CHAPTER

26

Neural Networks (and more!)

Traditional DSP is based on *algorithms*, changing data from one form to another through step-by-step procedures. Most of these techniques also need *parameters* to operate. For example: recursive filters use recursion *coefficients*, feature detection can be implemented by correlation and *thresholds*, an image display depends on the *brightness* and *contrast* settings, etc. Algorithms describe what is to be done, while parameters provide a benchmark to judge the data. The proper selection of parameters is often more important than the algorithm itself. Neural networks take this idea to the extreme by using very simple algorithms, but many highly optimized parameters. This is a revolutionary departure from the traditional mainstays of science and engineering: mathematical logic and theorizing followed by experimentation. Neural networks replace these problem solving strategies with trial & error, pragmatic solutions, and a "this works better than that" methodology. This chapter presents a variety of issues regarding parameter selection in both neural networks and more traditional DSP algorithms.

Target Detection

Scientists and engineers often need to know if a particular object or condition is present. For instance, geophysicists explore the earth for oil, physicians examine patients for disease, astronomers search the universe for extra-terrestrial intelligence, etc. These problems usually involve comparing the acquired data against a threshold. If the threshold is exceeded, the **target** (the object or condition being sought) is deemed present.

For example, suppose you invent a device for detecting cancer in humans. The apparatus is waved over a patient, and a number between 0 and 30 pops up on the video screen. Low numbers correspond to healthy subjects, while high numbers indicate that cancerous tissue is present. You find that the device works quite well, but isn't perfect and occasionally makes an error. The question is: how do you use this system to the benefit of the patient being examined?

451

Figure 26-1 illustrates a systematic way of analyzing this situation. Suppose the device is tested on two groups: several hundred volunteers known to be healthy (nontarget), and several hundred volunteers known to have cancer (target). Figures (a) & (b) show these test results displayed as histograms. The healthy subjects generally produce a lower number than those that have cancer (good), but there is some overlap between the two distributions (bad).

As discussed in Chapter 2, the histogram can be used as an estimate of the **probability distribution function (pdf)**, as shown in (c). For instance, imagine that the device is used on a randomly chosen healthy subject. From (c), there is about an 8% chance that the test result will be 3, about a 1% chance that it will be 18, etc. (This example does not specify if the output is a real number, requiring a *pdf*, or an integer, requiring a *pmf*. Don't worry about it here; it isn't important).

Now, think about what happens when the device is used on a patient of unknown health. For example, if a person we have never seen before receives a value of 15, what can we conclude? Do they have cancer or not? We know that the probability of a healthy person generating a 15 is 2.1%. Likewise, there is a 0.7% chance that a person with cancer will produce a 15. If no other information is available, we would conclude that the subject is three times as likely not to have cancer, as to have cancer. That is, the test result of 15 implies a 25% probability that the subject is from the target group. This method can be generalized to form the curve in (d), the probability of the subject having cancer based only on the number produced by the device [mathematically, $pdf_t / (pdf_t + pdf_{nt})$].

If we stopped the analysis at this point, we would be making one of the most common (and serious) errors in target detection. Another source of information must usually be taken into account to make the curve in (d) meaningful. This is the relative number of targets versus nontargets in the population to be tested. For instance, we may find that only one in one-thousand people have the cancer we are trying to detect. To include this in the analysis, the amplitude of the nontarget pdf in (c) is adjusted so that the area under the curve is 0.999. Likewise, the amplitude of the target pdf is adjusted to make the area under the curve be 0.001. Figure (d) is then calculated as before to give the probability that a patient has cancer.

Neglecting this information is a serious error because it greatly affects how the test results are interpreted. In other words, the curve in figure (d) is drastically altered when the prevalence information is included. For instance, if the fraction of the population having cancer is 0.001, a test result of 15 corresponds to only a 0.025% probability that this patient has cancer. This is very different from the 25% probability found by relying on the output of the machine alone.

This method of converting the output value into a probability can be useful for understanding the problem, but it is not the main way that target detection is accomplished. Most applications require a yes/no decision on
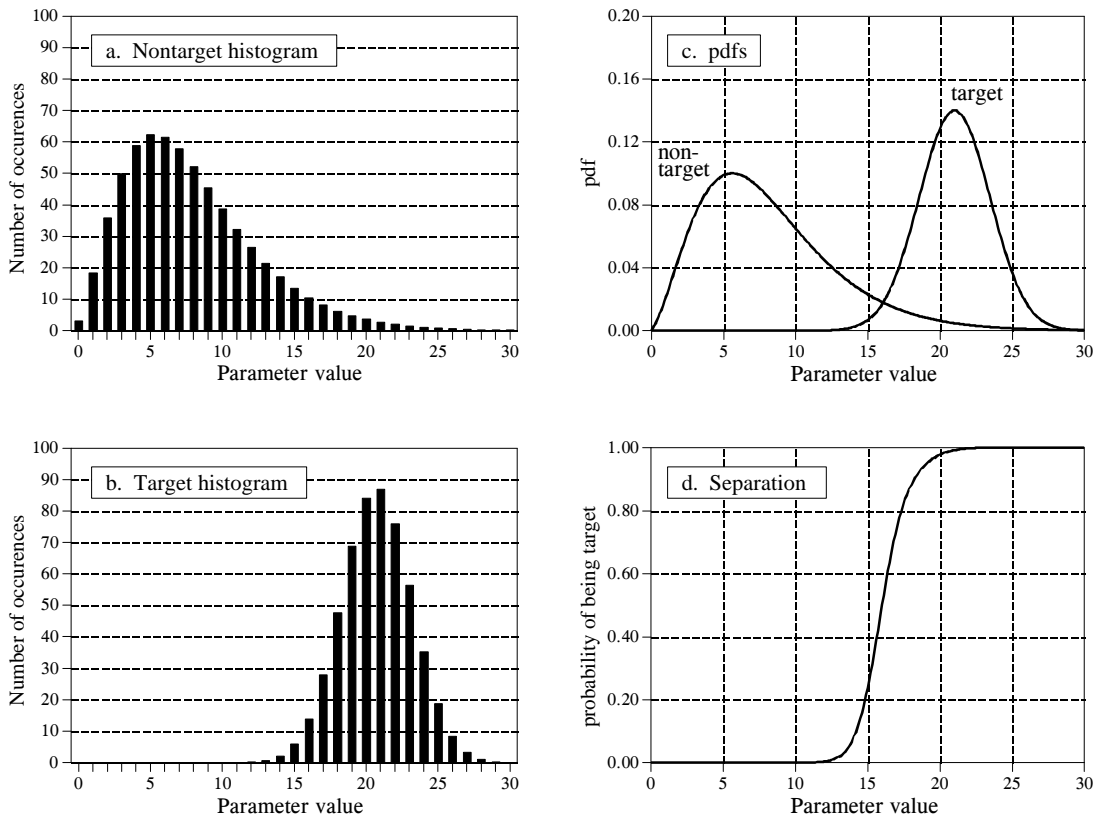
FIGURE 26-1
Probability of target detection. Figures (a) and (b) shows histograms of target and nontarget groups with respect to some parameter value. From these histograms, the probability distribution functions of the two groups can be estimated, as shown in (c). Using only this information, the curve in (d) can be calculated, giving the probability that a target has been found, based on a specific value of the parameter.

the presence of a target, since *yes* will result in one action and *no* will result in another. This is done by comparing the output value of the test to a **threshold**. If the output is above the threshold, the test is said to be **positive**, indicating that the target is present. If the output is below the threshold, the test is said to be **negative**, indicating that the target is not present. In our cancer example, a negative test result means that the patient is told they are healthy, and sent home. When the test result is positive, additional tests will be performed, such as obtaining a sample of the tissue by insertion of a biopsy needle.

Since the target and nontarget distributions overlap, some test results will not be correct. That is, some patients sent home will actually have cancer, and some patients sent for additional tests will be healthy. In the jargon of target detection, a *correct* classification is called **true**, while an *incorrect* classification is called **false**. For example, if a patient has cancer, and the test properly detects the condition, it is said to be a **true-positive**. Likewise, if a patient does not have cancer, and the test indicates that

cancer is not present, it is said to be a **true-negative**. A **false-positive** occurs when the patient does not have cancer, but the test erroneously indicates that they do. This results in needless worry, and the pain and expense of additional tests. An even worse scenario occurs with the **false-negative**, where cancer is present, but the test indicates the patient is healthy. As we all know, untreated cancer can cause many health problems, including premature death.

The human suffering resulting from these two types of errors makes the threshold selection a delicate balancing act. How many *false-positives* can be tolerated to reduce the number of *false-negatives*? Figure 26-2 shows a graphical way of evaluating this problem, the **ROC curve** (short for Receiver Operating Characteristic).   The ROC curve plots the percent of target signals reported as positive (higher is better), against the percent of nontarget signals erroneously reported as positive (lower is better), for various values of the threshold. In other words, each point on the ROC curve represents one possible tradeoff of true-positive and false-positive performance.

Figures (a) through (d) show four settings of the threshold in our cancer detection example.   For instance, look at (b) where the threshold is set at 17. Remember, every test that produces an output value *greater* than the threshold is reported as a *positive* result. About 13% of the area of the nontarget distribution is greater than the threshold (i.e., to the *right* of the threshold). Of all the patients that do not have cancer, 87% will be reported as negative (i.e., a true-negative), while 13% will be reported as positive (i.e., a false-positive). In comparison, about 80% of the area of the target distribution is greater than the threshold.   This means that 80% of those that have cancer will generate a positive test result (i.e., a true-positive). The other 20% that have cancer will be incorrectly reported as a negative (i.e., a false-negative).   As shown in the ROC curve in (b), this threshold results in a point on the curve at: *% nontargets positive = 13%*, and *% targets positive = 80%*.

The more efficient the detection process, the more the ROC curve will bend toward the upper-left corner of the graph. Pure guessing results in a straight line at a 45° diagonal. Setting the threshold relatively low, as shown in (a), results in nearly all the target signals being detected. This comes at the price of many false alarms (false-positives). As illustrated in (d), setting the threshold relatively high provides the reverse situation: few false alarms, but many missed targets.

