9 Faster Variations of Back-Propagation

One of the common complaints about back-propagation is that it can be very slow. A typical training session may require thousands of iterations. Large networks with large training sets might take days or weeks to train. This chapter reviews a number of relatively simple variations of the basic algorithm that are intended to speed up learning.

It should be noted that things such as the network structure, the input-output representation, the choice of error function, and so on, often have much stronger effects on learning time (possibly orders of magnitude) than variations in the optimization method. At the time of training, however, these choices have already been made and the goal of the methods described next is to accelerate learning in a given network with the given data.

Many variations of the basic algorithm have been proposed and new ones continue to appear regularly. We will not attempt to summarize them all. Many methods are heuristic and somewhat ad hoc; others are founded on principled theory. Some are specialized to certain problem types, for example, classification, and do not always work well on other sorts of problems. Some draw on general optimization techniques specialized to neural network applications. To appreciate these, it is worth reviewing the classic optimization techniques (chapter 10).

Next, a few methods are listed that have stood up to testing and seem to work reasonably well on a wide range of problems. Also listed are some well-known methods that deserve mention if only to inform the reader who has heard of them and wonders what is involved.

9.1 Adaptive Learning Rate Methods

Many of the methods listed here are adaptive learning rate schemes. As noted in section 6.1, the often recommended learning rate of $\eta = 0.1$ is a somewhat arbitrary value that may be completely inappropriate for a given problem. For one thing, the magnitude of the gradient depends on how the targets are scaled; for example, the average error will tend to be higher in a network with linear output nodes and targets in a $(-1000, 1000)$ range than in a network with sigmoid output nodes and targets in $(0, 1)$. Also, when sum-of-squares error is used rather than mean squared error, the size of the error and thus the best learning rate may depend on the size of the training set [114]. The effective learning rate is amplified by redundancies such as near duplication of training patterns and correlation between different elements of the same pattern, and by internal redundancies such as correlations between hidden unit activities. The latter depend in part on the size and configuration of the network but change as the network learns so different learning rates may be appropriate in different parts of the network and the best values may change as learning progresses.

Given the difficulty of choosing a good learning rate a priori, it makes sense to start with a "safe" value (i.e., small) and adjust it depending on system behavior. Some methods

adjust a single global learning rate while others assign different learning rates for each unit or each weight. Methods vary, but the general idea is to increase the step size when the error is decreasing consistently and decrease it when significant error increases occur (small increases may be tolerated).

In general, some care is needed to avoid instability. The best step size depends on the problem and local characteristics of the $E(\mathbf{w})$ surface (Chapter 8). Values that work well for some problems and some regions of the error space may not work well for others. It has been noted that neural networks often have error surfaces with many flat areas separated by steep cliffs. This is especially true for classification problems with small numbers of samples. As in driving a car, different speeds are reasonable in different conditions. A large step size is desirable to accelerate progress across the smooth, flat regions of the error surface while a small step size is necessary to avoid loss of control at the cliffs. If the step size is not reduced quickly when the system enters a sensitive region, the result could be a huge weight change that throws the network into a completely different region basically at random. Besides causing problems such as paralysis due to saturation of the sigmoid nonlinearities, this has the undesirable effect of essentially discarding previous learning and starting over somewhere else.

9.2 Vogl's Method (Bold Driver)

Vogl et al. [380] describe an adaptive learning rate method where the global learning rate $\eta(t)$ at time t is updated according to

$$\eta(t) = \begin{cases} \phi\eta(t-1) & \text{if } E(t) < E(t-1) \\ \beta\eta(t-1) & \text{if } E(t) > 1.05E(t-1) \\ \eta(t-1) & \text{otherwise} \end{cases} \tag{9.1}$$

where $\phi > 1$ and $\beta < 1$ are constants. Suggested values are $\phi = 1.05$ and $\beta = 0.7$. The name "bold driver" comes from Battiti [27]; there the value $\beta = 0.5$ is suggested based on the idea that an increase in E indicates a minimum has been overstepped and, on average it is reasonable to guess it is halfway between the current and previous weights.

In addition to decreasing the learning rate when the error increases significantly, the previous weight change is also retracted and the momentum parameter is reset $\alpha = 0$ for the next step. The justification for clearing α is that $\alpha > 0$ makes the current weight change similar to previous weight changes and the increase in the error indicates the need for a change in direction. Thus α is restored to its normal value after a successful step is taken.

In [380], learning speed increased by a factor of about 2.5 and 30 on two test problems. A similar method without momentum was unfavorably compared to conjugate gradient training on parity problems of various sizes in [27]. There it appears to give results similar to normal back-propagation with an optimally tuned fixed learning rate but without the need to search for the optimal learning rate.

The method was empirically compared to a number of other methods on a single test problem by Alpsan et al. [9]. In one case, learning was stopped as soon as all patterns were correctly classified (all outputs on all patterns correct within a tolerance 0.1 of the target values). With high momentum, it had about the same speed as optimally tuned back-propagation, but generalization was not as good. Generalization was better without momentum, but then learning was much slower than regular back-propagation. In a second case where convergence criteria required the outputs to essentially match the target values, the method converged whereas plain back-propagation did not, but it was not among the fastest methods. In an earlier test by the same authors, it was said to be somewhat unstable and no easier to tune than plain back-propagation.

