

### Recognition Based on Decision-Theoretic Methods

Decision-theoretic approaches to recognition are based on the use of *decision* (or *discriminant*) *functions*. Let  $\mathbf{x} = (x_1, x_2, \dots, x_n)^T$  represent an  $n$ -dimensional pattern vector, as discussed in Section 12.1. For  $W$  pattern classes  $\omega_1, \omega_2, \dots, \omega_W$ , the basic problem in decision-theoretic pattern recognition is to find  $W$  decision functions  $d_1(\mathbf{x}), d_2(\mathbf{x}), \dots, d_W(\mathbf{x})$  with the property that, if a pattern  $\mathbf{x}$  belongs to class  $\omega_i$ , then

$$d_i(\mathbf{x}) > d_j(\mathbf{x}) \quad j = 1, 2, \dots, W; j \neq i. \quad (12.2-1)$$

In other words, an unknown pattern  $\mathbf{x}$  is said to belong to the  $i$ th pattern class if, upon substitution of  $\mathbf{x}$  into all decision functions,  $d_i(\mathbf{x})$  yields the largest numerical value. Ties are resolved arbitrarily.

The *decision boundary* separating class  $\omega_i$  from  $\omega_j$  is given by values of  $\mathbf{x}$  for which  $d_i(\mathbf{x}) = d_j(\mathbf{x})$  or, equivalently, by values of  $\mathbf{x}$  for which

$$d_i(\mathbf{x}) - d_j(\mathbf{x}) = 0. \quad (12.2-2)$$

Common practice is to identify the decision boundary between two classes by the single function  $d_{ij}(\mathbf{x}) = d_i(\mathbf{x}) - d_j(\mathbf{x}) = 0$ . Thus  $d_{ij}(\mathbf{x}) > 0$  for patterns of class  $\omega_i$  and  $d_{ij}(\mathbf{x}) < 0$  for patterns of class  $\omega_j$ . The principal objective of the discussion in this section is to develop various approaches for finding decision functions that satisfy Eq. (12.2-1).

#### 12.2.1 Matching

Recognition techniques based on matching represent each class by a prototype pattern vector. An unknown pattern is assigned to the class to which it is closest in terms of a predefined metric. The simplest approach is the minimum-distance classifier, which, as its name implies, computes the (Euclidean) distance between the unknown and each of the prototype vectors. It chooses the smallest distance to make a decision. We also discuss an approach based on correlation, which can be formulated directly in terms of images and is quite intuitive.

##### Minimum distance classifier

Suppose that we define the prototype of each pattern class to be the mean vector of the patterns of that class:

$$\mathbf{m}_j = \frac{1}{N_j} \sum_{\mathbf{x} \in \omega_j} \mathbf{x}_j \quad j = 1, 2, \dots, W \quad (12.2-3)$$

where  $N_j$  is the number of pattern vectors from class  $\omega_j$  and the summation is taken over these vectors. As before,  $W$  is the number of pattern classes. One way to determine the class membership of an unknown pattern vector  $\mathbf{x}$  is to assign it to the class of its closest prototype, as noted previously. Using the Euclidean distance to determine closeness reduces the problem to computing the distance measures:

$$D_j(\mathbf{x}) = \|\mathbf{x} - \mathbf{m}_j\| \quad j = 1, 2, \dots, W \quad (12.2-4)$$

where  $\|\mathbf{a}\| = (\mathbf{a}^T \mathbf{a})^{1/2}$  is the Euclidean norm. We then assign  $\mathbf{x}$  to class  $\omega_i$  if  $D_i(\mathbf{x})$  is the smallest distance. That is, the smallest distance implies the best match in this formulation. It is not difficult to show (Problem 12.2) that selecting the smallest distance is equivalent to evaluating the functions

$$d_i(\mathbf{x}) = \mathbf{x}^T \mathbf{m}_i - \frac{1}{2} \mathbf{m}_i^T \mathbf{m}_i, \quad j = 1, 2, \dots, W \quad (12.2-5)$$

and assigning  $\mathbf{x}$  to class  $\omega_i$  if  $d_i(\mathbf{x})$  yields the largest numerical value. This formulation agrees with the concept of a decision function, as defined in Eq. (12.2-1).

From Eqs. (12.2-2) and (12.2-5), the decision boundary between classes  $\omega_i$  and  $\omega_j$  for a minimum distance classifier is

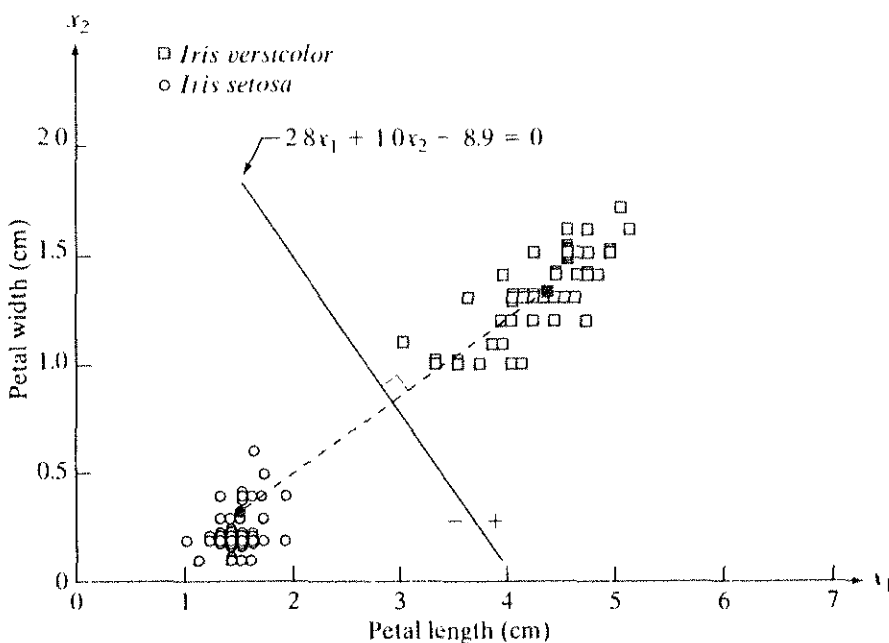
$$\begin{aligned} d_{i,j}(\mathbf{x}) &= d_i(\mathbf{x}) - d_j(\mathbf{x}) \\ &= \mathbf{x}^T (\mathbf{m}_i - \mathbf{m}_j) - \frac{1}{2} (\mathbf{m}_i - \mathbf{m}_j)^T (\mathbf{m}_i - \mathbf{m}_j) = 0. \end{aligned} \quad (12.2-6)$$

The surface given by Eq. (12.2-6) is the perpendicular bisector of the line segment joining  $\mathbf{m}_i$  and  $\mathbf{m}_j$  (see Problem 12.3). For  $n = 2$ , the perpendicular bisector is a line, for  $n = 3$  it is a plane, and for  $n > 3$  it is called a *hyperplane*.

Figure 12.6 shows two pattern classes extracted from the iris samples in Fig. 12.1. The two classes, *Iris versicolor* and *Iris setosa*, denoted  $\omega_1$  and  $\omega_2$ , respectively, have sample mean vectors  $\mathbf{m}_1 = (4.3, 1.3)^T$  and  $\mathbf{m}_2 = (1.5, 0.3)^T$ . From Eq. (12.2-5), the decision functions are

$$\begin{aligned} d_1(\mathbf{x}) &= \mathbf{x}^T \mathbf{m}_1 - \frac{1}{2} \mathbf{m}_1^T \mathbf{m}_1 \\ &= 4.3x_1 + 1.3x_2 - 10.1 \end{aligned}$$

**EXAMPLE 12.1:**  
Illustration of the minimum-distance classifier.



**FIGURE 12.6**  
Decision boundary of minimum distance classifier for the classes of *Iris versicolor* and *Iris setosa*. The dark dot and square are the means.

and

$$\begin{aligned} d_2(\mathbf{x}) &= \mathbf{x}^T \mathbf{m}_2 - \frac{1}{2} \mathbf{m}_2^T \mathbf{m}_2 \\ &= 1.5x_1 + 0.3x_2 - 1.17. \end{aligned}$$

From Eq. (12.2-6), the equation of the boundary is

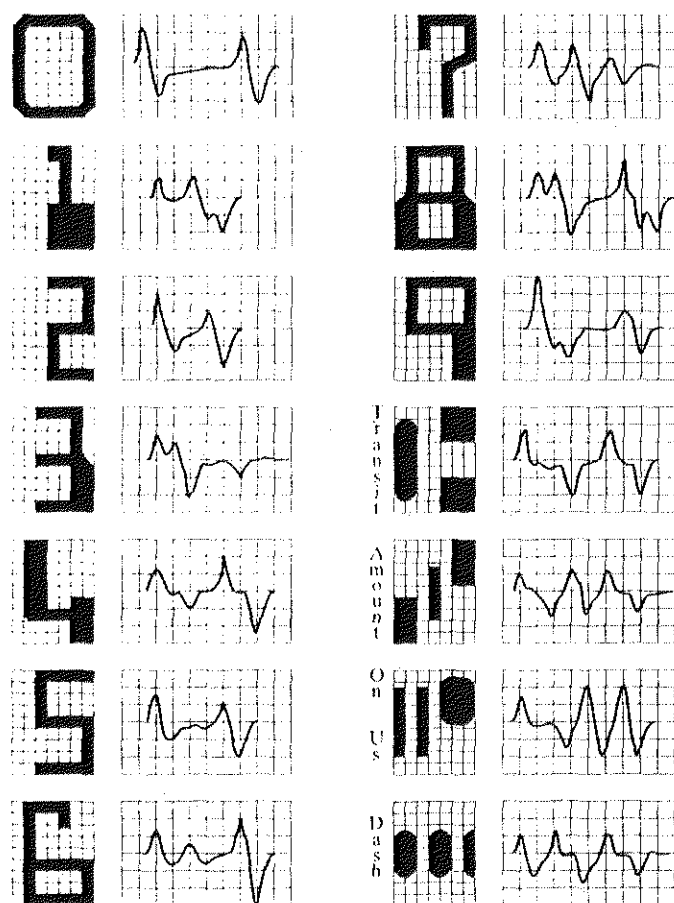
$$\begin{aligned} d_{12}(\mathbf{x}) &= d_1(\mathbf{x}) - d_2(\mathbf{x}) \\ &= 2.8x_1 + 1.0x_2 - 8.9 = 0. \end{aligned}$$

Figure 12.6 shows a plot of this boundary (note that the axes are not to the same scale). Substitution of any pattern vector from class  $\omega_1$  would yield  $d_{12}(\mathbf{x}) > 0$ . Conversely, any pattern from class  $\omega_2$  would yield  $d_{12}(\mathbf{x}) < 0$ . In other words, given an unknown pattern belonging to one of these two classes, the sign of  $d_{12}(\mathbf{x})$  would be sufficient to determine the pattern's class membership.

In practice, the minimum distance classifier works well when the distance between means is large compared to the spread or randomness of each class with respect to its mean. In Section 12.2.2 we show that the minimum distance classifier yields optimum performance (in terms of minimizing the average loss of misclassification) when the distribution of each class about its mean is in the form of a spherical "hypercloud" in  $n$ -dimensional pattern space.

The simultaneous occurrence of large mean separations and relatively small class spread occur seldomly in practice unless the system designer controls the nature of the input. An excellent example is provided by systems designed to read stylized character fonts, such as the familiar American Banker's Association E-13B font character set. As Fig. 12.7 shows, this particular font set consists of 14 characters that were purposely designed on a  $9 \times 7$  grid in order to facilitate their reading. The characters usually are printed in ink that contains finely ground magnetic material. Prior to being read, the ink is subjected to a magnetic field, which accentuates each character to simplify detection. In other words, the segmentation problem is solved by artificially highlighting the key characteristics of each character.

The characters typically are scanned in a horizontal direction with a single-slit reading head that is narrower but taller than the characters. As the head moves across a character, it produces a 1-D electrical signal (a signature) that is conditioned to be proportional to the rate of increase or decrease of the character area under the head. For example, consider the waveform associated with the number 0 in Fig. 12.7. As the reading head moves from left to right, the area seen by the head begins to increase, producing a positive derivative (a positive rate of change). As the head begins to leave the left leg of the 0, the area under the head begins to decrease, producing a negative derivative. When the head is in the middle zone of the character, the area remains nearly constant, producing a zero derivative. This pattern repeats itself as the head enters the right leg of the character. The design of the font ensures that the waveform of each character is distinct from that of all others. It also ensures that the peaks and zeros of each waveform occur approximately on the vertical lines of the background



**FIGURE 12.7**  
American  
Bankers  
Association  
E-13B font  
character set and  
corresponding  
waveforms.

grid on which these waveforms are displayed, as shown in Fig. 12.7. The E-13B font has the property that sampling the waveforms only at these points yields enough information for their proper classification. The use of magnetized ink aids in providing clean waveforms, thus minimizing scatter.

Designing a minimum distance classifier for this application is straightforward. We simply store the sample values of each waveform and let each set of samples be represented as a prototype vector  $\mathbf{m}_j$ ,  $j = 1, 2, \dots, 14$ . When an unknown character is to be classified, the approach is to scan it in the manner just described, express the grid samples of the waveform as a vector,  $\mathbf{x}$ , and identify its class by selecting the class of the prototype vector that yields the highest value in Eq. (12.2-5). High classification speeds can be achieved with analog circuits composed of resistor banks (see Problem 12.4).

### Matching by correlation

We introduced the basic concept of image correlation in Section 4.6.4. Here, we consider it as the basis for finding matches of a subimage  $w(x, y)$  of size  $J \times K$  within an image  $f(x, y)$  of size  $M \times N$ , where we assume that  $J \leq M$  and  $K \leq N$ . Although the correlation approach can be expressed in vector form (see Problem 12.5), working directly with an image or subimage format is more intuitive (and traditional).

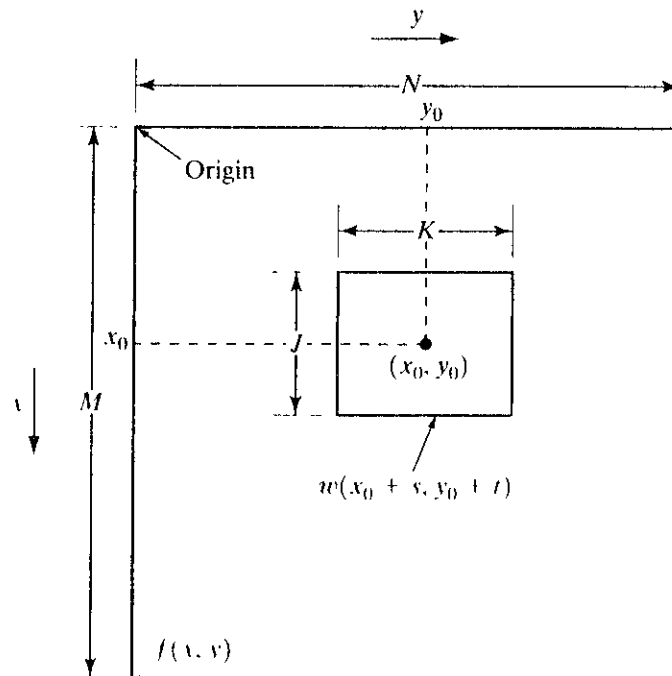
In its simplest form, the correlation between  $f(x, y)$  and  $w(x, y)$  is

$$c(x, y) = \sum_s \sum_t f(s, t)w(x + s, y + t) \quad (12.2-7)$$

for  $x = 0, 1, 2, \dots, M - 1, y = 0, 1, 2, \dots, N - 1$ , and the summation is taken over the image region where  $w$  and  $f$  overlap. Note by comparing this equation with Eq. (4.6-30) that it is implicitly assumed that the functions are real quantities and that we left out the  $MN$  constant. The reason is that we are going to use a normalized function in which these constants cancel out, and the definition given in Eq. (12.2-7) is used commonly in practice. We also used the symbols  $s$  and  $t$  in Eq. (12.2-7) to avoid confusion with  $m$  and  $n$ , which are used for other purposes in this chapter.

Figure 12.8 illustrates the procedure, where we assume that the origin of  $f$  is at its top left and the origin of  $w$  is at its center. For *one* value of  $(x, y)$ , say,  $(x_0, y_0)$  inside  $f$ , application of Eq. (12.2-7) yields *one* value of  $c$ . As  $x$  and  $y$  are varied,  $w$  moves around the image area, giving the function  $c(x, y)$ . The maximum value(s) of  $c$  indicates the position(s) where  $w$  best matches  $f$ . Note that accuracy is lost for values of  $x$  and  $y$  near the edges of  $f$ , with the amount of error being in the correlation proportional to the size of  $w$ . This is the familiar border problem that we encountered numerous times in Chapter 3.

The correlation function given in Eq. (12.2-7) has the disadvantage of being sensitive to changes in the amplitude of  $f$  and  $w$ . For example, doubling all values of  $f$  doubles the value of  $c(x, y)$ . An approach frequently used to over-



**FIGURE 12.8** Arrangement for obtaining the correlation of  $f$  and  $w$  at point  $(x_0, y_0)$ .

come this difficulty is to perform matching via the *correlation coefficient*, which is defined as

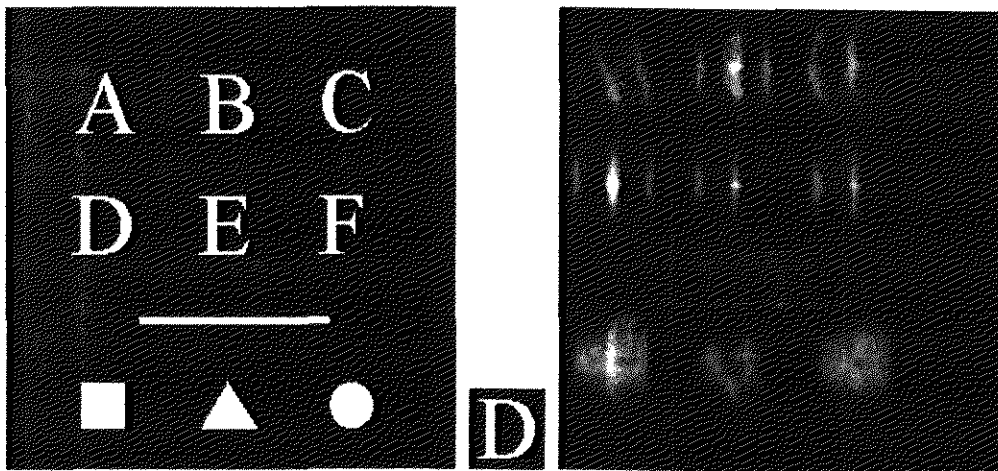
$$\gamma(x, y) = \frac{\sum_s \sum_t [f(s, t) - \bar{f}(s, t)][w(x + s, y + t) - \bar{w}]}{\left\{ \sum_s \sum_t [f(s, t) - \bar{f}(s, t)]^2 \sum_s \sum_t [w(x + s, y + t) - \bar{w}]^2 \right\}^{\frac{1}{2}}} \quad (12.2-8)$$

where  $x = 0, 1, 2, \dots, M - 1, y = 0, 1, 2, \dots, N - 1, \bar{w}$  is the average value of the pixels in  $w$  (computed only once),  $\bar{f}$  is the average value of  $f$  in the region coincident with the current location of  $w$ , and the summations are taken over the coordinates common to both  $f$  and  $w$ . The correlation coefficient  $\gamma(x, y)$  is scaled in the range  $-1$  to  $1$ , independent of scale changes in the amplitude of  $f$  and  $w$  (see Problem 12.5).

Figure 12.9 illustrates the concepts just discussed. Figure 12.9(a) is  $f(x, y)$  and Fig. 12.9(b) is  $w(x, y)$ . The correlation coefficient  $\gamma(x, y)$  is shown as an image in Fig. 12.9(c). The higher (brighter) value of  $\gamma(x, y)$  is in the position where the best match between  $f$  and  $w$  was found.

**EXAMPLE 12.2:** Object matching via the correlation coefficient.

Although the correlation function can be normalized for amplitude changes via the correlation coefficient, obtaining normalization for changes in size and rotation can be difficult. Normalizing for size involves spatial scaling, a process that in itself adds a significant amount of computation. Normalizing for rotation is even more difficult. If a clue regarding rotation can be extracted from  $f(x, y)$ , then we simply rotate  $w(x, y)$  so that it aligns itself with the degree of rotation in  $f(x, y)$ . However, if the nature of rotation is unknown, looking for the best match requires exhaustive rotations of  $w(x, y)$ . This procedure is impractical and, as a consequence, correlation seldom is used in cases when arbitrary or unconstrained rotation is present.



a b c  
**FIGURE 12.9**  
(a) Image.  
(b) Subimage.  
(c) Correlation coefficient of (a) and (b). Note that the highest (brighter) point in (c) occurs when subimage (b) is coincident with the letter "D" in (a).

In Section 4.6.4 we mentioned that correlation also can be carried out in the frequency domain via the FFT. If  $f$  and  $w$  are the same size, this approach can be more efficient than direct implementation of correlation in the spatial domain. Equation (12.2-7) is used when  $w$  is much smaller than  $f$ . A trade-off estimate performed by Campbell [1969] indicates that, if the number of nonzero terms in  $w$  is less than 132 (a subimage of approximately  $13 \times 13$  pixels), direct implementation of Eq. (12.2-7) is more efficient than the FFT approach. This number, of course, depends on the machine and algorithms used, but it does indicate approximate subimage size at which the frequency domain should be considered as an alternative. The correlation coefficient is more difficult to implement in the frequency domain. It generally is computed directly in the spatial domain.

### 12.2.2 Optimum Statistical Classifiers

In this section we develop a probabilistic approach to recognition. As is true in most fields that deal with measuring and interpreting physical events, probability considerations become important in pattern recognition because of the randomness under which pattern classes normally are generated. As shown in the following discussion, it is possible to derive a classification approach that is optimal in the sense that, on average, its use yields the lowest probability of committing classification errors (see Problem 12.10).

#### Foundation

The probability that a particular pattern  $\mathbf{x}$  comes from class  $\omega_i$  is denoted  $p(\omega_i/\mathbf{x})$ . If the pattern classifier decides that  $\mathbf{x}$  came from  $\omega_j$  when it actually came from  $\omega_i$ , it incurs a loss, denoted  $L_{ij}$ . As pattern  $\mathbf{x}$  may belong to any one of  $W$  classes under consideration, the average loss incurred in assigning  $\mathbf{x}$  to class  $\omega_j$  is

$$r_j(\mathbf{x}) = \sum_{k=1}^W L_{kj} p(\omega_k/\mathbf{x}). \quad (12.2-9)$$

This equation often is called the *conditional average risk* or *loss* in decision-theory terminology.

