Pattern Recognition and Neural Networks

Sergios Theodoridis[1] and Konstantinos Koutroumbas[2]

[1] Department of Informatics, Division of Communications and Signal Processing, University of Athens, Panepistimiopolis, T.Y.P.A. Buildings, 157 71 Athens, Greece
stheodor@di.uoa.gr
[2] National Observatory of Athens, Metaxa & V. Pavlou, 152 36 Palaia Penteli, Athens, Greece
koutroumbas@space.noa.gr

1 Introduction

Pattern Recognition (PR) is a fast growing field with applications in many diverse areas such as optical character recognition (OCR), computer - aided diagnosis and speech recognition, to name but a few.

Pattern recognition deals with the automatic classification of a given object into one from a number of different *categories (classes)*. The goal of PR is the development of an *automatic classification system*, which will be used for the classification of objects relevant to the application at hand.

In order to illustrate how an automatic classification system works, let us consider a computer-aided diagnosis classification system. The purpose of such a system is to classify regions from an image (e.g. X-rays) in either of two classes, depending on whether this image corresponds to a benign or malignant lession. Given the specific region of the image as input, the classification system will measure some prespecified quantities related to the image region (such as its perimeter, its area, etc.), known as *features*. In the sequel, each region is represented by the vector formed by the above features, which is called *feature vector*. Then, based on this vector, a new reduced order feature vector may be formed, each coordinate of which is either one of the coordinates of the original vector and/or a combination of them. This feature vector is then used for the classification of the corresponding lession either to the benign or to the malignant class.

The vector space into which the selected feature vectors lie is known as *feature space*. The dimension of the space depends on the number of features, l, and it can be either continuous or discrete valued. Thus, if in an application the features used are discrete valued (i.e., they may take values only in a finite data set C), the feature space is C^l. If, on the other hand, the features can take values in a continuous real number interval, the feature space is R^l. In the case where the first l_1 features are continuous valued and the remaining $l - l_1$ are discrete valued, taking values in C, the feature space is $R^{l_1} \times C^{l-l_1}$.

Having given the basic definitions, let us now focus on the major design stages of a classification system.

G. Paliouras, V. Karkaletsis, and C.D. Spyropoulos (Eds.): ACAI '99, LNAI 2049, pp. 169–195, 2001.

- *Feature generation:* This stage deals with the generation of the features that will be used to represent an object. These should be chosen so that their values vary significantly among the different classes. The choice is application dependent and they are ususally chosen in cooperation with experts in the field of application.
- *Feature selection:* This stage deals with the selection of features, from a larger number of generated ones, that are most representative for the problem. That is, they are rich in information concerning the classification task. Basically, their values must vary a lot for the different classes. This is an important task. If the selected features are poor in classification related information, the overall performance of the system will be poor.
- *Classifier design:* This stage can be considered as the heart of the classification system. The design of the vector classifier is usually carried out through the optimization of an optimality criterion. Speaking in geometrical terms, the task of a classifier is to divide the feature space into regions each of which corresponds to a class. Thus, for a given feature vector, the classifier identifies the region in the feature space where it belongs and assigns the vector to the corresponding class.
- *System Evaluation:* The final stage consists of the performance evaluation of the system, that is the estimation of the classification error probability. This is also an important task, since in the majority of cases this estimation is based on a limited number of test data. If the error probability turns to be higher than a prespecified threshold, the designer may have to redesign some or all the previous stages.

In the sequel, we will focus on the classfier design stage. Thus, we assume that appropriate features have been selected during the feature genereation and feature selection stage [1]. It is assumed that there is a one to one correspondence between objects and selected feature vectors. Also, unless otherwise stated, we assume that the feature vector consists of real valued coordinates and the feature space is R^l.

2 Bayes Decision Theory

Consider an m-class classification task, i.e., a problem where an object may belong to one out of m possible classes. Let ω_i denote the ith class. The problem that has to be solved by the classifier may be stated as: *"Given a feature vector $\boldsymbol{x}$ which is the most appropriate class to assign it?"*. The most reasonable solution is to assign it to the *most probable* class. Here is the point where probability theory enters into the scene. In this context, the feature vector $\boldsymbol{x}$ will be treated as a random vector, i.e., a vector whose coordinates (features) are treated as random variables.