9.3 Delta-Bar-Delta

Jacobs' delta-bar-delta algorithm [194] is one of the more often mentioned acceleration methods. Although some newer methods seem to perform better, it is well-known and many other methods are based on similar ideas. It is based on four heuristics:

1. Every parameter should have its own learning rate. It is not reasonable for every parameter to have the same learning rate because of differences in scaling, variance, and so on in different parts of the network.

2. Every learning rate should be allowed to vary over time because local properties of the error surface change as the weight vector moves over it. Learning rates that are appropriate in one area may not be appropriate in other areas.

3. The learning rate can be increased when the partial derivative of the error has the same sign over several steps. This tends to mean that the error surface has a small curvature and continues to slope in the same direction for some distance so it should be safe to increase the step size.

4. The learning rate should be decreased when the partial derivative changes sign several times in a row. This tends to mean that the weight vector is bouncing back and forth across a minimum and corresponds to high curvature in the error surface along that direction.

These heuristics lead to the following adjustment rule. Each weight w has its own learning rate $\eta(t)$, which is adjusted after each epoch according to

$$\Delta\eta(t) = \begin{cases} \kappa & \text{if } \bar{\delta}(t-1)\delta(t) > 0 \\ -\phi\eta(t) & \text{if } \bar{\delta}(t-1)\delta(t) < 0 \\ 0 & \text{otherwise} \end{cases} \tag{9.2}$$

where $\delta(t) = \frac{\partial E}{\partial w}$ at time t and $\bar{\delta}$ is the exponential average of past values of δ

$$\bar{\delta}(t) = (1-\theta)\delta(t) + \theta\bar{\delta}(t-1). \tag{9.3}$$

(Note, this δ is not the δ used in back-propagation.) The learning rate is incremented by a constant κ when δ and $\bar{\delta}$ have the same sign in consecutive iterations and it is decremented by a fraction of its current value when they have different signs. Note that the increase is linear while the decrease is exponential. The learning rate increases gradually when many consecutive steps all move in the same direction, but decreases quickly when conditions change.

As in normal back-propagation, the weight update is

$$w(t+1) = w(t) - \eta(t)\delta(t). \tag{9.4}$$

This is no longer equivalent to gradient descent on the error surface, however, because each weight has its own learning rate. In effect, the weights are updated based on partial derivatives plus estimates of curvature.

Typical parameter values are obtained from simulation results for several small problems reported by Jacobs [194]. Initial learning rates were $\eta_o = 0.8$ to 1. Typical parameter values were $\kappa = 0.03$ to 0.1 and $\phi = 0.1$ to 0.3 depending on the problem. Harder problems seem to require smaller values of κ and larger values of ϕ. This corresponds to a cautious policy: small increases in learning rate when things are going well and large decreases when things go badly. The averaging parameter $0 < \theta < 1$ does not seem to be critical, $\theta = 0.7$ was used in all cases. Larger values, approaching 1, give longer averaging times.

It is noted that these heuristics can fail in certain cases. For instance, the ideal situation would be to have separate learning rates for each direction identified by an eigenvector of the local Hessian matrix. Instead, it has separate learning rates for each of the coordinate directions in the $E(\mathbf{w})$ space. In the case of a ravine oriented 45° to two weight axes, for example, these heuristics cause the learning rates of both weights to decrease when the best option would be for them to increase *together*. Because the method is based on local computations only, the two weight changes cannot be coordinated. When changing one weight, the behavior of other weights is not considered.

In one empirical test [9], delta-bar-delta was among the fastest methods to learn to classify correctly (with all outputs within a loose tolerance of the desired values) but it was slow to reduce the error to very small values. In [239], it was slower than standard back-propagation with a carefully selected learning rate. The time difference was relatively

small, however, and the adaptive method would probably be faster if the time spent in tuning the learning rate for standard back-propagation were included.

According to some reports, delta-bar-delta seems to be more sensitive to parameters than Rprop or quickprop. That is, the default values (κ, ϕ, θ) may work reasonably well on easy problems, but different parameters may be needed on hard problems and it may not be easy to find a good set.

Section 9.4 summarizes a similar method using multiplicative weight increases and momentum. Both are said to be implementations of heuristics proposed by Sutton [363]. Minai and Williams [266] describe an extended delta-bar-delta algorithm that adapts the momentum as well. There are more parameters to be tuned, however.

9.3.1 Justification

Justification for the seemingly *ad hoc* heuristic of basing the learning rate changes on the signs of successive partial derivatives $\frac{\partial E}{\partial w}(t)$ and $\frac{\partial E}{\partial w}(t-1)$ can be found in [194] and [160: 194]. Assuming a single output node y for simplicity, the mean squared error at epoch t is

$$E(t) = \frac{1}{2}\left\langle (d-y)^2 \right\rangle. \tag{9.5}$$

The brackets $\langle\rangle$ denote the mean over the training set and are dropped in what follows. The derivative of the error *with respect to the learning rate* η_{ij} can be written

$$\frac{\partial E}{\partial \eta_{ij}}(t) = \frac{\partial E}{\partial y_i}(t)\frac{\partial y_i}{\partial a_i}(t)\frac{\partial a_i}{\partial \eta_{ij}}(t) \tag{9.6}$$

where $a_i(t) = \sum_j w_{ij}(t) y_j(t)$ is the weighted-sum input to node i, $y_i(t) = f(a_i(t))$ is the node output, and f is the node nonlinearity, for example, the sigmoid function. Because

$$w_{ij}(t) = w_{ij}(t-1) - \eta_{ij}(t)\frac{\partial E}{\partial w_{ij}}(t-1) \tag{9.7}$$

we have

$$a_i(t) = \sum_j y_j(t)\left[w_{ij}(t-1) - \eta_{ij}(t)\frac{\partial E}{\partial w_{ij}}(t-1) \right]. \tag{9.8}$$