These analysis techniques are useful in understanding the consequences of threshold selection, but the final decision is based on what some *human* will accept. Suppose you initially set the threshold of the cancer detection apparatus to some value you feel is appropriate.   After many patients have been screened with the system, you speak with a dozen or so patients that have been subjected to false-positives. Hearing how *your* system has unnecessarily disrupted the lives of these people affects you deeply, motivating you to increase the threshold. Eventually you encounter a
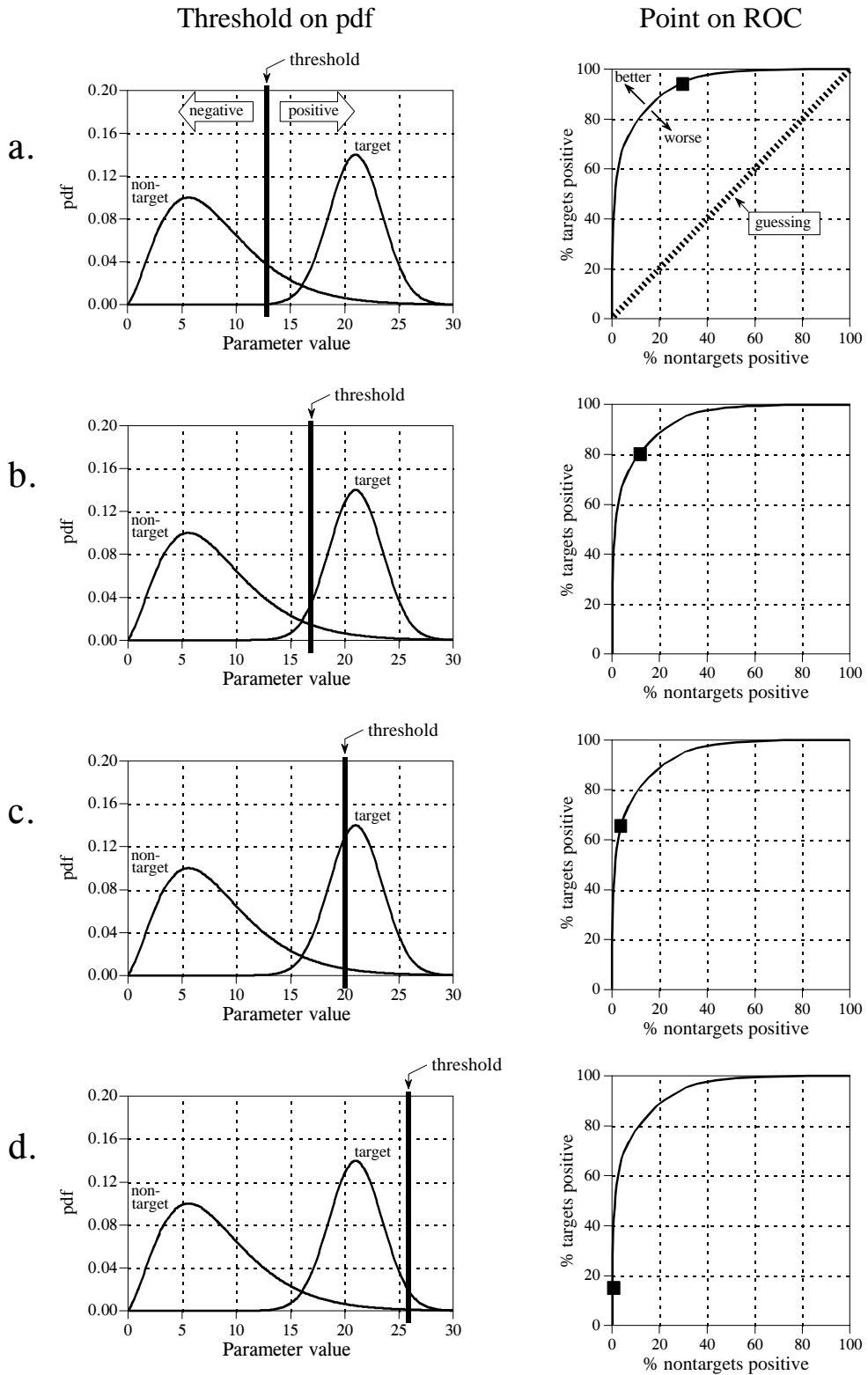
Threshold on pdf   Point on ROC

FIGURE 26-2
Relationship between ROC curves and pdfs.
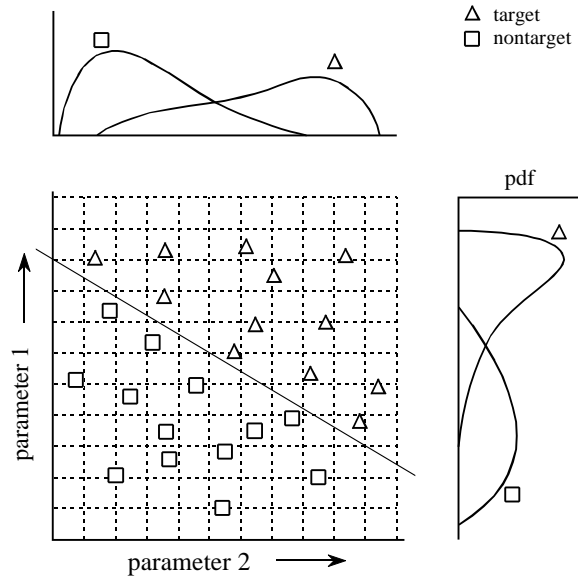
FIGURE 26-3
Example of a two-parameter space. The target (△) and nontarget (□) groups are completely separate in two-dimensions; however, they overlap in each individual parameter. This overlap is shown by the one-dimensional pdfs along each of the parameter axes.

situation that makes you feel even worse: you speak with a patient who is terminally ill with a cancer that *your* system failed to detect. You respond to this difficult experience by *greatly* lowering the threshold. As time goes on and these events are repeated many times, the threshold gradually moves to an *equilibrium* value. That is, the false-positive rate multiplied by a significance factor (lowering the threshold) is balanced by the false-negative rate multiplied by another significance factor (raising the threshold).

This analysis can be extended to devices that provide more than one output. For example, suppose that a cancer detection system operates by taking an x-ray image of the subject, followed by automated image analysis algorithms to identify tumors. The algorithms identify suspicious regions, and then measure key characteristics to aid in the evaluation. For instance, suppose we measure the *diameter* of the suspect region (parameter 1) and its *brightness* in the image (parameter 2). Further suppose that our research indicates that tumors are generally larger and brighter than normal tissue. As a first try, we could go through the previously presented ROC analysis for each parameter, and find an acceptable threshold for each. We could then classify a test as positive only if it met both criteria: parameter 1 greater than some threshold *and* parameter 2 greater than another threshold.

This technique of thresholding the parameters separately and then invoking logic functions (AND, OR, etc.) is very common. Nevertheless, it is very inefficient, and much better methods are available. Figure 26-3 shows why this is the case. In this figure, each triangle represents a single occurrence of a target (a patient with cancer), plotted at a location that corresponds to the value of its two parameters. Likewise, each square represents a single occurrence of a nontarget (a patient without cancer). As shown in the pdf

FIGURE 26-4
Example of a three-parameter space.
Just as a two-parameter space forms a
plane surface, a three parameter space
can be graphically represented using
the conventional x, y, and z axes.
Separation of a three-parameter space
into regions requires a dividing *plane*,
or a curved *surface*.
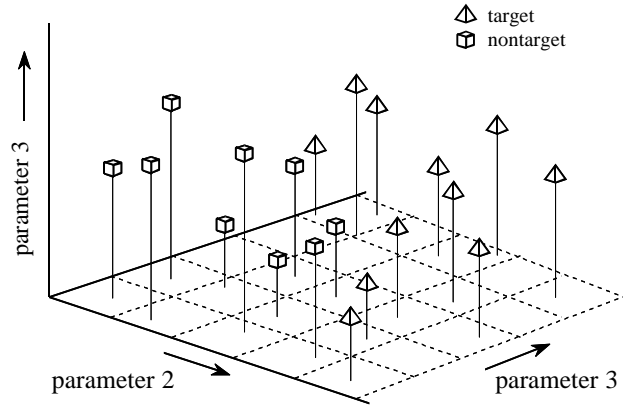
graph on the side of each axis, both parameters have a large overlap between
the target and nontarget distributions. In other words, each parameter, taken
individually, is a poor predictor of cancer. Combining the two parameters with
simple logic functions would only provide a small improvement. This is
especially interesting since the two parameters contain information to *perfectly*
separate the targets from the nontargets. This is done by drawing a diagonal
line between the two groups, as shown in the figure.

In the jargon of the field, this type of coordinate system is called a
parameter space.   For example, the two-dimensional plane in this example
could be called a diameter-brightness space. The idea is that targets will
occupy one region of the parameter space, while nontargets will occupy
another. Separation between the two regions may be as simple as a straight
line, or as complicated as closed regions with irregular borders. Figure 26-
4 shows the next level of complexity, a three-parameter space being
represented on the x, y and z axes. For example, this might correspond to
a cancer detection system that measures *diameter*, *brightness*, and some
third parameter, say, *edge sharpness*. Just as in the two-dimensional case,
the important idea is that the members of the target and nontarget groups
will (hopefully) occupy different regions of the space, allowing the two to
be separated. In three dimensions, regions are separated by planes and
curved surfaces. The term **hyperspace** (over, above, or beyond normal
space) is often used to describe parameter spaces with more than three
dimensions. Mathematically, hyperspaces are no different from one, two
and three-dimensional spaces; however, they have the practical problem of
not being able to be displayed in a graphical form in our three-dimensional
universe.

The threshold selected for a single parameter problem cannot (usually) be
classified as right or wrong.   This is because each threshold value results in
a unique combination of false-positives and false-negatives, i.e., some point
along the ROC curve. This is trading one goal for another, and has no
absolutely correct answer. On the other hand, parameter spaces with two or

more parameters can definitely have wrong divisions between regions. For instance, imagine increasing the number of data points in Fig. 26-3, revealing a small overlap between the target and nontarget groups. It would be possible to move the threshold line between the groups to trade the number of false-positives against the number of false-negatives. That is, the diagonal line would be moved toward the top-right, or the bottom-left. However, it would be wrong to rotate the line, because it would increase *both* types of errors.

