CHAPTER

Nonlinear Classifiers

4

4.1 INTRODUCTION

In the previous chapter we dealt with the design of linear classifiers described by linear discriminant functions (hyperplanes) $g(\boldsymbol{x})$. In the simple two-class case, we saw that the perceptron algorithm computes the weights of the linear function $g(\boldsymbol{x})$, provided that the classes are linearly separable. For nonlinearly separable classes, linear classifiers were optimally designed, for example, by minimizing the squared error. In this chapter we will deal with problems that are not linearly separable and for which the design of a linear classifier, even in an optimal way, does not lead to satisfactory performance. The design of nonlinear classifiers emerges now as an inescapable necessity.

4.2 THE XOR PROBLEM

To seek nonlinearly separable problems one does not need to go into complicated situations. The well-known *Exclusive OR (XOR)* Boolean function is a typical example of such a problem. Boolean functions can be interpreted as classification tasks. Indeed, depending on the values of the input binary data $\boldsymbol{x} = [x_1, x_2, \ldots, x_l]^T$, the output is either 0 or 1, and $\boldsymbol{x}$ is classified into one of the two classes $A(1)$ or $B(0)$. The corresponding truth table for the XOR operation is shown in Table 4.1.

Figure 4.1 shows the position of the classes in space. It is apparent from this figure that no single straight line exists that separates the two classes. In contrast, the other two Boolean functions, AND and OR, are linearly separable. The corresponding truth tables for the AND and OR operations are given in Table 4.2 and the respective class positions in the two-dimensional space are shown in Figure 4.2a and 4.2b. Figure 4.3 shows a perceptron, introduced in the previous chapter, with synaptic weights computed so as to realize an OR gate (verify).

Our major concern now is first to tackle the XOR problem and then to extend the procedure to more general cases of nonlinearly separable classes. Our kickoff point will be geometry.

Table 4.1 Truth Table for the XOR Problem

x_1	x_2	XOR	Class
0	0	0	B
0	1	1	A
1	0	1	A
1	1	0	B

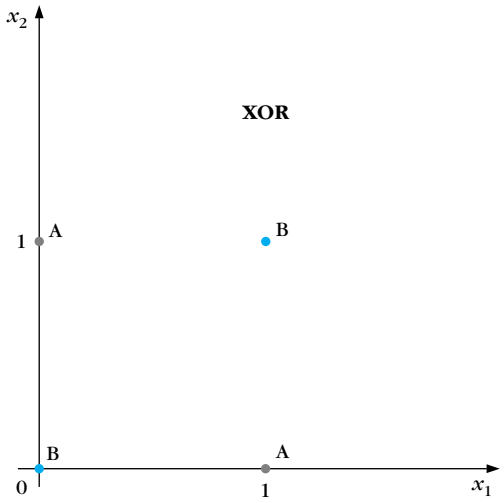

FIGURE 4.1

Classes A and B for the XOR problem.

Table 4.2 Truth Table for AND and OR Problems

x_1	x_2	AND	Class	OR	Class
0	0	0	B	0	B
0	1	0	B	1	A
1	0	0	B	1	A
1	1	1	A	1	A

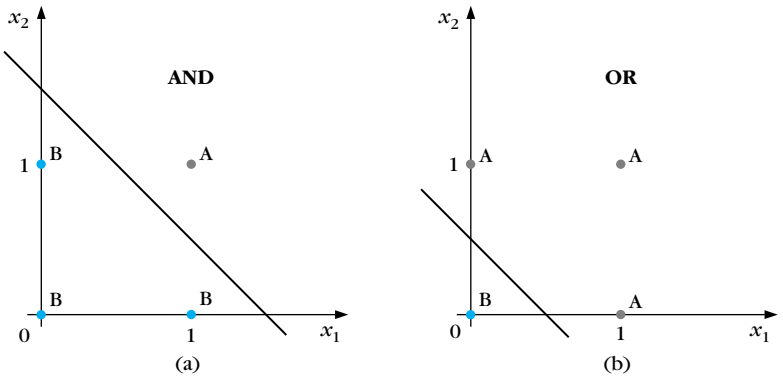

FIGURE 4.2

Classes A and B for (a) the AND and (b) OR problems.

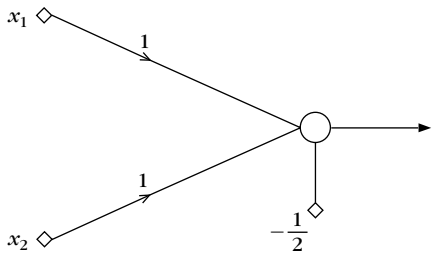

FIGURE 4.3

A perceptron realizing an OR gate.

4.3 THE TWO-LAYER PERCEPTRON

To separate the two classes A and B in Figure 4.1, a first thought that comes to mind is to draw two, instead of one, straight lines.

Figure 4.4 shows two such possible lines, $g_1(\boldsymbol{x}) = g_2(\boldsymbol{x}) = 0$, as well as the regions in space for which $g_1(\boldsymbol{x}) \gtrless 0, g_2(\boldsymbol{x}) \gtrless 0$. The classes can now be separated. Class A is to the right $(+)$ of $g_1(\boldsymbol{x})$ and to the left $(-)$ of $g_2(\boldsymbol{x})$. The region corresponding to class B lies either to the left or to the right of both lines. What we have really done is to attack the problem in *two* successive phases. During the first phase, we calculate the position of a feature vector $\boldsymbol{x}$ with respect to *each* of the two decision lines. In the second phase, we combine the results of the previous phase and we find the position of $\boldsymbol{x}$ with respect to *both* lines, that is, outside or inside the shaded area. We will now view this from a slightly different perspective, which will subsequently lead us easily to generalizations.

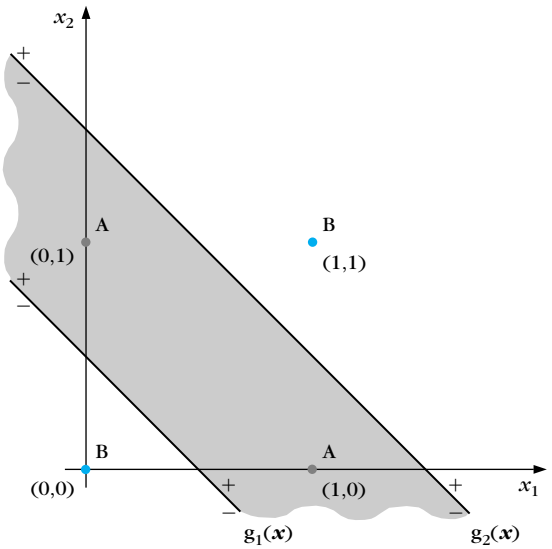

FIGURE 4.4

Decision lines realized by a two-layer perceptron for the XOR problem.

Table 4.3 Truth Table for the Two Computation Phases of the XOR Problem

1st Phase				
x_1	x_2	y_1	y_2	2nd Phase
0	0	0 (−)	0 (−)	B (0)
0	1	1 (+)	0 (−)	A (1)
1	0	1 (+)	0 (−)	A (1)
1	1	1 (+)	1 (+)	B (0)

Realization of the two decision lines (hyperplanes), $g_1(\cdot)$ and $g_2(\cdot)$, during the first phase of computations is achieved with the adoption of two perceptrons with inputs x_1, x_2 and appropriate synaptic weights. The corresponding outputs are $y_i = f(g_i(\boldsymbol{x})), i = 1, 2$, where the activation function $f(\cdot)$ is the step function with levels 0 and 1. Table 4.3 summarizes the y_i values for all possible combinations of the inputs. These are nothing else than the relative positions of the input vector $\boldsymbol{x}$ with respect to each of the two lines. From another point of view, the computations during the first phase *perform a mapping* of the input vector $\boldsymbol{x}$ to a new

one $\boldsymbol{y} = [y_1, y_2]^T$. The decision during the second phase is now based on the transformed data; that is, our goal is now to separate $[y_1, y_2] = [0, 0]$ and $[y_1, y_2] = [1, 1]$, which correspond to class B vectors, from the $[y_1, y_2] = [1, 0]$, which corresponds to class A vectors. As is apparent from Figure 4.5, this is easily achieved by drawing a third line $g(\boldsymbol{y})$, which can be realized via a third neuron. *In other words, the mapping of the first phase transforms the nonlinearly separable problem to a linearly separable one*. We will return to this important issue later on. Figure 4.6 gives a possible realization of these steps. Each of the three lines is realized via a neuron with appropriate synaptic weights. The resulting *multilayer* architecture can be considered as a generalization of the perceptron, and it is known as a *two-layer perceptron*

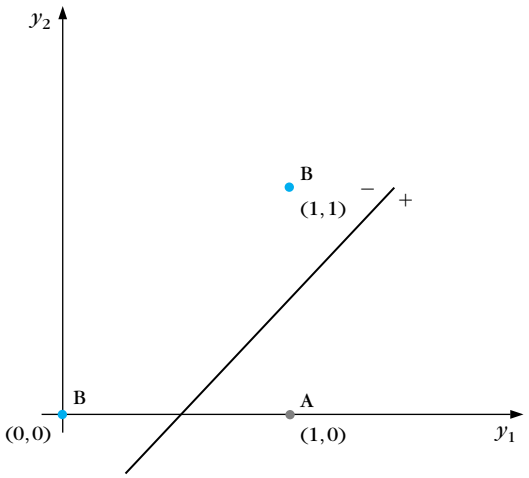

FIGURE 4.5

Decision line formed by the neuron of the second layer for the XOR problem.

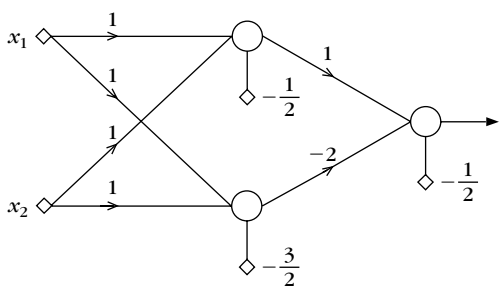

FIGURE 4.6

A two-layer perceptron solving the XOR problem.

or a *two-layer feedforward*[1] *neural network*. The two neurons (nodes) of the first layer perform computations of the first phase and they constitute the so-called *hidden layer*. The single neuron of the second layer performs the computations of the final phase and constitutes the *output layer*. In Figure 4.6 the *input layer* corresponds to the (nonprocessing) nodes where input data are applied. Thus, the number of input layer nodes equals the dimension of the input space. Note that at the input layer nodes no processing takes place. The lines that are realized by the two-layer perceptron of the figure are

$$g_1(\boldsymbol{x}) = x_1 + x_2 - \frac{1}{2} = 0$$

$$g_2(\boldsymbol{x}) = x_1 + x_2 - \frac{3}{2} = 0$$

$$g(\boldsymbol{y}) = y_1 - y_2 - \frac{1}{2} = 0$$

The multilayer perceptron architecture of Figure 4.6 can be generalized to l-dimensional input vectors and to more than two (one) neurons in the hidden (output) layer. We will now turn our attention to the investigation of the class discriminatory capabilities of such networks for more complicated nonlinear classification tasks.

4.3.1 Classification Capabilities of the Two-Layer Perceptron

A careful look at the two-layer perceptron of Figure 4.6 reveals that the action of the neurons of the hidden layer is actually a mapping of the input space $\boldsymbol{x}$ onto the vertices of a square of unit side length in the two-dimensional space (Figure 4.5).

For the more general case, we will consider input vectors in the l-dimensional space, that is, $\boldsymbol{x} \in \mathcal{R}^l$, and p neurons in the hidden layer (Figure 4.7). For the time being, we will keep one output neuron, although this can also be easily generalized to many. Again employing the step activation function, the mapping of the input space, performed by the hidden layer, is now onto the vertices of the hypercube of unit side length in the p-dimensional space, denoted by H_p. This is defined as

$$H_p = \{[y_1, \ldots, y_p]^T \in \mathcal{R}^p, y_i \in [0, 1], 1 \leq i \leq p\}$$

The vertices of the hypercube are all the points $[y_1, \ldots, y_p]^T$ of H_p with $y_i \in \{0, 1\}$, $1 \leq i \leq p$.

The mapping of the input space onto the vertices of the hypercube is achieved via the creation of p hyperplanes. Each of the hyperplanes is created by a neuron in the hidden layer, and the output of each neuron is 0 or 1, depending on the relevant position of the input vector with respect to the corresponding hyperplane.

[1] To distinguish it from other related structures where feedback paths from the output back to the input exist.

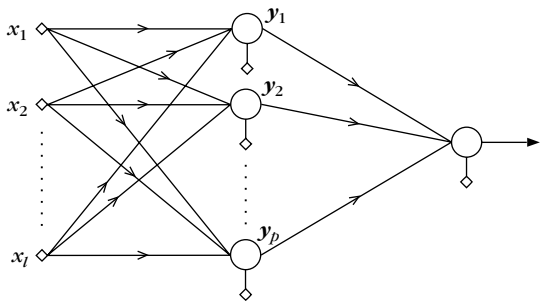

FIGURE 4.7

A two-layer perceptron.

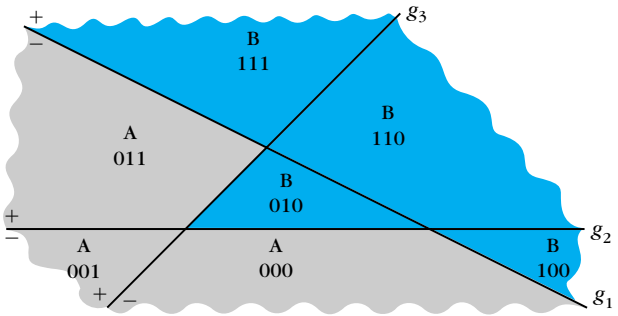

FIGURE 4.8

Polyhedra formed by the neurons of the first hidden layer of a multilayer perceptron.

Figure 4.8 is an example of three intersecting hyperplanes (three neurons) in the two-dimensional space. Each region defined by the intersections of these hyperplanes corresponds to a vertex of the unit three-dimensional hypercube, depending on its position with respect to each of these hyperplanes. The ith dimension of the vertex shows the position of the region with respect to the g_i hyperplane. For example, the 001 vertex corresponds to the region that is in the $(-)$ side of g_1, in the $(-)$ side of g_2, and in the $(+)$ side of g_3. Thus, the conclusion we reach is that *the first layer of neurons divides the input l-dimensional space into polyhedra,*[2] *which are formed by hyperplane intersections. All vectors located within one of these polyhedral regions are mapped onto a specific vertex of the unit H_p hypercube*. The output neuron subsequently realizes another hyperplane, which separates the hypercube into two parts, having some of its vertices on one and some

[2] A polyhedron or polyhedral set is the finite intersection of closed half-spaces of $\mathcal{R}^l$, which are defined by a number of hyperplanes.

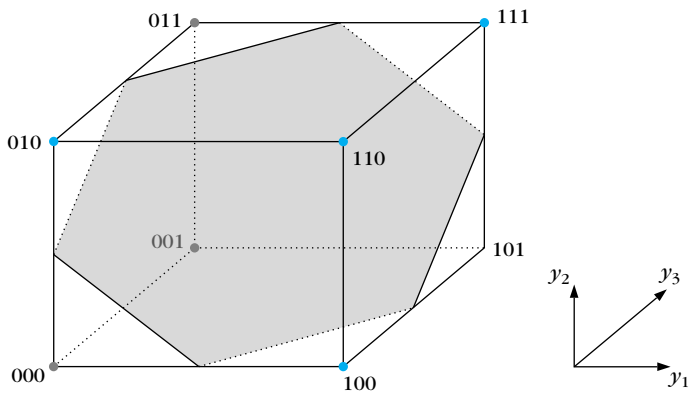

FIGURE 4.9

The neurons of the first hidden layer map an input vector onto one of the vertices of a unit (hyper)cube. The output neuron realizes a (hyper)plane to separate vertices according to their class label.

on the other side. This neuron provides the multilayer perceptron with the potential to classify vectors into *classes consisting of unions of the polyhedral regions*. Let us consider, for example, that class A consists of the union of the regions mapped onto vertices $000, 001, 011$ and class B consists of the rest (Figure 4.8). Figure 4.9 shows the H_3 unit (hyper)cube and a (hyper)plane that separates the space $\mathcal{R}^3$ into two regions with the (class A) vertices $000, 001, 011$ on one side and (class B) vertices $010, 100, 110, 111$ on the other. This is the $-y_1 - y_2 + y_3 + 0.5 = 0$ plane, which is realized by the output neuron. With such a configuration all vectors from class A result in an output of $1(+)$ and all vectors from class B in $0(-)$. On the other hand, if class A consists of the union $000 \cup 111 \cup 110$ and class B of the rest, it is not possible to construct a single plane that separates class A from class B vertices. Thus, we can conclude that *a two-layer perceptron can separate classes each consisting of unions of polyhedral regions but not any union of such regions*. It all depends on the relative positions of the vertices of H_p, where the classes are mapped, and on whether or not these are linearly separable. Before we proceed further to see ways to overcome this shortcoming, it should be pointed out that vertex 101 of the cube does not correspond to any of the polyhedral regions. Such vertices are said to correspond to *virtual polyhedra*, and they do not influence the classification task.

4.4 THREE-LAYER PERCEPTRONS

The inability of the two-layer perceptrons to separate classes resulting from *any* union of polyhedral regions springs from the fact that the output neuron can realize

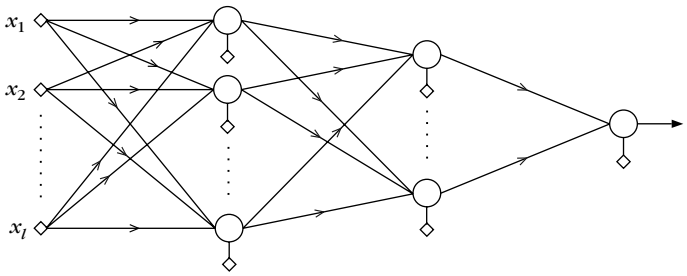

FIGURE 4.10

Architecture of a multilayer perceptron with two hidden layers of neurons and a single output neuron.

only a single hyperplane. This is the same situation confronting the basic perceptron when dealing with the XOR problem. The difficulty was overcome by constructing two lines instead of one. A similar escape path will be adopted here.

Figure 4.10 shows a three-layer perceptron architecture with two layers of hidden neurons and one output layer. We will show, constructively, that such an architecture can *separate classes resulting from any union of polyhedral regions*. Indeed, let us assume that all regions of interest are formed by intersections of p l-dimensional half-spaces defined by the p hyperplanes. These are realized by the p neurons of the first hidden layer, which also perform the mapping of the input space onto the vertices of the H_p hypercube of unit side length. In the sequel let us assume that class A consists of the union of K of the resulting polyhedra and class B of the rest. We then use K neurons in the second hidden layer. Each of these neurons realizes a hyperplane in the p-dimensional space. The synaptic weights for each of the second-layer neurons are chosen so that the realized hyperplane leaves only one of the H_p vertices on one side and *all* the rest on the other. For each neuron a different vertex is isolated, that is, one of the K A class vertices. In other words, each time an input vector from class A enters the network, one of the K neurons of the second layer results in a 1 and the remaining $K - 1$ give 0. In contrast, for class B vectors all neurons in the second layer output a 0. Classification is now a straightforward task. Choose the output layer neuron to realize an OR gate. Its output will be 1 for class A and 0 for class B vectors. The proof is now complete.

The number of neurons in the second hidden layer can be reduced by exploiting the geometry that results from each specific problem—for example, whenever two of the K vertices are located in a way that makes them separable from the rest, using a single hyperplane. Finally, the multilayer structure can be generalized to more than two classes. To this end, the output layer neurons are increased in number, realizing one OR gate for each class. Thus, one of them results in 1 every time a vector from the respective class enters the network, and all the others give 0. The number of second-layer neurons is also affected (why?).

In summary, we can say that *the neurons of the first layer form the hyperplanes, those of the second layer form the regions, and finally the neurons of the output layer form the classes*.

So far, we have focused on the potential capabilities of a three-layer perceptron to separate any union of polyhedral regions. To assume that in practice we know the regions where the data are located and we can compute the respective hyperplane equations analytically is no doubt wishful thinking, for this is as yet an unrealizable goal. All we know in practice is a set of training points with the respective class labels. As was the case with the perceptron, one has to resort to learning algorithms that learn the synaptic weights from the available training data vectors. We will focus our attention on two major directions. In one of them the network is constructed in a way that classifies correctly *all* the available training data, by building it as a succession of linear classifiers. The other direction relieves itself of the correct classification constraint and computes the synaptic weights so as to minimize a preselected cost function.

4.5 ALGORITHMS BASED ON EXACT CLASSIFICATION OF THE TRAINING SET

The starting point of these techniques is a small architecture (usually unable to solve the problem at hand), which is successively augmented until the correct classification of all N feature vectors of the training set X is achieved. Different algorithms follow different ways to augment their architectures. Thus, some algorithms expand their architectures in terms of the number of layers [Meza 89, Frea 90], whereas others use one or two hidden layers and expand them in terms of the number of their nodes (neurons) [Kout 94, Bose 96]. Moreover, some of these algorithms [Frea 90] allow connections between nodes of nonsuccessive layers. Others allow connections between nodes of the same layer [Refe 91]. A general principle adopted by most of these techniques is the decomposition of the problem into smaller problems that are easier to handle. For each smaller problem, a single node is employed. Its parameters are determined either iteratively using appropriate learning algorithms, such as the pocket algorithm or the LMS algorithm (Chapter 3), or directly via analytical computations. From the way these algorithms build the network, they are sometimes referred to as *constructive techniques*.

The *tiling algorithm* [Meza 89] constructs architectures with many (usually more than three) layers. We describe the algorithm for the two-class (A and B) case. The algorithm starts with a single node, $n(X)$, in the first layer, which is called the *master unit* of this layer.

This node is trained using the pocket algorithm (Chapter 3), and, after the completion of the training, it divides the training data set X into two subsets X^+ and X^- (line 1 in Figure 4.11). If X^+ (X^-) contains feature vectors from both classes, we introduce an additional node, $n(X^+)$ ($n(X^-)$), which is called the *ancillary*

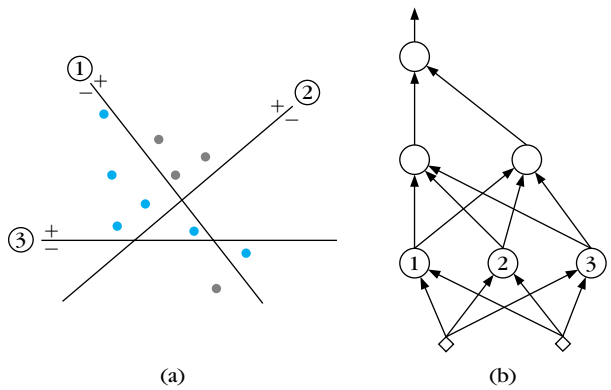

FIGURE 4.11

Decision lines and the corresponding architecture resulting from the tiling algorithm. The black (red) dots correspond to class A (B).

unit. This node is trained using only the feature vectors in X^+ (X^-) (line 2). If one of the X^{++}, X^{+-} (X^{-+}, X^{--}) produced by neuron $n(X^+)$ ($n(X^-)$) contains vectors from both classes, more ancillary nodes are added. This procedure stops after a finite number of steps, since the number of vectors a newly added (ancillary) unit has to discriminate decreases at each step. Thus, the first layer consists of a single master unit and, in general, more than one ancillary units. It is easy to show that in this way we succeed so that no two vectors from *different* classes give the same first-layer outputs.