Differentiation with respect to $\eta_{ij}(t)$ gives

$$\frac{\partial a_i}{\partial \eta_{ij}}(t) = -y_j(t)\frac{\partial E}{\partial w_{ij}}(t-1). \tag{9.9}$$

From the back-propagation derivation, equation 5.10, we know

$$\delta_i = \frac{\partial E}{\partial y_i}\frac{\partial y_i}{\partial a_i} \tag{9.10}$$

and

$$\frac{\partial E}{\partial w_{ij}} = -\delta_i y_j. \tag{9.11}$$

Combining these results allows (9.6) to be rewritten

$$\begin{aligned}\frac{\partial E}{\partial \eta_{ij}}(t) &= \frac{\partial E}{\partial y_i}(t)\frac{\partial y_i}{\partial a_i}(t)\frac{\partial a_i}{\partial \eta_{ij}}(t) \\ &= \delta_i(t)\left(-y_j(t)\frac{\partial E}{\partial w_{ij}}(t-1)\right) \\ &= -\frac{\partial E}{\partial w_{ij}}(t)\frac{\partial E}{\partial w_{ij}}(t-1).\end{aligned} \tag{9.12}$$

This says that the derivative of the error with respect to the learning rate η_{ij} is the negative of the product of the present and previous derivatives of the error with respect to the weight w_{ij}. Rather than being an ad hoc heuristic, this is actually a well-founded way of doing gradient descent on the error with respect to the learning rate. The delta-bar-delta update rule (9.2) modifies this slightly by smoothing $\frac{\partial E}{\partial w}(t-1)$.

9.4 Silva and Almeida

Delta-bar-delta is one of the more well-known adaptive learning rate methods. Silva and Almeida [346] proposed a variation using multiplicative weight increases and momentum. Both are said to be implementations of heuristics proposed by Sutton [363]. This is similar to the method of Vogl et al. with a separate learning rate for each weight.

The weight update rule is

$$w_{ij}(t) = w_{ij}(t-1) - \eta_{ij}\frac{\partial E}{\partial w_{ij}}(t) \tag{9.13}$$

where $\eta_{ij}(t)$ is the learning rate for weight w_{ij} at epoch t. The learning rate is adapted at each epoch according to

$$\eta_{ij}(n) = \begin{cases} u\eta_{ij}(n-1) & \text{if } \frac{\partial E}{\partial w_{ij}}(t)\frac{\partial E}{\partial w_{ij}}(t-1) > 0 \\ d\eta_{ij}(n-1) & \text{if } \frac{\partial E}{\partial w_{ij}}(t)\frac{\partial E}{\partial w_{ij}}(t-1) < 0 \\ \eta_{ij}(n-1) & \text{otherwise (no change)} \end{cases} \tag{9.14}$$

where constants $u > 1$ and $0 < d < 1$ control the rate of increases and decreases. Typical values are $1.1 < u < 1.3$ and d slightly below $1/u$, for example, $d = 0.7$. This gives a slight preference to learning rate decreases, making the system more stable.

In contrast to Jacobs' delta-bar-delta method where the learning rate increases incrementally (additively), here both increases and decreases are multiplicative. This allows faster increases in the learning rate and, possibly, faster convergence, but it may sometimes lead to instability. If the learning rate becomes too large, the error may sometimes jump abruptly (e.g., when the system oversteps a minimum and "climbs up a cliff"). To avoid instability, the bad weight change is retracted and in most cases reapplication of the learning rate update rule (9.14) using the gradient evaluated at the rejected point will reduce the learning rate adequately to avoid the bad step in following iterations; if not, the learning rate may need to be decreased directly. In a benchmarking test [320], it is suggested that if the algorithm fails to find an error decrease after five consecutive iterations, all the learning rate parameters should be halved.

Because the learning rate can increase quickly, there is not a huge cost in selecting an initial rate that is too small. Ideally, the algorithm should be able to correct for an overly large initial learning rate, but sigmoid saturation and instability may cause problems so it is probably best to start with a small value and let the algorithm increase it if necessary.

Performance seems to deteriorate in obliquely oriented ravines in the error surface. In order to better handle these cases, a modified weight update rule was proposed

$$\Delta w_{ij}(t) = \eta_{ij} v_{ij}(t) \tag{9.15}$$

where the 'smoothed gradient' is

$$v_{ij}(t) = \frac{\partial E}{\partial w_{ij}}(t) + \alpha v_{ij}(t-1) \tag{9.16}$$

and $0 \leq \alpha < 1$ functions like the momentum parameter.

It has been reported [9] that methods that increase the learning rate multiplicatively like this can be faster than methods that increase it additively, but they are less stable and parameter tuning may be difficult.

