5 Back-Propagation

Back-propagation is, by far, the most commonly used method for training multilayer feedforward networks. The term back-propagation refers to two different things. First, it describes a method to calculate the derivatives of the network training error with respect to the weights by a clever application of the derivative chain-rule. Second, it describes a training algorithm, basically equivalent to gradient descent optimization, for using those derivatives to adjust the weights to minimize the error.

The algorithm was popularized by Rumelhart, Hinton, and Williams [329, 330], although earlier work had been done by Werbos [390], Parker [295], and Le Cun ([89]; summarized in [90]). Together with the Hopfield network, it was responsible for much of the renewed interest in neural networks in the mid-1980s. Before back-propagation, most networks used nondifferentiable hard-limiting binary nonlinearities such as step functions and there were no well-known general methods for training multilayer networks. The breakthrough was perhaps not so much the application of the chain-rule, but the demonstration that layered networks of *differentiable* nonlinearities could perform useful nontrivial calculations and that they offer (in some implementations) attractive features such as fast response, fault tolerance, the ability to "learn" from examples, and some ability to generalize beyond the training data.

As a training algorithm, the purpose of back-propagation is to adjust the network weights so the network produces the desired output in response to every input pattern in a predetermined set of training patterns. It is a *supervised* algorithm in the sense that, for every input pattern, there is an externally specified "correct" output which acts as a target for the network to imitate. Any difference between the network output and the target is treated as an error to be minimized. A "teacher" must decide which patterns to include in the training set and specify the correct output for each. It is an *off-line* algorithm in the sense that training and normal operation occur at different times. In the usual case, training could be considered part of the "manufacturing" process wherein the network is trained once for a particular function, then frozen and put into operation. Normally, no further learning occurs after the initial training phase.

To train a network, it is necessary to have a set of input patterns and corresponding desired outputs, plus an error function (cost function) that measures the "cost" of differences between network outputs and desired values. The basic steps are these.

1. Present a training pattern and propagate it through the network to obtain the outputs.
2. Compare the outputs with the desired values and calculate the error.
3. Calculate the derivatives $\partial E / \partial w_{ij}$ of the error with respect to the weights.
4. Adjust the weights to minimize the error.
5. Repeat until the error is acceptably small or time is exhausted.

The error function measures the cost of differences between the network outputs and the desired values. The sum-of-squares function, below, is a common choice.

$$E_{SSE} = \sum_p \sum_i (d_{pi} - y_{pi})^2 \tag{5.1}$$

Here p indexes the patterns in the training set, i indexes the output nodes, and d_{pi} and y_{pi} are, respectively, the target and actual network output for the ith output node on the pth pattern. The mean-squared-error

$$E_{MSE} = \frac{1}{PN} E_{SSE} \tag{5.2}$$

normalizes E_{SSE} for the number of training patterns P and network outputs N. Advantages of the SSE and MSE functions include easy differentiability and the fact that the cost depends only on the magnitude of the error. In particular, a deviation of a given magnitude has the same cost independent of the input pattern and independent of errors on other outputs. For classification problems, logarithmic or cross-entropy error functions (section 2.1) are sometimes used. For real-world applications, the cost function may be specialized to assign different costs to different sorts of deviations; similar errors on different input patterns may have different costs and the cost of an error on one output could depend on the errors on other outputs.

5.1 Preliminaries

Back-propagation can be applied to any feedforward network with differentiable activation functions. In particular, it is not necessary that it have a layered structure. An arbitrary feedforward network will be assumed in the following.

Feedforward Indexing For simplicity, assume the nodes are indexed so that $i > j$ implies that node i follows node j in terms of dependency. That is, the state of node i may depend, perhaps indirectly, on the state of node j, but node $j < i$ does not depend on node i. Such an index order is possible in any feedforward network, though it will not be unique in general. The advantage of this format is that it works in any feedforward network, including those with irregular structure and short-cut (layer skipping) connections. In simulations, it also lets us avoid the need to deal with each layer separately, keeping track of layer indexes. Of course, this indexing scheme is compatible with standard layered structures.

Because the dependencies are transmitted by the connection weights, connections are allowed from nodes with low indexes to nodes with higher indexes, but not vice versa. If

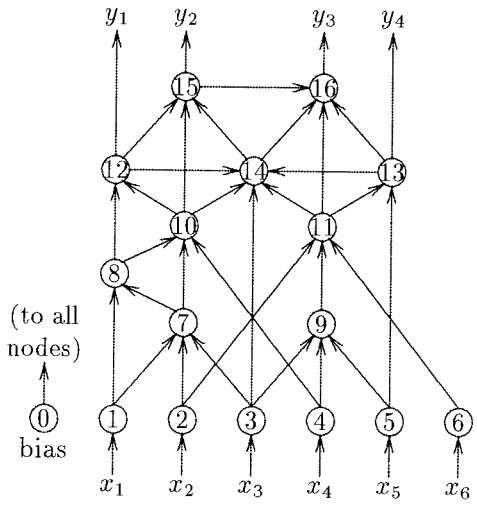

Figure 5.1
Feedforward indexing in an unlayered network. The nodes in a feedforward network can always be indexed so that $i > j$ if the state of node i depends on the state of node j (perhaps indirectly). Arbitrary connections are allowed from nodes with low indexes to nodes with higher indexes, but not vice versa; $i > j$ implies $w_{ji} \equiv 0$. This network has no particular function, but illustrates short-cut connections, apparently lateral (but still feedforward) connections, and the fact that outputs can be take from internal nodes.

w_{ij} denotes the weight *to* node i *from* node j then any forward link w_{ij}, $j < i$ is allowed, but backward links are prohibited, $w_{ji} \equiv 0$. Figure 5.1 illustrates a possibility. Normally, the system inputs and the bias node will have low indexes since they potentially affect all other nodes and outputs will have high indexes.

