10 Classical Optimization Techniques

Terms like training and learning are often used in an artificial neural network context to describe the process of adapting a set of parameters (the weights) to achieve a specific technical result (minimization of the error on a training set). As such, learning can be viewed as a general optimization problem that may be addressed by numerous techniques.

Back-propagation is by far the most commonly used method for training MLP neural networks. Batch-mode back-propagation with a small learning rate (and no momentum) is a specialization of gradient descent to MLPs while on-line back-propagation is related to stochastic gradient descent. Gradient descent is not highly regarded in the optimization community, however, mainly because of its slow rate of convergence. This is a particular problem when the Hessian matrix is poorly conditioned, that is, when the gradient changes quickly in some directions and slowly in others, as it does in so-called ravines of the error surface.

Optimization is a mature field and many algorithms, some quite sophisticated, have been developed over the years. In appropriate circumstances, they may be better alternatives to back-propagation. Many converge much faster than gradient descent in certain situations while others promise a higher probability of convergence to global minima. This chapter reviews some of the standard methods. Aside from performance improvements they might provide, familiarity with these techniques is useful to understand back-propagation and its variants, many of which draw on ideas from the standard methods.

10.1 The Objective Function

To treat network training as an optimization problem, an objective function (or cost function) must be defined that provides an unambiguous numerical rating of system performance. The cost function reduces all the various good and bad aspects of a possibly complex system down to a single number, a scalar value, which allows candidate solutions to be ranked and compared. In short, it provides the working definition of optimal for the search algorithm, telling it what kinds of solutions to look for. It is important, therefore, that the function faithfully represent our design goals. If we choose a poor error function and obtain unsatisfactory results, the fault is ours for badly specifying the goal of the search.

Selection of the objective function can be a problem in itself since it is not always easy to develop a function that measures exactly what we want when goals are vague. It is often necessary to compromise between what we want, what we can measure, and what we can optimize efficiently. A few basic functions are very commonly used. The mean squared error is popular for function approximation (regression) problems because of its convenience in mathematical analysis. The cross-entropy error function is often used for classification problems when outputs are interpreted as probabilities of membership in an

indicated class. In real-world applications, it may be necessary to complicate the function with additional terms to balance conflicting subgoals or to introduce heuristics favoring preferred classes of solutions.

10.2 Factors Affecting the Choice of a Method

Once an objective function has been chosen, many optimization algorithms may be applied. A few well-known methods are mentioned in the following sections. The list is not exhaustive by any means. We attempt to describe the most popular algorithms broadly, including the main assumptions behind them and their main advantages and disadvantages for neural network training. We do not attempt to give full implementation details or to describe special cases which need to be considered in real applications. Details of classical algorithms can be found in many texts, for example [138, 311, 302].

A minor point: optimization can be viewed as maximizing utility or as minimizing cost. To simplify the discussion, we consider it as minimization.

Ideally, all optimization routines would yield the same result, the global minimum of the error function. In practice though, some methods tend to work better than others in particular situations. There is generally a trade-off between speed, robustness, and the probability of finding the global minimum. Specialized algorithms may be very fast in certain situations, but not robust in other situations. Many methods embody assumptions about the problem that allow efficiencies to be obtained when the conditions hold, but may lead the algorithm astray otherwise. Algorithms that promise to find a global minimum tend to be much slower than less ambitious methods; most are relatively robust, however, because they make few assumptions about the problem.

Selection of an appropriate algorithm depends on many factors. Some include:

Differentiability. Continuous functions allow the use of efficient gradient methods. If the function is differentiable everywhere and the derivatives are easy to evaluate, conjugate gradient methods are often recommended if local minima are not an extreme problem.

If the function is not differentiable or derivatives are not available, then less efficient evaluation-only methods may be necessary. Gradient methods can be used with gradient estimates obtained by finite-differences. If the function is known to be smooth but derivatives are not available, the conjugate-directions method is often recommended. If the function is not smooth or has many local minima, then global search methods such as simulated annealing and genetic algorithms may be required.

Classification versus regression. In classification problems, the exact numerical output is not important as long as the class is indicated unambiguously. In continuous function

approximation problems, the numerical values are important and target values must be matched closely to obtain a small error.

The advantages of methods using second-order gradient information seem to show up mostly in the latter case. That is, they excel at homing in on the exact minimum once the general basin of attraction has been found, but they are not really much better at finding the basin than other, simpler methods when the function is very nonlinear.

Local minima. If there are relatively few really poor local minima, then random restarts of a local search algorithm may be sufficient. If there are many poor local minima, consider a global search method such as simulated annealing. Stochastic algorithms like simulated annealing promise convergence to the global minimum with probability 1, but can be very slow.

Problem size. Algorithms that scale poorly may be adequate for small problems but become impractical for large networks with large training sets. Second-order methods requiring exact evaluation of the Hessian are generally impractical for large networks. Storage and manipulation of large matrices may also be a problem even for methods that approximate the Hessian.

Robustness. Does the algorithm work well when preconditions are not satisfied exactly and when parameters are not optimally tuned? In general, robust algorithms make fewer assumptions and so tend to be slower than more optimistic algorithms (which are suited to the given problem).

Meta-optimization. Back-propagation, for example, has a learning rate that must be selected; if momentum is used, it also must be given a value. Other algorithms have their own parameters that must be selected. Tuning these parameters may be a meta-optimization problem in itself and algorithms that are overly sensitive to parameter choices will be difficult to use for general problems. Meta-optimization may not be feasible for large problems where each training session takes days or weeks.

Is the data noisy? If the data is noisy and there are not enough examples to suppress noise by averaging, then the observed errors must be viewed as random estimates and precise minimization of the training error will not necessarily correspond to minimization of the true expected error.

This is a generalization problem. If the problem is addressed by adding regularization terms to the objective function, then precise minimization may have real benefits and second-order methods may have advantages. If the problem is not addressed at all, then back-propagation (and other simple gradient descent methods) may be better *because of* their relative inability to locate the precise minimum. If the problem is addressed by early stopping, then the search may never enter the precise-minimization phase where second-order methods excel.

Clustering. Is the data smoothly distributed or clustered? If the data falls in well-separated clusters that correspond to target classes, learning is usually easier because fine distinctions are not required. As long as the decision boundary falls in the open space between clusters, its exact location is not critical so more solutions are feasible. When the clusters are well separated, on-line methods may be faster than batch methods because of redundancy in the data; almost all the information is contained in the cluster centroids.

In general, optimization algorithms can be classified as deterministic or stochastic. Most deterministic optimization algorithms can be classed as evaluation-only methods or gradient-based methods. The advantage of evaluation-only methods is their simplicity; no gradient calculation is required. They are most useful where internal system details are not accessible, where the objective function is not differentiable, or in complex systems where derivatives are difficult to calculate correctly. Their main disadvantage is inefficiency. It is almost always worth using gradient information if it can be obtained because convergence is usually much faster; the gradient points in precisely the direction that increases the function the fastest, after all. Because the gradient is easily calculated (by the back-propagation step) in MLPs with differentiable node nonlinearities, most neural network training algorithms use it. Even more information is available in second-order derivatives so methods that use the Hessian matrix, or approximations to it, can be very efficient under certain conditions. They may be impractical for large problems, however, because of storage and computation requirements involved in dealing with large matrices.

A problem with deterministic algorithms is that they always converge to the same endpoint given the same starting point. The system will converge either to a local minimum or to the global minimum depending on the starting point. When local minima are rare and the basin of attraction of the global minimum is large, a few repetitions of the algorithm with different starting points may be enough to find good solutions. In other cases, more powerful methods are needed. The advantage of stochastic methods is that every state has a nonzero chance of occurrence so, if the procedure runs long enough, the global minimum will be visited eventually. Under certain conditions, many stochastic algorithms can promise a high likelihood of convergence to the global minimum. The problem is that this may take a *very* long time and guarantees are lost if the algorithm is terminated early.

10.3 Line Search

In many methods, the choice of a search direction is treated separately from the problem of how far to move along that direction. A number of methods, in fact, are defined by the way they choose the search direction and the existence of a perfect line search is assumed. Line search routines are basically one-dimensional optimization methods to find the minimum

in a given interval. In general, they are a subcomponent of a larger overall algorithm, which applies them to find the minimum along a given line in a higher dimensional space.

