22 Learning by Training Neural Nets

In this chapter, you learn how *neuronlike elements, arranged in nets*, can be used to recognize instances of patterns, and you learn how neural nets can learn using the *back-propagation procedure.*

First, you review the most conspicuous properties of real neurons, and learn how those properties are modeled in neural nets.

Next, you learn how the *back-propagation procedure* alters the effect of one simulated neuron on another so as to improve overall performance.

By way of illustration, you see how a simulated neural net can be taught to recognize which people, among six, are acquaintances and which are siblings.

Once you have finished this chapter, you will know how simulated neural nets work, you will understand how the back-propagation procedure improves their performance, and you will understand why working with them remains an art.

SIMULATED NEURAL NETS

A vast literature explains what is known about how real neurons work from every conceivable perspective. Many books and papers explain neurons from the cellular perspective, diving deeply into membrane potentials and ion pumps. Others deal with neurotransmitters and the details of the activity at and near neuron synapses. Still others concentrate on how neurons are connected, tracing the paths taken by neurons as they process

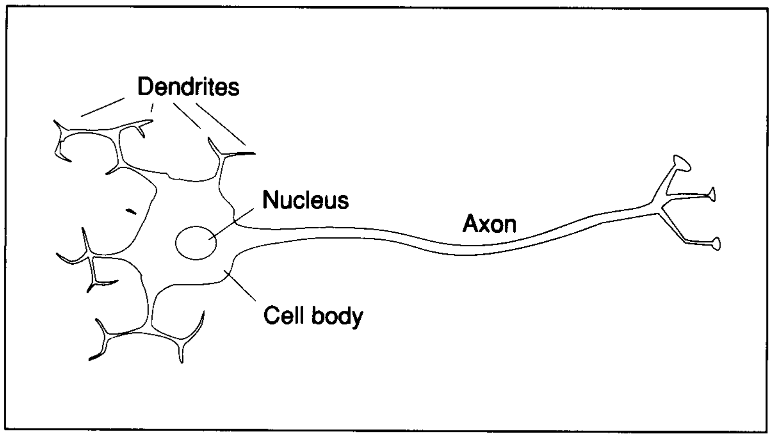


Figure 22.1 A neuron consists of a cell body, one axon, and many dendrites. Dendrites receive inputs from axons of other neurons via excitation or inhibition synapses. Real neurons may have many more dendrites.

information and carry messages from one place to another. And still others exploit ideas from contemporary engineering, drawing inspiration from subjects as diverse as transmission lines and frequency modulation.

Given this vast literature, the tendency of most people who try to understand and duplicate neural-net function has been to concentrate on only a few prominent characteristics of neurons.

In the rest of this section, you learn what those characteristics are and how they are mimicked in *feed-forward neural nets.* Although feed-forward nets are among the most popular, there are other kinds. For example, in Chapter 23, you learn about *perceptrons*, and in Chapter 24, you learn about *interpolation* and *approximation nets.*

Real Neurons Consist of Synapses, Dendrites, Axons, and Cell Bodies

Most neurons, like the one shown in figure 22.1, consist of a cell body plus one axon and many dendrites. The **axon** is a protuberance that delivers the neuron's output to connections with other neurons. **Dendrites** are protuberances that provide plenty of surface area, facilitating connection with the axons of other neurons. Dendrites often divide a great deal, forming extremely bushy dendritic trees. Axons divide to some extent, but far less than dendrites.

A neuron does nothing unless the collective influence of all its inputs reaches a threshold level. Whenever that threshold level is reached, the neuron produces a full-strength output in the form of a narrow pulse that proceeds from the cell body, down the axon, and into the axon's branches. Whenever this happens, the neuron is said to **fire**. Because a neuron either fires or does nothing, it is said to be an **all-or-none device**.

Axons influence dendrites over narrow gaps called **synapses**. Stimulation at some synapses encourages neurons to fire. Stimulation at others discourages neurons from firing. There is mounting evidence that learning

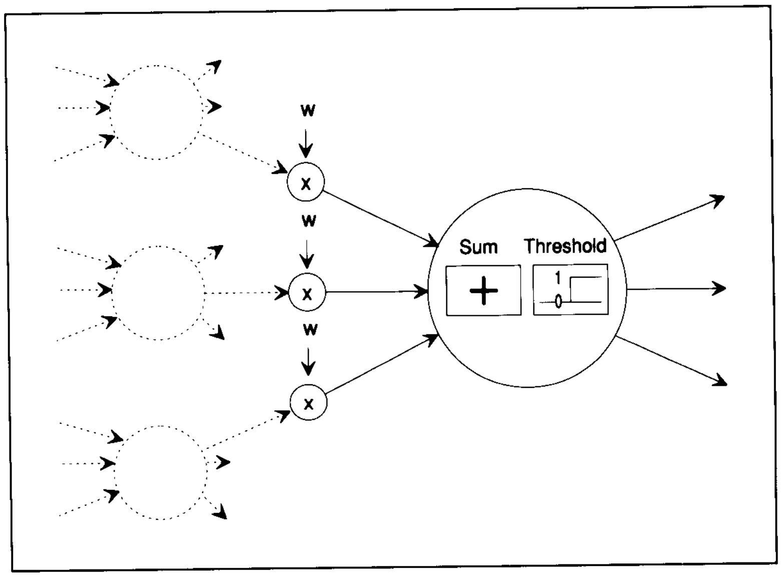


Figure 22.2 A simulated neuron. Inputs from other neurons are multiplied by weights, and then are added together. The sum is then compared with a threshold level. If the sum is above the threshold, the output is 1; otherwise, the output is 0.

takes place in the vicinity of synapses and has something to do with the degree to which synapses translate the pulse traveling down one neuron's axon into excitation or inhibition of the next neuron.

The number of neurons in the human brain is staggering. Current estimates suggest there may be on the order of 10^{11} neurons per person. If the number of neurons is staggering, the number of synapses must be toppling. In the cerebellum—that part of the brain that is crucial to motor coordination—a single neuron may receive inputs from as many as 10^5 synapses. Inasmuch as most of the neurons in the brain are in the cerebellum, each brain has on the order of 10^{16} synapses.

You can get a better feel for numbers like that if you know that there are about 1.5×10^6 characters in a book such as this one. Also, the United States Library of Congress holds on the order of 20×10^6 books. If each book were about the size of this one, that number of books would contain about 30×10^{12} characters. Accordingly, there are as many synapses in one brain as there would be characters in about 300 such libraries.

Simulated Neurons Consist of Multipliers, Adders, and Thresholds

Simulated neural nets typically consist of simulated neurons like the one shown in figure 22.2. The simulated neuron is viewed as a node connected to other nodes via links that correspond to axon–synapse–dendrite connections.

Each link is associated with a weight. Like a synapse, that weight determines the nature and strength of one node's influence on another. More specifically, one node's influence on another is the product of the influencing neuron's output value times the connecting link's weight. Thus, a large positive weight corresponds to strong excitation, and a small negative weight corresponds to weak inhibition.

Each node combines the separate influences received on its input links into an overall influence using an **activation function**. One simple activation function simply passes the sum of the input values through a **threshold function** to determine the node's output. The output of each node is either 0 or 1 depending on whether the sum of the inputs is below or above the **threshold value** used by the node's threshold function.

Now you can understand what is modeled in these simplified neurons: the weights model synaptic properties; the adder models the influence-combining capability of the dendrites; and comparison with a threshold models the all-or-none characteristic imposed by electrochemical mechanisms in the cell body.

Much of the character of real neurons is not modeled, however. A simulated neuron simply adds up a weighted sum of its inputs. Real neurons may process information via complicated dendritic mechanisms that call to mind electronic transmission lines and logical circuits. A simulated neuron remains on as long as the sum of its weighted inputs is above threshold. Real neurons may encode messages via complicated pulse arrangements that call to mind frequency modulation and multiplexing.

Accordingly, researchers argue hotly about whether simulated neural nets shed much light on real neural activity and whether these nets can perform anything like as wonderfully as the real thing. Many people consider real neural nets to be so different from contemporary simulations, that they always take care to use the qualifiers *simulated* or *real* whenever they use the phrase *neural net*. In the rest of this chapter, however, the word *simulated* is dropped to avoid tedious repetition.