As suggested by these examples, the conventional approach to target detection (sometimes called pattern recognition) is a two step process. The first step is called **feature extraction**. This uses algorithms to reduce the raw data to a few parameters, such as diameter, brightness, edge sharpness, etc. These parameters are often called **features** or **classifiers**. Feature extraction is needed to reduce the amount of data. For example, a medical x-ray image may contain more than a million pixels. The goal of feature extraction is to distill the information into a more concentrated and manageable form. This type of algorithm development is more of an art than a science. It takes a great deal of experience and skill to look at a problem and say: "*These are the classifiers that best capture the information.*" Trial-and-error plays a significant role.

In the second step, an evaluation is made of the classifiers to determine if the target is present or not.   In other words, some method is used to divide the parameter space into a region that corresponds to the targets, and a region that corresponds to the nontargets. This is quite straightforward for one and two-parameter spaces; the known data points are plotted on a graph (such as Fig. 26-3), and the regions separated by eye. The division is then written into a computer program as an equation, or some other way of defining one region from another. In principle, this same technique can be applied to a three-dimensional parameter space. The problem is, three-dimensional graphs are very difficult for humans to understand and visualize (such as Fig. 26-4). Caution: Don't try this in hyperspace; your brain will explode!

In short, we need a machine that can carry out a multi-parameter space division, according to examples of target and nontarget signals. This ideal target detection system is remarkably close to the main topic of this chapter, the *neural network*.

Neural Network Architecture

Humans and other animals process information with *neural networks*. These are formed from *trillions* of **neurons** (nerve cells) exchanging brief electrical pulses called **action potentials**. Computer algorithms that mimic these biological structures are formally called **artificial neural networks** to distinguish them from the squishy things inside of animals. However, most scientists and engineers are not this formal and use the term *neural network* to include both biological and nonbiological systems.

Neural network research is motivated by two desires: to obtain a better understanding of the human brain, and to develop computers that can deal with abstract and poorly defined problems. For example, conventional computers have trouble understanding speech and recognizing people's faces. In comparison, humans do extremely well at these tasks.

Many different neural network structures have been tried, some based on imitating what a biologist sees under the microscope, some based on a more mathematical analysis of the problem. The most commonly used structure is shown in Fig. 26-5. This neural network is formed in three layers, called the **input layer, hidden layer, and output layer**. Each layer consists of one or more **nodes**, represented in this diagram by the small circles. The lines between the nodes indicate the flow of information from one node to the next. In this particular type of neural network, the information flows only from the input to the output (that is, from left-to-right). Other types of neural networks have more intricate connections, such as feedback paths.

The nodes of the input layer are **passive**, meaning they do not modify the data. They receive a single value on their input, and duplicate the value to
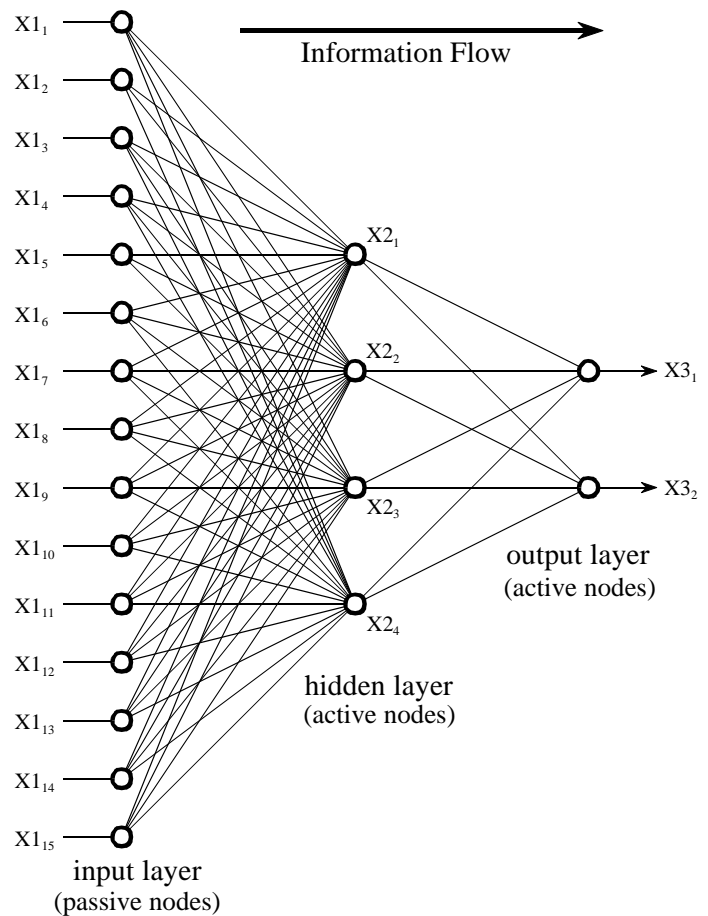
FIGURE 26-5
Neural network architecture. This is the most common structure for neural networks: three layers with full inter-connection. The input layer nodes are passive, doing nothing but relaying the values from their single input to their multiple outputs. In comparison, the nodes of the hidden and output layers are active, modifying the signals in accordance with Fig. 26-6. The action of this neural network is determined by the weights applied in the hidden and output nodes.

their multiple outputs. In comparison, the nodes of the hidden and output layer are **active**. This means they modify the data as shown in Fig. 26-6. The variables: $X1_1, X1_2 \cdots X1_{15}$ hold the data to be evaluated (see Fig. 26-5). For example, they may be pixel values from an image, samples from an audio signal, stock market prices on successive days, etc. They may also be the output of some other algorithm, such as the classifiers in our cancer detection example: diameter, brightness, edge sharpness, etc.

Each value from the input layer is duplicated and sent to *all* of the hidden nodes. This is called a **fully interconnected** structure.   As shown in Fig. 26-6, the values entering a hidden node are multiplied by **weights**, a set of predetermined numbers stored in the program. The weighted inputs are then added to produce a single number. This is shown in the diagram by the symbol, Σ. Before leaving the node, this number is passed through a nonlinear mathematical function called a *sigmoid*. This is an "s" shaped curve that limits the node's output. That is, the input to the sigmoid is a value between $-\infty$ and $+\infty$, while its output can only be between 0 and 1.

The outputs from the hidden layer are represented in the flow diagram (Fig 26-5) by the variables: $X2_1, X2_2, X2_3$ and $X2_4$. Just as before, each of these values is duplicated and applied to the next layer.   The active nodes of the output layer combine and modify the data to produce the two output values of this network, $X3_1$ and $X3_2$.

Neural networks can have any number of layers, and any number of nodes per layer. Most applications use the three layer structure with a maximum of a few hundred input nodes. The hidden layer is usually about 10% the size of the input layer. In the case of target detection, the output layer only needs a single node. The output of this node is thresholded to provide a positive or negative indication of the target's presence or absence in the input data.

Table 26-1 is a program to carry out the flow diagram of Fig. 26-5.   The key point is that this architecture is very simple and very generalized. This same flow diagram can be used for many problems, regardless of their particular quirks. The ability of the neural network to provide useful data manipulation lies in the proper selection of the *weights*. This is a dramatic departure from conventional information processing where solutions are described in step-by-step procedures.

As an example, imagine a neural network for recognizing objects in a sonar signal. Suppose that 1000 samples from the signal are stored in a computer. How does the computer determine if these data represent a submarine, whale, undersea mountain, or nothing at all? Conventional DSP would approach this problem with mathematics and algorithms, such as correlation and frequency spectrum analysis. With a neural network, the 1000 samples are simply fed into the input layer, resulting in values popping from the output layer. By selecting the proper weights, the output can be configured to report a wide range of information. For instance, there might be outputs for: submarine (yes/no), whale (yes/no), undersea mountain (yes/no), etc.

FIGURE 26-6
Neural network active node. This is a flow diagram of the active nodes used in the hidden and output layers of the neural network. Each input is multiplied by a weight (the w_N values), and then summed. This produces a single value that is passed through an "s" shaped nonlinear function called a *sigmoid*. The sigmoid function is shown in more detail in Fig. 26-7.
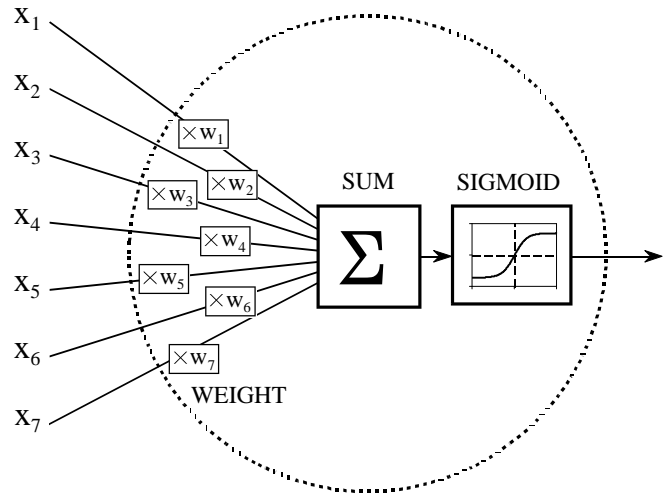
With other weights, the outputs might classify the objects as: metal or non-metal, biological or nonbiological, enemy or ally, etc. No algorithms, no rules, no procedures; only a relationship between the input and output dictated by the values of the weights selected.

```
100 'NEURAL NETWORK (FOR THE FLOW DIAGRAM IN FIG. 26-5)
110 '
120 DIM X1[15]                   'holds the input values
130 DIM X2[4]                    'holds the values exiting the hidden layer
140 DIM X3[2]                    'holds the values exiting the output layer
150 DIM WH[4,15]                 'holds the hidden layer weights
160 DIM WO[2,4]                  'holds the output layer weights
170 '
180 GOSUB XXXX                   'mythical subroutine to load X1[ ] with the input data
190 GOSUB XXXX                   'mythical subroutine to load the weights, WH[ , ] & W0[ , ]
200 '
210 '                            'FIND THE HIDDEN NODE VALUES, X2[  ]
220 FOR J% = 1 TO 4              'loop for each hidden layer node
230   ACC = 0                    'clear the accumulator variable, ACC
240   FOR I% = 1 TO 15           'weight and sum each input node
250     ACC = ACC + X1[I%] * WH[J%,I%]
260   NEXT I%
270   X2[J%] = 1 / (1 + EXP(-ACC) )     'pass summed value through the sigmoid
280 NEXT J%
290 '
300 '                            'FIND THE OUTPUT NODE VALUES, X3[  ]
310 FOR J% = 1 TO 2              'loop for each output layer node
320   ACC = 0                    'clear the accumulator variable, ACC
330   FOR I% = 1 TO 4            'weight and sum each hidden node
340     ACC = ACC + X2[I%] * WO[J%,I%]
350   NEXT I%
360   X3[J%] = 1 / (1 + EXP(-ACC) )     'pass summed value through the sigmoid
370 NEXT J%
380 '
390 END
```
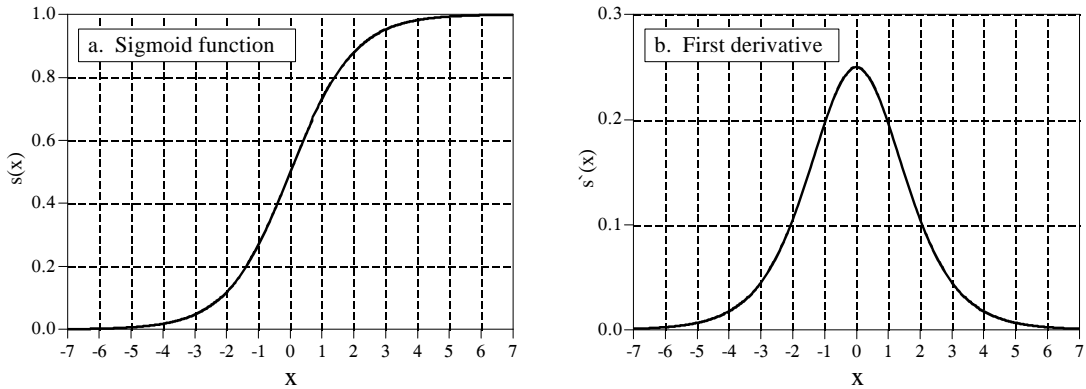
TABLE 26-1