The efficiency of the line search routine can be critical since, in many cases, the calculations needed to compute the search direction are relatively minor and most of the computation time will be spent in the line search. In general, there is a trade-off between the efficiency of the algorithm (the number of function and gradient evaluations required) and the precision with which the minimum can be located; more precision calls for more evaluations. General multidimensional algorithms that can tolerate some inexactness in the line search routine are preferred for this reason. If a lot is known about the function then evaluation at just a few points may be enough to locate the minimum exactly; for example, 3 points may be sufficient for a quadratic function. This is unusual, however, because functions are usually not so simple. In a typical case, a line search may require 10 to 20 function (and/or gradient) evaluations depending on the nonlinearity of the function and the precision demanded. In some cases, many more evaluations may be needed.

There are many different line search methods varying in efficiency and robustness. We will not describe them here since they are covered well in many optimization texts. As in the general case, there are specialized methods that may be very efficient in situations where they are appropriate, but may not work well in other cases. Methods using gradient information are usually more efficient than evaluation-only methods if the gradient calculation is inexpensive (as it is in neural network simulations). Whether the gradient calculation is worth the savings in function evaluations can be problem-dependent, however. If gradient information is used in the line search routine, it certainly pays to also use it in the computation of the search direction.

In most methods, the first task is to find an interval that brackets a minimum. This calls for three points where the interior point is lower than either end point and, unless you are lucky, will usually require more than three function evaluations to find. Given a bounding interval, the next task is to locate the minimum. One approach is to iteratively subdivide the interval until the uncertainty is tolerable; each step reduces the uncertainty by roughly half on average. The other main approach is to approximate the function on the interval (e.g., with a parabola) and then estimate the minimum location analytically. The first method is generally more robust, but the second method can be much faster. Some practical algorithms start with the first method and then switch to the second. Brent's method [302] is a common choice.

10.4 Evaluation-Only Methods

Evaluation-only methods search for minima by comparing values of the objective function at different points. In particular, they do not use gradient information. The basic idea is to

evaluate the function at points around the current search point, looking for lower values. Algorithms differ in how test points are generated.

Simplicity is the main advantage of evaluation-only methods. Inefficiency is their main disadvantage; for smooth functions it usually pays to use gradient information if it is available. For MLPs with continuous node nonlinearities, the gradient is easily calculated in the back-propagation step. In computer simulations, the incremental cost (in time and complexity) is approximately the same as the cost of a forward evaluation.

Evaluation-only methods are most useful where internal system details are inaccessible (as in some integrated circuit implementations), where the transfer function of the system is not differentiable, or in complex systems where derivatives are difficult to calculate correctly. In spite of their inefficiency, some evaluation-only methods do have global convergence properties lacking in gradient-based methods.

10.4.1 Hooke-Jeeves Pattern Search

One of the simplest search methods is to take small steps along each coordinate direction separately, varying one parameter at a time and checking if the error decreases. If a step in one direction increases the error, then a step in the opposite direction should decrease it (if the step size is small enough). After N steps, each of the N coordinate directions will have been tested. If none of the steps decrease the error, the step size may be too big relative to the curvature of the error surface and the step size should be reduced.

Although this is simple to implement, it is very inefficient in high-dimensional spaces. For a neural network, each step would require evaluation of all the training patterns but change only a single weight. There is also a small chance of getting stuck at the bottom of an obliquely oriented 'ravine' in the error surface since the error decreases along the axis of the ravine but is higher along each of the coordinate directions so no more steps will be taken. Usually, however, the point is slightly off-center so the step size will merely reduce to a very small value resulting in slow convergence.

The Hooke-Jeeves pattern search method [311] greatly accelerates convergence in this situation by remembering previous steps and attempting new steps in the same direction. An *exploratory move* consists of a step in each of the N coordinate directions ending up at the *base point* after N steps. A *pattern move* consists of a step along the line from the previous base point to the new one. This may be oblique, in general. This becomes a temporary base point for a new exploratory move. If the exploratory move results in a lower error than the previous base point, it becomes the new base point. If it does not decrease the error, then the temporary base point is discarded and an exploratory move is done around the current base point. If this exploratory search fails, the step size is reduced. The search is halted when the step size becomes sufficiently small.

There are many heuristic variations, but simplicity is the main advantage of the method and once it's lost, more sophisticated methods are preferable.

10.4.2 Nelder-Mead Simplex Search

Another direct search method is based on the idea of using a population of points to determine the local shape of the error surface. The basic steps are:

1. Create a simplex (a regular convex polytope) in N dimensions and evaluate the function at each of the $N + 1$ vertices.
2. Identify the vertex with the highest error.
3. Reflect the vertex across the centroid of the other vertices. This tends to be a step downhill but not parallel to the gradient, in general.
4. Evaluate the function at the new point. If the error is lower, go to 2. Otherwise, reject the new point and try reflecting less far across the centroid (i.e., move the reflected point in towards the centroid).
5. Go to 2.

There are additional rules to handle some special cases. Eventually, the points straddle a local minimum and the vertices converge to a single point as the size is reduced. One problem of the basic algorithm is that the simplex could become very small if it has to "squeeze through" a tight area; convergence would be very slow from then on. The Nelder-Mead variation introduces rules for expanding and contracting the simplex to accelerate convergence and close in on the final minimum.

This method is relatively simple to implement. It is said to be reasonably efficient and tolerant of noise in the objective function. (Powell's method may be more efficient for many problems, however, although it is not as easy to implement.) Like other evaluation-only methods, it can be used on nondifferentiable functions. It may not be suitable to large neural networks though because it requires storage of approximately N^2 values ($N + 1$ vertices each specified by a vector of N weights).

In a neural network training example [103], simplex search was slow at first, but eventually reduced the error to a lower value than back-propagation. Quasi-Newton and Levenberg-Marquardt methods achieved even lower errors, however.

10.4.3 Powell's Conjugate Direction Method

Powell's conjugate direction method, sometimes called the direction set method, uses information from previous steps in order to choose the next search direction. A quadratic error function $E = \mathbf{x}^T\mathbf{H}\mathbf{x}$ is assumed, where $\mathbf{x}$ are the parameters to be optimized.

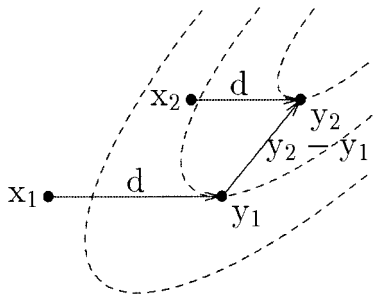

Figure 10.1
Conjugate directions. Powell's method uses the "parallel subspace property" of quadratic functions to find a set of conjugate directions without evaluating the Hessian. Given a quadratic function $\mathbf{x}^T\mathbf{H}\mathbf{x}$, a direction $\mathbf{d}$, and two points $\mathbf{x}_1$ and $\mathbf{x}_2$, if one does a line search from $\mathbf{x}_1$ along direction $\mathbf{d}$ to obtain a minimum $\mathbf{y}_1$ and another line search from $\mathbf{x}_2$ along direction $\mathbf{d}$ to obtain $\mathbf{y}_2$ then direction $(\mathbf{y}_2 - \mathbf{y}_1)$ is $\mathbf{H}$-conjugate to $\mathbf{d}$ [311].

A set of vectors $\mathbf{d}_1, \mathbf{d}_2, \ldots \mathbf{d}_r, r \leq N$, are mutually *conjugate* with respect to a symmetric $N \times N$ matrix $\mathbf{H}$ if they are linearly independent and [311]

$$\mathbf{d}_i^T \mathbf{H} \mathbf{d}_j = 0 \quad \text{for all } i \neq j. \tag{10.1}$$

Conjugate directions are useful because an N-dimensional quadratic function determined by $\mathbf{H}$ can be minimized by performing independent one-dimensional searches along any N mutually conjugate directions. The directions are uncoupled in the sense that minimizing along direction $\mathbf{d}_i$ does not undo any gains that were obtained by minimizing along previous directions. The reader is referred to an optimization text, for example, Reklaitis, Ravindras, and Ragsdell [311], for further details.