5.1.1 Forward Propagation

In the forward pass, the network computes an output based on its current inputs. Each node i computes a weighted sum a_i of its inputs and passes this through a nonlinearity to obtain the node output y_i (see figure 5.2)

$$a_i = \sum_{j<i} w_{ij} y_j \tag{5.3}$$

$$y_i = f(a_i). \tag{5.4}$$

Normally f is a bounded monotonic function such as the tanh or sigmoid. Arbitrary differentiable functions can be used, but sigmoid-like "squashing" functions are standard. The index j in the sum runs over all indexes $j < i$ of nodes that could send input to node i.

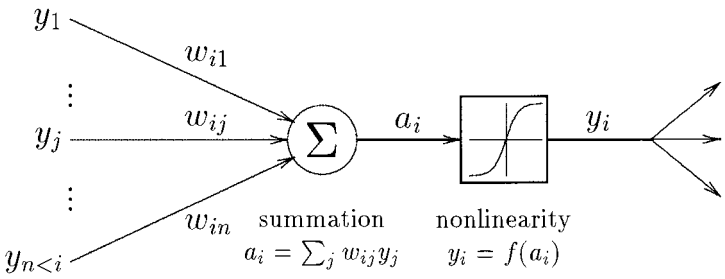

Figure 5.2
Forward propagation. In the forward pass, the input pattern is propagated through the network to obtain the output. Each node computes a weighted sum of its inputs and passes this through a nonlinearity, typically a sigmoid or tanh function.

If there is no connection from node j, weight w_{ij} is taken to be 0. As usual, it is assumed that there is a bias node with constant activation, $y_{bias} = 1$, to avoid the need for special handling of the bias weights.

Every node is evaluated in order, starting with the first hidden node and continuing to the last output node. In layered networks, the first hidden layer is updated based on the external inputs, the second hidden layer is updated based on the outputs of the first hidden layer, and so on to the output layer which is updated based on the outputs of the last hidden layer. In software simulations, it is sufficient to evaluate the nodes in order by node index. Because node i does not depend on any nodes $k > i$, all inputs to node i will be valid when it is evaluated. At the end of the sweep, the system outputs will be available at the output nodes.

5.1.2 Error Calculation

Unless the network is perfectly trained, the network outputs will differ somewhat from the desired outputs. The significance of these differences is measured by an error (or cost) function E. In the following, we use the SSE error function

$$E = \frac{1}{2} \sum_p \sum_i (d_{pi} - y_{pi})^2 \tag{5.5}$$

where p indexes the patterns in the training set, i indexes the output nodes, and d_{pi} and y_{pi} are, respectively, the desired target and actual network output for the ith output node on the pth pattern. The $\frac{1}{2}$ factor suppresses a factor of 2 later on. One of the reasons SSE is convenient is that errors on different patterns and different outputs are independent; the overall error is just the sum of the individual squared errors

$$E = \sum_p E_p \tag{5.6}$$

$$E_p = \frac{1}{2} \sum_i (d_{pi} - y_{pi})^2. \tag{5.7}$$

5.2 Back-Propagation: The Derivative Calculation

Having obtained the outputs and calculated the error, the next step is to calculate the derivative of the error with respect to the weights. First we note that $E_{SSE} = \sum_p E_p$ is just the sum of the individual pattern errors so the total derivative is just the sum of the per-pattern derivatives

$$\frac{\partial E}{\partial w_{ij}} = \sum_p \frac{\partial E_p}{\partial w_{ij}}. \tag{5.8}$$

The thing that makes back-propagation (the derivative calculation) efficient is how the operation is decomposed and the ordering of the steps. The derivative can be written

$$\frac{\partial E_p}{\partial w_{ij}} = \sum_k \frac{\partial E_p}{\partial a_k} \frac{\partial a_k}{\partial w_{ij}} \tag{5.9}$$

where the index k runs over all output nodes and a_j is the weighted-sum input for node j obtained in equation 5.3. It is convenient to first calculate a value δ_i for each node i

$$\delta_i = \frac{\partial E_p}{\partial a_i} \tag{5.10}$$

$$= \frac{\partial E_p}{\partial y_i} \frac{\partial y_i}{\partial a_i}, \tag{5.11}$$

which measures the contribution of a_i to the error on the current pattern. For simplicity, pattern indexes p are omitted on y_i, a_i, and other variables below.

For output nodes, $\partial E_p / \partial a_k$ is obtained directly

$$\delta_k = -(d_{pk} - y_{pk}) f_k'. \qquad \text{(for output nodes)} \tag{5.12}$$

The first term is obtained from equation 5.7,

$$\frac{\partial E_p}{\partial y_k} = -(d_{pk} - y_{pk}). \tag{5.13}$$

(This is the SSE result; different expressions will be obtained for other error functions.) The second term

$$\frac{\partial y_k}{\partial a_k} = f'(a_k) \tag{5.14}$$

is just the slope $f'_k \equiv f'(a_k)$ of the node nonlinearity at its current activation value. The sigmoid is convenient to use because f' is a simple function of the node output: $f'(a_k) = y(1-y)$, where $y = f(a_k)$. The tanh function is also convenient, $f'(a_k) = 1 - y^2$.

For hidden nodes, δ_i is obtained indirectly. Hidden nodes can influence the error only through their effect on the nodes k to which they send output connections so

$$\delta_i = \frac{\partial E_p}{\partial a_i} \quad = \quad \sum_k \frac{\partial E_p}{\partial a_k}\frac{\partial a_k}{\partial a_i}. \tag{5.15}$$

But the first factor is just the δ_k of node k so

$$\delta_i = \sum_k \delta_k \frac{\partial a_k}{\partial a_i}. \tag{5.16}$$

The second factor is obtained by noting that if node i connects directly to node k then $\partial a_k / \partial a_i = f'_i w_{ki}$, otherwise it is zero. So we end up with

$$\delta_i = f'_i \sum_k w_{ki}\delta_k \tag{5.17}$$

(for hidden nodes). In other words, δ_i is a weighted sum of the δ_k values of nodes k to which it has connections w_{ki} (see figure 5.3).