Let $X_1 = \{\boldsymbol{y} : \boldsymbol{y} = f_1(\boldsymbol{x}), \boldsymbol{x} \in X\}$, where f_1 is the mapping implemented by the first layer. Applying the procedure just described to the set X_1 of the transformed $\boldsymbol{y}$ samples, we construct the second layer of the architecture and so on. In [Meza 89] it is shown that proper choice of the weights between two adjacent layers ensures that each newly added master unit classifies correctly all the vectors that are correctly classified by the master unit of the previous layer, plus at least one more vector. Thus, the tiling algorithm produces an architecture that classifies correctly all patterns of X in a finite number of steps.

An interesting observation is that all but the first layer treat binary vectors. This reminds us of the unit hypercube of the previous section. Mobilizing the same arguments as before, we can show that this algorithm may lead to correct classification architectures having three layers of nodes at the most.

Another family of constructive algorithms builds on the idea of the nearest neighbor classification rule, discussed in Chapter 2. The neurons of the first layer implement the hyperplanes bisecting the line segments that join the training feature vectors [Murp 90]. The second layer forms the regions, using an appropriate number of neurons that implement AND gates, and the classes are formed via the neurons of the last layer, which implement OR gates. The major drawback of this

technique is the large number of neurons involved. Techniques that reduce this number have also been proposed ([Kout 94, Bose 96]).

4.6 THE BACKPROPAGATION ALGORITHM

The other direction we will follow to design a multilayer perceptron is to fix the architecture and compute its synaptic parameters so as to minimize an appropriate cost function of its output. This is by far the most popular approach, which not only overcomes the drawback of the resulting large networks of the previous section but also makes these networks powerful tools for a number of other applications, beyond pattern recognition. However, such an approach is soon confronted with a serious difficulty. This is the discontinuity of the step (activation) function, prohibiting differentiation with respect to the unknown parameters (synaptic weights). Differentiation enters into the scene as a result of the cost function minimization procedure. In the sequel we will see how this difficulty can be overcome.

The multilayer perceptron architectures we have considered so far have been developed around the McCulloch–Pitts neuron, employing as the activation function the step function

$$f(x) = \begin{cases} 1 & x > 0 \\ 0 & x < 0 \end{cases}$$

A popular family of continuous differentiable functions, which approximate the step function, is the family of *sigmoid functions*. A typical representative is the *logistic function*

$$f(x) = \frac{1}{1 + \exp(-ax)} \tag{4.1}$$

where a is a slope parameter.

Figure 4.12 shows the sigmoid function for different values of a, along with the step function. Sometimes a variation of the logistic function is employed that is antisymmetric with respect to the origin, that is, $f(-x) = -f(x)$. It is defined as

$$f(x) = \frac{2}{1 + \exp(-ax)} - 1 \tag{4.2}$$

It varies between 1 and -1, and it belongs to the family of hyperbolic tangent functions,

$$f(x) = c\frac{1 - \exp(-ax)}{1 + \exp(-ax)} = c\tanh\left(\frac{ax}{2}\right) \tag{4.3}$$

All these functions are also known as *squashing functions* since their output is limited in a finite range of values. In the sequel, we will adopt multilayer neural architectures like the one in Figure 4.10, and we will assume that the activation

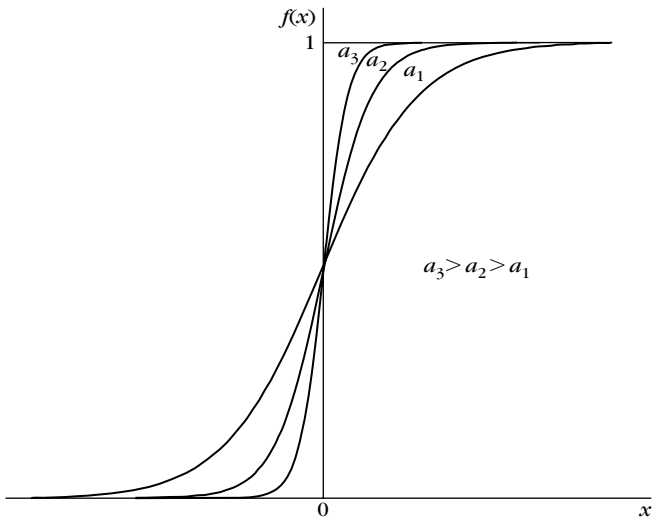

FIGURE 4.12

The logistic function. The larger the value of the slope parameter, a, the better the approximation of the unit-step function achieved.

functions are of the form given in (4.1)–(4.3). Our goal is to derive an iterative training algorithm that computes the synaptic weights of the network so that an appropriately chosen cost function is minimized. Before going into the derivation of such a scheme, an important point must be clarified. From the moment we move away from the step function, all we have said before about mapping the input vectors onto the vertices of a unit hypercube is no longer valid. It is now the cost function that takes on the burden for correct classification.

For the sake of generalization, let us assume that the network consists of a fixed number of L layers of neurons, with k_0 nodes in the input layer and k_r neurons in the rth layer, for $r = 1, 2, \ldots, L$. Obviously, k_0 equals l. All the neurons employ the same sigmoid activation function. As was the case in Section 3.3, we assume that N training pairs are available $(\boldsymbol{y}(i), \boldsymbol{x}(i)), i = 1, 2, \ldots, N$.[3] Because we have now assumed k_L output neurons, the output is no longer a scalar but a k_L-dimensional vector, $\boldsymbol{y}(i) = [y_1(i), \ldots, y_{k_L}(i)]^T$. The input (feature) vectors are k_0-dimensional vectors, $\boldsymbol{x}(i) = [x_1(i), \ldots, x_{k_0}(i)]^T$. During training, when vector $\boldsymbol{x}(i)$ is applied to the input, the output of the network will be $\hat{\boldsymbol{y}}(i)$, which is different from the desired value, $\boldsymbol{y}(i)$. *The synaptic weights are computed such that an appropriate (for each problem) cost function J, which is dependent on the values $\boldsymbol{y}(i)$ and*

[3] In contrast to other chapters, we use i in parentheses and not as an index. This is because, for the needs of the chapter, the latter notation can become very cumbersome.

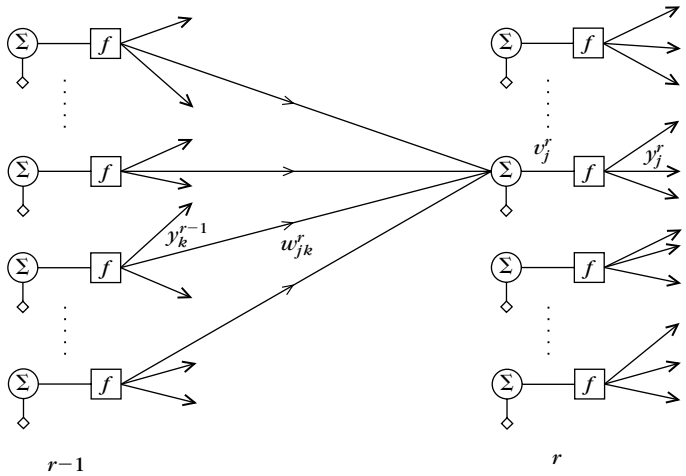

FIGURE 4.13

Definition of variables involved in the backpropagation algorithm.

$\hat{\boldsymbol{y}}(i), i = 1, 2, \ldots, N$, *is minimized*. It is obvious that J depends, through $\hat{\boldsymbol{y}}(i)$, on the weights and that this is a nonlinear dependence, due to the nature of the network itself. Thus, minimization of the cost function can be achieved via iterative techniques. In this section we will adopt the gradient descent scheme (Appendix C), which is the most widely used approach. Let $\boldsymbol{w}_j^r$ be the weight vector (including the threshold) of the jth neuron in the rth layer, which is a vector of dimension $k_{r-1} + 1$ and is defined as (Figure 4.13) $\boldsymbol{w}_j^r = [w_{j0}^r, w_{j1}^r, \ldots, w_{jk_{r-1}}^r]^T$. The basic iteration step will be of the form

$$\boldsymbol{w}_j^r(\text{new}) = \boldsymbol{w}_j^r(\text{old}) + \Delta \boldsymbol{w}_j^r$$

with

$$\Delta \boldsymbol{w}_j^r = -\mu \frac{\partial J}{\partial \boldsymbol{w}_j^r} \tag{4.4}$$

where $\boldsymbol{w}_j^r(\text{old})$ is the current estimate of the unknown weights and $\Delta \boldsymbol{w}_j^r$ the corresponding correction to obtain the next estimate $\boldsymbol{w}_j^r(\text{new})$.

In Figure 4.13 v_j^r is the weighted summation of the inputs to the jth neuron of the rth layer and y_j^r the corresponding output after the activation function. In the sequel we will focus our attention on cost functions of the form

$$J = \sum_{i=1}^{N} \mathcal{E}(i) \tag{4.5}$$

where $\mathcal{E}$ is an appropriately defined function depending on $\hat{\boldsymbol{y}}(i)$ and $\boldsymbol{y}(i), i = 1, 2, \ldots, N$. In other words, J is expressed as a sum of the N values that function $\mathcal{E}$

takes for each of the training pairs $(\boldsymbol{y}(i), \boldsymbol{x}(i))$. For example, we can choose $\mathcal{E}(i)$ as the sum of squared errors in the output neurons

$$\mathcal{E}(i) = \frac{1}{2}\sum_{m=1}^{k_L} e_m^2(i) \equiv \frac{1}{2}\sum_{m=1}^{k_L}(y_m(i) - \hat{y}_m(i))^2, \quad i = 1, 2, \ldots, N \tag{4.6}$$

For the computation of the correction term in (4.4) the gradient of the cost function J with respect to the weights is required and, consequently, the evaluation of $\partial\mathcal{E}(i)/\partial\boldsymbol{w}_j^r$.

Computation of the Gradients

Let $y_k^{r-1}(i)$ be the output of the kth neuron, $k = 1, 2, \ldots, k_{r-1}$, in the $(r-1)$th layer for the ith training pair and w_{jk}^r the current estimate of the corresponding weight leading to the jth neuron in the rth layer, with $j = 1, 2, \ldots, k_r$ (Figure 4.13). Thus, the argument of the activation function $f(\cdot)$ of the latter neuron will be

$$v_j^r(i) = \sum_{k=1}^{k_{r-1}} w_{jk}^r\, y_k^{r-1}(i) + w_{jo}^r \equiv \sum_{k=0}^{k_{r-1}} w_{jk}^r\, y_k^{r-1}(i) \tag{4.7}$$

where by definition $y_0^r(i) \equiv +1, \forall r, i$; so as to include the thresholds in the weights. For the output layer, we have $r = L, y_k^r(i) = \hat{y}_k(i), k = 1, 2, \ldots, k_L$, that is, the outputs of the neural network, and for $r = 1, y_k^{r-1}(i) = x_k(i), k = 1, 2, \ldots, k_0$, that is, the network inputs.

As is apparent from (4.7), the dependence of $\mathcal{E}(i)$ on $\boldsymbol{w}_j^r$ passes through $v_j^r(i)$. By the chain rule in differentiation, we have

$$\frac{\partial\mathcal{E}(i)}{\partial\boldsymbol{w}_j^r} = \frac{\partial\mathcal{E}(i)}{\partial v_j^r(i)}\frac{\partial v_j^r(i)}{\partial\boldsymbol{w}_j^r} \tag{4.8}$$

From (4.7) we obtain

$$\frac{\partial}{\partial\boldsymbol{w}_j^r}v_j^r(i) \equiv \begin{bmatrix} \frac{\partial}{\partial w_{j0}^r}v_j^r(i) \\ \vdots \\ \frac{\partial}{\partial w_{jk_{r-1}}^r}v_j^r(i) \end{bmatrix} = \boldsymbol{y}^{r-1}(i) \tag{4.9}$$

where

$$\boldsymbol{y}^{r-1}(i) = \begin{bmatrix} +1 \\ y_1^{r-1}(i) \\ \vdots \\ y_{k_{r-1}}^{r-1}(i) \end{bmatrix} \tag{4.10}$$

Let us define

$$\frac{\partial \mathcal{E}(i)}{\partial v_j^r(i)} \equiv \delta_j^r(i) \tag{4.11}$$

Then (4.4) becomes

$$\Delta \boldsymbol{w}_j^r = -\mu \sum_{i=1}^{N} \delta_j^r(i) \boldsymbol{y}^{r-1}(i) \tag{4.12}$$

Relation (4.12) is general for *any* differentiable cost function of the form (4.5). In the sequel we will compute $\delta_j^r(i)$ for the special case of least squares (4.6). *The procedure is similar for alternative cost function choices*.

Computation of $\delta_j^r(i)$ for the Cost Function in (4.6)

The computations start from $r = L$ and *propagate backward* for $r = L-1$, $L-2, \ldots, 1$. This is why the algorithm that will be derived is known as *the backpropagation algorithm*.

1. $r = L$

$$\delta_j^L(i) = \frac{\partial \mathcal{E}(i)}{\partial v_j^L(i)} \tag{4.13}$$

$$\mathcal{E}(i) \equiv \frac{1}{2} \sum_{m=1}^{k_L} e_m^2(i) \equiv \frac{1}{2} \sum_{m=1}^{k_L} (f(v_m^L(i)) - y_m(i))^2 \tag{4.14}$$

Hence

$$\delta_j^L(i) = e_j(i) f'(v_j^L(i)) \tag{4.15}$$

where f' is the derivative of $f(\cdot)$. In the last layer, the dependence of $\mathcal{E}(i)$ on $v_j^L(i)$ is explicit, and the computation of the derivative is straightforward. This is not true, however, for the hidden layers, where the computations of the derivatives need more elaboration.

2. $r < L$. Due to the successive dependence among the layers, the value of $v_j^{r-1}(i)$ influences all $v_k^r(i), k = 1, 2, \ldots, k_r$, of the next layer. Employing the chain rule in differentiation once more, we obtain

$$\frac{\partial \mathcal{E}(i)}{\partial v_j^{r-1}(i)} = \sum_{k=1}^{k_r} \frac{\partial \mathcal{E}(i)}{\partial v_k^r(i)} \frac{\partial v_k^r(i)}{\partial v_j^{r-1}(i)} \tag{4.16}$$

and from the respective definition (4.11)

$$\delta_j^{r-1}(i) = \sum_{k=1}^{k_r} \delta_k^r(i) \frac{\partial v_k^r(i)}{\partial v_j^{r-1}(i)} \tag{4.17}$$

But

$$\frac{\partial v_k^r(i)}{\partial v_j^{r-1}(i)} = \frac{\partial \left[\sum_{m=0}^{k_{r-1}} w_{km}^r y_m^{r-1}(i)\right]}{\partial v_j^{r-1}(i)} \tag{4.18}$$

with

$$y_m^{r-1}(i) = f(v_m^{r-1}(i)) \tag{4.19}$$

Hence,

$$\frac{\partial v_k^r(i)}{\partial v_j^{r-1}(i)} = w_{kj}^r f'(v_j^{r-1}(i)) \tag{4.20}$$

From (4.20) and (4.17) the following results:

$$\delta_j^{r-1}(i) = \left[\sum_{k=1}^{k_r} \delta_k^r(i) w_{kj}^r\right] f'(v_j^{r-1}(i)) \tag{4.21}$$

and for uniformity with (4.15)

$$\delta_j^{r-1}(i) = e_j^{r-1}(i) f'(v_j^{r-1}(i)) \tag{4.22}$$

where

$$e_j^{r-1}(i) = \sum_{k=1}^{k_r} \delta_k^r(i) w_{kj}^r \tag{4.23}$$

Relations (4.15), (4.22), and (4.23) constitute the iterations leading to the computation of $\delta_j^r(i), r = 1, 2, \ldots, L, j = 1, 2, \ldots, k_r$. The only quantity that is not yet computed is $f'(\cdot)$. For the function in (4.1) we have

$$f'(x) = af(x)(1 - f(x))$$

The algorithm has now been derived. The algorithmic scheme was first presented in [Werb 74] in a more general formulation.

The Backpropagation Algorithm

- *Initialization:* Initialize all the weights with small random values from a pseudorandom sequence generator.
- *Forward computations:* For each of the training feature vectors $\boldsymbol{x}(i), i = 1, 2, \ldots, N$, compute all the $v_j^r(i)$, $y_j^r(i) = f(v_j^r(i)), j = 1, 2, \ldots, k_r$, $r = 1, 2, \ldots, L$, from (4.7). Compute the cost function for the current estimate of weights from (4.5) and (4.14).

- *Backward computations:* For each $i = 1, 2, \ldots, N$ and $j = 1, 2, \ldots, k_L$ compute $\delta_j^L(i)$ from (4.15) and in the sequel compute $\delta_j^{r-1}(i)$ from (4.22) and (4.23) for $r = L, L-1, \ldots, 2$, and $j = 1, 2, \ldots, k_r$
- *Update the weights:* For $r = 1, 2, \ldots, L$ and $j = 1, 2, \ldots, k_r$

$$\boldsymbol{w}_j^r(\text{new}) = \boldsymbol{w}_j^r(\text{old}) + \Delta \boldsymbol{w}_j^r$$

$$\Delta \boldsymbol{w}_j^r = -\mu \sum_{i=1}^{N} \delta_j^r(i) \boldsymbol{y}^{r-1}(i)$$

Remarks

- A number of criteria have been suggested for terminating the iterations. In [Kram 89] it is suggested that we terminate the iterations either when the cost function J becomes smaller than a certain threshold or when its gradient with respect to the weights becomes small. Of course, the latter has a direct effect on the rate of change of the weights between successive iteration steps.
- As with all the algorithms that spring from the gradient descent method, the convergence speed of the backpropagation scheme depends on the value of the learning constant μ. Its value must be sufficiently small to guarantee convergence but not too small, because the convergence speed becomes very slow. The best choice of μ depends very much on the problem and the cost function shape in the weight space. Broad minima yield small gradients; thus large values of μ lead to faster convergence. On the other hand, for steep and narrow minima small values of μ are required to avoid overshooting the minimum. As we will soon see, scenarios with adaptive μ are also possible.
- The cost function minimization for a multilayer perceptron is a nonlinear minimization task. Thus, the existence of local minima in the corresponding cost function surface is an expected reality. Hence, the backpropagation algorithm runs the risk of being trapped in a local minimum. If the local minimum is deep enough, this may still be a good solution. However, in cases in which this is not true, getting stuck in such a minimum is an undesirable situation, and the algorithm should be reinitialized from a different set of initial conditions.
- The algorithm described in this section updates the weights once *all* the training (input–desired output) pairs have appeared in the network. This mode of operation is known as the *batch mode*. A variation of this approach is to update the weights for each of the training pairs. This is known as the *pattern* or *online mode*. This is analogous to the LMS, where, according to

the Robbins–Monro approach, the instantaneous value of the gradient is computed instead of its mean. In the backpropagation case, the sum of $\delta_j^r(i)$ over all i is substituted with *each* of them. The algorithm in its pattern mode of operation then becomes

$$\boldsymbol{w}_j^r(i+1) = \boldsymbol{w}_j^r(i) - \mu\delta_j^r(i)\boldsymbol{y}^{r-1}(i)$$

Compared with the pattern mode, the batch mode is an inherent averaging process. This leads to a better estimate of the gradient, and thus to more well-behaved convergence. On the other hand, the pattern mode presents a higher degree of randomness during training. This may help the algorithm to avoid being trapped in a local minimum. In [Siet 91] it is suggested that the beneficial effects that randomness may have on training can be further emphasized by adding a (small) white noise sequence in the training data. Another commonly used practice focuses on the way the training data are presented in the network. *During training, the available training vectors are used in the update equation more than once* until the algorithm converges. One complete presentation of all N training pairs constitutes an *epoch*. As successive epochs are applied, it is good practice from the convergence point of view to randomize the order of presentation of the training pairs. Randomization can again help the pattern mode algorithm to jump out of regions around local minima, when this occurs. However, the final choice between the batch and pattern modes of operation depends on the specific problem [Hert 91, p. 119].

- Once training of the network has been achieved, the values to which the synapses and thresholds have converged are frozen, and the network is ready for classification. This is a much easier task than training. An unknown feature vector is presented in the input and is classified in the class that is indicated by the output of the network. The computations performed by the neurons are of the multiply–add type followed by a nonlinearity. This has led to various hardware implementations ranging from optical to VLSI chip design. Furthermore, neural networks have a natural built-in parallelism, and computations in each layer can be performed in parallel. These distinct characteristics of neural networks have led to the development of special *neurocomputers*. A number of those are already commercially available; see, for example, [Koli 97].

4.7 VARIATIONS ON THE BACKPROPAGATION THEME

Both versions of the backpropagation scheme—the batch and the pattern modes—inherit the disadvantage of all methods built on the gradient descent approach: *their convergence to the cost function minimum is slow*. Appendix C discusses the fact that this trait becomes more prominent if the eigenvalues of the corresponding

Hessian matrix exhibit large spread. In such cases, the change of the cost function gradient between successive iteration steps is not smooth but oscillatory, leading to slow convergence. One way to overcome this problem is to use a *momentum term* that smoothes out the oscillatory behavior and speeds up the convergence. The backpropagation algorithm with momentum term takes the form

$$\Delta \boldsymbol{w}_j^r(\text{new}) = \alpha \Delta \boldsymbol{w}_j^r(\text{old}) - \mu \sum_i^N \delta_j^r(i) \boldsymbol{y}^{r-1}(i) \tag{4.24}$$

$$\boldsymbol{w}_j^r(\text{new}) = \boldsymbol{w}_j^r(\text{old}) + \Delta \boldsymbol{w}_j^r(\text{new}) \tag{4.25}$$

Compared with (4.4), we see that the correction vector $\Delta \boldsymbol{w}_j^r$ depends not only on the gradient term but also on its value in the previous iteration step. The constant α is called the *momentum factor* and in practice is chosen between 0.1 and 0.8. To see the effect of the momentum factor, let us look at the correction term for a number of successive iteration steps. At the tth iteration step we have

$$\Delta \boldsymbol{w}_j^r(t) = \alpha \Delta \boldsymbol{w}_j^r(t-1) - \mu \boldsymbol{g}(t) \tag{4.26}$$

where the last term denotes the gradient. For a total of T successive iteration steps we obtain

$$\Delta \boldsymbol{w}_j^r(T) = -\mu \sum_{t=0}^{T-1} \alpha^t \boldsymbol{g}(T-t) + \alpha^T \Delta \boldsymbol{w}_j^r(0) \tag{4.27}$$

Since $\alpha < 1$, the last term gets close to zero after a few iteration steps and the smoothing (averaging) effect of the momentum term becomes apparent. Let us now assume that the algorithm is at a low-curvature point of the cost function surface in the weight space. We can then assume that the gradient is approximately constant over a number of iteration steps. Applying this, we can write that

$$\Delta \boldsymbol{w}_j^r(T) \simeq -\mu(1 + \alpha + \alpha^2 + \alpha^3 + \cdots)\boldsymbol{g} = -\frac{\mu}{1-\alpha}\boldsymbol{g}$$

In other words, in such cases the effect of the momentum term is to effectively increase the learning constant. In practice, improvements in converging speed by a factor of 2 or even more have been reported [Silv 90].