9.5 SuperSAB

SuperSAB [372] is another adaptive learning rate method based on the delta-bar-delta heuristics. It is based on an earlier method called SAB, which stands for "self-adapting back propagation." Like the method of Vogl et al. (section 9.4), the learning rate is both increased and decreased multiplicatively.

Parameters include the initial learning rate η_{start}, an increase factor $\eta^+ > 1$, and a decrease factor $0 < \eta^- < 1$. Each weight has its own learning rate $\eta_{ij}(t)$ which changes with time t. The algorithm is:

1. Initialize all learning rates $\eta_{ij}(0) = \eta_{start}$.
2. Do a back-propagation step with momentum.
3. For each weight w_{ij}

• if the sign of its derivative is unchanged then increase the learning rate, $\eta_{ij}(t+1) = \eta^+ \cdot \eta_{ij}(t)$;

• otherwise (the sign changed), retract the step $w_{ij}(t+1) = w_{ij}(t) - \Delta w_{ij}(t)$, decrease the learning rate $\eta_{ij}(t+1) = \eta^- \cdot \eta_{ij}(t)$, and set $\Delta w_{ij}(t+1) = 0$ so momentum has no effect in the next cycle.

4. Go to 2.

Typical suggested values are $\eta^+ = 1.2$ and $\eta^- = 0.5$. (There appear to be typographical errors in [372]. This is based on the explanation accompanying the formula.)

Reported results have been inconsistent. In some cases SuperSAB is among the fastest methods [9]; others have reported it to be very unstable [8]. The possibility of instability, especially when momentum is high, is noted in the original paper. This shows itself as a sudden large increase in the error. Sometimes the error will correct itself in subsequent steps; otherwise a restart may be necessary. Because η increases multiplicatively and can become large quickly, it is reasonable to set limiting values on both η and the maximum allowed weight magnitude. Because of the instability problems and because it does not appear to have major speed advantages, other methods may be preferable in general.

9.6 Rprop

Rprop [315, 314] stands for "resilient propagation." The main difference between it and most other heuristic back-propagation variations is that the learning rate adjustments and weight changes depend only on the signs of the gradient terms, not their magnitudes. It is argued that the gradient magnitude depends on scaling of the error function and can

change greatly from one step to the next. On a complicated nonlinear error surface, the magnitude is basically unpredictable a priori and there is no reason why the step size should be proportional to the magnitude in general. In fact, it can be argued that the step size should be inversely proportional in order to take large steps where the gradient is small and to take small careful steps where the gradient is large [363].

Rprop is a batch update method; the weights and step sizes are changed once per epoch. Each weight w_{ij} has its own step size, or update-value, Δ_{ij}, which varies with time t according to

$$\Delta_{ij}(t) = \begin{cases} \eta^+ \cdot \Delta_{ij}(t-1), & \text{if } \frac{\partial E}{\partial w_{ij}}(t-1) \cdot \frac{\partial E}{\partial w_{ij}}(t) > 0 \\ \eta^- \cdot \Delta_{ij}(t-1), & \text{if } \frac{\partial E}{\partial w_{ij}}(t-1) \cdot \frac{\partial E}{\partial w_{ij}}(t) < 0 \\ \Delta_{ij}(t-1), & \text{otherwise} \end{cases} \tag{9.17}$$

where $0 < \eta^- < 1 < \eta^+$. A change in sign of the partial derivative corresponding to weight w_{ij} indicates that the last update was too big and the system has jumped over a minimum so the update value Δ_{ij} is decreased by a factor η^-. Consecutive derivatives with the same sign indicate that the system is moving steadily in one direction so the update value is increased slightly in order to accelerate convergence in shallow regions.

The weights are changed according to

$$\Delta w_{ij}(t) = \begin{cases} -\Delta_{ij}(t), & \text{if } \frac{\partial E}{\partial w_{ij}}(t) > 0 \\ +\Delta_{ij}(t), & \text{if } \frac{\partial E}{\partial w_{ij}}(t) < 0 \\ 0 & \text{otherwise} \end{cases} \tag{9.18}$$

Note that the change depends only on the sign of the partial derivative and is independent of its magnitude. If the derivative is positive, the weight is decremented by $\Delta_{ij}(t)$; if the derivative is negative, the weight is incremented by $\Delta_{ij}(t)$.

There is one exception. If the partial derivative changes sign (indicating that the previous step was too large and a minimum was missed), the previous weight-update is retracted

$$\Delta w_{ij}(t) = -\Delta w_{ij}(t-1) \quad \text{if } \frac{\partial E}{\partial w_{ij}}(t-1)\, \frac{\partial E}{\partial w_{ij}}(t) < 0 \tag{9.19}$$

Because this would cause another sign change on the next step, leading $\Delta_{ij}(t)$ to be decrease, the update-value is not adapted on the next step. In software, this can be achieved by storing $\frac{\partial E}{\partial w_{ij}}(t-1) = 0$, which prevents the change in the next step.

All update-values are initialized to a constant $\Delta_{ij} = \Delta_o$, which determines the size of the first weight change. A reasonable value is $\Delta_o = 0.1$. This is somewhat affected by the size

of the initial weights, but does not seem to be critical for simple problems. It is probably better to err in favor of choosing too small a value because an overly large value could lead to immediate node saturation. In [314], $\Delta_o = 0.001$ was used for the two-spirals problem, but values between 10^{-5} and 0.01 gave similar results.