It would seem to be necessary to evaluate $\mathbf{H}$, the Hessian matrix of second derivatives, in order to find a set of conjugate directions, but in fact it is not. Powell's method exploits the parallel subspace property of quadratic functions, defined as follows, to find a set of conjugate directions without evaluating the Hessian. Given a quadratic function $\mathbf{x}^T\mathbf{H}\mathbf{x}$, a direction $\mathbf{d}$, and two points $\mathbf{x}_1$ and $\mathbf{x}_2$, if one does a line search from $\mathbf{x}_1$ along direction $\mathbf{d}$ to obtain a minimum $\mathbf{y}_1$ and another line search from $\mathbf{x}_2$ along direction $\mathbf{d}$ to obtain $\mathbf{y}_2$ then direction $(\mathbf{y}_2 - \mathbf{y}_1)$ is $\mathbf{H}$-conjugate to $\mathbf{d}$ [311] (see figure 10.1).

In N dimensions, Powell's method does N line searches to identify each conjugate direction using only function evaluations. If the error function actually is quadratic, it will find the minimum after N^2 (exact) line searches. Of course, more searches will usually be needed for general nonlinear functions and inexact calculations, but convergence is still quadratic. It is said to be as reliable as, and usually much more efficient than, other direct search methods.

For neural networks in which the gradient is easily obtained, the conjugate gradient method is preferred since it is much more efficient ($O(N)$ line searches for a quadratic function). In either case, for general nonlinear functions, the quadratic approximation is only reasonable near a minimum and other methods may be better to reach the general neighborhood in the first place.

10.5 First-Order Gradient Methods

The main disadvantage of evaluation-only methods is their relative inefficiency. When gradient information is available, it is almost always worth using because it tells exactly which parameter changes will minimize the error most at the current point. In digital simulations at least, it is easy to calculate the gradient for MLPs with differentiable node nonlinearities so most training methods are gradient based.

10.5.1 Gradient Descent

As noted, back-propagation is a variety of gradient descent. In gradient descent, new points are obtained by moving along the (negative) gradient direction. That is,

$$\mathbf{w}(t+1) = \mathbf{w}(t) - \eta \mathbf{g} \tag{10.2}$$

where $\mathbf{g} = \frac{\partial E(\mathbf{w})}{\partial \mathbf{w}}$ is the gradient, $E(\mathbf{w})$ is the error function evaluated at $\mathbf{w}(t)$ and η is a step size or learning rate parameter. For a true approximation to gradient descent, $\eta \to 0$ should be very small (figure 10.2a), but larger values are often used in practice to speed-up convergence. Batch-mode back-propagation with a small learning rate is a specialization of gradient descent to MLPs where the back-propagation step is just a way of calculating the gradient by application of the derivative chain rule in the MLP structure. Back-propagation and its variations are discussed in chapters 5 and 9.

Gradient descent is not highly regarded in the optimization community mainly because of its slow rate of convergence. In theory, the asymptotic rate of convergence is linear. That is, the error is reduced by a constant factor at each step, $|e_{k+1}| \leq C|e_k|$ where $C < 1$ is a constant. One problem is that the gradient never points to the global minimum except in the case where the error contours are spherical so many small steps are needed to arrive at the minimum. Another problem is that the gradient vanishes at a minimum so $\Delta \mathbf{w}$ approaches 0 and final convergence is very slow. Slow convergence is also a problem when the Hessian is poorly conditioned, that is, when the gradient changes rapidly in some directions but slowly in others. This is the case in so-called ravines of the error surface. When the step size is too large, the weight state may bounce from one side of the ravine to the other while making only slow progress along the ravine towards the true minimum (figure 10.2b),

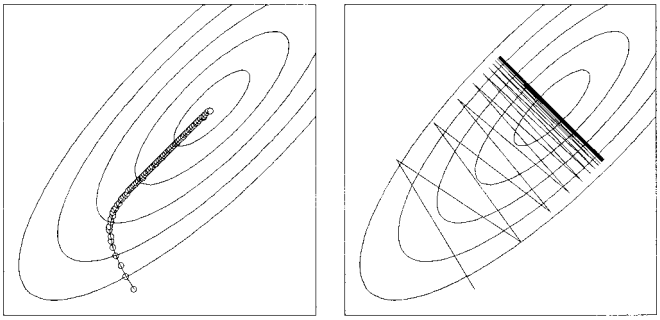

Figure 10.2
Gradient descent trajectories: (a) With a small step size (0.01), gradient descent follows a smooth trajectory, but progress may be very slow. (b) Cross-stitching: When the step size is too big (0.1) and the error surface has valleys, the trajectory may oscillate wildly from one side to the other while creeping slowly along the length of the valley to the minimum. Once it reaches the minimum, it may overshoot many times before settling.

an effect sometimes called cross-stitching. (Section A.2 discusses the convergence rate of gradient descent in linear problems.) The possibility of convergence to local minima is another common criticism applicable to all gradient-based methods.

The learning rate η is a parameter that must be selected. When it is too small, convergence will be very slow. When it is too large, the procedure may diverge (see figure 10.2). In practice, noninfinitesimal values of η are used in order to speed-up convergence and true gradient descent is only approximated. This is an unimportant detail in most cases. Stability requires that $0 < \eta < 2/\lambda_{max}$, where λ_{max} is the largest eigenvalue of the Hessian matrix. The optimum value is $\eta = 1/\lambda_{max}$ and with this choice, the convergence rate is governed by the slowest time constant $\frac{\lambda_{max}}{\lambda_{min}}$, where λ_{min} is the smallest nonzero eigenvalue (section A.2). When the Hessian is badly conditioned (i.e., nearly singular) the time constant can be large and convergence very slow. For nonlinear functions, the Hessian changes as the weight vector moves over the error surface. It is expensive to reevaluate the Hessian at each point so either a small learning rate is chosen arbitrarily or η is adjusted heuristically.

For best performance, different values of η are appropriate at different points on the error surface. Moderately large values are useful to reach the vicinity of a minimum quickly, but care has to be taken to avoid node saturation. Once a minimum has been located approximately, smaller step sizes are needed to avoid cross-stitching and allow the system to settle to a minimum. This is the situation where second-order methods usually outperform gradient descent. Effects of different learning rates on back-propagation are described in chapter 6. Variations of back-propagation that tune η dynamically are described in chapter 9.

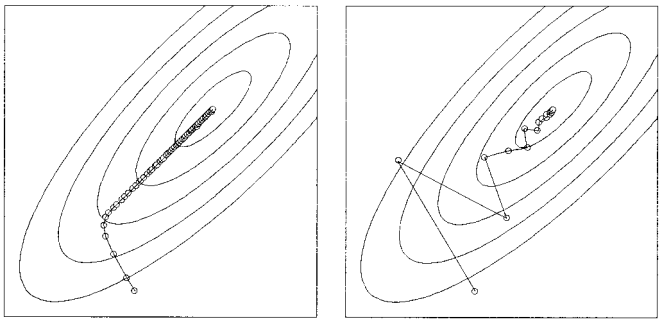

Figure 10.3
Gradient descent trajectories with momentum. With momentum, opposite (side-to-side) changes tend to cancel while complementary changes (along the length of the valley) tend to sum. The overall effect is to stabilize the oscillations and accelerate convergence: (a) When the step size is small, momentum acts to accelerate convergence (step size 0.01 and momentum 0.99, cf. figure 10.2a); (b) small amounts of momentum also help to damp oscillations when the step size is too large (step size 0.1 and momentum 0.2, cf. figure 10.2b).

The use of momentum is common in back-propagation. This can be thought of as gradient descent with smoothing controlled by the momentum parameter, $0 < \alpha < 1$. Briefly, the weight update rule is

$$\Delta \mathbf{w}(t) = -\eta \mathbf{g} + \alpha \Delta \mathbf{w}(t-1) \tag{10.3}$$

where, as before, $\mathbf{g}$ is the gradient of the error evaluated at $\mathbf{w}(t)$ and η is the step size or learning rate parameter. When the step size is too large, momentum helps to suppress cross-stitching because consecutive opposing changes tend to cancel while complementary changes sum. That is, for $\alpha \to 1^-$, the side to side oscillations across the valley effectively cancel while the components along the axis of the ravine add up. When the step size is too small, on the other hand, momentum helps accelerate learning by amplifying the effective learning rate. When the gradient is constant, for example, the effective learning rate is $\eta' = \eta/(1-\alpha)$. Figure 10.3 illustrates these effects. Momentum is discussed in more detail in section 6.2.