FIGURE 26-7
The sigmoid function and its derivative. Equations 26-1 and 26-2 generate these curves.

Figure 26-7a shows a closer look at the sigmoid function, mathematically described by the equation:

EQUATION 26-1
The sigmoid function. This is used in neural networks as a smooth threshold. This function is graphed in Fig. 26-7a.

$$s(x) = \frac{1}{1 + e^{-x}}$$

The exact shape of the sigmoid is not important, only that it is a **smooth threshold**. For comparison, a **simple threshold** produces a value of *one* when $x > 0$, and a value of *zero* when $x < 0$. The sigmoid performs this same basic thresholding function, but is also *differentiable*, as shown in Fig. 26-7b. While the derivative is not used in the flow diagram (Fig. 25-5), it is a critical part of finding the proper weights to use. More about this shortly. An advantage of the sigmoid is that there is a shortcut to calculating the value of its derivative:

EQUATION 26-2
First derivative of the sigmoid function. This is calculated by using the value of the sigmoid function itself.

$$s'(x) = s(x)[1 - s(x)]$$

For example, if $x = 0$, then $s(x) = 0.5$ (by Eq. 26-1), and the first derivative is calculated: $s'(x) = 0.5(1 - 0.5) = 0.25$. This isn't a critical concept, just a trick to make the algebra shorter.

Wouldn't the neural network be more flexible if the sigmoid could be adjusted left-or-right, making it centered on some other value than $x = 0$? The answer is yes, and most neural networks allow for this. It is very simple to implement; an additional node is added to the input layer, with its input always having a

value of *one*. When this is multiplied by the weights of the hidden layer, it provides a *bias* (DC offset) to each sigmoid. This addition is called a **bias node**. It is treated the same as the other nodes, except for the constant input.

Can neural networks be made without a sigmoid or similar nonlinearity? To answer this, look at the three-layer network of Fig. 26-5. If the sigmoids were not present, the three layers would *collapse* into only two layers. In other words, the summations and weights of the hidden and output layers could be combined into a single layer, resulting in only a two-layer network.

Why Does It Work?

The weights required to make a neural network carry out a particular task are found by a **learning algorithm**, together with **examples** of how the system *should* operate. For instance, the examples in the sonar problem would be a database of several hundred (or more) of the 1000 sample segments. Some of the example segments would correspond to submarines, others to whales, others to random noise, etc. The learning algorithm uses these examples to calculate a set of weights appropriate for the task at hand. The term **learning** is widely used in the neural network field to describe this process; however, a better description might be: *determining an optimized set of weights based on the statistics of the examples*. Regardless of what the method is called, the resulting weights are virtually impossible for humans to understand. Patterns may be observable in some rare cases, but generally they appear to be random numbers. A neural network using these weights can be observed to have the proper input/output relationship, but *why* these particular weights work is quite baffling. This mystic quality of neural networks has caused many scientists and engineers to shy away from them. Remember all those science fiction movies of renegade computers taking over the earth?

In spite of this, it is common to hear neural network advocates make statements such as: "neural networks are well understood." To explore this claim, we will first show that it is possible to pick neural network weights through traditional DSP methods. Next, we will demonstrate that the learning algorithms provide *better* solutions than the traditional techniques. While this doesn't explain *why* a particular set of weights works, it does provide confidence in the method.

In the most sophisticated view, the neural network is a method of labeling the various regions in *parameter space*. For example, consider the sonar system neural network with 1000 inputs and a single output. With proper weight selection, the output will be near *one* if the input signal is an echo from a submarine, and near *zero* if the input is only noise. This forms a parameter hyperspace of 1000 dimensions. The neural network is a method of assigning a value to each location in this hyperspace. That is, the 1000 input values define a *location* in the hyperspace, while the output of the neural network provides the *value* at that location. A look-up table could perform this task perfectly, having an output value stored for each possible input address. The

difference is that the neural network *calculates* the value at each location (address), rather than the impossibly large task of *storing* each value. In fact, neural network architectures are often evaluated by how well they separate the hyperspace for a given number of weights.

This approach also provides a clue to the number of nodes required in the hidden layer. A parameter space of N dimensions requires N numbers to specify a location. Identifying a *region* in the hyperspace requires $2N$ values (i.e., a minimum and maximum value along each axis defines a hyperspace rectangular solid). For instance, these simple calculations would indicate that a neural network with 1000 inputs needs 2000 weights to identify one region of the hyperspace from another. In a fully interconnected network, this would require two hidden nodes. The number of regions needed depends on the particular problem, but can be expected to be far less than the number of dimensions in the parameter space. While this is only a crude approximation, it generally explains why most neural networks can operate with a hidden layer of 2% to 30% the size of the input layer.

A completely different way of understanding neural networks uses the DSP concept of *correlation*. As discussed in Chapter 7, correlation is the optimal way of detecting if a known pattern is contained within a signal. It is carried out by multiplying the signal with the pattern being looked for, and adding the products. The higher the sum, the more the signal resembles the pattern. Now, examine Fig. 26-5 and think of each hidden node as looking for a specific pattern in the input data. That is, each of the hidden nodes *correlates* the input data with the set of weights associated with that hidden node. If the pattern is present, the sum passed to the sigmoid will be large, otherwise it will be small.

The action of the sigmoid is quite interesting in this viewpoint. Look back at Fig. 26-1d and notice that the probability curve separating two bell shaped distributions resembles a sigmoid. If we were manually designing a neural network, we could make the output of each hidden node be the *fractional probability* that a specific pattern is present in the input data. The output layer repeats this operation, making the entire three-layer structure a correlation of correlations, a network that looks for *patterns of patterns*.

Conventional DSP is based on two techniques, *convolution* and *Fourier analysis*. It is reassuring that neural networks can carry out both these operations, plus *much more*. Imagine an N sample signal being filtered to produce another N sample signal. According to the output side view of convolution, each sample in the output signal is a weighted sum of samples from the input. Now, imagine a two-layer neural network with N nodes in each layer. The value produced by each output layer node is also a weighted sum of the input values. If each output layer node uses the same weights as all the other output nodes, the network will implement linear convolution. Likewise, the DFT can be calculated with a two layer neural network with N nodes in each layer. Each output layer node finds the amplitude of one frequency component. This is done by making the weights of each output layer node the same as the sinusoid being looked for. The resulting network correlates the

input signal with each of the basis function sinusoids, thus calculating the DFT. Of course, a two-layer neural network is much less powerful than the standard three layer architecture. This means neural networks can carry out *nonlinear* as well as *linear* processing.

Suppose that one of these conventional DSP strategies is used to design the weights of a neural network. Can it be claimed that the network is *optimal*? Traditional DSP algorithms are usually based on assumptions about the characteristics of the input signal. For instance, Wiener filtering is optimal for maximizing the signal-to-noise ratio *assuming* the signal and noise spectra are both known; correlation is optimal for detecting targets *assuming* the noise is white; deconvolution counteracts an undesired convolution *assuming* the deconvolution kernel is the inverse of the original convolution kernel, etc. The problem is, scientist and engineer's seldom have a perfect knowledge of the input signals that will be encountered. While the underlying mathematics may be elegant, the overall performance is limited by how well the data are understood.

For instance, imagine testing a traditional DSP algorithm with actual input signals. Next, repeat the test with the algorithm changed slightly, say, by increasing one of the parameters by one percent. If the second test result is better than the first, the original algorithm is not optimized for the task at hand. Nearly all conventional DSP algorithms can be significantly improved by a trial-and-error evaluation of small changes to the algorithm's parameters and procedures. This is the strategy of the neural network.

Training the Neural Network

Neural network design can best be explained with an example. Figure 26-8 shows the problem we will attack, identifying individual letters in an image of text. This pattern recognition task has received much attention. It is easy enough that many approaches achieve partial success, but difficult enough that there are no perfect solutions. Many successful commercial products have been based on this problem, such as: reading the addresses on letters for postal routing, document entry into word processors, etc.

The first step in developing a neural network is to create a database of examples. For the text recognition problem, this is accomplished by printing the 26 capital letters: A,B,C,D ⋯ Y,Z, 50 times on a sheet of paper. Next, these 1300 letters are converted into a digital image by using one of the many scanning devices available for personal computers. This large digital image is then divided into small images of 10×10 pixels, each containing a single letter. This information is stored as a 1.3 Megabyte database: 1300 images; 100 pixels per image; 8 bits per pixel. We will use the first 260 images in this database to *train* the neural network (i.e., determine the weights), and the remainder to *test* its performance. The database must also contain a way of identifying the letter contained in each image. For instance, an additional byte could be added to each 10×10 image, containing the letter's ASCII code. In another scheme, the position

FIGURE 26-8
Example image of text. Identifying letters in images of text is one of the classic pattern recognition problems. In this example, each letter is contained in a 10×10 pixel image, with 256 gray levels per pixel. The database used to train and test the example neural network consists of 50 sets of the 26 capital letters, for a total of 1300 images. The images shown here are a portion of this database.

of each 10×10 image in the database could indicate what the letter is. For example, images 0 to 49 might all be an "A", images 50-99 might all be a "B", etc.