The range of update-values is limited to $\Delta_{min} = 10^{-6}$ and $\Delta_{max} = 50$ to avoid floating-point underflow-overflow problems. Limiting Δ_{max} to smaller values, for example, 1, may give smoother decreases in the error at the cost of slower convergence. In [314], $\Delta_{max} = 0.1$ was used for the two-spirals problem.

The value $\eta^- = 0.5$ was chosen based on the reasoning that when the system overshoots a minimum, the minimum will be halfway between the current and previous weights, on average, so the step size should be reduced to half its previous value.

The value $\eta^+ = 1.2$ is a compromise. It should be large enough to allow fast growth in flat regions of the error function, but not so large that the system has to immediately reduce the update-value in the next step. The value 1.2 seems to work well on many problems and usually is not critical.

These default values seem to work well for most problems. In most cases, no changes are needed. In [315], only $\Delta_{max} = 0.001$ was changed for the two-spirals problem in order to avoid early saturation of the sigmoids. In most cases, Δ_o is the only other parameter that needs to be changed and its value is not critical as long as it is not too large.

Although it is not mentioned in the derivation, momentum can be used with beneficial effects on many problems. As usual, very high values of momentum may lead to instability.

In empirical comparisons, Rprop seems to be one of the faster and more reliable heuristic methods for a wide range of problems. There are, of course, cases where other methods do better, but Rprop is often a good choice for initial tests. For certain classification problems where the error criteria are satisfied as soon as all outputs are within a tolerance (e.g., 0.1) of their target values, it can be faster than second-order gradient methods such as conjugate gradient or Levenberg-Marquardt. This is problem dependent, however.

The success of Rprop can be explained, in part, by two factors. First, one reason for the slow convergence of gradient descent is that the gradient vanishes at a minimum so the step size becomes smaller and smaller as it nears the minimum. The error tends to decrease exponentially: fast at first, but slower later on. With Rprop, the step size does not depend on the magnitude of the gradient so learning does not slow to a crawl in the final stages.

Second, another problem with back-propagation in layered networks is that the derivatives tend to be attenuated as they propagate back from the output layer toward the inputs (see section 6.1.8). Each layer inserts a sigmoid derivative factor that is less than 1 (≤ 0.25 for sigmoids, ≤ 1 for tanh nodes) with the result that $|\partial E/\partial w|$ tends to be very small for weights far from the outputs and learning is correspondingly slow. Deep networks with many layers have been avoided for this reason because almost no learning occurs in the

initial layers. Heuristic methods for setting different learning rates for each layer have been investigated, but they are difficult to tune by hand and a fixed learning rate is not necessarily appropriate anyway. Rprop seems to work better than some other adaptive learning rate techniques in this case because the learning rate adjustments and weight updates depend only on the signs of the derivatives, not their magnitudes. Appropriate values can be found for each layer so early layers learn faster than they would otherwise and deep networks are not as difficult to train.

9.7 Quickprop

Fahlman's Quickprop [121] differs from most of the other methods mentioned here in that it is not an adaptive learning rate technique. Like back-propagation, it is a local method; each weight w is considered separately.

It is "based on 2 risky assumptions":

- that $E(w)$ for each weight can be approximated by a parabola that opens upward and
- that the change in slope of $E(w)$ for this weight is not affected by all the other weights that change at the same time.

The weight update rule is dominated by a quadratic term

$$\Delta w(t) = \frac{S(t)}{S(t-1) - S(t)} \Delta w(t-1) \tag{9.20}$$

where $S(t) = \frac{\partial E}{\partial w}(t)$. Call the $S(t)/(S(t-1) - S(t))$ term β. The numerator is the derivative of the error with respect to the weight and $(S(t-1) - S(t))/\Delta w(t-1)$ is a finite difference approximation of the second derivative. Together these approximate Newton's method for minimizing a one-dimensional function $f(x)$: $\Delta x = -f'(x)/f''(x)$. Sutton [363] suggested a similar update term.

Three cases occur:

1. If the current slope has the same sign but is somewhat smaller in magnitude than the previous one, then $\beta > 0$ and the weight will change again in the same direction. The size of the change will depend on how much the slope was reduced by the previous step.

2. If the current slope has a different sign from the previous slope, then the weight has crossed over the minimum and is now on the opposite side of the valley. Since $\beta < 0$, the next step will backtrack, landing somewhere between the current and previous positions.

3. The third case occurs when the current slope has the same sign as the previous slope, but is the same size or larger in magnitude. This indicates that the first "risky assumption"

was not met and could occur where the function is not well-approximated by a parabola or where the assumed parabola opens downward.

To avoid taking an infinite step or a backward uphill move in case 3, a "maximum growth factor" parameter μ is introduced. No weight change is allowed to be larger than μ times the previous weight change. A value of $\mu = 1.75$ is recommended. Chaotic behavior may result when it is too large,

For cases 1 and 3, an additional term $-\eta S(t)$ representing simple gradient descent is added to (9.20) to bootstrap the process when the previous change $\Delta w(t-1) = 0$. It is ignored in case 2 when the current slope is nonzero and differs in sign from the previous one since the quadratic term handles this case well.

In addition to these weight update rules, several other heuristics are sometimes used.