10.5.2 Best-Step Steepest Descent (Cauchy's Method)

Simple continuous gradient descent is the rule "wherever you are, evaluate the gradient and take a step down it." When the step size is infinitesimal, this produces a curved path that is orthogonal to the contours of the error surface at all points (figure 10.2a). This is slightly different from optimal, or best-step, steepest descent. As the term is used in the optimization community, this seems to mean Cauchy's method, for example [311], which

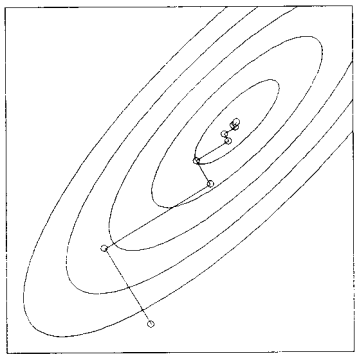

Figure 10.4
Cauchy's method uses the rule: wherever you are, evaluate the gradient and then move along that line until you find a minimum. In each major iteration, it evaluates the gradient once and then does a line search to find the minimum along that line.

uses the rule "wherever you are, evaluate the gradient and then move along that line until you find a minimum." In each major iteration, it evaluates the gradient once and then does a line search to find the minimum along that line (figure 10.4).

Although this may be useful in problems where the gradient calculation is expensive, it doesn't seem to offer many benefits for (simulated MLP) neural networks in which the gradient calculation has basically the same cost as a function evaluation. The minimum on the line has no special significance and the calculations used in locating it precisely are basically wasted. Few recommend the method, but it is often used for comparison to show the benefits of more sophisticated methods. In neural network simulations, it is usually slower than simple gradient descent.

10.5.3 Conjugate Gradient Descent

Conjugate gradient descent is one of the most often recommended optimization methods for differentiable functions with many variables. Although it uses only first-order gradient information, it is nearly as fast as full second-order methods. In addition, it is practical for large problems because it avoids the need to store and manipulate the Hessian matrix by assuming a locally quadratic function and using the property of conjugate directions.

As mentioned in section 10.4.3, A set of vectors $\mathbf{d}_1, \mathbf{d}_2, \ldots \mathbf{d}_r$, $r \leq N$, are mutually conjugate with respect to a symmetric $N \times N$ matrix $\mathbf{H}$ if they are linearly independent and [311]

$$\mathbf{d}_i^T \mathbf{H} \mathbf{d}_j = 0 \quad \text{for all } i \neq j. \tag{10.4}$$

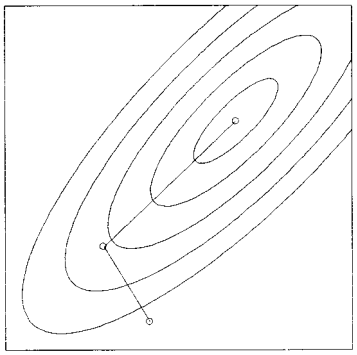

Figure 10.5
The conjugate gradient method converges in just N iterations (2 in this case) for an N-dimensional quadratic error function, but each iteration involves a line search that may require many evaluations of the error function.

This is useful because an N-dimensional quadratic function determined by $\mathbf{H}$ can be minimized by performing independent one-dimensional searches along any set of N mutually conjugate directions. The directions are uncoupled in the sense that minimizing along direction $\mathbf{d}_i$ doesn't undo gains that were obtained by minimizing along direction $\mathbf{d}_j$. Figure 10.5 illustrates the (two) steps taken for a two-dimensional quadratic error surface.

As in Powell's method, a set of conjugate search directions can be obtained without evaluating the Hessian matrix. In practice, the search direction $\mathbf{d}_k$ for step k is a combination of the current gradient $\mathbf{g}_k$ and the previous search direction $\mathbf{d}_{k-1}$

$$\mathbf{d}_k = -\mathbf{g}_k + \gamma \mathbf{d}_{k-1} \tag{10.5}$$

where the subscripts index time, not elements of the vectors. It has been noted that this could be thought of as back-propagation with the momentum parameter optimally adapted at each step. Parameter γ controls how much the next search direction is influenced by the previous search direction. There are a number of variations differing in how γ is chosen

$$\gamma = \frac{\|\mathbf{g}_k\|^2}{\|\mathbf{g}_{k-1}\|^2} \quad \text{(Fletcher-Reeves)} \tag{10.6}$$

$$\gamma = \frac{(\mathbf{g}_k - \mathbf{g}_{k-1})^T \mathbf{g}_k}{\|\mathbf{g}_{k-1}\|^2} \quad \text{(Polak-Ribiere)} \tag{10.7}$$

Although these are mathematically equivalent for quadratic functions, they give different results for general nonlinear functions, in which case the Polak-Ribiere form is preferred [302].

Minimization of a nonquadratic function will require more than N line searches but only N conjugate directions are available, so it is necessary to reinitialize the procedure periodically. There are various prescriptions for how and when to restart. The simplest is to reset the search direction to the gradient after N line searches.

Without restarts, later directions become linearly dependent and convergence is almost always linear [138]. With restarts, convergence is supralinear in theory, but in practice it is nearly always linear [138] because of round off errors, inexact line searches, the failure of the quadratic assumption, and so on. Still, it is usually much faster than simple gradient descent (which also converges linearly) because the scale factor may be much lower. Conjugate gradient and back-propagation were compared empirically on neural network N/2/N encoder problems in [360]; conjugate gradient was about an order of magnitude faster and back-propagation rarely converged for the harder (large N) problems, but it was estimated that both have roughly equal median time complexities. That is, training time scaled with N in approximately the same way.

The efficiency of the line search routine is critical because this is where most of the computation time is spent. Often, there are several parameters that must be tuned to obtain good performance. On simple problems, good line search routines may require as few as 3 to 4 function or gradient evaluations, but a typical search may require 10 to 20 evaluations, depending on parameter choices, nonlinearity of the function, and the precision required. It should be noted that some popular line search routines use parabolic approximations which may offer fast convergence for quadratic problems, but which may be slow or divergent for neural network error surfaces with stair-step shapes.

Scaled conjugate gradient descent [270] is a variation that eliminates the line search and all tunable parameters, but seems to have properties similar to the basic algorithm. It has been reported to be faster than the basic algorithm in some cases and slower in others. In [9], it was somewhat slower than other conjugate gradient methods.

Conjugate gradient descent has been compared with back-propagation in a number of studies [e.g., 26, 225, 27, 124, 197, 14, 19, 47, 69, 210, 75, 208, 360, 179, 296, 313, 383]. A detailed discussion and source code are provided in [258]. Most studies report an order of magnitude improvement in the number of iterations to convergence, an improved chance of convergence on hard problems, and smaller final errors. The reader should take care to note, however, if the iterations reported are the number of line searches or the total number of function (and/or gradient) evaluations. Where function evaluations or run-time are reported, it still appears to be faster, but the difference is not as great.

The results of performance comparisons with back-propagation seem to be task dependent [e.g., 377, 9]. In some cases, conjugate gradient is much faster; in others, it seems to have no advantage, or is slower. In [239] conjugate gradient worked well on simple prob-

lems, but simply converged (quickly) to a poor local minimum on a more difficult problem. Alpsan et al. [9] suggest that the difference depends on whether the problem is function approximation or classification. It is suggested that conjugate gradient (and other advanced methods) outperform back-propagation on function approximation problems where it is necessary to reduce the error to small values and that back-propagation generally does better on classification problems where training is stopped as soon as all patterns are correctly classified within a loose tolerance band. In [9], on a single network training problem, the time for conjugate gradient descent to correctly classify all patterns was similar to that of back-propagation without momentum and only half the speed of back-propagation with momentum. Generalization was poor.