Because δ_k must be calculated before δ_i, $i < k$, the process starts at the output nodes and works backward toward the inputs, hence the name "back-propagation." First δ values are calculated for the output nodes, then values are calculated for nodes that send connections to the outputs, then values are calculated for nodes two steps removed from the outputs, and so forth.

To summarize so far,

$$\delta_i = \begin{cases} -(d_{pi} - y_{pi}) f'_i & \text{(for output nodes)} \\ f'_i \sum_k w_{ki}\delta_k & \text{(for hidden nodes)} \end{cases} \tag{5.18}$$

For output nodes, δ_i depends only on the error $d_i - y_i$ and the local slope f'_i of the node activation function. For hidden nodes, δ_i is a weighted sum of the δs of all the nodes it connects to, times its own slope f'_i. Because of the way the nodes are indexed, all delta

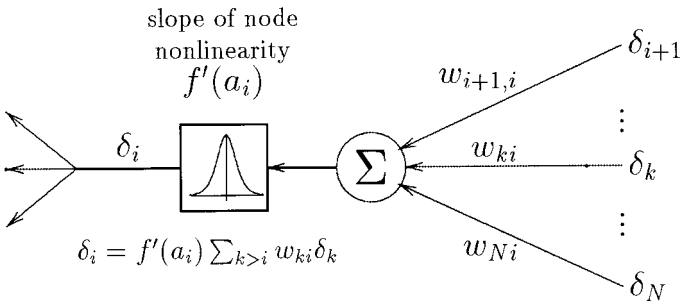

Figure 5.3
Backward propagation. Node deltas are calculated in the backward pass, starting at the output nodes and proceeding backwards. At each hidden node i, δ_i is calculated as a weighted linear combination of values δ_k, $k > i$, of the nodes k to which node i sends outputs. Note that the δ values travel backward against the normal direction of the connecting links.

values can be updated in a single sweep through the nodes in reverse order. In layered networks, delta values are first evaluated at the output nodes based on the current pattern errors, the last hidden layer is then evaluated based on the output delta values, the second-to-last hidden layer is updated based on the values of the last hidden layer, and so on backwards to the input layer. Normally it is not necessary to calculate delta values for the input layer, so the process usually stops with the first hidden layer.

Having obtained the node deltas, it is an easy step to find the partial derivatives $\partial E_p/\partial w$ with respect to the weights. The second term in (5.9) is $\partial a_k/\partial w_{ij}$. Because a_k is a simple linear sum, this is zero if $k \neq i$; otherwise

$$\frac{\partial a_i}{\partial w_{ij}} = y_j \tag{5.19}$$

where y_j is the output activation of node j. Finally, from (5.9), the derivative of pattern error E_p with respect to weight w_{ij} is then

$$\frac{\partial E_p}{\partial w_{ij}} = \delta_i y_j, \tag{5.20}$$

the product of the delta value at node i and the output of node j.

Derivatives with Respect to the Inputs Normally, delta values are not calculated for the input nodes because they are not needed to adjust the weights. Notice, however, that for input nodes $\delta_i = \partial E/\partial a_i$ is the derivative of the error with respect to the input. Also, if the network has a single output y then setting $E = 1$ and back-propagating gives the derivative

of the output with respect to the input, $\partial y/\partial x$. There are, of course, useful applications for these derivatives. They can be used, for example, in inverse problems where we seek input values that produce a particular output.

A Finite-Difference Approximation As an alternative, the derivative can be estimated by a finite-difference approximation [44: 147]

$$\frac{\partial E_p}{\partial w_{ij}} = \frac{E_p(w_{ij} + \epsilon) - E_p(w_{ij} - \epsilon)}{2\epsilon} + O(\epsilon^2) \tag{5.21}$$

where $\epsilon \ll 1$ is a small offset. This is a weight perturbation technique [191, 192, 125]. Each weight must be perturbed twice and the error must be reevaluated for each perturbation. In a serial implementation, the error measurement takes $O(W)$ time so the total time required scales like $O(W^2)$, where W is the number of weights in the network. This is slower than back-propagation, which takes $O(W)$ time to find the derivative, but it is robust and simple to implement.

The method described perturbs each weight separately. A node perturbation technique is more efficient [44]. (The madeline III learning rule [403] is similar.) Recall from equation 5.10 that δ_i is the derivative of the error with respect to the node input sum, a_i. Instead of perturbing the weights, the a_i values of the hidden nodes are perturbed to obtain an approximation of the node deltas

$$\begin{aligned}\delta_i &\equiv \frac{\partial E_p}{\partial a_i} \\ &= \frac{E_p(a_i + \epsilon) - E_p(a_i - \epsilon)}{2\epsilon} + O(\epsilon^2).\end{aligned} \tag{5.22}$$

Then equation 5.20 is used to calculate the gradient with respect to the weights. Each node must be perturbed twice and the error reevaluated. If there are H hidden nodes and W weights, this takes $O(2HW)$ steps. Calculation of E_p takes $O(W)$ time so the overall time scales like $O(HW)$. Depending on the relative sizes of H and W, this can be much faster than the $O(W^2)$ time of the previous method, but it is still longer than the $O(W)$ time of back-propagation. Summed weight perturbation [125] is a hybrid of weight and node perturbation methods.

Although finite-difference techniques are slower than back-propagation, they are simple to implement and useful for verify the correctness of more efficient calculations. They are also useful for training systems where analytic derivative calculation is not possible. Electronic circuit implementations, for example, may lack dedicated circuitry to do the back-propagation calculations. Finite-difference methods make it possible to estimate the

derivatives using only feedforward calculations. In addition, they automatically account for nonideal circuit effects and faults that may occur in real systems.