Feed-Forward Nets Can Be Viewed as Arithmetic Constraint Nets

A handy way to do the computation required by a neural net is by arithmetic constraint propagation. Accordingly, you need a representation specification for neurons that builds on arithmetic constraint-propagation nets:

A **neural net** is a representation

That is an arithmetic constraint net

In which

▷ Operation frames denote arithmetic constraints modeling synapses and neurons.

▷ Demon procedures propagate stimuli through synapses and neurons.

And, of course, you need the demon procedures. One moves information across neurons; another moves information from one neuron to another.

When a value is written into a synapse's input slot,

- ▷ Write the product of the value and the synapse's weight into the synapse's output slot.

When a value is written into a synapse's output slot,

- ▷ Check the following neuron to see whether all its input synapses' outputs have values.
  - ▷ If they do, add the output values of the input synapses together, compare the sum with the neuron's threshold, and write the appropriate value into the neuron's output slot.
  - ▷ Otherwise, do nothing.

Feed-Forward Nets Can Recognize Regularity in Data

To get an idea of how neural nets can recognize regularity, consider the neural net shown in figure 22.3. The net is equipped with weights that enable it to recognize properties of pairs of people. Some of the pairs involve siblings, and others involve acquaintances.

Two input connections receive a value of 1 to identify the pair of people under consideration. All other input connections receive values of 0 because the corresponding people are not part of the pair under consideration. Assume that the people in the top group of three are siblings, as are people in the bottom group of three. Further assume that any pair of people who are not siblings are acquaintances.

The nodes just to the right of the input links—the ones labeled H1 and H2—are called **hidden nodes** because their outputs are not observable. The output nodes convey conclusions. The Acquaintances node, for example, receives a value of 1 when the input arrangement corresponds to two people who are acquaintances.

The net is not fully connected so as to simplify discussion: The Robert, Raquel, and Romeo inputs are not connected to the Siblings node, and the Joan, James, and Juliet nodes are not connected to the Acquaintances node.

Any of the first three inputs produces enough stimulation to fire H1, because all the connecting weights are 1.0 and because H1's threshold is 0.5. Similarly, any of the second three produces enough to fire H2. Thus, H1 and H2 act as logical Or gates. At least one of H1 and H2 has to fire because two inputs are always presumed to be on.

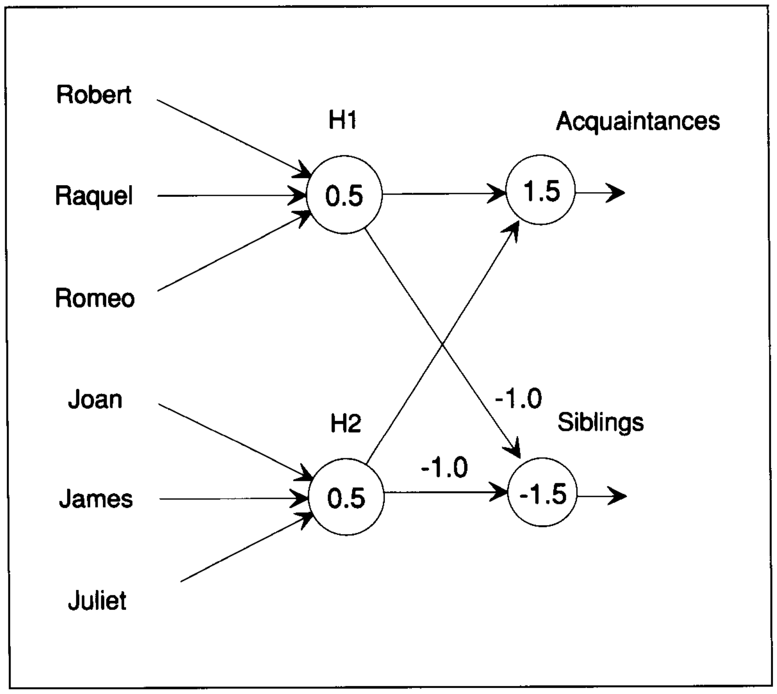


Figure 22.3 A neural net that recognizes siblings and acquaintances. All but the two indicated weights are 1.0. Thresholds are indicated inside the nodes.

If both H1 and H2 fire, then the weighted sum presented to the Acquaintance node is 2 because both the weights involved are 1.0 and because the Acquaintance node's threshold is 1.5. If only one of H1 and H2 fire, then the Acquaintance node does not fire. Thus, the Acquaintance node acts as a logical And gate: It fires only when the input pair are acquaintances.

On the other hand, if both H1 and H2 fire, then the weighted sum presented to the Siblings node is -2 because of the inhibiting -1 weights. The value -2 is below that node's threshold of -1.5, so the node does not fire. If only one of H1 and H2 fire, then the weighted sum is -1, which is above the Sibling node's threshold of -1.5, so it fires. Thus, the Sibling node fires if and only if the input pair causes exactly one hidden node to fire; this happens only when the input pair are siblings.

Note that each link and node in this example has a clear role. Generally, however, recognition capability is distributed diffusely over many more nodes and weights. Accordingly, the role of particular links and hidden nodes becomes obscure.

HILL CLIMBING AND BACK PROPAGATION

There are surprisingly simple procedures that enable neural-net weights to be learned automatically from training samples. In this section, you learn that hill climbing is one of those simple procedures.

The Back-Propagation Procedure Does Hill Climbing by Gradient Ascent

In the context of neural-net learning, each hill-climbing step amounts to small changes to the weights. The quality measurement is a measurement of how well the net deals with sample inputs for which the appropriate outputs are known.

The hill-climbing procedure explained in Chapter 4 requires you to try each possible step so that you can choose a step that does the most good. If you were to carry that hill-climbing idea over straightforwardly, you would try changing each weight, one at a time, keeping all other weights constant. Then, you would change only the weight that does the most good.

Fortunately, you can do much better whenever the hill you are climbing is a sufficiently smooth function of the weights. In fact, you can move in the direction of most rapid performance improvement by varying all the weights simultaneously in proportion to how much good is done by individual changes. When you use this strategy, you are said to move in the direction of the gradient in weight space and you are said to be doing **gradient ascent**.

The **back-propagation procedure** is a relatively efficient way to compute how much performance improves with individual weight changes. The procedure is called the back-propagation procedure because, as you soon see, it computes changes to the weights in the final layer first, reuses much of the same computation to compute changes to the weights in the penultimate layer, and ultimately goes *back* to the initial layer.

In the rest of this section, you learn about back propagation from two perspectives. The first, heuristic perspective is intended to make the back-propagation procedure seem reasonable; the second, mathematical perspective is intended to validate the heuristic explanation.

First, however, you need to learn about two neural-net modifications required in preparation for back propagation and gradient ascent.

Nonzero Thresholds Can Be Eliminated

You might think that, to learn, you would need separate procedures for adjusting weights and for adjusting thresholds. Fortunately, however, there is a trick that enables you to treat thresholds as though they were weights.

More specifically, a nonzero-threshold neuron is computationally equivalent to a zero-threshold neuron with an extra link connected to an input that is always held at -1.0. As shown in figure 22.4, the nonzero threshold value becomes the connecting weight's value. These threshold-equivalent weights can be changed in the course of learning just like the other weights, thus simplifying learning.

Gradient Ascent Requires a Smooth Threshold Function

Actually, the stair-step threshold function is unsuited for gradient ascent because gradient ascent requires performance to be a smooth function of

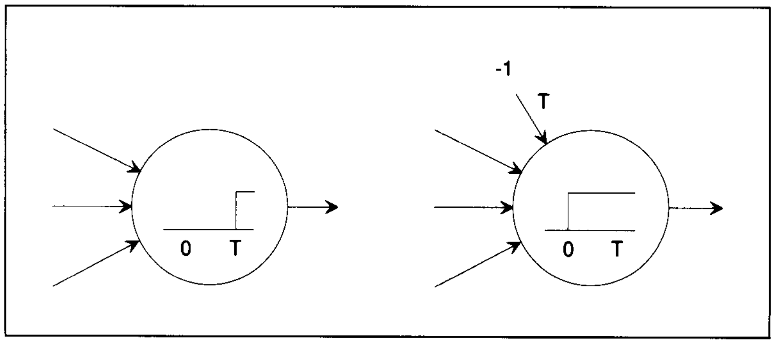


Figure 22.4 Thresholds are equivalent to links, with weight values equal to the threshold values, connected to inputs held at −1.0.