A heuristic variation of the previous technique is to use an adaptive value for the learning factor μ, depending on the cost function values at successive iteration steps. A possible procedure is the following: Let $J(t)$ be the value of the cost at the tth iteration step. If $J(t) < J(t-1)$, then increase the learning rate by a factor of r_i. If, on the other hand, the new value of the cost is larger than the old one by a factor c, then decrease the learning rate by a factor of r_d. Otherwise use the same value. In summary

$$\frac{J(t)}{J(t-1)} < 1, \quad \mu(t) = r_i \mu(t-1)$$

$$\frac{J(t)}{J(t-1)} > c, \quad \mu(t) = r_d \mu(t-1)$$

$$1 \leq \frac{J(t)}{J(t-1)} \leq c, \quad \mu(t) = \mu(t-1)$$

Typical values of the parameters which are adopted in practice are $r_i = 1.05, r_d = 0.7, c = 1.04$. For iteration steps where the cost increases, it may be advantageous not only to decrease the learning rate but also to set the momentum term equal to 0. Others suggest that the update of the weighting not to be done at this step.

Another strategy for updating the learning factor μ is followed in the so-called *delta-delta* rule and in its modification *delta-bar-delta* rule [Jaco 88]. The idea here is to use a different learning factor for each weight and to increase the particular learning factor if the gradient of the cost function with respect to the corresponding weight has the same sign on two successive iteration steps. Conversely, if the sign changes, this is an indication of a possible oscillation and the learning factor should be reduced. A number of alternative techniques for speeding up convergence have also been suggested. In [Cich 93] a more extensive review of such techniques is provided.

The other option for faster convergence is to free ourselves from the gradient descent rationale and to adopt alternative searching schemes, usually at the expense of increased complexity. A number of such algorithmic techniques have appeared in the related literature. For example, [Kram 89, Barn 92, Joha 92] present algorithmic schemes based on the conjugate gradient algorithm; [Batt 92, Rico 88, Barn 92, Watr 88] provide schemes of the Newton family; [Palm 91, Sing 89] propose algorithms based on the Kalman filtering approach; and [Bish 95] a scheme based on the Levenberg–Marquardt algorithm. In many of these algorithms, elements of the Hessian matrix need to be computed, that is, the second derivatives of the cost function with respect to the weights

$$\frac{\partial^2 J}{\partial w_{jk}^q \partial w_{nm}^r}$$

The computations of the Hessian matrix are performed by adopting, once more, the backpropagation concept (see also Problems 4.12, 4.13). More on these issues can be found in [Hayk 99, Zura 92].

A popular scheme that is loosely based on Newton's method is the *quickprop* scheme [Fahl 90]. It is a heuristic method and treats the weights as if they were quasi-independent. It then approximates the error surface, as a function of each weight, by a quadratic polynomial. If this has its minimum at a sensible value, the latter is used as the new weight for the iterations; otherwise a number of heuristics are used. A usual form of the algorithm, for the weights in the various layers, is

$$\Delta w_{ij}(t) = \begin{cases} \alpha_{ij}(t) \Delta w_{ij}(t-1), & \text{if } \Delta w_{ij}(t-1) \neq 0 \\ \mu \frac{\partial J}{\partial w_{ij}}, & \text{if } \Delta w_{ij}(t-1) = 0 \end{cases} \tag{4.28}$$

where

$$\alpha_{ij}(t) = \min\left\{ \frac{\frac{\partial J}{\partial w_{ij}}(t)}{\frac{\partial J}{\partial w_{ij}}(t-1) - \frac{\partial J}{\partial w_{ij}}(t)}, \alpha_{\max} \right\} \tag{4.29}$$

with typical values of the involved variables being $0.01 \leq \mu \leq 0.6, \alpha_{\max} \approx 1.75$ [Cich 93]. An algorithm similar in spirit to quickprop has been proposed in [Ried 93]. It is reported that it is as fast as quickprop, and it requires less adjustment of the parameters to be stable.

4.8 THE COST FUNCTION CHOICE

It will not come as a surprise that the least squares cost function in (4.6) is not the unique choice available to the user. Depending on the specific problem, other cost functions can lead to better results. Let us look, for example, at the least squares cost function more carefully. Since all errors in the output nodes are first squared and summed up, large error values influence the learning process much more than the small errors. Thus, if the dynamic ranges of the desired outputs are not *all* of the same order, the least squares criterion will result in weights that have "learned" via a process of unfair provision of information. Furthermore, in [Witt 00] it is shown that for a class of problems, the gradient descent algorithm with the squared error criterion can be trapped in a local minimum and fail to find a solution, although (at least) one exists. In the current context, a solution is assumed to be a classifier that classifies correctly all training samples. In contrast, it is shown that there is an alternative class of functions, satisfying certain criteria, which guarantee that the gradient descent algorithm converges to such a solution, provided that one exists. This class of cost functions is known as *well-formed functions*. We will now present a cost function of this type, which is well suited for pattern recognition tasks.

The multilayer network performs a nonlinear mapping of the input vectors $\boldsymbol{x}$ to the output values $\hat{y}_k = \phi_k(\boldsymbol{x}; \boldsymbol{w})$ for each of the output nodes $k = 1, 2, \ldots k_L$, where the dependence of the mapping on the values of the weights is explicitly shown. In Chapter 3 we have seen that, if we adopt the least squares cost function and the desired outputs y_k are binary (belong to or not in class ω_k), then for the optimal values of the weights $\boldsymbol{w}^*$ the corresponding output of the network, $\hat{y}_k$, is *the least squares optimal estimate of the posterior probability* $P(\omega_k|\boldsymbol{x})$. (The question of how good or bad this estimate is will be of interest to us soon.) At this point we will adopt this probabilistic interpretation of the real outputs $\hat{y}_k$ as the basis on which our cost function will be built. Let us assume that the desired output values, y_k, are independent binary random variables and that $\hat{y}_k$ are the respective posterior probabilities that these random variables are 1 [Hint 90, Baum 88].

The *cross-entropy* cost function is then defined by

$$J = -\sum_{i=1}^{N}\sum_{k=1}^{k_L}(y_k(i)\ln\hat{y}_k(i) + (1-y_k(i))\ln(1-\hat{y}_k(i))) \tag{4.30}$$

J takes its minimum value when $y_k(i) = \hat{y}_k(i)$, and for binary desired response values the minimum is zero. There are various interpretations of this cost function [Hint 90, Baum 88, Gish 90, Rich 91]. Let us consider, for example, the output vector $\boldsymbol{y}(i)$ when $\boldsymbol{x}(i)$ appears at the input. This consists of a 1 at the true class node and zero elsewhere. If we take into account that the probability of the kth node to be 1(0) is $\hat{y}_k(i)(1-\hat{y}_k(i))$ and by considering nodes *independently*, then

$$p(\boldsymbol{y}) = \prod_{k=1}^{k_L}(\hat{y}_k)^{y_k}(1-\hat{y}_k)^{1-y_k} \tag{4.31}$$

where the dependence on i has been suppressed for notational convenience. Then it is straightforward to check that J results from the negative log-likelihood of the training sample pairs. If $y_k(i)$ were true probabilities in (0, 1) then subtracting the minimum value from J (4.30) becomes

$$J = -\sum_{i=1}^{N}\sum_{k=1}^{k_L}\left(y_k(i)\ln\frac{\hat{y}_k(i)}{y_k(i)} + (1-y_k(i))\ln\frac{1-\hat{y}_k(i)}{1-y_k(i)}\right) \tag{4.32}$$

For binary valued y_ks the above is still valid if we use the limiting value $0\ln 0 = 0$.

It is not difficult to show (Problem 4.5) that the *cross-entropy cost function depends on the relative errors and not on the absolute errors, as its least squares counterpart; thus it gives the same weight to small and large values*. Furthermore, it has been shown that it satisfies the conditions of the well-formed functions [Adal 97]. Finally, it can be shown that adopting the cross-entropy cost function and binary values for the desired responses, *the outputs $\hat{y}_k$ corresponding to the optimal weights $\boldsymbol{w}^*$ are indeed estimates of $P(\omega_k|\boldsymbol{x})$*, as in the least squares case [Hamp 90].

A major advantage of the cross-entropy cost function is that it diverges if one of the outputs converges to the wrong extreme, hence the gradient descent reacts fast. On the other hand, the squared error cost function approaches a constant in this case, and the gradient descent on the LS will wander on a plateau, even though the error may not be small. This advantage of the cross-entropy cost function is demonstrated in the channel equalization context in [Adal 97].

A different cost function results if we treat $\hat{y}_k(i)$ and $y_k(i)$ as the true and desired probabilities, respectively. Then a measure of their similarity is given by the cross-entropy function (Appendix A)

$$J = -\sum_{i=1}^{N}\sum_{k=1}^{k_L} y_k(i)\ln\frac{\hat{y}_k(i)}{y_k(i)} \tag{4.33}$$

This is also valid for binary target values (using the limiting form). However, although we have interpreted the outputs as probabilities, there is no guarantee that they sum up to unity. This can be imposed onto the network by adopting an alternative activation function for the output nodes. In [Brid 90] the so-called *softmax* activation function was suggested, given by

$$\hat{y}_k = \frac{\exp(v_k^L)}{\sum_{k'} \exp(v_{k'}^L)} \tag{4.34}$$

This guarantees that the outputs lie in the interval [0, 1] and that they sum up to unity (note that in contrast to (4.32) the output probabilities are not considered independently). It is easy to show, (Problem 4.7) that in this case the quantity δ_j^L required by the backpropagation is equal to $\hat{y}_k - y_k$.

Besides the cross-entropy cost function in (4.33) a number of alternative cost functions has been proposed. For example, in [Kara 92] a generalization of the quadratic error cost function is utilized with the aim of speeding up convergence. Another direction is to minimize the classification error, which after all is the major goal in pattern recognition. A number of techniques have been suggested with this philosophy [Nede 93, Juan 92, Pado 95], which is known as *discriminative learning*. The basic potential advantage of discriminative learning is that essentially it tries to move the decision surfaces so as to reduce classification error. To achieve this goal, it puts more emphasis on the largest of the class *a posteriori* probability estimates. In contrast, the squared error cost function, for example, assigns the same importance to all posterior probability estimates. In other words, it tries to learn more than what is necessary for classification, which may limit its performance for a fixed size network. Most of the discriminative learning techniques use a smoothed version of the classification error, so as to be able to apply differentiation in association with gradient descent approaches. This, of course, presents the danger that the minimization procedure will be trapped in a local minimum. In [Mill 96] a *deterministic annealing* procedure is employed to train the networks, with an enhanced potential to avoid local minima (see Chapter 15).

The final choice of the cost function depends on the specific problem under consideration. However, as is pointed out in [Rich 91], in a number of practical situations the use of alternative, to least squares, cost functions did not necessarily lead to substantial performance improvements.

So far, the task of training multilayer perceptrons has been approached via the *unconstrained* optimization route (Appendix C). However, more recent research has shown that it is often beneficial to incorporate *additional knowledge* in the learning rule using *constrained* optimization. This has been shown to lead to the formulation of efficient learning algorithms with accelerated learning properties. The additional knowledge can be encoded in the form of objectives leading to single- or multi-objective optimization criteria that have to be satisfied simultaneously, with the demand for a long-term decrease of the cost function [Pera 00].

In this approach, an optimization problem is formulated at each epoch of the learning process. For example, the requirement of partial alignment of current and previous epoch weight vector updates (respectively denoted by $\Delta \boldsymbol{w}(t)$ and $\Delta \boldsymbol{w}(t-1)$), which basically underlies the use of the momentum term in the backpropagation algorithm, is enhanced by requiring maximization of the quantity $\Phi = \Delta \boldsymbol{w}(t)^T \Delta \boldsymbol{w}(t-1)$ and simultaneously allowing for a controlled decrease of the cost function. This leads to a constrained first-order algorithm, and it has been reported that it outperforms backpropagation and several of its variants in different benchmark learning tasks [Pera 95].

Generalizations of the method involving the Hessian matrix have also been proposed and used successfully in several benchmarks and applications [Ampa 02, Huan 04]. Under certain conditions, the solution of the problem for each epoch is provided analytically, leading to a closed formula for the weight update rule [Pera 03].

A Bayesian Framework for Network Training

All the cost functions considered so far aim at computing a single set of optimal values for the unknown parameters of the network. An alternative rationale is to look at the probability distribution function of the unknown weights, $\boldsymbol{w}$, in the weight space. The idea behind this approach stems from the Bayesian inference technique used for the estimating an unknown parametric pdf, as we discussed in Chapter 2. The basic steps followed for this type of network training, known as *Bayesian learning*, are (e.g., [Mack 92a]):

- Assume a model for the prior distribution $p(\boldsymbol{w})$ of the weights. This must be rather broad in shape in order to provide equal chance to a rather large range of values.
- Let $Y = \{\boldsymbol{y}(i), i = 1, 2, \ldots, N\}$ be the set of the desired output training vectors for a given input data set $X = \{\boldsymbol{x}(i), i = 1, 2, \ldots, N\}$. Assume a model for the likelihood function $p(Y|\boldsymbol{w})$, for example, Gaussian.[4] This basically models the error distribution between the true and desired output values, and it is the stage at which the input training data come into the scene.
- Using Bayes's theorem, we obtain

$$p(\boldsymbol{w}|Y) = \frac{p(Y|\boldsymbol{w})p(\boldsymbol{w})}{p(Y)} \tag{4.35}$$

  where $p(Y) = \int p(Y|\boldsymbol{w})p(\boldsymbol{w})\, d\boldsymbol{w}$. The resulting posterior pdf will be more sharply shaped around a value $\boldsymbol{w}_0$, since it has learned from the available training data.

[4] Strictly speaking we should write $P(Y|\boldsymbol{w}, X)$. However, all probabilities and pdf's are conditioned on X, and we omit it for notational convenience.

- Interpreting the true outputs of a network, $\hat{y}_k = \phi_k(\boldsymbol{x}; \boldsymbol{w})$, as the respective class probabilities, conditioned on the input $\boldsymbol{x}$ and the weight vector $\boldsymbol{w}$, the conditional class probability is computed by averaging over all $\boldsymbol{w}$ [Mack 92b]:

$$P(\omega_k|\boldsymbol{x}; Y) = \int \phi_k(\boldsymbol{x}; \boldsymbol{w}) p(\boldsymbol{w}|Y)\, d\boldsymbol{w} \tag{4.36}$$

The major computational cost associated with this type of technique is due to the required integration in the multidimensional space. This is not an easy task, and various practical implementations have been suggested in the literature. Further discussion of these issues is beyond the scope of this book. A good introduction to Bayesian learning, including a discussion of related practical implementations, is provided in [Bish 95].

4.9 CHOICE OF THE NETWORK SIZE

In the previous sections, we assumed the number of layers and neurons for each layer to be known and fixed. How one determines the appropriate number of layers and neurons was not of interest to us. This task will become our major focus now.

One answer to the problem could be to choose the size of the network large enough and leave the training to decide about the weights. A little thought reveals that such an approach is rather naive. Besides the associated computational complexity problems, there is a major reason why the size of the network should be kept as small as possible. This is imposed by the generalization capabilities that the network must possess. As has already been pointed out in Section 3.7, the term *generalization* refers to the capability of the multilayer neural network (and of any classifier) to classify correctly feature vectors that were not presented to it during the training phase—that is, the capability of a network to decide upon data unknown to it, based on what it has learned from the training set. Taking for granted the finite (and in many cases small) number N of training pairs, the number of free parameters (synaptic weights) to be estimated should be *(a) large enough to learn what makes "similar" the feature vectors within each class and at the same time what makes one class different from the other and (b) small enough, with respect to N, so as not to be able to learn the underlying differences among the data of the same class*. When the number of free parameters is large, the network tends to adapt to the particular details of the specific training data set. This is known as *overfitting* and leads to poor generalization performance, when the network is called to operate on feature vectors unknown to it. In conclusion, the network should have the smallest possible size to adjust its weights to the largest regularities in the data and ignore the smaller ones, which might also be the result of noisy measurements. Some theoretical touches concerning the generalization aspects of a classifier will

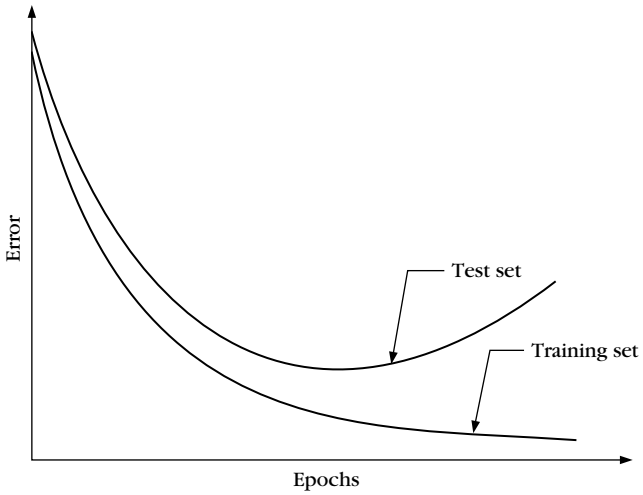

FIGURE 4.14

Trend of the output error versus the number of epochs illustrating overtraining of the training set.

be presented in Chapter 5, when we discuss the Vapnik–Chervonenkis dimension. The bias–variance dilemma, discussed in Chapter 3, is another side of the same problem.

Adaptation of the free parameters to the peculiarities of the specific training set may also occur as the result of *overtraining* (e.g., see [Chau 90]). Let us assume that we can afford the luxury of having a large set of training data. We divide this set into two subsets, one for training and one for test. The latter is known as the *validation* or *test set*. Figure 4.14 shows the trend of two curves of the output error as a function of iteration steps. One corresponds to the training set, and we observe that the error keeps decreasing as the weights converge. The other corresponds to the error of the validation set. Initially, the error decreases, but at some later stage it starts increasing. This is because the weights, computed from the training set, adapt to the idiosyncrasies of the specific training set, thus affecting the generalization performance of the network. This behavior could be used in practice to determine the point where the learning process iterations must terminate. This is the point where the two curves start departing. However, this methodology assumes the existence of a large number of data sets, which is not usually the case in practice.

Besides generalization, other performance factors also demand to keep the size of a network as small as possible. Small networks are computationally faster and cheaper to build. Furthermore, their performance is easier to understand, which is important in some critical applications.

In this section we focus on methods that select the appropriate number of free parameters, under certain criteria and for a given dimension of the input vector space. The latter is very important, because the input data dimension is no doubt related to the number of free parameters to be used; thus it also affects generalization properties. We will come to issues related to input space dimension reduction in Chapter 5.

The most widely used approaches to selecting the size of a multilayer network come under one of the following categories:

- *Analytical methods*. This category employs algebraic or statistical techniques to determine the number of its free parameters.
- *Pruning techniques*. A large network is initially chosen for training, and then the number of free parameters is successively reduced, according to a preselected rule.
- *Constructive techniques*. A small network is originally selected, and neurons are successively added, based on an appropriately adopted learning rule.

Algebraic Estimation of the Number of Free Parameters

We have already discussed in Section 4.3.1 the capabilities of a multilayer perceptron, with one hidden layer and units of the McCulloch–Pitts type, to divide the input l-dimensional space into a number of polyhedral regions. These are the result of intersections of hyperplanes formed by the neurons. In [Mirc 89] it is shown that in the l-dimensional space a multilayer perceptron of a single hidden layer with K neurons can form *a maximum* of M polyhedral regions with M given by

$$M = \sum_{m=0}^{l} \binom{K}{m}, \quad \text{where } \binom{K}{m} = 0, \quad \text{for } K < m \tag{4.37}$$

and

$$\binom{K}{m} \equiv \frac{K!}{m!(K-m)!}$$

For example, if $l = 2$, and $K = 2$, this results in $M = 4$; thus, the XOR problem, with $M = 3(< 4)$, can be solved with two neurons. The disadvantage of this method is that it is static and does not take into consideration the cost function used as well as the training procedure.

Pruning Techniques

These techniques start training a sufficiently large network, and then they remove, in a stepwise procedure, the free parameters that have little influence on the cost function. There are two major methodological directions:

Methods Based on Parameter Sensitivity Calculations

Let us take for example the technique suggested in [Lecu 90]. Using a Taylor series expansion, the variation imposed on the cost function by parameter perturbations is

$$\delta J = \sum_i g_i \delta w_i + \frac{1}{2} \sum_i h_{ii} \delta w_i^2 + \frac{1}{2} \sum_{\substack{i,j \\ i \neq j}} h_{ij} \delta w_i \delta w_j$$

$$+ \text{ higher order terms}$$

where

$$g_i = \frac{\partial J}{\partial w_i}, \quad h_{ij} = \frac{\partial^2 J}{\partial w_i \partial w_j}$$

and i, j runs over all the weights. The derivatives can be computed via the backpropagation methodology (Problem 4.13). In practice, the derivatives are computed after some initial period of training. This allows us to adopt the assumption that a point near a minimum has been reached and the first derivatives can be set equal to zero. A further computational simplification is to assume that the Hessian matrix is diagonal. Under these assumptions, the cost function sensitivity is approximately given by

$$\delta J = \frac{1}{2} \sum_i h_{ii} \delta w_i^2 \tag{4.38}$$

and the contribution of each parameter is determined by the *saliency* value s_i, given approximately by

$$s_i = \frac{h_{ii} w_i^2}{2} \tag{4.39}$$

where we assume that a weight of value w_i is changed to zero. Pruning is now achieved in an iterative fashion according to the following steps:

- The network is trained using the backpropagation algorithm for a number of iteration steps so that its cost function is reduced to a sufficient percentage.
- For the current weight estimates, the respective saliency values are computed and weights with small salencies are removed.
- The training process is continued with the remaining weights, and the process is repeated after some iteration steps. The process is stopped when a chosen stopping criterion is met.

