[Fuku 75] Fukunaga F., Narendra P.M. "A branch and bound algorithm for computing k-nearest neighbors," *IEEE Transactions on Computers*, Vol. 24, pp. 750–753, 1975.

[Fuku 90] Fukunaga F. *Introduction to Statistical Pattern Recognition*, Academic Press, 2nd ed., 1990.

[Hast 96] Hastie T., Tibshirani R. "Discriminant adaptive nearest neighbor classification," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 18(6), pp. 607–616, 1996.

[Hatt 00] Hattori K., Takahashi M. "A new edited k-nearest neighbor rule in the pattern classification problem,"*Pattern Recognition*, Vol. 33, pp. 521–528, 2000.

[Huan 02] Huang Y.S., Chiang C.C., Shieh J.W., Grimson E. "Prototype optimization for nearest-neighbor classification," *Pattern Recognition*, Vol. 35, pp. 1237–1245, 2002.

[Jayn 82] Jaynes E.T. "On the rationale of the maximum entropy methods," *Proceedings of the IEEE*, Vol. 70(9), pp. 939–952, 1982.

[Kris 00] Krishna K., Thathachar M.A.L., Ramakrishnan K.R. "Voronoi networks and their probability of misclassification,"*IEEE Transactions on Neural Networks*, Vol. 11(6), pp. 1361–1372, 2000.

[McLa 88] McLachlan G.J., Basford K.A. *Mixture Models: Inference and Applications to Clustering*, Marcel Dekker, 1988.

[Moon 96] Moon T. "The expectation maximization algorithm," *Signal Processing Magazine*, Vol. 13(6), pp. 47–60, 1996.

[Nene 97] Nene S.A., Nayar S.K. "A simple algorithm for nearest neighbor search in high dimensions," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 19(9), pp. 989–1003, 1997.

[Papo 91] Papoulis A. *Probability Random Variables and Stochastic Processes*, 3rd ed., McGraw-Hill 1991.

[Parz 62] Parzen E. "On the estimation of a probability density function and mode," *Ann. Math. Stat.* Vol. 33, pp. 1065–1076, 1962.

[Redn 84] Redner R.A., Walker H.F. "Mixture densities, maximum likelihood and the EM algorithm," *SIAM Review*, Vol. 26(2), pp. 195–239, 1984.

[Titt 85] Titterlington D.M., Smith A.F.M., Makov U.A. *Statistical Analysis of Finite Mixture Distributions*, John Wiley, 1985.

[Yan 93] Yan H. "Prototype optimization for nearest neighbor classifiers using a two layer perceptron," *Pattern Recognition*, Vol. 26(2), pp. 317–324, 1993.

[Wand 95] Wand M., Jones M. *Kernel Smoothing*, Chapman & Hall, London, 1995.

[Wu 83] Wu C. "On the convergence properties of the EM algorithm," *Annals of Statistics*, Vol. 11(1), pp. 95–103, 1983.

CHAPTER 3

LINEAR CLASSIFIERS

3.1 INTRODUCTION

Our major concern in Chapter 2 was to design classifiers based on probability density or probability functions. In some cases, we saw that the resulting classifiers were equivalent to a set of linear discriminant functions. In this chapter we will focus on the design of linear classifiers, *irrespective of the underlying distributions describing the training data*. The major advantage of linear classifiers is their simplicity and computational attractiveness. The chapter starts with the assumption that *all* feature vectors from the available classes can be classified correctly using a linear classifier, and we will develop techniques for the computation of the corresponding linear functions. In the sequel we will focus on a more general problem, in which a linear classifier cannot classify correctly all vectors, yet we will seek ways to design an *optimal linear* classifier by adopting an appropriate optimality criterion.

3.2 LINEAR DISCRIMINANT FUNCTIONS AND DECISION HYPERPLANES

Let us once more focus on the two-class case and consider linear discriminant functions. Then the respective decision hypersurface in the l-dimensional feature space is a hyperplane, that is

$$g(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{x} + w_0 = 0 \tag{3.1}$$

where $\boldsymbol{w} = [w_1, w_2, \ldots, w_l]^T$ is known as the *weight vector* and w_0 as the *threshold*. If $\boldsymbol{x}_1, \boldsymbol{x}_2$ are two points on the decision hyperplane, then the following is valid

$$\begin{aligned} 0 = \boldsymbol{w}^T\boldsymbol{x}_1 + w_0 = \boldsymbol{w}^T\boldsymbol{x}_2 + w_0 \quad \Rightarrow \\ \boldsymbol{w}^T(\boldsymbol{x}_1 - \boldsymbol{x}_2) = 0 \end{aligned} \tag{3.2}$$

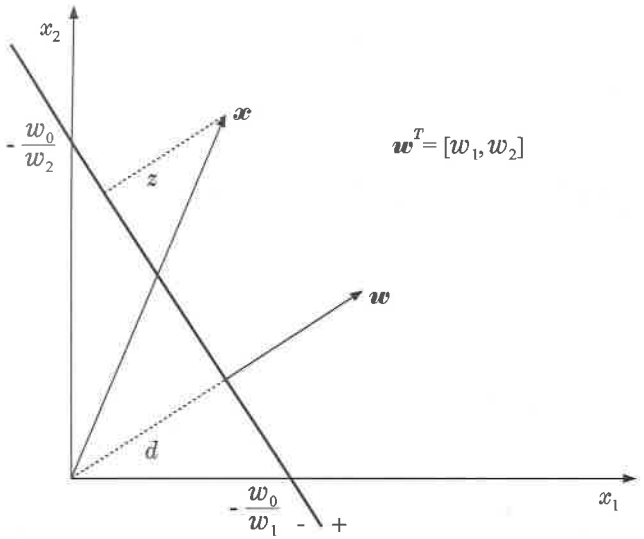


FIGURE 3.1: Geometry for the decision line. On one side of the line it is $g(\boldsymbol{x}) > 0(+)$ and on the other $g(\boldsymbol{x}) < 0$ $(-)$.

Since the difference vector $\boldsymbol{x}_1 - \boldsymbol{x}_2$ obviously lies on the decision hyperplane (for any $\boldsymbol{x}_1, \boldsymbol{x}_2$), it is apparent from Eq. (3.2) that the vector $\boldsymbol{w}$ is *orthogonal* to the decision hyperplane.