Let $P(\omega_i|\boldsymbol{x})$ be the probability that $\boldsymbol{x}$ stems from the ω_i class. This is also known as *a posteriori probability*. In mathematical terms the notion of "most probable" is stated as:

[1] Techniques for feature generation and feature selection are discussed in [30]

Assign $\boldsymbol{x}$ *to class* ω_i *if*

$$P(\omega_i|\boldsymbol{x}) = \max_{j=1,\ldots,m} P(\omega_j|\boldsymbol{x}) \ . \tag{1}$$

This is the well known *Bayes classification rule* [2].

Let $P(\omega_i)$ be the a priori probability for class ω_i and $p(\boldsymbol{x}|\omega_i)$ be the class conditional probability density function (pdf) for class ω_i, which describes the distribution of the feature vectors of the ith class, $i = 1, \ldots, m$. Finally, let $p(\boldsymbol{x})$ denote the pdf of $\boldsymbol{x}$. Recalling that

$$P(\omega_i|\boldsymbol{x}) = \frac{P(\omega_i)p(\boldsymbol{x}|\omega_i)}{p(\boldsymbol{x})} \ , \tag{2}$$

the Bayes rule may be stated as

Assign $\boldsymbol{x}$ *to class* ω_i *if*

$$P(\omega_i)p(\boldsymbol{x}|\omega_i) = \max_{j=1,\ldots,m} P(\omega_j)p(\boldsymbol{x}|\omega_j) \ . \tag{3}$$

$p(\boldsymbol{x})$ has been omitted since it is the same for all classes, i.e., independent of ω_i.

In the special case of equiprobable classes, i.e., if $P(\omega_j) = 1/m$, $j = 1, \ldots, m$, the Bayes rule can simply be stated as

Assign $\boldsymbol{x}$ *to class* ω_i *if*

$$p(\boldsymbol{x}|\omega_i) = \max_{j=1,\ldots,m} p(\boldsymbol{x}|\omega_j) \ . \tag{4}$$

That is, in this special case, the Bayes decision rule rests on the values of the class conditional probability densities evaluated at $\boldsymbol{x}$.

Let us consider now fig. 1, where the class conditional pdf's of an one-dimensional two class problem are depicted. The classes are assumed equiprobable. The points $\boldsymbol{x_0}$ and $\boldsymbol{x_1}$ correspond to thresholds that partition the feature space into three regions. Thus, according to the third version of the Bayes decision rule, all $\boldsymbol{x}$ that belong either to R_1 or to R_3 are assigned to class ω_1, and all $\boldsymbol{x} \in R_2$ are assigned to class ω_2.

The above example shows how the condition of the Bayes decision rule may be interpreted in terms of conditions in the feature space. Moreover, the above example indicates that classification errors are unavoidable. Speaking in terms of fig. 1, there is a finite probability for some $\boldsymbol{x}$ to stem from ω_2 and lie in R_1. However, the classifier will assign it to class ω_1.

The probability of a vector $\boldsymbol{x}$ from class ω_i to be wrongly assigned to a different class is

$$\sum_{j=1, j\neq i}^{m} \int_{R_j} p(\boldsymbol{x}|\omega_i)P(\omega_i)d\boldsymbol{x} = 1 - \int_{R_i} p(\boldsymbol{x}|\omega_i)P(\omega_i)d\boldsymbol{x}. \tag{5}$$

[2] In the case where two classes satisfy (1) we may choose arbitrarily one of the classes.

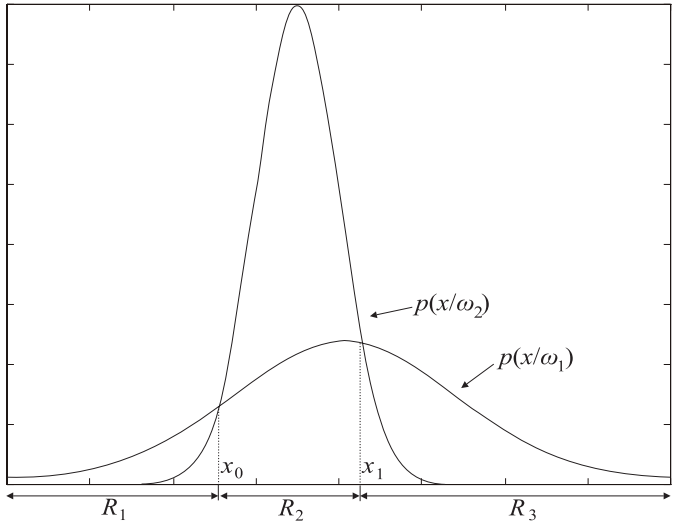

Fig. 1. The class conditional pdf's of a one dimensional two-class problem. The regions R_1 and R_3 are assigned to class ω_1, while R_3 is assigned to class ω_2

