees of freedom presumably provide more ways for potential local minima to flow into lower areas and eventually reach a global minimum. Of course, there are no guarantees as it is always possible to add extra degrees of freedom in ways that do not fix the problem.

A series of papers [301, 413, 150, 414, 415] have shown that if there are as many hidden nodes as there are training patterns then, with probability 1, there are no suboptimal local minima. According to Yu [413], a sufficient condition is that the network be able to fit every training pattern exactly, giving $E_{min} = 0$. Assuming consistent training data, this is always possible in a single-hidden-layer net when the number H of hidden nodes is as large as the number M of unique training patterns (actually $H \geq M - 1$ will do). Because most data sets contain regularities, fewer nodes will suffice in most cases; however, $M - 1$ hidden nodes in a single hidden layer will be *necessary* when the patterns are colinear with alternating target classes [280].

According to these results, the error surface will have no nonglobal minima if there are enough hidden nodes. This helps explain empirical observations that it is often easier to train larger networks than it is to train small ones. Larger networks seem to be less sensitive to parameters and less likely to become stuck during training. Of course, the requirement that the network be able to fit the data exactly allows it to "memorize" the data and conflicts with heuristic rules for obtaining good generalization so steps such as early stopping or pruning will be needed to avoid overfitting.

8.6 Properties of the Hessian Matrix

The Hessian, **H**, of the error with respect to the weights is the matrix of second derivatives with elements

$$h_{ij} = \frac{\partial^2 E}{\partial w_i \, \partial w_j}.$$

Knowledge about the Hessian is useful for a number of reasons:

• The convergence of many optimization algorithms is governed by characteristics of the Hessian matrix. In second-order optimization methods, the matrix is used explicitly to calculate search directions. In other cases, it may have an implicit role. Slow convergence of gradient descent, for example, can often be explained as an effect of an ill-conditioned Hessian (see appendix A.2). The eigenvalues of **H** also determine how large the learning rate can be before learning becomes unstable. When **H** is known, an optimal rate can be chosen; in other cases, a more conservative choice must be made.

• Some pruning algorithms use Hessian information to decide which weights to remove. "Optimal brain damage" uses a diagonal approximation while "Optimal brain surgeon" uses the full approximation.

• The inverse Hessian can be used to calculate confidence intervals for the network outputs [44: 399].

• Hessian information can be used to calculate regularization parameters [44, section 10.4].

• Hessian information can be used for fast retraining of an existing network when additional training data becomes available [41].

• At a local minimum of the error function, the Hessian will be positive definite. This provides a way to determine if learning has stopped because the network reached a true minimum or because it "ran out of gas" on a flat spot.

Bishop [44, section 4.10] and Buntine and Weigend [62] provide summaries of a number of methods for calculation and approximation of the Hessian in neural networks. (The preceding list is partly drawn from [44, section 4.10].)

Sometimes the Hessian is required only as a means to obtain the product **Hv** for some vector **v**. Pearlmutter [297] describes a fast method for finding this product which does not require evaluation of the Hessian.

8.6.1 Ill-Conditioning

An *ill-conditioned* matrix has a large ratio $|\lambda_{max}/\lambda_{min}|$, where λ_{max} is the largest eigenvalue and λ_{min} is the smallest (nonzero) eigenvalue. In the Hessian of the error with respect

to the weights, this reflects the fact that the gradient changes slowly along one direction (determined by the eigenvector associated with λ_{min}) and changes rapidly along another direction (determined by the eigenvector associated with λ_{max}). The effect this has on gradient descent is discussed in section 10.5.1. Briefly, it requires that a small learning rate be used to avoid instability along the quickly changing direction, but then progress along the slowly changing direction will be slow and convergence times will be long. Appendix A.2 discusses the convergence rate of gradient descent in linear problems.

It appears that the Hessian is very often ill-conditioned in neural network training problems [37, 331, 92, 93] and it is common for many eigenvalues to be near-zero. Sigmoid saturation may cause effective loss of rank. Intuitively, small eigenvalues can be related to flat regions of the $E(\mathbf{w})$ surface where the error changes very slowly in most directions. It is worth noting that the outer-product approximation (see the following) will be rank-deficient when the number of weights exceeds the number of training patterns. This is not necessarily recommended, but not uncommon in neural networks.

An implication of ill-conditioning is that the expected efficiencies of higher order optimization methods may not be realized [331]. Ill-conditioning could lead to numerical instability or very large steps that take the system out of the region where the local approximation is valid. This is a well-known problem with Newton's method, for example, which is addressed by standard methods. The fix used in the Levenberg-Marquardt method is to choose the search direction based on a combination of the gradient and Newton directions.