Figure 3.1 shows the corresponding geometry (for $w_1 > 0, w_2 > 0, w_0 < 0$). Recalling our high school math, it is easy to see that the quantities entering in the figure are given by

$$d = \frac{|w_0|}{\sqrt{w_1^2 + w_2^2}} \tag{3.3}$$

and

$$z = \frac{|g(\boldsymbol{x})|}{\sqrt{w_1^2 + w_2^2}} \tag{3.4}$$

In other words, $|g(\boldsymbol{x})|$ is a measure of the Euclidean distance of the point $\boldsymbol{x}$ from the decision hyperplane. On one side of the plane $g(\boldsymbol{x})$ takes positive values and on the other negative. In the special case that $w_0 = 0$ the hyperplane passes through the origin.

3.3 THE PERCEPTRON ALGORITHM

Our major concern now is to compute the unknown parameters $w_i, i = 0, \ldots, l$, defining the decision hyperplane. In this section we assume that the two classes ω_1, ω_2 are *linearly separable*. In other words we assume that *there exists* a hyperplane, defined by $\boldsymbol{w}^{*T}\boldsymbol{x} = 0$, such that

$$\begin{aligned} \boldsymbol{w}^{*T}\boldsymbol{x} &> 0 \quad \forall \boldsymbol{x} \in \omega_1 \\ \boldsymbol{w}^{*T}\boldsymbol{x} &< 0 \quad \forall \boldsymbol{x} \in \omega_2 \end{aligned} \tag{3.5}$$

The formulation above also covers the case of a hyperplane not crossing the origin, that is, $\boldsymbol{w}^{*T}\boldsymbol{x} + w_0^* = 0$, since this can be brought into the previous formulation by defining the extended $(l+1)$-dimensional vectors $\boldsymbol{x}' \equiv [\boldsymbol{x}^T, 1]^T$, $\boldsymbol{w}' \equiv [\boldsymbol{w}^{*T}, w_0^*]^T$. Then $\boldsymbol{w}^{*T}\boldsymbol{x} + w_0^* = \boldsymbol{w}'^T\boldsymbol{x}'$.

We will approach the problem as a typical optimization task (Appendix C). Thus we need to adopt (a) an appropriate cost function and (b) an algorithmic scheme to optimize it. To this end, we choose the *perceptron cost* defined as

$$J(\boldsymbol{w}) = \sum_{\boldsymbol{x} \in Y} (\delta_x \boldsymbol{w}^T \boldsymbol{x}) \tag{3.6}$$

where Y is the subset of the training vectors, which are misclassified by the hyperplane defined by the weight vector $\boldsymbol{w}$. The variable δ_x is chosen so that $\delta_x = -1$ if $\boldsymbol{x} \in \omega_1$ and $\delta_x = +1$ if $\boldsymbol{x} \in \omega_2$. Obviously, the sum in (3.6) is always positive and it becomes zero when Y becomes the empty set, that is, if there are not misclassified vectors $\boldsymbol{x}$. Indeed, if $\boldsymbol{x} \in \omega_1$ and it is misclassified, then $\boldsymbol{w}^T\boldsymbol{x} < 0$ and $\delta_x < 0$, and the product is positive. The result is the same for vectors originating from class ω_2. When the cost function takes its minimum value, 0, a solution has been obtained, since all training feature vectors are correctly classified.

The perceptron cost function in (3.6) is *continuous and piecewise linear*. Indeed, if we change the weight vector smoothly, the cost $J(\boldsymbol{w})$ changes linearly until the point at which there is a change in the number of misclassified vectors (Problem 3.1). At these points the gradient is not defined and the gradient function is discontinuous.

To derive the algorithm for the iterative minimization of the cost function, we will adopt an iterative scheme in the spirit of the *gradient descent* method (Appendix C), that is,

$$\boldsymbol{w}(t+1) = \boldsymbol{w}(t) - \rho_t \frac{\partial J(\boldsymbol{w})}{\partial \boldsymbol{w}}\bigg|_{\boldsymbol{w}=\boldsymbol{w}(t)} \tag{3.7}$$

where $\boldsymbol{w}(t)$ is the weight vector estimate at the tth iteration step, and ρ_t is a sequence of positive real numbers. However, we must be careful here. This is not

defined at the points of discontinuity. From the definition in (3.6), and at the points where this is valid, we get

$$\frac{\partial J(\boldsymbol{w})}{\partial \boldsymbol{w}} = \sum_{\boldsymbol{x}\in Y} \delta_{\boldsymbol{x}} \boldsymbol{x} \tag{3.8}$$

Substituting (3.8) into (3.7) we obtain

$$\boldsymbol{w}(t+1) = \boldsymbol{w}(t) - \rho_t \sum_{\boldsymbol{x}\in Y} \delta_{\boldsymbol{x}} \boldsymbol{x} \tag{3.9}$$

The algorithm is known as the *perceptron algorithm* and is quite simple in its structure. *Note that Eq. (3.9) is defined at all points.* The algorithm is initialized from an arbitrary weight vector $\boldsymbol{w}(0)$, and the correction vector $\sum_{\boldsymbol{x}\in Y} \delta_{\boldsymbol{x}} \boldsymbol{x}$ is formed using the misclassified features. The weight vector is then corrected according to the preceding rule. This is repeated until the algorithm converges to a solution, that is, all features are correctly classified.

Figure 3.2 provides a geometric interpretation of the algorithm. It has been assumed that at step t there is only one misclassified sample, $\boldsymbol{x}$, and $\rho_t = 1$. The perceptron algorithm corrects the weight vector in the direction of $\boldsymbol{x}$. Its effect is to turn the corresponding hyperplane so that $\boldsymbol{x}$ is classified in the correct class ω_1.