For this demonstration, the neural network will be designed for an arbitrary task: determine which of the 10×10 images contains a *vowel*, i.e., A, E, I, O, or U. This may not have any practical application, but it does illustrate the ability of the neural network to learn very abstract pattern recognition problems. By including ten examples of each letter in the training set, the network will (hopefully) learn the key features that distinguish the target from the nontarget images.

The neural network used in this example is the traditional three-layer, fully interconnected architecture, as shown in Figs. 26-5 and 26-6. There are 101 nodes in the input layer (100 pixel values plus a bias node), 10 nodes in the hidden layer, and 1 node in the output layer. When a 100 pixel image is applied to the input of the network, we want the output value to be close to *one* if a vowel is present, and near *zero* if a vowel is not present. Don't be worried that the input signal was acquired as a two-dimensional array (10×10), while the input to the neural network is a one-dimensional array. This is *your* understanding of how the pixel values are interrelated; the *neural network* will find relationships of its own.

Table 26-2 shows the main program for calculating the neural network weights, with Table 26-3 containing three subroutines called from the main program. The array elements: X1[1] through X1[100], hold the input layer values. In addition, X1[101] always holds a value of 1, providing the input to the bias node. The output values from the hidden nodes are contained

```
100 'NEURAL NETWORK TRAINING (Determination of weights)
110 '
120                                        'INITIALIZE
130 MU = .000005                          'iteration step size
140 DIM X1[101]                           'holds the input layer signal + bias term
150 DIM X2[10]                            'holds the hidden layer signal
160 DIM WH[10,101]                        'holds hidden layer weights
170 DIM WO[10]                            'holds output layer weights
180 '
190 FOR H% = 1 TO 10                      'SET WEIGHTS TO RANDOM VALUES
200   WO[H%] = (RND-0.5)                  'output layer weights: -0.5 to 0.5
210   FOR I% = 1 TO 101                   'hidden layer weights: -0.0005 to 0.0005
220     WH[H%,I%] = (RND-0.5)/1000
230   NEXT I%
240 NEXT H%
250 '
260 '                                     'ITERATION LOOP
270 FOR ITER% = 1 TO 800                  'loop for 800 iterations
280 '
290   ESUM = 0                            'clear the error accumulator, ESUM
300   '
310   FOR LETTER% = 1 TO 260              'loop for each letter in the training set
320     GOSUB 1000                        'load X1[ ] with training set
330     GOSUB 2000                        'find the error for this letter, ELET
340     ESUM = ESUM + ELET^2              'accumulate error for this iteration
350     GOSUB 3000                        'find the new weights
360   NEXT LETTER%
370   '
380   PRINT  ITER%  ESUM                  'print the progress to the video screen
390 '
400 NEXT ITER%
410 '
420 GOSUB XXXX                            'mythical subroutine to save the weights
430 END
```

TABLE 26-2

in the array elements: X2[1] through X2[10]. The variable, X3, contains the network's output value.   The weights of the hidden layer are contained in the array, WH[,], where the first index identifies the hidden node (1 to 10), and the second index is the input layer node (1 to 101). The weights of the output layer are held in WO[1] to WO[10]. This makes a total of 1020 weight values that define how the network will operate.

The first action of the program is to set each weight to an arbitrary initial value by using a random number generator. As shown in lines 190 to 240, the hidden layer weights are assigned initial values between -0.0005 and 0.0005, while the output layer weights are between -0.5 and 0.5.   These ranges are chosen to be the same order of magnitude that the final weights *must* be. This is based on: (1) the range of values in the input signal, (2) the number of inputs summed at each node, and (3) the range of values over which the sigmoid is active, an input of about $-5 < x < 5$, and an output of 0 to 1. For instance, when 101 inputs with a typical value of 100 are multiplied by the typical weight value of 0.0002, the sum of the products is about 2, which is in the active range of the sigmoid's input.

If we evaluated the performance of the neural network using these random weights, we would expect it to be the same as random guessing. The learning algorithm improves the performance of the network by gradually changing each weight in the proper direction. This is called an **iterative** procedure, and is controlled in the program by the FOR-NEXT loop in lines 270-400. Each iteration makes the weights slightly more efficient at separating the target from the nontarget examples. The iteration loop is usually carried out until no further improvement is being made. In typical neural networks, this may be anywhere from ten to ten-thousand iterations, but a few hundred is common. This example carries out 800 iterations.

In order for this iterative strategy to work, there must be a *single* parameter that describes how well the system is currently performing. The variable ESUM (for error sum) serves this function in the program. The first action inside the iteration loop is to set ESUM to zero (line 290) so that it can be used as an accumulator. At the end of each iteration, the value of ESUM is printed to the video screen (line 380), so that the operator can insure that progress is being made. The value of ESUM will start high, and gradually decrease as the neural network is trained to recognize the targets. Figure 26-9 shows examples of how ESUM decreases as the iterations proceed.

All 260 images in the training set are evaluated during each iteration, as controlled by the FOR-NEXT loop in lines 310-360. Subroutine 1000 is used to retrieve images from the database of examples. Since this is not something of particular interest here, we will only describe the parameters passed to and from this subroutine. Subroutine 1000 is entered with the parameter, LETTER%, being between 1 and 260. Upon return, the input node values, X1[1] to X1[100], contain the pixel values for the image in the database corresponding to LETTER%. The bias node value, X1[101], is always returned with a constant value of *one*. Subroutine 1000 also returns another parameter, CORRECT. This contains the desired output value of the network for this particular letter. That is, if the letter in the image is a vowel, CORRECT will be returned with a value of *one*. If the letter in the image is not a vowel, CORRECT will be returned with a value of *zero*.

After the image being worked on is loaded into X1[1] through X1[100], subroutine 2000 passes the data through the current neural network to produce the output node value, X3. In other words, subroutine 2000 is the same as the program in Table 26-1, except for a different number of nodes in each layer. This subroutine also calculates how well the current network identifies the letter as a target or a nontarget. In line 2210, the variable ELET (for error-letter) is calculated as the difference between the output value actually generated, X3, and the desired value, CORRECT. This makes ELET a value between -1 and 1. All of the 260 values for ELET are combined (line 340) to form ESUM, the total squared error of the network for the entire training set.

Line 2220 shows an option that is often included when calculating the error: assigning a different *importance* to the errors for targets and nontargets. For example, recall the cancer example presented earlier in this chapter,

```
1000 'SUBROUTINE TO LOAD X1[ ] WITH IMAGES FROM THE DATABASE
1010 'Variables entering routine: LETTER%
1020 'Variables exiting routine: X1[1] to X1[100],  X1[101] = 1, CORRECT
1030 '
1040 'The variable, LETTER%, between 1 and 260, indicates which image in the database is to be
1050 'returned in X1[1] to X1[100].  The bias node, X1[101], always has a value of one.  The variable,
1060 'CORRECT, has a value of one if the image being returned is a vowel, and zero otherwise.
1070 '(The details of this subroutine are unimportant, and not listed here).
1900 RETURN


2000 'SUBROUTINE TO CALCULATE THE ERROR WITH THE CURRENT WEIGHTS
2010 'Variables entering routine:  X1[ ], X2[ ], WI[ , ], WH[ ], CORRECT
2020 'Variables exiting routine:   ELET
2030 '
2040 '                                        'FIND THE HIDDEN NODE VALUES, X2[ ]
2050 FOR H% = 1 TO 10                         'loop for each hidden nodes
2060   ACC = 0                                'clear the accumulator
2070   FOR I% = 1 TO 101                      'weight and sum each input node
2080     ACC = ACC + X1[I%] * WH[H%,I%]
2090   NEXT I%
2100   X2[H%] = 1 / (1 + EXP(-ACC) )          'pass summed value through sigmoid
2110 NEXT H%
2120 '
2130 '                                        'FIND THE OUTPUT VALUE: X3
2140 ACC = 0                                  'clear the accumulator
2150 FOR H% = 1 TO 10                         'weight and sum each hidden node
2160   ACC = ACC + X2[H%] * WO[H%]
2170 NEXT H%
2180 X3 = 1 / (1 + EXP(-ACC) )                'pass summed value through sigmoid
2190 '
2200 '                                        'FIND ERROR FOR THIS LETTER, ELET
2210 ELET = (CORRECT - X3)                    'find the error
2220 IF CORRECT = 1 THEN ELET = ELET*5        'give extra weight to targets
2230 '
2240 RETURN


3000 'SUBROUTINE TO FIND NEW WEIGHTS
3010 'Variables entering routine:  X1[ ], X2[ ], X3, WI[ , ], WH[ ], ELET, MU
3020 'Variables exiting routine:   WI[ , ], WH[ ]
3030 '
3040 '                                        'FIND NEW WEIGHTS FOR INPUT NODES
3050 FOR H% = 1 TO 10
3060   FOR I% = 1 TO 101
3070     SLOPEO = X3 * (1 - X3)
3080     SLOPEH = X2(H%) * (1 - X2[H%])
3090     DX3DW = X1[I%] * SLOPEH * WO[H%] * SLOPEO
3100     WH[H%,I%] = WH[H%,I%] + DX3DW * ELET * MU
3110   NEXT I%
3120 NEXT H%
3130 '
3140 '                                        'FIND NEW WEIGHTS FOR HIDDEN NODES
3150 FOR H% = 1 TO 10
3160   SLOPEO = X3 * (1 - X3)
3170   DX3DW = X2[H%] * SLOPEO
3180   WO[H%] = WO[H%] + DX3DW * ELET * MU
3190 NEXT H%
3200 '
3210 RETURN
```

TABLE 26-3

and the consequences of making a false-positive error versus a false-negative error. In the present example, we will arbitrarily declare that the error in detecting a target is *five* times as bad as the error in detecting a nontarget. In effect, this tells the network to do a better job with the targets, even if it hurts the performance of the nontargets.