5.3 Back-Propagation: The Weight Update Algorithm

Having obtained the derivatives, the next step is to update the weights so as to decrease the error. As noted earlier, the term back-propagation refers to (1) an efficient method to calculate the derivatives $\partial E/\partial w$ and (2) an optimization algorithm that uses those derivatives to adjust the weights to reduce the error. Having obtained the derivatives, we have the choice of continuing with back-propagation, the optimization algorithm, or using one of many alternative optimization methods that may be better adapted to the given problem.

Back-propagation (the optimization method) is basically equivalent to gradient descent. By definition, the gradient of E points in the direction that increases E the fastest. In order to minimize E, the weights are adjusted in the opposite direction. The weight update formula is

$$\Delta w_{ij} = -\eta \frac{\partial E}{\partial w_{ij}} \tag{5.23}$$

where the *learning rate* $\eta > 0$ is a small positive constant. Sometimes η is also called the *step size* parameter. If the derivative is positive (so increases in w causes increases in E) then the weight change is negative and vice versa. This approaches pure gradient descent when η is infinitesimal. Very small η values mean very long learning times though so larger rates are usually used. Typical values are in the range $0.05 < \eta < 0.75$. (This is just a rule of thumb, however; see section 6.1 for more discussion.)

The network is usually initialized with small random weights. Values are often selected uniformly from a range $[-a, +a]$ where $0.1 < a < 2$ typically. Random values are needed to break symmetry while small values are necessary to avoid immediate saturation of the sigmoid nonlinearities. Chapter 7 considers initialization methods in more detail.

5.3.1 Batch Learning

There are two basic weight-update variations, *batch-mode* and *on-line*. In batch-mode, every pattern p is evaluated to obtain the derivative terms $\partial E_p/\partial w$; these are summed to obtain the total derivative

$$\frac{\partial E}{\partial w} = \sum_p \frac{\partial E_p}{\partial w}. \tag{5.24}$$

and only then are the weights updated. This comes from the derivative rule for sums. The individual $\partial E_p/\partial w$ terms are obtained by application of the method of section 5.2 to each pattern p.

The basic steps are

- For every pattern p in the training set,

1. apply pattern p and forward propagate to obtain network outputs, and

2. calculate the pattern error E_p and back-propagate to obtain the single-pattern derivatives $\partial E_p/\partial w$.

- Add up all the single-pattern terms to get the total derivative.
- Update the weights

$$w(t+1) = w(t) - \eta \frac{\partial E}{\partial w}.$$

- Repeat.

Each such pass through the training set is called an *epoch*.

The gradient is calculated exactly and weight changes are proportional to the gradient so batch-mode learning approximates gradient descent when the step size η is small (figure 5.4). In general, each weight update reduces the error by only a small amount so many epochs are needed to minimize the error.

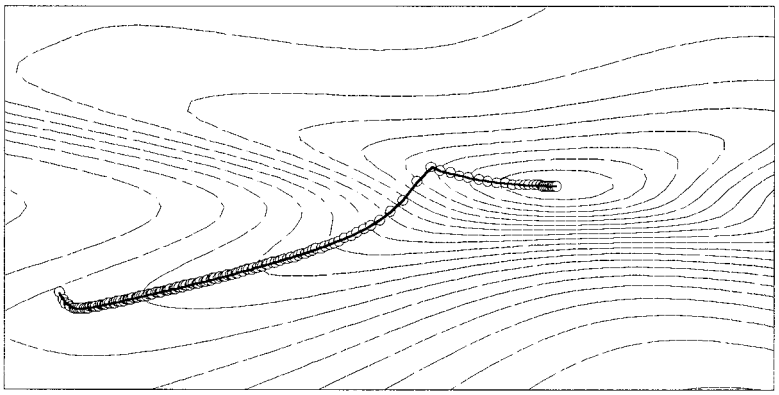

Figure 5.4
Batch-mode back-propagation is a close approximation to true gradient descent. At each step, the weights are adjusted in the direction that minimizes the error the fastest. When the learning rate is small, the weights trace a smooth trajectory down the gradient of the error surface.

5.3.2 On-Line Learning

An alternative to batch-mode is *on-line* or *pattern-mode* learning. In on-line learning, the weights are updated after each pattern presentation. Generally, a pattern p is chosen at random and presented to the network. The output is compared with the target for that pattern and the errors are back-propagated to obtain the single-pattern derivative $\partial E_p/\partial w$. The weights are then updated immediately, using the gradient of the single-pattern error. Generally, the patterns are presented in a random, constantly changing order to avoid cyclic effects.

The steps are:

- Pick a pattern p at random from the training set,

1. apply pattern p and forward propagate to obtain network outputs, and
2. calculate the pattern error E_p and back-propagate to obtain the single-pattern derivatives $\partial E_p/\partial w$.

- Update the weights immediately using the single-pattern derivative

$$w(t+1) = w(t) - \eta \frac{\partial E_p}{\partial w}.$$

- Repeat.

An advantage of this approach is that there is no need to store and sum the individual $\partial E_p/\partial w$ contributions; each pattern derivative is evaluated, used immediately, and then discarded. This may make hardware implementation easier when resources are limited. Another possible advantage is that many more weight updates occur in a given amount of time. If the training set contains M patterns, for example, on-line learning would make M weight changes in the time that batch-mode learning makes only one.

A possible disadvantage (from an analysis standpoint, at least) is that this is no longer a simple approximation to gradient descent. Figure 5.5 illustrates the relationship between the true gradient and the single-pattern terms in a typical case. The single-pattern derivatives can be viewed as noisy estimates of the true gradient. As a group, they sum to the gradient but each has a random deviation that need not be small. When the gradient is strong, the average single-pattern derivative has a positive projection on the gradient (because it is the sum of the single-pattern terms) so the error usually decreases after most weight changes. Still, there may be terms with negative projections or large orthogonal deviations which may cause the error to increase after some updates. On average though, the weight change will at least move downhill even if it doesn't take the most direct path.