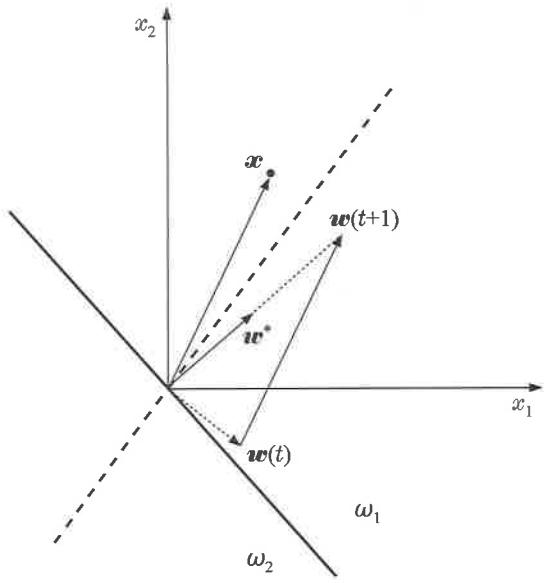


FIGURE 3.2: Geometric interpretation of the perceptron algorithm.

Note that in order to achieve this, it may take more than one iteration step, depending on the value(s) of ρ_t. No doubt, this sequence is critical for the convergence. We will now show that the perceptron algorithm converges to a solution in a *finite number of iteration steps*, provided that the sequence ρ_t is properly chosen. The solution is not unique, because there are more than one hyperplanes separating two linearly separable classes. The convergence proof is necessary because the algorithm is not a true gradient descent algorithm and the general tools for the convergence of gradient descent schemes cannot be applied.

Proof of the Perceptron Algorithm Convergence

Let α be a positive real number and $\boldsymbol{w}^*$ a solution. Then from (3.9) we have

$$\boldsymbol{w}(t+1) - \alpha \boldsymbol{w}^* = \boldsymbol{w}(t) - \alpha \boldsymbol{w}^* - \rho_t \sum_{\boldsymbol{x}\in Y} \delta_{\boldsymbol{x}} \boldsymbol{x} \tag{3.10}$$

Squaring the Euclidean norm of both sides results in

$$\begin{aligned} \|\boldsymbol{w}(t+1) - \alpha \boldsymbol{w}^*\|^2 = \|\boldsymbol{w}(t) - \alpha \boldsymbol{w}^*\|^2 + \rho_t^2 \|\sum_{\boldsymbol{x}\in Y} \delta_{\boldsymbol{x}} \boldsymbol{x}\|^2 \\ - 2\rho_t \sum_{\boldsymbol{x}\in Y} \delta_{\boldsymbol{x}} (\boldsymbol{w}(t) - \alpha \boldsymbol{w}^*)^T \boldsymbol{x} \end{aligned} \tag{3.11}$$

But $-\sum_{\boldsymbol{x}\in Y} \delta_{\boldsymbol{x}} \boldsymbol{w}^T(t)\, \boldsymbol{x} < 0$. Hence

$$\begin{aligned} \|\boldsymbol{w}(t+1) - \alpha \boldsymbol{w}^*\|^2 \leq \|\boldsymbol{w}(t) - \alpha \boldsymbol{w}^*\|^2 + \rho_t^2 \|\sum_{\boldsymbol{x}\in Y} \delta_{\boldsymbol{x}} \boldsymbol{x}\|^2 \\ + 2\rho_t \alpha \sum_{\boldsymbol{x}\in Y} \delta_{\boldsymbol{x}} \boldsymbol{w}^{*T} \boldsymbol{x} \end{aligned} \tag{3.12}$$

Define

$$\beta^2 = \max_{\bar{Y} \subseteq \omega_1 \cup \omega_2} \|\sum_{\boldsymbol{x}\in \bar{Y}} \delta_{\boldsymbol{x}} \boldsymbol{x}\|^2 \tag{3.13}$$

That is, β^2 is the maximum value that the involved vector norm can take by considering *all* possible (nonempty) subsets of the available training feature vectors. Similarly, let

$$\gamma = \max_{\bar{Y} \subseteq \omega_1 \cup \omega_2} \sum_{\boldsymbol{x}\in \bar{Y}} \delta_{\boldsymbol{x}} \boldsymbol{w}^{*T} \boldsymbol{x} \tag{3.14}$$

Recall that the summation in this equation is negative; thus, its maximum value over all possible subsets of $\boldsymbol{x}$'s will also be a negative number. Hence, (3.12) can

now be written as

$$\|\boldsymbol{w}(t+1)-\alpha\boldsymbol{w}^*\|^2 \leq \|\boldsymbol{w}(t)-\alpha\boldsymbol{w}^*\|^2+\rho_t^2\beta^2-2\rho_t\alpha|\gamma| \tag{3.15}$$

Choose $\alpha=\frac{\beta^2}{2|\gamma|}$ and apply (3.15) successively for steps $t, t-1, \ldots, 0$. Then

$$\|\boldsymbol{w}(t+1)-\alpha\boldsymbol{w}^*\|^2 \leq \|\boldsymbol{w}(0)-\alpha\boldsymbol{w}^*\|^2+\beta^2\left(\sum_{k=0}^{t}\rho_k^2-\sum_{k=0}^{t}\rho_k\right) \tag{3.16}$$

If the sequence ρ_t is chosen to satisfy the following two conditions:

$$\lim_{t\to\infty}\sum_{k=0}^{t}\rho_k=\infty \tag{3.17}$$

$$\lim_{t\to\infty}\sum_{k=0}^{t}\rho_k^2<\infty \tag{3.18}$$

then there will be a constant t_0 such that the right-hand side of (3.16) becomes nonpositive. Thus

$$0\leq\|\boldsymbol{w}(t_0+1)-\alpha\boldsymbol{w}^*\|\leq 0 \tag{3.19}$$

or

$$\boldsymbol{w}(t_0+1)=\alpha\boldsymbol{w}^* \tag{3.20}$$

That is, the algorithm converges to a solution in a finite number of steps. An example of a sequence satisfying conditions (3.17), (3.18) is $\rho_t=c/t$, where c is a constant. In other words, the corrections become increasingly small. What these conditions basically state is that ρ_t should vanish as $t\to\infty$ [Eq. (3.18)] but on the other hand should not go to zero very fast [Eq. (3.17)]. Following arguments similar to those used before, it is easy to show that the algorithm also converges for constant $\rho_t=\rho$, provided ρ is properly bounded (Problem 3.2). In practice, the proper choice of the sequence ρ_t is vital for the convergence speed of the algorithm.