Many of the techniques suggested for accelerating back-propagation are simple algorithmic modifications that optimize certain steps of the procedure without addressing the fundamental problem of ill-conditioning. A more basic way to decrease training time is to modify the problem so that the Hessian is better conditioned. The effectiveness of some common techniques can be explained in terms of their effect on the Hessian:

- normalization of inputs to zero-mean values with similar variances,
- use of bias nodes (which remove the mean internally),
- use of tanh nonlinearities rather than logistic sigmoids (because the outputs of tanh nodes tend to be zero-mean when their inputs are zero-mean whereas the outputs of sigmoid nodes always have a positive mean), and
- preprocessing (e.g., principal components analysis) to remove input redundancy.

8.6.2 Exact Calculation of the Hessian

Bishop, [41, 39] and [44, section 4.10.5], describes an $O(W^2)$ method for calculating the exact Hessian of a feedforward network. The procedure is somewhat involved for a general

feedforward network so the simplified version [44, section 4.10.6] for single-hidden-layer networks is summarized here. For simplicity, the terms are given for a single pattern p; the complete expression is obtained by summing contributions from each pattern. Let indexes i and i' denote input nodes, j and j' denote hidden nodes, and k and k' denote output nodes. Each nondiagonal term of the Hessian involves two weights.

1. If both weights are in the second layer,

$$\frac{\partial^2 E_p}{\partial w_{kj}\,\partial w_{k'j'}} = z_j z_{j'} \delta_{kk'} H_{kk} \tag{8.2}$$

where z_j is the output of the jth node, $\delta_{kk'}$ is the Kronecker delta, and H_{kk} is defined at the end of the section.

2. If both weights are in the first layer,

$$\begin{aligned}\frac{\partial^2 E_p}{\partial w_{ji}\,\partial w_{j'i'}} = {} & x_i x_{i'} f''(a_{j'}) \delta_{jj'} \sum_k w_{kj'} \delta_k \\ & + x_i x_{i'} f'(a_{j'}) f'(a_j) \sum_k w_{kj'} w_{kj} H_{kk}\end{aligned} \tag{8.3}$$

where x_i is the ith input, a_j is the weight sum into the jth hidden node, $f()$ is the node nonlinearity function, and $\delta_k = \partial E_p / \partial a_k$ is the delta term calculated by back-propagation.

3. If one weight is in each layer,

$$\frac{\partial^2 E_p}{\partial w_{ji}\,\partial w_{kj'}} = x_i f'(a_j) \left\{ \delta_k \delta_{jj'} + z_{j'} w_{kj} H_{kk} \right\}. \tag{8.4}$$

The term $H_{kk'}$

$$H_{kk'} = \frac{\partial^2 E_p}{\partial a_k\,\partial a_{k'}}$$

describes how errors in different output nodes interact to affect the overall error. For the sum-of-squares error function and *linear* output units, $H_{kk'} = \delta_{kk'}$.

8.6.3 Approximations

Second order optimization methods often require the Hessian matrix or an approximation. For mean-squared-error cost functions, the outer-product approximation is a common choice.

Outer-Product Approximation For the mean-square error function, the Hessian can often be approximated reasonably well by the average of outer-products of the gradient vectors. This approximation is used by the Gauss-Newton optimization method (section 10.6.2).

The cost function E is the mean squared error between the desired output d and the actual output y which is a function of the weights w_k

$$E = \left\langle (d - y)^2 \right\rangle . \tag{8.5}$$

The $\langle\ \rangle$ brackets denote the average over the training set. For simplicity, assume y is a scalar. A single weight index (e.g., w_j) is used here since it is not necessary to identify the weight by the nodes it connects. The gradient $\mathbf{g}$ has components

$$g_j = \frac{\partial E}{\partial w_j} = 2 \left\langle (d - y) \frac{\partial y}{\partial w_j} \right\rangle \tag{8.6}$$

and the Hessian $\mathbf{H}$ has components

$$h_{ij} = \frac{\partial^2 E}{\partial w_i\, \partial w_j} = 2 \left\langle -\frac{\partial y}{\partial w_i} \frac{\partial y}{\partial w_j} + (d - y) \frac{\partial^2 y}{\partial w_i\, \partial w_j} \right\rangle . \tag{8.7}$$

This can be written

$$\mathbf{H} = 2(-\mathbf{P} + \mathbf{Q}) \tag{8.8}$$

where $\mathbf{P} = \left\langle \mathbf{g}\mathbf{g}^T \right\rangle$ is the average outer-product of first-order gradient terms while $\mathbf{Q}$, $q_{ij} = \left\langle (d - y) \frac{\partial^2 y}{\partial w_i\, \partial w_j} \right\rangle$ contains second order terms. Because $\mathbf{P}$ is the sum of outer products of the gradient vector, it is real, symmetric, and thus nonnegative-definite ($\mathbf{w}^T \mathbf{P} \mathbf{w} \geq 0$ for all $\|\mathbf{w}\| \not= 0$). The number of training samples must be greater than the number of weights for it to have full rank.