From basic probability theory, we know that  $p(A/B) = [p(A)p(B/A)]/p(B)$ . Using this expression, we write Eq. (12.2-9) in the form

$$r_j(\mathbf{x}) = \frac{1}{p(\mathbf{x})} \sum_{k=1}^W L_{kj} p(\mathbf{x}/\omega_k) P(\omega_k) \quad (12.2-10)$$

where  $p(\mathbf{x}/\omega_k)$  is the probability density function of the patterns from class  $\omega_k$  and  $P(\omega_k)$  is the probability of occurrence of class  $\omega_k$ . Because  $1/p(\mathbf{x})$  is positive and common to all the  $r_j(\mathbf{x})$ ,  $j = 1, 2, \dots, W$ , it can be dropped from Eq. (12.2-10) without affecting the relative order of these functions from the smallest to the largest value. The expression for the average loss then reduces to

$$r_j(\mathbf{x}) = \sum_{k=1}^W L_{kj} p(\mathbf{x}/\omega_k) P(\omega_k). \quad (12.2-11)$$



See inside front cover

Consult the book web site for a brief review of probability theory.

The classifier has  $W$  possible classes to choose from for any given unknown pattern. If it computes  $r_1(\mathbf{x}), r_2(\mathbf{x}), \dots, r_W(\mathbf{x})$  for each pattern  $\mathbf{x}$  and assigns the pattern to the class with the smallest loss, the total average loss with respect to all decisions will be minimum. The classifier that minimizes the total average loss is called the *Bayes classifier*. Thus the Bayes classifier assigns an unknown pattern  $\mathbf{x}$  to class  $\omega_i$  if  $r_i(\mathbf{x}) < r_j(\mathbf{x})$  for  $j = 1, 2, \dots, W; j \neq i$ . In other words,  $\mathbf{x}$  is assigned to class  $\omega_i$  if

$$\sum_{k=1}^W L_{ki} p(\mathbf{x}/\omega_k) P(\omega_k) < \sum_{q=1}^W L_{qi} p(\mathbf{x}/\omega_q) P(\omega_q) \quad (12.2-12)$$

for all  $j; j \neq i$ . The “loss” for a correct decision generally is assigned a value of zero, and the loss for any incorrect decision usually is assigned the same nonzero value (say, 1). Under these conditions, the loss function becomes

$$L_{ij} = 1 - \delta_{ij} \quad (12.2-13)$$

where  $\delta_{ij} = 1$  if  $i = j$  and  $\delta_{ij} = 0$  if  $i \neq j$ . Equation (12.2-13) indicates a loss of unity for incorrect decisions and a loss of zero for correct decisions. Substituting Eq. (12.2-13) into Eq. (12.2-11) yields

$$\begin{aligned} r_j(\mathbf{x}) &= \sum_{k=1}^W (1 - \delta_{kj}) p(\mathbf{x}/\omega_k) P(\omega_k) \\ &= p(\mathbf{x}) - p(\mathbf{x}/\omega_j) P(\omega_j). \end{aligned} \quad (12.2-14)$$

The Bayes classifier then assigns a pattern  $\mathbf{x}$  to class  $\omega_i$  if, for all  $j \neq i$ ,

$$p(\mathbf{x}) - p(\mathbf{x}/\omega_i) P(\omega_i) < p(\mathbf{x}) - p(\mathbf{x}/\omega_j) P(\omega_j) \quad (12.2-15)$$

or, equivalently, if

$$p(\mathbf{x}/\omega_i) P(\omega_i) > p(\mathbf{x}/\omega_j) P(\omega_j) \quad j = 1, 2, \dots, W; j \neq i. \quad (12.2-16)$$

With reference to the discussion leading to Eq. (12.2-1), we see that the Bayes classifier for a 0-1 loss function is nothing more than computation of decision functions of the form

$$d_j(\mathbf{x}) = p(\mathbf{x}/\omega_j) P(\omega_j) \quad j = 1, 2, \dots, W \quad (12.2-17)$$

where a pattern vector  $\mathbf{x}$  is assigned to the class whose decision function yields the largest numerical value.

The decision functions given in Eq. (12.2-7) are optimal in the sense that they minimize the average loss in misclassification. For this optimality to hold, however, the probability density functions of the patterns in each class, as well as the probability of occurrence of each class, must be known. The latter requirement usually is not a problem. For instance, if all classes are equally likely to occur, then  $P(\omega_j) = 1/M$ . Even if this condition is not true, these probabilities generally can be inferred from knowledge of the problem. Estimation of the probability density functions  $p(\mathbf{x}/\omega_j)$  is another matter. If the pattern vectors,  $\mathbf{x}$ , are  $n$  dimensional, then  $p(\mathbf{x}/\omega_j)$  is a function of  $n$  variables, which, if its form is not known, requires methods from multivariate probability theory for its estimation. These methods are difficult to apply in practice,



especially if the number of representative patterns from each class is not large or if the underlying form of the probability density functions is not well behaved. For these reasons, use of the Bayes classifier generally is based on the assumption of an analytic expression for the various density functions and then an estimation of the necessary parameters from sample patterns from each class. By far the most prevalent form assumed for  $p(\mathbf{x}/\omega_j)$  is the Gaussian probability density function. The closer this assumption is to reality, the closer the Bayes classifier approaches the minimum average loss in classification.

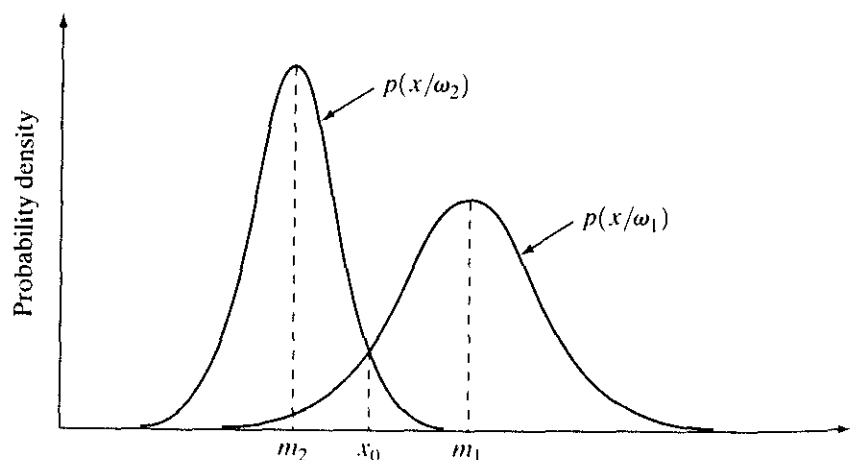
### Bayes classifier for Gaussian pattern classes

To begin, let us consider a 1-D problem ( $n = 1$ ) involving two pattern classes ( $W = 2$ ) governed by Gaussian densities, with means  $m_1$  and  $m_2$  and standard deviations  $\sigma_1$  and  $\sigma_2$ , respectively. From Eq. (12.2-17) the Bayes decision functions have the form

$$\begin{aligned} d_j(x) &= p(x/\omega_j)P(\omega_j) \\ &= \frac{1}{\sqrt{2\pi}\sigma_j} e^{-\frac{(x-m_j)^2}{2\sigma_j^2}} P(\omega_j) \quad j = 1, 2 \end{aligned} \quad (12.2-18)$$

where the patterns are now scalars, denoted by  $x$ . Figure 12.10 shows a plot of the probability density functions for the two classes. The boundary between the two classes is a single point, denoted  $x_0$  such that  $d_1(x_0) = d_2(x_0)$ . If the two classes are equally likely to occur, then  $P(\omega_1) = P(\omega_2) = 1/2$ , and the decision boundary is the value of  $x_0$  for which  $p(x_0/\omega_1) = p(x_0/\omega_2)$ . This point is the intersection of the two probability density functions, as shown in Fig. 12.10. Any pattern (point) to the right of  $x_0$  is classified as belonging to class  $\omega_1$ . Similarly, any pattern to the left of  $x_0$  is classified as belonging to class  $\omega_2$ . When the classes are not equally likely to occur,  $x_0$  moves to the left if class  $\omega_1$  is more likely to occur or, conversely, to the right if class  $\omega_2$  is more likely to occur. This result is to be expected, because the classifier is trying to minimize the loss of misclassification. For instance, in the extreme case, if class  $\omega_2$  never occurs, the classifier would never make a mistake by always assigning all patterns to class  $\omega_1$  (that is,  $x_0$  would move to negative infinity).

**FIGURE 12.10**  
Probability density functions for two 1-D pattern classes. The point  $x_0$  shown is the decision boundary if the two classes are equally likely to occur.



In the  $n$ -dimensional case, the Gaussian density of the vectors in the  $j$ th pattern class has the form

$$p(\mathbf{x}/\omega_j) = \frac{1}{(2\pi)^{n/2} |\mathbf{C}_j|^{1/2}} e^{-\frac{1}{2}(\mathbf{x}-\mathbf{m}_j)^T \mathbf{C}_j^{-1}(\mathbf{x}-\mathbf{m}_j)} \quad (12.2-19)$$

where each density is specified completely by its mean vector  $\mathbf{m}_j$ , and covariance matrix  $\mathbf{C}_j$ , which are defined as

$$\mathbf{m}_j = E_j\{\mathbf{x}\} \quad (12.2-20)$$

and

$$\mathbf{C}_j = E_j\{(\mathbf{x} - \mathbf{m}_j)(\mathbf{x} - \mathbf{m}_j)^T\} \quad (12.2-21)$$

where  $E_j\{\cdot\}$  denotes the expected value of the argument over the patterns of class  $\omega_j$ . In Eq. (12.2-19),  $n$  is the dimensionality of the pattern vectors, and  $|\mathbf{C}_j|$  is the determinant of the matrix  $\mathbf{C}_j$ . Approximating the expected value  $E_j$  by the average value of the quantities in question yields an estimate of the mean vector and covariance matrix:

$$\mathbf{m}_j = \frac{1}{N_j} \sum_{\mathbf{x} \in \omega_j} \mathbf{x} \quad (12.2-22)$$

and

$$\mathbf{C}_j = \frac{1}{N_j} \sum_{\mathbf{x} \in \omega_j} \mathbf{x}\mathbf{x}^T - \mathbf{m}_j\mathbf{m}_j^T \quad (12.2-23)$$

where  $N_j$  is the number of pattern vectors from class  $\omega_j$ , and the summation is taken over these vectors. Later in this section we give an example of how to use these two expressions.

The covariance matrix is symmetric and positive semidefinite. As explained in Section 11.4, the diagonal element  $c_{kk}$  is the variance of the  $k$ th element of the pattern vectors. The off-diagonal element  $c_{jk}$  is the covariance of  $x_j$  and  $x_k$ . The multivariate Gaussian density function reduces to the product of the univariate Gaussian density of each element of  $\mathbf{x}$  when the off-diagonal elements of the covariance matrix are zero. This happens when the vector elements  $x_j$  and  $x_k$  are uncorrelated.

According to Eq. (12.2-17), the Bayes decision function for class  $\omega_j$  is  $d_j(\mathbf{x}) = p(\mathbf{x}/\omega_j)P(\omega_j)$ . However, because of the exponential form of the Gaussian density, working with the natural logarithm of this decision function is more convenient. In other words, we can use the form

$$\begin{aligned} d_j(\mathbf{x}) &= \ln[p(\mathbf{x}/\omega_j)P(\omega_j)] \\ &= \ln p(\mathbf{x}/\omega_j) + \ln P(\omega_j). \end{aligned} \quad (12.2-24)$$

This expression is equivalent to Eq. (12.2-17) in terms of classification performance because the logarithm is a monotonically increasing function. In other words, the numerical *order* of the decision functions in Eqs. (12.2-17) and (12.2-24) is the same. Substituting Eq. (12.2-19) into Eq. (12.2-24) yields

$$d_j(\mathbf{x}) = \ln P(\omega_j) - \frac{n}{2} \ln 2\pi - \frac{1}{2} \ln |\mathbf{C}_j| - \frac{1}{2} [(\mathbf{x} - \mathbf{m}_j)^T \mathbf{C}_j^{-1}(\mathbf{x} - \mathbf{m}_j)]. \quad (12.2-25)$$



See inside front cover  
Consult the book web site  
for a brief review of vectors  
and matrices.

The term  $(n/2) \ln 2\pi$  is the same for all classes, so it can be eliminated from Eq. (12.2-25), which then becomes

$$d_j(\mathbf{x}) = \ln P(\omega_j) - \frac{1}{2} \ln |\mathbf{C}_j| - \frac{1}{2} [(\mathbf{x} - \mathbf{m}_j)^T \mathbf{C}_j^{-1} (\mathbf{x} - \mathbf{m}_j)] \quad (12.2-26)$$

for  $j = 1, 2, \dots, W$ . Equation (12.2-26) represents the Bayes decision functions for Gaussian pattern classes under the condition of a 0-1 loss function.

The decision functions in Eq. (12.2-26) are hyperquadrics (quadratic functions in  $n$ -dimensional space), because no terms higher than the second degree in the components of  $\mathbf{x}$  appear in the equation. Clearly, then, the best that a Bayes classifier for Gaussian patterns can do is to place a general second-order decision surface between each pair of pattern classes. If the pattern populations are truly Gaussian, however, no other surface would yield a lesser average loss in classification.

If all covariance matrices are equal, then  $\mathbf{C}_j = \mathbf{C}$ , for  $j = 1, 2, \dots, W$ . By expanding Eq. (12.2-26) and dropping all terms independent of  $j$ , we obtain

$$d_j(\mathbf{x}) = \ln P(\omega_j) + \mathbf{x}^T \mathbf{C}^{-1} \mathbf{m}_j - \frac{1}{2} \mathbf{m}_j^T \mathbf{C}^{-1} \mathbf{m}_j, \quad (12.2-27)$$

which are linear decision functions (*hyperplanes*) for  $j = 1, 2, \dots, W$ .

If, in addition,  $\mathbf{C} = \mathbf{I}$ , where  $\mathbf{I}$  is the identity matrix, and also  $P(\omega_j) = 1/W$ , for  $j = 1, 2, \dots, W$ , then

$$d_j(\mathbf{x}) = \mathbf{x}^T \mathbf{m}_j - \frac{1}{2} \mathbf{m}_j^T \mathbf{m}_j \quad j = 1, 2, \dots, W. \quad (12.2-28)$$

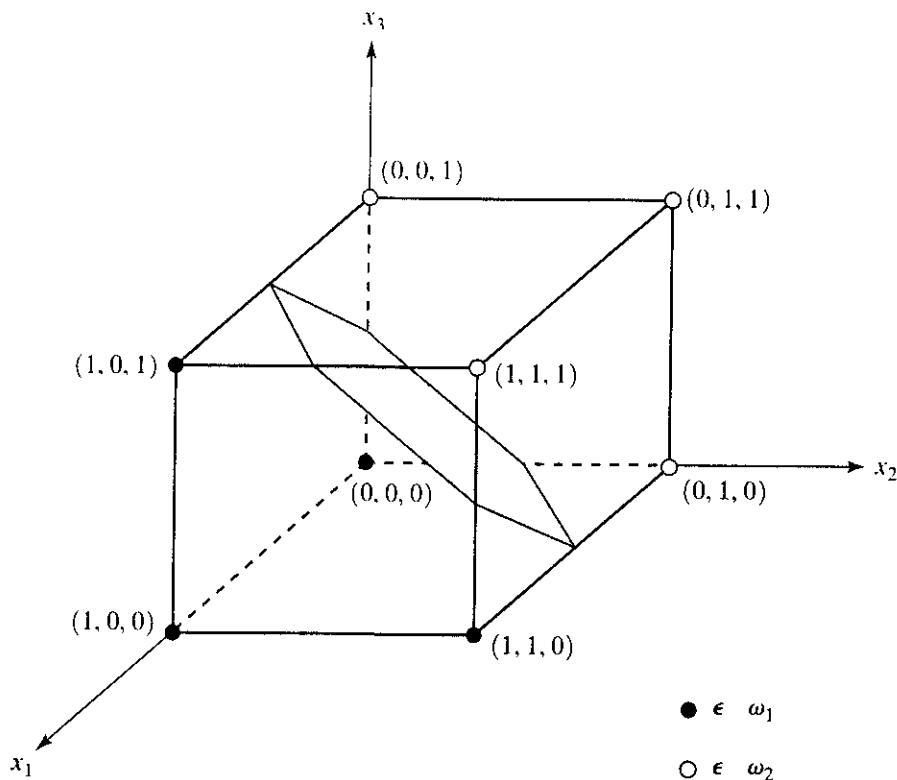
These are the decision functions for a minimum distance classifier, as given in Eq. (12.2-5). Thus the minimum distance classifier is optimum in the Bayes sense if (1) the pattern classes are Gaussian, (2) all covariance matrices are equal to the identity matrix, and (3) all classes are equally likely to occur. Gaussian pattern classes satisfying these conditions are spherical clouds of identical shape in  $n$  dimensions (called *hyperspheres*). The minimum distance classifier establishes a hyperplane between every pair of classes, with the property that the hyperplane is the perpendicular bisector of the line segment joining the center of the pair of hyperspheres. In two dimensions, the classes constitute circular regions, and the boundaries become lines that bisect the line segment joining the center of every pair of such circles.

**EXAMPLE 12.3:**  
A Bayes classifier for three-dimensional patterns.

Figure 12.11 shows a simple arrangement of two pattern classes in three dimensions. We use these patterns to illustrate the mechanics of implementing the Bayes classifier, assuming that the patterns of each class are samples from a Gaussian distribution.

Applying Eq. (12.2-22) to the patterns of Fig. 12.11 yields

$$\mathbf{m}_1 = \frac{1}{4} \begin{bmatrix} 3 \\ 1 \\ 1 \end{bmatrix} \quad \text{and} \quad \mathbf{m}_2 = \frac{1}{4} \begin{bmatrix} 1 \\ 3 \\ 3 \end{bmatrix}.$$



**FIGURE 12.11**  
Two simple pattern classes and their Bayes decision boundary (shown shaded).

Similarly, applying Eq. (12.2-23) to the two pattern classes in turn yields two covariance matrices, which in this case are equal:

$$\mathbf{C}_1 = \mathbf{C}_2 = \frac{1}{16} \begin{bmatrix} 3 & 1 & 1 \\ 1 & 3 & -1 \\ 1 & -1 & 3 \end{bmatrix}.$$

Because the covariance matrices are equal the Bayes decision functions are given by Eq. (12.2-27). If we assume that  $P(\omega_1) = P(\omega_2) = 1/2$ , then Eq. (12.2-28) applies, giving

$$d_j(\mathbf{x}) = \mathbf{x}^T \mathbf{C}^{-1} \mathbf{m}_j - \frac{1}{2} \mathbf{m}_j^T \mathbf{C}^{-1} \mathbf{m}_j$$

in which

$$\mathbf{C}^{-1} = \begin{bmatrix} 8 & -4 & -4 \\ -4 & 8 & 4 \\ -4 & 4 & 8 \end{bmatrix}.$$

Carrying out the vector-matrix expansion for  $d_j(\mathbf{x})$  provides the decision functions:

$$d_1(\mathbf{x}) = 4x_1 - 1.5 \quad \text{and} \quad d_2(\mathbf{x}) = -4x_1 + 8x_2 + 8x_3 - 5.5.$$

The decision surface separating the two classes then is

$$d_1(\mathbf{x}) - d_2(\mathbf{x}) = 8x_1 - 8x_2 - 8x_3 + 4 = 0.$$

Figure 12.11 shows a section of this surface, where we note that the classes were separated effectively.

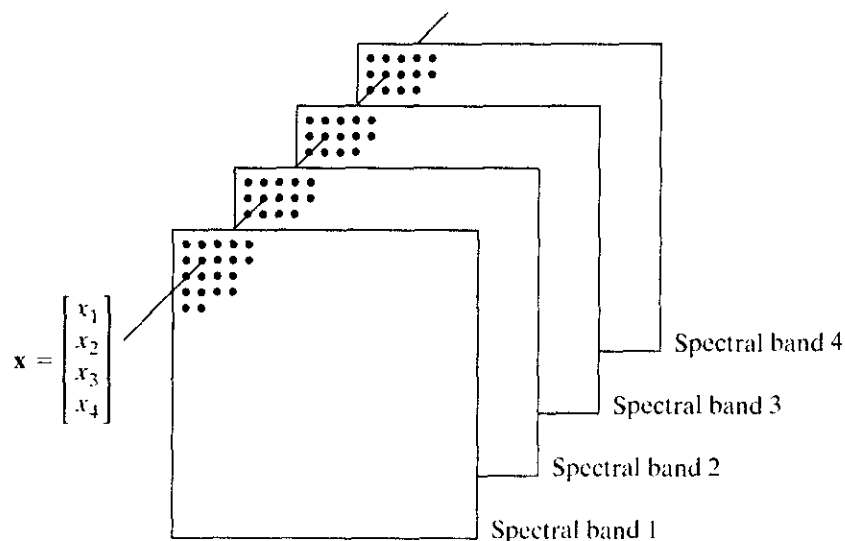
One of the most successful applications of the Bayes classifier approach is in the classification of remotely sensed imagery generated by multispectral scanners aboard aircraft, satellites, or space stations. The voluminous image data generated by these platforms make automatic image classification and analysis a task of considerable interest in remote sensing. The applications of remote sensing are varied and include land use, crop inventory, crop disease detection, forestry, air and water quality monitoring, geological studies, weather prediction, and a score of other applications having environmental significance. The following example shows a typical application.