Variants of the Perceptron Algorithm

The algorithm we have presented is just one form of a number of variants that have been proposed for the training of a linear classifier in the case of linearly separable classes. We will now state another simpler and also popular form. The N training vectors enter the algorithm cyclically, one after the other. If the algorithm has not converged after the presentation of all the samples once, then the procedure

keeps repeating until convergence is achieved, that is, when all training samples have been classified correctly. Let $\boldsymbol{w}(t)$ be the weight vector estimate and $\boldsymbol{x}_{(t)}$ the corresponding feature vector, presented at the tth iteration step. The algorithm is stated as follows:

$$\begin{aligned}
&\boldsymbol{w}(t+1)=\boldsymbol{w}(t)+\rho\boldsymbol{x}_{(t)} \quad \text{if } \boldsymbol{x}_{(t)}\in\omega_1 \text{ and } \boldsymbol{w}^T(t)\boldsymbol{x}_{(t)}\leq 0\\
&\boldsymbol{w}(t+1)=\boldsymbol{w}(t)-\rho\boldsymbol{x}_{(t)} \quad \text{if } \boldsymbol{x}_{(t)}\in\omega_2 \text{ and } \boldsymbol{w}^T(t)\boldsymbol{x}_{(t)}\geq 0\\
&\boldsymbol{w}(t+1)=\boldsymbol{w}(t) \quad \text{otherwise}
\end{aligned} \tag{3.21}$$

In other words, if the current training sample is classified correctly, no action is taken. Otherwise, if the sample is misclassified, the weight vector is corrected by adding (subtracting) an amount proportional to $\boldsymbol{x}_{(t)}$. The algorithm belongs to a more general algorithmic family known as *reward and punishment* schemes. If the classification is correct, the reward is that no action is taken. If the current vector is misclassified, the punishment is the cost of correction. It can be shown that this form of the perceptron algorithm also converges in a finite number of iteration steps (Problem 3.3).

The perceptron algorithm was originally proposed by Rosenblatt in the late 1950s. The algorithm was developed for training the *perceptron*, the basic unit used for modeling neurons of the brain. This was considered central in developing powerful models for machine learning [Rose 58, Min 88].

The Perceptron

Once the perceptron algorithm has converged to a weight vector $\boldsymbol{w}$ and a threshold w_0, our next goal is the classification of an unknown feature vector to either of the two classes. Classification is achieved via the simple rule

$$\begin{aligned}
&\text{If} \quad \boldsymbol{w}^T\boldsymbol{x}+w_0>0 \quad \text{assign } \boldsymbol{x} \text{ to } \omega_1\\
&\text{If} \quad \boldsymbol{w}^T\boldsymbol{x}+w_0<0 \quad \text{assign } \boldsymbol{x} \text{ to } \omega_2
\end{aligned} \tag{3.22}$$

A basic network unit that implements the operation is shown in Figure 3.3(a).

The elements of the feature vector $x_1, x_2, \ldots, x_l$ are applied to the *input nodes* of the network. Then each one is multiplied by the corresponding weights $w_i, i=1,2,\ldots,l$. These are known as *synaptic weights* or simply *synapses*. The products are summed up together with the *threshold* value w_0. The result then goes through a nonlinear device, which implements the so-called *activation function*. A common choice is a hard limiter; that is, $f(\cdot)$ is the step function [$f(x)=-1$ if $x<0$ and $f(x)=1$ if $x>0$]. The corresponding feature vector is classified in one of the classes depending on the sign of the output. Besides $+1$ and -1, other values (class labels) for the hard limiter are also possible. Another popular choice is 1 and 0 and it is achieved by choosing the two levels of the step function appropriately.

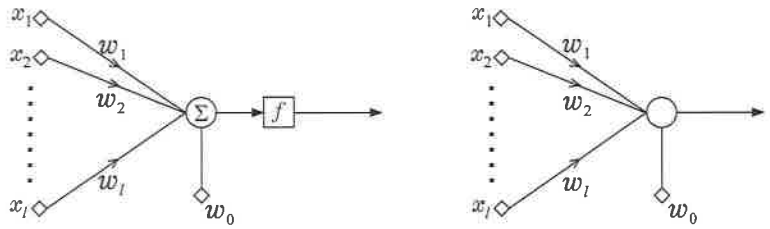


FIGURE 3.3: The basic perceptron model.

This basic network is known as a *perceptron* or *neuron*. Perceptrons are simple examples of the so-called *learning machines*, that is, structures whose free parameters are updated by a *learning algorithm*, such as the perceptron algorithm, in order to "learn" a specific task, based on a set of training data. Later on we will use the perceptron as the basic building element for more complex learning networks. Figure 3.3b is a simplified graph of the neuron where the summer and nonlinear device have been merged for notational simplification. Sometimes a neuron with a hard limiter device is referred to as a McCulloch–Pitts neuron. Other types of neurons will be considered in Chapter 4.

Example 3.1. Figure 3.4 shows the dashed line

$$x_1 + x_2 - 0.5 = 0$$

corresponding to the weight vector $[1, 1, -0.5]^T$, which has been computed from the latest iteration step of the perceptron algorithm (3.9), with $\rho_t = \rho = 0.7$. The line classifies correctly all the vectors except $[0.4, 0.05]^T$ and $[-0.20, 0.75]^T$. According to the algorithm, the next weight vector will be

$$\boldsymbol{w}(t+1) = \begin{bmatrix} 1 \\ 1 \\ -0.5 \end{bmatrix} - 0.7(-1) \begin{bmatrix} 0.4 \\ 0.05 \\ 1 \end{bmatrix} - 0.7(+1) \begin{bmatrix} -0.2 \\ 0.75 \\ 1 \end{bmatrix}$$

or

$$\boldsymbol{w}(t+1) = \begin{bmatrix} 1.42 \\ 0.51 \\ -0.5 \end{bmatrix}$$

The resulting new (solid) line $1.42x_1 + 0.51x_2 - 0.5 = 0$ classifies all vectors correctly and the algorithm is terminated.