- It is argued that one of the reasons for the slow convergence of back-propagation is that the derivatives become very small when sigmoid node nonlinearities saturate. The sigmoid-prime heuristic simply adds 0.1 to the derivative of the sigmoid function so that it is always nonzero. This may accelerate learning in flat regions, but it may also make it difficult to settle to a minimum.
- Since the quadratic term may cause some weights to get very big, leading to floating-point overflow errors, a small decay term is added to the slope $S(t)$ calculated for each weight. Note that this is different from normal weight-decay, which acts directly on the weights.
- Finally, in some cases, a hyperbolic arctangent error function is used. That is, when back-propagating the error, the true derivative of the error with respect to the activation y of an output unit

$$\frac{\partial E}{\partial y} = -(d - y)$$

is replaced by

$$-\text{arctanh}(d - y).$$

Strictly speaking, this is not an error function, as it modifies the calculated derivative, rather than the error itself. This goes to $\pm\infty$ at ± 1 and greatly magnifies the error for output units that are far from their target values. It also tends to cancel the vanishing derivative for nodes that are saturated at the wrong value, but this case is already handled by the sigmoid-prime term. To avoid numerical problems, a value of 17 (-17) is used for inputs greater than 0.9999999 (less than -0.9999999). This assumes the errors are in $(-1, +1)$. Simple scale changes will be needed for tanh nonlinearities and other cases. This heuristic is somewhat nonstandard and is not used in most cases.

In empirical comparisons, quickprop is often one of the faster, more reliable methods and outperforms most other heuristic variations of back-propagation on a wide range of problems. Only Rprop seems to be consistently better; it is perhaps somewhat more reliable, has fewer parameters to tune, and seems to be less sensitive to their values.

Quickprop does have a fixed learning rate parameter η that needs to be chosen to suit the problem. It might be possible to use adaptive methods to control this, but no methods have been described.

9.8 Search Then Converge

Most of the other methods mentioned in this chapter are designed for batch-mode learning. The following describes an adaptive learning rate method for on-line learning.

As noted in section 5.3.2, the weight trajectory in on-line learning is stochastic and jitters around the error surface. This randomness helps search more of the weight space and makes the system more likely to find a good minimum, but it also keeps the weights from settling to a solution so the asymptotic error may be relatively high. The standard solution is to reduce the learning rate gradually as learning progresses.

The classic schedule used in stochastic approximation [318] is $\eta(t) = c/t$ where c is a constant. This guarantees asymptotic convergence and is optimal for c greater than some threshold c^*, which depends on the problem [97]. There are problems with this, however. Convergence is slow when c is small, but if c is increased too much then excessively large parameter changes may occur at small t.

Darken and Moody [97] proposed the "search then converge" schedule

$$\eta(t) = \frac{\eta_o}{1 + t/\tau}. \tag{9.21}$$

This avoids the unstable behavior at small t, yet still has the desired asymptotic behavior c/t for $t \gg \tau$. For $t \ll \tau$, $\eta(t) \approx \eta_o$ and the system behaves like normal on-line learning with a constant learning rate. It is hoped that by the time $t \approx \tau$ the system will converge to and then hover around a good minimum. At $t \approx \tau$, $\eta(t)$ begins decreasing to allow the weights to settle to the solution. For $t \gg \tau$, $\eta(t) \approx c/t$ where $c = \tau\eta_o$, and the learning rate approaches the optimum stochastic approximation schedule. The schedule [98]

$$\eta(t) = \eta_o \frac{1 + \frac{c}{\eta_o}\frac{t}{\tau}}{1 + \frac{c}{\eta_o}\frac{t}{\tau} + \tau\frac{t^2}{\tau^2}} \tag{9.22}$$

has similar behavior, but decreases $\eta(t)$ faster at intermediate values of t.

A defect of these schedules is that they require the user to choose the parameter c. The optimal value is $c^* \equiv 1/2\alpha$ where α is the smallest eigenvalue of the Hessian evaluated at the minimum [98]. c^* is usually unknown, however, because the minimum has not been found yet. In theory, the Hessian could be estimated and its eigenvalues calculated, but this is computationally intensive and may not be possible in an on-line learning situation. (The main advantages of on-line learning are its computational simplicity and small storage requirements).

Darken and Moody [98] propose a way to do an on-line estimate of whether $c < c^*$ by observing the trajectory of the weight vector. The idea is that when c is too small, successive weight update vectors will be highly correlated. Convergence is slow because the weight changes are small; the gradient changes little from one step to the next so successive weight updates tend to point in similar directions.

An estimate of the drift is

$$D(t) \equiv \sum_k d_k^2(t) \tag{9.23}$$

$$d_k(t) \equiv \sqrt{T}\,\frac{\langle \delta_k(t) \rangle_T}{\sqrt{\left\langle (\delta_k(t) - \langle \delta_k(t) \rangle_T)^2 \right\rangle_T}} \tag{9.24}$$

where $\delta_k(t)$ is the change in the kth component of the weight vector at time t, and the brackets $\langle \cdot \rangle_T$ indicate the average over T weight changes. In [98], $T = at$, $a \ll 1$. The numerator is the average parameter change. The denominator is the standard deviation of the weight changes and becomes small when weight updates are highly correlated over time. $D(t)$ grows like a power of t when c is too small but remains finite otherwise.

9.9 Fuzzy Control of Back-Propagation

Network training is a dynamic process and the algorithms described in this chapter can be viewed as control systems whose purpose is to accelerate learning while avoiding instability. Fuzzy logic is a convenient way to convert a set of heuristics into a working algorithm and has been used with success in many simple control applications.