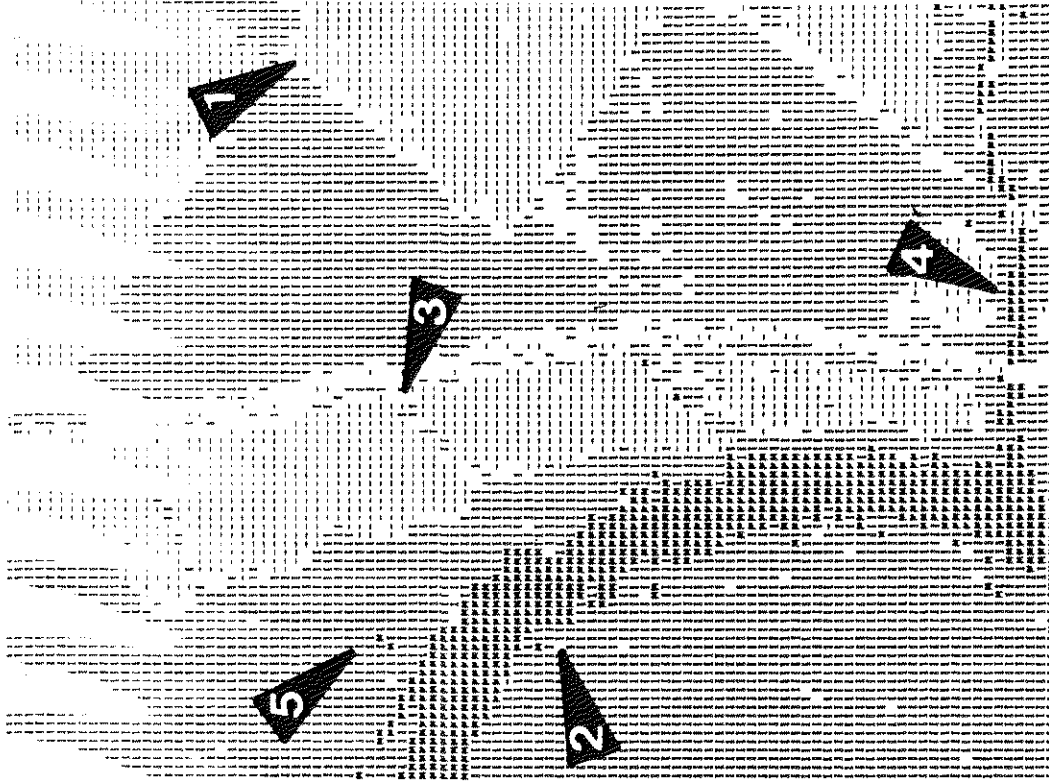
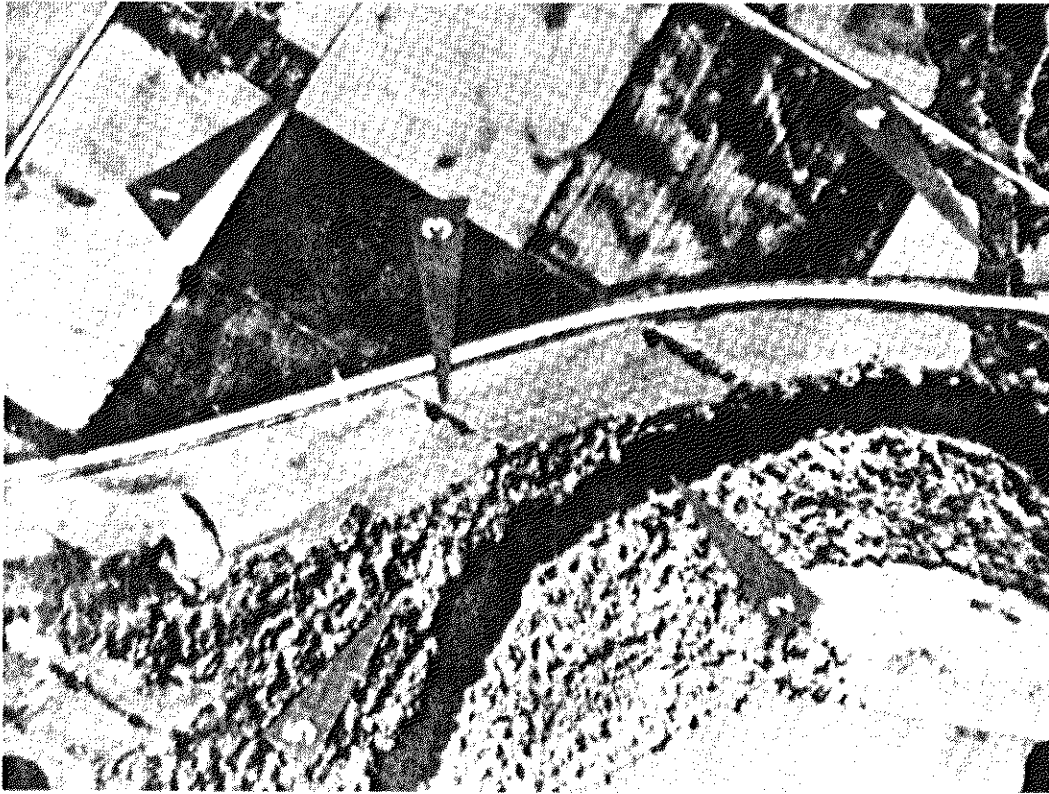
**EXAMPLE 12.4:** Classification of multispectral data using the Bayes classifier.

As discussed in Sections 1.3.4 and 11.4, a multispectral scanner responds to electromagnetic energy in selected wavelength bands; for example, 0.40-0.44, 0.58-0.62, 0.66-0.72, and 0.80-1.00 microns. These ranges are in the violet, green, red, and infrared bands, respectively. A region on the ground scanned in this manner produces four digital images, one image for each band. If the images are registered, a condition which is generally true in practice, they can be visualized as being stacked one behind the other, as Fig. 12.12 shows. Thus, just as we did in Section 11.4, every point on the ground can be represented by a 4-element pattern vector of the form  $\mathbf{x} = (x_1, x_2, x_3, x_4)^T$ , where  $x_1$  is a shade of violet,  $x_2$  is a shade of green, and so on. If the images are of size  $512 \times 512$  pixels, each stack of four multispectral images can be represented by 262,144 4-dimensional pattern vectors.

As noted previously, the Bayes classifier for Gaussian patterns requires estimation of the mean vector and covariance matrix for each class. In remote sensing applications these estimates are obtained by collecting multispectral data for each region of interest and then using these samples, as described in the preceding example. Figure 12.13(a) shows a typical image sensed remotely from an aircraft (this is a monochrome version of a multispectral original). In this

**FIGURE 12.12** Formation of a pattern vector from registered pixels of four digital images generated by a multispectral scanner.





a b

**FIGURE 12.13** (a) Multispectral image. (b) Printout of machine classification results using a Bayes classifier. (Courtesy of the Laboratory for Applications of Remote Sensing, Purdue University.)

particular case, the problem was to classify areas such as vegetation, water, and bare soil. Figure 12.13(b) shows the results of machine classification, using a Gaussian Bayes classifier. The arrows indicate some features of interest. Arrow 1 points to a corner of a field of green vegetation, and arrow 2 points to a river. Arrow 3 identifies a small hedgerow between two areas of bare soil. Arrow 4 indicates a tributary correctly identified by the system. Arrow 5 points to a small pond that is almost indistinguishable in Fig. 12.13(a). Comparing the original image with the computer output reveals recognition results that are very close to those that a human would generate by visual analysis.

Before leaving this section, it is of interest to note that pixel-by-pixel classification of an image as described in the previous example actually segments the image into various classes. This approach is like segmentation by thresholding with several variables, as discussed briefly in Section 10.3.7.

### 12.2.3 Neural Networks

The approaches discussed in the preceding two sections are based on the use of sample patterns to estimate statistical parameters of each pattern class. The minimum distance classifier is specified completely by the mean vector of each class. Similarly, the Bayes classifier for Gaussian populations is specified completely by the mean vector and covariance matrix of each class. The patterns (of *known* class membership) used to estimate these parameters usually are called *training patterns*, and a set of such patterns from each class is called a *training set*. The process by which a training set is used to obtain decision functions is called *learning* or *training*.

In the two approaches just discussed, training is a simple matter. The training patterns of each class are used to compute the parameters of the decision function corresponding to that class. After the parameters in question have been estimated, the structure of the classifier is fixed, and its eventual performance will depend on how well the actual pattern populations satisfy the underlying statistical assumptions made in the derivation of the classification method being used.

The statistical properties of the pattern classes in a problem often are unknown or cannot be estimated (recall our brief discussion in the preceding section regarding the difficulty of working with multivariate statistics). In practice, such decision-theoretic problems are best handled by methods that yield the required decision functions directly via training. Then, making assumptions regarding the underlying probability density functions or other probabilistic information about the pattern classes under consideration is unnecessary. In this section we discuss various approaches that meet this criterion.

### Background

The essence of the material that follows is the use of a multitude of elemental nonlinear computing elements (called *neurons*) organized as networks reminiscent of the way in which neurons are believed to be interconnected in the brain. The resulting models are referred to by various names, including *neural*

*networks, neurocomputers, parallel distributed processing (PDP) models, neuromorphic systems, layered self-adaptive networks, and connectionist models.* Here, we use the name *neural networks*, or *neural nets* for short. We use these networks as vehicles for adaptively developing the coefficients of decision functions via successive presentations of training sets of patterns.

Interest in neural networks dates back to the early 1940s, as exemplified by the work of McCulloch and Pitts [1943]. They proposed neuron models in the form of binary threshold devices and stochastic algorithms involving sudden 0-1 and 1-0 changes of states in neurons as the bases for modeling neural systems. Subsequent work by Hebb [1949] was based on mathematical models that attempted to capture the concept of learning by reinforcement or association.

During the mid-1950s and early 1960s, a class of so-called *learning machines* originated by Rosenblatt [1959, 1962] caused significant excitement among researchers and practitioners of pattern recognition theory. The reason for the great interest in these machines, called *perceptrons*, was the development of mathematical proofs showing that perceptrons, when trained with linearly separable training sets (i.e., training sets separable by a hyperplane), would converge to a solution in a finite number of iterative steps. The solution took the form of coefficients of hyperplanes capable of correctly separating the classes represented by patterns of the training set.

Unfortunately, the expectations following discovery of what appeared to be a well-founded theoretic model of learning soon met with disappointment. The basic perceptron and some of its generalizations at the time were simply inadequate for most pattern recognition tasks of practical significance. Subsequent attempts to extend the power of perceptron-like machines by considering multiple layers of these devices, although conceptually appealing, lacked effective training algorithms such as those that had created interest in the perceptron itself. The state of the field of learning machines in the mid-1960s was summarized by Nilsson [1965]. A few years later, Minsky and Papert [1969] presented a discouraging analysis of the limitation of perceptron-like machines. This view was held as late as the mid-1980s, as evidenced by comments by Simon [1986]. In this work, originally published in French in 1984, Simon dismisses the perceptron under the heading "Birth and Death of a Myth."

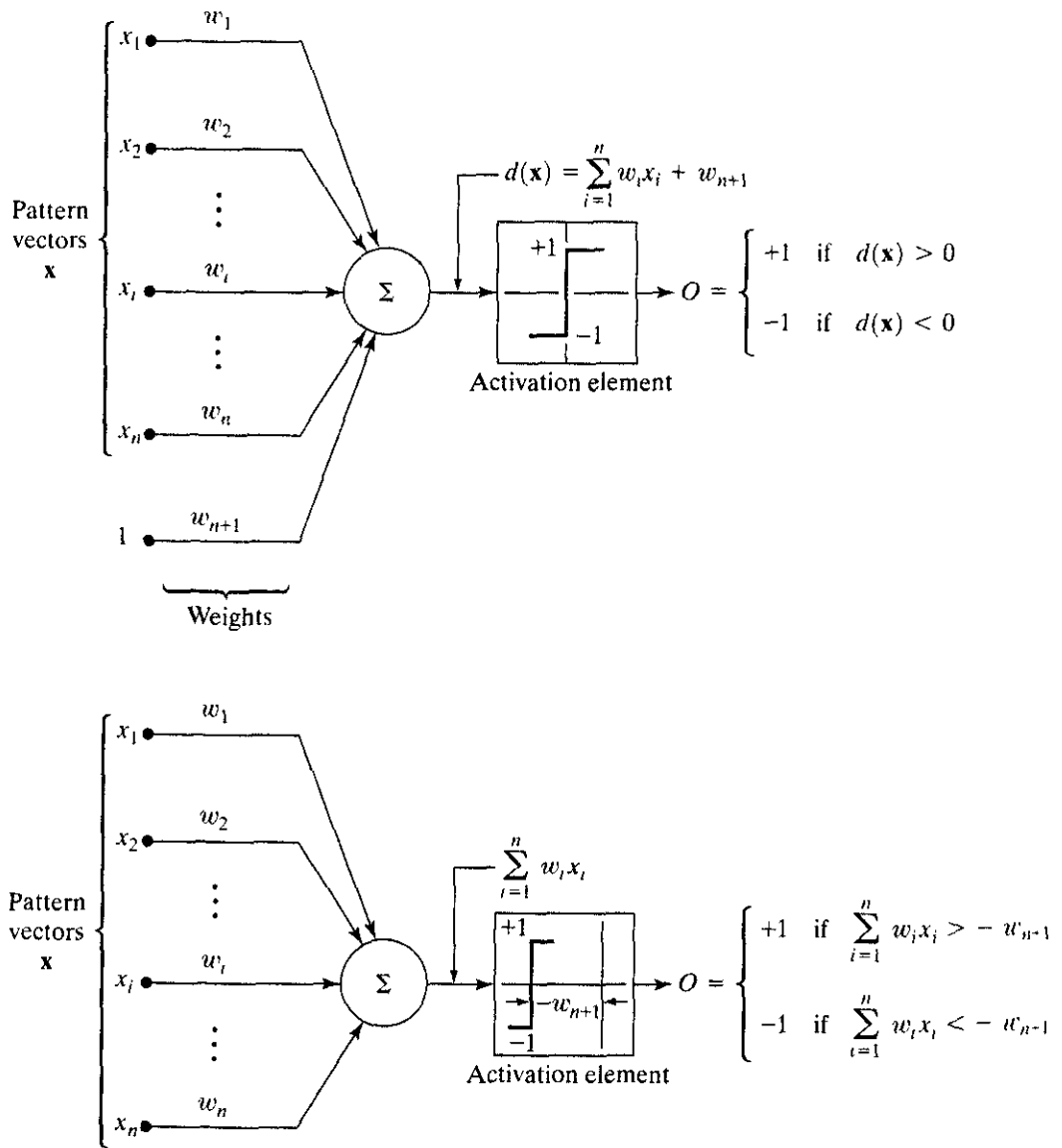
More recent results by Rumelhart, Hinton, and Williams [1986] dealing with the development of new training algorithms for multilayer perceptrons have changed matters considerably. Their basic method, often called the *generalized delta rule for learning by backpropagation*, provides an effective training method for multilayer machines. Although this training algorithm cannot be shown to converge to a solution in the sense of the analogous proof for the single-layer perceptron, the generalized delta rule has been used successfully in numerous problems of practical interest. This success has established multilayer perceptron-like machines as one of the principal models of neural networks currently in use.

### Perceptron for two pattern classes

In its most basic form, the perceptron learns a linear decision function that dichotomizes two linearly separable training sets. Figure 12.14(a) shows schematically the perceptron model for two pattern classes. The response of this basic



a  
b  
**FIGURE 12.14**  
Two equivalent  
representations of  
the perceptron  
model for two  
pattern classes.



device is based on a weighted sum of its inputs; that is,

$$d(\mathbf{x}) = \sum_{i=1}^n w_i x_i + w_{n+1}, \tag{12.2-29}$$

which is a linear decision function with respect to the components of the pattern vectors. The coefficients  $w_i, i = 1, 2, \dots, n, n + 1$ , called *weights*, modify the inputs before they are summed and fed into the threshold element. In this sense, weights are analogous to synapses in the human neural system. The function that maps the output of the summing junction into the final output of the device sometimes is called the *activation function*.

When  $d(\mathbf{x}) > 0$  the threshold element causes the output of the perceptron to be +1, indicating that the pattern  $\mathbf{x}$  was recognized as belonging to class  $w_1$ .

The reverse is true when  $d(\mathbf{x}) < 0$ . This mode of operation agrees with the comments made earlier in connection with Eq. (12.2-2) regarding the use of a single decision function for two pattern classes. When  $d(\mathbf{x}) = 0$ ,  $\mathbf{x}$  lies on the decision surface separating the two pattern classes, giving an indeterminate condition. The decision boundary implemented by the perceptron is obtained by setting Eq. (12.2-29) equal to zero:

$$d(\mathbf{x}) = \sum_{i=1}^n w_i x_i + w_{n+1} = 0 \quad (12.2-30)$$

or

$$w_1 x_1 + w_2 x_2 + \cdots + w_n x_n + w_{n+1} = 0, \quad (12.2-31)$$

which is the equation of a hyperplane in  $n$ -dimensional pattern space. Geometrically, the first  $n$  coefficients establish the orientation of the hyperplane, whereas the last coefficient,  $w_{n+1}$ , is proportional to the perpendicular distance from the origin to the hyperplane. Thus if  $w_{n+1} = 0$ , the hyperplane goes through the origin of the pattern space. Similarly, if  $w_j = 0$ , the hyperplane is parallel to the  $x_j$ -axis.

The output of the threshold element in Fig. 12.14(a) depends on the sign of  $d(\mathbf{x})$ . Instead of testing the entire function to determine whether it is positive or negative, we could test the summation part of Eq. (12.2-29) against the term  $w_{n+1}$ , in which case the output of the system would be

$$O = \begin{cases} +1 & \text{if } \sum_{i=1}^n w_i x_i > -w_{n+1} \\ -1 & \text{if } \sum_{i=1}^n w_i x_i < -w_{n+1}. \end{cases} \quad (12.2-32)$$

This implementation is equivalent to Fig. 12.14(a) and is shown in Fig. 12.14(b), the only differences being that the threshold function is displaced by an amount  $-w_{n+1}$  and that the constant unit input is no longer present. We return to the equivalence of these two formulations later in this section when we discuss implementation of multilayer neural networks.

Another formulation used frequently is to augment the pattern vectors by appending an additional  $(n + 1)$ st element, which is always equal to 1, regardless of class membership. That is, an augmented pattern vector  $\mathbf{y}$  is created from a pattern vector  $\mathbf{x}$  by letting  $y_i = x_i, i = 1, 2, \dots, n$ , and appending the additional element  $y_{n+1} = 1$ . Equation (12.2-29) then becomes

$$\begin{aligned} d(\mathbf{y}) &= \sum_{i=1}^{n+1} w_i y_i \\ &= \mathbf{w}^T \mathbf{y} \end{aligned} \quad (12.2-33)$$

where  $\mathbf{y} = (y_1, y_2, \dots, y_n, 1)^T$  is now an *augmented pattern vector*, and  $\mathbf{w} = (w_1, w_2, \dots, w_n, w_{n+1})^T$  is called the *weight vector*. This expression is usually more convenient in terms of notation. Regardless of the formulation used, however, the key problem is to find  $\mathbf{w}$  by using a given training set of pattern vectors from each of two classes.

**Training algorithms**

The algorithms developed in the following discussion are representative of the numerous approaches proposed over the years for training perceptrons.

*Linearly separable classes* A simple, iterative algorithm for obtaining a solution weight vector for two linearly separable training sets follows. For two training sets of augmented pattern vectors belonging to pattern classes  $\omega_1$  and  $\omega_2$ , respectively, let  $\mathbf{w}(1)$  represent the initial weight vector, which may be chosen arbitrarily. Then, at the  $k$ th iterative step, if  $\mathbf{y}(k) \in \omega_1$  and  $\mathbf{w}^T(k)\mathbf{y}(k) \leq 0$ , replace  $\mathbf{w}(k)$  by

$$\mathbf{w}(k + 1) = \mathbf{w}(k) + c\mathbf{y}(k) \tag{12.2-34}$$

where  $c$  is a positive correction increment. Conversely, if  $\mathbf{y}(k) \in \omega_2$  and  $\mathbf{w}^T(k)\mathbf{y}(k) \geq 0$ , replace  $\mathbf{w}(k)$  with

$$\mathbf{w}(k + 1) = \mathbf{w}(k) - c\mathbf{y}(k). \tag{12.2-35}$$

Otherwise, leave  $\mathbf{w}(k)$  unchanged:

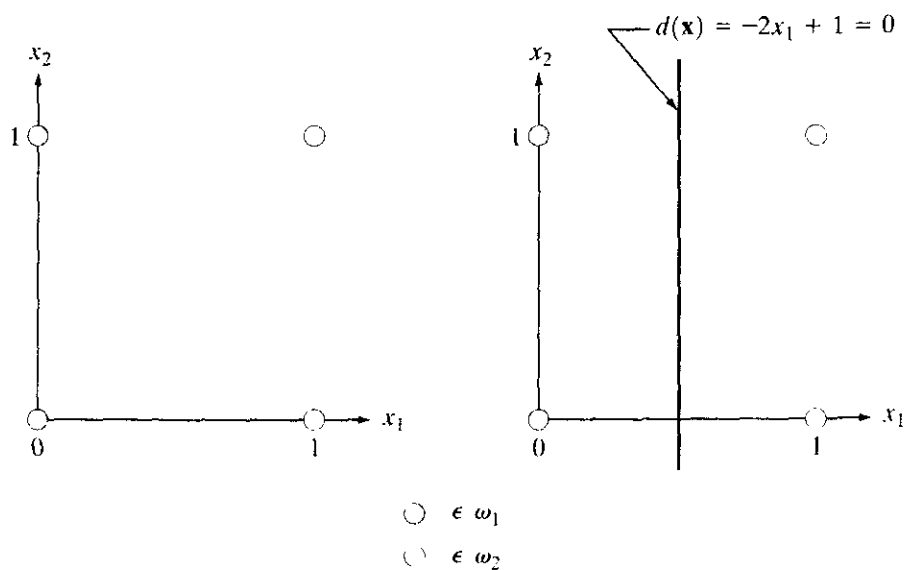
$$\mathbf{w}(k + 1) = \mathbf{w}(k). \tag{12.2-36}$$

This algorithm makes a change in  $\mathbf{w}$  only if the pattern being considered at the  $k$ th step in the training sequence is misclassified. The correction increment  $c$  is assumed to be positive and, for now, to be constant. This algorithm sometimes is referred to as the *fixed increment correction rule*.

Convergence of the algorithm occurs when the entire training set for both classes is cycled through the machine without any errors. The fixed increment correction rule converges in a finite number of steps if the two training sets of patterns are linearly separable. A proof of this result, sometimes called the *perceptron training theorem*, can be found in the books by Duda, Hart, and Stork [2001]; Tou and Gonzalez [1974]; and Nilsson [1965].

**EXAMPLE 12.5:** Illustration of the perceptron algorithm.

■ Consider the two training sets shown in Fig. 12.15(a), each consisting of two patterns. The training algorithm will be successful because the two training sets are linearly separable. Before the algorithm is applied the patterns are augmented.