An explanation for these results is that for very nonlinear functions, the quadratic approximation is not valid at a large scale so the conjugate gradient method has no advantage over simpler methods in initial stages of search (and its exact line searches may be superfluous). The quadratic approximation is usually valid, however, in a small region around an optimum, in which case conjugate gradient converges very quickly whereas gradient descent may actually slow down. For classification problems, the relaxed error criteria may be satisfied over a large region around a minimum so the search may end before the quadratic approximation becomes valid and the advantages of conjugate gradient come into play. For continuous regression problems, however, the region satisfying the error criteria is often much smaller so the search continues into a phase where conjugate gradient is by far superior.

Conjugate gradient descent was compared with (among other methods) back-propagation with an adaptive stepsize by Barnard and Holm [20]. The conjugate gradient method had lower errors and better generalization in early stages, but an adaptive algorithm achieved the same results after more iterations.

10.6 Second-Order Gradient Methods

Gradient methods using second-derivatives (the Hessian matrix) can be very efficient under certain conditions. Where first-order methods use a linear local approximation of the error surface, second-order methods use a quadratic approximation. If the function really is quadratic then a solution can be found very quickly (one step in Newton's method). The approximation is often good *in the vicinity of a minimum* in which case final convergence to the endpoint can be very fast. The approximation is often poor on a large-scale, however, so other methods may be better for initially finding the general neighborhood. The Levenberg-Marquardt method effectively switches from gradient descent to Newton's method as a minimum is approached.

10.6.1 Newton's Method

For smooth functions, Newton's method is the theoretical standard by which other optimization methods are judged. Because it uses all the first and second order derivative information in its exact form, its local convergence properties are excellent. Unfortunately though, it is often impractical because explicit calculation of the full Hessian matrix can be very expensive in large problems. Many 'second-order' methods therefore use approximations built up from first order information. Some methods for calculating and approximating second derivatives in neural networks are reviewed by Buntine and Weigend [62].

Newton's method is based on a quadratic model of the error function

$$\hat{E}(\mathbf{w}) = E_o + \mathbf{g}^T\mathbf{w} + \frac{1}{2}\mathbf{w}^T\mathbf{H}\mathbf{w} \tag{10.8}$$

where $\mathbf{g} = \frac{\partial E}{\partial \mathbf{w}}$ is the gradient vector, and $\mathbf{H}$ is the Hessian matrix of second derivatives with elements $h_{ij} = \frac{\partial^2 E}{\partial w_i \, \partial w_j}$. $\mathbf{H}$ is assumed to be positive definite. This is simply a Taylor series approximation truncated after the second order terms. Taking the derivative of $\hat{E}(\mathbf{w})$ with respect to $\mathbf{w}$ and setting it equal to 0 gives

$$\mathbf{g} + \mathbf{H}\mathbf{w} = 0, \tag{10.9}$$

which has solution $\mathbf{w}^* = -\mathbf{H}^{-1}\mathbf{g}$. If $\mathbf{H}$ is positive definite and $E(\mathbf{w})$ really is quadratic, the solution could be obtained in a single step (figure 10.6). Usually, of course, E is not exactly quadratic so iteration is necessary

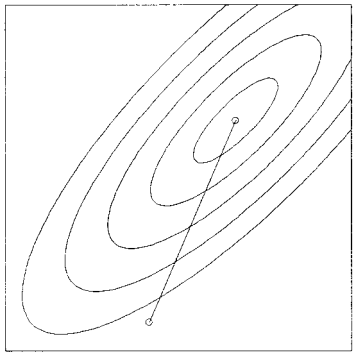

Figure 10.6
For quadratic error functions, Newton's method converges in a single step. For more general nonlinear functions, Newton's method converges very quickly where the Hessian matrix is positive definite (e.g., in the vicinity of a minimum) but may diverge elsewhere.

$$\mathbf{w}(t+1) = \mathbf{w}(t) - \eta \mathbf{H}^{-1}\mathbf{g} \tag{10.10}$$

where η is a step-size parameter ($\eta = 1$ normally).

The advantage of Newton's method is that it converges very quickly in the vicinity of a minimum. Convergence is quadratic for quadratic error functions; that is, in the limit $|e_{k+1}| \leq C|e_k|^2$ where $e = \mathbf{w} - \mathbf{w}^*$. Convergence requires that $\mathbf{H}$ be positive definite and $0 < \eta < 2/\lambda_{max}$ where λ_{max} is the largest eigenvalue of $\mathbf{H}$. In any case, η should be small enough to stay in the region where the quadratic approximation is valid.

A problem with the method is that it converges only where the Hessian is positive definite. (A symmetric matrix $\mathbf{H}$ is positive-definite if $\mathbf{x}^T\mathbf{H}\mathbf{x} > 0$ for all $\mathbf{x} \neq \mathbf{0}$. If the result is 0 for some $\mathbf{x} \neq \mathbf{0}$, $\mathbf{H}$ is positive-semi-definite. If the sign is positive or negative depending on $\mathbf{x}$, $\mathbf{H}$ is indefinite. All eigenvalues of a positive-definite matrix are nonzero and positive. Some eigenvalues of a positive-semi-definite matrix are zero. An indefinite matrix has eigenvalues of both signs.)

$\mathbf{H}$ could easily be indefinite in general nonlinear functions, especially far from a minimum, and it is common for it to be badly conditioned (nearly singular) in neural networks, especially if the number of training samples is smaller than the number of weights. Section 8.6 summarizes some properties of the Hessian in typical neural networks. When $\mathbf{H}$ has one or more negative eigenvalues, E has negative curvature in some directions, which would suggest that arbitrarily large steps would reduce the error by arbitrarily large amounts. These would likely move the system out of the region where its model is valid and, if really large, could lead to saturation of the sigmoid node functions. $\mathbf{H}$ might then be rank deficient if saturation makes enough second derivatives small relative to machine precision [28].

Modifications of the pure algorithm are necessary to prevent divergence in such cases. Ideas include limitations on the maximum step length, the use of line searches along the Newton direction when the step fails to decrease the error, and the use of matrix decompositions to solve (10.9).

Another problem, purely practical, is the need to evaluate, store, and invert the Hessian matrix at each iteration. In practice, dozens of iterations may be needed. Storage of the matrix requires $O(N^2)$ space and could become a problem on small computers and work stations as N exceeds 1000–2000. Exact evaluation of the matrix at a single point requires computations equivalent to approximately $O(N^2)$ epochs in a network with N weights. A simpler method might be able to make great gains with those N^2 function evaluations, possibly even solving the problem before Newton's method can take a single step. Finally, solution of (10.9) requires approximately $O(N^3)$ operations whether the matrix is actually inverted or not. (These are simple operations like addition and multiplication, however, rather than complex operations like evaluation of the network error function.) The N^2

evaluations will probably take more time for most neural networks when the training set is large because evaluation of each pattern will take $O(N)$ time and it can be argued that the number of training patterns should be at least $O(N)$ to ensure good generalization, in which case exact evaluation of the matrix could scale like $O(N^4)$.

The problem is illustrated in [38], where a modern nonlinear least squares optimization program (NL2SOL, ACM Algorithm 573) using the *exact* Hessian, generally achieved smaller errors than back-propagation, but training times were an order of magnitude longer (39 hours to do 50 iterations versus 2.2 hours to do 500,000 back-propagation iterations); the paper describes a hybrid quasi-Newton alternative.

10.6.2 Gauss-Newton

Calculation of the exact Hessian matrix can be expensive, so approximations are often used. The Gauss-Newton and Levenberg-Marquardt techniques take advantage of special structure in the Hessian for least squares problems and use the outer-product approximation of section 8.6.3. As noted in section 8.6.3, the Hessian can written as

$$\mathbf{H} = 2(-\mathbf{P} + \mathbf{Q}) \tag{10.11}$$

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. Gauss-Newton is based on the assumption that the first-order terms dominate the second order terms near a solution. Of course, this assumption is not always valid, but the approximation seems to work reasonably well, especially when the larger algorithm does not depend too heavily on its accuracy.