In [Hass 93] the full Hessian matrix has been employed for the pruning procedure. It should be stressed that, although the backpropagation concept is present in this technique, the learning procedure is distinctly different from the backpropagation

training algorithm of Section 4.6. *There, the number of free parameters was fixed throughout the training. In contrast, the philosophy here is exactly the opposite*.

Methods Based on Cost Function Regularization

These methods achieve the reduction of the originally large size of the network by including *penalty terms* in the cost function. The cost function now has the form

$$J = \sum_{i=1}^{N} \mathcal{E}(i) + \alpha \mathcal{E}_p(\boldsymbol{w}) \tag{4.40}$$

The first term is the performance cost function, and it is chosen according to what we have already discussed (e.g., least squares, cross entropy). The second depends on the weight vector, and it is chosen to *favor small values for the weights*. The constant α is the so-called *regularization parameter*, and it controls the relative significance of these two terms. A popular form for the penalty term is

$$\mathcal{E}_p(\boldsymbol{w}) = \sum_{k=1}^{K} h(w_k^2) \tag{4.41}$$

with K being the total number of weights in the network and $h(\cdot)$ an appropriately chosen differentiable function. According to such a choice, weights that do not contribute significantly in the formation of the network output do not materially affect much the first term of the cost function. Hence, the existence of the penalty term drives them to small values. Thus, pruning is achieved. In practice, a threshold is preselected and weights are compared against it after a number of iteration steps. Weights that become smaller than it are removed, and the process is continued. This type of pruning is known as *weight elimination*. Function $h(\cdot)$ can take various forms. For example, in [Wein 90] the following is suggested:

$$h(w_k^2) = \frac{w_k^2}{w_0^2 + w_k^2} \tag{4.42}$$

where w_0 is a preselected parameter close to unity. Closer observation of this penalty term reveals that it goes to zero very fast for values $w_k < w_0$; thus such weights become insignificant. In contrast, the penalty term tends to unity for $w_k > w_0$.

A variation of (4.41) is to include in the regularized cost function another penalty term that favors small values of y_k^r, that is, small neuron outputs. Such techniques lead to removal of insignificant neurons as well as weights. A summary and discussion of various pruning techniques can be found in [Refe 91, Russ 93].

Keeping the size of the weights small is in line with the theoretical results obtained in [Bart 98]. Assume that a large multilayer perceptron is used as a classifier and that the learning algorithm finds a network with (a) small weights and (b) small squared error on the training patterns. Then, it can be shown that the

generalization performance depends on the size of the weights rather than the number of weights. More specifically, for a two-layer perceptron with sigmoid activation functions if A is an upper bound of the sum of the magnitudes of the weights associated with each neuron, then the associated classification error probability is no more than a certain error estimate (related to the output squared error) plus $A^3\sqrt{(\log l)/N}$.

This is a very interesting result indeed. It confronts the generally accepted fact that the generalization performance is directly related to the number of training points and the number of free parameters. It also explains why sometimes the generalization error performance of multilayer perceptrons is good, although the training has been performed with a relatively small (compared to the size of the network) number of training points. Further discussion concerning the generalization performance of a classifier and some interesting theoretical results is found at the end of Chapter 5.

Constructive Techniques

In Section 4.5 we have already discussed such techniques for training neural networks. However, the activation function was the unit-step function, and also the emphasis was put on classifying *correctly all* input training data and not on the generalization properties of the resulting network. In [Fahl 90] an alternative constructive technique for training neural networks, with a single hidden layer and sigmoid activation functions, was proposed, known as *cascade correlation*. The network starts with input and output units only. Hidden neurons are added one by one and are connected to the network with two types of weights. The first type connects the new unit with the input nodes as well as the outputs of previously added hidden neurons. Each time a new hidden neuron is added in the network, these weights are trained so as to maximize the correlation between the new unit's output and the residual error signal in the network outputs prior to the addition of the new unit. Once a neuron is added, these weights are computed once and then they remain fixed. The second type of synaptic weights connects the newly added neuron with the output nodes. These weights are not fixed and are trained adaptively, each time a new neuron is installed, in order to minimize a sum of squares error cost function. The procedure stops when the performance of the network meets the prespecified goals. Discussion of constructive techniques with an emphasis on pattern recognition can be found in [Pare 00].

4.10 A SIMULATION EXAMPLE

This section demonstrates the capability of a multilayer perceptron to classify nonlinearly separable classes. The classification task consists of two distinct classes, each being the union of four regions in the two-dimensional space. Each region consists of normally distributed random vectors with statistically independent

components and each with variance $\sigma^2 = 0.08$. The mean values are different for each of the regions. Specifically, the regions of the class denoted by red "$\circ$" (see Figure 4.15) are formed around the mean vectors

$$[0.4, 0.9]^T, [2.0, 1.8]^T, [2.3, 2.3]^T, [2.6, 1.8]^T$$

and those of the class denoted by black "$+$" around the values

$$[1.5, 1.0]^T, [1.9, 1.0]^T, [1.5, 3.0]^T, [3.3, 2.6]^T$$

A total of 400 training vectors were generated, 50 from each distribution. A multilayer perceptron, with three neurons in the first and two neurons in the second hidden layer, were used, with a single output neuron. The activation function was the logistic one with $a = 1$ and the desired outputs 1 and 0, respectively, for the two classes. Two different algorithms were used for the training, namely, the momentum and the adaptive momentum. After some experimentation the algorithmic parameters employed were (a) for the momentum $\mu = 0.05, \alpha = 0.85$ and (b) for the adaptive momentum $\mu = 0.01, \alpha = 0.85, r_i = 1.05, c = 1.05, r_d = 0.7$. The weights were initialized by a uniform pseudorandom distribution between 0 and 1. Figure 4.15a shows the respective output error convergence curves for the two algorithms as a function of the number of epochs (each epoch consisting of the 400 training feature vectors). The respective curves can be considered typical and the adaptive momentum algorithm leads to faster convergence. Both curves correspond to the batch mode of operation. Figure 4.15b shows the resulting decision surface using

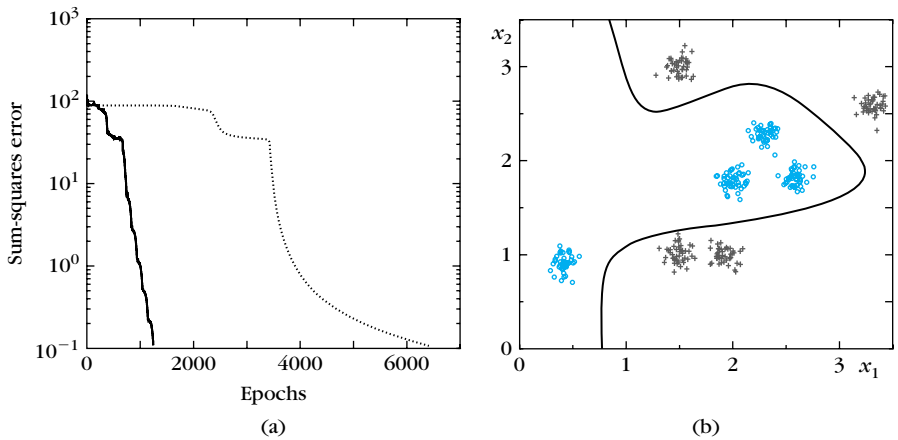

FIGURE 4.15

(a) Error convergence curves for the adaptive momentum (dark line) and the momentum algorithms. Note that the adaptive momentum leads to faster convergence. (b) The decision curve formed by the multilayer perceptron.

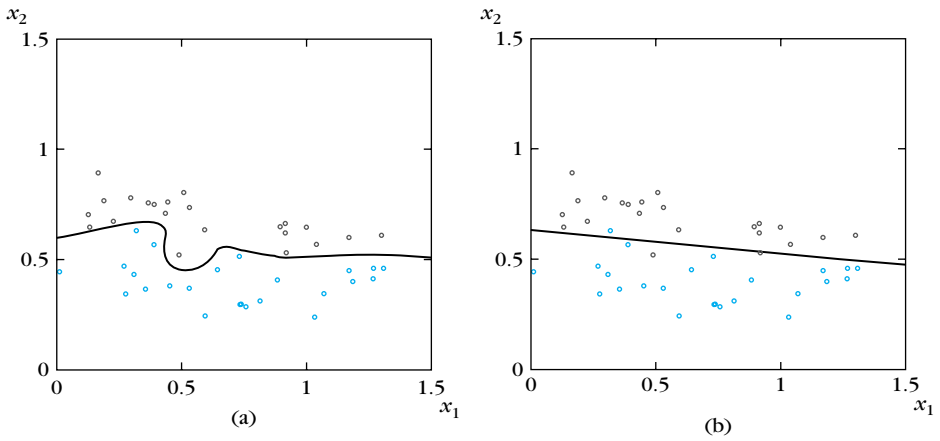

FIGURE 4.16

Decision curve (a) before pruning and (b) after pruning.

the weights estimated from the adaptive momentum training. Once the weights of the network have been estimated, the decision surface can easily be drawn. To this end, a two-dimensional grid is constructed over the area of interest, and the points of the grid are given as inputs to the network, row by row. The decision surface is formed by the points where the output of the network changes from 0 to 1 or vice versa.

A second experiment was conducted in order to demonstrate the effect of the pruning. Figure 4.16 shows the resulting decision surfaces separating the samples of the two classes, denoted by black and red "○," respectively. Figure 4.16a corresponds to a multilayer perceptron (MLP), with two hidden layers and 20 neurons in each of them, amounting to a total of 480 weights. Training was performed via the backpropagation algorithm. The overfitting nature of the resulting curve is readily observed. Figure 4.16b corresponds to the same MLP trained with a pruning algorithm. Specifically, the method based on parameter sensitivity was used, testing the saliency values of the weights every 100 epochs and removing weights with saliency value below a chosen threshold. Finally, only 25 of the 480 weights were left, and the curve is simplified to a straight line.

4.11 NETWORKS WITH WEIGHT SHARING

One major issue encountered in many pattern recognition applications is that of transformation invariance. This means that the pattern recognition system should classify correctly, independent of transformations performed on the input space, such as translation, rotation, and scaling. For example, the character "5" should "look the same" to an optical character recognition system, regardless of its position,

orientation, and size. There are a number of ways to approach this problem. One is to choose appropriate feature vectors, which are invariant under such transformations. This will be one of our major goals in Chapter 7. Another way is to make the classifier responsible for it in the form of *built-in constraints. Weight sharing is such a constraint, which forces certain connections in the network to have the same weights*.

One type of network in which the concept of weight sharing has been adopted is the so-called *higher order network*. These are multilayer perceptrons with activation functions acting on nonlinear, instead of linear, combinations of the input parameters. The outputs of the neurons are now of the form

$$f(v) = f\left(w_0 + \sum_i w_i x_i + \sum_{jk} w_{jk} x_j x_k\right)$$

This can be generalized to include higher order products. Let us now assume that the inputs to the network originate from a two-dimensional grid (image). Each point of the grid corresponds to a specific x_i and each pair (x_i, x_j) to a line segment. Invariance to translation is built in by constraining the weights $w_{jk} = w_{rs}$, whenever the respective line segments, defined by the points (x_j, x_k) and (x_r, x_s), are of the same gradient. Invariance to rotation can be built in by sharing weights corresponding to segments of equal length. Of course, all these are subject to inaccuracies caused by the resolution coarseness of the grid. Higher order networks can accommodate more complex transformations [Kana 92, Pera 92, Delo 94]. Because of the weight sharing, the number of free parameters for optimization is substantially reduced. However, it must be pointed out that, so far, such networks have not been widely used in practice. A special type of network called the *circular backpropagation model* results if

$$f(v) = f\left(w_0 + \sum_i w_i x_i + w_s \sum_i x_i^2\right)$$

The increase in the number of parameters is now small, and in [Ride 97] it is claimed that the involvement of the nonlinear term offers the network increased representation power without affecting its generalization capabilities.

Besides the higher order networks, weight sharing has been used to impose invariance on first-order networks used for specific applications [Fuku 82, Rume 86, Fuku 92, Lecu 89]. The last, for example, is a system for handwritten zip code recognition. It is a hierarchical structure with three hidden layers and inputs the gray levels of the image pixels. Nodes in the first two layers form groups of two-dimensional arrays known as *feature maps*. Each node in a given map receives inputs from a specific window area of the previous layer, known as the *receptive field*. Translation invariance is imposed by forcing corresponding nodes in the same map, looking at different receptive fields, to share weights. Thus, if an object moves from one input receptive field to the other, the network responds in the same way.

4.12 GENERALIZED LINEAR CLASSIFIERS

In Section 4.3, dealing with the nonlinearly separable XOR problem, we saw that the neurons of the hidden layer performed a mapping that transformed the problem to a linearly separable one. The actual mapping was

$$\boldsymbol{x} \rightarrow \boldsymbol{y}$$

with

$$\boldsymbol{y} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} f(g_1(\boldsymbol{x})) \\ f(g_2(\boldsymbol{x})) \end{bmatrix} \tag{4.43}$$

where $f(\cdot)$ is the activation function and $g_i(\boldsymbol{x}), i = 1, 2$, the linear combination of the inputs performed by each neuron. This will be our kickoff point for this section.

Let us consider our feature vectors to be in the l-dimensional space $\mathcal{R}^l$ and assume that they belong to either of the two classes A, B, which are nonlinearly separable. Let $f_1(\cdot), f_2(\cdot), \ldots, f_k(\cdot)$ be nonlinear (in the general case) functions

$$f_i : \mathcal{R}^l \rightarrow \mathcal{R}, \quad i = 1, 2, \ldots, k$$

which define the mapping $\boldsymbol{x} \in \mathcal{R}^l \rightarrow \boldsymbol{y} \in \mathcal{R}^k$

$$\boldsymbol{y} \equiv \begin{bmatrix} f_1(\boldsymbol{x}) \\ f_2(\boldsymbol{x}) \\ \vdots \\ f_k(\boldsymbol{x}) \end{bmatrix} \tag{4.44}$$

Our goal now is to investigate whether there is an appropriate value for k and functions f_i so that classes A, B are linearly separable in the k-dimensional space of the vectors $\boldsymbol{y}$. In other words, we investigate whether there exists a k-dimensional space where we can construct a hyperplane $\boldsymbol{w} \in \mathcal{R}^k$ so that

$$w_0 + \boldsymbol{w}^T \boldsymbol{y} > 0, \quad \boldsymbol{x} \in A \tag{4.45}$$

$$w_0 + \boldsymbol{w}^T \boldsymbol{y} < 0, \quad \boldsymbol{x} \in B \tag{4.46}$$

Assuming that in the original space the two classes were separable by a (nonlinear) hypersurface $g(\boldsymbol{x}) = 0$, relations (4.45), (4.46) are basically equivalent to approximating the nonlinear $g(\boldsymbol{x})$ as a linear combination of $f_i(\boldsymbol{x})$, that is,

$$g(\boldsymbol{x}) = w_0 + \sum_{i=1}^{k} w_i f_i(\boldsymbol{x}) \tag{4.47}$$

This is a typical problem of function approximation in terms of a preselected class of *interpolation functions* $f_i(\cdot)$. This is a well-studied task in numerical analysis,

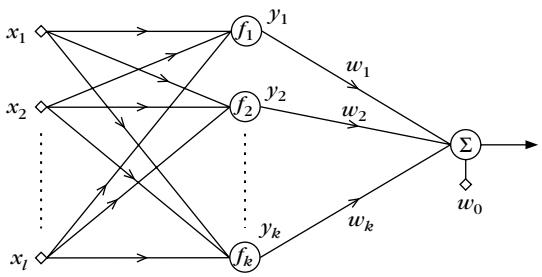

FIGURE 4.17

Generalized linear classifier.

and a number of different interpolating functions have been proposed (exponential, polynomial,Tchebyshev,etc.). In the next sections,we will focus on two such classes of functions, which have been widely used in pattern recognition.

Once the functions f_i have been selected, the problem becomes a typical design of a linear classifier, that is, to estimate the weights w_i in the k-dimensional space. This justifies the term *generalized linear classification*. Figure 4.17 shows the corresponding block diagram. The first layer of computations performs the mapping to the $\boldsymbol{y}$ space; the second layer performs the computation of the decision hyperplane. In other words, (4.47) corresponds to a *two-layer network* where the nodes of the hidden layer have different activation functions $f_i(\cdot)$, $i = 1, 2, \ldots, k$. For an M-class problem we need to design M such weight vectors $\boldsymbol{w}_r$, $r = 1, 2, \ldots, M$, one for each class, and select the rth class according to the maximum output $\boldsymbol{w}_r^T \boldsymbol{y} + w_{r0}$.

A similar expression to (4.47) expansion of $g(\boldsymbol{x})$ is known as *projection pursuit*, introduced in [Fried 81], and it is defined as

$$g(\boldsymbol{x}) = \sum_{i=1}^{k} f_i(\boldsymbol{w}_i^T \boldsymbol{x})$$

Observe that the argument in each of the functions $f_i(\cdot)$ is not the feature vector $\boldsymbol{x}$ but its *projection* on the direction determined by the respective $\boldsymbol{w}_i$. Optimization with respect to f_i and $\boldsymbol{w}_i$, $i = 1, 2, \ldots, k$, results in the best choice for directions to project as well as the interpolation functions. If $f_i(\cdot)$ are all chosen *a priori* to be sigmoid functions, the projection pursuit method becomes identical to a neural network with a single hidden layer. Projection pursuit models *are not* members of the generalized linear models family, and their optimization is carried out iteratively in a two-stage fashion. Given the functions $f_i(\cdot)$, $\boldsymbol{w}_i$s are estimated and in the next stage optimization is performed with respect to the f_is. See, for example, [Fried 81]. Although interesting from a theoretical point of view, it seems that in practice projection pursuit methods have been superseded by the multilayer perceptrons.

In the sequel, we will first try to justify our expectations, that by going to a higher dimensional space the classification task may be transformed into a linear one, and then study popular alternatives for the choice of functions $f_i(\cdot)$ in (4.44).

4.13 CAPACITY OF THE *l*-DIMENSIONAL SPACE IN LINEAR DICHOTOMIES

Let us consider N points in the l-dimensional space. We will say that these points are *in general position* or *well distributed* if there is no subset of $l+1$ of them that lie on an $(l-1)$-dimensional hyperplane. Such a definition excludes detrimental cases, such as in the two-dimensional space having three points on a straight line (a one-dimensional hyperplane). The number $O(N,l)$ of groupings that can be formed by $(l-1)$-dimensional hyperplanes to separate the N points in two classes, taking all possible combinations, is given by ([Cove 65] and Problem 4.18):

$$O(N,l) = 2\sum_{i=0}^{l}\binom{N-1}{i} \tag{4.48}$$

where

$$\binom{N-1}{i} = \frac{(N-1)!}{(N-1-i)!i!} \tag{4.49}$$

Each of these two class groupings is also known as a (linear) *dichotomy*. From the properties of the binomial coefficients, it turns out that for $N \leq l+1, O(N,l) = 2^N$. Figure 4.18 shows two examples of such hyperplanes resulting in $O(4,2) = 14$ and $O(3,2) = 8$ two-class groupings, respectively. The seven lines of Figure 4.18a form the following groupings. [(ABCD)], [A,(BCD)], [B,(ACD)], [C,(ABD)], [D,(ABC)], [(AB),(CD)], and [(AC),(BD)]. Each grouping corresponds to two possibilities. For example, (ABCD) can belong to either class ω_1 or ω_2. Thus, the total number of

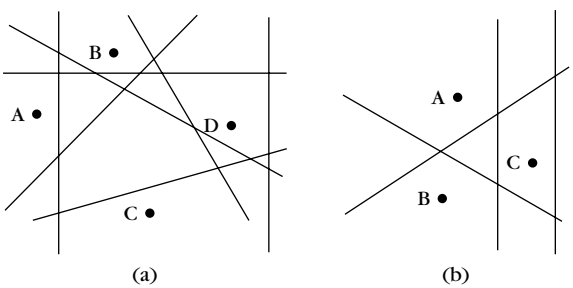

FIGURE 4.18

Number of linear dichotomies (a) for four and (b) for three points.

combinations of assigning four points in the two-dimensional space in two *linearly separable classes is* 14. This number is obviously smaller than the total number of combinations of assigning N points in two classes, which is known to be 2^N. This is because the latter also involves nonlinearly separable combinations. In the case of our example, this is 16, which arises from the two extra possibilities of the grouping [(AD), (BC)]. We are now ready to write the probability (percentage) of grouping N points in the l-dimensional space in two *linearly* separable classes [Cove 65]. This is given by:

$$P_N^l = \frac{O(N,l)}{2^N} = \begin{cases} \frac{1}{2^{N-1}} \sum_{i=0}^{l} \binom{N-1}{i} & N > l+1 \\ 1 & N \leq l+1 \end{cases} \tag{4.50}$$

A practical way to study the dependence of P_N^l on N and l is to assume that $N = r(l+1)$ and investigate the probability for various values of r. The curve in Figure 4.19 shows the probability of having linearly separable classes for various values of l. It is readily observed that there are two regions, one to the left of $r = 2$, that is, $N = 2(l+1)$, and one to the right. Furthermore, all curves go through the point $(P_N^l, r) = (1/2, 2)$, since $O(2l+2, l) = 2^{2l+1}$ (Problem 4.19). The transition from one region to the other becomes sharper as $l \to \infty$. Thus, for large values of l and if $N < 2(l+1)$ the probability of any two groups of the N points being linearly separable approaches unity. The opposite is true if $N > 2(l+1)$. In practice, where we cannot afford the luxury of very large values of N and l, our findings

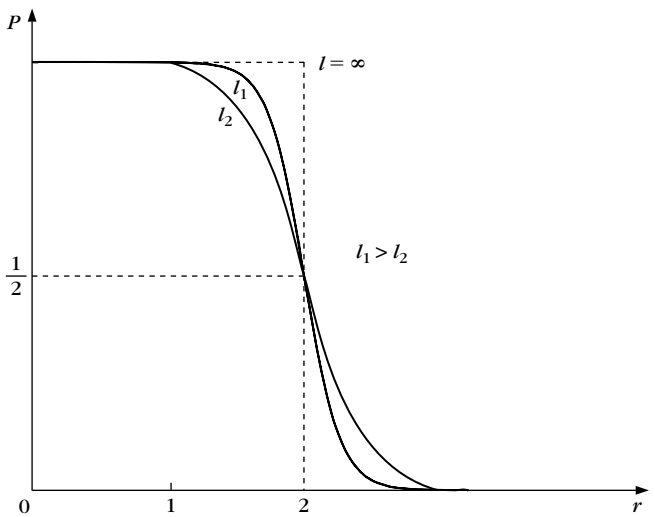

FIGURE 4.19

Probability of linearly separable groupings of $N = r(l+1)$ points in the l-dimensional space.

guarantee that *if we are given N points, then mapping into a higher dimensional space increases the probability of locating them in linearly separable two-class groupings*.