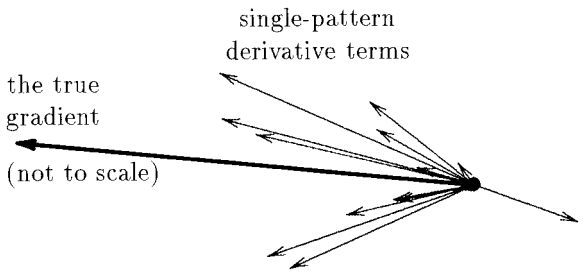

Figure 5.5
In on-line learning, weight updates occur after each pattern presentation. The single-pattern derivative terms can be viewed as noisy estimates of the gradient. They are not parallel to it in general, but on average they have a positive projection onto it (because the gradient is the sum of the single-pattern terms) so the error usually decreases after most weight changes. Some terms may have negative projections or large orthogonal deviations, however, which may cause the error to increase occasionally.

On-line learning differs from pure gradient descent in that the sum (5.24) is never evaluated exactly because the weights change after each pattern so the individual terms are evaluated at different points. The difference is minimal when η is very small; the weights won't change much between steps so the effect after all patterns have been evaluated is approximately the same as if all terms had been evaluated at a single point and summed to perform a single weight change which has the same overall result.

Very small learning rates tend to make learning very slow, however, so larger values are often used and the stochastic elements become important. Instead of following a smooth trajectory down the gradient, the weight vector tends to jitter around the $E(w)$ surface, mostly moving downhill, but occasionally jumping uphill (figure 5.6). To a first approximation, the magnitude of the jitter is proportional to the learning rate η. The randomness arises because training patterns are selected in a random, constantly changing order. Cyclic, fixed orders are generally avoided because of the possibility of convergence to a limit cycle (figure 5.7).

This randomness has advantages and disadvantages. On the plus side, it gives the algorithm some stochastic search properties. When pure gradient descent arrives at a local minimum, it is simply stuck. In on-line (per-pattern) learning, however, the weight state tends to jitter around its equilibrium value. Instead of sitting quietly at the minimum, it visits many nearby points and occasionally, if chance allows, may bounce out of a poor minimum and find a better solution. Thus, on-line learning may have a better chance of finding a global minimum than true gradient descent. On the minus side, the weight vector never settles to a stable value. Having found a good minimum, it may then wander off. Also on the minus side, when the jitter is very large, it may completely hide any deterministic gradient information so the system may be unable to follow subtle paths in the $E(w)$ surface.

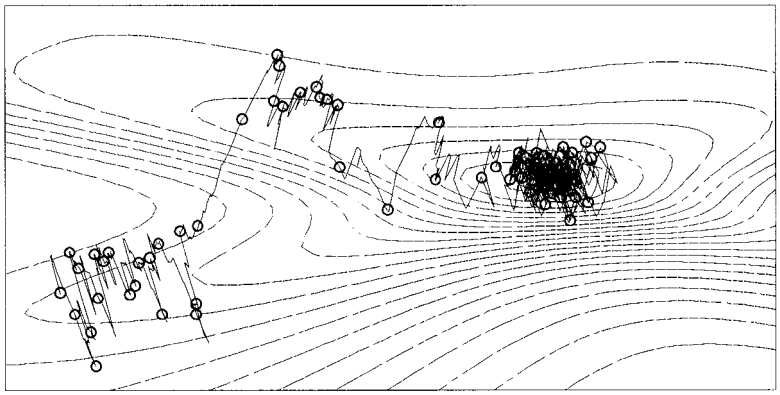

Figure 5.6
In on-line learning, patterns are generally presented in a random, changing order and weights are updated after each pattern presentation. Instead of smoothly rolling down the error gradient, the weight vector dances along a semi-random path, mostly moving downhill, but occasionally jumping uphill. Upon reaching a low spot, the weight vector jitters around the minimum but is unable to settle unless the step size is reduced. Circles show the weights at the start of each epoch and line segments show the single-pattern steps within each epoch.

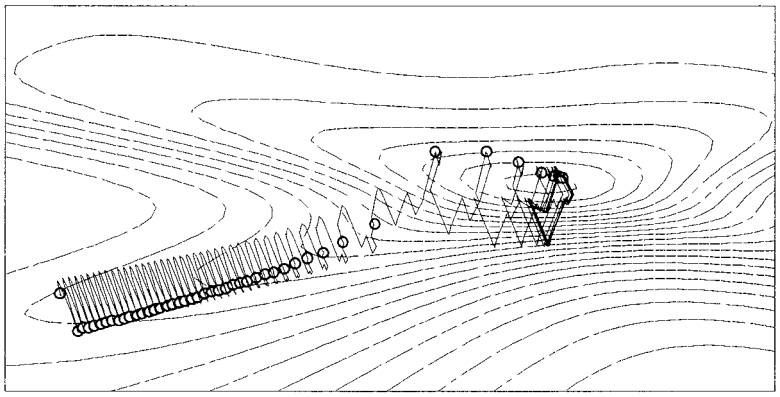

Figure 5.7
When patterns are presented in a cyclic order during on-line learning, as above, the sequence of steps in epoch $t + 1$ tends to be similar to the sequence in epoch t so the weight trajectory has a semi-periodic behavior. A danger is that the trajectory will converge to a limit cycle, as shown, and be unable to reach the minimum. One way to break the cycle is to change the pattern selection order. Reduction of the learning rate may also break the cycle or, if done gradually, may reduce the "diameter," allowing it to close in around the minimum. Circles show the weights at the start of each epoch and line segments show the single-pattern steps within each epoch.