**FIGURE 12.15**  
 (a) Patterns belonging to two classes.  
 (b) Decision boundary determined by training.

yielding the training set  $\{(0, 0, 1)^T, (0, 1, 1)^T\}$  for class  $\omega_1$  and  $\{(1, 0, 1)^T, (1, 1, 1)^T\}$  for class  $\omega_2$ . Letting  $c = 1$ ,  $\mathbf{w}(1) = \mathbf{0}$ , and presenting the patterns in order results in the following sequence of steps:

$$\begin{aligned} \mathbf{w}^T(1)\mathbf{y}(1) &= [0, 0, 0] \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} = 0 & \mathbf{w}(2) &= \mathbf{w}(1) + \mathbf{y}(1) = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \\ \mathbf{w}^T(2)\mathbf{y}(2) &= [0, 0, 1] \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix} = 1 & \mathbf{w}(3) &= \mathbf{w}(2) = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \\ \mathbf{w}^T(3)\mathbf{y}(3) &= [0, 0, 1] \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} = 1 & \mathbf{w}(4) &= \mathbf{w}(3) - \mathbf{y}(3) = \begin{bmatrix} -1 \\ 0 \\ 0 \end{bmatrix} \\ \mathbf{w}^T(4)\mathbf{y}(4) &= [-1, 0, 0] \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} = -1 & \mathbf{w}(5) &= \mathbf{w}(4) = \begin{bmatrix} -1 \\ 0 \\ 0 \end{bmatrix} \end{aligned}$$

where corrections in the weight vector were made in the first and third steps because of misclassifications, as indicated in Eqs. (12.2-34) and (12.2-35). Because a solution has been obtained only when the algorithm yields a complete error-free iteration through all training patterns, the training set must be presented again. The machine learning process is continued by letting  $\mathbf{y}(5) = \mathbf{y}(1)$ ,  $\mathbf{y}(6) = \mathbf{y}(2)$ ,  $\mathbf{y}(7) = \mathbf{y}(3)$ , and  $\mathbf{y}(8) = \mathbf{y}(4)$ , and proceeding in the same manner. Convergence is achieved at  $k = 14$ , yielding the solution weight vector  $\mathbf{w}(14) = (-2, 0, 1)^T$ . The corresponding decision function is  $d(\mathbf{y}) = -2y_1 + 1$ . Going back to the original pattern space by letting  $x_i = y_i$  yields  $d(\mathbf{x}) = -2x_1 + 1$ , which, when set equal to zero, becomes the equation of the decision boundary shown in Fig. 12.15(b).

**Nonseparable classes** In practice, linearly separable pattern classes are the (rare) exception, rather than the rule. Consequently, a significant amount of research effort during the 1960s and 1970s went into development of techniques designed to handle nonseparable pattern classes. With recent advances in the training of neural networks, many of the methods dealing with nonseparable behavior have become merely items of historical interest. One of the early methods, however, is directly relevant to this discussion: the original delta rule. Known as the *Widrow-Hoff*, or *least-mean-square (LMS) delta rule* for training perceptrons, the method minimizes the error between the actual and desired response at any training step.

Consider the criterion function

$$J(\mathbf{w}) = \frac{1}{2}(r - \mathbf{w}^T\mathbf{y})^2 \quad (12.2-37)$$

where  $r$  is the desired response (that is,  $r = +1$  if the augmented training pattern vector  $\mathbf{y}$  belongs to class  $\omega_1$ , and  $r = -1$  if  $\mathbf{y}$  belongs to class  $\omega_2$ ). The task

is to adjust  $\mathbf{w}$  incrementally in the direction of the negative gradient of  $J(\mathbf{w})$  in order to seek the minimum of this function, which occurs when  $r = \mathbf{w}^T \mathbf{y}$ ; that is, the minimum corresponds to correct classification. If  $\mathbf{w}(k)$  represents the weight vector at the  $k$ th iterative step, a general gradient descent algorithm may be written as

$$\mathbf{w}(k+1) = \mathbf{w}(k) - \alpha \left[ \frac{\partial J(\mathbf{w})}{\partial \mathbf{w}} \right]_{\mathbf{w}=\mathbf{w}(k)} \quad (12.2-38)$$

where  $\mathbf{w}(k+1)$  is the new value of  $\mathbf{w}$ , and  $\alpha > 0$  gives the magnitude of the correction. From Eq. (12.2-37),

$$\frac{\partial J(\mathbf{w})}{\partial \mathbf{w}} = -(r - \mathbf{w}^T \mathbf{y}) \mathbf{y}. \quad (12.2-39)$$

Substituting this result into Eq. (12.2-38) yields

$$\mathbf{w}(k+1) = \mathbf{w}(k) + \alpha [r(k) - \mathbf{w}^T(k) \mathbf{y}(k)] \mathbf{y}(k) \quad (12.2-40)$$

with the starting weight vector,  $\mathbf{w}(1)$ , being arbitrary.

By defining the change (delta) in weight vector as

$$\Delta \mathbf{w} = \mathbf{w}(k+1) - \mathbf{w}(k) \quad (12.2-41)$$

we can write Eq. (12.2-40) in the form of a *delta correction algorithm*:

$$\Delta \mathbf{w} = \alpha e(k) \mathbf{y}(k) \quad (12.2-42)$$

where

$$e(k) = r(k) - \mathbf{w}^T(k) \mathbf{y}(k) \quad (12.2-43)$$

is the error committed with weight vector  $\mathbf{w}(k)$  when pattern  $\mathbf{y}(k)$  is presented.

Equation (12.2-43) gives the error with weight vector  $\mathbf{w}(k)$ . If we change it to  $\mathbf{w}(k+1)$ , but leave the pattern the same, the error becomes

$$e(k) = r(k) - \mathbf{w}^T(k+1) \mathbf{y}(k). \quad (12.2-44)$$

The change in error then is

$$\begin{aligned} \Delta e(k) &= [r(k) - \mathbf{w}^T(k+1) \mathbf{y}(k)] - [r(k) - \mathbf{w}^T(k) \mathbf{y}(k)] \\ &= -[\mathbf{w}^T(k+1) - \mathbf{w}^T(k)] \mathbf{y}(k) \\ &= -\Delta \mathbf{w}^T \mathbf{y}(k). \end{aligned} \quad (12.2-45)$$

But  $\Delta \mathbf{w} = \alpha e(k) \mathbf{y}(k)$ , so

$$\begin{aligned} \Delta e &= -\alpha e(k) \mathbf{y}^T(k) \mathbf{y}(k) \\ &= -\alpha e(k) \|\mathbf{y}(k)\|^2. \end{aligned} \quad (12.2-46)$$

Hence changing the weights reduces the error by a factor  $\alpha \|\mathbf{y}(k)\|^2$ . The next input pattern starts the new adaptation cycle, reducing the next error by a factor  $\alpha \|\mathbf{y}(k+1)\|^2$ , and so on.

The choice of  $\alpha$  controls stability and speed of convergence (Widrow and Stearns [1985]). Stability requires that  $0 < \alpha < 2$ . A practical range for  $\alpha$  is  $0.1 < \alpha < 1.0$ . Although the proof is not shown here, the algorithm of

Eq. (12.2-40) or Eqs. (12.2-42) and (12.2-43) converges to a solution that minimizes the mean square error over the patterns of the training set. When the pattern classes are separable, the solution given by the algorithm just discussed may or may not produce a separating hyperplane. That is, a mean-square-error solution does not imply a solution in the sense of the perceptron training theorem. This uncertainty is the price of using an algorithm that converges under both the separable and nonseparable cases in this particular formulation.

The two perceptron training algorithms discussed thus far can be extended to more than two classes and to nonlinear decision functions. Based on the historical comments made earlier, exploring multiclass training algorithms here has little merit. Instead, we address multiclass training in the context of neural networks.

### Multilayer feedforward neural networks

In this section we focus on decision functions of multiclass pattern recognition problems, independent of whether or not the classes are separable, and involving architectures that consist of layers of perceptron computing elements.

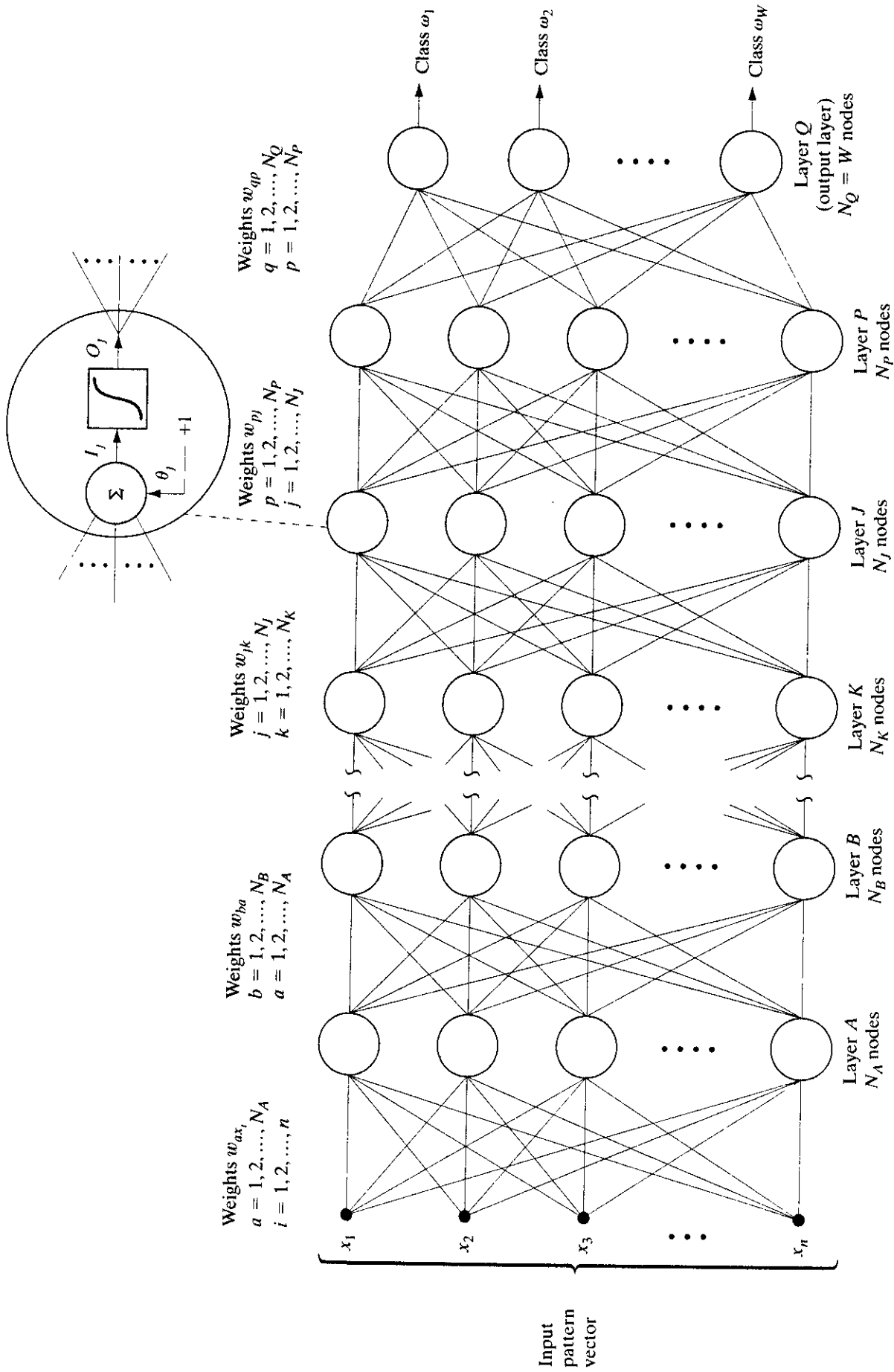
**Basic architecture** Figure 12.16 shows the architecture of the neural network model under consideration. It consists of layers of structurally identical computing nodes (neurons) arranged so that the output of every neuron in one layer feeds into the input of every neuron in the next layer. The number of neurons in the first layer, called layer  $A$ , is  $N_A$ . Often,  $N_A = n$ , the dimensionality of the input pattern vectors. The number of neurons in the output layer, called layer  $Q$ , is denoted  $N_Q$ . The number  $N_Q$  equals  $W$ , the number of pattern classes that the neural network has been trained to recognize. The network recognizes a pattern vector  $\mathbf{x}$  as belonging to class  $\omega_i$  if the  $i$ th output of the network is “high” while all other outputs are “low,” as explained in the following discussion.

As the blowup in Fig. 12.16 shows, each neuron has the same form as the perceptron model discussed earlier (see Fig. 12.14), with the exception that the hard-limiting activation function has been replaced by a soft-limiting “sigmoid” function. Differentiability along all paths of the neural network is required in the development of the training rule. The following sigmoid activation function has the necessary differentiability:

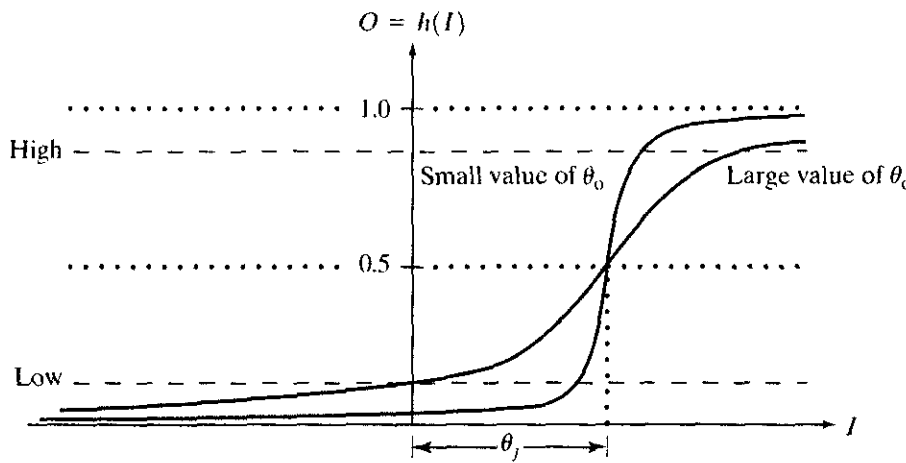
$$h_j(I_j) = \frac{1}{1 + e^{-(I_j + \theta_j)/\theta_o}} \quad (12.2-47)$$

where  $I_j$ ,  $j = 1, 2, \dots, N_j$ , is the input to the activation element of each node in layer  $J$  of the network,  $\theta_j$  is an offset, and  $\theta_o$  controls the shape of the sigmoid function.

Equation (12.2-47) is plotted in Fig. 12.17, along with the limits for the “high” and “low” responses out of each node. Thus when this particular function is used, the system outputs a high reading for any value of  $I_j$  greater than  $\theta_j$ . Similarly, the system outputs a low reading for any value of  $I_j$  less than  $\theta_j$ . As Fig. 12.17 shows, the sigmoid activation function always is positive, and it can reach its limiting values of 0 and 1 only if the input to the activation element is infinitely negative or positive, respectively. For this reason, values near 0 and 1



**FIGURE 12.16** Multilayer feedforward neural network model. The blowup shows the basic structure of each neuron element throughout the network. The offset,  $\theta_j$ , is treated as just another weight.



**FIGURE 12.17** The sigmoidal activation function of Eq. (12.2-47).

(say, 0.05 and 0.95) define low and high values at the output of the neurons in Fig. 12.16. In principle, different types of activation functions could be used for different layers or even for different nodes in the same layer of a neural network. In practice, the usual approach is to use the same form of activation function throughout the network.

With reference to Fig. 12.14(a), the offset  $\theta_j$  shown in Fig. 12.17 is analogous to the weight coefficient  $w_{n+1}$  in the earlier discussion of the perceptron. Implementation of this displaced threshold function can be done in the form of Fig. 12.14(a) by absorbing the offset  $\theta_j$  as an additional coefficient that modifies a constant unity input to all nodes in the network. In order to follow the notation predominantly found in the literature, we do not show a separate constant input of +1 into all nodes of Fig. 12.16. Instead, this input and its modifying weight  $\theta_j$  are integral parts of the network nodes. As noted in the blowup in Fig. 12.16, there is one such coefficient for each of the  $N_j$  nodes in layer  $J$ .

In Fig. 12.16, the input to a node in any layer is the weighted sum of the outputs from the previous layer. Letting layer  $K$  denote the layer preceding layer  $J$  (no alphabetical order is implied in Fig. 12.16) gives the input to the activation element of each node in layer  $J$ , denoted  $I_j$ :

$$I_j = \sum_{k=1}^{N_k} w_{jk} O_k \tag{12.2-48}$$

for  $j = 1, 2, \dots, N_j$ , where  $N_j$  is the number of nodes in layer  $J$ ,  $N_k$  is the number of nodes in layer  $K$ , and  $w_{jk}$  are the weights modifying the outputs  $O_k$  of the nodes in layer  $K$  before they are fed into the nodes in layer  $J$ . The outputs of layer  $K$  are

$$O_k = h_k(I_k) \tag{12.2-49}$$

for  $k = 1, 2, \dots, N_k$ .

A clear understanding of the subscript notation used in Eq. (12.2-48) is important, because we use it throughout the remainder of this section. First, note that  $I_j, j = 1, 2, \dots, N_j$ , represents the input to the *activation element* of the  $j$ th node in layer  $J$ . Thus  $I_1$  represents the input to the activation element of the



first (topmost) node in layer  $J$ ,  $I_2$  represents the input to the activation element of the second node in layer  $J$ , and so on. There are  $N_K$  inputs to every node in layer  $J$ , but *each* individual input can be weighted differently. Thus the  $N_K$  inputs to the first node in layer  $J$  are weighted by coefficients  $w_{1k}$ ,  $k = 1, 2, \dots, N_K$ ; the inputs to the second node are weighted by coefficients  $w_{2k}$ ,  $k = 1, 2, \dots, N_K$ ; and so on. Hence a total of  $N_J \times N_K$  coefficients are necessary to specify the weighting of the outputs of layer  $K$  as they are fed into layer  $J$ . An additional  $N_J$  offset coefficients,  $\theta_j$ , are needed to specify completely the nodes in layer  $J$ .

Substitution of Eq. (12.2-48) into (12.2-47) yields

$$h_j(I_j) = \frac{1}{1 + e^{-\left(\sum_{k=1}^{N_K} w_{jk} O_k + \theta_j\right)/\theta_0}}, \quad (12.2-50)$$

which is the form of activation function used in the remainder of this section.

During training, adapting the neurons in the output layer is a simple matter because the desired output of each node is known. The main problem in training a multilayer network lies in adjusting the weights in the so-called *hidden layers*. That is, in those other than the output layer.

**Training by back propagation** We begin by concentrating on the output layer. The total squared error between the desired responses,  $r_q$ , and the corresponding actual responses,  $O_q$ , of nodes in (output) layer  $Q$ , is

$$E_Q = \frac{1}{2} \sum_{q=1}^{N_Q} (r_q - O_q)^2 \quad (12.2-51)$$

where  $N_Q$  is the number of nodes in output layer  $Q$  and the  $\frac{1}{2}$  is used for convenience in notation for taking the derivative later.

The objective is to develop a training rule, similar to the delta rule, that allows adjustment of the weights in each of the layers in a way that seeks a minimum to an error function of the form shown in Eq. (12.2-51). As before, adjusting the weights in proportion to the partial derivative of the error with respect to the weights achieves this result. In other words,

$$\Delta w_{qp} = -\alpha \frac{\partial E_Q}{\partial w_{qp}} \quad (12.2-52)$$

where layer  $P$  precedes layer  $Q$ ,  $\Delta w_{qp}$  is as defined in Eq. (12.2-42), and  $\alpha$  is a positive correction increment.

The error  $E_Q$  is a function of the outputs,  $O_q$ , which in turn are functions of the inputs  $I_q$ . Using the chain rule, we evaluate the partial derivative of  $E_Q$  as follows:

$$\frac{\partial E_Q}{\partial w_{qp}} = \frac{\partial E_Q}{\partial I_q} \frac{\partial I_q}{\partial w_{qp}}. \quad (12.2-53)$$

From Eq. (12.2-48),

$$\frac{\partial I_q}{\partial w_{qp}} = \frac{\partial}{\partial w_{qp}} \sum_{p=1}^{N_P} w_{qp} O_p = O_p. \quad (12.2-54)$$

Substituting Eqs. (12.2-53) and (12.2-54) into Eq. (12.2-52) yields

$$\begin{aligned}\Delta w_{qp} &= -\alpha \frac{\partial E_Q}{\partial I_q} O_p \\ &= \alpha \delta_q O_p\end{aligned}\quad (12.2-55)$$

where

$$\delta_q = -\frac{\partial E_Q}{\partial I_q}.\quad (12.2-56)$$

In order to compute  $\partial E_Q/\partial I_q$ , we use the chain rule to express the partial derivative in terms of the rate of change of  $E_Q$  with respect to  $O_q$  and the rate of change of  $O_q$  with respect to  $I_q$ . That is,

$$\delta_q = -\frac{\partial E_Q}{\partial I_q} = -\frac{\partial E_Q}{\partial O_q} \frac{\partial O_q}{\partial I_q}.\quad (12.2-57)$$

From Eq. (12.2-51),

$$\frac{\partial E_Q}{\partial O_q} = -(r_q - O_q)\quad (12.2-58)$$

and, from Eq. (12.2-49),

$$\frac{\partial O_q}{\partial I_q} = \frac{\partial}{\partial I_q} h_q(I_q) = h'_q(I_q).\quad (12.2-59)$$

Substituting Eqs. (12.2-58) and (12.2-59) into Eq. (12.2-57) gives

$$\delta_q = (r_q - O_q)h'_q(I_q),\quad (12.2-60)$$

which is proportional to the error quantity  $(r_q - O_q)$ . Substitution of Eqs. (12.2-56) through (12.2-58) into Eq. (12.2-55) finally yields

$$\begin{aligned}\Delta w_{qp} &= \alpha(r_q - O_q)h'_q(I_q)O_p \\ &= \alpha\delta_q O_p.\end{aligned}\quad (12.2-61)$$

After the function  $h_q(I_q)$  has been specified, all the terms in Eq. (12.2-61) are known or can be observed in the network. In other words, upon presentation of any training pattern to the input of the network, we know what the desired response,  $r_q$ , of each output node should be. The value  $O_q$  of each output node can be observed as can  $I_q$ , the input to the activation elements of layer  $Q$ , and  $O_p$ , the output of the nodes in layer  $P$ . Thus we know how to adjust the weights that modify the links between the last and next-to-last layers in the network.

Continuing to work our way back from the output layer, let us now analyze what happens at layer  $P$ . Proceeding in the same manner as above yields

$$\begin{aligned}\Delta w_{pj} &= \alpha(r_p - O_p)h'_p(I_p)O_j \\ &= \alpha\delta_p O_j\end{aligned}\quad (12.2-62)$$

where the error term is

$$\delta_p = (r_p - O_p)h'_p(I_p).\quad (12.2-63)$$

With the exception of  $r_p$ , all the terms in Eqs. (12.2-62) and (12.2-63) either are known or can be observed in the network. The term  $r_p$  makes no sense in an internal layer because we do not know what the response of an internal node in terms of pattern membership should be. We may specify what we want the response  $r$  to be only at the outputs of the network where final pattern classification takes place. If we knew that information at internal nodes, there would be no need for further layers. Thus we have to find a way to restate  $\delta_p$  in terms of quantities that are known or can be observed in the network.