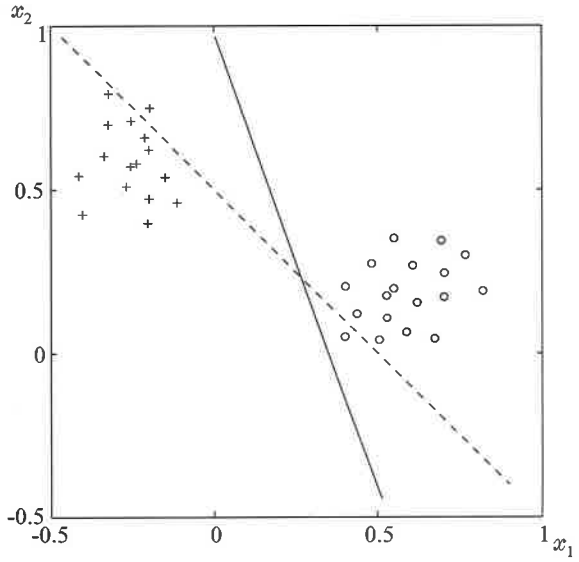


FIGURE 3.4: An example of the perceptron algorithm.

The Pocket Algorithm

A basic requirement for the convergence of the perceptron algorithm is the linear separability of the classes. If this is not true, as is usually the case in practice, the perceptron algorithm does not converge. A variant of the perceptron algorithm was suggested in [Gal 90] that converges to an optimal solution even if the linear separability condition is not fulfilled. The algorithm is known as the *pocket* algorithm and consists of the following two steps

- Initialize the weight vector $\boldsymbol{w}(0)$ randomly. Define a stored (in the pocket!) vector $\boldsymbol{w}_s$. Set a history counter h_s of the $\boldsymbol{w}_s$ to zero.
- At the tth iteration step compute the update $\boldsymbol{w}(t+1)$, according to the perceptron rule. Use the updated weight vector to test the number h of training vectors that are classified correctly. If $h > h_s$ replace $\boldsymbol{w}_s$ with $\boldsymbol{w}(t+1)$ and h_s with h. Continue the iterations.

It can be shown that this algorithm converges with probability one to the optimal solution, that is, the one that produces the minimum number of misclassifications [Gal 90, Muse 97]. Other related algorithms that find reasonably good solutions when the classes are not linearly separable are the *thermal perceptron algorithm*

[Frea 92], the *loss minimization algorithm* [Hryc 92] and the barycentric correction procedure [Poul 95].

Kesler's Construction

So far we have dealt with the two-class case. The generalization to an M-class task is straightforward. A linear discriminant function $\boldsymbol{w}_i, i = 1, 2, \ldots, M$, is defined for each of the classes. A feature vector $\boldsymbol{x}$ (in the $(l+1)$-dimensional space to account for the threshold) is classified in class ω_i if

$$\boldsymbol{w}_i^T \boldsymbol{x} > \boldsymbol{w}_j^T \boldsymbol{x}, \quad \forall j \neq i \tag{3.23}$$

This condition leads to the so-called Kesler's construction. For each of the training vectors from class $\omega_i, i=1, 2, \ldots, M$, we construct $M-1$ vectors $\boldsymbol{x}_{ij} = [\mathbf{0}^T, \mathbf{0}^T, \ldots, \boldsymbol{x}^T, \ldots, -\boldsymbol{x}^T, \ldots, \mathbf{0}^T]^T$ of dimension $(l+1)M \times 1$. That is, they are block vectors having zeros everywhere except at the ith and jth block positions, where they have $\boldsymbol{x}$ and $-\boldsymbol{x}$ respectively, for $j \neq i$. We also construct the block vector $\boldsymbol{w} = [\boldsymbol{w}_1^T, \ldots, \boldsymbol{w}_M^T]^T$. If $\boldsymbol{x} \in \omega_i$, this imposes the requirement that $\boldsymbol{w}^T \boldsymbol{x}_{ij} > 0, \forall j = 1, 2, \ldots, M, j \neq i$. The task now is to design a linear classifier, in the extended $(l+1)M$-dimensional space, so that each of the $(M-1)N$ training vectors lies in its positive side. The perceptron algorithm will have no difficulty in solving this problem for us, provided that such a solution is possible, that is, if all the training vectors can be correctly classified using a set of linear discriminant functions.

Example 3.2. Let us consider a three class problem in the two dimensional space. The training vectors for each of the classes are the following

$$\omega_1: [1, 1]^T, [2, 2]^T, [2, 1]^T$$

$$\omega_2: [1, -1]^T, [1, -2]^T, [2, -2]^T$$

$$\omega_3: [-1, 1]^T, [-1, 2]^T, [-2, 1]^T$$

This is obviously a linearly separable problem, since the vectors of different classes lie in different quadrants.

To compute the linear discriminant functions we first extend the vectors to the 3-dimensional space and then we use Kesler's construction. For example

For $[1, 1]^T$ we get $[1, 1, 1, -1, -1, -1, 0, 0, 0]^T$ and $[1, 1, 1, 0, 0, 0, -1, -1, -1]^T$

For $[1, -2]^T$ we get $[-1, 2, -1, 1, -2, 1, 0, 0, 0]^T$ and $[0, 0, 0, 1, -2, 1, -1, 2, -1]^T$

For $[-2, 1]^T$ we get $[2, -1, -1, 0, 0, 0, -2, 1, 1]^T$ and $[0, 0, 0, 2, -1, -1, -2, 1, 1]^T$

Similarly we obtain the other twelve vectors. To obtain the corresponding weight vectors

$$\boldsymbol{w}_1 = [w_{11}, w_{12}, w_{10}]^T$$

$$\boldsymbol{w}_2 = [w_{21}, w_{22}, w_{20}]^T$$

$$\boldsymbol{w}_3 = [w_{31}, w_{32}, w_{30}]^T$$

we can run the perceptron algorithm by requiring $\boldsymbol{w}^T \boldsymbol{x} > 0$, $\boldsymbol{w} = [\boldsymbol{w}_1^T, \boldsymbol{w}_2^T, \boldsymbol{w}_3^T]^T$, for each of the eighteen 9-dimensional vectors. That is, we require all the vectors to lie on the same side of the decision hyperplane. The initial vector of the algorithm $\boldsymbol{w}(0)$ is computed using the uniform pseudorandom sequence generator in [0, 1]. The learning sequence ρ_t was chosen to be constant and equal to 0.5. The algorithm converges after four iterations and gives

$$\boldsymbol{w}_1 = [5.13, 3.60, 1.00]^T$$

$$\boldsymbol{w}_2 = [-0.05, -3.16, -0.41]^T$$

$$\boldsymbol{w}_3 = [-3.84, 1.28, 0.69]^T$$

3.4 LEAST SQUARES METHODS

As we have already pointed out, the attractiveness of linear classifiers lies in their simplicity. Thus, in many cases, although we know that the classes are not linearly separable, we still wish to adopt a linear classifier, despite the fact that this will lead to *suboptimal* performance from the classification error probability point of view. The goal now is to compute the corresponding weight vector under a suitable optimality criterion. The least squares methods are familiar to us, in one way or another, from our early college courses. Let us then build upon them.