Subroutine 3000 is the heart of the neural network strategy, the algorithm for changing the weights on each iteration. We will use an analogy to explain the underlying mathematics. Consider the predicament of a military paratrooper dropped behind enemy lines. He parachutes to the ground in unfamiliar territory, only to find it is so dark he can't see more than a few feet away. His orders are to proceed to the bottom of the nearest valley to begin the remainder of his mission. The problem is, without being able to see more than a few feet, how does he make his way to the valley floor? Put another way, he needs an algorithm to adjust his x and y position on the earth's surface in order to *minimize* his elevation. This is analogous to the problem of adjusting the neural network weights, such that the network's error, ESUM, is minimized.

We will look at two algorithms to solve this problem: **evolution** and **steepest descent**. In evolution, the paratrooper takes a flying jump in some random direction. If the new elevation is *higher* than the previous, he curses and returns to his starting location, where he tries again. If the new elevation is *lower*, he feels a measure of success, and repeats the process from the new location.   Eventually he will reach the bottom of the valley, although in a very inefficient and haphazard path. This method is called *evolution* because it is the same type of algorithm employed by nature in biological evolution. Each new generation of a species has random variations from the previous. If these differences are of benefit to the species, they are more likely to be retained and passed to the *next* generation. This is a result of the improvement allowing the animal to receive more food, escape its enemies, produce more offspring, etc. If the new trait is detrimental, the disadvantaged animal becomes lunch for some predator, and the variation is discarded. In this sense, each new generation is an iteration of the evolutionary optimization procedure.

When evolution is used as the training algorithm, each weight in the neural network is slightly changed by adding the value from a random number generator. If the modified weights make a better network (i.e., a lower value for ESUM), the changes are retained, otherwise they are discarded. While this works, it is very slow in **converging**. This is the jargon used to describe that continual improvement is being made toward an optimal solution (the bottom of the valley).   In simpler terms, the program is going to need days to reach a solution, rather than minutes or hours.

Fortunately, the *steepest descent* algorithm is much faster. This is how the paratrooper would naturally respond: evaluate which way is *downhill*, and move in that direction. Think about the situation this way. The paratrooper can move one step to the north, and record the change in elevation. After returning to his original position, he can take one step to the east, and

FIGURE 26-9
Neural network convergence. This graph shows how the neural network error (the value of ESUM) decreases as the iterations proceed. Three separate trials are shown, each starting with different initial weights.
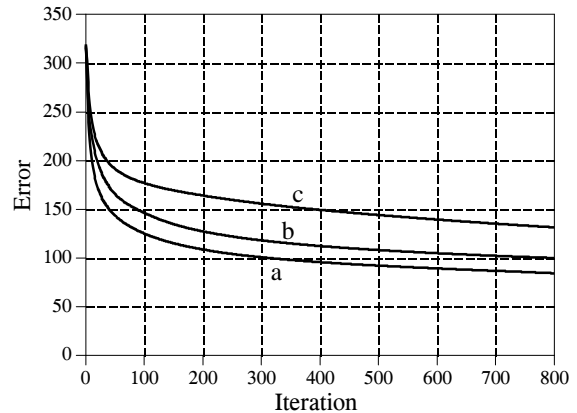
record that elevation change. Using these two values, he can determine which direction is downhill. Suppose the paratrooper drops 10 cm when he moves one step in the northern direction, and drops 20 cm when he moves one step in the eastern direction. To travel directly downhill, he needs to move along each axis an amount proportional to the slope along that axis. In this case, he might move north by 10 steps and east by 20 steps. This moves him down the steepest part of the slope a distance of $\sqrt{10^2 + 20^2}$ = 22.4 steps. Alternatively, he could move in a straight line to the new location, 22.4 steps along the diagonal. The key point is: *the steepest descent is achieved by moving along each axis a distance proportional to the slope along that axis*.

Subroutine 3000 implements this same steepest decent algorithm for the network weights. Before entering subroutine 3000, one of the example images has been applied to the input layer, and the information propagated to the output. This means that the values for: X1[ ], X2[ ] and X3 are all specified, as well as the current weight values: WH[,] and WO[ ]. In addition, we know the error the network produces for this particular image, ELET. The hidden layer weights are updated in lines 3050 to 3120, while the output layer weights are modified in lines 3150 to 3190. *This is done by calculating the slope for each weight, and then changing each weight by an amount proportional to that slope*. In the paratrooper case, the slope along an axis is found by moving a small distance along the axis (say, Δx), measuring the change in elevation (say, ΔE), and then dividing the two ($\Delta E/\Delta x$). The slope of a neural network weight can be found in this same way: add a small increment to the weight value (Δw), find the resulting change in the output signal ($\Delta X3$), and divide the two ($\Delta X3/\Delta w$). Later in this chapter we will look at an example that calculates the slope this way. However, in the present example we will use a more efficient method.

Earlier we said that the nonlinearity (the sigmoid) needs to be *differentiable*. Here is where we will use this property. If we know the slope at each point on the nonlinearity, we can directly write an equation for the slope of each weight ($\Delta X3/\Delta w)$ without actually having to perturb it. Consider a specific weight, for

example, WO[1], corresponding to the first input of the output node. Look at the structure in Figs. 26-5 and 26-6, and ask: how will the output (*X3*) be affected if this particular weight (*w*) is changed slightly, but everything else is kept the same? The answer is:

EQUATION 26-3
Slope of the output layer weights.
This equation is written for the weight, WO[1].

$$\frac{\Delta X3}{\Delta w} \; = \; X2[1] \; \text{SLOPE}_O$$

where SLOPE_O is the first derivative of the output layer sigmoid, evaluated where we are operating on its curve. In other words, SLOPE_O describes how much the *output* of the sigmoid changes in response to a change in the *input* to the sigmoid. From Eq. 26-2, SLOPE_O can be calculated from the current output value of the sigmoid, X3. This calculation is shown in line 3160. In line 3170, the slope for this weight is calculated via Eq. 26-3, and stored in the variable DX3DW (i.e., $\Delta X3/\Delta w$).

Using a similar analysis, the slope for a weight on the hidden layer, such as WH[1,1], can be found by:

EQUATION 26-4
Slope of the hidden layer weights.
This equation is written for the weight, WH[1,1].

$$\frac{\Delta X3}{\Delta w} \; = \; X1[1] \; \text{SLOPE}_{H1} \, WO[1] \; \text{SLOPE}_O$$

SLOPE_{H1} is the first derivative of the hidden layer sigmoid, evaluated where we are operating on its curve. The other values, X1[1] and WO[1], are simply constants that the weight change sees as it makes its way to the output. In lines 3070 and 3080, the slopes of the sigmoids are calculated using Eq. 26-2. The slope of the hidden layer weight, DX3DW is calculated in line 3090 via Eq. 26-4.

Now that we know the *slope* of each of the weights, we can look at how each weight is changed for the next iteration. The new value for each weight is found by taking the current weight, and adding an amount that is proportional to the slope:

EQUATION 26-5
Updating the weights. Each of the weights is adjusted by adding an amount proportional to the slope of the weight.

$$w_{new} \; = \; w_{old} \; + \; \frac{\Delta X3}{\Delta w} \; \text{ELET} \; \text{MU}$$

This calculation is carried out in line 3100 for the hidden layer, and line 3180 for the output layer. The proportionality constant consists of two factors,

ELET, the error of the network for this particular input, and MU, a constant set at the beginning of the program. To understand the need for ELET in this calculation, imagine that an image placed on the input produces a *small* error in the output signal. Next, imagine that another image applied to the input produces a *large* output error. When adjusting the weights, we want to nudge the network more for the second image than the first. If something is working poorly, we want to change it; if it is working well, we want to leave it alone. This is accomplished by changing each weight in proportion to the current error, ELET.

To understand how MU affects the system, recall the example of the paratrooper. Once he determines the downhill direction, he must decide how far to proceed before reevaluating the slope of the terrain. By making this distance short, one meter for example, he will be able to precisely follow the contours of the terrain and always be moving in an optimal direction. The problem is that he spends most of his time evaluating the slope, rather than actually moving down the hill. In comparison, he could choose the distance to be large, say 1000 meters. While this would allow the paratrooper to move rapidly along the terrain, he might overshoot the downhill path. Too large of a distance makes him jump all over the country-side without making the desired progress.

In the neural network, MU controls how much the weights are changed on each iteration. The value to use depends on the particular problem, being as low as 10^{-6}, or as high as 0.1. From the analogy of the paratrooper, it can be expected that too small of a value will cause the network to converge too slowly. In comparison, too large of a value will cause the convergence to be erratic, and will exhibit chaotic oscillation around the final solution. Unfortunately, the way neural networks react to various values of MU can be difficult to understand or predict. This makes it critical that the network error (i.e., ESUM) be monitored during the training, such as printing it to the video screen at the end of each iteration. If the system isn't converging properly, stop the program and try another value for MU.

Evaluating the Results

So, how does it work? The training program for vowel recognition was run three times using different random values for the initial weights. About one hour is required to complete the 800 iterations on a 100 MHz Pentium personnel computer. Figure 26-9 shows how the error of the network, ESUM, changes over this period. The gradual decline indicates that the network is learning the task, and that the weights reach a near optimal value after several hundred iterations. Each trial produces a different solution to the problem, with a different final performance. This is analogous to the paratrooper starting at different locations, and thereby ending up at the bottom of different valleys. Just as some valleys are deeper than others, some neural network solutions are better than others. This means that the learning algorithm should be run several times, with the best of the group taken as the final solution.
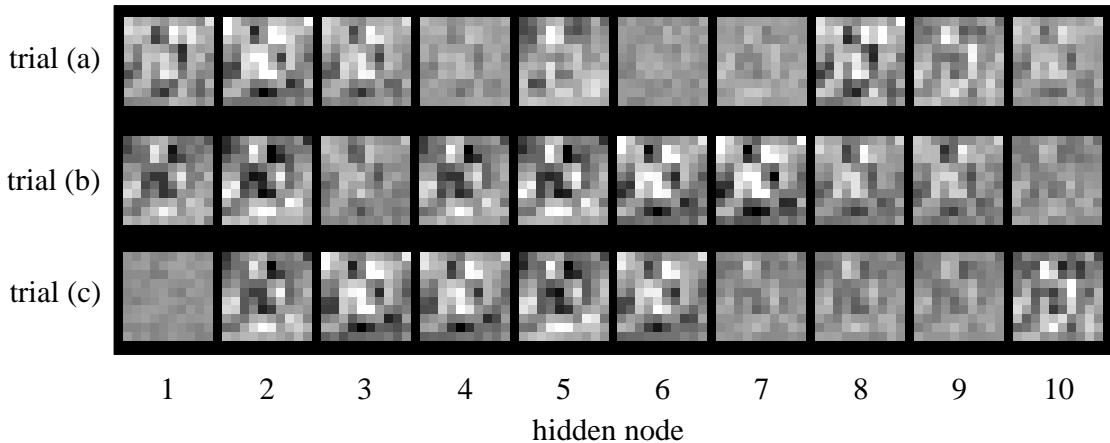
FIGURE 26-10
Example of neural network weights. In this figure, the hidden layer weights for the three solutions are displayed as images. All three of these solutions appear random to the human eye.