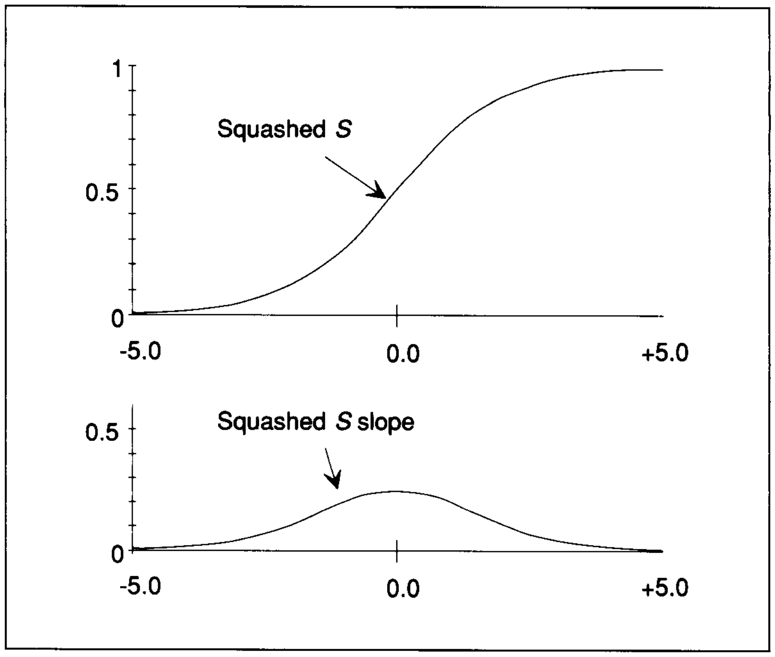


Figure 22.5 The Squashed *S* function and its slope. The slope of the Squashed *S* function approaches 0 when the sum of the inputs is either very negative or very positive; the slope reaches its maximum, 0.25, when the input is 0. Because the slope of the Squashed *S* function is given by a particularly simple formula, the Squashed *S* is a popular threshold function.

the weights. The all-or-none character of the stair-step produces flat plains and abrupt cliffs in weight space. Thus, the small steps inherent in gradient ascent do nothing almost everywhere, thus defeating the whole procedure.

Accordingly, a squashed S threshold function, shown in figure 22.5, replaces the stair-step function. The stair-step threshold function is somewhat more faithful to real neuron action, but the squashed S function provides nearly the same effect with the added mathematical advantage of smoothness that is essential for gradient ascent.

To make these modifications to the way neuron outputs are computed, you need only to replace one when-written procedure with another:

When a value is written into a synapse's output slot,

- ▷ Check the following neuron to see whether all its input synapses' outputs have values.
  - ▷ If they do, add the output values of the input synapses together, pass the sum through the squashed S function, determine whether the result is greater than 0, and write the appropriate value into the neuron's output slot.
  - ▷ Otherwise, do nothing.

Back Propagation Can Be Understood Heuristically

In this subsection, the back-propagation procedure is explained heuristically, with a minimum of mathematical equipment. If you prefer a briefer, more mathematical approach, skip ahead to the next subsection.

The overall idea behind back propagation is to make a large change to a particular weight, w, if the change leads to a large reduction in the errors observed at the output nodes. For each sample input combination, you consider each output's desired value, d, its actual value, o, and the influence of a particular weight, w, on the error, $d - o$. A big change to w makes sense if that change can reduce a large output error and if the size of that reduction is substantial. On the other hand, if a change to w does not reduce any large output error substantially, little should be done to that weight.

Note that most of the computation needed to compute the change to any particular weight is also needed to compute the changes to weights that are closer to the output nodes. Consider the net shown in figure 22.6. Once you see how to compute a change for a typical weight, $w_{i \to j}$, between a node in layer i and a node in layer j, you see that the required computations involve computations needed for the weights, $w_{j \to k}$, between nodes in layer j and nodes in layer k.

First note that a change in the input to node j results in a change in the output at node j that depends on the slope of the threshold function. Where the slope is steepest, a change in the input has the maximum effect on the output. Accordingly, you arrange for the change in $w_{i \to j}$ to depend on the slope of the threshold function at node j on the ground that change should be liberal only where it can do a lot of good.

The slope of the squashed S function is given by a particularly simple formula, $o(1-o)$. Thus, the use of the squashed S function as the threshold function leads to the following conclusion about changes to $w_{i \to j}$:

- Let the change in $w_{i \to j}$ be proportional to $o_j(1 - o_j)$.

Next, the change in the input to node j, given a change in the weight, $w_{i \to j}$, depends on the output of node i. Again, on the ground that change

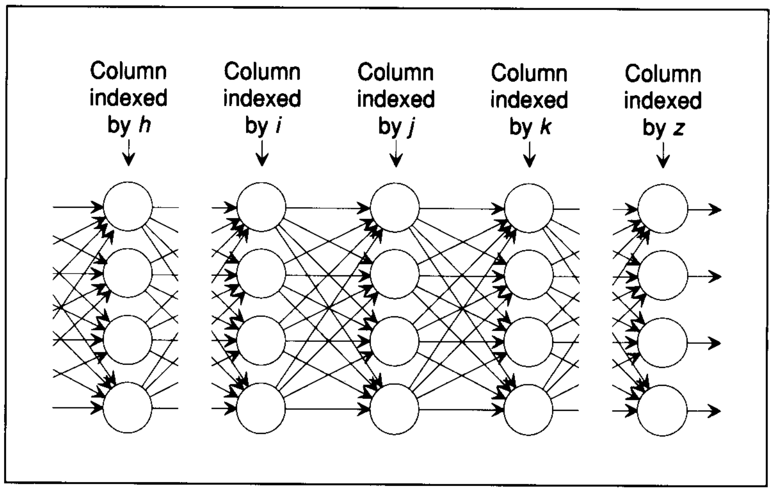


Figure 22.6 A trainable neural net. Each link has a weight that can be changed so as to improve the net's ability to produce the correct outputs for input combinations in the training set. Threshold-replacing links are not shown.

should be liberal only where it can do a lot of good, you arrange for $w_{i \to j}$ to change substantially only if the output of node i is high:

- Let the change in $w_{i \to j}$ be proportional to o_i, the output at node i.

Putting these considerations together, it seems that the change to a weight, $w_{i \to j}$, should be proportional to o_i, to $o_j(1 - o_j)$, and to a factor that captures how beneficial it is to change the output of node j. To make it easier to write things down, let us agree that the Greek letter β stands for the benefit obtained by changing the output value of a node. Thus, the change to $w_{i \to j}$ should be proportional to $o_i \times o_j(1 - o_j) \times \beta_j$.

Just how beneficial is it to change the output of node j? Imagine first that node j is connected to just one node in the next layer—namely, node k. Then, the reasoning just completed can be reapplied with a slight modification:

- Because change should be liberal only where it can do substantial good, the change to o_j should be proportional to $o_k(1 - o_k)$, the slope of the threshold function at node k.
- For the same reason, the change to o_j should be proportional to $w_{j \to k}$, the weight on the link connecting node j to node k.

Of course, node j is connected to many nodes in the next layer. The overall benefit obtained by changing o_j must be the sum of the individual effects, each of which includes a weight, $w_{j \to k}$, a slope $o_k(1 - o_k)$, and the factor β_k that indicates how beneficial it is to change o_k, the output of node k. Thus, the benefit that you obtain by changing the output of node j is summarized as follows:

$$\beta_j = \sum_k w_{j \to k} o_k (1 - o_k) \beta_k.$$

At this point, recall that the change to $w_{i \to j}$ is proportional to β_j. Evidently, the weight change in any layer of weights depends on a benefit calculation, β_j, that depends, in turn, on benefit calculations, β_k, needed to deal with weights closer to the output nodes.

To finish the analysis, you need to answer only one remaining question concerning the benefit you obtain by changing the value of an output node. This value depends, of course, on how wrong the output node's value happens to be. If the difference between d_z, the desired output at node z, and o_z, the actual output at that same node, is small, then the change in the output at node z should be relatively small. On the other hand, if the difference is large, the change in the output at node z should be large. Accordingly, the appropriate change to o_z should be in proportion to the difference, $d_z - o_z$. Recasting this conclusion in the framework of benefit, you have the following:

$$\beta_z = d_z - o_z.$$

Finally, weight changes should depend on a rate parameter, r, that should be as large as possible to encourage rapid learning, but not so large as to cause changes to the output values that considerably overshoot the desired values:

- Let the change in $w_{i \to j}$ be proportional to a rate parameter, r, determined experimentally.