The Gauss-Newton weight update is a solution of

$$\mathbf{g} = -\mathbf{P}\Delta\mathbf{w}. \tag{10.12}$$

This could be solved by inverting $\mathbf{P}$, although other methods have practical advantages. Convergence is eventually quadratic once $\mathbf{Q}$ vanishes leaving $\mathbf{P}$ as an accurate approximation of $\mathbf{H}$. This avoids the need for a costly exact evaluation of $\mathbf{H}$, but it still requires storage of the matrix and solution of a matrix equation.

10.6.3 The Levenberg-Marquardt Method

A problem with Newton's method is that it converges only where $\mathbf{H}$ is positive definite, but $\mathbf{H}$ could easily be indefinite for a general nonlinear function, especially far from a minimum. The Levenberg-Marquardt method (figure 10.7) is a compromise be-

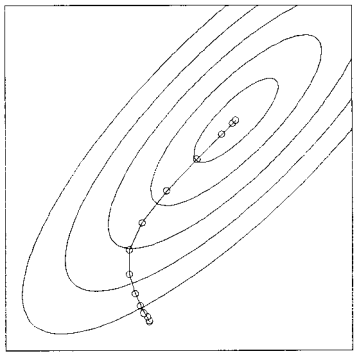

Figure 10.7
The Levenberg-Marquardt method is a compromise between Newton's method, which converges very quickly in the vicinity of a minimum but may diverge elsewhere, and gradient descent, which converges everywhere, albeit slowly. In general, the trajectory starts out like gradient descent but gradually approaches the Newton direction.

tween Newton's method, which converges quickly near a minimum but may diverge elsewhere, and gradient descent, which converges everywhere, though slowly. The search direction is a linear combination of the steepest descent direction $\mathbf{g}$ and the Newton direction $\mathbf{H}^{-1}\mathbf{g}$

$$\mathbf{w}_{k+1} = \mathbf{w}_k - (\mathbf{H} + \lambda\mathbf{I})^{-1}\,\mathbf{g}. \tag{10.13}$$

Parameter λ controls the compromise. This can be viewed as forcing $\mathbf{H} + \lambda\mathbf{I}$ to be positive definite by adding a scaled identity matrix. The minimum value of λ needed to achieve this depends on the eigenvalues of $\mathbf{H}$. The algorithm starts with λ large and adjusts it dynamically so that every step decreases the error. Generally it is held near the smallest value that causes the error to decrease. In the early stages when λ is large, the system effectively does gradient descent. In later stages, λ approaches 0, effectively switching to Newton's method for final convergence.

This avoids the divergence problem of Newton's method and the need for a line search, but still calls for storage and inversion of an $N \times N$ matrix. In the preceding description, $\mathbf{H}$ could be the true Hessian, but in practice the outer-product approximation of section 8.6.3 is usually used so Levenberg-Marquardt is commonly considered to be a first-order method applicable only to least squares problems.

The Levenberg-Marquardt method is compared to back-propagation and conjugate gradient descent by Hagan and Menhaj [148]. A variation combined with adaptive stepsize heuristics is described by Kollias and Anastassiou [218]. A detailed discussion and source code are provided in [258]. Results seem to be good on moderately sized problems.

10.6.4 Quasi-Newton Methods

Quasi-Newton methods, sometimes called variable metric methods, build an approximation of the inverse Hessian matrix iteratively using only first-order gradient information. This removes the need to explicitly calculate and invert the Hessian, but not the need to store the approximation. The two major variations are the Davidon-Fletcher-Powell (DFP) and the Broyden-Fletcher-Goldfarb-Shanno (BFGS) methods. They differ in details, but BFGS is generally recommended.

From [138], the BFGS update for the approximation $\mathbf{B}_k$ at step $k+1$ is

$$\mathbf{B}_{k+1} = \mathbf{B}_k - \frac{1}{\mathbf{s}_k^T \mathbf{B}\mathbf{s}_k} \mathbf{B}_k \mathbf{s}_k \mathbf{s}_k^T \mathbf{B}_k + \frac{1}{\mathbf{y}_k^T \mathbf{s}_k} \mathbf{y}_k \mathbf{y}_k^T \tag{10.14}$$

where $\mathbf{s}_k$ is the step taken and $\mathbf{g}_k$ is the gradient vector at iteration k, and $\mathbf{y}_k = \mathbf{g}_{k+1} - \mathbf{g}_k$. When the search direction $\mathbf{p}_k$ is computed by solving

$$\mathbf{B}_k \mathbf{p}_k = \mathbf{g}_k \tag{10.15}$$

and $\mathbf{s}_k = \alpha_k \mathbf{p}_k$ is the step taken after doing a line search along $\mathbf{p}_k$, this simplifies to

$$\mathbf{B}_{k+1} = \mathbf{B}_k + \frac{1}{\mathbf{g}_k^T \mathbf{p}_k} \mathbf{g}_k \mathbf{g}_k^T + \frac{1}{\alpha_k \mathbf{y}_k^T \mathbf{p}_k} \mathbf{y}_k \mathbf{y}_k^T. \tag{10.16}$$

The initial approximation $\mathbf{B}_0$ is usually the identity matrix so the first step is an iteration of best-step steepest-descent.

It is apparent that the formula preserves symmetry. For stability, it is critical that it also preserve positive-definiteness. Some care must be taken in the implementation because loss of positive-definiteness is possible due to limited-precision line searches and numerical round-off errors. Exact line searches are not necessary for convergence, however, and the method often converges faster (in terms of the number of function evaluations) using inexact line searches.

Storage requirements may make the method impractical for large networks, but it may be worth considering for small to moderate size networks. There do not seem to be major performance advantages compared to conjugate gradient methods, however. In [19], BFGS and conjugate gradient with restarts gave similar average errors on three neural network test problems and had similar convergence speed. On a neural network function approximation problem in [103], BFGS was able to achieve lower error than back-propagation, conjugate gradient, and simplex search; only Levenberg-Marquardt achieved a lower error. In [146], BFGS was the fastest and most reliable method in a comparison with conjugate gradient and an adaptive stepsize version of back-propagation on a small problem. Quasi-Newton training is specifically addressed in [2].

Although quasi-Newton methods have fast local convergence, this does not guarantee convergence to a good minimum. In [377], the DFP version converged to good solutions in only a third of 10,000 trials. In [9], DFP and BFGS gave basically the same results on a single real-world MLP training problem. Both located good minima in only half of the trials. In the other half they converged to local minima or suffered from numerical instability. Both were slower than standard back-propagation (with or without momentum) to learn the training set (classify correctly within error tolerances) and generalization was poor.

Limited Memory BFGS One-step limited memory BFGS [138] is a variation that avoids the need to store a matrix. A quasi-Newton update formula is used, but $\mathbf{H}^{-1}$ is taken to be the identity matrix so only $O(N)$ storage is needed. In implementation, it is similar to conjugate gradients.

The search direction at step k is

$$\mathbf{d}_k = -\mathbf{g}_k + A_k \mathbf{p}_k + B_k \mathbf{y}_k \tag{10.17}$$

where A_k and B_k are scalar weighting factors

$$A_k = -\left(1 + \frac{\mathbf{y}_k \cdot \mathbf{y}_k}{\mathbf{p}_k \cdot \mathbf{y}_k}\right) \frac{\mathbf{p}_k \cdot \mathbf{g}_k}{\mathbf{p}_k \cdot \mathbf{y}_k} + \frac{\mathbf{y}_k \cdot \mathbf{g}_k}{\mathbf{p}_k \cdot \mathbf{y}_k} \tag{10.18}$$

$$B_k = \frac{\mathbf{p}_k \cdot \mathbf{g}_k}{\mathbf{p}_k \cdot \mathbf{y}_k} \tag{10.19}$$

and $\mathbf{g}_k$ is the gradient, $\mathbf{p}_k = \mathbf{w}_k - \mathbf{w}_{k-1}$, and $\mathbf{y}_k = \mathbf{g}_k - \mathbf{g}_{k-1}$.

An advantage of the method over conjugate gradients is its tolerance of inexact line searches, which allows significant computational savings. In an MLP training test for a real-world problem, computation time was reduced by a factor of eight relative to conjugate gradient with exact line searches even though four times as many inexact line searches were required [29] (as reported by Alpson et al. [9]). Formula (10.17) is used in [27], and a variation is described in [28].