3.4.1 Mean Square Error Estimation

Let us once more focus on the two-class problem. In the previous section we saw that the perceptron output was ± 1, depending on the class ownership of $\boldsymbol{x}$. Since the classes were linearly separable, these outputs were correct for all the training feature vectors, after, of course, the perceptron algorithm's convergence. In this section we will attempt to design a linear classifier so that its *desired* output is again ± 1, depending on the class ownership of the input vector. However, we will have to live with errors; that is, the true output will not always be equal to the desired one. Given a vector $\boldsymbol{x}$, the output of the classifier will be $\boldsymbol{w}^T \boldsymbol{x}$ (thresholds can be accommodated by vector extensions). The desired output will be denoted as $y(\boldsymbol{x}) \equiv y = \pm 1$. The weight vector will be computed so as to minimize the

CHAPTER 4

NONLINEAR CLASSIFIERS

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 non–linearly 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 unescapable 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

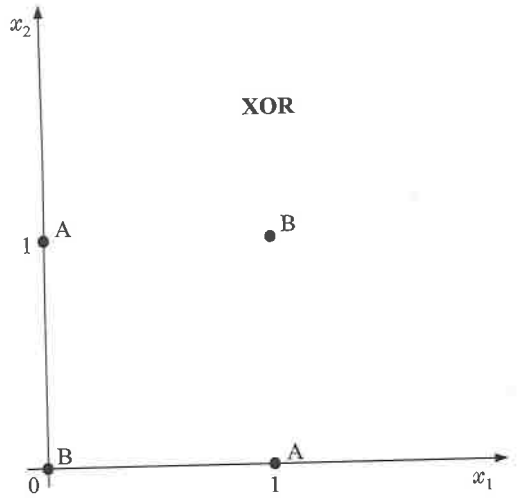


FIGURE 4.1: Classes A and B for the XOR problem.

4.3 THE TWO-LAYER PERCEPTRON

To separate the two classes A and B in Figure 4.1, a first thought that comes into 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.

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

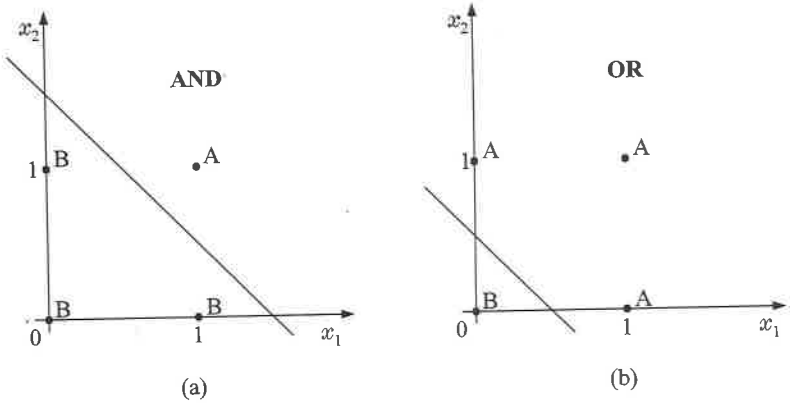


FIGURE 4.2: Classes A and B for the AND and OR problems.

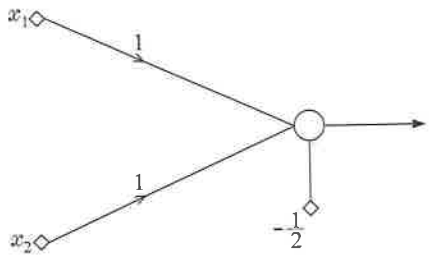


FIGURE 4.3: A perceptron realizing an OR gate.

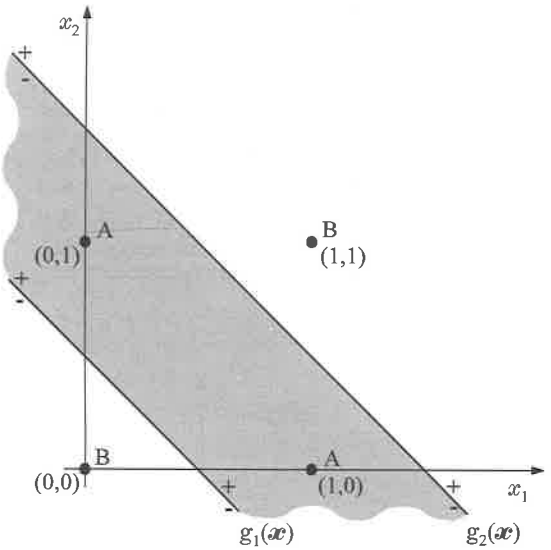


FIGURE 4.4: Decision lines realized by a two-layer perceptron for the XOR problem.

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

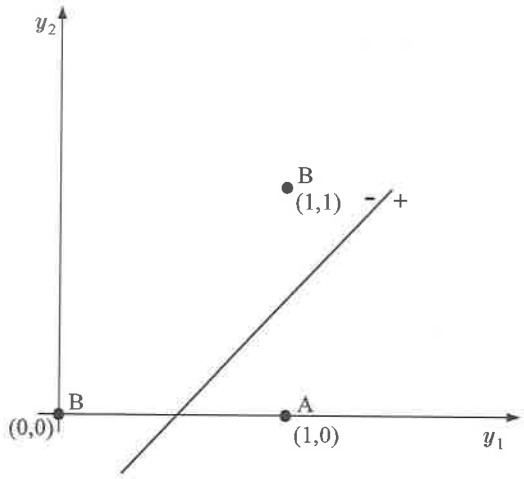


FIGURE 4.5: Decision line formed by the neuron of the second layer for the XOR problem.