In Fig. 26-10, the hidden layer weights of the three solutions are displayed as images. This means the first action taken by the neural network is to correlate (multiply and sum) these images with the input signal. They look like random noise! These weights values can be *shown* to work, but *why* they work is something of a mystery. Here is something else to ponder. The human brain is composed of about 100 *trillion* neurons, each with an average of 10,000 interconnections. If we can't understand the simple neural network in this example, how can we study something that is at least 100,000,000,000,000 times more complex? This is 21st century research.

Figure 26-11a shows a histogram of the neural network's output for the 260 letters in the training set. Remember, the weights were selected to make the output near *one* for vowel images, and near *zero* otherwise. Separation has been perfectly achieved, with no overlap between the two distributions. Also notice that the vowel distribution is narrower than the nonvowel distribution. This is because we declared the target error to be five times more important than the nontarget error (see line 2220).

In comparison, Fig. 26-11b shows the histogram for images 261 through 1300 in the database. While the target and nontarget distributions are reasonably distinct, they are not completely separated. Why does the neural network perform better on the first 260 letters than the last 1040? Figure (a) is cheating! It's easy to take a test if you have already seen the answers. In other words, the neural network is recognizing specific images in the training set, not the general patterns identifying vowels from nonvowels.

Figure 26-12 shows the performance of the three solutions, displayed as ROC curves. Trial (b) provides a significantly better network than the
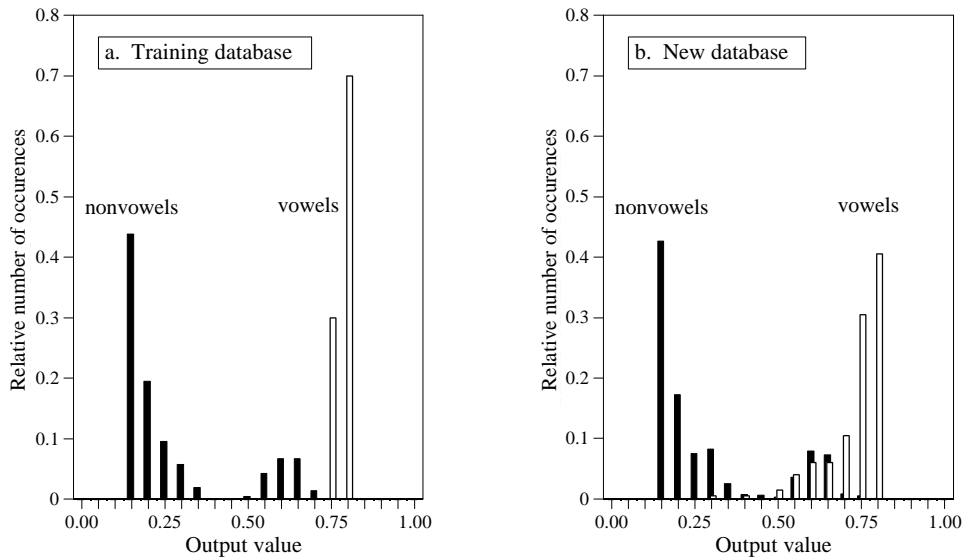
FIGURE 26-11
Neural network performance. These are histograms of the neural network's output values, (a) for the training data, and (b) for the remaining images. The neural network performs better with the training data because it has already seen the answers to the test.

other two. This is a matter of random chance depending on the initial weights used. At one threshold setting, the neural network designed in trial "b" can detect 24 out of 25 targets (i.e., 96% of the vowel images), with a false alarm rate of only 1 in 25 nontargets (i.e., 4% of the nonvowel images). Not bad considering the abstract nature of this problem, and the very general solution applied.

FIGURE 26-12
ROC analysis of neural network examples. These curves compare three neural networks designed to detect images of vowels. Trial (b) is the best solution, shown by its curve being closer to the upper-left corner of the graph. This network can correctly detect 24 out of 25 targets, while providing only 1 false alarm for each 25 nontargets. That is, there is a point on the ROC curve at $x = 4\%$ and $y = 96\%$
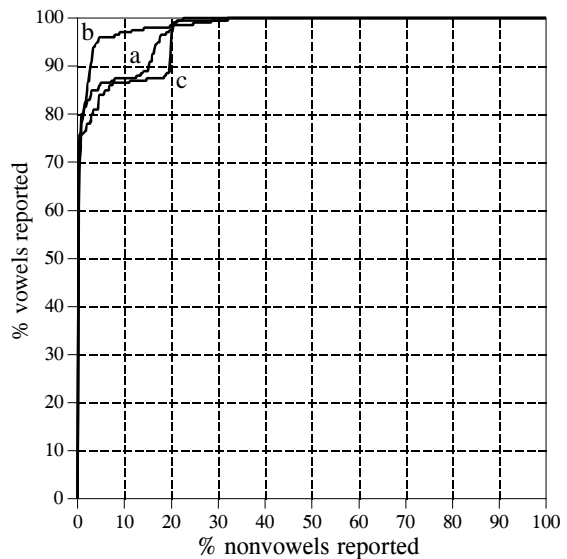
Some final comments on neural networks. Getting a neural network to converge during training can be tricky.   If the network error (ESUM) doesn't steadily decrease, the program must be terminated, changed, and then restarted. This may take several attempts before success is reached. Three things can be changed to affect the convergence: (1) MU, (2) the magnitude of the initial random weights, and (3) the number of hidden nodes (in the order they should be changed).

The most critical item in neural network development is the *validity* of the training examples. For instance, when new commercial products are being developed, the only test data available are from prototypes, simulations, educated guesses, etc. If a neural network is trained on this preliminary information, it might not operate properly in the final application. Any difference between the training database and the eventual data will degrade the neural network's performance (Murphy's law for neural networks). Don't try to second guess the neural network on this issue; you can't!

Recursive Filter Design

Chapters 19 and 20 show how to design recursive filters with the standard frequency responses: high-pass, low-pass, band-pass, etc. What if you need something custom? The answer is to design a recursive filter just as you would a neural network: start with a generic set of recursion coefficients, and use iteration to slowly mold them into what you want. This technique is important for two reasons. First, it allows custom recursive filters to be designed without having to hassle with the mathematics of the z-transform. Second, it shows that the ideas from conventional DSP and neural networks can be combined to form superb algorithms.

The main program for this method is shown in Table 26-4, with two subroutines in Table 26-5. The array, T[ ], holds the desired frequency response, some kind of curve that we have manually designed. Since this program is based around the FFT, the lengths of the signals must be a power of two. As written, this program uses an FFT length of 256, as defined by the variable, N%, in line 130. This means that T[0] to T[128] correspond to the frequencies between 0 and 0.5 of the sampling rate. Only the magnitude is contained in this array; the phase is not controlled in this design, and becomes whatever it becomes.

The recursion coefficients are set to their initial values in lines 270-310, typically selected to be the *identity* system. Don't use random numbers here, or the initial filter will be unstable. The recursion coefficients are held in the arrays, A[ ] and B[ ]. The variable, NP%, sets the number of poles in the designed filter. For example, if NP% is 5, the "a" coefficients run from A[0] to A[5], while the "b" coefficients run from B[1] to B[5].

As previously mentioned, the iterative procedure requires a *single* value that describes how well the current system is functioning. This is provided by the variable, ER (for error), and is calculated in subroutine 3000.   Lines

```
100 'ITERATIVE DESIGN OF RECURSIVE FILTER
110 '
120                                  'INITIALIZE
130 N%  = 256                        'number of points in FFT
140 NP% = 8                          'number of poles in filter
150 DELTA = .00001                   'perturbation increment
160 MU = .2                          'iteration step size
170 DIM REX[255]                     'real part of signal during FFT
180 DIM IMX[255]                     'imaginary part of signal during FFT
190 DIM T[128]                       'desired frequency response (mag only)
200 DIM A[8]                         'the "a" recursion coefficients
210 DIM B[8]                         'the "b" recursion coefficients
220 DIM SA[8]                        'slope for "a" coefficients
230 DIM SB[8]                        'slope for "b" coefficients
240 '
250 GOSUB XXXX                       'mythical subroutine to load T[ ]
260 '
270 FOR P% = 0 TO NP%               'initialize coefficients to the identity system
280   A[P%] = 0
290   B[P%] = 0
300 NEXT P%
310 A[0] = 1
320 '
330 '                                'ITERATION LOOP
340 FOR ITER% = 1 TO 100             'loop for desired number of iterations
350   GOSUB 2000                     'calculate new coefficients
360   PRINT ITER%  ENEW  MU          'print current status to video screen
370   IF ENEW > EOLD THEN MU = MU/2  'adjust the value of MU
380 NEXT ITER%
390 '
400 '
410 FOR P% = 0 TO NP%               'PRINT OUT THE COEFFICIENTS
420   PRINT  A[P%]   B[P%]
430 NEXT P%
440 '
450 END
```

TABLE 26-4

3040 to 3080 load an impulse in the array, IMX[ ]. Next, lines 3100-3150 use this impulse as an input signal to the recursive filter defined by the current values of A[ ] and B[ ]. The output of this filter is thus the *impulse response* of the current system, and is stored in the array, REX[ ]. The system's frequency response is then found by taking the FFT of the impulse response, as shown in line 3170. Subroutine 1000 is the FFT program listed in Table 12-4 in Chapter 12. This FFT subroutine returns the frequency response in rectangular form, overwriting the arrays REX[ ] and IMX[ ].