The outer-product approximation is based on the assumption that first-order terms dominate higher order terms near a minimum. This is reasonable when the residual errors $(d - y)$ are zero-mean, independent, identically distributed values uncorrelated with the second derivatives $\partial^2 y / (\partial w_i\, \partial w_j)$. If this is true then, when the number of training points is large, $\mathbf{Q}$ should average to zero with a small variance and so can be ignored.

Of course, this assumption is not always valid. If there are too few training points, the variance of $\mathbf{Q}$ may be large. Also, if the network is too simple to fit the data, the errors may not be small or may be correlated with the second derivatives of y. (It can be argued that the errors and second derivatives will be correlated in many cases because an overly smooth network function will tend to "shave off the peaks" and "fill in the dips" of the

target function. The function tends to peak where the target peaks and dip where it dips, but not quite enough, so at a peak of y the second derivatives are necessarily negative while the local errors $d - y$ tend to be positive.) A second caution is that the approximation may be valid only near points in the training set used to calculate the approximation. In practice, however, the approximation seems to work reasonably well especially when the larger algorithm does not rely too heavily on its accuracy.

The Diagonal Approximation A further approximation is to assume that the Hessian is diagonal. This provides at least some of the Hessian information while avoiding the $O(W^2)$ storage and calculation costs demanded by the full approximation. In reality, **H** would be diagonal only if all weights affected the error independently, that is, only if there are no significant interactions between different weights. Most networks, however, have strong weight interactions so the Hessian usually has nonnegligible off-diagonal elements. The approximation therefore is not expected to be especially accurate; its main advantage is computational convenience. Effects of the approximation are discussed by Becker and LeCun [37]. A diagonal approximation is used in the "optimal brain damage" pruning method (section 13.2.3).

The second derivatives h_{kk} can be calculated by a modified back-propagation rule in about the same amount of time as a single back-propagation epoch. From [91], the diagonal elements are

$$h_{kk} = \frac{\partial^2 E}{\partial w_{ij}^2} = \frac{\partial^2 E}{\partial a_i^2} x_j^2 \tag{8.9}$$

where a_i is the weighted sum into node i and $x_j = f(a_j)$ is the output of node j. Here the weights are indexed both by k and by the indexes i and j of the nodes they link. At the output nodes, the second derivatives are

$$\frac{\partial^2 E}{\partial a_i^2} = 2 f'(a_i)^2 - 2(d_i - x_i) f''(a_i) \tag{8.10}$$

(for all units i in the output layer). The second derivatives for internal nodes are obtained by a modified back-propagation rule

$$\frac{\partial^2 E}{\partial a_i^2} = f'(a_i)^2 \sum_{\ell} w_{\ell i}^2 \frac{\partial^2 E}{\partial a_\ell^2} - f''(a_i) \frac{\partial E}{\partial x_i}. \tag{8.11}$$

The $f''(a_i)$ terms are sometimes ignored in the last two equations. This corresponds to using the diagonal of the outer-product approximation and gives guaranteed positive estimates of the second derivatives [91].

Finite-Difference Approximation A finite-difference approximation of **H** is [44, pg. 154]

$$\frac{\partial^2 E}{\partial w_{ji}\,\partial w_{lk}} = \frac{1}{2\epsilon}\left\{\frac{\partial E}{\partial w_{ji}}(w_{lk}+\epsilon) - \frac{\partial E}{\partial w_{ji}}(w_{lk}-\epsilon)\right\} + O(\epsilon^2). \tag{8.12}$$

Weight w_{lk} is perturbed first by $+\epsilon$ and then by $-\epsilon$. The first-order derivatives for all weights w_{ji} are calculated in each case and the second derivative is approximated by the difference between the two first-order derivatives, scaled by 2ϵ. There are W weights to perturb and each gradient calculation takes $O(W)$ time so the approximation take $O(W^2)$ time. The approximation errors are of size $O(\epsilon^2)$; small ϵ values are desirable for accuracy, but larger values are desirable to avoid numerical problems.

Because of its simplicity, the finite-difference approximation is useful during debugging to verify the correctness of other evaluation methods.