The main difference between fuzzy logic and conventional Boolean logic is that fuzzy logic deals with propositions that can have varying degrees of membership between true and false. This is useful for control applications because it allows the behavior to be described by easily understood IF–THEN rules that are interpolated to give smooth transitions between regions where different rules are active. Mechanisms of fuzzy inferencing are described in many references, so the details will be omitted here.

Many papers have been written on applications of fuzzy logic to neural network training. Control of back-propagation using fuzzy logic has been proposed by Arabshahi et al. [10], Choi et al. [81], and others. Arabshahi et al. [10] controls a single global learning rate while Choi [81] extends control to include the momentum term. Comparisons are made with standard back-propagation and Jacobs' delta-bar-delta rule on the 3-bit parity problem.

The central idea is to establish a set of IF–THEN rules for parameter control that are implemented using fuzzy logic. In Arabshahi et al. [10], the rule antecedants (the IF parts) are expressions involving the error E and the change in error $CE = E_n - E_{n-1}$ from one iteration to the next while the THEN parts (the rule consequents) specify $\Delta\eta$, how the learning rate should change given the conditions described in the antecedent. One rule might say that when E is low, and CE is low then the learning rate should increase by a small amount, for example. If both E and CE were actually low then this rule would be satisfied and the consequent would be asserted strongly. For slightly different values of E or CE, however, for example, when E is "lowish, but not really low," the rule would be satisfied less well and its consequent would be asserted less strongly.

At any particular time, many different rules will be satisfied to varying degrees and conflicts between active rules suggesting different actions are resolved by methods of fuzzy inferencing to obtain a single overall output.

In simple applications like this, fuzzy logic is used for interpolation. That is, the designer specifies the desired response at selected points on a grid in the input space and relies on fuzzy inferencing to interpolate between the points in a reasonable way. The designer thereby avoids the sometimes difficult problem of finding a clever algorithm or function that generates the desired response from the given inputs. An advantage of fuzzy systems, and local interpolation methods in general, is that the effects of each rule are relatively localized; rules don't interact globally so it is relatively easy to tune individual rules to improve local performance without worry that this will cause problems in other areas.

A fuzzy implementation of a set of heuristics, for example, the delta-bar-delta heuristics, will usually have the same basic behavior as the heuristics implemented by other means, although there may be small-scale differences. That is, the final performance of the system is determined much more by the quality of the heuristics than by the mechanisms of how they are implemented. Factors such as which input variables are considered, how they are represented, and how they are presumed to interact in their effects on the response will have much stronger effects on performance than whether they are implemented by fuzzy logic or by some other means. Fuzzy logic does not substitute for understanding a problem, but it is a convenient way to convert understanding into a working algorithm.

9.10 Other Heuristics

9.10.1 Gradient Reuse

Hush and Salas [182] suggest stepping along the line of the computed gradient as long as the error continues to decrease. This is similar to Cauchy's method (section 10.5.2), but it does not search for the exact minimum on the line. It uses fixed size steps along the line rather than, say, bisection search. As in Cauchy's method, there is a savings since the gradient calculation is avoided for each successful step along the line. The step size is increased when the reuse rate is high (indicating that steps are too small) and it is decreased when it's low (because the step size it too large).

9.10.2 Gradient Correlation

Franzini [126], Chan and Fallside [67] and Schreibman and Norris [336] describe gradient correlation methods that monitor the angle between successive gradient vectors to control the learning rate. An advantage of this approach is that a major change in gradient direction can be detected and the learning rate reduced *before* taking a step, thus reducing the need to retract bad steps.

The gradient correlation measures the cosine of the angle between successive values of the gradient $\mathbf{g}(t)$

$$\cos(\theta) = \frac{\mathbf{g}(t-1)^T \mathbf{g}(t)}{\|\mathbf{g}(t-1)\| \, \|\mathbf{g}(t)\|}. \tag{9.25}$$

When the vectors are nearly parallel, $\cos\theta \approx +1$ and the learning rate can probably be increased. When $\cos\theta < 0$, the gradient has doubled back on itself to some extent and the learning rate should be decreased.

In [126] the learning rate is adjusted according to

$$\eta(t) = \begin{cases} \eta(t-1)\beta^+ \cos(\theta) & \text{if } \cos(\theta) > 0 \\ \eta(t-1)\beta^- & \text{otherwise} \end{cases} \tag{9.26}$$

where values of $\beta^+ = 1.005$ and $\beta^- = 0.8$ are suggested. This tends to keep η near the maximum value such that successive gradients are nearly parallel and eliminates the oscillatory cross-stitching behavior in ravines of the error surface. In the single-problem benchmark [9], this method was slightly slower than standard back-propagation to learn to classify the training set, but when used with momentum, it was the fastest method to reduce the error to near zero. Removal of the $\cos\theta$ term from the η adjustment rule was suggested.

In [336], the learning rate switches between high and low values based on the correlation. It is reduced to its minimum value (0.01) and momentum is set to 0 as soon as the

correlation became negative. The momentum returns to its normal value (0.9) gradually. Modifications may be needed to apply the idea in practice. In [9] it was slow to learn to classify correctly and could not further reduce the error to small values in the given amount of time, but generalization was said to be good.