Combining all the equations, you have the following **back-propagation formulas**:

$$\begin{aligned}\Delta w_{i \to j} &= r o_i o_j (1 - o_j) \beta_j, \\ \beta_j &= \sum_k w_{j \to k} o_k (1 - o_k) \beta_k \text{ for nodes in hidden layers,} \\ \beta_z &= d_z - o_z \text{ for nodes in the output layer.}\end{aligned}$$

Once you have worked out the appropriate change in the weights for one input combination, you face an important choice. Some neural-net enthusiasts make changes after considering each sample input. Others add up the changes suggested by individual sample inputs and make actual changes only after all the sample inputs are considered. In the example experiments described later in this chapter, changes are made only after all the sample inputs are considered, because this is the only way that is consistent with the mathematics of gradient ascent described in the following subsection.

Back-Propagation Follows from Gradient Descent and the Chain Rule

The previous subsection provides an heuristic argument leading to the back-propagation formulas. This subsection provides a mathematical argument leading to the same formulas. This mathematical argument rests on two ideas drawn from calculus:

- Suppose that y is a smooth function of several variables, x_i. Further suppose that you want to know how to make incremental changes to the initial values of each x_i so as to increase the value of y as fast as possible. Then, the change to each initial x_i value should be in proportion to the partial derivative of y with respect to that particular x_i. In other words,

$$\Delta x_i \propto \frac{\partial y}{\partial x_i}.$$

When you make such a change, you are doing **gradient ascent**.

- Suppose that y is a function of several intermediate variables, x_i, and that each x_i is a function of one variable, z. Further suppose that you want to know the derivative of y with respect to z. You obtain that derivative by adding up the results that you obtained by multiplying each partial derivative of y with respect to the x_i by the derivative of x_i with respect to z:

$$\frac{dy}{dz} = \sum_i \frac{\partial y}{\partial x_i}\frac{dx_i}{dz} = \sum_i \frac{dx_i}{dz}\frac{\partial y}{\partial x_i}.$$

When you compute such a derivative, you are using the **chain rule**.

Now recall that you have a set of weights that you want to improve, and you have a sample set of inputs along with each input's desired output. You need a way to measure how well your weights are performing, and you need a way to improve that measured performance.

The standard way of measuring performance is to pick a particular sample input and then sum up the squared error at each of the outputs. Once that is done for each sample input, you sum over all sample inputs and add a minus sign for an overall measurement of performance that peaks at 0:

$$P = -\sum_s \big(\sum_z (d_{sz} - o_{sz})^2\big),$$

where

P is the measured performance,

s is an index that ranges over all sample inputs,

z is an index that ranges over all output nodes,

d_{sz} is the desired output for sample input s at the zth node,

o_{sz} is the actual output for sample input s at the zth node.

Note that the reason that the sum of the squared errors is so popular is that it is the choice that most often leads to pretty and tractable mathematics. Otherwise, something else, such as adding up the absolute errors, would do as well.

Of course, the performance measure, P, is a function of the weights. Thus, you can deploy the idea of gradient ascent if you can calculate the partial derivative of performance with respect to each weight. With these partial derivatives in hand, you can climb the performance hill most rapidly by altering all weights in proportion to the corresponding partial derivative.

First, however, note that performance is given as a sum over all sample inputs. Accordingly, you can compute the partial derivative of performance with respect to a particular weight by adding up the partial derivative of performance for each sample input considered separately. Thus, you can reduce notational clutter by dropping the s subscript, thus focusing on the sample inputs one at a time, with the understanding that each weight will be adjusted by summing the adjustments derived from each sample input. Consider, then, the partial derivative

$$\frac{\partial P}{\partial w_{i\to j}},$$

where the weight, $w_{i\to j}$ is a weight connecting the ith layer of nodes to the jth layer of nodes.

Now your goal is to find an efficient way to compute the partial derivative of P with respect to $w_{i\to j}$. You reach that goal by expressing the partial derivative mostly in terms of computations that have to be done anyway to deal with weights closer to the output layer of nodes.

The effect of $w_{i\to j}$ on performance, P, is through the intermediate variable, o_j, the output of the jth node. Accordingly, you use the chain rule to express the derivative of P with respect to $w_{i\to j}$:

$$\frac{\partial P}{\partial w_{i\to j}} = \frac{\partial P}{\partial o_j}\frac{\partial o_j}{\partial w_{i\to j}} = \frac{\partial o_j}{\partial w_{i\to j}}\frac{\partial P}{\partial o_j}.$$

Now consider $\partial o_j/\partial w_{i\to j}$. You know that you determine o_j by adding up all the inputs to node j and passing the result through a threshold function. Hence, $o_j = f\left(\sum_i o_i w_{i\to j}\right)$, where f is the threshold function. Treating the sum as an intermediate variable, $\sigma_j = \sum_i o_i w_{i\to j}$, you can apply the chain rule again:

$$\frac{\partial o_j}{\partial w_{i\to j}} = \frac{df(\sigma_j)}{d\sigma_j}\frac{\partial \sigma_j}{\partial w_{i\to j}} = \frac{df(\sigma_j)}{d\sigma_j}o_i = o_i\frac{df(\sigma_j)}{d\sigma_j}.$$

Substituting this result back into the equation for $\partial P/\partial w_{i\to j}$ yields the following key equation:

$$\frac{\partial P}{\partial w_{i\to j}} = o_i\frac{df(\sigma_j)}{d\sigma_j}\frac{\partial P}{\partial o_j}.$$

Note that the partial derivative, $\partial P/\partial o_j$ can be expressed in terms of the partial derivatives, $\partial P/\partial o_k$, in the next layer to the right. Because the effect of o_j on P is through the outputs of the nodes in the next layer, the o_k, you can apply the chain rule to calculate $\partial P/\partial o_j$:

$$\frac{\partial P}{\partial o_j} = \sum_k \frac{\partial P}{\partial o_k}\frac{\partial o_k}{\partial o_j} = \sum_k \frac{\partial o_k}{\partial o_j}\frac{\partial P}{\partial o_k}.$$

But you know that you determine o_k by adding up all the inputs to node k and passing the result through a threshold function. Hence, $o_k = f\left(\sum_j o_j w_{j\to k}\right)$ where f is the threshold function. Treating the sum as an intermediate variable, σ_k, and applying the chain rule again, you have the following:

$$\frac{\partial o_k}{\partial o_j} = \frac{df(\sigma_k)}{d\sigma_k}\frac{\partial \sigma_k}{\partial o_j} = \frac{df(\sigma_k)}{d\sigma_k} w_{j\to k} = w_{j\to k}\frac{df(\sigma_k)}{d\sigma_k}$$

Substituting this result back into the equation for $\partial P/\partial o_j$ yields the following, additional key equation:

$$\frac{\partial P}{\partial o_j} = \sum_k w_{j\to k}\frac{df(\sigma_k)}{d\sigma_k}\frac{\partial P}{\partial o_k}.$$

Thus, in summary, the two key equations have two important consequences: first, the partial derivative of performance with respect to a weight depends on the partial derivative of performance with respect to the following output; and second, the partial derivative of performance with respect to one output depends on the partial derivatives of performance with respect to the outputs in the next layer. From these results, you conclude that the partial derivative of P with respect to any weight in the ith layer must be given in terms of computations already required one layer to the right in the jth layer.

To anchor the computation, however, you still have to determine the partial derivative of performance with respect to each output in the final layer. This computation, however, is easy:

$$\begin{aligned}\frac{\partial P}{\partial o_z} &= \frac{\partial}{\partial o_z} - (d_z - o_z)^2\\ &= 2(d_z - o_z).\end{aligned}$$

It remains to deal with the derivative of the threshold function, f, with respect to its argument, σ, which corresponds to the sum of the inputs seen by a node. Naturally, you choose f such that it is both intuitively satisfying and mathematically tractable:

$$\begin{aligned}f(\sigma) &= \frac{1}{1+e^{-\sigma}}\\ \frac{df(\sigma)}{d\sigma} &= \frac{d}{d\sigma}\left[\frac{1}{(1+e^{-\sigma})}\right]\\ &= (1+e^{-\sigma})^{-2}e^{-\sigma}\\ &= f(\sigma)(1-f(\sigma))\\ &= o(1-o).\end{aligned}$$

Unusually, the derivative is expressed in terms of each node's output, $o = f(\sigma)$, rather than the sum of the inputs, σ. This way of expressing the derivative is exactly what you want, however, because your overall goal is to produce equations that express values in terms of other values to their right.