4.14 POLYNOMIAL CLASSIFIERS

In this section we will focus on one of the most popular classes of interpolation functions $f_i(\boldsymbol{x})$ in (4.47). Function $g(\boldsymbol{x})$ is approximated in terms of up to order r polynomials of the $\boldsymbol{x}$ components, for large enough r. For the special case of $r = 2$ we have

$$g(\boldsymbol{x}) = w_0 + \sum_{i=1}^{l} w_i x_i + \sum_{i=1}^{l-1} \sum_{m=i+1}^{l} w_{im} x_i x_m + \sum_{i=1}^{l} w_{ii} x_i^2 \tag{4.51}$$

If $\boldsymbol{x} = [x_1, x_2]^T$, then the general form of $\boldsymbol{y}$ will be

$$\boldsymbol{y} = [x_1, x_2, x_1 x_2, x_1^2, x_2^2]^T$$

and

$$g(\boldsymbol{x}) = \boldsymbol{w}^T \boldsymbol{y} + w_0$$
$$\boldsymbol{w}^T = [w_1, w_2, w_{12}, w_{11}, w_{22}]$$

The number of free parameters determines the new dimension k. The generalization of (4.51) for rth-order polynomials is straightforward, and it will contain products of the form $x_1^{p_1} x_2^{p_2} \ldots x_l^{p_l}$ where $p_1 + p_2 + \cdots + p_l \leq r$. For an rth-order polynomial and l-dimensional $\boldsymbol{x}$ it can be shown that

$$k = \frac{(l + r)!}{r!l!}$$

For $l = 10$ and $r = 10$ we obtain $k = 184{,}756$ (!!). That is, even for medium-size values of the network order and the input space dimensionality the number of free parameters gets very high.

Let us consider, for example, our familiar nonlinearly separable XOR problem. Define

$$\boldsymbol{y} = \begin{bmatrix} x_1 \\ x_2 \\ x_1 x_2 \end{bmatrix} \tag{4.52}$$

The input vectors are mapped onto the vertices of a three-dimensional unit (hyper) cube, as shown in Figure 4.20a ((00) $\rightarrow$ (000), (11) $\rightarrow$ (111), (10) $\rightarrow$ (100), (01) $\rightarrow$ (010)). These vertices are separable by the plane

$$y_1 + y_2 - 2y_3 - \frac{1}{4} = 0$$

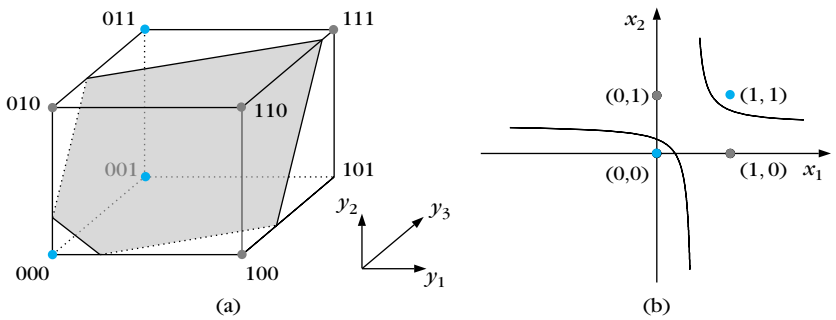

FIGURE 4.20

The XOR classification task, via the polynomial generalized linear classifier. (a) Decision plane in the three-dimensional space and (b) decision curves in the original two-dimensional space.

The plane in the three-dimensional space is equivalent to the decision function

$$g(\boldsymbol{x}) = -\frac{1}{4} + x_1 + x_2 - 2x_1x_2 \quad \begin{matrix} >0 & \boldsymbol{x} \in A \\ <0 & \boldsymbol{x} \in B \end{matrix}$$

in the original two-dimensional space, which is shown in Figure 4.20b.

4.15 RADIAL BASIS FUNCTION NETWORKS

The interpolation functions (kernels) that will be considered in this section are of the general form

$$f(\|\boldsymbol{x} - \boldsymbol{c}_i\|)$$

That is, the argument of the function is the Euclidean distance of the input vector $\boldsymbol{x}$ from a center $\boldsymbol{c}_i$, which justifies the name *radial basis function (RBF)*. Function f can take various forms, for example,

$$f(\boldsymbol{x}) = \exp\Big(-\frac{1}{2\sigma_i^2}\|\boldsymbol{x} - \boldsymbol{c}_i\|^2\Big) \tag{4.53}$$

$$f(\boldsymbol{x}) = \frac{\sigma^2}{\sigma^2 + \|\boldsymbol{x} - \boldsymbol{c}_i\|^2} \tag{4.54}$$

The Gaussian form is more widely used. For a large enough value of k, it can be shown that the function $g(\boldsymbol{x})$ is sufficiently approximated by [Broo 88, Mood 89]

$$g(\boldsymbol{x}) = w_0 + \sum_{i=1}^{k} w_i \exp\Big(-\frac{(\boldsymbol{x} - \boldsymbol{c}_i)^T(\boldsymbol{x} - \boldsymbol{c}_i)}{2\sigma_i^2}\Big) \tag{4.55}$$

That is, the approximation is achieved via a summation of RBFs, where each is located on a different point in the space. One can easily observe the close relation that exists between this and the Parzen approximation method for the probability density functions of Chapter 2. Note, however, that there the number of the kernels was chosen to be equal to the number of training points $k = N$. In contrast, in (4.55) $k \ll N$. Besides the gains in computational complexity, this reduction in the number of kernels is beneficial for the generalization capabilities of the resulting approximation model.

Coming back to Figure 4.17, we can interpret (4.55) as the output of a network with *one* hidden layer of RBF activation functions (e.g., (4.53), (4.54)) and a *linear* output node. As has already been said in Section 4.12, for an M-class problem there will be M linear output nodes. At this point, it is important to stress one basic difference between RBF networks and multilayer perceptrons. In the latter, the inputs to the activation functions, of the first hidden layer, are linear combinations of the input feature parameters $\left(\sum_j w_j x_j\right)$. That is, the output of each neuron is the same for all $\{\boldsymbol{x}: \sum_j w_j x_j = c\}$, where c is a constant. *Hence, the output is the same for all points on a hyperplane.* In contrast, in the RBF networks the output of each RBF node, $f_i(\cdot)$, is the same for all points having *the same Euclidean distance from the respective center* $\boldsymbol{c}_i$ and decreases exponentially (for Gaussians) with the distance. *In other words, the activation responses of the nodes are of a local nature in the RBF and of a global nature in the multilayer perceptron networks.* This intrinsic difference has important repercussions for both the convergence speed and the generalization performance. In general, multilayer perceptrons learn slower than their RBF counterparts. In contrast, multilayer perceptrons exhibit improved generalization properties, especially for regions that are not represented sufficiently in the training set [Lane 91]. Simulation results in [Hart 90] show that, in order to achieve performance similar to that of multilayer perceptrons, an RBF network should be of much higher order. This is due to the locality of the RBF activation functions, which makes it necessary to use a large number of centers to fill in the space in which $g(\boldsymbol{x})$ is defined, and this number exhibits an exponential dependence on the dimension of the input space (curse of dimensionality) [Hart 90].

Let us now come back to our XOR problem and adopt an RBF network to perform the mapping to a linearly separable class problem. Choose $k = 2$, the centers $\boldsymbol{c}_1 = [1, 1]^T$, $\boldsymbol{c}_2 = [0, 0]^T$, and $f(\boldsymbol{x}) = \exp(-\|\boldsymbol{x} - \boldsymbol{c}_i\|^2)$. The corresponding $\boldsymbol{y}$ resulting from the mapping is

$$\boldsymbol{y} = \boldsymbol{y}(\boldsymbol{x}) = \begin{bmatrix} \exp(-\|\boldsymbol{x} - \boldsymbol{c}_1\|^2) \\ \exp(-\|\boldsymbol{x} - \boldsymbol{c}_2\|^2) \end{bmatrix}$$

Hence $(0, 0) \rightarrow (0.135, 1)$, $(1, 1) \rightarrow (1, 0.135)$, $(1, 0) \rightarrow (0.368, 0.368)$, $(0, 1) \rightarrow (0.368, 0.368)$. Figure 4.21a shows the resulting class position after the mapping

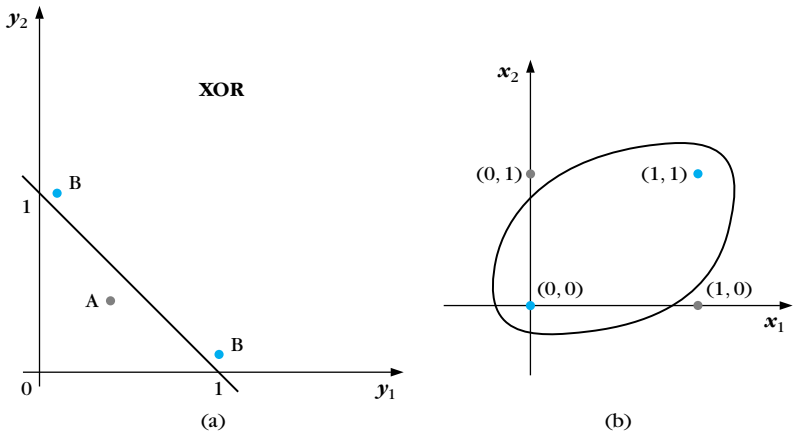

FIGURE 4.21

Decision curves formed by an RBF generalized linear classifier for the XOR task. The decision curve is linear in the transformed space (a) and nonlinear in the original space (b).

in the $\boldsymbol{y}$ space. Obviously, the two classes are now linearly separable and the straight line

$$g(\boldsymbol{y}) = y_1 + y_2 - 1 = 0$$

is a possible solution. Figure 4.21b shows the equivalent decision curve,

$$g(\boldsymbol{x}) = \exp(-\|\boldsymbol{x} - \boldsymbol{c}_1\|^2) + \exp(-\|\boldsymbol{x} - \boldsymbol{c}_2\|^2) - 1 = 0$$

in the input vector space. In our example we selected the centers $\boldsymbol{c}_1, \boldsymbol{c}_2$ as $[0, 0]^T$ and $[1, 1]^T$. The question now is, why these specific ones? *This is an important issue for RBF networks.* Some basic directions on how to tackle this problem are given in the following.

Fixed Centers

Although in some cases the nature of the problem suggests a specific choice for the centers [Theo 95], in the general case these centers can be selected randomly from the training set. Provided that the training set is distributed in a representative manner over all the feature vector space, this seems to be a reasonable way to choose the centers. Having now selected k centers for the RBF functions, the problem has become a typical linear one in the k-dimensional space of the vectors $\boldsymbol{y}$,

$$\boldsymbol{y} = \begin{bmatrix} \exp\left(\frac{-\|\boldsymbol{x}-\boldsymbol{c}_1\|^2}{2\sigma_1^2}\right) \\ \vdots \\ \exp\left(\frac{-\|\boldsymbol{x}-\boldsymbol{c}_k\|^2}{2\sigma_k^2}\right) \end{bmatrix}$$

where the variances are also considered to be known, and

$$g(\boldsymbol{x}) = w_0 + \boldsymbol{w}^T \boldsymbol{y}$$

All methods described in Chapter 3 can now be recalled to estimate w_0 and $\boldsymbol{w}$.

Training of the Centers

If the centers are not preselected, they have to be estimated during the training phase along with the weights w_i and the variances σ_i^2, if the latter are also considered unknown. Let N be the number of input–desired output training pairs $(\boldsymbol{x}(j), y(j), j = 1, \ldots, N)$. We select an appropriate cost function of the output error

$$J = \sum_{j=1}^{N} \phi(e(j))$$

where $\phi(\cdot)$ is a differentiable function (e.g., the square of its argument) of the error

$$e(j) = y(j) - g(\boldsymbol{x}(j))$$

Estimation of the weights w_i, the centers $\boldsymbol{c}_i$, and the variances σ_i^2 becomes a typical task of a nonlinear optimization process. For example, if we adopt the gradient descent approach, the following algorithm results:

$$w_i(t+1) = w_i(t) - \mu_1 \frac{\partial J}{\partial w_i}\bigg|_t, \quad i = 0, 1, \ldots, k \tag{4.56}$$

$$\boldsymbol{c}_i(t+1) = \boldsymbol{c}_i(t) - \mu_2 \frac{\partial J}{\partial \boldsymbol{c}_i}\bigg|_t, \quad i = 1, 2, \ldots, k \tag{4.57}$$

$$\sigma_i(t+1) = \sigma_i(t) - \mu_3 \frac{\partial J}{\partial \sigma_i}\bigg|_t, \quad i = 1, 2, \ldots, k \tag{4.58}$$

where t is the current iteration step. The computational complexity of such a scheme is prohibitive for a number of practical situations. To overcome this drawback, alternative techniques have been suggested.

One way is to choose the centers in a manner that is representative of the way data are distributed in space. This can be achieved by unraveling the clustering properties of the data and choosing a representative for each cluster as the corresponding center [Mood 89]. This is a typical problem of *unsupervised learning*, and algorithms discussed in the relevant chapters later in the book can be employed. The unknown weights, w_i, are then learned via a supervised scheme (i.e., gradient descent algorithm) to minimize the output error. Thus, such schemes use a combination of supervised and unsupervised learning procedures.

An alternative strategy is described in [Chen 91]. A large number of candidate centers is initially chosen from the training vector set. Then, a forward linear regression technique is employed, such as orthogonal least squares, which leads to a

parsimonious set of centers. This technique also provides a way to estimate the order of the model k. A recursive form of the method, which can lead to computational savings, is given in [Gomm 00].

Another method has been proposed based on support vector machines. The idea behind this methodology is to look at the RBF network as a mapping machine, through the kernels, into a high-dimensional space. Then we design a hyperplane classifier using the vectors that are closest to the decision boundary. These are the support vectors and correspond to the centers of the input space. The training consists of a quadratic programming problem and guarantees a global optimum [Scho 97]. The nice feature of this algorithm is that it automatically computes all the unknown parameters including the number of centers. We will return to it later in this chapter.

In [Plat 91] an approach similar in spirit to the constructive techniques, discussed for the multilayer perceptrons, has been suggested. The idea is to start training the RBF network with a few nodes (initially one) and keep growing the network by allocating new ones, based on the "novelty" in the feature vectors that arrive sequentially. The novelty of each training input–desired output pair is determined by two conditions: (a) the input vector to be very far (according to a threshold) from all already existing centers *and* (b) the corresponding output error (using the RBF network trained up to this point) greater than another predetermined threshold. If both conditions are satisfied, then the new input vector is assigned as the new center. If not, the input–desired output pair is used to update the parameters of the network according to the adopted training algorithm, for example, the gradient descent scheme. A variant of this scheme that allows removal of previously assigned centers has also been suggested in [Ying 98]. This is basically a combination of the constructive and pruning philosophies. The procedure suggested in [Kara 97] also moves along the same direction. However, the assignment of the new centers is based on a procedure of progressive splitting (according to a splitting criterion) of the feature space using clustering or learning vector quantization techniques (Chapter 14). The representatives of the regions are then assigned as the centers of the RBF's. As was the case with the aforementioned techniques, network growing and training is performed concurrently. In [Yang 06] a weight structure is imposed that binds the weights to a specified probability density function, and estimation is achieved in the Bayesian framework rationale.

A number of other techniques have also been suggested. For a review see, for example, [Hush 93]. A comparison of RBF networks with different center selection strategies versus multilayer perceptrons in the context of speech recognition is given in [Wett 92]. Reviews involving RBF networks and related applications are given in [Hayk 96, Mulg 96].

4.16 UNIVERSAL APPROXIMATORS

In this section we provide the basic guidelines concerning the approximation properties of the nonlinear functions used throughout this chapter—that is, sigmoid,

polynomial, and radial basis functions. The theorems that are stated justify the use of the corresponding networks as decision surface approximators as well as probability function approximators, depending on how we look at the classifier.

In (4.51) the polynomial expansion was used to approximate the nonlinear function $g(\boldsymbol{x})$. This choice for the approximation functions is justified by the Weierstrass theorem.

Theorem *Let $g(\boldsymbol{x})$ be a continuous function defined in a compact (closed) subset $S \subset \mathcal{R}^l$, and $\epsilon > 0$. Then there are an integer $r = r(\epsilon)$ and a polynomial function $\phi(\boldsymbol{x})$ of order r so that*

$$|g(\boldsymbol{x}) - \phi(\boldsymbol{x})| < \epsilon, \quad \forall \boldsymbol{x} \in S$$

In other words, function $g(\boldsymbol{x})$ can be approximated arbitrarily closely for sufficiently large r. A major problem associated with polynomial expansions is that good approximations are usually achieved for large values of r. That is, the convergence to $g(\boldsymbol{x})$ is slow. In [Barr 93] it is shown that the approximation error is reduced according to an $O(\frac{1}{r^{2/l}})$ rule, where $O(\cdot)$ denotes order of magnitude. *Thus, the error decreases more slowly with increasing dimension l of the input space*, and large values of r are necessary for a given approximation error. However, large values of r, besides the computational complexity and generalization issues (due to the large number of free parameters required), also lead to poor numerical accuracy behavior in the computations, because of the large number of products involved. On the other hand, the polynomial expansion can be used effectively for piecewise approximation, where smaller r's can be adopted.

The slow decrease of the approximation error with respect to the system order and the input space dimension is common to all expansions of the form (4.47) with fixed basis functions $f_i(\cdot)$. The scenario becomes different if data-adaptive functions are chosen, as is the case with the multilayer perceptrons. In the latter, the argument in the activation functions is $f(\boldsymbol{w}^T\boldsymbol{x})$, with $\boldsymbol{w}$ computed in an optimal fashion from the available data.

Let us now consider a two-layer perceptron with one hidden layer, having k nodes with activation functions $f(\cdot)$ and an output node with *linear* activation. The output of the network is then given by

$$\phi(\boldsymbol{x}) = \sum_{j=1}^{k} w_j^o f(\boldsymbol{w}_j^{bT}\boldsymbol{x}) + w_o^o \tag{4.59}$$

where b refers to the weights, including the thresholds, of the hidden layer and o to the weights of the output layer. Provided that $f(\cdot)$ is a squashing function, the following theorem establishes the universal approximation properties of such a network [Cybe 89, Funa 89, Horn 89, Ito 91, Kalo 97].

Theorem *Let $g(\boldsymbol{x})$ be a continuous function defined in a compact subset $S \subset \mathcal{R}^l$ and $\epsilon > 0$. Then there exist $k = k(\varepsilon)$ and a two-layer perceptron (4.59) so that*

$$|g(\boldsymbol{x}) - \phi(\boldsymbol{x})| < \epsilon, \quad \forall \boldsymbol{x} \in S$$

In [Barr 93] is shows that, in contrast to the polynomial expansion, the approximation error decreases according to an $O(\frac{1}{k})$ rule. *In other words, the input space dimension does not enter explicitly into the scene and the error is inversely proportional to the system order, that is, the number of neurons*. Obviously, the price we pay for it is that the optimization process is now nonlinear, with the associated disadvantage of the potential for convergence to local minima. The question that now arises is whether we gain anything by using more than one hidden layer, since a single one is sufficient for the function approximation. An answer is that using more than one layer may lead to a more efficient approximation; that is, the same accuracy is achieved with fewer neurons in the network.

The universal approximation property is also true for the class of RBF functions. For sufficiently large values of k in (4.55) the resulting expansion can approximate arbitrarily closely any continuous function in a compact subset S [Park 91, Park 93].

4.17 PROBABILISTIC NEURAL NETWORKS

In Section 2.5.6 we have seen that the Parzen estimate of an unknown pdf, using a Gaussian kernel, is given by

$$\hat{p}(\boldsymbol{x}|\omega_i) = \frac{1}{N_i}\sum_{i=1}^{N_i}\frac{1}{(2\pi)^{\frac{l}{2}}h^l}\exp\left(-\frac{(\boldsymbol{x}-\boldsymbol{x}_i)^T(\boldsymbol{x}-\boldsymbol{x}_i)}{2h^2}\right) \tag{4.60}$$

where now we have explicitly included in the notation the class dependence, since decisions according to the Bayesian rule rely on the maximum value, with respect to ω_i, of $P(\omega_i)\hat{p}(\boldsymbol{x}|\omega_i)$. Obviously, in Eq. (4.60) only the training samples, $\boldsymbol{x}_i$, $i = 1, 2, \ldots, N_i$, of class ω_i are involved.

The objective of this section is to develop an efficient architecture for implementing Eq. (4.60), which is inspired by the multilayer NN rationale. The critical computation involving the unknown feature vector, $\boldsymbol{x}$, in Eq. (4.60) is the inner product norm

$$(\boldsymbol{x}-\boldsymbol{x}_i)^T(\boldsymbol{x}-\boldsymbol{x}_i) = ||\boldsymbol{x}||^2 + ||\boldsymbol{x}_i||^2 - 2\boldsymbol{x}_i^T\boldsymbol{x} \tag{4.61}$$

Let us now normalize *all* the feature vectors, which are involved in the game, to unit norm. This is achieved by dividing each vector $\boldsymbol{x}$ by its norm $||\boldsymbol{x}|| = \sqrt{\sum_{i=1}^{l} x_i^2}$. After normalization, and combining Eqs. (4.61) and (4.60), Bayesian classification now relies on searching for the maximum of the following discriminant functions

$$g(\omega_i) = \frac{P(\omega_i)}{N_i}\sum_{i=1}^{N_i}\exp\left(\frac{\boldsymbol{x}_i^T\boldsymbol{x}-1}{h^2}\right) \tag{4.62}$$

where the constant multiplicative weights have been omitted. The above can be efficiently implemented by the network of Figure 4.22, when parallel processing resources are available. The input consists of nodes where an unknown feature

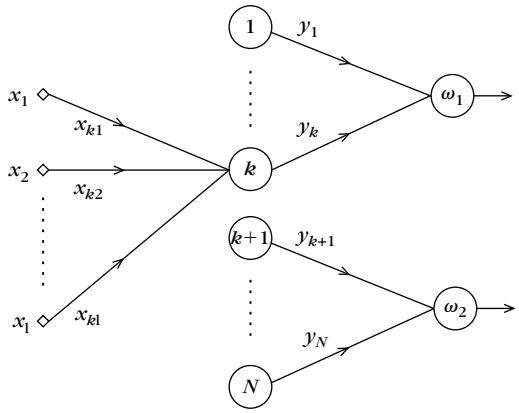

FIGURE 4.22

A probabilistic neural network architecture with N training data points. Each node corresponds to a training point, and it is numbered accordingly. Only the synaptic weights for the kth node are drawn. We have assumed that there are two classes and that the first k points originate from class ω_1 and the rest from class ω_2.

vector $\boldsymbol{x} = [x_1, x_2, \ldots, x_l]^T$ is applied. The number of hidden layer nodes is equal of the number of training data, $N = \sum_{i=1}^{M} N_i$, where M is the number of classes. In the figure, for the sake of simplicity, we have assumed two classes, although generalizing to more classes is obvious. The synaptic weights, leading to the kth hidden node, consist of the components of the respective *normalized* training feature vector $\boldsymbol{x}_k$, i.e., $x_{k,j}$, $j = 1, 2, \ldots, l$ and $k = 1, 2, \ldots, N$. In other words, the training of this type of network is very simple and is directly dictated by the values of the training points. Hence, the input presented to the activation function of the kth hidden layer node is given by

$$input_k = \sum_{j=1}^{l} x_{k,j} x_j = \boldsymbol{x}_k^T \boldsymbol{x}$$

Using as activation function for each node the Gaussian kernel, the output of the kth node is equal to

$$y_k = \exp\left(\frac{input_k - 1}{h^2}\right)$$

There are M output nodes, one for each class. Output nodes are linear combiners. Each output node is connected to *all* hidden layer nodes *associated with the respective class*. The output for the mth output node, $m = 1, 2, \ldots, M$, will be

$$output_m = \frac{P(\omega_m)}{N_m} \sum_{i=1}^{N_m} y_i$$

where N_m is the number of hidden layer nodes (number of training points) associated with the mth class. The unknown vector is classified according to the class giving the maximum output value. Probabilistic neural network architectures were introduced in [Spec 90], and they have been used in a number of applications, for example, [Rome 97, Stre 94, Rutk 04].