It is common, therefore, to adjust the learning rate as training progresses. The simplest scheme is to start with an intermediate value, let the system train to approximate convergence, and then gradually reduce the learning rate to zero to allow the system to settle to the minimum. The learning rate can also be adjusted dynamically depending on conditions encountered during training. It is desirable to maintain a balance between stochastic search and efficient progress down the error gradient. The learning rate should not be so large that any single weight update (or likely sequence of updates) can move the weight state to a completely new area of the weight space, but it should not be so small that the system merely approximates gradient descent. If a large majority of single-pattern vectors have a common direction (positive projection on the average vector, the gradient) then the learning rate can probably be increased. If the single-pattern vectors have no apparent common direction then the learning rate should be reduced. This will occur near a minimum, where the gradient goes to zero because the single-pattern vectors cancel. It may also occur at the bottom of a "ravine" in the error surface, where the single-pattern vectors often group into two bundles pointing in opposite directions across the long axis of the ravine. In these cases, smaller learning rates would allow the system to settle to the minimum. Section 9.8 describes the "search then converge" algorithm, an adaptive learning method that controls the learning rate automatically.

A side note about terminology: The label "on-line learning" may be confusing because it implies that learning may occur in the field during normal operation and that it is not necessary to take the system off-line for training. But on-line learning, like batch-mode learning, is normally done off-line during a separate training phase with controlled data sets. The label "pattern-mode learning" is sometimes used instead.

5.4 Common Modifications

5.4.1 Momentum

A common modification of the basic weight update rule is the addition of a *momentum* term. The idea is to stabilize the weight trajectory by making the weight change a combination of the gradient-decreasing term in equation 5.23 plus a fraction of the previous weight change. The modified weight change formula is

$$\Delta \mathbf{w}(t) = -\eta \frac{\partial E}{\partial \mathbf{w}}(t) + \alpha \Delta \mathbf{w}(t-1). \tag{5.25}$$

That is, the weight change $\Delta \mathbf{w}(t)$ is a combination of a step down the negative gradient, $-\eta \frac{\partial E}{\partial \mathbf{w}}(t)$, plus a fraction $0 \leq \alpha < 1$ of the previous weight change. Typical values are $0 \leq \alpha < 0.9$.

This gives the system a certain amount of inertia since the weight vector will tend to continue moving in the same direction unless opposed by the gradient term. Effects of momentum are considered in more detail in section 6.2. Briefly, momentum tends to damp oscillations in the weight trajectory and accelerate learning in regions where $\partial E/\partial w$ is small.

5.4.2 Weight Decay

Another common modification of the weight update rule is the addition of a *weight decay* term. Weight decay is sometimes used to help adjust the complexity of the network to the difficulty of the problem. The idea is that if the network is overly complex, then it should be possible to delete many weights without increasing the error significantly. One way to do this is to give the weights a tendency to drift to zero by reducing their magnitudes slightly at each iteration. The update rule with weight decay is then

$$\Delta \mathbf{w}(t) = -\eta \frac{\partial E}{\partial \mathbf{w}}(t) - \rho \mathbf{w}(t). \tag{5.26}$$

where $0 \leq \rho \ll 1$ is the weight decay parameter. If $\partial E/\partial w_i = 0$ for some weight w_i, then w_i will decay to zero exponentially. Otherwise, if the weight really is necessary then $\partial E/\partial w_i$ will be nonzero and the two terms will balance at some point, preventing the weight from decaying to zero. Weight decay is considered in more detail in sections 6.2.4 and 16.5 and chapter 13.

5.5 Pseudocode Examples

At present, most artificial neural networks exist only as simulations on serial computers. The following samples illustrate the basic steps of back-propagation in 'C' pseudocode. Note, the purpose of the code is only to illustrate the algorithm. Real code would have to include many distracting details!

Forward Propagation In the feedforward step, an input pattern is propagated through the network to obtain an output. In 'C' pseudocode, this might look like

```
void forward_propagate( double *input_pattern )
   {
   /* copy pattern to input nodes */
   for( i=0; i<number_of_inputs; i++ )
      node_output[i+1] = pattern[i];
   node_output[0] = 1;    /* set bias to 1 */
```

```
   /* compute outputs of the remaining nodes */
   for( i=first_hidden_index; i<number_of_nodes; i++ ) {
      double sum = 0;
      for( j=0; j<i; j++ )
         sum += weight[i][j] * node_output[j];
      node_output[i] = sigmoid( sum );
      }
   }
```

When the function returns, the network outputs are available in the values of the output nodes. Because of the feedforward node indexing scheme, each `node_output[j]` is ready and available when it is needed as input to following nodes $i > j$. The bias node has index 0 and input nodes have indices from 1 to `number_of_inputs`. The rest of the nodes are indexed in feedforward order, with the output nodes last. The entire network can thus be evaluated in a single sweep through the nodes without extra bookkeeping to keep track of layers or short-cut connections.

For simplicity, the weight matrix is square with slots for all possible connections (including unallowed backward connections). Mathematically, weight w_{ij} can be treated as zero if there is no connection from node j to node i. In practice, of course, it would be more efficient to store connection information for each node so that only those weights that actually exist are examined. Similar details are ignored here for simplicity.