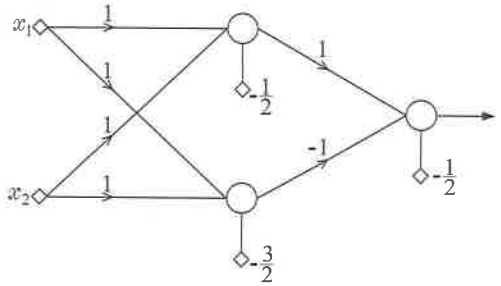


FIGURE 4.6: A two-layer perceptron solving the XOR problem.

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* 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 (non-processing) 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 $y_i \in \{0, 1\}$, $1 \leq i \leq p$.

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

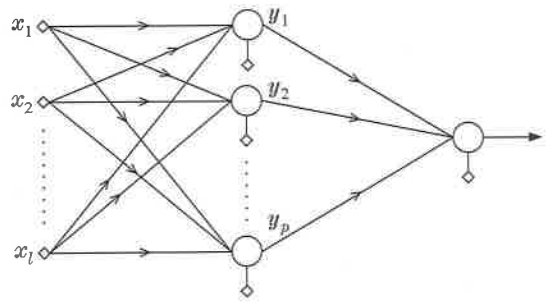


FIGURE 4.7: A two-layer perceptron.

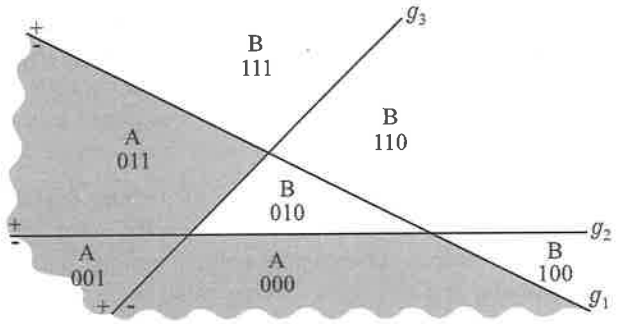


FIGURE 4.8: Polyhedra formed by the neurons of the first hidden layer of a multilayer perceptron.

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. 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]

[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.

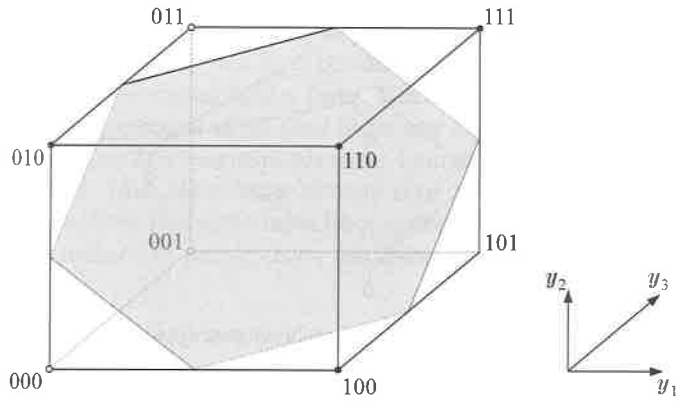


FIGURE 4.9: The neurons of the first hidden layer map an input vector onto one of the vertices of a unit (hyper)cube. The output neuron realizes a (hyper)plane to separate vertices according to their class label.

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 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 whether these are linearly separable or not. 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 only a single hyperplane. This is the same situation the basic perceptron was confronted with 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 J of the resulting polyhedra and class B of the rest. We then use J 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 J A class vertices. In other words, each time an input vector from class A enters the network, one of the J neurons of the second layer results in a 1 and the remaining $J - 1$ 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 J vertices are located in a way that makes them separable from the rest, using

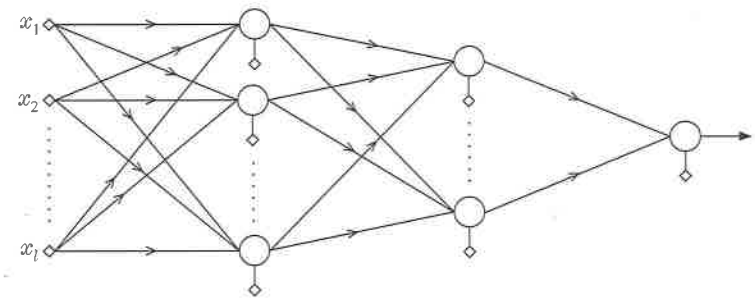


FIGURE 4.10: Architecture of a multilayer perceptron with two hidden layers of neurons and a single output neuron.

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?).

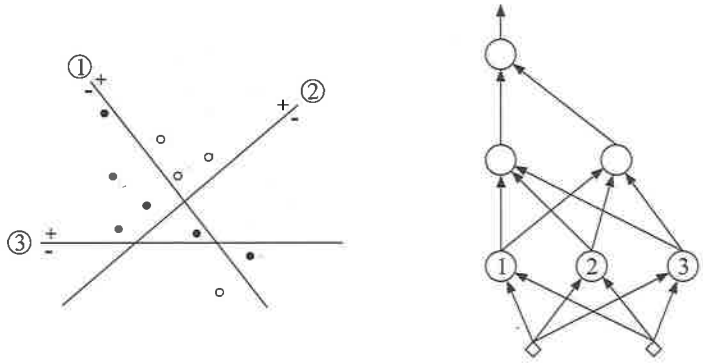
In summary, we can say that *the neurons of the first layer form the hyperplanes, those of the second layer form the regions, and finally the neurons of the output layer form the classes.*

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 are points within these regions. 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. There are two major directions in which we will focus on our attention. 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.



(a) (b)

FIGURE 4.11: Decision lines and the corresponding architecture resulting from the tiling algorithm. The open (filled) circles correspond to class A (B).