4.18 SUPPORT VECTOR MACHINES: THE NONLINEAR CASE

In Chapter 3, we discussed support vector machines (SVM) as an optimal design methodology of a linear classifier. Let us now assume that there exists a mapping

$$\boldsymbol{x} \in \mathcal{R}^l \longrightarrow \boldsymbol{y} \in \mathcal{R}^k$$

from the input feature space into a k-dimensional space, where the classes can satisfactorily be separated by a hyperplane. Then, in the framework discussed in Section 4.12, the SVM method can be mobilized for the design of the hyperplane classifier in the new k-dimensional space. However, there is an elegant property in the SVM methodology that can be exploited for the development of a more general approach. This will also allow us for (implicit) mappings in infinite dimensional spaces, if required.

Recall from Chapter 3 that, in the computations involved in the Wolfe dual representation the feature vectors participate in pairs, via the inner product operation. Also, once the optimal hyperplane $(\boldsymbol{w}, w_0)$ has been computed, classification is performed according to whether the sign of

$$\begin{aligned} g(\boldsymbol{x}) &= \boldsymbol{w}^T \boldsymbol{x} + w_0 \\ &= \sum_{i=1}^{N_s} \lambda_i y_i \boldsymbol{x}_i^T \boldsymbol{x} + w_0 \end{aligned}$$

is $+$ or $-$, where N_s is the number of support vectors. Thus, once more, only inner products enter into the scene. If the design is to take place in the new k-dimensional space, the only difference is that the involved vectors will be the k-dimensional mappings of the original input feature vectors. A naive look at it would lead to the conclusion that now the complexity is much higher, since, usually, k is much higher than the input space dimensionality l, in order to make the classes linearly separable. However, there is a nice surprise just waiting for us. Let us start with a simple example. Assume that

$$\boldsymbol{x} \in \mathcal{R}^2 \longrightarrow \boldsymbol{y} = \begin{bmatrix} x_1^2 \\ \sqrt{2}x_1x_2 \\ x_2^2 \end{bmatrix}$$

Then, it is a matter of simple algebra to show that

$$\boldsymbol{y}_i^T \boldsymbol{y}_j = \left(\boldsymbol{x}_i^T \boldsymbol{x}_j\right)^2$$

In words, the inner product of the vectors in the new (higher dimensional) space has been expressed as a function of the inner product of the corresponding vectors in the original feature space. Most interesting!

Theorem *Mercer's Theorem. Let* $\boldsymbol{x} \in \mathcal{R}^l$ *and a mapping* $\boldsymbol{\phi}$

$$\boldsymbol{x} \mapsto \boldsymbol{\phi}(\boldsymbol{x}) \in H$$

where H is a Hilbert space.[5] *The inner product operation has an equivalent representation*

$$\langle \boldsymbol{\phi}(\boldsymbol{x}), \boldsymbol{\phi}(\boldsymbol{z}) \rangle = K(\boldsymbol{x}, \boldsymbol{z}) \tag{4.63}$$

where $\langle \cdot, \cdot \rangle$ *denotes the inner product operation in H and* $K(\boldsymbol{x}, \boldsymbol{z})$ *is a symmetric continuous function satisfying the following condition:*

$$\int_C \int_C K(\boldsymbol{x}, \boldsymbol{z}) g(\boldsymbol{x}) g(\boldsymbol{z})\, d\boldsymbol{x}\, d\boldsymbol{z} \geq 0 \tag{4.64}$$

for any $g(\boldsymbol{x})$, $\boldsymbol{x} \in C \subset \mathcal{R}^l$ *such that*

$$\int_C g(\boldsymbol{x})^2\, d\boldsymbol{x} < +\infty \tag{4.65}$$

where C is a compact (finite) subset of $\mathcal{R}^l$. *The opposite is always true; that is, for* any *symmetric, continuous function* $K(\boldsymbol{x}, \boldsymbol{z})$ *satisfying (4.64) and (4.65) there exists a space in which* $K(\boldsymbol{x}, \boldsymbol{z})$ *defines an inner product! Such functions are also known as kernels and the space H as* Reproducing kernel Hilbert space (RKHS) *(e.g., [Shaw 04, Scho 02]). What Mercer's theorem does not disclose to us, however, is how to find this space. That is, we do not have a general tool to construct the mapping* $\boldsymbol{\phi}(\cdot)$ *once we know the inner product of the corresponding space. Furthermore, we lack the means to know the dimensionality of the space, which can even be infinite. This is the case, for example, for the radial basis (Gaussian) kernel ([Burg 99]). For more on these issues, the mathematically inclined reader is referred to [Cour 53].*

Typical examples of kernels used in pattern recognition applications are as follows:

Polynomials

$$K(\boldsymbol{x}, \boldsymbol{z}) = (\boldsymbol{x}^T \boldsymbol{z} + 1)^q, \quad q > 0 \tag{4.66}$$

[5] A Hilbert space is a complete linear space equipped with an inner product operation. A finite dimensional Hilbert space is a Euclidean space.

Radial Basis Functions

$$K(\boldsymbol{x}, \boldsymbol{z}) = \exp\left(-\frac{\|\boldsymbol{x} - \boldsymbol{z}\|^2}{\sigma^2}\right) \tag{4.67}$$

Hyperbolic Tangent

$$K(\boldsymbol{x}, \boldsymbol{z}) = \tanh\left(\beta \boldsymbol{x}^T \boldsymbol{z} + \gamma\right) \tag{4.68}$$

for appropriate values of β and γ so that Mercer's conditions are satisfied. One possibility is $\beta = 2$, $\gamma = 1$. In [Shaw 04] a unified treatment of kernels is presented, focusing on their mathematical properties as well as methods for pattern recognition and regression that have been developed around them.

Once an appropriate kernel has been adopted that implicitly defines a mapping into a higher dimensional space (RKHS), the Wolfe dual optimization task (Eqs. (3.103)–(3.105)) becomes

$$\max_{\boldsymbol{\lambda}} \left(\sum_i \lambda_i - \frac{1}{2} \sum_{i,j} \lambda_i \lambda_j y_i y_j K(\boldsymbol{x}_i, \boldsymbol{x}_j) \right) \tag{4.69}$$

$$\text{subject to} \quad 0 \leq \lambda_i \leq C, \quad i = 1, 2, \ldots, N \tag{4.70}$$

$$\sum_i \lambda_i y_i = 0 \tag{4.71}$$

and the resulting linear (in the RKHS) classifier is

$$\text{assign } \boldsymbol{x} \text{ in } \omega_1(\omega_2) \text{ if } g(\boldsymbol{x}) = \sum_{i=1}^{N_s} \lambda_i y_i K(\boldsymbol{x}_i, \boldsymbol{x}) + w_0 > (<)\ 0 \tag{4.72}$$

Due to the nonlinearity of the kernel function, the resulting classifier is a nonlinear one in the original $\mathcal{R}^l$ space. Similar arguments hold true for the ν-SVM formulation.

Figure 4.23 shows the corresponding architecture. This is nothing else than a special case of the generalized linear classifier of Figure 4.17. The number of nodes is determined by the number of support vectors N_s. The nodes perform the inner products between the mapping of $\boldsymbol{x}$ and the corresponding mappings of the support vectors in the high-dimensional space, via the kernel operation.

Figure 4.24 shows the resulting SVM classifier for two nonlinearly separable classes, where the Gaussian radial basis function kernel, with $\sigma = 1.75$, has been used. Dotted lines mark the margin and circled points the support vectors.

Remarks

- Notice that if the kernel function is the RBF, then the architecture is the same as the RBF network architecture of Figure 4.17. However, the approach followed here is different. In Section 4.15, a mapping in a k-dimensional space

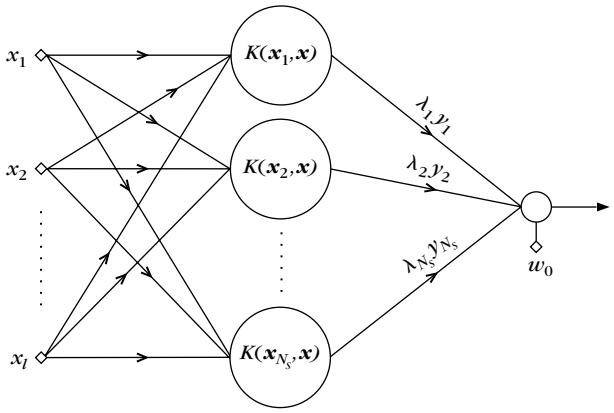

FIGURE 4.23

The SVM architecture employing kernel functions.

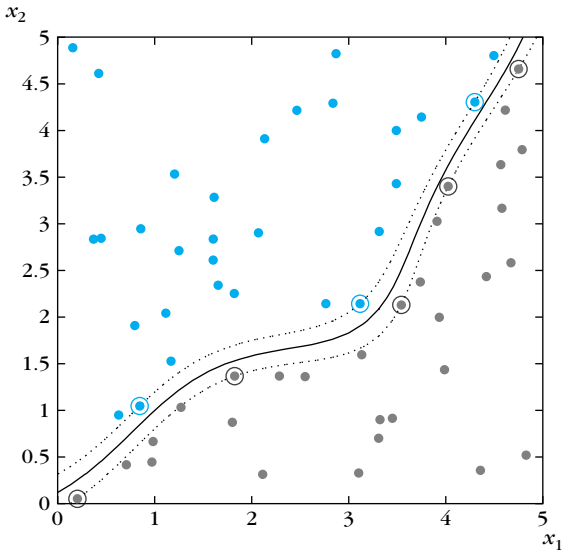

FIGURE 4.24

Example of a nonlinear SVM classifier for the case of two nonlinearly separable classes. The Gaussian RBF kernel was used. Dotted lines mark the margin and circled points the support vectors.

was first performed, and the centers of the RBF functions had to be estimated. In the SVM approach, the number of nodes as well as the centers are the result of the optimization procedure.

- The hyperbolic tangent function is a sigmoid one. If it is chosen as a kernel, the resulting architecture is a special case of a two-layer perceptron. Once more, the number of nodes is the result of the optimization procedure. This is important. Although the SVM architecture is the same as that of a two-layer perceptron, the training procedure is entirely different for the two methods. The same is true for the RBF networks.

- A notable characteristic of the support vector machines is that the computational complexity is independent of the dimensionality of the kernel space, where the input feature space is mapped. Thus, the curse of dimensionality is bypassed. In other words, one designs in a high-dimensional space without having to adopt explicit models using a large number of parameters, as this would be dictated by the high dimensionality of the space. This also has an influence on the generalization properties, and indeed, SVMs tend to exhibit *good generalization performance*. We will return to this issue at the end of Chapter 5.

- A major limitation of the support vector machines is that up to now there has been no efficient practical method for selecting the best kernel function. This is still an unsolved, yet challenging, research issue. Once a kernel function has been adopted, the so-called kernel parameters (e.g., σ for the Gaussian kernel) as well as the smoothing parameter, C, in the cost function are selected so that the error performance of the resulting classifier can be optimized. Indeed, this set of parameters, also known as hyperparameters, is crucial for the generalization capabilities of the classifier (i.e., its error performance when it is "confronted" with data outside the training set).

  To this end, a number of easily computed bounds, which relate to the generalization performance of the classifier, have been proposed and used for the best choice of the hyperparameters. The most common procedure is to solve the SVM task for different sets of hyperparameters and finally select the SVM classifier corresponding to the set optimizing the adopted bound. See, for example, [Bart 02, Lin 02, Duan 03, Angu 03, Lee 04]. [Chap 02] treats this problem in a minimax framework: maximize the margin over the $\boldsymbol{w}$ and minimize the bound over the hyperparameters.

  A different approach to the task of data-adaptive kernel tuning, with the same goal of improving the error performance, is to use information geometry arguments [Amar 99]. The basic idea behind this approach is to introduce a conformal mapping into the Riemannian geometry induced by the chosen kernel function, aiming at enhancing the margin. [Burg 99] points out that the feature vectors, which originally lie in the l-dimensional space, after the mapping induced by the kernel function lie in an l-dimensional surface, S,

in the high-dimensional space. It turns out that (under some very general assumptions) S is a Riemannian manifold with a metric that can be expressed solely in terms of the kernel.

- Support vector machines have been applied to a number of diverse applications, ranging from handwritten digit recognition ([Cort 95]), to object recognition ([Blan 96]), person identification ([Ben 99]), spam categorization ([Druc 99]), channel equalization ([Seba 00]), and medical imaging [ElNa 02, Flao 06]. The results from these applications indicate that SVM classifiers exhibit enhanced generalization performance, which seems to be the power of support vector machines. An extensive comparative study concerning the performance of SVM against sixteen other popular classifiers, using twenty-one different data sets, is given in [Meye 03]. The results verify that SVM classifiers rank at the very top among these classifiers, although there are cases for which other classifiers gave lower error rates.

4.19 BEYOND THE SVM PARADIGM

One of the most attractive properties of the support vector machines, which has contributed to their popularity, is that their computational structure allows for the use of a kernel function, as discussed in the previous section. Sometimes this is also known as the *kernel trick*. This powerful tool makes the design of a linear classifier in the high-dimensional space independent of the dimensionality of this space. Moreover, due to the implicit nonlinear mapping, dictated by the adopted kernel function, the designed classifier is a nonlinear one in the original space. The success of the SVMs in practice has inspired a research effort to extend a number of linear classifiers to nonlinear ones, by embedding the kernel trick in their structure. This is possible if all the computations can be expressed in terms of inner product operations. Let us illustrate the procedure for the case of the classical Euclidean distance classifier.

Assume two classes ω_1 and ω_2, with N_1 and N_2 training pairs, $(y_i, \boldsymbol{x}_i)$, respectively, with $y_i = \pm 1$ being the class label of the ith sample. Let $K(\cdot, \cdot)$ be a kernel function associated with an implicit mapping $\boldsymbol{x} \mapsto \boldsymbol{\phi}(\boldsymbol{x})$ from the original $\mathcal{R}^l$ space to a high-dimensional RKHS. Given an unknown $\boldsymbol{x}$, the Euclidean classifier, in the RKHS, classifies it to the ω_2 class if

$$||\boldsymbol{\phi}(\boldsymbol{x}) - \boldsymbol{\mu}_1||^2 > ||\boldsymbol{\phi}(\boldsymbol{x}) - \boldsymbol{\mu}_2||^2 \tag{4.73}$$

or, after some basic algebra, if

$$\langle \phi(\boldsymbol{x}), (\boldsymbol{\mu}_2 - \boldsymbol{\mu}_1) \rangle > \frac{1}{2} \left(||\boldsymbol{\mu}_2||^2 - ||\boldsymbol{\mu}_1||^2 \right) \equiv \theta \tag{4.74}$$

where $\langle \cdot, \cdot \rangle$ denotes the inner product operation in the RKHS and

$$\boldsymbol{\mu}_1 = \frac{1}{N_1} \sum_{i:y_i=+1} \boldsymbol{\phi}(\boldsymbol{x}_i) \quad \text{and} \quad \boldsymbol{\mu}_2 = \frac{1}{N_2} \sum_{i:y_i=-1} \boldsymbol{\phi}(\boldsymbol{x}_i) \tag{4.75}$$

Combining Eqs. (4.74) and (4.75), we conclude that we assign $\boldsymbol{x}$ in ω_2 if

$$\frac{1}{N_2}\sum_{i:y_i=-1} K(\boldsymbol{x},\boldsymbol{x}_i) - \frac{1}{N_1}\sum_{i:y_i=+1} K(\boldsymbol{x},\boldsymbol{x}_i) > \theta \tag{4.76}$$

where

$$2\theta = \frac{1}{N_2^2}\sum_{i:y_i=-1}\sum_{j:y_j=-1} K(\boldsymbol{x}_i,\boldsymbol{x}_j) - \frac{1}{N_1^2}\sum_{i:y_i=+1}\sum_{j:y_j=+1} K(\boldsymbol{x}_i,\boldsymbol{x}_j)$$

The left-hand side in formula (4.76) reminds us of the Parzen pdf estimate. Adopting the Gaussian kernel, the first term can be taken as the pdf estimator associated with the class ω_2 and the second one with the ω_1 one. Besides the Euclidean classifier, other classical cases, including Fisher's linear discriminant (Chapter 5), have been extended to nonlinear ones by employing the kernel trick, see, for example, [Mull 01, Shaw 04]. Another notable and pedagogically attractive example of a "kernelized" version of a linear classifier is the kernel perceptron algorithm.

The perceptron rule was introduced in Section 3.3. There it was stated that the perceptron algorithm converges in a finite number of steps, provided that the two classes are linearly separable. This drawback has prevented the perceptron algorithm to be used in realistic practical applications. However, after mapping the original feature space to a high-dimensional (even of infinite dimensionality) space and utilizing Cover's theorem (Section 4.13), one expects the classification task to be linearly separable, with high probability, in the RKHS space. In this perspective, the kernelized version of the perceptron rule transcends its historical, theoretical, and educational role and asserts a more practical value as a candidate for solving linearly separable tasks in the RKHS. We will choose to work on the perceptron algorithm in its reward and punishment form, given in Eqs. (3.21)–(3.23).

The heart of the method is the update given by the Eqs. (3.21), and (3.22). These recursions, take place in the extended RKHS (its dimension is increased by one to account for the bias term), and they are compactly written as

$$\begin{bmatrix} \boldsymbol{w}(t+1) \\ w_0(t+1) \end{bmatrix} = \begin{bmatrix} \boldsymbol{w}(t) \\ w_0(t) \end{bmatrix} + y_{(t)} \begin{bmatrix} \boldsymbol{\phi}(\boldsymbol{x}_{(t)}) \\ 1 \end{bmatrix}$$

each time a misclassification occurs—that is, if $y_{(t)}(\langle \boldsymbol{w}(t), \boldsymbol{\phi}(\boldsymbol{x}_{(t)})\rangle + w_0) \leq 0$, where the coefficient ρ has been taken to be equal to one. Let α_i, $i = 1, 2, \ldots, N$, be a counter corresponding to each one of the training points. The counter α_i is increased by one every time $\boldsymbol{x}_{(t)} = \boldsymbol{x}_i$, *and* a misclassification occurs leading to a respective update of the classifier. If one starts from a zero initial vector, then the solution, after all points have been correctly classified, can be written as

$$\boldsymbol{w} = \sum_{i=1}^{N} \alpha_i y_i \boldsymbol{\phi}(\boldsymbol{x}_i), \; w_0 = \sum_{i=1}^{N} \alpha_i y_i$$

The final resulting nonlinear classifier, in the original feature space, then becomes

$$g(\boldsymbol{x}) \equiv \langle \boldsymbol{w}, \boldsymbol{\phi}(\boldsymbol{x}) \rangle + w_0 = \sum_{i=1}^{N} \alpha_i y_i K(\boldsymbol{x}, \boldsymbol{x}_i) + \sum_{i=1}^{N} \alpha_i y_i$$

A pseudocode for the kernel perceptron algorithm follows.

The Kernel Perceptron Algorithm

- Set $\alpha_i = 0,\ i = 1, 2, \ldots, N$
- Repeat
  - count_misclas = 0
  - For $i = 1$ to N
    - If $y_i \left(\sum_{j=1}^{N} \alpha_j y_j K(\boldsymbol{x}_i, \boldsymbol{x}_j) + \sum_{j=1}^{N} \alpha_j y_j \right) \leq 0$ then
      - $\alpha_i = \alpha_i + 1$
      - count_misclas = count_misclas + 1
  - End {For}
- Until count_misclas = 0

4.19.1 Expansion in Kernel Functions and Model Sparsification

In this subsection, we will briefly discuss classifiers that resemble to (or, even are inspired by) the SVMs, in an effort to establish bridges among different methodologies. We have already done so in Section 4.18 for the SVM, RBF, and multilayer neural networks. After all, it is a small world! Using the Gaussian kernel in Eq. (4.72), we obtain

$$g(\boldsymbol{x}) = \sum_{i=1}^{N_s} a_i \exp\left(-\frac{||\boldsymbol{x} - \boldsymbol{x}_i||^2}{2\sigma^2} \right) + w_0 \tag{4.77}$$

where we have used $a_i = \lambda_i y_i$. Equation (4.77) is very similar to the Parzen expansion of a pdf, discussed in Chapter 2. There are a few differences, however. In contrast to the Parzen expansion, $g(\boldsymbol{x})$ in (4.77) is not a pdf function; that is, in general, it does not integrate to unity. Moreover, from the practical point of view, the most important difference lies in the different number of terms involved in the summation. In the Parzen expansion *all* the training samples offer their contribution to the final solution. In contrast, in the solution provided by the SVM formulation only the support vectors, that is, the points lying either in the margin or in the wrong side of the resulting classifier, are assigned as the "significant" ones and are selected to contribute to the solution. In practice, a small fraction of the training points enter in the summation in Eq. (4.77), that is $N_s \ll N$. In fact, as we

will discuss at the end of Section 5.10, if the number of support vectors gets large, the generalization performance of the classifier is expected to degrade. If $N_s \ll N$, we say that the solution is *sparse*. A sparse solution spends computational resources only on the most relevant of the training patterns. Besides the computational complexity aspects, having a sparse solution is in line with our desire to avoid overfitting (see also Section 4.9). In real-world data, the presence of noise in regression and the overlap of classes in classification, as well as the presence of outliers, imply that the modeling must be such that to avoid overfitting to the specific training data set.