Going back to Eq. (12.2-57), we write the error term for layer  $P$  as

$$\delta_p = -\frac{\partial E_p}{\partial I_p} = -\frac{\partial E_p}{\partial O_p} \frac{\partial O_p}{\partial I_p}. \quad (12.2-64)$$

The term  $\partial O_p / \partial I_p$  presents no difficulties. As before, it is

$$\frac{\partial O_p}{\partial I_p} = \frac{\partial h_p(I_p)}{\partial I_p} = h'_p(I_p), \quad (12.2-65)$$

which is known once  $h_p$  is specified because  $I_p$  can be observed. The term that produced  $r_p$  was the derivative  $\partial E_p / \partial O_p$ , so this term must be expressed in a way that does not contain  $r_p$ . Using the chain rule, we write the derivative as

$$\begin{aligned} -\frac{\partial E_p}{\partial O_p} &= -\sum_{q=1}^{N_Q} \frac{\partial E_p}{\partial I_q} \frac{\partial I_q}{\partial O_p} = \sum_{q=1}^{N_Q} \left( -\frac{\partial E_p}{\partial I_q} \right) \frac{\partial}{\partial O_p} \sum_{p=1}^{N_P} w_{qp} O_p \\ &= \sum_{q=1}^{N_Q} \left( -\frac{\partial E_p}{\partial I_q} \right) w_{qp} \\ &= \sum_{q=1}^{N_Q} \delta_q w_{qp} \end{aligned} \quad (12.2-66)$$

where the last step follows from Eq. (12.2-56). Substituting Eqs. (12.2-65) and (12.2-66) into Eq. (12.2-64) yields the desired expression for  $\delta_p$ :

$$\delta_p = h'_p(I_p) \sum_{q=1}^{N_Q} \delta_q w_{qp}. \quad (12.2-67)$$

The parameter  $\delta_p$  can be computed now because all its terms are known. Thus Eqs. (12.2-62) and (12.2-67) establish completely the training rule for layer  $P$ . The importance of Eq. (12.2-67) is that it computes  $\delta_p$  from the quantities  $\delta_q$  and  $w_{qp}$ , which are terms that were computed in the layer immediately following layer  $P$ . After the error term and weights have been computed for layer  $P$ , these quantities may be used similarly to compute the error and weights for the layer immediately preceding layer  $P$ . In other words, we have found a way to propagate the error back into the network, starting with the error at the output layer.

We may summarize and generalize the training procedure as follows. For any layers  $K$  and  $J$ , where layer  $K$  immediately precedes layer  $J$ , compute the weights  $w_{jk}$ , which modify the connections between these two layers, by using

$$\Delta w_{jk} = \alpha \delta_j O_k. \quad (12.2-68)$$

If layer  $J$  is the output layer,  $\delta_j$  is

$$\delta_j = (r_j - O_j)h'_j(I_j). \quad (12.2-69)$$

If layer  $J$  is an internal layer and layer  $P$  is the next layer (to the right), then  $\delta_j$  is given by

$$\delta_j = h'_j(I_j) \sum_{p=1}^{N_p} \delta_p w_{jp} \quad (12.2-70)$$

for  $j = 1, 2, \dots, N_j$ . Using the activation function in Eq. (12.2-50) with  $\theta_o = 1$  yields

$$h'_j(I_j) = O_j(1 - O_j) \quad (12.2-71)$$

in which case Eqs. (12.2-69) and (12.2-70) assume the following, particularly attractive forms:

$$\delta_j = (r_j - O_j)O_j(1 - O_j) \quad (12.2-72)$$

for the output layer, and

$$\delta_j = O_j(1 - O_j) \sum_{p=1}^{N_p} \delta_p w_{jp} \quad (12.2-73)$$

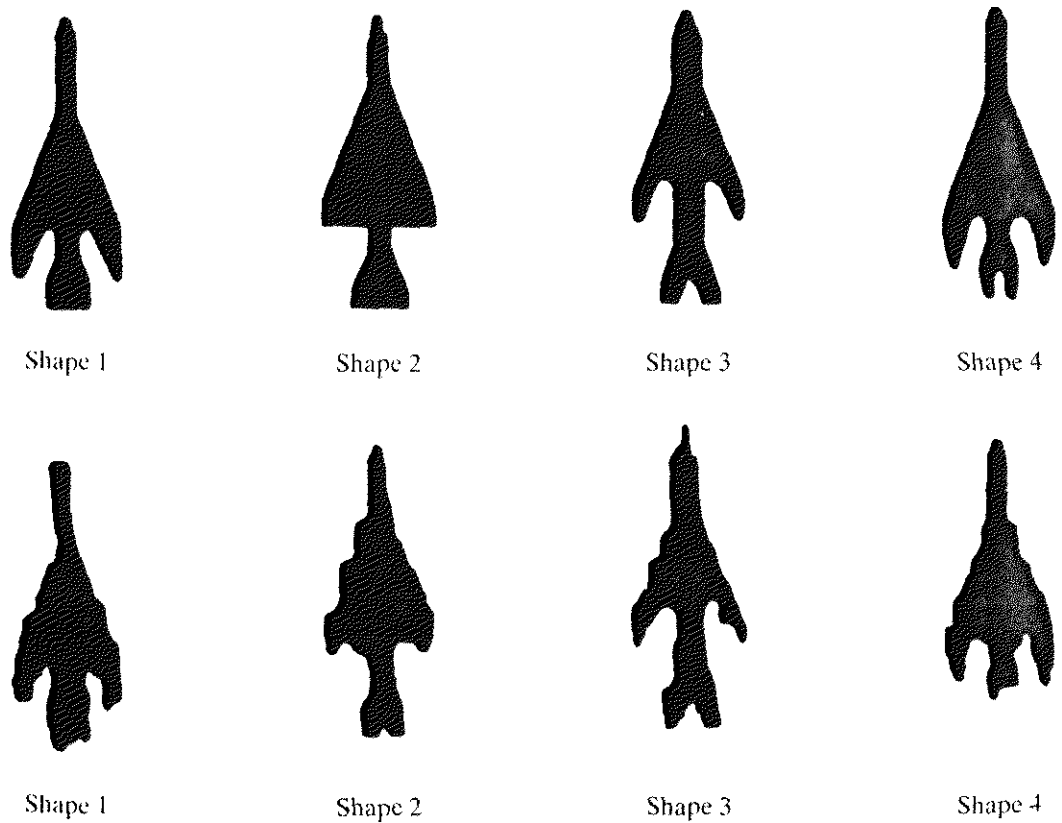
for internal layers. In both Eqs. (12.2-72) and (12.2-73),  $j = 1, 2, \dots, N_j$ .

Equations (12.2-68) through (12.2-70) constitute the generalized delta rule for training the multilayer feedforward neural network of Fig. 12.16. The process starts with an arbitrary (but not all equal) set of weights throughout the network. Then application of the generalized delta rule at any iterative step involves two basic phases. In the first phase, a training vector is presented to the network and is allowed to propagate through the layers to compute the output  $O_j$  for each node. The outputs  $O_q$  of the nodes in the output layer are then compared against their desired responses,  $r_p$ , to generate the error terms  $\delta_q$ . The second phase involves a backward pass through the network during which the appropriate error signal is passed to each node and the corresponding weight changes are made. This procedure also applies to the bias weights  $\theta_j$ . As discussed earlier in some detail, these are treated simply as additional weights that modify a unit input into the summing junction of every node in the network.

Common practice is to track the network error, as well as errors associated with individual patterns. In a successful training session, the network error decreases with the number of iterations and the procedure converges to a stable set of weights that exhibit only small fluctuations with additional training. The approach followed to establish whether a pattern has been classified correctly during training is to determine whether the response of the node in the output layer associated with the pattern class from which the pattern was obtained is high, while all the other nodes have outputs that are low, as defined earlier.

After the system has been trained, it classifies patterns using the parameters established during the training phase. In normal operation, all feedback paths are disconnected. Then any input pattern is allowed to propagate through the various layers, and the pattern is classified as belonging to the class of the output node that was high, while all the others were low. If more than one output is labeled high, or if none of the outputs is so labeled, the choice is one of declaring a misclassification or simply assigning the pattern to the class of the output node with the highest numerical value.

a  
b  
**FIGURE 12.18**  
(a) Reference shapes and (b) typical noisy shapes used in training the neural network of Fig. 12.19. (Courtesy of Dr. Lalit Gupta, ECE Department, Southern Illinois University.)

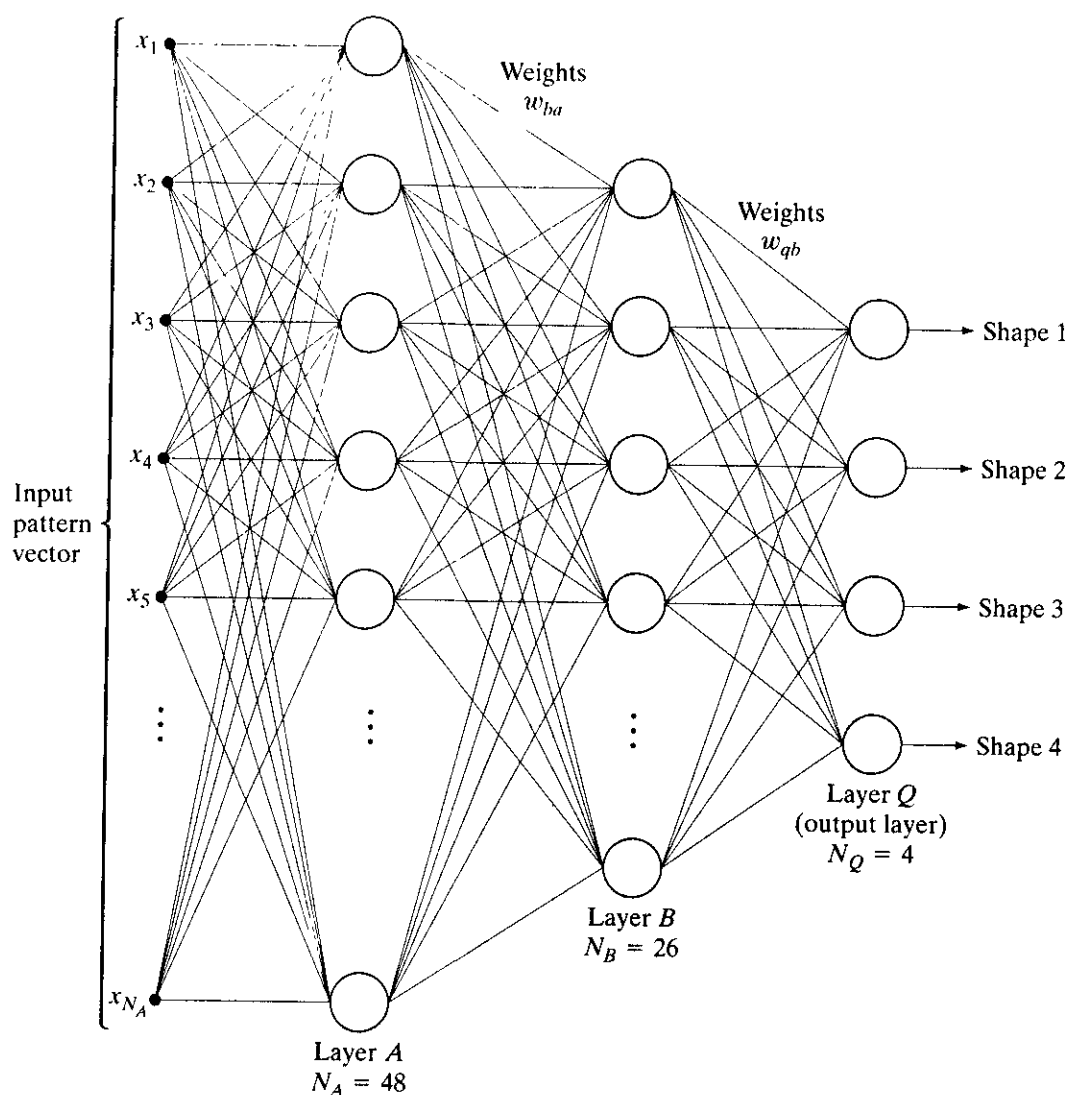


**EXAMPLE 12.6:**  
Shape classification using a neural network.

We illustrate now how a neural network of the form shown in Fig. 12.16 was trained to recognize the four shapes shown in Fig. 12.18(a), as well as noisy versions of these shapes, samples of which are shown in Fig. 12.18(b).

Pattern vectors were generated by computing the normalized signatures of the shapes (see Section 11.1.3) and then obtaining 48 uniformly spaced samples of each signature. The resulting 48-dimensional vectors were the inputs to the three-layer feedforward neural network shown in Fig. 12.19. The number of neuron nodes in the first layer was chosen to be 48, corresponding to the dimensionality of the input pattern vectors. The four neurons in the third (output) layer correspond to the number of pattern classes, and the number of neurons in the middle layer was heuristically specified as 26 (the average of the number of neurons in the input and output layers). There are no known rules for specifying the number of nodes in the internal layers of a neural network, so this number generally is based either on prior experience or simply chosen arbitrarily and then refined by testing. In the output layer, the four nodes from top to bottom in this case represent the classes  $\omega_j, j = 1, 2, 3, 4$ , respectively. After the network structure has been set, activation functions have to be selected for each unit and layer. All activation functions were selected to satisfy Eq. (12.2-50) with  $\theta_0 = 1$  so that, according to our earlier discussion, Eqs. (12.2-72) and (12.2-73) apply.

The training process was divided in two parts. In the first part, the weights were initialized to small random values with zero mean, and the network was then trained with pattern vectors corresponding to noise-free samples like the

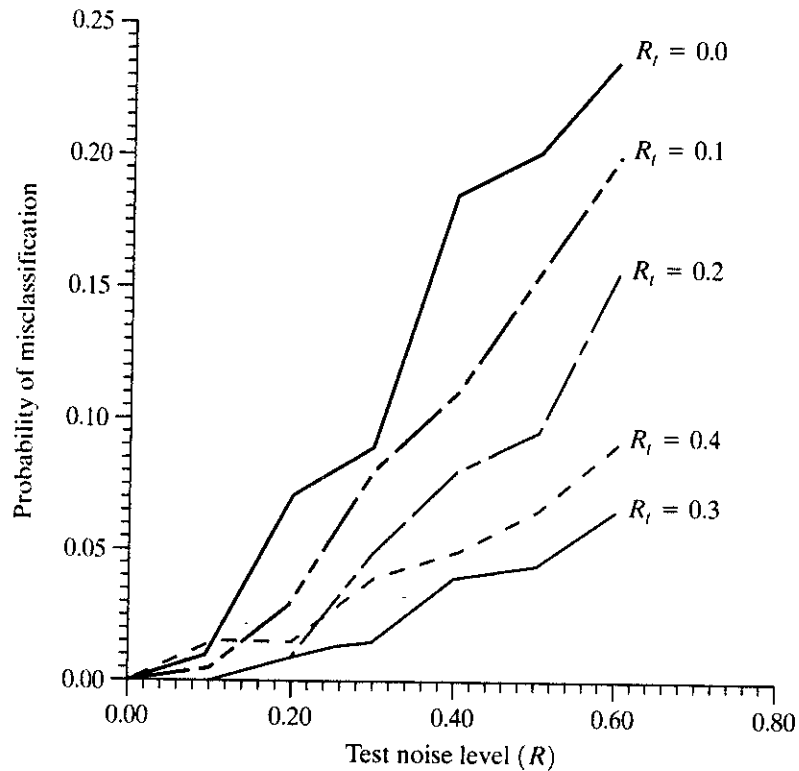


**FIGURE 12.19**  
Three-layer  
neural network  
used to recognize  
the shapes in  
Fig. 12.18.  
(Courtesy of Dr.  
Lalit Gupta, ECE  
Department,  
Southern Illinois  
University.)

shapes shown in Fig. 12.18(a). The output nodes were monitored during training. The network was said to have learned the shapes from all four classes when, for any training pattern from class  $\omega_i$ , the elements of the output layer yielded  $O_i \geq 0.95$  and  $O_q \leq 0.05$ , for  $q = 1, 2, \dots, N_Q; q \neq i$ . In other words, for any pattern of class  $\omega_i$ , the output unit corresponding to that class had to be high ( $\geq 0.95$ ) while, simultaneously, the output of all other nodes had to be low ( $\leq 0.05$ ).

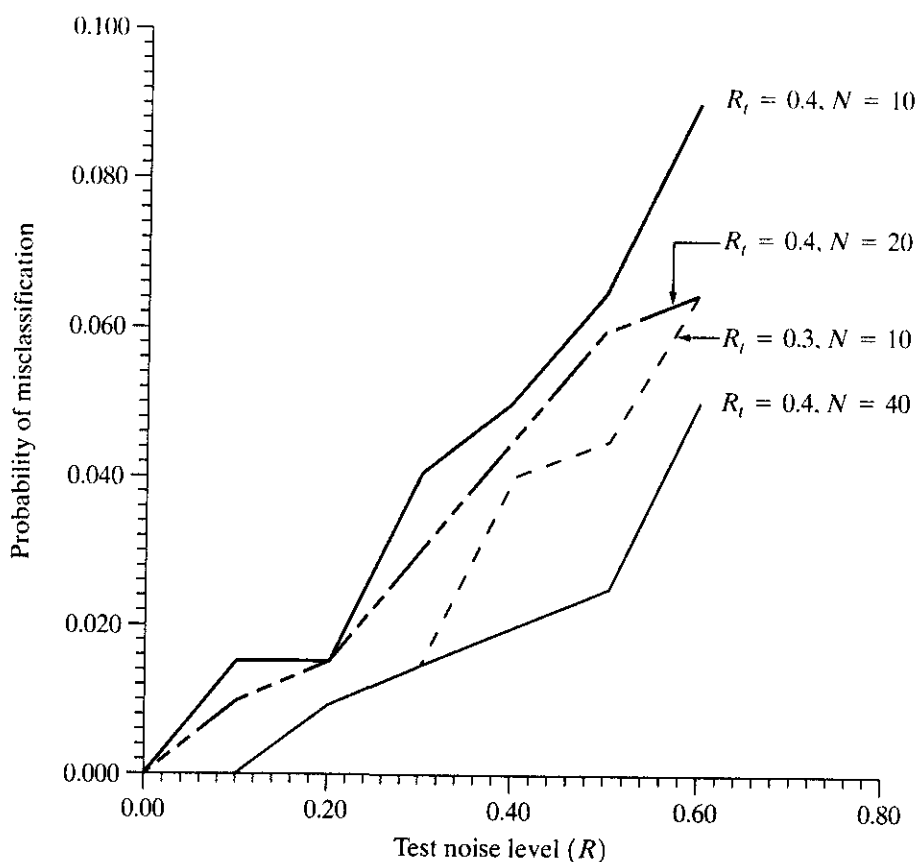
The second part of training was carried out with noisy samples, generated as follows. Each contour pixel in a noise-free shape was assigned a probability  $V$  of retaining its original coordinate in the image plane and a probability  $R = 1 - V$  of being randomly assigned to the coordinates of one of its eight neighboring pixels. The degree of noise was increased by decreasing  $V$  (that is, increasing  $R$ ). Two sets of noisy data were generated. The first consisted of 100 noisy patterns of each class generated by varying  $R$  between 0.1 and 0.6, giving a total of 400 patterns. This set, called the *test set*, was used to establish system a performance after training.

**FIGURE 12.20**  
Performance of the neural network as a function of noise level. (Courtesy of Dr. Lalit Gupta, ECE Department, Southern Illinois University.)



Several noisy sets were generated for training the system with noisy data. The first set consisted of 10 samples for each class, generated by using  $R_t = 0$ , where  $R_t$  denotes a value of  $R$  used to generate training data. Starting with the weight vectors obtained in the first (noise-free) part of training, the system was allowed to go through a learning sequence with the new data set. Because  $R_t = 0$  implies no noise, this retraining was an extension of the earlier, noise-free training. Using the resulting weights learned in this manner, the network was subjected to the test data set yielding the results shown by the curve labeled  $R_t = 0$  in Fig. 12.20. The number of misclassified patterns divided by the total number of patterns tested gives the probability of misclassification, which is a measure commonly used to establish neural network performance.

Next, starting with the weight vectors learned by using the data generated with  $R_t = 0$ , the system was retrained with a noisy data set generated with  $R_t = 0.1$ . The recognition performance was then established by running the test samples through the system again with the new weight vectors. Note the significant improvement in performance. Figure 12.20 shows the results obtained by continuing this retraining and retesting procedure for  $R_t = 0.2, 0.3$ , and  $0.4$ . As expected if the system is learning properly, the probability of misclassifying patterns from the test set decreased as the value of  $R_t$  increased because the system was being trained with noisier data for higher values of  $R_t$ . The one exception in Fig. 12.20 is the result for  $R_t = 0.4$ . The reason is the small number of samples used to train the system. That is, the network was not able to adapt itself sufficiently to the larger variations in shape at higher noise levels with the number of samples used. This hypothesis is verified by the results in Fig. 12.21, which show a lower probability of misclassification as the number of training



**FIGURE 12.21** Improvement in performance for  $R_t = 0.4$  by increasing the number of training patterns (the curve for  $R_t = 0.3$  is shown for reference). (Courtesy of Dr. Lalit Gupta, ECE Department, Southern Illinois University.)

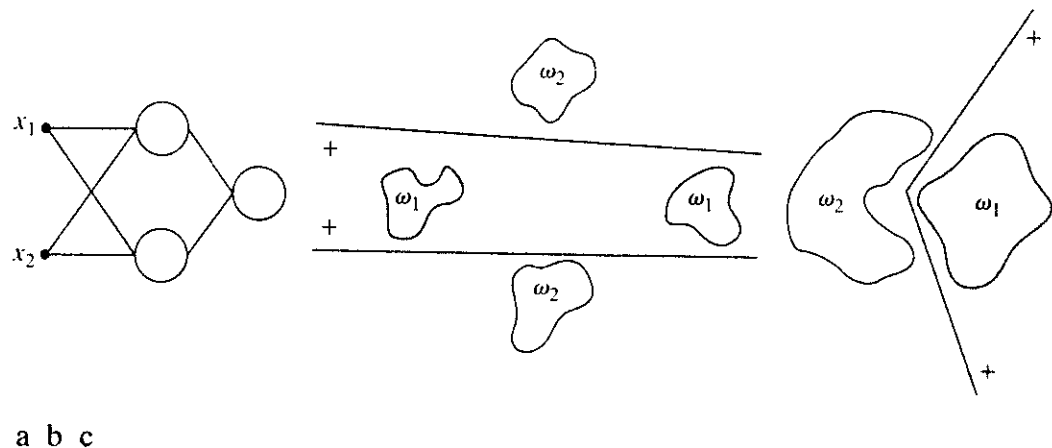
samples was increased. Figure 12.21 also shows as a reference the curve for  $R_t = 0.3$  from Fig. 12.20.