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 *ancillary 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, additional 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 joining 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

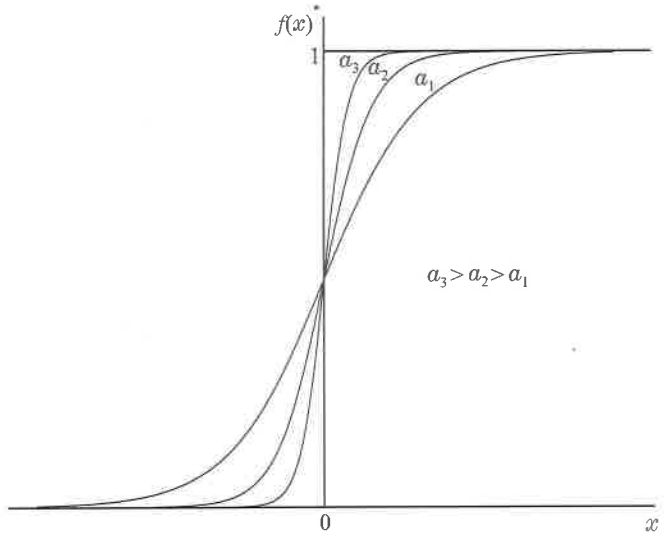


FIGURE 4.12: The logistic function.

defined as

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

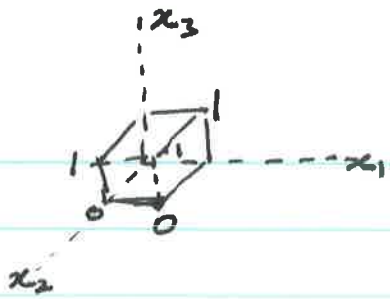
It varies between 1 and -1, and it belongs to the family of hyperbolic tangent functions,

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

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 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

x_1	x_2	x_3	class
1	1	0	0
0	1	1	1
0	1	0	0
1	0	1	1
0	0	0	1



$$w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 = 0$$

Append $x_4 = 1$ (= Augmented Data Set).

x_1	x_2	x_3	x_4	class.
1	1	0	1	0
0	1	1	1	1
0	1	0	1	0
1	0	1	1	1
0	0	0	1	1

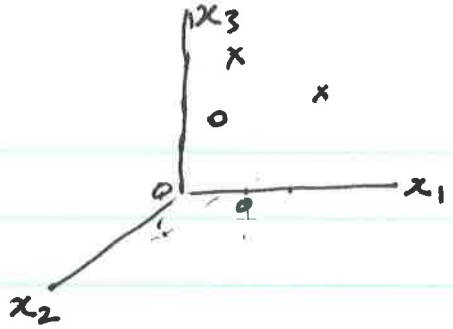
Randomly selected initial weights:

w_1	w_2	w_3	w_4
0	0	0	0

epoch	$\vec{x}$ x_1	x_2	x_3	x_4	$\vec{w}_t$ w_1	w_2	w_3	w_4	$\vec{x}.\vec{w}$	error?	$\vec{w}_t \pm \vec{x}$ w_1	w_2	w_3	w_4
	1	1	0	1	0	0	0	0	0	Y	-1	-1	0	-1
	0	1	1	1	-1	-1	0	-1	-2	Y	-1	0	1	0
	0	1	0	1	-1	0	1	0	0	Y	-1	-1	1	-1
	1	0	1	1	-1	-1	1	-1	-1	Y	0	-1	2	0
	0	0	0	1	0	-1	2	0	0	Y	0	-1	2	1 ✓
	1	1	0	1	0	-1	2	1	0	Y	-1	-2	2	0
	0	1	1	1	-1	-2	2	0	0	Y	-1	-1	3	1
	0	1	0	1	-1	-1	3	1	0	Y	-1	-2	3	0
	1	0	1	1	-1	-2	3	0	2	N	-1	-2	3	0]
	0	0	0	1	-1	-2	3	0	0	Y	-1	-2	3	1
	1	1	0	1	-1	-2	3	1	-2	N	-1	-2	3	1]
	0	1	1	1	-1	-2	3	1	+2	N	-1	-2	3	1]
	0	1	0	1	-1	-2	3	1	-1	N	-1	-2	3	1]
	1	0	1	1	-1	-2	3	1	3	N	-1	-2	3	1]
	0	0	0	1	-1	-2	3	1	1	N	-1	-2	3	1

all ok.

$-1x_1 - 2x_2 + 3x_3 + 1 = 0$



x_1	x_2	x_3	class
3	4	6	0
5	1	9	1
2	5	3	0
3	2	8	1
5	3	1	0

Augmented Data set:

x_1	x_2	x_3	x_4	class
3	4	6	1	0
5	1	9	1	1
2	5	3	1	0
3	2	8	1	1
5	3	1	1	0

Initial weight vector: 0 0 0 0

$\vec{x}$				$\vec{w}$						$\vec{w}_{t+1}$			
x_1	x_2	x_3	x_4	w_1	w_2	w_3	w_4	$\vec{x}\cdot\vec{w}$	Groups	w_1	w_2	w_3	w_4
3	4	6	1	0	0	0	0	0	Y	-3	-4	-6	-1
5	1	9	1	-3	-4	-6	-1	-74	Y	2	-3	3	0
2	5	3	1	2	-3	3	0	-2	N	2	-3	3	0]
3	2	8	1	2	-3	3	0	24	N	2	-3	3	0]
5	3	1	1	2	-3	3	0	5	Y	-3	-6	2	-1
3	4	6	1	-3	-6	2	-1	-23	N	-3	-6	2	-1]
5	1	9	1	-3	-6	2	-1	-4	Y	+2	-5	11	0
2	5	3	1	2	-5	11	0	12	Y	0	-10	8	-1
3	2	8	1	0	-10	8	-1	43	N	0	-10	8	-1]
5	3	1	1	0	-10	8	-1	-23	N	0	-10	8	-1]
3	4	6	1	0	-10	8	-1	7	Y	-3	-14	2	-2
5	1	9	1	-3	-14	2	-2	-13	Y	2	-13	11	-1
2	5	3	1	2	-13	11	-1	-29	N	2	-13	11	-1]
3	2	8	1	2	-13	11	-1	67	N	2	-13	11	-1]
5	3	1	1	2	-13	11	-1	-19	N	2	-13	11	-1]
3	4	6	1	2	-13	11	-1	19	Y	-1	-17	5	-2
5	1	9	1	-1	-17	5	-2	21	N	-1	-17	5	-2]
2	5	3	1	-1	-17	5	-2	-70	N	-1	-17	5	-2]
3	2	8	1	-1	-17	5	-2	1	N	-1	-17	5	-2]
5	3	1	1	-1	-17	5	-2	-47	N	-1	-17	5	-2]
3	4	6	1	-1	-17	5	-2	-43	N	-1	-17	5	-2]