Finally, weight changes should depend on a rate parameter, r, that should be as large as possible to encourage rapid learning, but not so large as to cause changes to the output values that considerably overshoot the desired values.

Now, at last, you are ready to look at the **back-propagation formulas**. So that they look the same as the back-propagation formulas developed in the previous, heuristic subsection, $\partial P/\partial o$ is written as β, and a factor of 2 is absorbed into the rate parameter, r.

$$\begin{aligned}\Delta w_{i\to j} &= r o_i o_j(1 - o_j)\beta_j,\\ \beta_j &= \sum_k w_{j\to k} o_k(1 - o_k)\beta_k \text{ for nodes in hidden layers,}\\ \beta_z &= d_z - o_z \text{ for nodes in the output layer.}\end{aligned}$$

Once you compute changes for each sample input combination, the chain rule dictates that you must add up the weight changes suggested by those individual sample input combinations. Then you can make actual changes to the weights.

The Back-Propagation Procedure Is Straightforward

The back-propagation equations are incorporated into the following back-propagation procedure:

```
To do back propagation to train a neural net,
▷ Pick a rate parameter, $r$.
▷ Until performance is satisfactory,
  ▷ For each sample input,
    ▷ Compute the resulting output.
    ▷ Compute $\beta$ for nodes in the output layer using
          $\beta_z = d_z - o_z$.
    ▷ Compute $\beta$ for all other nodes using
          $\beta_j = \sum_k w_{j\to k} o_k(1 - o_k)\beta_k$.
    ▷ Compute weight changes for all weights using
          $\Delta w_{i\to j} = r o_i o_j(1 - o_j)\beta_j$.
  ▷ Add up the weight changes for all sample inputs, and
    change the weights.
```

Because weight changes are proportional to output errors, the outputs will only approach the 1 and 0 values used as training targets; they will never reach those values. Accordingly, performance is usually deemed satisfactory when all outputs that are trained using 1 as the target value actually exhibit values that are greater than 0.9 and all that are trained using 0 as the target value actually exhibit values that are less than 0.1.

BACK-PROPAGATION CHARACTERISTICS

In this section, you learn that back-propagation performance depends critically on your detailed choices and on the nature of the problem to be solved.

Training May Require Thousands of Back Propagations

Changing weights by back propagation is efficient from a computational point of view because the maximum number of additions and multiplications required for the adjustment of any particular weight is on the order of the maximum number of links emanating from a node. Impracticably many steps may be required, however.

Consider, for example, the net shown in figure 22.7, which is similar to part of the net shown in figure 22.3. Assume that exactly two of the inputs presented to the net have values of 1, and that the rest have values of 0. The purpose of the net is to determine whether the two people corresponding to the on inputs are acquaintances. The two people are judged to be acquaintances if the output value is greater than 0.9; they are judged to be not acquaintances if the output value is less than 0.1; and the result is considered ambiguous otherwise.

The problem is to adjust the weights in the net, starting from some set of initial values, until all judgments are consistent with the knowledge that everyone knows everyone else, but that Robert, Raquel, and Romeo are siblings and therefore know one another too well to be considered acquaintances, as are Joan, James, and Juliet.

Table 1 expresses the same knowledge by listing the appropriate output in the column labeled A, for acquaintances, for all 15 possible input combinations. The table also has a column for identifying siblings, which is involved in subsequent training.

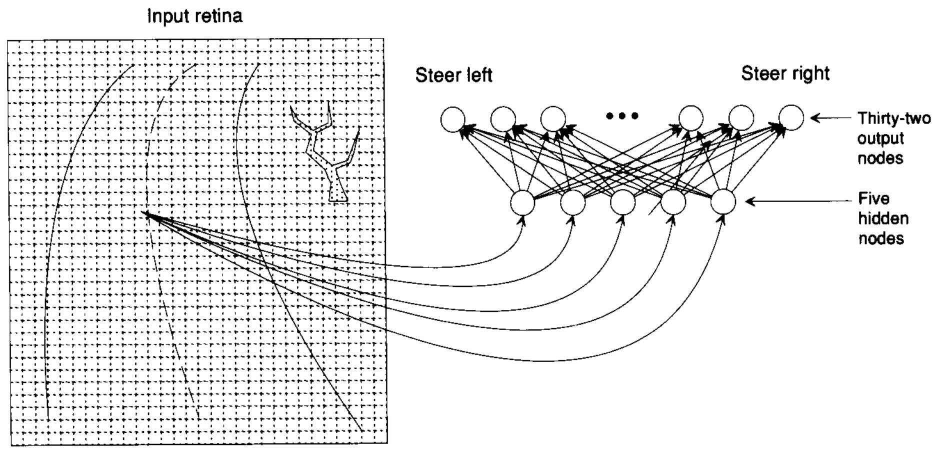
These sample inputs are just what you need to execute the back-propagation procedure. Suppose, for example, that the value of the rate parameter is 1.0. Further suppose that the back-propagation procedure is given the initial values for thresholds and weights shown in the first column of table 2. Note that the first initial value is 0.1, the second is 0.2, and the rest range up to 1.1 in 0.1 increments. These choices constitute a departure from the usual practice of using random numbers for initial values. The reason for the departure is that the use of a regular pattern of initial values makes it easier for you to see how training changes the weights. For the illustrations in this chapter, using random numbers for initial values produces results that are similar to the results for the regular pattern. In fact, just about any numbers will do, as long as they differ from one another.

When all sample inputs produce an appropriate output value, the thresholds and weights are as shown in the second column of table 2. These thresholds and weights are, of course, much different from the ones used earlier during the basic explanation of how neural nets work. The way the

ALVINN Learns to Drive

The ALVINN system learns to drive a van along roads viewed through a television camera. Once ALVINN has been trained on a particular road, it can drive at speeds in excess of 40 miles per hour.

ALVINN, an acronym for autonomous land vehicle in a neural net, contains one hidden layer of nodes, one output layer, and nearly 5000 trainable weights. Each of ALVINN's 960 inputs produce an image intensity recorded on a 30 by 32 photosensitive array. As shown in the following diagram, each of these 960 inputs is connected, via a trainable weight, to all of the five middle-level nodes. And finally, each of the five middle-level nodes is connected, via a trainable weight, to all of the 32 output nodes.



If ALVINN's leftmost output node exhibits the highest output level, ALVINN directs the van's steering mechanism to turn the van sharply left; if the rightmost output node exhibits the highest output level, ALVINN directs the van sharply right; when an intermediate node exhibits the highest output level, ALVINN directs the van in a proportionately intermediate direction.

To smooth out the steering, ALVINN calculates the actual steering direction as the average direction suggested not only by the node with the highest output level but also by that node's immediate neighbors, all contributing in proportion to their output level.

To learn, ALVINN monitors the choices of a human driver. As the human driver steers the van down the training road, periodic sampling of the inputs and the human-selected steering direction provide fodder for back propagation.

One special twist is required, however. Because the human driver does so well, few, if any, of the periodic samples cover situations in which the van is seriously misaligned with the road. Accordingly, monitoring a human driver is not sufficient to ensure that ALVINN can get back on track if the van drifts off track for some reason. Fortunately, however, ALVINN can enrich the set of human-supplied training samples by manufacturing synthetic views from those actually witnessed. Using straightforward geometrical formulas, ALVINN transforms a straight-ahead view of a road seen through the windshield of a well-steered van into a view of what the road would look like if the van were, say, 10° or so off course to the left, thus inviting a right turn that would bring the van back on course.