The preceding results show that a three-layer neural network was capable of learning to recognize shapes corrupted by noise after a modest level of training. Even when trained with noise-free data ( $R_t = 0$  in Fig. 12.20), the system was able to achieve a correct recognition level of close to 77% when tested with data highly corrupted by noise ( $R = 0.6$  in Fig. 12.20). The recognition rate on the same data increased to about 99% when the system was trained with noisier data ( $R_t = 0.3$  and 0.4). It is important to note that the system was trained by increasing its classification power via systematic, small incremental additions of noise. When the nature of the noise is known, this method is ideal for improving the convergence and stability properties of a neural network during learning.

**Complexity of decision surfaces** We have already established that a single-layer perceptron implements a hyperplane decision surface. A natural question at this point is, What is the nature of the decision surfaces implemented by a multilayer network, such as the model in Fig. 12.16? It is demonstrated in the following discussion that a three-layer network is capable of implementing arbitrarily complex decision surfaces composed of intersecting hyperplanes.

As a starting point, consider the two-input, two-layer network shown in Fig. 12.22(a). With two inputs, the patterns are two dimensional, and therefore, each node in the first layer of the network implements a line in 2-D space. We



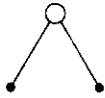
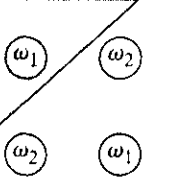
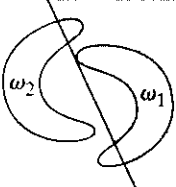
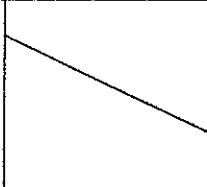
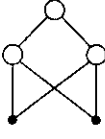
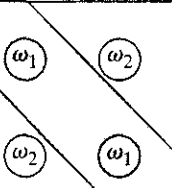
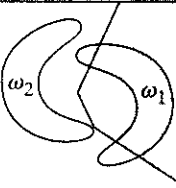
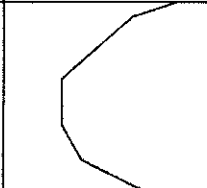
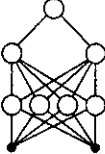
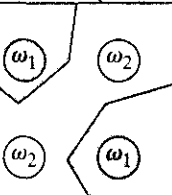
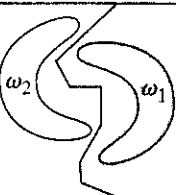
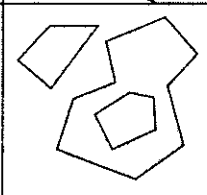


**FIGURE 12.22** (a) A two-input, two-layer, feedforward neural network. (b) and (c) Examples of decision boundaries that can be implemented with this network.

denote by 1 and 0, respectively, the high and low outputs of these two nodes. We assume that a 1 output indicates that the corresponding input vector to a node in the first layer lies on the positive side of the line. Then the possible combinations of outputs feeding the single node in the second layer are (1, 1), (1, 0), (0, 1), and (0, 0). If we define two regions, one for class  $\omega_1$  lying on the positive side of both lines and the other for class  $\omega_2$  lying anywhere else, the output node can classify any input pattern as belonging to one of these two regions simply by performing a logical AND operation. In other words, the output node responds with a 1, indicating class  $\omega_1$ , only when both outputs from the first layer are 1. The AND operation can be performed by a neural node of the form discussed earlier if  $\theta_j$  is set to a value in the half open interval  $(1, 2]$ . Thus if we assume 0 and 1 responses out of the first layer, the response of the output node will be high, indicating class  $\omega_1$ , only when the sum performed by the neural node on the two outputs from the first layer is greater than 1. Figures 12.22(b) and (c) show how the network of Fig. 12.22(a) can successfully dichotomize two pattern classes that could not be separated by a single linear surface.

If the number of nodes in the first layer were increased to three, the network of Fig. 12.22(a) would implement a decision boundary consisting of the intersection of three lines. The requirement that class  $\omega_1$  lie on the positive side of all three lines would yield a convex region bounded by the three lines. In fact, an arbitrary open or closed convex region can be constructed simply by increasing the number of nodes in the first layer of a two-layer neural network.

The next logical step is to increase the number of layers to three. In this case the nodes of the first layer implement lines, as before. The nodes of the second layer then perform AND operations in order to form regions from the various lines. The nodes in the third layer assign class membership to the various regions. For instance, suppose that class  $\omega_1$  consists of two distinct regions, each of which is bounded by a different set of lines. Then two of the nodes in the second layer are for regions corresponding to the same pattern class. One of the output nodes

Network structure	Type of decision region	Solution to exclusive-OR problem	Classes with meshed regions	Most general decision surface shapes
Single layer 	Single hyperplane			
Two layers 	Open or closed convex regions			
Three layers 	Arbitrary (complexity limited by the number of nodes)			

**FIGURE 12.23** Types of decision regions that can be formed by single- and multilayer feed-forward networks with one and two layers of hidden units and two inputs. (Lippman)

needs to be able to signal the presence of that class when either of the two nodes in the second layer goes high. Assuming that high and low conditions in the second layer are denoted 1 and 0, respectively, this capability is obtained by making the output nodes of the network perform the logical OR operation. In terms of neural nodes of the form discussed earlier, we do so by setting  $\theta_j$  to a value in the half-open interval  $[0, 1)$ . Then, whenever at least one of the nodes in the second layer associated with that output node goes high (outputs a 1), the corresponding node in the output layer will go high, indicating that the pattern being processed belongs to the class associated with that node.

Figure 12.23 summarizes the preceding comments. Note in the third row that the complexity of decision regions implemented by a three-layer network is, in principle, arbitrary. In practice, a serious difficulty usually arises in structuring the second layer to respond correctly to the various combinations associated with particular classes. The reason is that lines do not just stop at their intersection with other lines, and, as a result, patterns of the same class may occur on both sides of lines in the pattern space. In practical terms, the second layer may have difficulty figuring out which lines should be included in the AND operation for a given pattern class—or it may even be impossible. The reference to the exclusive-OR problem in the third column of Fig. 12.23 deals with the fact that, if the input patterns were binary, only four different patterns could be constructed in two dimensions. If the patterns are so arranged that class  $\omega_1$  consists of patterns  $\{(0, 1), (1, 0)\}$  and class  $\omega_2$  consists of the patterns  $\{(0, 0), (1, 1)\}$ , class membership of the patterns in these two classes is given by the exclusive-OR (XOR) logical function, which is 1 only when one or the other of the two variables is 1, and it is 0 otherwise. Thus an XOR value of 1 indicates patterns of class  $\omega_1$ , and an XOR value of 0 indicates patterns of class  $\omega_2$ .

The preceding discussion is generalized to  $n$  dimensions in a straightforward way: Instead of lines, we deal with hyperplanes. A single-layer network implements a single hyperplane. A two-layer network implements arbitrarily convex regions consisting of intersections of hyperplanes. A three-layer network implements decision surfaces of arbitrary complexity. The number of nodes used in each layer determines the complexity of the last two cases. The number of classes in the first case is limited to two. In the other two cases, the number of classes is arbitrary, because the number of output nodes can be selected to fit the problem at hand.

Considering the preceding comments, it is logical to ask, Why would anyone be interested in studying neural networks having more than three layers? After all, a three-layer network can implement decision surfaces of arbitrary complexity. The answer lies in the method used to train a network to utilize only three layers. The training rule for the network in Fig. 12.16 minimizes an error measure but says nothing about how to associate groups of hyperplanes with specific nodes in the second layer of a three-layer network of the type discussed earlier. In fact, the problem of how to perform trade-off analyses between the number of layers and the number of nodes in each layer remains unresolved. In practice, the trade-off is generally resolved by trial and error or by previous experience with a given problem domain.

### 12.3 Structural Methods

The techniques discussed in Section 12.2 deal with patterns quantitatively and largely ignore any structural relationships inherent in a pattern's shape. The structural methods discussed in this section, however, seek to achieve pattern recognition by capitalizing precisely on these types of relationships.

#### 12.3.1 Matching Shape Numbers

A procedure analogous to the minimum distance concept introduced in Section 12.2.1 for pattern vectors can be formulated for the comparison of region boundaries that are described in terms of shape numbers. With reference to the discussion in Section 11.2.2, the *degree of similarity*,  $k$ , between two region boundaries (shapes) is defined as the largest order for which their shape numbers still coincide. For example, let  $a$  and  $b$  denote shape numbers of closed boundaries represented by 4-directional chain codes. These two shapes have a degree of similarity  $k$  if

$$\begin{aligned} s_j(a) &= s_j(b) && \text{for } j = 4, 6, 8, \dots, k \\ s_j(a) &\neq s_j(b) && \text{for } j = k + 2, k + 4, \dots \end{aligned} \quad (12.3-1)$$

where  $s$  indicates shape number and the subscript indicates order. The *distance* between two shapes  $a$  and  $b$  is defined as the inverse of their degree of similarity:

$$D(a, b) = \frac{1}{k}, \quad (12.3-2)$$

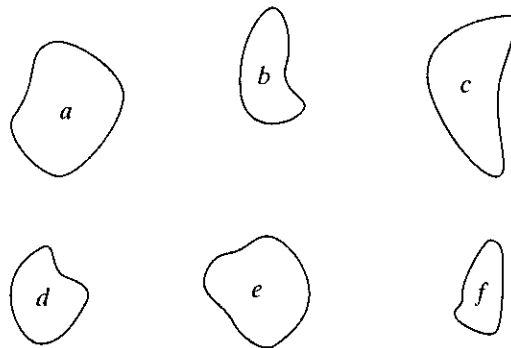
This distance satisfies the following properties:

$$\begin{aligned}
 D(a, b) &\geq 0 \\
 D(a, b) &= 0 \quad \text{iff } a = b \\
 D(a, c) &\leq \max[D(a, b), D(b, c)].
 \end{aligned}
 \tag{12.3-3}$$

Either  $k$  or  $D$  may be used to compare two shapes. If the degree of similarity is used, the larger  $k$  is, the more similar the shapes are (note that  $k$  is infinite for identical shapes). The reverse is true when the distance measure is used.

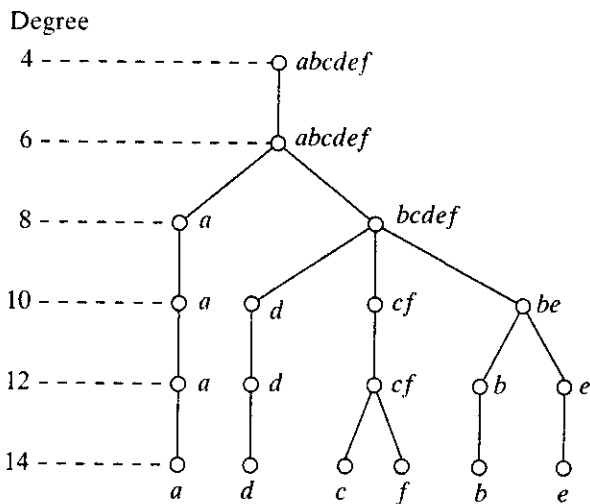
Suppose that we have a shape  $f$  and want to find its closest match in a set of five other shapes ( $a, b, c, d$ , and  $e$ ), as shown in Fig. 12.24(a). This problem is analogous to having five prototype shapes and trying to find the best match to a given unknown shape. The search may be visualized with the aid of the similarity tree shown in Fig. 12.24(b). The root of the tree corresponds to the lowest possible degree of similarity, which, for this example, is 4. Suppose that the shapes are identical up to degree 8, with the exception of shape  $a$ , whose degree of similarity with respect to all other shapes is 6. Proceeding down the tree, we find that shape  $d$  has degree of similarity 8 with respect to all others, and so on. Shapes

**EXAMPLE 12.7:**  
Using shape numbers to compare shapes.



a  
b c

**FIGURE 12.24**  
(a) Shapes.  
(b) Hypothetical similarity tree.  
(c) Similarity matrix. (Bribiesca and Guzman.)



	a	b	c	d	e	f
a	$\infty$	6	6	6	6	6
b		$\infty$	8	8	10	8
c			$\infty$	8	8	12
d				$\infty$	8	8
e					$\infty$	8
f						$\infty$

$f$  and  $c$  match uniquely, having a higher degree of similarity than any other two shapes. At the other extreme, if  $a$  had been an unknown shape, all we could have said using this method is that  $a$  was similar to the other five shapes with degree of similarity 6. The same information can be summarized in the form of a *similarity matrix*, as shown in Fig. 12.24(c).

### 12.3.2 String Matching

Suppose that two region boundaries,  $a$  and  $b$ , are coded into strings (see Section 11.5) denoted  $a_1a_2, \dots, a_n$  and  $b_1b_2, \dots, b_m$ , respectively. Let  $\alpha$  represent the number of matches between the two strings, where a match occurs in the  $k$ th position if  $a_k = b_k$ . The number of symbols that do not match is

$$\beta = \max(|a|, |b|) - \alpha \quad (12.3-4)$$

where  $|arg|$  is the length (number of symbols) in the string representation of the argument. It can be shown that  $\beta = 0$  if and only if  $a$  and  $b$  are identical (see Problem 12.21).

A simple measure of similarity between  $a$  and  $b$  is the ratio

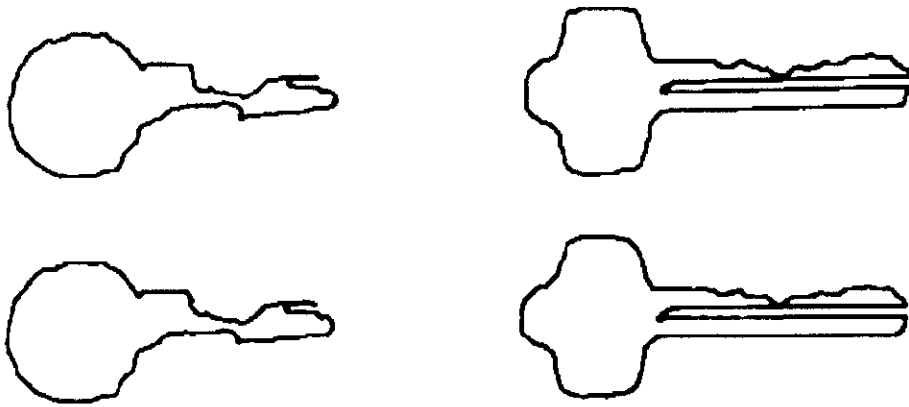
$$R = \frac{\alpha}{\beta} = \frac{\alpha}{\max(|a|, |b|) - \alpha}. \quad (12.3-5)$$

Hence  $R$  is infinite for a perfect match and 0 when none of the symbols in  $a$  and  $b$  match ( $\alpha = 0$  in this case). Because matching is done symbol by symbol, the starting point on each boundary is important in terms of reducing the amount of computation. Any method that normalizes to, or near, the same starting point is helpful, so long as it provides a computational advantage over brute-force matching, which consists of starting at arbitrary points on each string and then shifting one of the strings (with wraparound) and computing Eq. (12.3-5) for each shift. The largest value of  $R$  gives the best match.

#### EXAMPLE 12.8: Illustration of string matching.

Figures 12.25(a) and (b) show sample boundaries from each of two object classes, which were approximated by a polygonal fit (see Section 11.1.2). Figures 12.25(c) and (d) show the polygonal approximations corresponding to the boundaries shown in Figs. 12.25(a) and (b), respectively. Strings were formed from the polygons by computing the interior angle,  $\theta$ , between segments as each polygon was traversed clockwise. Angles were coded into one of eight possible symbols, corresponding to  $45^\circ$  increments; that is,  $\alpha_1: 0^\circ < \theta \leq 45^\circ$ ;  $\alpha_2: 45^\circ < \theta \leq 90^\circ$ ; ...;  $\alpha_8: 315^\circ < \theta \leq 360^\circ$ .

Figure 12.25(e) shows the results of computing the measure  $R$  for five samples of object 1 against themselves. The entries correspond to  $R$  values and, for example, the notation 1.c refers to the third string from object class 1. Figure 12.25(f) shows the results of comparing the strings of the second object class against themselves. Finally, Fig. 12.25(g) shows a tabulation of  $R$  values obtained by comparing strings of one class against the other. Note that, here, all  $R$  values are considerably smaller than any entry in the two preceding tabulations, indicating that the  $R$  measure achieved a high degree of discrimination between the two classes of objects. For example, if the class membership of string 1.a had



a b  
c d  
e f  
g

**FIGURE 12.25** (a) and (b) Sample boundaries of two different object classes; (c) and (d) their corresponding polygonal approximations; (e)–(g) tabulations of  $R$ . (Sze and Yang.)

$R$	1.a	1.b	1.c	1.d	1.e	1.f
1.a	$\infty$					
1.b	16.0	$\infty$				
1.c	9.6	26.3	$\infty$			
1.d	5.1	8.1	10.3	$\infty$		
1.e	4.7	7.2	10.3	14.2	$\infty$	
1.f	4.7	7.2	10.3	8.4	23.7	$\infty$

$R$	2.a	2.b	2.c	2.d	2.e	2.f
2.a	$\infty$					
2.b	33.5	$\infty$				
2.c	4.8	5.8	$\infty$			
2.d	3.6	4.2	19.3	$\infty$		
2.e	2.8	3.3	9.2	18.3	$\infty$	
2.f	2.6	3.0	7.7	13.5	27.0	$\infty$

$R$	1.a	1.b	1.c	1.d	1.e	1.f
2.a	1.24	1.50	1.32	1.47	1.55	1.48
2.b	1.18	1.43	1.32	1.47	1.55	1.48
2.c	1.02	1.18	1.19	1.32	1.39	1.48
2.d	1.02	1.18	1.19	1.32	1.29	1.40
2.e	0.93	1.07	1.08	1.19	1.24	1.25
2.f	0.89	1.02	1.02	1.24	1.22	1.18

been unknown, the *smallest* value of  $R$  resulting from comparing this string against sample (prototype) strings of class 1 would have been 4.7 [Fig. 12.25(e)]. By contrast, the *largest* value in comparing it against strings of class 2 would have been 1.24 [Fig. 12.25(g)]. This result would have led to the conclusion that string 1.a is a member of object class 1. This approach to classification is analogous to the minimum distance classifier introduced in Section 12.2.1. ■

### 12.3.2 Syntactic Recognition of Strings

Syntactic methods provide a unified methodology for handling structural recognition problems. Basically, the idea behind syntactic pattern recognition is the specification of a set of pattern *primitives* (see Section 11.5), a set of rules (in the form of a *grammar*) that governs their interconnection, and a *recognizer* (called an *automaton*), whose structure is determined by the set of rules in the grammar. First we consider string grammars and automata and then extend these ideas in the next section to tree grammars and their corresponding automata.

### String grammars

Suppose that we have two classes,  $\omega_1$  and  $\omega_2$ , whose patterns are strings of primitives, generated by one of the methods discussed in Section 11.5. We can interpret each primitive as being a symbol permissible in the *alphabet* of some *grammar*, where a grammar is a set of rules of syntax (hence the name syntactic recognition) that govern the generation of *sentences* formed from symbols of the alphabet. The set of sentences generated by a grammar,  $G$ , is called its *language* and is denoted  $L(G)$ . Here, sentences are strings of symbols (which in turn represent patterns), and languages correspond to pattern classes.

Consider two grammars,  $G_1$  and  $G_2$ , whose rules of syntax are such that  $G_1$  only allows generation of sentences that correspond to patterns from class  $\omega_1$ , and  $G_2$  only allows generation of sentences corresponding to patterns from class  $\omega_2$ . After two grammars with these properties have been established, the syntactic pattern recognition process, in principle, is straightforward. For a sentence representing an unknown pattern, the task is to decide in which language the pattern represents a valid sentence. If the sentence belongs to  $L(G_1)$ , we say that the pattern is from class  $\omega_1$ . Similarly, the pattern is said to be from class  $\omega_2$  if the sentence is valid in  $L(G_2)$ . A unique decision cannot be made if the sentence belongs to both languages. A sentence that is invalid in both languages is rejected.

When there are more than two pattern classes, the syntactic classification approach is the same as described in the preceding paragraph, with the exception that more grammars (at least one per class) are involved in the process. For multiclass classification, a pattern belongs to class  $\omega$ , if it is a valid sentence only of  $L(G_i)$ . As before, a unique decision cannot be made if a sentence belongs to more than one language. A sentence that is invalid over all languages is rejected.

When dealing with strings, we define a grammar as the 4-tuple

$$G = (N, \Sigma, P, S) \quad (12.3-6)$$

where

- $N$  is a finite set of variables called *nonterminals*,
- $\Sigma$  is a finite set of constants called *terminals*,
- $P$  is a set of rewriting rules called *productions*, and
- $S$  in  $N$  is called the *starting symbol*.

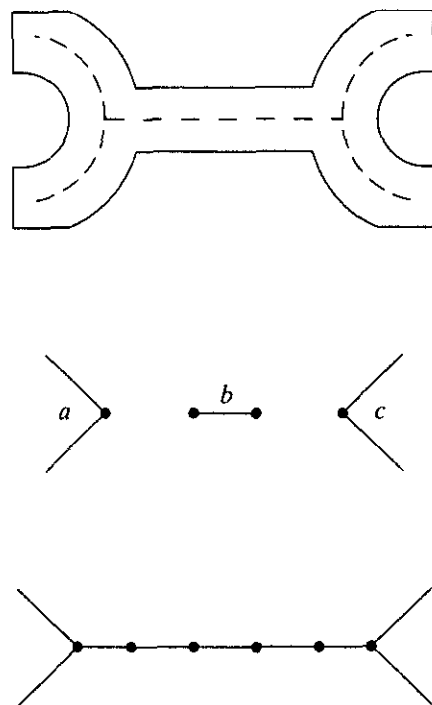
It is required that  $N$  and  $\Sigma$  be disjoint sets. In the following discussion, capital letters,  $A, B, \dots, S, \dots$ , denote nonterminals. Lowercase letters,  $a, b, c, \dots$  at the beginning of the alphabet denote terminals. Lowercase letters,  $v, w, x, y, z$  toward the end of the alphabet denote strings of terminals. Lowercase Greek letters  $\alpha, \beta, \theta, \dots$  denote strings of mixed terminals and nonterminals. The *empty sentence* (the sentence with no symbols) is denoted  $\lambda$ . Finally, for a set  $V$  of symbols, the notation  $V^*$  denotes the set of all sentences composed of elements from  $V$ .