A closer look behind the SVM philosophy reveals that the source of sparsity in the obtained solution is the presence of the margin term in the cost function. Another way to view the term $\|\boldsymbol{w}\|^2$ in the cost function in (3.93), that is,

$$J(\boldsymbol{w}, w_0, \boldsymbol{\xi}) = \frac{1}{2}\|\boldsymbol{w}\|^2 + C\sum_{i=1}^{N} I(\xi_i)$$

is as a *regularization* term, whose presence satisfies our will to keep the norm of the solution as "small" and "simple" as possible, while, at the same time, trying to minimize the number of margin errors $\left(\sum_{i=1}^{N} I(\xi_i)\right)$. This implicitly forces most of the λ_is in the solution to be zero, keeping only the most significant of the samples, that is, the support vectors. In Section 4.9, regularization was also used in order to keep the size of the neural networks small. For a deeper and an elegant discussion of the use of regularization in the context of regression/classification the interested reader can refer to [Vapn 00].

With the sparsification goal in mind, a major effort has been invested to develop techniques, both for classification and for regression tasks, which lead to classifiers/regressors of the form

$$g(\boldsymbol{x}) = \sum_{j=1}^{N} a_j K(\boldsymbol{x}, \boldsymbol{x}_j) \tag{4.78}$$

for an appropriately chosen kernel function. A bias constant term can also be added, but it has been omitted for simplicity. The task is to estimate the unknown weights a_j, $j = 1, 2 \ldots, N$, of the expansion. Functions of the form in (4.78) are justified by the following theorem ([Kime 71, Scho 02]):

Representer Theorem
Let $\mathcal{L}(\cdot, \cdot) : \mathcal{R}^2 \mapsto [0, \infty)$ be an arbitrary nonnegative loss function, measuring the deviation between a desired response, y, and the value of $g(\boldsymbol{x})$. Then the minimizer $g(\cdot) \in H$, where H is a RKHS defined by a kernel function $K(\cdot, \cdot)$, of the regularized cost

$$\sum_{i=1}^{N} \mathcal{L}(g(\boldsymbol{x}_i), y_i) + \Omega(||g||) \tag{4.79}$$

admits a representation of the form in (4.78). In (4.79), $(y_i, \boldsymbol{x}_i)$, $i = 1, 2, \ldots, N$, are the training data, $\Omega(\cdot) : [0, \infty) \mapsto \mathcal{R}$ is a strictly monotonic increasing function and $||\cdot||$ is the norm operation in H. For a more mathematical treatment of this result, the interested reader may refer to, for example, [Scho 02]. For those who are not familiar with functional analysis and some of the mathematical secrets behind RKHS, recall that the set of functions $\mathcal{R}^l \mapsto \mathcal{R}$ form a linear space, which can be equipped with an inner product operation to become a Hilbert space. Hence, by restricting $g(\cdot) \in H$, we limit our search for solutions in (4.79) among the points in an RKHS (function space) defined by the specific kernel function.

This is an important theorem because it states that, although working in a high- (even infinite) dimensional space, the optimal solution, minimizing (4.79), is expressed as a linear combination of only N kernels *placed at the training points*! In order to see how this theorem can simplify the search for the optimal solution in practice, let us consider the following example.

Example 4.1

The kernel least squares solution. Let $(y_i, \boldsymbol{x}_i)$, $i = 1, 2, \ldots, N$, be the training points. The goal is to design the optimal linear least squares classifier (regressor) in a RKHS space, which is defined by the kernel function $K(\cdot, \cdot)$.

According to the definition of the least squares cost in Section 3.4.3, we have to minimize, with respect to $g \in H$, the cost

$$\sum_{i=1}^{N} \big(y_i - g(\boldsymbol{x}_i)\big)^2 \tag{4.80}$$

According to the Representer Theorem, we can write

$$g(\boldsymbol{x}) = \sum_{j=1}^{N} a_j K(\boldsymbol{x}, \boldsymbol{x}_j) \tag{4.81}$$

Substituting (4.81) into (4.80), we get the equivalent task of minimizing with respect to a *finite* number of parameters, a_i, $i = 1, 2, \ldots, N$, the cost

$$J(\boldsymbol{a}) = \sum_{i=1}^{N} \left(y_i - \sum_{j=1}^{N} a_j K(\boldsymbol{x}_i, \boldsymbol{x}_j) \right)^2 \tag{4.82}$$

The cost in (4.82) can be written in terms of the Euclidean norm in the $\mathcal{R}^N$ space, that is,

$$J(\boldsymbol{a}) = (\boldsymbol{y} - \mathcal{K}\boldsymbol{a})^T (\boldsymbol{y} - \mathcal{K}\boldsymbol{a}) \tag{4.83}$$

where $\boldsymbol{y} = [y_1, y_2, \ldots, y_N]^T$ and $\mathcal{K}$ is the $N \times N$ matrix known as the ***Gram*** matrix, which is defined as

$$\mathcal{K}(i, j) \equiv K(\boldsymbol{x}_i, \boldsymbol{x}_j) \tag{4.84}$$

Expanding (4.83) and taking the gradient with respect to $\boldsymbol{a}$ to be equal to zero, we obtain

$$\boldsymbol{a} = \mathcal{K}^{-1}\boldsymbol{y} \tag{4.85}$$

provided that the Gram matrix is invertible. Hence, recalling (4.81), the kernel least squares estimate can be written compactly as

$$g(\boldsymbol{x}) = \boldsymbol{a}^T\boldsymbol{p} = \boldsymbol{y}^T\mathcal{K}^{-1}\boldsymbol{p} \tag{4.86}$$

where

$$\boldsymbol{p} \equiv [K(\boldsymbol{x}, \boldsymbol{x}_1), \ldots, K(\boldsymbol{x}, \boldsymbol{x}_N)]^T \tag{4.87}$$

The Representer Theorem has been exploited in [Tipp 01] in the context of the so-called *relevance vector machine* (RVM) methodology. Based on (4.78), a conditional probability model is built for the desired response (label) given the values of $\boldsymbol{a}$. The computation of the unknown weights is carried out in the Bayesian framework rationale (Chapter 2). Sparsification is achieved by constraining the unknown weight parameters and imposing an explicit prior probability distribution over each one of them. It is reported that RVMs lead to sparser solutions compared to the SVMs, albeit at a higher complexity. Memory requirements scale with the square, and the required computational resources scale with the cube of the number of basis functions, which makes the algorithm less practical for large data sets. In contrast, the amount of memory requirements for the SVMs is linear, and the number of computations is somewhere between linear and (approximately) quadratic in the size of the training set ([Plat 99]).

A more recent trend is to obtain solutions of the form in Eq. (4.78) in an online time-adaptive fashion. That is, the solution is updated each time a new training pair $(y_i, \boldsymbol{x}_i)$ is received. This is most important when the statistics of the involved data are slowly time varying. To this end, in [Kivi 04] a kernelized online LMS-type algorithm (see Section 3.4.2) is derived that minimizes the cost function

$$J(g_t) = \sum_{i=1}^{t} \mathcal{L}(g_t(\boldsymbol{x}_i), y_i) + \lambda ||g_t||^2 \tag{4.88}$$

where the index t has been used in g_t to denote the time dependence explicitly. $\mathcal{L}(\cdot, \cdot)$ is a loss function that quantifies the deviation between the desired output value, y_i, and the true one that is provided by the current estimate, $g_t(\cdot)$, of the unknown function. The summation accounts for the total number of errors committed on all the samples that have been received up to the time instant t. Sparsification is achieved by regularizing the cost function by the square norm $||g_t||^2$ of the required solution.

Another way to look at the regularization term and better understand how its presence beneficially affects the sparsification process is the following. Instead of minimizing, for example, (4.88) one can choose to work with an alternative formulation of the optimization task, that is,

$$\text{minimize} \quad \sum_{i=1}^{t} \mathcal{L}(g_t(\boldsymbol{x}_i), y_i) \tag{4.89}$$

$$\text{subject to} \quad ||g_t||^2 \leq s \tag{4.90}$$

The use of Lagrange multipliers leads to minimizing $J(g_t)$ in (4.88). It can be shown (see, e.g., [Vapn 00]) that the two problems are equivalent for appropriate choices of the parameters s and λ. However, formulating the optimization task as in (4.89)–(4.90) makes our desire for constraining the size of the solution explicitly stated.

In [Slav 08] an adaptive solution of the cost in Eq. (4.78) is given based on projections and convex set arguments. Sparsification is achieved by constraining the solution to lie within a (hyper)sphere in the RKHS. It is shown that such a constraint becomes equivalent to imposing a forgetting factor that forces the algorithm to forget data in the remote past and adaptation focuses on the most recent samples. The algorithm scales linearly with the number of data corresponding to its effective memory (due to the forgetting factor). An interesting feature of this algorithmic scheme is that it provides as special cases a number of well-known algorithms, such as the kernelized normalized LMS (NLMS) [Saye 04] and the kernelized affine projection [Slav 08a] algorithms. Another welcome feature of this methodology is that it can accommodate differentiable as well as nondifferentiable cost functions, in a unified way, due to the possibility of employing subdifferentials of the cost function, in place of the gradient in the correction term, in each time-update recursion.

A different root to the online sparsification is followed in [Enge 04, Slav 08a]. A dictionary of basis functions is adaptively formed. For each received sample, its dependence on the samples that are already contained in the dictionary is tested, according to a predefined criterion. If the dependence measure is below a threshold value, the new sample is included in the dictionary whose cardinality is now increasing by one; otherwise the dictionary remains unaltered. It is shown that the size of the dictionary does not increase indefinitely and that it remains finite. The expansion of the solution is carried out by using only the basis functions associated with the samples in the dictionary. A pitfall of this technique is that the complexity scales with the square of the size of the dictionary, as opposed to the linear complexity of the two previous adaptive techniques.

In our discussion so far we have assumed the use of a loss function. The choice of the loss function is user-dependent. Some typical choices that can be used and have frequently been adopted in classification tasks are as follows:

- *Soft margin loss*

$$\mathcal{L}(g(\boldsymbol{x}), y) = \max(0, \rho - yg(\boldsymbol{x}))$$

  where ρ defines the margin parameter. In words, a (margin) error is committed if $yg(\boldsymbol{x})$ cannot achieve a value of at least ρ. For smaller values, the loss function becomes positive, and it is also linearly increasing as the value of $yg(\boldsymbol{x})$ becomes smaller moving toward negative values. That is, it provides a measure of how far from the margin the estimate lies. Figure 4.25 shows the respective graph for $\rho = 0$.

- *Exponential loss*

$$\mathcal{L}(g(\boldsymbol{x}), y) = \exp\big(-yg(\boldsymbol{x})\big)$$

As shown in Figure 4.25, this loss function penalizes heavily nonpositive values of $yg(\boldsymbol{x})$, which lead to wrong decisions. We will use this loss function very soon in Section 4.22.

- *Logistic loss*

$$\mathcal{L}(g(\boldsymbol{x}), y) = \ln\big(1 + \exp(-yg(\boldsymbol{x}))\big)$$

The logistic loss function is basically the negative log-likelihood of a logistic-like probabilistic model, discussed in Section 3.6, operating in the RKHS. Indeed, interpreting $g(\boldsymbol{x})$ as a linear function in the RKHS, that is, $g(\boldsymbol{x}) = \langle \boldsymbol{w}, \boldsymbol{\phi}(\boldsymbol{x}) \rangle + w_0$, and modeling the probability of the class label as

$$P(y|\boldsymbol{x}) = \frac{1}{1 + \exp\big(-y(w_0 + \langle \boldsymbol{w}, \boldsymbol{\phi}(\boldsymbol{x}) \rangle)\big)}$$

then the logistic loss is the respective negative log-likelihood function. This loss function has also been used in the context of support vector machines, see [Keer 05].

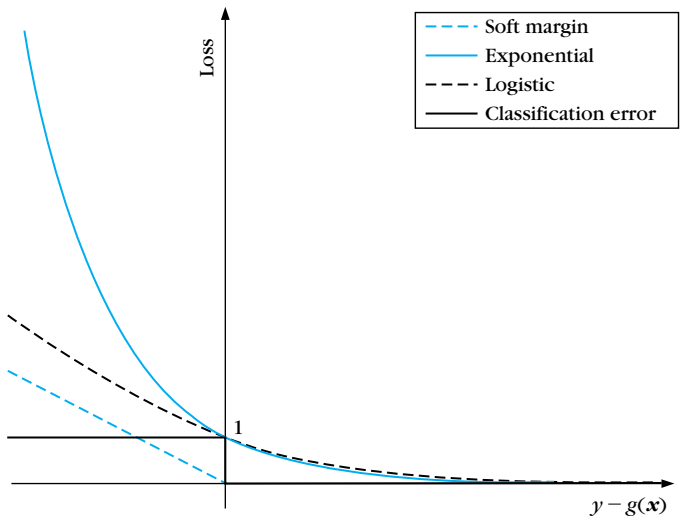

FIGURE 4.25

Typical loss functions used in classification tasks. The margin parameter ρ for the soft margin loss has been set equal to 0. The logistic loss has been normalized in order to pass through the [1,0] point, to facilitate the comparison among the different loss functions. The classification error loss function scores a one if an error is committed and zero otherwise.

4.19.2 Robust Statistics Regression

The regression task was introduced in Section 3.5.1. Let $y \in \mathcal{R}, \boldsymbol{x} \in \mathcal{R}^l$ be two statistically dependent random entities. Given a set of training samples $(y_i, \boldsymbol{x}_i)$, the goal is to compute a function $g(\boldsymbol{x})$ that optimally estimates the value of y when $\boldsymbol{x}$ is measured. In a number of cases, mean square or least squares type of costs are not the most appropriate ones. For example, in cases where the statistical distribution of the data has long tails, then using the least squares criterion will lead to a solution dominated by a small number of points that have very large values (outliers). A similar situation can occur from incorrectly labeled data. Take, for example, a single training data point whose target value has been incorrectly labeled by a large amount. This point will have an unjustifiably (by the true statistics of the data) strong influence on the solution. Such situations can be handled more efficiently by using alternative cost functions, which are known as *robust statistics* loss functions. Typical examples of such loss functions are:

- *Linear ϵ-insensitive loss*

$$\mathcal{L}(g(\boldsymbol{x}), y) = |y - g(\boldsymbol{x})|_\epsilon \equiv \max(0, |y - g(\boldsymbol{x})| - \epsilon)$$

- *Quadratic ϵ-insensitive loss*

$$\mathcal{L}(g(\boldsymbol{x}), y) = |y - g(\boldsymbol{x})|_\epsilon^2 \equiv \max(0, |y - g(\boldsymbol{x})|^2 - \epsilon)$$

- *Huber loss*

$$\mathcal{L}(g(\boldsymbol{x}), y) = \begin{cases} c\,|y - g(\boldsymbol{x})| - \frac{c^2}{2} & \text{if } |y - g(\boldsymbol{x})| > c \\ \frac{1}{2}\left(y - g(\boldsymbol{x})\right)^2 & \text{if } |y - g(\boldsymbol{x})| \leq c \end{cases}$$

where ϵ and c are user-defined parameters. Huber's loss function reduces from quadratic to linear the contributions of samples with absolute error values greater than c. Such a choice makes the optimization task less sensitive to outliers. Figure 4.26 shows the curves associated with the previous loss functions. In the sequel, we will focus on the linear ϵ-insensitive loss.

We are by now experienced enough to solve for the nonlinear $g(\boldsymbol{x})$ case by expressing the problem as a linear one in an RKHS. For the linear ϵ-insensitive case, nonzero contributions to the cost have samples with error values $|y - g(\boldsymbol{x})|$ larger than ϵ. This setup can be compactly expressed by adopting two slack variables, $\xi,\ \xi^*$, and the optimization task is now cast as

$$\text{minimize} \quad J(\boldsymbol{w}, w_0, \boldsymbol{\xi}, \boldsymbol{\xi}^*) = \frac{1}{2}\|\boldsymbol{w}\|^2 + C\left(\sum_{i=1}^{N} \xi_i + \sum_{i=1}^{N} \xi_i^*\right) \tag{4.91}$$

$$\text{subject to} \quad y_i - \left\langle \boldsymbol{w}, \boldsymbol{\phi}(\boldsymbol{x}_i) \right\rangle - w_0 \leq \epsilon + \xi_i^*, \quad i = 1, 2, \ldots, N \tag{4.92}$$

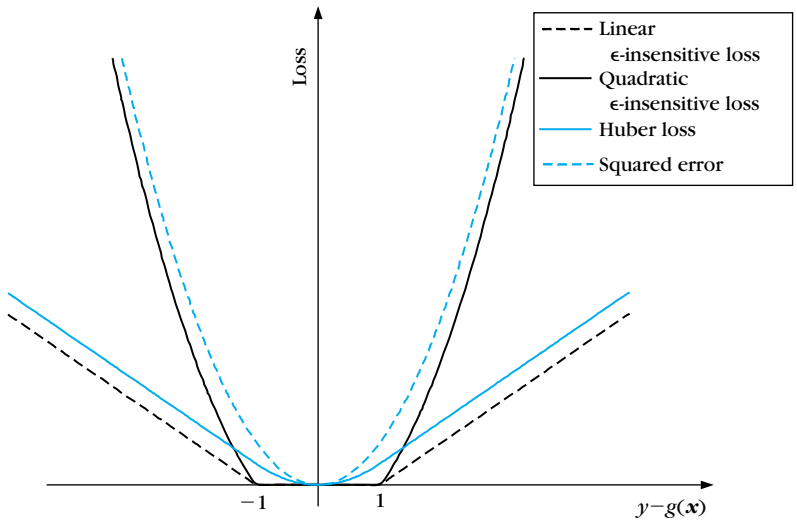

FIGURE 4.26

Loss functions used for regression tasks. The parameters ϵ and c have been set equal to one. In Huber's loss, observe the change from quadratic to linear beyond $\pm c$.

$$\langle \boldsymbol{w}, \boldsymbol{\phi}(\boldsymbol{x}_i) \rangle + w_0 - y_i \leq \epsilon + \xi_i, \quad i = 1, 2, \ldots, N \tag{4.93}$$

$$\xi_i \geq 0, \quad \xi_i^* \geq 0, \quad i = 1, 2, \ldots, N \tag{4.94}$$

The above setup guarantees that ξ_i, ξ_i^* are zero if $|y_i - \langle \boldsymbol{w}, \boldsymbol{\phi}(\boldsymbol{x}_i) \rangle - w_0| \leq \epsilon$ and contribution to the cost function occurs if either $y_i - \langle \boldsymbol{w}, \boldsymbol{\phi}(\boldsymbol{x}_i) \rangle - w_0 > \epsilon$ or if $y_i - \langle \boldsymbol{w}, \boldsymbol{\phi}(\boldsymbol{x}_i) \rangle - w_0 < -\epsilon$. The presence of the norm $||\boldsymbol{w}||$ guards against overfitting, as has already been discussed. Following similar arguments made in Section 3.7.2, it turns out that the solution is given by

$$\boldsymbol{w} = \sum_{i=1}^{N} (\lambda_i^* - \lambda_i) \boldsymbol{\phi}(\boldsymbol{x}_i) \tag{4.95}$$

where λ_i^*, λ_i are the Lagrange multipliers associated with the set of constraints in (4.92)–(4.93), respectively. The corresponding KKT conditions are (in analogy of (3.98)–(3.102))

$$\lambda_i^* (y_i - \langle \boldsymbol{w}, \boldsymbol{\phi}(\boldsymbol{x}_i) \rangle - w_0 - \epsilon - \xi_i^*) = 0, \quad i = 1, 2, \ldots, N \tag{4.96}$$

$$\lambda_i (\langle \boldsymbol{w}, \boldsymbol{\phi}(\boldsymbol{x}_i) \rangle + w_0 - y_i - \epsilon - \xi_i) = 0, \quad i = 1, 2, \ldots, N \tag{4.97}$$

$$C - \lambda_i - \mu_i = 0, \; C - \lambda_i^* - \mu_i^* = 0, \quad i = 1, 2, \ldots, N \tag{4.98}$$

$$\mu_i \xi_i = 0,\ \mu_i^* \xi_i^* = 0, \quad i = 1, 2, \ldots, N \tag{4.99}$$

$$\lambda_i \geq 0,\ \lambda_i^* \geq 0,\ \mu_i \geq 0,\ \mu_i^* \geq 0, \quad i = 1, 2, \ldots, N \tag{4.100}$$

$$\sum_{i=1}^{N} \lambda_i = \sum_{i=1}^{N} \lambda_i^* \tag{4.101}$$

$$\xi_i \xi_i^* = 0,\ \lambda_i \lambda_i^* = 0, \quad i = 1, 2, \ldots, N \tag{4.102}$$

where $\mu_i^*,\ \mu_i$ are the Lagrange multipliers associated with the set of constraints in (4.94). Note that $\xi_i,\ \xi_i^*$ cannot be nonzero simultaneously, and the same applies for the Lagrange multipliers $\lambda_i^*,\ \lambda_i$. Furthermore, a careful look at the KKT conditions reveals that:

- The points with absolute error values strictly less than ϵ, i.e., $|y_i - \langle \boldsymbol{w}, \boldsymbol{\phi}(\boldsymbol{x}_i) \rangle - w_0| < \epsilon$ result in *zero* Lagrange multipliers, $\lambda_i,\ \lambda_i^*$. This is a direct consequence of (4.96) and (4.97). These points are the counterparts of the points that lie strictly outside the margin in the SVM classification task.
- Support vectors are those points satisfying the inequality $|y_i - \langle \boldsymbol{w}, \boldsymbol{\phi}(\boldsymbol{x}_i) \rangle - w_0| \geq \epsilon$.
- The points associated with errors satisfying the strict inequality $|y_i - \langle \boldsymbol{w}, \boldsymbol{\phi}(\boldsymbol{x}_i) \rangle - w_0| > \epsilon$ result in either $\lambda_i = C$ or $\lambda_i^* = C$. This is a consequence of (4.99) and (4.98) and of the fact that, in this case, ξ_i (or ξ_i^*) is nonzero. For those of the points that equality holds, that is, $|y_i - \langle \boldsymbol{w}, \boldsymbol{\phi}(\boldsymbol{x}_i) \rangle - w_0| = \epsilon$, the respective $\xi_i\ (\xi_i^*) = 0$ and from (4.97) (or (4.96)) the respective $\lambda_i\ (\lambda_i^*)$ can be nonzero. Then from (4.99), (4.100), and (4.98) it turns out that $0 \leq \lambda_i\ (\lambda_i^*) \leq C$.