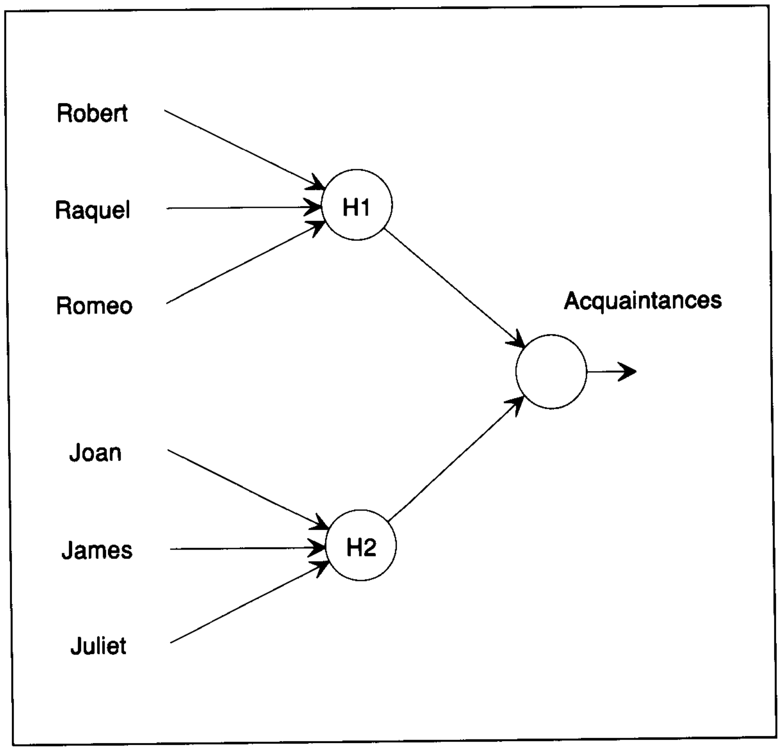


Figure 22.7 A learning problem involving acquaintances. The task is to learn that anyone in the top group of three is an acquaintance of anyone in the bottom group of three. Threshold-replacing links are not shown.

Table 1. Data for the neural-net learning experiments. The first six columns record the possible input combinations. The final two record the corresponding outputs. The column labeled A identifies those pairs of people who are acquaintances; the column labeled S identifies siblings. The first task involves only the acquaintance column; the second task involves both the acquaintance and sibling columns.

Robert	Raquel	Romeo	Joan	James	Juliet	A	S
1	1	0	0	0	0	0	1
1	0	1	0	0	0	0	1
1	0	0	1	0	0	1	0
1	0	0	0	1	0	1	0
1	0	0	0	0	1	1	0
0	1	1	0	0	0	0	1
0	1	0	1	0	0	1	0
0	1	0	0	1	0	1	0
0	1	0	0	0	1	1	0
0	0	1	1	0	0	1	0
0	0	1	0	1	0	1	0
0	0	1	0	0	1	1	0
0	0	0	1	1	0	0	1
0	0	0	1	0	1	0	1
0	0	0	0	1	1	0	1

Table 2. Weight changes observed in training a neural net. Eventually, pairs of people who are acquaintances are recognized. Initial values are changed through back propagation until all outputs are within 0.1 of the required 0.0 or 1.0 value.

Weight	Initial value	End of first task
t_{H1}	0.1	1.99
$w_{\text{Robert}\rightarrow\text{H1}}$	0.2	4.65
$w_{\text{Raquel}\rightarrow\text{H1}}$	0.3	4.65
$w_{\text{Romeo}\rightarrow\text{H1}}$	0.4	4.65
t_{H2}	0.5	2.28
$w_{\text{Joan}\rightarrow\text{H2}}$	0.6	5.28
$w_{\text{James}\rightarrow\text{H2}}$	0.7	5.28
$w_{\text{Juliet}\rightarrow\text{H2}}$	0.8	5.28
$t_{\text{Acquaintances}}$	0.9	9.07
$w_{\text{H1}\rightarrow\text{Acquaintances}}$	1.0	6.27
$w_{\text{H2}\rightarrow\text{Acquaintances}}$	1.1	6.12

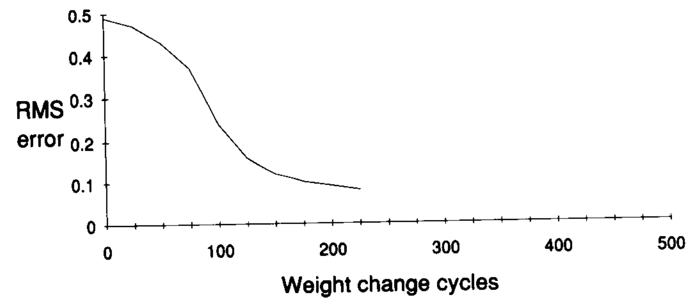


Figure 22.8 Results for a learning experiment. The square root of the average squared error seen at the output nodes is plotted versus the number of back propagations done during staged learning about acquaintances.

thresholds and weights work for acquaintances is the same, however. Any input in the first group of three pushes H1's output near 1; and any input in the second group pushes H2's output near 1; the Acquaintances node is near 1 only if both H1 and H2 are near 1.

This net training took more than a few steps, however. As shown in figure 22.8, performance becomes satisfactory only after about 225 weight changes. The weights are changed after each complete set of sample inputs is processed with the current weights. Because there are 15 sample inputs, the sample inputs are processed 225 × 15 = 3375 times.

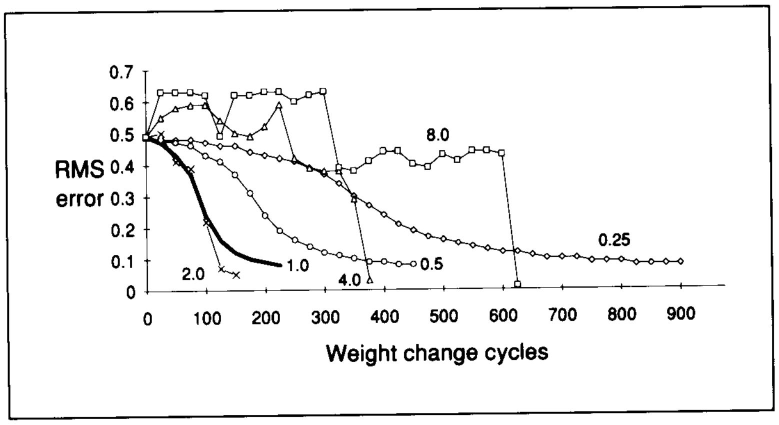


Figure 22.9 Learning behavior can depend considerably on the rate parameter. Six different rate parameters, from 0.25 to 8.0, with everything else the same, produced these six results.

Back Propagation Can Get Stuck or Become Unstable

Now it is time to consider what happens as the rate parameter varies. Specifically, the experiments in the next group repeat the previous experiment, but now the rate parameter varies from 0.25 to 8.0.

You have already seen, in figure 22.8, that the back-propagation procedure can produce a satisfactory solution after about 225 weight changes. For that result, the value of the rate parameter is 1.0. As shown in figure 22.9, decreasing the rate parameter to 0.25 produces a satisfactory solution too, but only after 900 weight changes—about four times as many as for 1.0, as you would expect. Similarly, an intermediate value of 0.5 produces a satisfactory solution after 425 weight changes—about twice as many as for 1.0.

Increasing the rate parameter to 2.0 again reduces the number of weight changes required to produce a satisfactory solution, but now performance is worse, rather than better, for a short time after getting started.

Increasing the rate parameter still further to 4.0 or 8.0 introduces serious instability—errors increase as well as decrease. The reason for the instability is that the steps are so large that the locally computed gradients are not valid.

Thus, a rate parameter of 1.0 produces reasonably rapid solution, but the steps are not so big as to introduce any apparent instability. Accordingly, a rate parameter of 1.0 is used in the rest of the experiments in this chapter. Note, however, that there is no right size for rate parameter in general; the right size depends on the problem you are solving.

Back Propagation Can Be Done in Stages

For further illumination, suppose you add another output node as shown in figure 22.10. The intent is that the output of the new node is to be 1 when the input combination indicates two siblings.

Further suppose that every pair of people who are not acquaintances are siblings, as reflected in the column labeled S, for siblings, in table 1.