9.10.3 Pattern Weighting Heuristics

A number of heuristics attempt to focus attention on the patterns with the worst errors. Often, this can be viewed as a modification of the error function to one which gives more emphasis to larger errors. If attention is focused only on the pattern with the largest error, the ideal result is to minimize the maximum error. Cater [65] gives each pattern a different weighting. Basically, the method identifies the pattern with the worst error and roughly doubles its learning rate in the next epoch.

The heuristic of "learn only if misclassified," used in [329] and later work, says that the actual output values do not really matter for classification problems as long as the classification is unambiguous. A tolerance band is defined and the error is considered to be zero for all outputs within the band. If the target is 0 and the output is 0.06, for example, the classification is obvious and there is no need to adjust the weights for this pattern.

Many methods like this can be considered as modifications of the error function and will lead to different solutions from the mean-squared-error function, in general. This may be a drawback if you actually want to optimize the mean-squared-error, but this normally is not the case for classification problems.

9.11 Remarks

All the methods summarized in this chapter were proposed to accelerate learning. It should be remembered that there are other factors affecting learning time that have not been considered here. As noted earlier, things such as network structure, data representation, choice of error function, and so on may have much stronger effects on performance and training time than the optimization method. Standard practices like the use of momentum, the use of tanh rather than sigmoid nodes, centering and normalization of inputs and outputs, the use of on-line versus batch updates, and so on also affect training times. If training time is a concern, it is best to explore these options before looking for fast training methods. Still, it may be necessary to train candidate networks in the process of comparing these factors and adaptive learning rate algorithms are a reasonable compromise between speed and robustness.

Often, adaptive learning rate methods are not any faster than standard back-propagation with *optimally tuned* parameters [239]. Even so, they effectively automate the search for good parameters and so may be more reliable and much easier to use. When the time

needed to select optimal parameters by hand is considered, adaptive methods may retain the speed advantage. In any case, they are usually much faster than back-propagation with poor parameter choices.

A potential problem with some adaptive methods is that they introduce additional parameters that need to be tuned. In the worst case, it may be no easier to find a good set of parameters than it is to find a good set of parameters for standard back-propagation. Another concern is that they may require storage of more information. Delta-bar-delta, for example, stores a separate learning rate and $\bar{\delta}$ for each weight. This is not a problem in small computerized simulations, but it may be a factor in applications using limited hardware (e.g., custom integrated circuits). Standard on-line back-propagation requires the least amount of storage.

Alpsan et al. [8] asked if modified back-propagation algorithms were worth the effort and concluded that many were not. They note that optimally tuned back-propagation is often as fast as any other method and that many of the adaptive methods are sensitive to parameters and no easier to tune than standard back-propagation. They considered delta-bar-delta, superSAB, and Vogl's method, among others, but not Rprop or quickprop.

At this point, Rprop and quickprop seem to be the favored methods. Rprop has fewer critical parameters and may be more reliable in general. Other methods will often do better on specific problems, however, so it may be worth experimenting.

When training time is very important, it is worth considering the standard optimization algorithms, some of which are reviewed in chapter 10. These may be much faster than simple variations of back-propagation in some cases. This is somewhat problem dependent, of course. The second order methods seem to be most helpful in the final stages of function approximation problems where it is necessary to reduce the error to very small values. Methods like conjugate gradient descent or Newton's method will converge very quickly *in the neighborhood* of a local minimum, but they are not necessarily any faster (and may be slower) than simpler first order methods in the initial search stages. For classification problems where training is stopped as soon as all outputs are correct within a tolerance, for example, 0.2, of the target values on all patterns, the methods of this chapter may be as fast or faster than conventional second-order optimization methods. If it is necessary to continue the search to locate the minimum very precisely, then it may be worth switching over to a more sophisticated second order method for the final tuning.

If training speed is extremely critical, it may also be worth considering a completely different sort of approximation system since back-propagation training of MLP networks is one of the slowest training methods for any approximation system [239]. Many other approximation methods can achieve similar error rates (on suitable problems) with much shorter training times. Nearest-neighbor methods, for example, require almost no training time (simply store the patterns) but have longer recall times. Decision trees and para-

metric classifiers can also be developed quickly when they are applicable. Within neural network models, alternatives include radial basis function networks, LVQ, and ART networks.

9.12 Other Notes

• The focus in this chapter has been on training speed. Generalization is a different issue and the fastest training method will not always give the best generalization. At best, speed of learning and quality of generalization are orthogonal issues—completely independent—and a fast training method would achieve the same generalization as another method except it would get there faster. In the best case, a fast training method will simply arrive sooner at the point where cross-validation says training should stop. Of course, if no specific steps are taken to ensure good generalization, then a fast method might generalize worse than a slower method as it may have more chance to overfit in the same amount of time.

There have been suggestions that some of the faster methods generalize worse than slower methods [8, 9], but this has not been studied much. There is some reason to expect techniques that take long steps (e.g., Newton's method) to generalize less well because they may go well past the point of overfitting before it can be detected by cross-validation on a test set. This does not have to occur, however, and it can be addressed by methods such as weight decay, pruning, and regularization penalty terms.

• Most of these methods are for batch mode training. Chen and Mars [76] describe an adaptive stepsize algorithm said to be suitable for on-line training. Modifications may be required, however, and tuning may be difficult [9].