The Lagrange multipliers can be obtained by writing the problem in its equivalent dual representation form, that is,

$$\begin{aligned} \text{maximize} \quad & \sum_{i=1}^{N} y_i(\lambda_i^* - \lambda_i) - \epsilon \sum_{i=1}^{N} (\lambda_i^* + \lambda_i) - \\ & \frac{1}{2} \sum_{i,j} (\lambda_i^* - \lambda_i)(\lambda_j^* - \lambda_j) \langle \boldsymbol{\phi}(\boldsymbol{x}_i), \boldsymbol{\phi}(\boldsymbol{x}_j) \rangle \end{aligned} \tag{4.103}$$

$$\text{subject to} \quad 0 \leq \lambda_i \leq C, \quad 0 \leq \lambda_i^* \leq C, \quad i = 1, 2, \ldots, N \tag{4.104}$$

$$\sum_{i=1}^{N} \lambda_i^* = \sum_{i=1}^{N} \lambda_i \tag{4.105}$$

where maximization is with respect to the Lagrange multipliers $\lambda_i,\ \lambda_i^*,\ i = 1, 2, \ldots, N$. This optimization task is similar to the problem defined by (3.103) and (3.105).

Once the Lagrange multipliers have been computed, the nonlinear regressor is obtained as

$$g(\boldsymbol{x}) \equiv \left\langle \boldsymbol{\phi}(\boldsymbol{x}), \sum_{i=1}^{N}(\lambda_i^* - \lambda_i)\boldsymbol{\phi}(\boldsymbol{x}_i) \right\rangle + w_0 = \sum_{i=1}^{N}(\lambda_i^* - \lambda_i)K(\boldsymbol{x}, \boldsymbol{x}_i) + w_0$$

where w_0 is computed from the KKT conditions in (4.97) and (4.96) for $0 < \lambda_i < C$, $0 < \lambda_i^* < C$. Sparsification is achieved via the points associated with zero Lagrange multipliers, that is, points resulting in absolute error values *strictly* less than ϵ.

If instead of the linear ϵ-insensitive loss one adopts the quadratic ϵ-insensitive loss or the Huber loss functions, the resulting sets of formulas are similar to the ones derived here, see, for example, [Vapn 00].

Throughout the derivations in this subsection, we kept referring to the optimization of the SVM classification task considered in Section 3.7.2. The similarity is not accidental. Indeed, it is a matter of a few simple arithmetic manipulations to see that if we set $\epsilon = 0$ and $y_i = \pm 1$, depending on the class origin, our regression task becomes the same as the problem considered in Section 3.7.2.

Ridge Regression

We will close this section by establishing the connection of the regression task, which was considered before, with the classical regression problem known as *ridge regression*. This concept has been used extensively in statistical learning and has been rediscovered under different names. If in the quadratic ϵ-insensitive loss we set $\epsilon = 0$ and, for simplicity, $w_0 = 0$, the result is the standard sum of squared errors cost function. Substituting in the associated constraints the inequalities with equalities and slightly rephrasing the cost (to bring it in its classical formulation), we end up with the following

$$\text{minimize} \quad J(\boldsymbol{w}, \boldsymbol{\xi}) = \mathcal{C}\|\boldsymbol{w}\|^2 + \sum_{i=1}^{N}\xi_i^2 \tag{4.106}$$

$$\text{subject to} \quad y_i - \langle \boldsymbol{w}, \boldsymbol{\phi}(\boldsymbol{x}_i)\rangle = \xi_i, \quad i = 1, 2, \ldots, N \tag{4.107}$$

where $\mathcal{C} = \frac{1}{2C}$. The task defined in (4.106)–(4.107) is a regularized version of the least squares cost function expressed in an RKHS. If we work on the dual Wolfe representation, it turns out that the solution of the kernel ridge regression is expressed in closed form (see Problem 4.25), that is,

$$\boldsymbol{w} = \frac{1}{2\mathcal{C}}\sum_{i=1}^{N}\lambda_i\boldsymbol{\phi}(\boldsymbol{x}_i) \tag{4.108}$$

$$[\lambda_1, \ldots, \lambda_N]^T = 2\mathcal{C}\,(\mathcal{K} + \mathcal{C}I)^{-1}\boldsymbol{y} \tag{4.109}$$

and

$$g(\boldsymbol{x}) \equiv \langle \boldsymbol{w}, \boldsymbol{\phi}(\boldsymbol{x}) \rangle = \boldsymbol{y}^T (\mathcal{K} + \mathcal{C}I)^{-1} \boldsymbol{p} \tag{4.110}$$

where I is the $N \times N$ identity matrix and $\mathcal{K}$ is the $N \times N$ Gram matrix, defined in (4.84) and $\boldsymbol{p}$ the N-dimensional vector defined in (4.87). Observe that the only difference from the kernel least squares solution is the presence of the $\mathcal{C}I$ factor.

An advantage of the (kernel) ridge regression, compared to the robust statistics regression, is that a neat closed form solution results. However, by having adopted $\epsilon = 0$ we have lost in model sparseness. As we have already pointed out for the case of the linear ϵ-insensitive loss (the same is true for the quadratic version), training points that result in error with absolute value strictly less than ϵ *do not contribute* in the solution. There is no free lunch in real life!

To establish another bridge with Chapter 3, let us employ the linear kernel, that is, $K(\boldsymbol{x}_i, \boldsymbol{x}_j) = \boldsymbol{x}_i^T \boldsymbol{x}_j$ (which implies that one works in the input low-dimensional space and no mapping in a high-dimensional RKHS is performed), and solve the primal instead of the dual task ridge regression task. It is easy to show (Problem 4.26) that the solution becomes

$$\boldsymbol{w} = (X^T X + \mathcal{C}I)^{-1} X^T \boldsymbol{y} \tag{4.111}$$

In other words, the solution is the same as the least squares error solution, given in (3.45). The only difference lies in the presence of the $\mathcal{C}I$ factor. The latter is the result of the regularization term in the minimized cost function (4.106). In practice, the term $\mathcal{C}I$ is used in the LS solution in cases where $X^T X$ has a small determinant and matrix inversion problems arise. Adding a small positive value across the diagonal acts beneficially from the numerical stability point of view.

As a last touch on this section, let us comment on (4.111) and (4.108)–(4.109). For the linear kernel case, the Gram matrix becomes XX^T, and the solution resulting from the dual formulation is given by

$$\boldsymbol{w} = X^T (XX^T + \mathcal{C}I)^{-1} \boldsymbol{y} \tag{4.112}$$

Since this is a convex programming task, both solutions, in (4.111) and (4.112), must be the same. This can be verified by simple algebra (Problem 4.27).

4.20 DECISION TREES

In this section we briefly review a large class of nonlinear classifiers known as *decision trees*. These are *multistage* decision systems in which classes are sequentially rejected until we reach a finally accepted class. To this end, the feature space is split into unique regions, corresponding to the classes, *in a sequential manner*. Upon the arrival of a feature vector, the searching of the region to which the feature vector will be assigned is achieved via a sequence of decisions along a path of

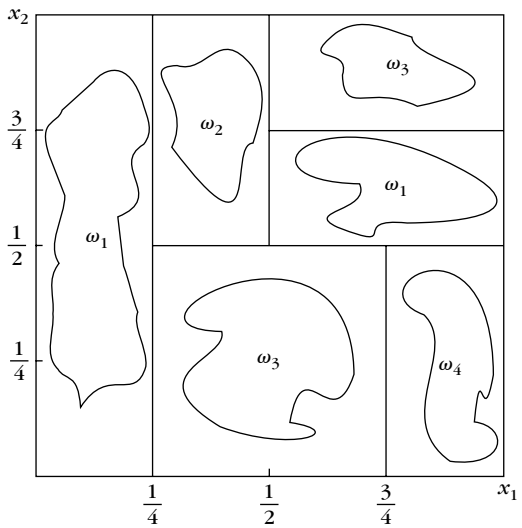

FIGURE 4.27

Decision tree partition of the space.

nodes of an appropriately constructed *tree*. Such schemes offer advantages when a large number of classes are involved. The most popular decision trees are those that split the space into hyperrectangles with sides parallel to the axes. The sequence of decisions is applied to individual features, and the questions to be answered are of the form "*is feature* $x_i \leq \alpha$?" where α is a threshold value. Such trees are known as *ordinary binary classification trees (OBCTs)*. Other types of trees are also possible that split the space into convex polyhedral cells or into pieces of spheres.

The basic idea behind an OBCT is demonstrated via the simplified example of Figure 4.27. By a successive sequential splitting of the space, we have created regions corresponding to the various classes.

Figure 4.28 shows the respective binary tree with its decision nodes and leaves. Note that it is possible to reach a decision without having tested *all* the available features.

The task illustrated in Figure 4.27 is a simple one in the two-dimensional space. The thresholds used for the *binary splits* at each node of the tree in Figure 4.28 were dictated by a simple observation of the geometry of the problem. However, this is not possible in higher dimensional spaces. Furthermore, we started the queries by testing x_1 against $\frac{1}{4}$. An obvious question is why to consider x_1 first and not another feature. In the general case, in order to develop a binary decision tree, the designer has to consider the following design elements in the training phase:

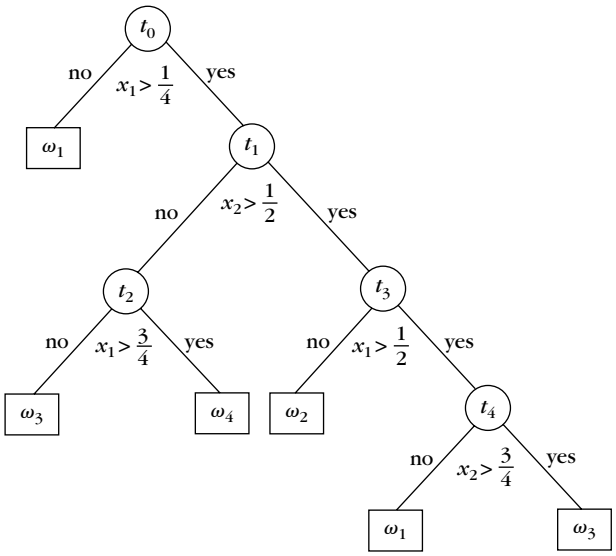

FIGURE 4.28

Decision tree classification for the case of Figure 4.27.

- At each node, the set of candidate questions to be asked has to be decided. Each question corresponds to a specific binary split into two *descendant* nodes. Each node, t, is associated with a specific subset X_t of the training set X. Splitting of a node is equivalent to the split of the subset X_t into two *disjoint* descendant subsets, X_{tY}, X_{tN}. The first of the two consists of the vectors in X_t that correspond to the answer "Yes" of the question and those of the second to the "No." The first (*root*) node of the tree is associated with the training set X. For every split, the following is true:

$$X_{tY} \bigcap X_{tN} = \emptyset$$

$$X_{tY} \bigcup X_{tN} = X_t$$

- A *splitting criterion* must be adopted according to which the best split from the set of candidate ones is chosen.
- A stop-splitting rule is required that controls the growth of the tree, and a node is declared as a terminal one (*leaf*).
- A rule is required that assigns each leaf to a specific class.

We are now experienced enough to understand that more than one method can be used to approach each of the above design elements.

4.20.1 Set of Questions

For the OBCT type of trees, the questions are of the form "*Is* $x_k \leq \alpha$?" For *each* feature, every possible value of the threshold α defines a specific split of the subset X_t. Thus in theory, an infinite set of questions has to be asked if α varies in an interval $Y_\alpha \subseteq \mathcal{R}$. In practice, only a finite set of questions can be considered. For example, since the number, N, of training points in X is finite, any of the features $x_k, k = 1, \ldots, l$, can take at most $N_t \leq N$ different values, where N_t is the cardinality of the subset $X_t \subseteq X$. Thus, for feature x_k, one can use $\alpha_{kn}, n = 1, 2, \ldots, N_{tk}$ ($N_{tk} \leq N_t$), where α_{kn} are taken halfway between consecutive distinct values of x_k in the training subset X_t. The same has to be repeated for all features. Thus in such a case, the total number of candidate questions is $\sum_{k=1}^{l} N_{tk}$. However, only one of them has to be chosen to provide the binary split at the current node, t, of the tree. This is selected to be the one that leads to the best split of the associated subset X_t. The best split is decided according to a splitting criterion.

4.20.2 Splitting Criterion

Every binary split of a node, t, generates two descendant nodes. Let us denote them by t_Y and t_N according to the "Yes" or "No" answer to the single question adopted for the node t, also referred as the *ancestor* node. As we have already mentioned, the descendant nodes are associated with two new subsets, that is, X_{tY}, X_{tN}, respectively. In order for the tree growing methodology, from the root node down to the leaves, to make sense, every split must generate subsets that are more "class homogeneous" compared to the ancestor's subset X_t. This means that the training feature vectors in each one of the new subsets show a higher preference for specific class(es), whereas data in X_t are more equally distributed among the classes. As an example, let us consider a four-class task and assume that the vectors in subset X_t are distributed among the classes with equal probability (percentage). If one splits the node so that the points that belong to ω_1, ω_2 classes form the X_{tY} subset, and the points from ω_3, ω_4 classes form the X_{tN} subset, then the new subsets are more homogeneous compared to X_t or "purer" in the decision tree terminology. The goal, therefore, is to define a measure that quantifies node impurity and split the node so that the overall impurity of the descendant nodes is optimally decreased with respect to the ancestor node's impurity.

Let $P(\omega_i|t)$ denote the probability that a vector in the subset X_t, associated with a node t, belongs to class ω_i, $i = 1, 2, \ldots, M$. A commonly used definition of *node impurity*, denoted as $I(t)$, is given by

$$I(t) = -\sum_{i=1}^{M} P(\omega_i|t) \log_2 P(\omega_i|t) \tag{4.113}$$

where $\log_2$ is the logarithm with base 2. This is nothing else than the entropy associated with the subset X_t, known from Shannon's Information Theory. It is not difficult to show that $I(t)$ takes its maximum value if all probabilities are equal to

$\frac{1}{M}$ (highest impurity) and it becomes zero (recall that $0 \log 0 = 0$) if all data belong to a single class, that is, if only one of the $P(\omega_i|t) = 1$ and all the others are zero (least impurity). In practice, probabilities are estimated by the respective percentages, N_t^i/N_t, where N_t^i is the number of points in X_t that belong to class ω_i. Assume now that performing a split, N_{tY} points are sent into the "Yes" node (X_{tY}) and N_{tN} into the "No" node (X_{tN}). The *decrease in node impurity* is defined as

$$\Delta I(t) = I(t) - \frac{N_{tY}}{N_t} I(t_Y) - \frac{N_{tN}}{N_t} I(t_N) \tag{4.114}$$

where $I(t_Y)$, $I(t_N)$ are the impurities of the t_Y and t_N nodes, respectively. *The goal now becomes to adopt, from the set of candidate questions, the one that performs the split leading to the highest decrease of impurity*.

4.20.3 Stop-Splitting Rule

The natural question that now arises is when one decides to stop splitting a node and declares it as a leaf of the tree. A possibility is to adopt a threshold T and stop splitting if the maximum value of $\Delta I(t)$, over all possible splits, is less than T. Other alternatives are to stop splitting either if the cardinality of the subset X_t is small enough or if X_t is pure, in the sense that all points in it belong to a single class.

4.20.4 Class Assignment Rule

Once a node is declared to be a leaf, then it has to be given a class label. A commonly used rule is the majority rule, that is, the leaf is labeled as ω_j where

$$j = arg \max_i P(\omega_i|t)$$

In words, we assign a leaf, t, to that class to which the majority of the vectors in X_t belong.

Having discussed the major elements needed for the growth of a decision tree, we are now ready to summarize the basic algorithmic steps for constructing a binary decision tree

- Begin with the root node, that is, $X_t = X$
- For each new node t
  - For every feature $x_k, k = 1, 2, \ldots, l$
    - For every value $\alpha_{kn}, n = 1, 2, \ldots, N_{tk}$
      - Generate X_{tY} and X_{tN} according to the answer in the question: is $x_k(i) \leq \alpha_{kn},\ i = 1, 2, \ldots, N_t$
      - Compute the impurity decrease
    - End
    - Choose α_{kn_0} leading to the maximum decrease w.r. to x_k

- End
- Choose x_{k_0} and associated $\alpha_{k_0 n_0}$ leading to the overall maximum decrease of impurity
- If the stop-splitting rule is met, declare node t as a leaf and designate it with a class label
- If not, generate two descendant nodes t_Y and t_N with associated subsets X_{tY} and X_{tN}, depending on the answer to the question: is $x_{k_0} \leq \alpha_{k_0 n_0}$

■ End

Remarks

■ A variety of node impurity measures can be defined. However, as pointed out in [Brei 84], the properties of the resulting final tree seem to be rather insensitive to the choice of the splitting criterion. Nevertheless, this is very much a problem-dependent task.

■ A critical factor in designing a decision tree is its size. As was the case with the multilayer perceptrons, the size of a tree must be large enough but not too large; otherwise it tends to learn the particular details of the training set and exhibits poor generalization performance. Experience has shown that use of a threshold value for the impurity decreases as the stop-splitting rule does not lead to trees of the right size. Many times it stops tree growing either too early or too late. The most commonly used approach is to grow a tree up to a large size first and then prune nodes according to a pruning criterion. This philosophy is similar to that for pruning multilayer perceptrons. A number of pruning criteria have been suggested in the literature. A commonly used criterion is to combine an estimate of the error probability with a complexity measuring term (e.g., number of terminal nodes). For more on this issue the interested reader may refer to [Brei 84, Ripl 94].

■ A drawback associated with tree classifiers is their high variance. In practice it is not uncommon for a small change in the training data set to result in a very different tree. The reason for this lies in the hierarchical nature of the tree classifiers. An error that occurs in a node high in the tree propagates all the way down to the leaves below it. *Bagging* (bootstrap aggregating) [Brei 96, Gran 04] is a technique that can reduce variance and improve the generalization error performance. The basic idea is to create a number of, say, B variants, $X_1, X_2, \ldots, X_B$, of the training set, X, using *bootstrap* techniques, by uniformly sampling from X with replacement (see also Section 10.3). For each of the training set variants, X_i, a tree, T_i, is constructed. The final decision is in favor of the class predicted by the majority of the subclassifiers, T_i, $i = 1, 2, \ldots, B$.

Random forests use the idea of bagging in tandem with random feature selection [Brei 01]. The difference with bagging lies in the way the decision trees are constructed. The feature to split in each node is selected as the best among a set of F *randomly* chosen features, where F is a user-defined parameter. This extra introduced randomness is reported to have a substantial effect in performance improvement.

- Our discussion so far was focused on the OBCT type of tree. More general partition of the feature space, via hyperplanes not parallel to the axis, is possible via questions of the type: *Is* $\sum_{k=1}^{l} c_k x_k \leq \alpha$*?* This can lead to a better partition of the space. However, the training now becomes more involved; see, for example, [Quin 93].
- Constructions of fuzzy decision trees have also been suggested, by allowing the possibility of partial membership of a feature vector in the nodes that make up the tree structure. Fuzzification is achieved by imposing a fuzzy structure over the basic skeleton of a standard decision tree; see, for example, [Suar 99] and the references therein.
- Decision trees have emerged as one of the most popular methods for classification. An OBCT performs binary splits on single variables, and classifying a pattern may only require a few tests. Moreover, they can naturally treat mixtures of numeric and categorical variables. Also, due to their structural simplicity, they are easily interpretable.

Example 4.2

In a tree classification task, the set X_t, associated with node t, contains $N_t = 10$ vectors. Four of these belong to class ω_1, four to class ω_2, and two to class ω_3, in a three-class classification task. The node splitting results into two new subsets X_{tY}, with three vectors from ω_1, and one from ω_2, and X_{tN} with one vector from ω_1, three from ω_2, and two from ω_3. The goal is to compute the decrease in node impurity after splitting.

We have that

$$I(t) = -\frac{4}{10}\log_2\frac{4}{10} - \frac{4}{10}\log_2\frac{4}{10} - \frac{2}{10}\log_2\frac{2}{10} = 1.521$$

$$I(t_Y) = -\frac{3}{4}\log_2\frac{3}{4} - \frac{1}{4}\log_2\frac{1}{4} = 0.815$$

$$I(t_N) = -\frac{1}{6}\log_2\frac{1}{6} - \frac{3}{6}\log_2\frac{3}{6} - \frac{2}{6}\log_2\frac{2}{6} = 1.472$$

Hence, the impurity decrease after splitting is

$$\Delta I(t) = 1.521 - \frac{4}{10}(0.815) - \frac{6}{10}(1.472) = 0.315$$

For further information and a deeper study of decision tree classifiers, the interested reader may consult the seminal book [Brei 84]. A nonexhaustive sample of later contributions in the area is [Datt 85, Chou 91, Seth 90, Graj 86, Quin 93]. A comparative guide for a number of well-known techniques is provided in [Espo 97].

Finally, it must be stated that there are close similarities between the decision trees and the neural network classifiers. Both aim at forming complex decision boundaries in the feature space. A major difference lies in the way decisions are made. Decision trees employ a hierarchically structured decision function in a sequential fashion. In contrast, neural networks utilize a set of soft (not final) decisions in a parallel fashion.

Furthermore, their training is performed via different philosophies. However, despite their differences, it has been shown that linear tree classifiers (with a linear splitting criterion) can be adequately mapped to a multilayer perceptron structure [Seth 90, Seth 91, Park 94].

So far, from the performance point of view, comparative studies seem to give an advantage to the multilayer perceptrons with respect to the classification error, and an advantage to the decision trees with respect to the required training time [Brow 93].

4.21 COMBINING CLASSIFIERS

The present chapter is the third one concerning the classifier design phase. Although we have not exhausted the list (a few more cases will be discussed in the chapters to follow), we feel that we have presented to the reader the most popular directions currently used for the design of a classifier.

Another trend that offers more possibilities to the designer is to *combine different classifiers*. Thus, one can exploit their individual advantages in order to reach an overall better performance than could be achieved by using each of them separately. An important observation that justifies such an approach is the following. From the different (candidate) classifiers we design in order to choose the one that fits our needs, one results in the best performance; that is, minimum classification error rate. However, different classifiers may fail (to classify correctly) on different patterns. That is, even the "best" classifier can fail on patterns that other classifiers succeed on.

Combining classifiers aims at exploiting this complementary information that seems to reside in the various classifiers. This is illustrated in Figure 4.29. Many interesting design issues have now come onto the scene. What is the strategy that one has to adopt for combining the individual outputs in order to reach the final conclusion? Should one combine the results following the product rule, the sum rule, the min rule, the max rule, or the median rule? Should all classifiers be fed with the same feature vectors, or must different feature vectors be selected for the different classifiers? Let us now highlight some of these issues a bit further.