8.7 Gain Scaling

The typical node function can be written

$$a_i = \sum_j w_{ij} y_j$$

$$y_i = f(\beta_i a_i)$$

where $f()$ is the node nonlinearity and β_i is a gain parameter which controls the steepness of the function at 0. For sigmoid nonlinearities, $\frac{\partial y}{\partial a} = \beta y(1-y)$ and at $a = 0$ the slope is $\beta/4$. Normally $\beta = 1$. Larger values increase the slope at 0 and narrow the width of the semilinear transition region. As $\beta \to \infty$, the response approaches a step function.

A number of studies, [196, 369] for example, have shown that every network with nonunity gains can be transformed into an equivalent network with unity gains by appropriate scaling of the weights (table 8.1). Further, if learning rates are also scaled appropriately, both networks will follow equivalent trajectories during training and produce equivalent outputs at the end of training.

Gain Control for Faster Learning In many cases, the motivation for gain scaling is to accelerate the training process. Izui and Pentland [190] show that convergence time scales like $1/\beta$ without momentum and like $1/\sqrt{\beta}$ with momentum.

Lee and Bien [237] include parameters for the slope, magnitude, and vertical offset of the sigmoid function

Table 8.1
Relationship of Node Gain, Learning Rate, and Weight Magnitude (from [369]).

	with gain β	without gain
Node function	$\phi(\beta x)$	$\phi(x)$
Gain	β	1
Learning rate	η	$\beta^2\eta$
Weights	$\mathbf{w}$	$\beta\mathbf{w}$

$$y_j = K/(1 + e^{-\beta a_j}) - L.$$

Here the gain is a fixed nonunity value. In empirical tests [9], the changes had weak effects. For $0.4 \leq \beta \leq 1.2$, learning speed and generalization increased with β, but for $\beta > 1.2$, learning became unstable "suddenly and severely" with few trials converging.

Several studies [354, 366, 312, 84] attempt to optimize the gain during training, most using gradient descent on the error. Most claim increased convergence speed and fewer problems with convergence to poor local minima. As noted, gain changes are equivalent to learning rate changes in a network without gains so optimization of gains has effects like an adaptive learning rate method. A gain change $\Delta\beta$ is equivalent to a learning rate change from $\beta^2\eta$ to $(\beta + \Delta\beta)^2\eta$ and a weight change from $\beta\mathbf{w}$ to $(\beta + \Delta\beta)\mathbf{w}$ [196, 369].

Gain Control to Prevent Sigmoid Saturation Many weight initialization heuristics involve choosing an appropriate range for the initial random weights (see chapter 7). The equivalence between scaling the weights by a constant factor and introducing a gain term in the sigmoid function means that similar results can be obtained by gain scaling. In [240, 241] initial gains are chosen to avoid sigmoid saturation and its detrimental effects on learning time.

In [411], the gain is adjusted during training to prevent sigmoid saturation. If, during training, the errors are large but the back-propagated deltas are small then all the node gains are halved and the iteration repeated.

Gain Control for Improved Generalization Gain scaling has been suggested as a way to improve generalization. In most cases, the idea is to start with small gains that increase gradually during training. This is said to be related to "continuation" or "homotopy" methods in numerical analysis. The intent is to force the system to fit large-scale features of the target function first by making it harder to fit small-scale details. The small initial gains make the network compute a smoother function than it otherwise would with the same weights and larger gains. Later, once large-scale features are learned, the gain is increased to let the system fit smaller features. The hope is that by forcing the system to start with

a smooth fit and then gradually increasing its flexibility, this will increase the chance of convergence to the global minimum.

Kruschke [228, 229, 230] describes a pruning procedure based on gain-competition (section 13.4.2). Sperduti and Starita [354] describe a similar pruning method in conjunction with the use of gain scaling for faster training.

Gain Scaling to Train Networks of Hard-Limiters In electronic circuit implementations, it is often desirable to use hard-limiting step functions for the node nonlinearity because they can be implemented with a simple switch. One way to train such networks is to gradually shift from a sigmoid to a step function during learning. (Training must be done in off-line simulations if the hardware can't implement the sigmoid.) A linear combination of the two functions

$$g(a) = \lambda f(a) + (1 - \lambda)h(a)$$

is used in [373]. Here a is the weighted-sum into the node, $f(a)$ is the sigmoid function, $h(a)$ is a step function, and λ changes from 1 to 0 linearly. A possible problem with this approach is that the $g(a)$ is still nondifferentiable at $a = 0$. Selection of the adjustment schedule for λ is another problem. Yu *et al.* [412] adjust the sigmoid gain instead, setting $\beta = 0.5e^{-\mathrm{SSE}}$ where SSE is the sum-of-squares error. Initially, when the error is large, the gain is small; later the gain increases as the error decreases. Corwin, Logar, and Oldham [86] and Yu, Loh, and Miller [412] also use gain adjustment to train networks of hard-limiters.