Lines 3200-3250 then calculate ER, the *mean squared error* between the magnitude of the current frequency response, and the desired frequency response. Pay particular attention to how this error is found. The iterative action of this program optimizes this error, making the way it is defined very important. The FOR-NEXT loop runs through each frequency in the frequency response. For each frequency, line 3220 calculates the magnitude of the current frequency response from the rectangular data. In line 3230, the error at this frequency is found by subtracting the desired magnitude, T[ ], from the current magnitude, MAG. This error is then squared, and added to the

accumulator variable, ER. After looping through each frequency, line 3250 completes the calculation to make ER the mean squared error of the entire frequency response.

Lines 340 to 380 control the iteration loop of the program. Subroutine 2000 is where the changes to the recursion coefficients are made. The first action in this subroutine is to determine the current value of ER, and store it in another variable, EOLD (lines 2040 & 2050).   After the subroutine updates the coefficients, the value of ER is again determined, and assigned to the variable, ENEW (lines 2270 and 2280).

The variable, MU, controls the iteration step size, just as in the previous neural network program. An advanced feature is used in this program: an *automated* adjustment to the value of MU. This is the reason for having the two variables, EOLD and ENEW.   When the program starts, MU is set to the relatively high value of 0.2 (line 160). This allows the convergence to proceed rapidly, but will limit how close the filter can come to an optimal solution. As the iterations proceed, points will be reached where no progress is being made, identified by ENEW being *higher* than EOLD. Each time this occurs, line 370 reduces the value of MU.

Subroutine 2000 updates the recursion coefficients according to the steepest decent method: calculate the slope for each coefficient, and then change the coefficient an amount proportional to its slope. Lines 2080-2130 calculate the slopes for the "a" coefficients, storing them in the array, SA[ ]. Likewise, lines 2150-2200 calculate the slopes for the "b" coefficients, storing them in the array, SB[ ]. Lines 2220-2250 then modify each of the recursion coefficients by an amount proportional to these slopes.   In this program, the proportionality constant is simply the step size, MU. No error term is required in the proportionality constant because there is only *one* example to be matched: the desired frequency response.

The last issue is how the program calculates the slopes of the recursion coefficients. In the neural network example, an *equation* for the slope was derived. This procedure cannot be used here because it would require taking the derivative *across* the DFT. Instead, a brute force method is applied: actually change the recursion coefficient by a small increment, and then directly calculate the new value of ER.   The slope is then found as the change in ER divided by the amount of the increment. Specifically, the current value of ER is found in lines 2040-2050, and stored in the variable, EOLD.   The loop in lines 2080-2130 runs through each of the "a" coefficients. The first action inside this loop is to add a small increment, DELTA, to the recursion coefficient being worked on (line 2090). Subroutine 3000 is invoked in line 2100 to find the value of ER with the modified coefficient. Line 2110 then calculates the slope of this coefficient as: $(ER - EOLD)/DELTA$. Line 2120 then restores the modified coefficient by subtracting the value of DELTA.

Figure 26-13 shows several examples of filters designed using this program. The dotted line is the desired frequency response, while the solid line is the

```
2000 'SUBROUTINE TO CALCULATE THE NEW RECURSION COEFFICIENTS
2010 'Variables entering routine:      A[ ], B[ ], DELTA, MU
2020 'Variables exiting routine:       A[ ], B[ ], EOLD, ENEW
2030 '
2040 GOSUB 3000                                'FIND THE CURRENT ERROR
2050 EOLD = ER                                 'store current error in variable, EOLD
2060 '
2070                                            'FIND THE ERROR SLOPES
2080 FOR P% = 0 TO NP%                          'loop through each "a" coefficient
2090   A[P%] = A[P%] + DELTA                    'add a small increment to the coefficient
2100   GOSUB 3000                               'find the error with the change
2110   SA[P%] = (ER-EOLD)/DELTA                 'calculate the error slope, store in SA[ ]
2120   A[P%] = A[P%] - DELTA                    'return coefficient to original value
2130 NEXT P%
2140 '
2150 FOR P% = 1 TO NP%                          'repeat process for each "b" coefficient
2160   B[P%] = B[P%] + DELTA
2170   GOSUB 3000
2180   SB[P%] = (ER-EOLD)/DELTA                 'calculate the error slope, store in SB[ ]
2190   B[P%] = B[P%] - DELTA
2200 NEXT P%
2210 '                                          'CALCULATE NEW COEFFICIENTS
2220 FOR P% = 0 TO NP%                          'loop through each coefficient
2230   A[P%] = A[P%] - SA[P%] * MU              'adjust coefficients to move "downhill"
2240   B[P%] = B[P%] - SB[P%] * MU
2250 NEXT P%
2260 '
2270 GOSUB 3000                                 'FIND THE NEW ERROR
2280 ENEW = ER                                  'store new error in variable, ENEW
2290 '
2300 RETURN


3000 'SUBROUTINE TO CALCULATE THE FREQUENCY DOMAIN ERROR
3010 'Variables entering routine:      A[ ], B[ ], T[ ]
3020 'Variables exiting routine:       ER
3030 '
3040 FOR I% = 0 TO N%-1                         'LOAD SHIFTED  IMPULSE INTO IMX[ ]
3050   REX[I%] = 0
3060   IMX[I%] = 0
3070 NEXT I%
3080 IMX[12] = 1
3090 '                                          'CALCULATE IMPULSE RESPONSE
3100 FOR I% = 12 TO N%-1
3110   FOR J% = 0 TO NP%
3120     REX[I%] = REX[I%]  +  A[J%] * IMX[I%-J%]  +  B[J%] * REX[I%-J%]
3130   NEXT J%
3140 NEXT I%
3150 IMX[12] = 0
3160 '                                          'CALCULATE THE FFT
3170 GOSUB 1000                                 'Table 12-4, uses REX[ ], IMX[ ], N%
3180 '
3190                                            'FIND FREQUENCY DOMAIN ERROR
3200 ER = 0                                     'zero ER, to use as an accumulator
3210 FOR I% = 0 TO N%/2                         'loop through each positive frequency
3220   MAG = SQR(REX[I%]^2 + IMX[I%]^2)         'rectangular --> polar conversion
3230   ER = ER + ( MAG - T[I%] )^2              'calculate and accumulate squared error
3240 NEXT I%
3250 ER = SQR( ER/(N%/2+1) )                    'finish calculation of error, ER
3260 '
3270 RETURN
```
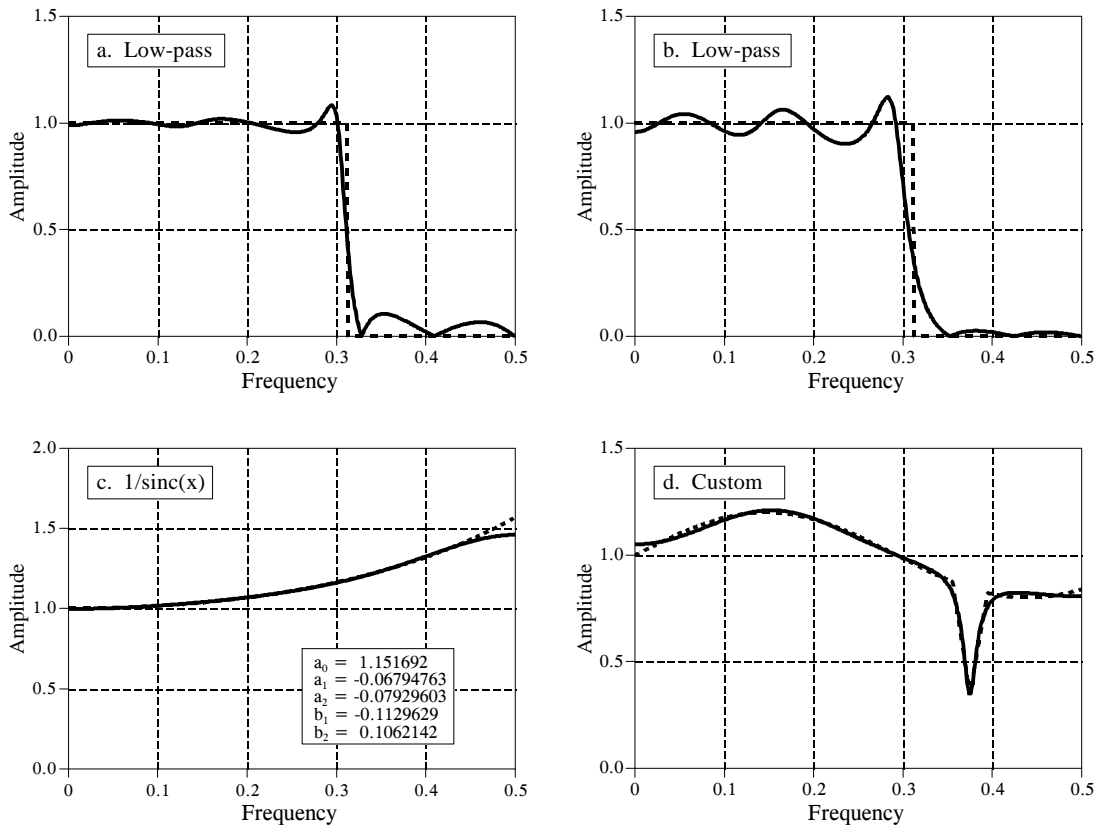
TABLE 26-5

FIGURE 26-13
Iterative design of recursive filters. Figure (a) shows an 8 pole low-pass filter with the error equally distributed between 0 and 0.5. In (b), the error has been weighted to force better performance in the stopband, at the expense of error in the passband. Figure (c) shows a 2 pole filter used for the 1/sinc(x) correction in digital-to-analog conversion. The frequency response in (d) is completely custom. In each figure, the desired frequency response is shown by the dotted line, and the actual frequency response by the solid curve.

frequency response of the designed filter. Each of these filters requires several minutes to converge on a 100 MHz Pentium. Figure (a) is an 8 pole low-pass filter, where the error is equally weighted over the entire frequency spectrum (the program as written). Figure (b) is the same filter, except the error in the stopband is multiplied by *eight* when ER is being calculated. This forces the filter to have less stopband ripple, at the expense of greater ripple in the passband.

Figure (c) shows a 2 pole filter for: $1/sinc(x)$. As discussed in Chapter 3, this can be used to counteract the zeroth-order hold during digital-to-analog conversion (see Fig. 3-6). The error in this filter was only summed between 0 and 0.45, resulting in a better match over this range, at the expense of a worse match between 0.45 and 0.5. Lastly, (d) is a very irregular 6 pole frequency response that includes a sharp dip. To achieve convergence, the recursion coefficients were initially set to those of a notch filter.