Backward Error Propagation The derivatives of the error on the current pattern with respect to the weights are calculated in the back-propagation step. The following pseudocode shows how this might be implemented assuming the network response to the pattern has just been calculated by forward propagation. The `*targets` argument points to an array of target values. For simplicity, assume there is one array element per network node so the same index can be used to access nodes and their target values. A sigmoid node function is assumed.

```
void backprop_node_deltas( double *targets )
   {
   /* calculate node deltas for output nodes */
   for( i=last_output_node; i>=first_output_node; i-- ) {
      double err = targets[i] - node_value[i];
      delta[i] = err * node_value[i]*(1-node_value[i]);
                     /* (sigmoid slope term) */
      }
```

```
   /* then calculate deltas for hidden nodes, working backwards */
   for( i=last_hidden_node; i>=first_hidden_node; i-- ) {
      delta[i] = 0;
      for( k=i+1; k<=last_output_node; k++ )
         delta[i] += weight[k][i] * delta[k];
      delta[i] *= node_value[i]*(1-node_value[i]);
                  /* (sigmoid slope term) */
      }
   }
```

Batch-Mode Weight Update The code just described would normally be included in a larger loop to add up the weight change contributions from each pattern. One epoch of batch-mode training could be done as follows.

```
void backprop_batch_one_epoch(void)
   {
   /* clear the dEdW accumulators */
   for( i=0; i<number_of_nodes; i++ )
      for( j=0; j<i; j++ )
         dEdW[i][j] = 0;

   /* add up dEdW contributions from each pattern */
   for( ip=0; ip<number_of_patterns; ip++ ) {
      forward_propagate( pattern[ip] );
      backprop_node_deltas( targets[ip] );
      for( i=0; i<number_of_nodes; i++ )
         for( j=0; j<i; j++ )
            if ( weight_really_exists(i,j) )
               dEdW[i][j] += delta[i] * node_value[j];
      }

   /* change the weights */
   for( i=0; i<number_of_nodes; i++ )
      for( j=0; j<i; j++ )
         weights[i][j] += learning_rate * dEdW[i][j];
   }
```

On-Line Weight Update On-line training is even simpler because there is no need to clear and accumulate the single-pattern derivative terms. One pass through the training set in on-line mode could be done as follows.

```
void backprop_online_one_epoch(void)
   {
   /* for each pattern... */
   for( ip=0; ip<number_of_patterns; ip++ ) {
      index = choose_one_randomly();
      forward_propagate( pattern[index] );
      backprop_node_deltas( targets[index] );

      /* change the weights */
      for( i=0; i<number_of_nodes; i++ )
         for( j=0; j<i; j++ )
            if ( weight_really_exists(i,j) )
               weights[i][j] += learning_rate * delta[i]*node_value[j];
      }
   }
```

5.6 Remarks

To reiterate, back-propagation refers to (1) an efficient method to calculate derivatives of the training error with respect to the weights, and (2) a training algorithm that uses those derivatives to adjust the weights to minimize the error. Other optimization methods can be used to update the weights, so it is not uncommon to hear of a network trained by, say, the conjugate gradient method using back-propagation to calculate the gradient.

Confusion may arise because the term *back-propagation network* is sometimes used to refer to a standard multilayer network trained by back-propagation. Although most people understand the term, it is not strictly correct because (1) the same network could be trained by other methods and (2) back-propagation can be used to train other types of networks. Back-propagation is simply one method, albeit the most common, for training these types of networks.

Although the algorithm is usually derived for a fully connected layered network, it can be applied to networks with arbitrary feedforward structure. Any number of weights can be held constant. It is also possible for internal nodes to have targets. This may be useful when it is known that the network must compute some intermediate function in order to calculate the final desired output. In this case, the node delta is the sum of deltas obtained by considering it as both an output node and a hidden node. (Section 16.10 discusses the use of this sort of information as hint functions.)

The thing that makes back-propagation more than a simple application of the derivative chain-rule is the ordering of the calculations. A naive application of the chain-rule sepa-

rately for each of the W weights in a network could result in an $O(W^2)$ time algorithm: $O(W)$ time to calculate $\partial E/\partial w$ for a single weight multiplied by the W weights in the network. Back-propagation, in contrast, is an $O(W)$ time algorithm. Bishop [44] likens the practical importance of this difference to that of the fast Fourier transform (FFT).

A side note: In this book we mainly discuss feedforward networks of sigmoidal units and a large part is devoted to back-propagation and its variations. Although back-propagation is one the most popular learning techniques for neural networks, it is a mistake to equate the entire field with back-propagation in layered perceptrons. Even considering only non-biological networks, there are many optimization methods besides back-propagation and there are many structures in addition to layered sigmoidal networks. Most of the properties that make artificial neural networks attractive (e.g., potential parallelism, fast response, fault tolerance, learning from examples, generalization, etc.) have nothing to do with back-propagation per se. The algorithm is simply one of many possible methods to select the network weights. Ideally, any optimization method minimizing the same error function would produce the same weights so the resulting properties are not attributable to back-propagation alone. Likewise, the neural networks field contains more than just layered perceptrons. Although back-propagation and layered networks are adequate for many applications, there are good reasons to explore alternatives. Back-propagation, for example, often requires very long training times so much research has been devoted to finding faster methods. Similarly, there are applications where it is useful to build more structure into the network rather than using a simple fully connected layered structure. In a sense, back-propagation in layered feedforward networks could be viewed as a local minimum and it is hoped that further research will discover better methods. In any case, biological networks are certainly not simple layered feedforward structures and it is very unlikely that they adapt by back-propagation so we may have more to learn.

5.7 Training Time

Although back-propagation has been used successfully on a wide range of problems, one of the common complaints is that it is slow. Even simple problems may take hundreds of iterations to converge and harder problems may take many thousands of iterations. Training times of days or even weeks are not unusual for large practical applications.

Much work, therefore, has been done in search of faster methods. Effects of the learning rate and momentum are considered in chapter 6. Methods of weight initialization are considered in chapter 7. Chapter 9 summarizes a number of variations of the back-propagation algorithm intended to reduce training times and chapter 10 reviews some classical optimization methods that have better theoretical convergence properties.