10.7 Stochastic Evaluation-Only Methods

A problem with deterministic gradient-based methods is that they can get trapped in local minima. In some cases, a few repetitions of the algorithm with different starting points may be enough to find the global minimum, but in other cases, for example, when there are many poor local minima or the basin of attraction of the global minimum is small, more powerful methods may be needed.

The advantage of stochastic methods is that every state has a nonzero chance of occurrence so if the procedure runs long enough the global minimum is likely to be found eventually. (This in itself is not special; pure random search will also find the global minimum if given enough time, but no one recommends it.) Under certain conditions, methods like simulated annealing and genetic algorithms can guarantee convergence (in probability) to the global minimum. This may take a very long time, however, and the guarantee is lost if the search is terminated early.

As evaluation-only methods, both simulated annealing and the genetic algorithm have the advantage of being relatively easy to implement in the sense that the algorithm is uncomplicated and there are no complex matrix manipulations. They need very little problem-specific information. They do not require a gradient vector so they may be used on discontinuous functions or functions described empirically rather than analytically. Under certain conditions, they will tolerate a noisy evaluation function.

10.7.1 Simulated Annealing

Simulated annealing [211, 264] is a general optimization technique based on a physical analogy. A physical system will generally settle to the lowest accessible energy state, but random thermal agitation will sometimes excite it to higher energy states. This is undesirable if we want the system to settle to the lowest possible energy, but occasionally it helps by giving the system enough energy to overcome a barrier separating it from even lower states.

The example of a liquid freezing into a solid is often used for illustration. A liquid is a disordered system with high energy. As energy is withdrawn, the liquid cools and begins to freeze. If the liquid is cooled very quickly, it tends to freeze into a disordered mass of tiny crystals with many imperfections and dislocations between grain boundaries. These imperfections are sites of internal stress that have high energy. If the system is cooled very slowly on the other hand, it tends to form large well-ordered crystals with few imperfections and low internal energy.

By cooling the system slowly, we allow many opportunities for random thermal agitation to rearrange atoms into more stable low-energy configurations. Because these states are more stable, they tend to survive longer. Although the probability of entering a particular stable state by random chance may be low, the probability of leaving is even lower so the system as a whole thus tends to a more ordered, lower energy state.

According to the Boltzmann statistics for a system in thermal equilibrium at temperature T, the probability of a state s with energy E_s is

$$P_s = \frac{1}{Z} e^{-E_s/kT} \tag{10.20}$$

where Z is a normalization constant and k is Boltzmann's constant. At very high temperatures, the exponent is small and all states are almost equally likely. At intermediate temperatures, there is a strong bias toward lower energy states, but higher energy states still have significant probability. At very low temperatures, low energy states are much more likely than higher energy states.

For optimization, an analogy is made between the system energy and the error function; parameter vectors with low errors correspond to system states with low energy. New candidate vectors are generated by modifying the current vector by some random process, for example, by adding noise to the current search point; different noise distributions give different convergence properties. If the new point has a lower error (energy), it is accepted as the new search point. If the new point increases the error (energy) by an amount ΔE, it is accepted with probability

$$P = e^{-\Delta E/T}.$$

That is, there is a chance P that the higher-error vector will be accepted as the new search point and a chance $(1 - P)$ that it will be rejected, in which case new candidates will be generated from the original point. At high temperatures, the exponent is very small and almost all proposed changes are accepted. At low temperatures, the probability decreases very quickly as ΔE increases; changes that increase the error have very low probability of being accepted and the trajectory approaches gradient descent.

The advantage of stochastic algorithms like simulated annealing is that every state has a nonzero chance of occurrence so if the process continues long enough, every state, including the global minimum, will be visited eventually. A purely random search would immediately hop to another state after visiting the global minimum, but simulated annealing has a bias to lower energy states so it is likely to remain near or revisit the global minimum often.

The promise of the method is that if the system is cooled slowly enough, it will eventually converge to the global minimum (with probability 1). The catch is that this may take a very very long time and all guarantees are lost if the system is cooled too quickly (quenched) or stopped too soon. It has been shown that a temperature reduction schedule inversely proportional to the logarithm of time will guarantee convergence (in probability) to a global minimum [135]

$$T(t) = \frac{T_0}{\log(1+t)}. \tag{10.21}$$

Because the logarithm increases slowly with t, this can take a very long time. In practice, this schedule takes too long and it is often more efficient to repeat the algorithm a

number of times using a faster schedule (sacrificing claims of provable convergence in the process).

10.7.2 Genetic Algorithms

The genetic algorithm [173, 141] is a general optimization method based on an analogy to the evolution of species in nature. (Chapter 11 describes the algorithm in more detail; the following remarks outline the main points.) The idea is that individuals in a large population have varying traits which affect their reproductive success in the given environment. Successful individuals live long enough to mate and pass their traits to the next generation. Offspring inherit traits from successful parents so they also have a good chance of being successful. Over many generations, the population adapts to its environment; disadvantageous traits become rare and the average fitness tends to increase over time.

There are a number of evolutionary algorithms based on similar ideas. The genetic algorithm includes effects of mating—combination of traits from two successful parents to yield offspring that are similar to, but slightly different from, either parent. There is an element of randomness so it has stochastic properties similar to simulated annealing, but it acts on an entire population rather than a single individual.

For optimization purposes, individuals are represented by a bit-string that encodes the parameters of a particular solution and their fitness depends on the objective function. Offspring are generated from two parents by combining their bit-strings in various ways.

The principal advantage of the method is that it needs very little problem-specific information—just a function to evaluate parameter sets and assign fitness scores. In particular, it does not require gradient information and so may be used on discontinuous functions or functions that are described empirically rather than analytically. The mating selection and crossover operations are already somewhat random so, assuming appropriate parameters, the algorithm will tolerate moderate amounts of noise in the evaluation function. The crossover operation is nonlinear so the algorithm is not necessarily a hill-climbing method and is not particularly bothered by local maxima (minima). It is also easily parallelizable.

The principle disadvantage of the method is the amount of processing needed to evaluate a large population of candidates and converge to a solution over many generations. There are claims of convergence to a global maximum of the fitness function, but with small populations and aggressive culling of less successful solutions, convergence to a local maximum is possible. (This is similar to the case of cooling too quickly in simulated annealing.) Also, there are many parameters to be selected (population size, mutation rate, crossover method, fitness scaling, etc.) and it is not obvious how these affect the convergence properties. At this point, it is unclear if the algorithm is better than other methods such as simulated annealing.

A difficulty with the method for neural network optimization is the problem of incompatible genomes mentioned in section 11.3.1. The problem is that two successful individuals do not always yield a successful offspring when they mate; their bit-strings might represent points on two different local maxima and the combination might fall in a valley between them, for example. Peculiarities of the typical neural network structure aggravate this problem. Variations have been proposed to avoid the problem but they complicate the algorithm and detract from its advantage of simplicity.

10.8 Discussion

10.8.1 Are Assumptions Met?

When the objective function has special characteristics, specialized algorithms can often be developed to take advantage of them and perhaps obtain great efficiencies relative to more general algorithms. The problem is that although they may be very fast in the situations they were developed for, they may not be robust. That is, the assumptions they make may lead them astray and they may actually do worse than less specialized methods if their preconditions are not satisfied.

A number of methods such as conjugate gradient descent assume a quadratic function and exploit its special properties to achieve fast convergence. Unfortunately, neural network error surfaces may be highly nonlinear and are definitely not quadratic in the large scale. (Chapter 8 discusses properties of the $E(\mathbf{w})$ error surface that may cause problems—the existence of large flat areas separated by steep cliffs, for example.) Quoting from [302: 313]: "Quadratic convergence is of no particular advantage to a program which must slalom down the length of a valley floor that twists one way and another (and another, and another, . . . —there are N dimensions!). Along the long direction, a quadratically convergent method is trying to extrapolate to the minimum of a parabola which just isn't (yet) there; while the conjugacy of the $N - 1$ transverse directions keeps getting spoiled by the twists."