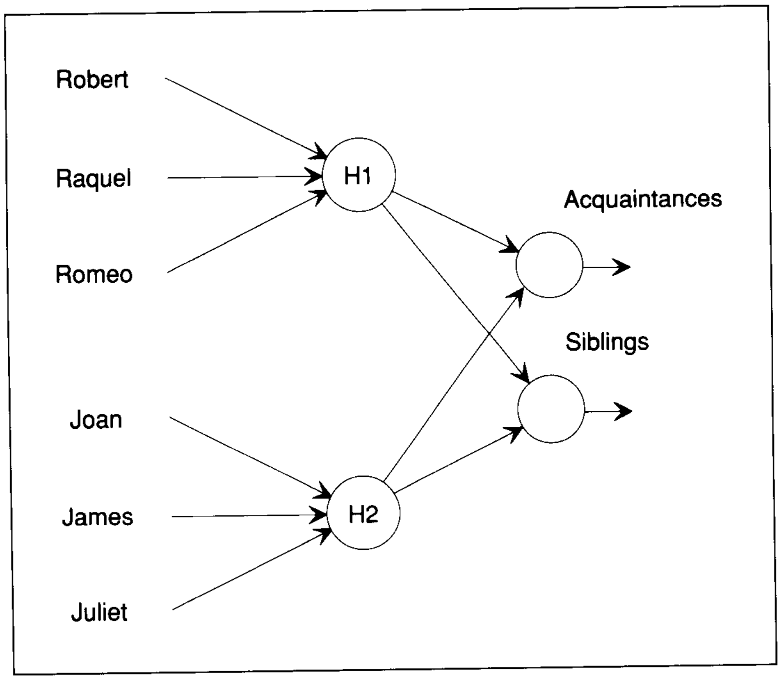


Figure 22.10 A learning problem involving acquaintances and siblings. Having learned that anyone in the top group of three is an acquaintance of anyone in the bottom group of three, the net is to learn that each group consists of siblings. Threshold-replacing links are not shown.

Now you can execute the back-propagation procedure again with the added siblings information. This time, however, you can start with the weights produced by the first experiment; you need new weights for only the new node's threshold and the new connecting links.

When all sample inputs produce an appropriate output value, the weights and thresholds are as shown in the end-of-second-task column of table 3.

As shown by the right portion of the line in figure 22.11, performance becomes satisfactory after about 175 additional weight changes are performed, given a value of 1.0 for the rate parameter. Again, it takes a large number of steps to produce the result, but not so many as required to adjust the weights to recognize acquaintances. This reduction occurs because many of the weights acquired to recognize acquaintances are appropriate for recognizing siblings as well.

Back Propagation Can Train a Net to Learn to Recognize Multiple Concepts Simultaneously

At this point, it is natural to ask what happens when you try to deal both output nodes from the very beginning. For this particular example, as shown in figure 22.12, about 425 weight changes produce a satisfactory set of weights, whereas a total of 400 weight changes were required with staged learning. In general, simultaneous learning may be either faster or slower

Table 3. Further weight changes observed in training a neural net. At first, only the acquaintance relation is learned. Then, for the second task, the sibling relation is learned. Dashes indicate that the corresponding weight is not yet present. Initial, random values are changed through back propagation until all outputs are within 0.1 of the required 0.0 or 1.0 value.

Weight	Initial value	End of 1st task	End of 2nd task
t_{H1}	0.1	1.99	2.71
$w_{\text{Robert}\rightarrow\text{H1}}$	0.2	4.65	6.02
$w_{\text{Raquel}\rightarrow\text{H1}}$	0.3	4.65	6.02
$w_{\text{Romeo}\rightarrow\text{H1}}$	0.4	4.65	6.02
t_{H2}	0.5	2.28	2.89
$w_{\text{Joan}\rightarrow\text{H2}}$	0.6	5.28	6.37
$w_{\text{James}\rightarrow\text{H2}}$	0.7	5.28	6.37
$w_{\text{Juliet}\rightarrow\text{H2}}$	0.8	5.28	6.37
$t_{\text{Acquaintances}}$	0.9	9.07	10.29
$w_{\text{H1}\rightarrow\text{Acquaintances}}$	1.0	6.27	7.04
$w_{\text{H2}\rightarrow\text{Acquaintances}}$	1.1	6.12	6.97
t_{Siblings}	1.2	–	-8.32
$w_{\text{H1}\rightarrow\text{Siblings}}$	1.3	–	-5.72
$w_{\text{H2}\rightarrow\text{Siblings}}$	1.4	–	-5.68

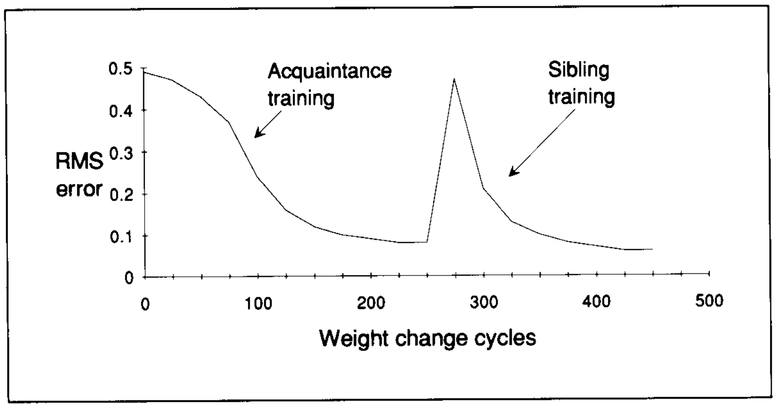


Figure 22.11 Results for staged learning. First, the net is taught about acquaintances; then, it is taught about siblings. The square root of the average squared error seen at the output nodes is plotted versus the number of back propagations done during staged learning about acquaintances.

than staged learning. Which method is faster depends on the problem and on the initial weights.

Trained Neural Nets Can Make Predictions

So far, you have learned that you can train a neural net to recognize acquaintances and siblings. In each experiment, however, the training set of

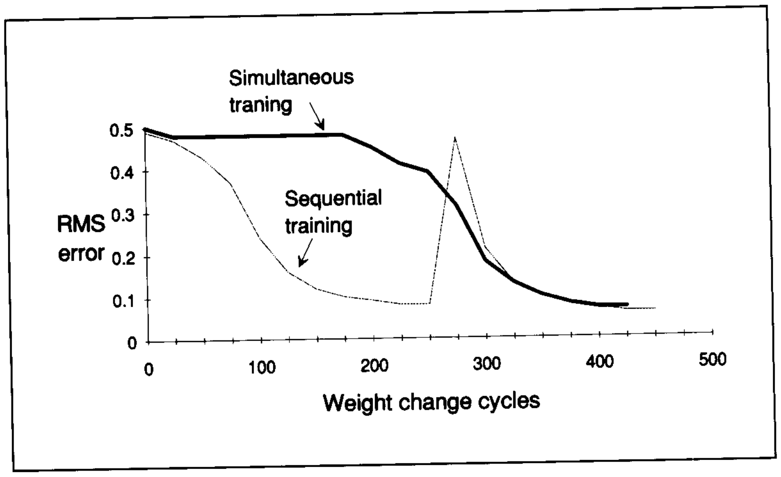


Figure 22.12 The dotted line shows the square root of the average squared error seen at the output nodes during staged learning. The solid line shows the square root of the average squared error seen at the output nodes during simultaneous learning. A little more work is required, in this experiment, if the learning is done in stages.

sample input–output combinations included every possible pair of people. Thus, you have yet to see a neural net use what it has learned from sample input–output combinations to predict the correct outputs for previously unseen inputs.

Accordingly, suppose you divide the data shown in table 1 into a **training set** and a **test set**. Let the training set consist of the original data with every fifth sample input–output combination removed. The test set consists of every fifth sample. Thus, you reserve 20 percent of the data for testing whether the neural net can generalize from the data in the training set.

The back-propagation procedure successfully trains the net to deal with all the sample input–output combinations in the training set after only 225 weight changes. That is fewer changes than were required when the net was trained on all the sample input–output combinations, because there are three fewer input–output combinations to be accommodated.