String grammars are characterized by the form of their productions. Of particular interest in syntactic pattern recognition are *regular grammars* and *context-free grammars*. Regular grammars have productions only of the form  $A \rightarrow aB$  or

$A \rightarrow a$ , with  $A$  and  $B$  in  $N$  and  $a$  in  $\Sigma$ . Context-free grammars have productions only of the form  $A \rightarrow \alpha$ , with  $A$  in  $N$  and  $\alpha$  in the set  $(N \cup \Sigma)^*$ ; that is,  $\alpha$  can be any string composed of terminals and nonterminals, except the empty string.

Before proceeding, it will be useful to consider the mechanics of how grammars generate object classes. Suppose that the object shown in Fig. 12.26(a) is represented by its (pruned) skeleton and that we define the primitives shown in Fig. 12.26(b) to describe the structure of this (and similar) skeletons. Consider the grammar  $G = (N, \Sigma, P, S)$ , with  $N = \{A, B, S\}$ ,  $\Sigma = \{a, b, c\}$  and  $P = \{S \rightarrow aA, A \rightarrow bA, A \rightarrow bB, B \rightarrow c\}$ , where the terminals  $a, b$ , and  $c$  correspond to the primitives shown in Fig. 12.26(b). As indicated earlier,  $S$  is the starting symbol from which the strings of  $L(G)$  are generated. For instance, applying the first production followed by two applications of the second production yields  $S \Rightarrow aA \Rightarrow abA \Rightarrow abbA$ , where  $(\Rightarrow)$  indicates a string derivation starting from  $S$  and using productions from the set  $P$ . The first production allowed rewriting  $S$  as  $aA$ , and the second production allowed rewriting  $A$  as  $bA$ . With a nonterminal in the string  $abbA$ , we can continue the derivation. For example, applying the second production two more times, followed by one application of the third production and one application of the fourth production, yields the string  $abbbbcb$ , which corresponds to the structure shown in Fig. 12.26(c). No nonterminals remain after application of the fourth production, so the derivation terminates when this production is used. The language generated by the rules of this grammar is  $L(G) = \{ab^n c | n \geq 1\}$ , where  $b^n$  indicates  $n$  repetitions of the symbol  $b$ . In other words,  $G$  is capable of generating *only* skeletons of the form shown in Fig. 12.26(c) but having arbitrary length.

**EXAMPLE 12.9:** Object class generation using a regular string grammar.



a  
b  
c  
**FIGURE 12.26**  
(a) Object represented by its (pruned) skeleton.  
(b) Primitives.  
(c) Structure generated by using a regular string grammar.



**TABLE 12.1**  
Example of semantic information attached to production rules.

Production	Semantic Information
$S \rightarrow aA$	Connections to $a$ are made only at the dot. The direction of $a$ , denoted $\theta$ , is given by the direction of the perpendicular bisector of the line joining the end points of the two undotted segments. The line segments are 3 cm each.
$A \rightarrow bA$	Connections to $b$ are made only at the dots. No multiple connections are allowed. The direction of $b$ must be the same as the direction of $a$ . The length of $b$ is 0.25 cm. This production cannot be applied more than 10 times.
$A \rightarrow bB$	The direction of $a$ and $b$ must be the same. Connections must be simple and made only at the dots.
$B \rightarrow c$	The direction of $c$ and $a$ must be the same. Connections must be simple and made only at the dots.

### Use of semantics

In the preceding example we assumed that the interconnection between primitives takes place only at the dots shown in Fig. 12.26(b). In more complicated situations the rules of connectivity, as well as information regarding other factors (such as primitive length and direction) and the number of times a production can be applied, must be made explicit. This can be accomplished by using *semantic rules* stored in the *knowledge base* of Fig. 1.23. Basically, the syntax inherent in the production rules establishes the structure of an object, whereas semantics deal with its correctness. For example, in a programming language like C, the statement  $A = D/E$  is syntactically correct, but it is semantically correct only if  $E \neq 0$ .

Suppose that we attach semantic information to the grammar discussed in the preceding example. The information can be attached to the production rules in the form shown in Table 12.1. By using semantic information, we are able to use a few rules of syntax to describe a broad (but limited as desired) class of patterns. For instance, by specifying the direction of  $\theta$  in Table 12.1, we avoid having to specify primitives for each possible orientation. Similarly, by requiring that all primitives be oriented in the same direction, we eliminate from consideration nonsensical structures that deviate from the basic shapes typified by Fig. 12.26(a).

### Automata as string recognizers

So far we have demonstrated that grammars are *generators* of patterns. In the following discussion we consider the problem of recognizing whether a pattern belongs to the language  $L(G)$  generated by a grammar  $G$ . The basic concepts underlying syntactic recognition may be illustrated by the development of mathematical models of computing machines, called *automata*. Given an input pattern string, an automaton is capable of recognizing whether the pattern belongs to the language with which the automaton is associated. Here, we focus only on *finite automata*, which are the recognizers of languages generated by regular grammars.

A *finite automaton* is defined as the 5-tuple

$$A_f = (Q, \Sigma, \delta, q_0, F) \quad (12.3-7)$$

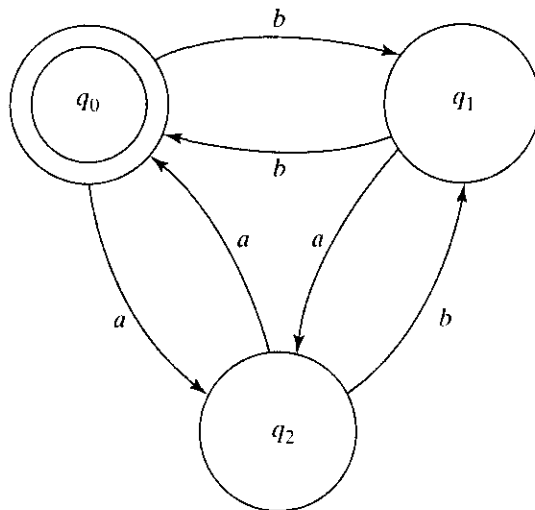
where  $Q$  is a finite, nonempty set of *states*,  $\Sigma$  is a finite input *alphabet*,  $\delta$  is a *mapping* from  $Q \times \Sigma$  (the set of ordered pairs formed from elements of  $Q$  and  $\Sigma$ ) into the collection of all subsets of  $Q$ ,  $q_0$  is the *starting state*, and  $F$  (a subset of  $Q$ ) is a set of *final*, or *accepting*, *states*.

Consider an automaton given by Eq. (12.3-7), with  $Q = \{q_0, q_1, q_2\}$ ,  $\Sigma = \{a, b\}$ ,  $F = \{q_0\}$ , and mappings  $\delta(q_0, a) = \{q_2\}$ ,  $\delta(q_0, b) = \{q_1\}$ ,  $\delta(q_1, a) = \{q_2\}$ ,  $\delta(q_1, b) = \{q_0\}$ ,  $\delta(q_2, a) = \{q_0\}$ , and  $\delta(q_2, b) = \{q_1\}$ . If, for example, the automaton is in state  $q_0$  and an  $a$  is input, its state changes to  $q_2$ . Similarly, if a  $b$  is input next, the automaton changes to state  $q_1$ , and so on. The initial and final states are the same in this case.  $\square$

**EXAMPLE 12.10:**  
A simple automaton.

Figure 12.27 shows a *state diagram* for the automaton just discussed. The state diagram consists of a node for each state and directed arcs showing the possible transitions between states. The final state is shown as a double circle, and each arc is labeled with the symbol that causes the transition between the states joined by that arc. In this case the initial and final states are the same. A string  $w$  of terminal symbols is said to be *accepted* or *recognized* by an automaton if, starting in state  $q_0$ , the sequence of symbols (encountered as  $w$  is scanned from left to right) causes the automaton to be in a final state after the last symbol from  $w$  has been scanned. For example, the automaton in Fig. 12.27 recognizes the string  $w = abbabb$  but rejects the string  $w = aabab$ .

There is a one-to-one correspondence between regular grammars and finite automata. That is, a language is recognized by a finite automaton if and only if it is generated by a regular grammar. The design of a syntactic string recognizer based on the concepts discussed so far is a straightforward procedure, consisting of obtaining a finite automaton from a given regular grammar. Let the grammar be denoted  $G = (N, \Sigma, P, X_0)$ , where  $X_0 \equiv S$ , and suppose that  $N$  is composed of  $X_0$  plus  $n$  additional nonterminals  $X_1, X_2, \dots, X_n$ . The set  $Q$  for the automaton is formed by introducing  $n + 2$  states  $\{q_0, q_1, \dots, q_n, q_{n+1}\}$  such that  $q_i$  corresponds



**FIGURE 12.27** A finite automaton.

to  $X_i$  for  $0 \leq i \leq n$ , and  $q_{n+1}$  is the final state. The set of input symbols is identical to the set of terminals in  $G$ . The mappings in  $\delta$  are obtained by using two rules based on the productions of  $G$ ; namely, for each  $i$  and  $j$ , with  $0 \leq i \leq n, 0 \leq j \leq n$ ,

1. If  $X_i \rightarrow aX_j$  is in  $P$ , then  $\delta(q_i, a)$  contains  $q_j$ .
2. If  $X_i \rightarrow a$  is in  $P$ , then  $\delta(q_i, a)$  contains  $q_{n+1}$ .

Conversely, given a finite automaton,  $A_f = (Q, \Sigma, \delta, q_0, F)$ , we obtain the corresponding regular grammar,  $G = (N, \Sigma, P, X_0)$  by letting  $N$  consist of the elements of  $Q$ , with the starting symbol  $X_0$  corresponding to  $q_0$ , and the productions of  $G$  obtained as follows:

1. If  $q_j$  is in  $\delta(q_i, a)$ , there is a production  $X_i \rightarrow aX_j$  in  $P$ .
2. If a state in  $F$  is in  $\delta(q_i, a)$  there is a production  $X_i \rightarrow a$  in  $P$ .

The terminal set,  $\Sigma$ , is the same in both cases.

**EXAMPLE 12.11:**  
Finite automaton  
for recognizing  
the patterns in  
Fig. 12.26.

■ The finite automaton for the grammar given in connection with Fig. 12.26 is obtained by writing the productions as  $X_0 \rightarrow aX_1, X_1 \rightarrow bX_1, X_1 \rightarrow bX_2$ , and  $X_2 \rightarrow c$ . Then  $A_f = (Q, \Sigma, \delta, q_0, F)$ , with  $Q = \{q_0, q_1, q_2, q_3\}, \Sigma = \{a, b, c\}, F = \{q_3\}$  and mappings  $\delta(q_0, a) = \{q_1\}, \delta(q_1, b) = \{q_1, q_2\}, \delta(q_2, c) = \{q_3\}$ . For completeness, we write  $\delta(q_0, b) = \delta(q_0, c) = \delta(q_1, a) = \delta(q_1, c) = \delta(q_2, a) = \delta(q_2, b) = \emptyset$ , where  $\emptyset$  is the null set, indicating that these transitions are not defined for this automaton.

### 12.3.4 Syntactic Recognition of Trees

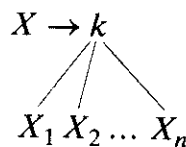
Following a format similar to the preceding discussion for strings, we now expand the discussion to include tree descriptions of patterns. We assume that the image regions or objects of interest have been expressed in the form of trees by using the appropriate primitive elements, as discussed in Section 11.5.

#### Tree grammars

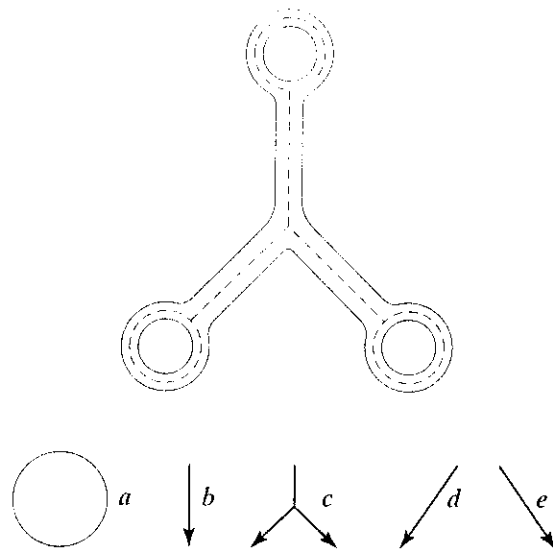
A *tree grammar* is defined as the 5-tuple

$$G = (N, \Sigma, P, r, S) \tag{12.3-8}$$

where, as before,  $N$  and  $\Sigma$  are sets of nonterminals and terminals, respectively;  $S$ , contained in  $N$ , is the start symbol, which in general can be a tree;  $P$  is a set of productions of the form  $T_i \rightarrow T_j$ , where  $T_i$  and  $T_j$  are trees; and  $r$  is a *ranking function* that denotes the number of direct descendants (offspring) of a node whose label is a terminal in the grammar. Of particular relevance to our discussion are *expansive* tree grammars having productions of the form



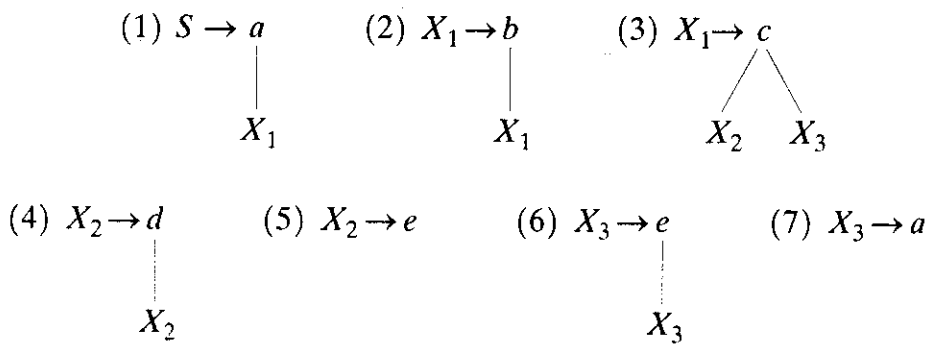
where  $X_1, X_2, \dots, X_n$  are nonterminals and  $k$  is a terminal.



a  
b  
**FIGURE 12.28**  
(a) An object and  
(b) primitives  
used for  
representing the  
skeleton by  
means of a tree  
grammar.

■ The skeleton of the structure shown in Fig. 12.28(a) can be generated by using a tree grammar with  $N = \{X_1, X_2, X_3, S\}$  and  $\Sigma = \{a, b, c, d, e\}$ , where the terminals represent the primitives shown in Fig. 12.28(b). Assuming head-to-tail connectivity of the line primitives, and arbitrary connections to the circle along its circumference, the grammar under consideration has productions of the form

**EXAMPLE 12.12:**  
A simple tree  
grammar.



The ranking functions in this case are  $r(a) = \{0, 1\}$ ,  $r(b) = r(d) = \{1\}$ ,  $r(e) = \{0, 1\}$ , and  $r(c) = \{2\}$ . Restricting application of productions 2, 4, and 6 to the same number of times would generate a structure in which all three legs have the same length. Similarly, requiring application of productions 4 and 6 the same number of times would produce a structure that is symmetrical about its vertical axis. This type of semantic information is similar to the earlier discussion in connection with Table 12.1 and the knowledge base of Fig. 1.23.

**Tree automata**

Whereas a conventional finite automaton scans an input string symbol by symbol from left to right, a tree automaton must begin simultaneously at each node on the frontier (the leaves taken in order from left to right) of an input tree

and proceed along parallel paths toward the root. Specifically, a *frontier-to-root automaton* is defined as

$$A_t = (Q, F, \{f_k | k \in \Sigma\}) \tag{12.3-9}$$

where

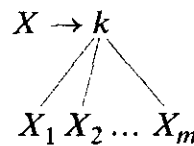
$Q$  is a finite set of states,

$F$ , a subset of  $Q$ , is a set of final states, and

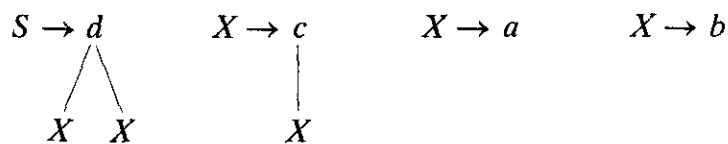
$f_k$  is a relation in  $Q^m \times Q$  such that  $m$  is a rank of  $k$ .

The notation  $Q^m$  indicates the Cartesian product of  $Q$  with itself  $m$  times:  $Q^m = Q \times Q \times Q \times \dots \times Q$ . From the definition of the Cartesian product, we know that this expression means the set of all ordered  $m$ -tuples with elements from  $Q$ . For example, if  $m = 3$ , then  $Q^3 = Q \times Q \times Q = \{x, y, z | x \in Q, y \in Q, z \in Q\}$ . Recall that a *relation*  $R$  from a set  $A$  to a set  $B$  is a subset of the Cartesian product of  $A$  and  $B$ ; that is,  $R \subseteq A \times B$ . Thus a relation in  $Q^m \times Q$  is simply a subset of the set  $Q^m \times Q$ .

For an expansive tree grammar,  $G = (N, \Sigma, P, r, S)$ , we construct the corresponding tree automaton by letting  $Q = N$ , with  $F = \{S\}$  and, for each symbol  $a$  in  $\Sigma$ , defining a relation  $f_k$  such that  $(X_1, X_2, \dots, X_m, X)$  is in  $f_k$  if and only if there is in  $G$  a production



For example, consider the tree grammar  $G = (N, \Sigma, P, r, S)$ , with  $N = \{S, X\}$ ,  $\Sigma = \{a, b, c, d\}$ , productions



and rankings  $r(a) = \{0\}$ ,  $r(b) = \{0\}$ ,  $r(c) = \{1\}$ , and  $r(d) = \{2\}$ . The corresponding tree automaton,  $A_t = (Q, F, \{f_k | k \in \Sigma\})$ , is specified by letting  $Q = \{S, X\}$ ,  $F = \{S\}$ , and  $\{f_k | k \in \Sigma\} = \{f_a, f_b, f_c, f_d\}$ , where the relations are defined as

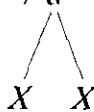
$$f_a = \{(\emptyset, X)\}, \text{ arising from production } X \rightarrow a$$

$$f_b = \{(\emptyset, X)\}, \text{ arising from production } X \rightarrow b$$

$$f_c = \{(X, X)\}, \text{ arising from production } X \rightarrow c$$

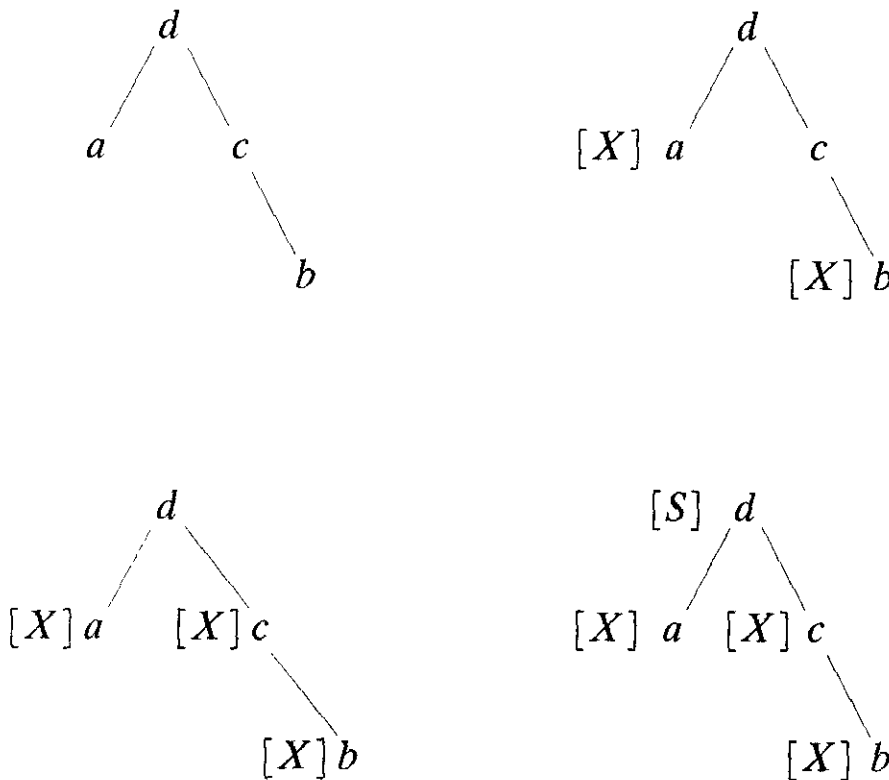


and

$$f_d = \{(X, X, S)\}, \text{ arising from production } S \rightarrow d$$


The interpretation of relation  $f_a$  is that a node labeled  $a$  with no offspring (hence the null symbol  $\emptyset$ ) is assigned state  $X$ . The interpretation of  $f_c$  is that a node labeled  $c$ , with one offspring having state  $X$ , is assigned state  $X$ . The interpretation of relation  $f_d$  is that a node labeled  $d$  with two offspring, each having state  $X$ , is assigned state  $S$ .

In order to see how this tree automaton goes about recognizing a tree generated by the grammar discussed earlier, consider the tree shown in Fig. 12.29(a). Automaton  $A$ , first assigns states to the frontier nodes  $a$  and  $b$  via relations  $f_a$  and  $f_b$ , respectively. In this case, according to these two relations, state  $X$  is assigned to both leaves, as Fig. 12.29(b) shows. The automaton now moves up one level from the frontier and makes a state assignment to node  $c$  on the basis of  $f_c$  and the state of this node's offspring. The state assignment based on  $f_c$  again is  $X$ , as indicated in Fig. 12.29(c). Moving up one more level, the automaton encounters node  $d$  and, as its two offspring have been assigned states, relation  $f_d$ , which calls for assigning state  $S$  to node  $d$ , is used. Because this is the last node and the state  $S$  is in  $F$ , the automaton accepts (recognizes) the tree as being a valid member of the language of the tree grammar given earlier. Figure 12.29(d) shows the final representation of the state sequences followed along the frontier-to-root paths.



a b  
c d



**FIGURE 12.29**  
Processing stages of a frontier-to-root tree automaton:  
(a) Input tree.  
(b) State assignment to frontier nodes.  
(c) State assignment to intermediate nodes.  
(d) State assignment to root node.