Another problem with methods based on a quadratic approximation is that the Hessian is often ill-conditioned in neural network training problems [37, 331] and so higher order methods may be no more efficient than simpler methods. When the number of weights exceeds the number of training samples, an outer-product Hessian approximation will be rank deficient. Sigmoid saturation may also lead to effective loss of rank. Ill-conditioning could lead to numerical instability or a large step that takes the system out of the region where the local approximation is valid. This is a well-known problem with Newton's method and there are standard fixes, but these add complications beyond the pure algorithm. The point

is that a straight-forward implementation of a pure algorithm will not always work better than a simpler method unless these sorts of complications are addressed.

The quadratic approximation is usually valid in the neighborhood of a minimum though so it may be useful to use a more robust method for initial optimization, followed by a few iterations of a fast second-order method to tune the solution. Some theoretical justification is given in [394].

10.8.2 Back-Propagation Is Sometimes Good Enough

It has been said that "back-propagation is the second-best method for everything." That is, there are many algorithms which are faster and give better results in particular situations, but back-propagation and simple variants often do surprisingly well on a wide range of neural network training problems.

Standard optimization methods have been considered for neural network training in many studies. Most report faster training times, smaller errors, and better chance of convergence. This might leave the impression that they are uniformly better. There are cases, however, where back-propagation and its variations are faster or less prone to convergence to poor local minima than more sophisticated algorithms which should be better in theory. In [77], for example, several second-order methods are compared to back-propagation on a simple problem (seven weights plus an additional scaling parameter; the target is a sine function). All achieved smaller training errors, but all took more time. Alpsan et al. [9] evaluated approximately 25 different optimization techniques, including numerous variations of back-propagation, on a single real-world classification problem (classification of brainstem auditory evoked potentials). Quoting from the abstract [9]:

> It was found that, comparatively, standard BP was sufficiently fast and provided good generalization when the task was to learn the training set within a given error tolerance. However, if the task was to find the global minimum [i.e., reduce the error to very small value], then standard BP failed to do so within 100000 iterations, but first order methods which adapt the stepsize were as fast as, if not faster than, conjugate gradient and quasi-Newton methods. Second order methods required the same amount of fine tuning of line search and restart parameters as did the first order methods of their parameters in order to achieve optimum performance.

In the same study, second-order methods showed a greater tendency to converge to local minima and the solutions found generalized worse than those found by first order methods. In other remarks, "None of the more sophisticated second order methods were able to learn [to classify] the training set faster than BP" [9]. Similar results have been reported elsewhere.

Part of the relative success of back-propagation may be due to its simple-mindedness; it makes very few assumptions. Part may also be due to special characteristics of the neural network training problem, for example, the shape of the typical error surface, that conflict

with assumptions used by more sophisticated methods developed for other purposes. The remarks in section 10.5.3 about possible causes for the difference in performance on classification and continuous function approximation apply to most methods which assume a quadratic approximation. Basically, in classification, training is often stopped when all output errors are less than a certain value (e.g., 0.2) so the search may end before the quadratic approximation becomes valid and the advantages of the specialized algorithms don't have a chance to come into play. This may also be true to a lesser degree even for regression problems where the error tolerance is smaller. Early stopping based on cross-validation tests may also end the game before second-order methods become advantageous.

Part of back-propagation's success may also be due to network designers adapting to the algorithm. That is, it has been found that tricks such as input variable normalization, the use of tanh instead of sigmoid node functions, the use of $\{-0.9, 0.9\}$ targets instead of $\{0, 1\}$, the use of larger-than-necessary networks with early stopping and/or pruning, and so on seem to make learning easier and many of these have become standard practice. Some might view this as cheating because it changes the problem to fit the optimization method, but there is certainly no reason to intentionally design networks that are hard to train. Often, when such steps are taken, back-propagation or a simple variant may outperform more sophisticated methods. Some papers comparing conjugate gradients to back-propagation, for example, report on the order of 50,000 epochs to train a 4-4-1 network on the 4-input XOR problem. Back-propagation may actually take this long for the parameters used, but when simple fixes like those mentioned previously are taken, the problem becomes very easy and can generally be solved by back-propagation in a few hundred epochs. 40–50 epochs is typical for this problem with Rprop. On the simpler problem, conjugate gradient is not even as fast as back-propagation because it wastes evaluations in precise line searches.

Finally, aside from performance issues, there are sometimes good reasons for preferring simple algorithms.

- Simplicity may be important because computational resources are limited.
- Local algorithms are preferable for integrated-circuit implementation. Complicated matrix manipulations are not feasible on analog retina chips, for example. Simple methods like on-line back-propagation use information that is locally available at the weights being modified.
- Training time may not be a major consideration because it is usually a one-time procedure. The environment may change very slowly so frequent retraining is not necessary.
- In some studies, the algorithm is intended to be feasible in terms of what real neurons can do. The fact that more sophisticated algorithms exist is not relevant if they are not used by the system under study.

• Many sophisticated algorithms simply do not give better or faster solutions than simple variations of back-propagation on the problems considered.

• Some specialized algorithms are not robust. They do not work in all situations and break down if assumptions are not met. Simple implementations that ignore complications such as round-off error may not work well.

• More sophisticated algorithms often do not yield good generalization. Second-order methods can often achieve much smaller training set errors than back-propagation, but this may simply amount to overfitting when training data is limited.

10.8.3 Remarks

It is easy to get side-tracked into tinkering with optimization methods. It should be remembered that the optimization method is of secondary importance and factors such as representation, network structure, and choice of error function are more fundamental. If solutions are poor or training times are long, the problem could be more basic than the way the weights are tuned. A good optimization algorithm cannot fix problems introduced by a representation that hides needed information or an error function that does not measure the appropriate thing, after all. Sophisticated algorithms have their place, but they should generally be considered only other options have been exhausted.

Still, efficiency is important, especially for large networks with large data sets or in cases where many nets will be trained on similar data, so there are situations where it pays to "optimize the optimization method." If sophisticated algorithms are needed, consider using one of the highly efficient "canned" optimization programs that are available rather than writing from scratch. The best programs are already debugged, handle many special cases, and correctly deal with important implementation details that are seldom considered in simple descriptions of the pure algorithms. Some can switch between robust and specialized methods when appropriate. Most programs offer a selection of methods. Generally, it is necessary to understand something of the methods used by the program in order to fit the technique to the problem.

Other Notes

• Conjugate gradient descent and Levenberg-Marquardt are the classical methods most often mentioned for neural network training. Both are discussed in detail in this context in [258].

• Second order methods require the Hessian matrix or an approximation. Some possibilities are described in section 8.6.

• In several places above, inversion of the Hessian or other large matrixes is mentioned as a way of obtaining a solution. In practice, other methods that can handle rank-deficient

and poorly conditioned cases are preferred. Techniques are discussed in numerical analysis texts.

• In optimization, it is usually desirable to find a solution as accurately as possible and overfitting is not considered directly. That is, the objective function is considered to be a completely accurate measure of what is sought; if generalization is one of the goals, it is assumed to be reflected in the objective function (e.g., via penalty terms). Similar considerations are encountered in the development of robust algorithms and the analysis of the sensitivity of the solutions obtained.

• It should be noted that if training is stopped when the error is less than a given tolerance, then it is usually the case that $E \neq 0$ and $\| \frac{\partial E}{\partial \mathbf{w}} \| \neq 0$ at the stopping point. This is contrary to the assumptions of some analyses. Some pruning algorithms, for example, assume $\| \frac{\partial E}{\partial \mathbf{w}} \| = 0$.

• There is nothing in these techniques that limits them to multilayer network structures. They can be applied to most other (continuous) neural network models as well.

• The optimization approach to training makes no claim of biological plausibility. Many of the methods are nonlocal, complex, and batch-oriented. The goal is simply to find a good set of weights for the problem at hand. Although back-propagation is not considered to be a particularly plausible model of learning in biological networks, it is at least driven by local computations.

• Neural networks are sometimes used as optimizing systems themselves, for example, the proposed use of Hopfield networks for solving the traveling salesman problem. The application of neural networks to problems normally cast as optimization or signal processing problems is considered in detail by Cichoki and Unbehaven [82].

• Shanno [341] discusses recent work in optimization methods in light of its utility to neural network training.