The size of the training set obviously affects training times because each pass through the data takes twice as long when the data set is twice as big. Additional patterns that contain no new information may simply make learning slower. This depends on the training method and cost function, among other things. In batch-mode, each pass is slower but with a SSE cost function the weight changes will be twice as large so learning may converge in half the number of epochs if it remains stable. In on-line mode, each pass is slower but twice as many weight updates are done in each pass so the overall time may not change. Less obviously, the additional patterns may supply new information that restrict possible solutions and make the problem harder so that more iterations are required or they may supply missing information that makes the problem easier.

5.7.1 Scaling of Training Time

Hinton [169, section 6.10] argues that a network with W weights typically requires $O(W^3)$ training time on a serial machine. $O(W)$ cycles are required for each forward and backward propagation of a single pattern, $O(W)$ training patterns are typically needed to achieve good generalization, and (perhaps) $O(W)$ weight updates are required for each pattern. Implementation on parallel hardware would reduce this only by a factor of W resulting in $O(W^2)$ training time.

Tesauro and Janssens [367] observed that training time for the parity problem increases approximately as 4^d where d is the number of inputs (the predicate order). Although some problems are easy, Judd [202, 203] has shown that the general problem is NP-complete. That is, the training time scales exponentially with the problem size in the worst case. Networks of hard-limiting linear threshold elements are considered, but the results seem applicable to networks of sigmoidal units.

This problem is not unique to back-propagation, of course. Optimization methods cannot be held responsible for the rapid growth of the search space with the size of the problem. It does suggest, though, that a simple algorithm like back-propagation alone will not be adequate to find solutions for hard problems. Indeed, when solutions to interesting problems are found, the critical factor is often external information supplied by the network designer in selecting a network architecture, collecting and editing *relevant* data, choosing input-output representations, selecting an error function, and so forth.

Some possible causes of slow training include:

- Overly restrictive convergence criteria. If the trained network will be used for classification purposes it usually isn't necessary to train every output to within 10^{-6} of the target value.
- "Paralysis" due to sigmoid saturation.

- Flat regions in the error surface where the gradient is small.
- Ill-conditioning of the Hessian matrix (the matrix of second derivatives of the error with respect to the weights).
- A poor choice of parameters such as learning rate and momentum.
- The simple-mindedness of gradient descent: more sophisticated algorithms may use available information more efficiently.
- The global nature of sigmoid functions. A change in one weight may alter the network response over the entire input space. This changes the derivatives fed back to every other weight and produces further weight changes whose effects reverberate throughout the network. It takes time for these interactions to settle.
- The size and distribution of the training set.
- Poor representations, irrelevant inputs.
- Delta attenuation in deep networks.
- Poor network architectures. The minimal-size network just adequate to represent the data may require a very specific set of weights that may be very hard to find. Larger networks may have more ways to fit the data and so may be easier to train with less chance of convergence to poor local minima.

Many of these factors are related of course. Sigmoid saturation may cause flat spots in the error surface, for example, which are reflected in a poorly conditioned Hessian, which, in turn, makes it difficult to choose a good learning rate. Some of these factors are due to weaknesses of back-propagation and might be avoided by algorithmic improvements while others are more fundamental.

One way to reduce training times is to increase the efficiency of the optimization procedure. Chapter 6 discusses the effects of the learning rate and momentum parameters and gives hints for selecting reasonable values. Chapter 9 summarizes variations of the back-propagation algorithm intended to improve training times; many are techniques for adaptively controlling the learning rate. Chapter 10 reviews more sophisticated optimization methods that are reputed to have better convergence properties than gradient descent.

Another way to improve training times is to give the network a headstart on a good solution. Chapter 7 discusses some network initialization techniques based on this idea.

Yet another way to reduce training times is to identify and fix the problems that cause slow convergence. The structure of the $E(\mathbf{w})$ landscape has a fundamental effect on training time; some common properties are discussed in chapter 8. Many of these are reflected in numerical properties of the Hessian matrix discussed in section 8.6. Delta-attenuation is described in section 6.1.8 in conjunction with learning-rate adjustment methods.

Finally, another way to speed things up is to change the problem. Part of the reason for slow learning is that the algorithm is limited to adjusting existing weights. If the network structure is poorly matched to the problem, this may be very difficult. Algorithms that are free to change the structure during learning are often able to learn much faster. Chapters 12 and 13 discuss constructive algorithms, which "grow" networks to fit the problem, and pruning algorithms, which reduce large networks to fit the problem.

Paralysis In some cases, long learning times can be attributed to paralysis due to sigmoid saturation. That is, the sigmoid and related functions have nearly flat tails where the derivative is approximately zero for large inputs (positive or negative). Because δ_i is proportional to the slope f_i', this leads to small derivatives for weights feeding into the node and so on backward through the network. If many nodes are saturated, then weight derivatives may become very small and learning will be slow. In digital simulations, deltas may become so small that they are quantized to zero and learning stops (double precision arithmetic is sometimes recommended for this reason).

Large external inputs or large weights are a typical cause of saturation. Normalization of the inputs to a reasonable range and initialization with small random weights are standard remedies. The weight initialization range required to avoid saturation depends on the number of inputs to a node and their correlations so different ranges may be needed in different cases. Chapter 7 discusses some weight initialization techniques based on this idea. Gain-scaling (section 8.7) has also been suggested as a way to avoid and correct for saturation.

Regardless of how the network is initialized, the weights change during learning so saturation may become a problem at a later stage. Because paralysis can have such a strong affect on learning time, it is helpful to detect and correct it before it becomes serious. In software simulations, it is relatively easy to check for paralysis after each pattern presentation [382]. If a significant fraction (e.g., 1%) of nodes are near saturation (e.g., have absolute magnitude greater than 0.9 assuming tanh nodes), then steps can be taken to fix the problem (e.g., reduce the learning rate, reduce the sigmoid gain, or scale the weights). The computational cost of the test is insignificant so it can be done often to allow detection of imminent paralysis before it becomes a problem.