**EXAMPLE 12.13:** Use of tree grammars for recognizing events in bubble chamber images.

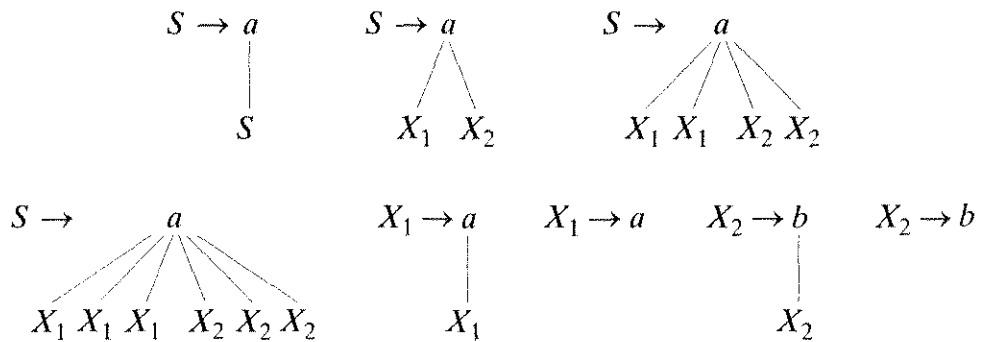
Images of bubble chamber events are taken routinely during experiments in high-energy physics in which a beam of particles of known properties is directed onto a target of known nuclei. A typical event consists of tracks of secondary particles emanating from the point of collision, such as the example shown in Fig. 12.30. The incoming tracks are the horizontal parallel lines. Note the natural tree structure of the event near the middle of the photograph.

A typical experiment produces hundreds of thousands of photographs, many of which do not contain events of interest. Examining and categorizing these photographs is tedious and time-consuming for a human interpreter, thus creating a need for automatic event recognition techniques.

A tree grammar  $G = (N, \Sigma, P, r, S)$  can be specified that generates trees representing events typical of those found in a hydrogen bubble chamber as a result of incoming positively charged particle streams. In this case,  $N = \{S, X_1, X_2\}$ ,  $\Sigma = \{a, b\}$ , and the primitives  $a$  and  $b$  are interpreted as follows:

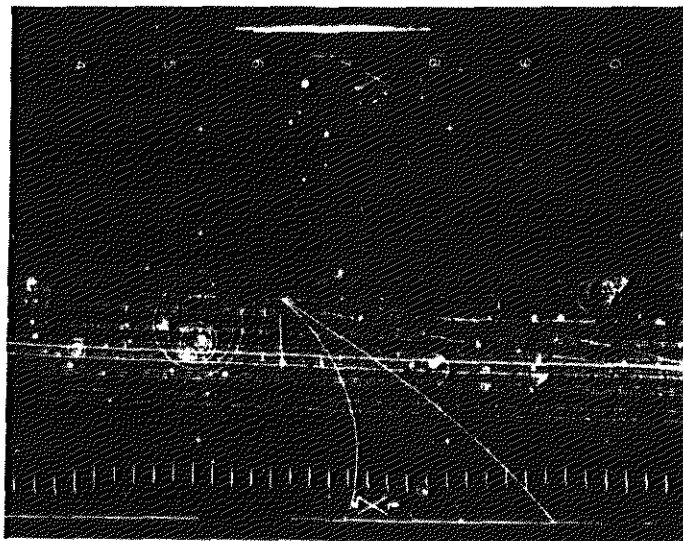
- $a$ :  convex arc
- $b$ :  concave arc.

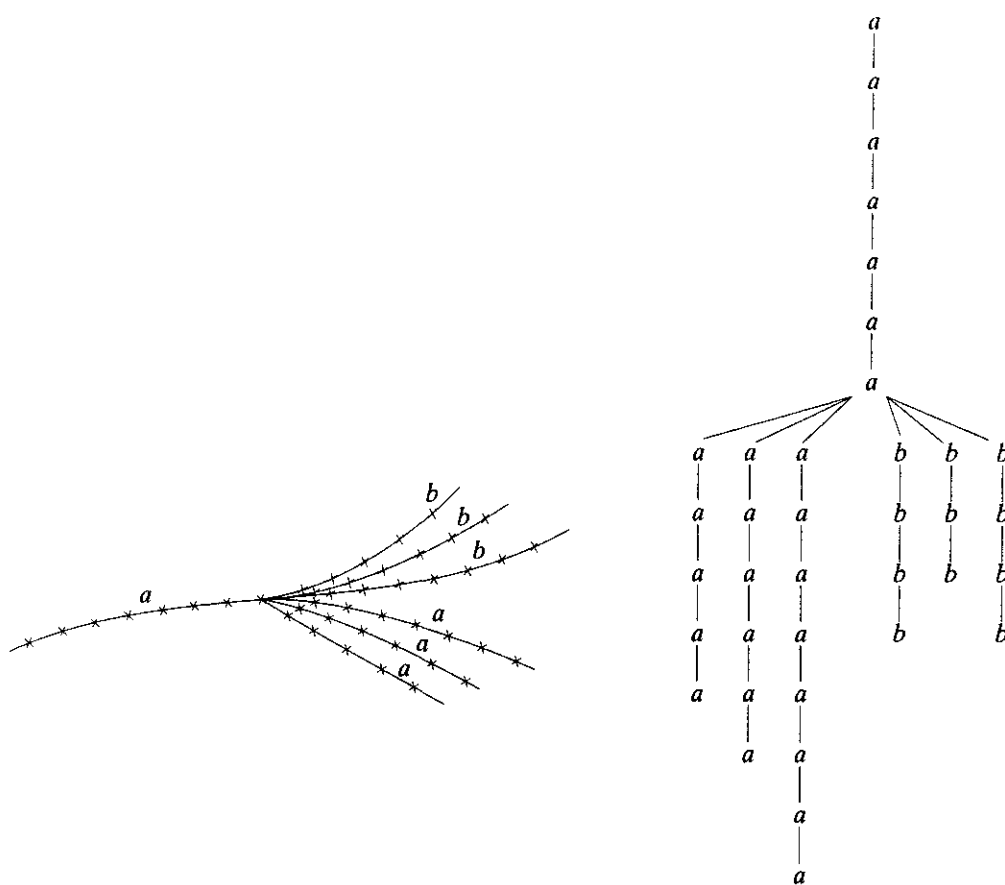
The productions in  $P$  are



The rankings are  $r(a) = \{0, 1, 2, 4, 6\}$  and  $r(b) = \{0, 1\}$ . The branching productions represent the number of tracks emanating from a collision, which occur

**FIGURE 12.30** A bubble chamber photograph. (Fu and Bhargava.)





a b  
**FIGURE 12.31**  
 (a) Coded event  
 from Fig. 12.30.  
 (b) Corresponding  
 tree representation.  
 (Fu and Bhargava.)

in pairs and usually do not exceed six. Figure 12.31(a) shows the collision event in Fig. 12.30 segmented into convex and concave sections, and Fig. 12.31(b) shows the corresponding tree representation. This tree, as well as variations of it, can be generated by the grammar given above.

The tree automaton needed to recognize the types of trees just discussed is defined by using the procedure outlined in the preceding discussion. Thus,  $A_t = (Q, F, \{f_k | k \in \Sigma\})$  is specified by letting  $Q = \{S, X_1, X_2\}$ ,  $F = \{S\}$ , and  $\{f_k | k \in \Sigma\} = \{f_a, f_b\}$ . The relations are defined as  $f_a = \{(S, S), (X_1, X_2, S), (X_1, X_1, X_2, X_2, S), (X_1, X_1, X_1, X_2, X_2, X_2, S), (X_1, X_1), (\emptyset, X_1)\}$  and  $f_b = \{(X_2, X_2), (\emptyset, X_2)\}$ . We leave it as an exercise to show that this automaton accepts the tree in Fig. 12.31(b).

### Learning

The syntactic recognition approaches introduced in the preceding discussion require specification of the appropriate automata (recognizers) for each class under consideration. In simple situations, inspection may yield the necessary automata. In more complicated cases, an algorithm for learning the automata from sample patterns (such as strings or trees) may be required. Because of the one-to-one correspondence between automata and grammars described previously, the learning problem sometimes is posed in terms of learning grammars directly from sample patterns, a process that usually is called *grammatical inference*. Our



focus is on learning finite automata directly from sample pattern strings. The references at the end of this chapter provide a guide to methods for learning tree grammars and automata, as well as other syntactic recognition approaches.

Suppose that all patterns of a class are generated by an *unknown* grammar  $G$  and that a finite set of samples  $R^+$  with the property

$$R^+ \subseteq \{v \mid v \text{ in } L(G)\} \quad (12.3-10)$$

is available. The set  $R^+$ , called a *positive sample set*, is simply a set of training patterns from the class associated with grammar  $G$ . This sample set is said to be *structurally complete* if each production in  $G$  is used to generate at least one element of  $R^+$ . We want to learn (synthesize) a finite automaton  $A_f$  that will accept the strings of  $R^+$  and possibly some strings that resemble those of  $R^+$ .

Based on the definition of a finite automaton and the correspondence between  $G$  and  $A_f$ , it follows that  $R^+ \subseteq \Sigma^*$ , where  $\Sigma^*$  is the set of all strings composed of elements from  $\Sigma$ . Let  $z$  in  $\Sigma^*$  be a string such that  $zw$  is in  $R^+$  for some  $w$  in  $\Sigma^*$ . For a positive integer  $k$ , we define the  $k$  tail of  $z$  with respect to  $R^+$  as the set  $h(z, R^+, k)$ , where

$$h(z, R^+, k) = \{w \mid zw \text{ in } R^+, |w| \leq k\}. \quad (12.3-11)$$

In other words, the  $k$  tail of  $z$  is the set of strings  $w$  with the properties (1)  $zw$  is in  $R^+$ , and (2) the length of  $w$  is less than or equal to  $k$ .

A procedure for learning an automaton  $A_f(R^+, k) = (Q, \Sigma, \delta, q_0, F)$  from a sample set  $R^+$  and a particular value of  $k$  consists of letting

$$Q = \{q \mid q = h(z, R^+, k) \text{ for } z \text{ in } \Sigma^*\} \quad (12.3-12)$$

and, for each  $a$  in  $\Sigma$ ,

$$\delta(q, a) = \{q' \text{ in } Q \mid q' = h(za, R^+, k), \text{ with } q = h(z, R^+, k)\}. \quad (12.3-13)$$

In addition, we let

$$q_0 = h(\lambda, R^+, k) \quad (12.3-14)$$

and

$$F = \{q \mid q \text{ in } Q, \lambda \text{ in } q\} \quad (12.3-15)$$

where  $\lambda$  is the empty string (the string with no symbols). We note that the automaton  $A_f(R^+, k)$  has as states subsets of the set of all  $k$  tails that can be constructed from  $R^+$ .

**EXAMPLE 12.14:**  
Inferring a finite automaton from sample patterns.

Suppose that  $R^+ = \{a, ab, abb\}$  and  $k = 1$ . Then from the preceding discussion.

$$\begin{aligned} z = \lambda, \quad h(\lambda, R^+, 1) &= \{w \mid \lambda w \text{ in } R^+, |w| \leq 1\} \\ &= \{a\} \\ &= q_0 \\ z = a, \quad h(a, R^+, 1) &= \{w \mid aw \text{ in } R^+, |w| \leq 1\} \\ &= \{\lambda, b\} \\ &= q_1 \end{aligned}$$

$$\begin{aligned}
 z = ab, \quad h(ab, R^+, 1) &= \{\lambda, b\} \\
 &= q_1 \\
 z = abb, \quad h(abb, R^+, 1) &= \{\lambda\} \\
 &= q_2.
 \end{aligned}$$

In this case, other strings  $z$  in  $\Sigma^*$  yield strings  $zw$  that do not belong to  $R^+$ , giving rise to a fourth state, denoted  $q_\emptyset$ , which corresponds to the condition that  $h$  is the null set. The states, therefore, are  $q_0 = \{a\}$ ,  $q_1 = \{\lambda, a\}$ ,  $q_2 = \{\lambda\}$ , and  $q_\emptyset$ , which give the set  $Q = \{q_0, q_1, q_2, q_\emptyset\}$ . Although the states are obtained as sets of symbols ( $k$  tails), only the state labels  $q_0, q_1, \dots$  are used in forming the set  $Q$ . The next step is to obtain the transition functions. Since  $q_0 = h(\lambda, R^+, 1)$ , it follows that

$$\delta(q_0, a) = h(\lambda a, R^+, 1) = h(a, R^+, 1) = q_1$$

and

$$\delta(q_0, b) = h(\lambda b, R^+, 1) = h(b, R^+, 1) = q_\emptyset.$$

Similarly,  $q_1 = h(a, R^+, 1) = h(ab, R^+, 1)$  and it follows that

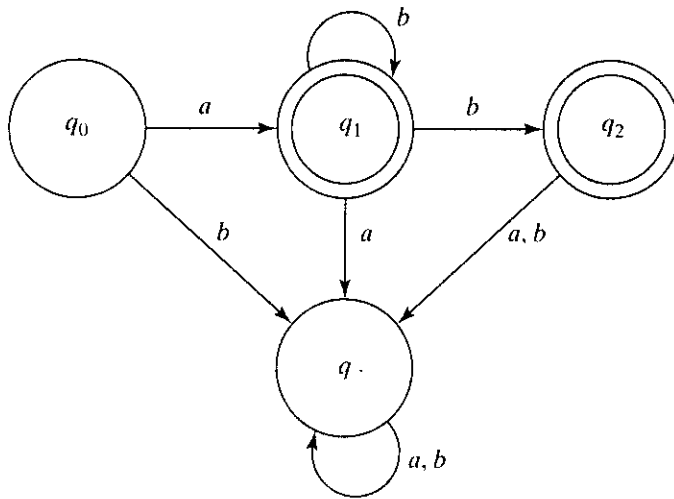
$$\delta(q_1, a) = h(aa, R^+, 1) = h(aba, R^+, 1) = q_\emptyset.$$

Also,  $\delta(q_1, b) \supseteq h(ab, R^+, 1) = q_1$  and  $\delta(q_1, b) \supseteq h(abb, R^+, 1) = q_2$ ; that is,  $\delta(q_1, b) = \{q_1, q_2\}$ . Following the procedure just described gives  $\delta(q_2, a) = \delta(q_2, b) = \delta(q_\emptyset, a) = \delta(q_\emptyset, b) = q_\emptyset$ . The set of final states contains those states that have the empty string  $\lambda$  in their  $k$ -tail representation. In this case,  $q_1 = \{\lambda, a\}$  and  $q_2 = \{\lambda\}$ , so  $F = \{q_1, q_2\}$ .

Based on these results, the inferred automaton is given by

$$A_f(R^+, 1) = (Q, \Sigma, \delta, q_0, F)$$

where  $Q = \{q_0, q_1, q_2, q_\emptyset\}$ ,  $\Sigma = \{a, b\}$ ,  $F = \{q_1, q_2\}$ , and the transition functions are as given above. Figure 12.32 shows the state diagram. The automaton accepts strings of the form  $a, ab, abb, \dots, ab^n$ , which are consistent with the given sample set.



**FIGURE 12.32**  
State diagram for the finite automaton inferred from the sample set  $R^+ = \{a, ab, abb\}$ .

The preceding example shows that the value of  $k$  controls the nature of the resulting automaton. The following properties exemplify the dependence of  $A_f(R^+, k)$  on this parameter.

*Property 1.*  $R^+ \subseteq L[A_f(R^+, k)]$  for all  $k \geq 0$ , where  $L[A_f(R^+, k)]$  is the language accepted by  $A_f(R^+, k)$ .

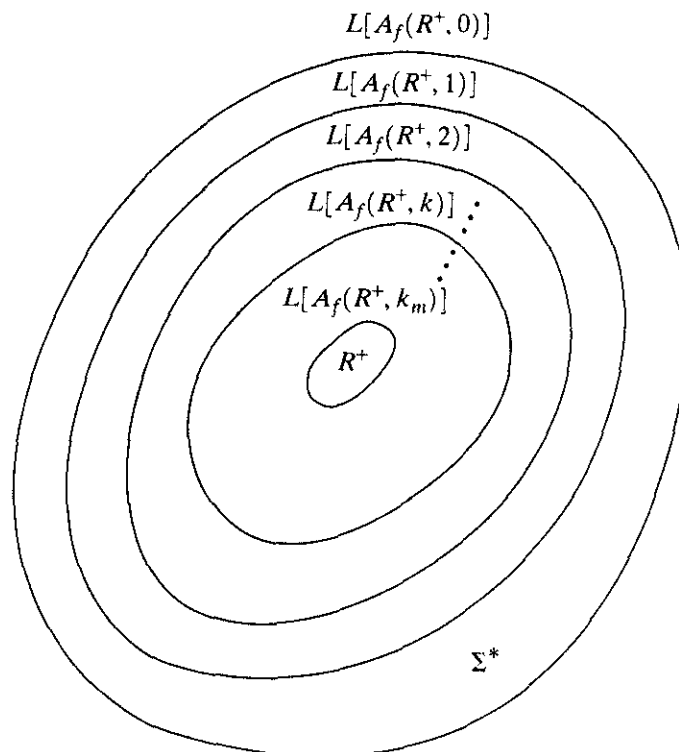
*Property 2.*  $L[A_f(R^+, k)] = R^+$  if  $k$  is equal to, or greater than, the length of the longest string in  $R^+$ ;  $L[A_f(R^+, k)] = \Sigma^*$  if  $k = 0$ .

*Property 3.*  $L[A_f(R^+, k + 1)] \subseteq L[A_f(R^+, k)]$ .

Property 1 guarantees that  $A_f(R^+, k)$  will, as a minimum, accept the strings in the sample set  $R^+$ . If  $k$  is equal to, or greater than, the length of the longest string in  $R^+$ , then by Property 2 the automation will accept *only* the strings in  $R^+$ . If  $k = 0$ ,  $A_f(R^+, 0)$  will consist of one state  $q_0 = \{\lambda\}$ , which will act as both the initial and final states. The transition functions will then be of the form  $\delta(q_0, a) = q_0$  for  $a$  in  $\Sigma$ . Therefore,  $L[A_f(R^+, 0)] = \Sigma^*$ , and the automaton will accept the empty string  $\lambda$  and all strings composed of symbols from  $\Sigma$ . Finally, Property 3 indicates that the scope of the language accepted by  $A_f(R^+, k)$  decreases as  $k$  increases.

These three properties allow control of the nature of  $A_f(R^+, k)$  simply by varying the parameter  $k$ . If  $L[A_f(R^+, k)]$  is a guess of the language  $L_0$  from which the sample  $R^+$  was chosen and if  $k$  is very small, this guess of  $L_0$  will constitute a liberal inference that may include most or all of the strings in  $\Sigma^*$ . However, if  $k$  is equal to the length of the longest string in  $R^+$ , the inference will be conservative in the sense that  $A_f(R^+, k)$  will accept only the strings contained in  $R^+$ . Figure 12.33 shows these concepts graphically.

**FIGURE 12.33**  
Relationship between  $L[A_f(R^+, k)]$  and  $k$ . The value of  $k_m$  is such that  $k_m \geq$  (length of the longest string in  $R^+$ ).



Consider the set  $R^+ = \{caaab, bbaab, caab, bbab, cab, bbb, cb\}$ . For  $k = 1$ , following the same procedure used in the preceding example gives

**EXAMPLE 12.15:** Another example of inferring an automaton from a given set of patterns.

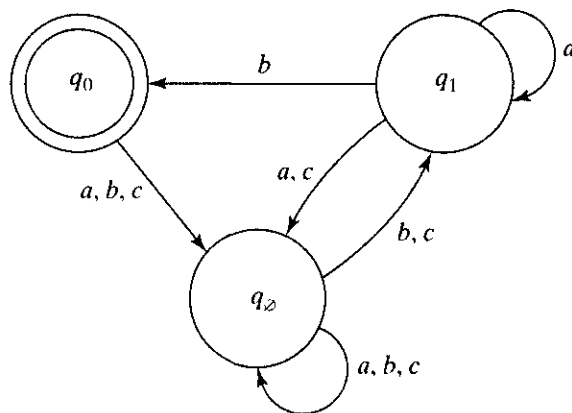
- 1.  $z = \lambda, \quad h(\lambda, R^+, 1) = \{\emptyset\} = q_\emptyset;$
- 2.  $z = c, \quad h(z, R^+, 1) = \{b\} = q_1;$
- 3.  $z = ca, \quad h(z, R^+, 1) = \{b\} = q_1;$
- 4.  $z = cb, \quad h(z, R^+, 1) = \{\lambda\} = q_\emptyset;$
- 5.  $z = caa, \quad h(z, R^+, 1) = \{b\} = q_1;$
- 6.  $z = cab, \quad h(z, R^+, 1) = \{\lambda\} = q_\emptyset;$
- 7.  $z = caaa, \quad h(z, R^+, 1) = \{b\} = q_1;$
- 8.  $z = caab, \quad h(z, R^+, 1) = \{\lambda\} = q_\emptyset;$
- 9.  $z = caaab, \quad h(z, R^+, 1) = \{\lambda\} = q_\emptyset;$
- 10.  $z = b, \quad h(z, R^+, 1) = \{\emptyset\} = q_\emptyset;$
- 11.  $z = bb, \quad h(z, R^+, 1) = \{b\} = q_1;$
- 12.  $z = bba, \quad h(z, R^+, 1) = \{b\} = q_1;$
- 13.  $z = bbb, \quad h(z, R^+, 1) = \{\lambda\} = q_\emptyset;$
- 14.  $z = bbba, \quad h(z, R^+, 1) = \{b\} = q_1;$
- 15.  $z = bbab, \quad h(z, R^+, 1) = \{\lambda\} = q_\emptyset;$
- 16.  $z = bbaab, \quad h(z, R^+, 1) = \{\lambda\} = q_\emptyset.$

The automaton is

$$A_f(R^+, 1) = (Q, \Sigma, \delta, q_0, F)$$

with  $Q = \{q_0, q_1, q_\emptyset\}$ ,  $\Sigma = \{a, b, c\}$ ,  $F = \{q_0\}$ , and the transitions shown in the state diagram in Fig. 12.34. To be accepted by the automaton, a string must begin with  $a, b$ , or  $c$  and end with a symbol  $b$ . Also, strings with repetitions of  $a, b$ , or  $c$  are accepted by  $A_f(R^+, 1)$ .

The principal advantage of the preceding method is simplicity of implementation. The synthesis procedure can be simulated in a digital computer with a modest amount of effort. The main disadvantage is deciding on a proper value for  $k$ , although this problem is simplified to some degree by the three properties discussed earlier.



**FIGURE 12.34** State diagram for the automaton  $A_f(R^+, 1)$  inferred from the sample set  $R^+ = \{caaab, bbaab, caab, bbab, cab, bbb, cb\}$ .