Pleasingly, the trained net also deals successfully with the input–output combinations in the test set, as demonstrated by the following table, in which the *d* subscript denotes desired value and the *o* subscript denotes observed value:

Robert	Raquel	Romeo	Joan	James	Juliet	A_d	A_o	S_d	S_o
1	0	0	0	0	1	1	0.92	0	0.06
0	0	1	1	0	0	1	0.92	0	0.06
0	0	0	0	1	1	0	0.09	1	0.91

Excess Weights Lead to Overfitting

Intuitively, you might think that, if one neural net does well, a neural net with more trainable weights would do even better. You must learn to

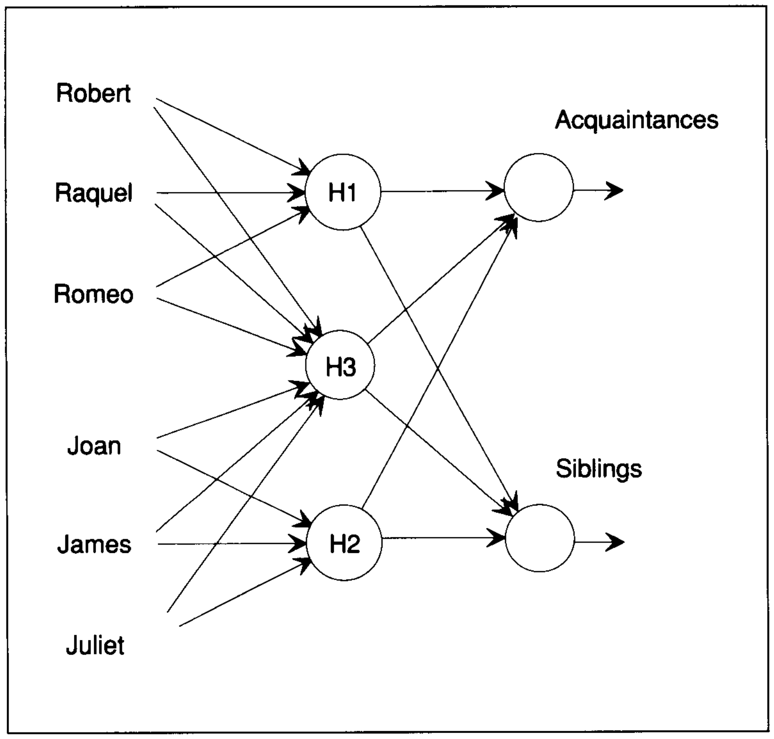


Figure 22.13 Another neural net for dealing with the acquaintances–siblings problem. This one has too many weights, and illustrates the overfitting problem.

suppress this intuition, because neural nets become erratic and unreliable if they have too many weights.

Consider, for example, the net shown in figure 22.13. Given that it is enriched with a new node and nine new weights—including the one for the new node's threshold—you might think that this net would handle the previous training problem with no difficulty, perhaps converging faster to an equally good solution. In any event, you would think that the additional new weights could do no harm, because the back-propagation procedure conceivably could drive them all toward 0.

Given the new weights shown in figure 22.13, the back-propagation procedure requires 300 weight changes to deal with all the sample input–output combinations. Thus, adding new weights does speed convergence. Unfortunately, however, the performance on the test set now exhibits errors, as demonstrated by the following table:

Robert	Raquel	Romeo	Joan	James	Juliet	A_d	A_o	S_d	S_o
1	0	0	0	0	1	1	0.99	0	0.00
0	0	1	1	0	0	1	0.06 (?)	0	0.94 (?)
0	0	0	0	1	1	0	0.97 (?)	1	0.01 (?)

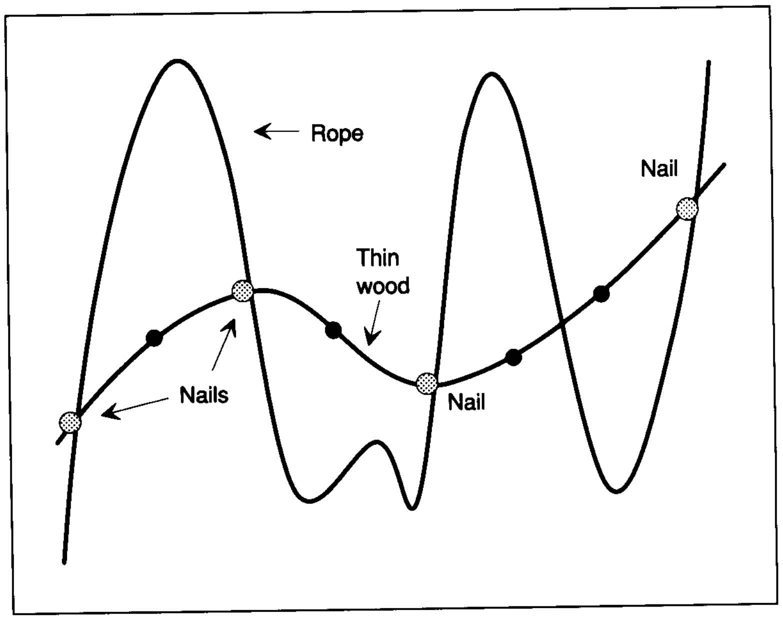


Figure 22.14 Overfitting is a consequence of too much flexibility. Here, a piece of wood provides a nice fit to the black dots when forced into conformance by a few nails. A steel rod cannot be forced into any sort of conformance, however; and a rope can meander all over the terrain.

The problem is that the additional weights provide too much flexibility, making it possible to deal with the training set too easily. To see why both too little and too much flexibility are bad, consider the points shown in figure 22.14. You could drive nails into a few points and use those nails to force a thin piece of wood into a shape that would fit smoothly through all the points. If you tried to use a heavy steel rod, however, you would find it too stiff to bend. If you tried to use a rope, you would find it too flexible, because you could fit the rope to all the nails yet have wild meandering in between.

The rope is analogous to a neural net with too many trainable weights: A neural net with too many weights can conform to the input–output samples in the training set in many ways, some of which correspond to wild meandering. A net that conforms to the data with wild meandering is said to exhibit **overfitting**.

To avoid overfitting, you can use one good heuristic: Be sure that the number of trainable weights influencing any particular output is smaller than the number of training samples. For the acquaintance–sibling net shown in figure 22.3, each output value is determined by 11 trainable weights and there are 12 input–output samples—a dangerously small margin, but training was successful nevertheless. For the acquaintance–sibling net shown in figure 22.13, each output value is determined by 19 trainable weights—an excess of more than 50 percent over the number of training samples. Overfitting is inevitable.

Neural-Net Training Is an Art

You now know that you face many choices after you decide to work on a problem by training a neural net using back propagation:

- How can you represent information in neural net terms? How can you use neural net inputs to express what you know? How can you use neural net outputs to determine what you want to know?
- How many neurons should you have in your neural net? How many inputs? How many outputs? How many weights? How many hidden layers?
- What rate parameter should you use in the back-propagation formula?
- Should you train your neural net in stages or simultaneously?

The wrong choices lead to poor performance. A small neural net may not learn what you want it to learn. A big net will learn slowly, may get stuck on local maxima, and may exhibit overfitting. A small rate parameter may waste time. A large rate parameter may promote instability or provide poor predictions.

Unfortunately, the proper choices depend on the nature of the samples. Mathematically, you can view the samples as representative glimpses of a hidden function, with one dimension for each input. If there are many inputs, the function's multidimensional character makes the function hard to think about and impossible to visualize.

Accordingly, the best guide to your choices is trial and error, buttressed, if possible, by reference to the choices that have worked well in similar problems. Thus, the successful deployment of neural-net technology requires time and experience. Neural-net experts are artists; they are not mere handbook users.

SUMMARY

- Real neurons consist of synapses, dendrites, axons, and cell bodies. Simulated neurons consist of multipliers, adders, and thresholds.
- One way to learn is to train a simulated neural net to recognize regularity in data.
- The back-propagation procedure is a procedure for training neural nets. Back propagation can be understood heuristically or by way of a mathematical analysis.
- To enable back propagation, you need to perform a simple trick that eliminates nonzero neuron thresholds. You also need to convert stair-step threshold functions into squashed S threshold functions.
- You can teach a neural net, via back propagation, to recognize several concepts. These concepts can be taught one at a time or all at once.
- You must choose a back-propagation rate parameter carefully. A rate parameter that is too small leads to slow training; a rate parameter that is too large leads to instability.

- An excess of trainable weights, relative to the number of training samples, can lead to overfitting and poor performance on test data.

BACKGROUND

There is a vast literature on the subject of neural nets. For an overview, see *Parallel Distributed Processing*, edited by James L. McClelland and David E. Rumelhart [1986], or *Neurocomputing: Foundations of Research*, edited by James A. Anderson and Edward Rosenfeld [1989].

In the literature on neural nets, the papers by Geoffrey E. Hinton [1989, 1990] and by J. J. Hopfield [1982] have been particularly influential.

The discussion of ALVINN is based on the work of Dean A. Pomerleau [1991]. NETtalk, a system that learns to speak, is another, often cited application of neural nets [Terrence J. Sejnowski and Charles R. Rosenberg 1989].