Thus, the total probability of committing a decision error P_e is defined as

$$P_e = \sum_{i=1}^{m}[1 - \int_{R_i} p(\boldsymbol{x}|\omega_i)P(\omega_i)d\boldsymbol{x}] \ . \tag{6}$$

It may be shown (see, eg., [30]) that *the Bayes decision rule minimizes the total probability of error.*

3 Discriminant Functions and Discriminant Surfaces

In some cases, instead of working directly with $P(\omega_i|\boldsymbol{x})$ it is more convenient to use $g_i(\boldsymbol{x}) = f(P(\omega_i|\boldsymbol{x}))$, where $f(.)$ is a monotonically increasing function. The $g_i(.)$'s are known as *discriminant functions.* In this case, the Bayes decision rule becomes

Assign $\boldsymbol{x}$ *to class* ω_i *if*

$$g_i(\boldsymbol{x}) = \max_{j=1,\ldots,m} g_j(\boldsymbol{x}) \ . \tag{7}$$

In other words, the set of discriminant functions identifies a partition of the feature space into regions, each one corresponding to a class.

The contiguous regions of the feature space R^l that correspond to different classes are separated by continuous surfaces, known as *decision surfaces.* Thus, the decision surface separating the two regions R_i and R_j associated with classes ω_i and ω_j, respectively, is described by the following equation:

$$g_{ij}(\boldsymbol{x}) = g_i(\boldsymbol{x}) - g_j(\boldsymbol{x}) = 0, \qquad i, j = 1, 2, \ldots, m, \quad i \neq j \ . \tag{8}$$

In the sequel we focus on the case where $p(\boldsymbol{x}|\omega_i)$ are *normal (Gaussian) distributions*, i.e.,

$$p(\boldsymbol{x}|\omega_i) \equiv \mathcal{N}(\boldsymbol{\mu_i}, \Sigma_i) = \frac{1}{(2\pi)^{l/2}|\Sigma_i|^{1/2}} \exp(-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu_i})^T \Sigma_i^{-1}(\boldsymbol{x}-\boldsymbol{\mu_i})), \ , \quad (9)$$

$i = 1, 2, \ldots, m$, where $\boldsymbol{\mu_i} = E[\boldsymbol{x}|\boldsymbol{x} \in \omega_i]$ is the mean vector of the class ω_i, $\Sigma_i = E[(\boldsymbol{x}-\boldsymbol{\mu_i})(\boldsymbol{x}-\boldsymbol{\mu_i})^T]$ is the $l \times l$ covariance matrix for class ω_i and $|\Sigma_i|$ is the determinant of Σ_i. It is clear that only the vectors $\boldsymbol{x} \in \omega_i$ contribute Σ_i.

Our objective is to design a Bayesian classifier taking into account that each $p(\boldsymbol{x}|\omega_i)$ is a normal distribution. Having in mind the Bayesian rule given in (3), we define the discriminant functions

$$g_i(\boldsymbol{x}) = \ln(P(\omega_i)p(\boldsymbol{x}|\omega_i)) \ . \quad (10)$$

Substituting (9) into (10) we obtain

$$g_i(\boldsymbol{x}) = -\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu_i})^T \Sigma_i^{-1}(\boldsymbol{x}-\boldsymbol{\mu_i}) + \ln P(\omega_i) + c_i \ , \quad (11)$$

or, after some maninulations,

$$g_i(\boldsymbol{x}) = -\frac{1}{2}\boldsymbol{x}^T \Sigma_i^{-1}\boldsymbol{x} + \frac{1}{2}\boldsymbol{x}^T \Sigma_i^{-1}\boldsymbol{\mu_i} - \frac{1}{2}\boldsymbol{\mu_i}^T \Sigma_i^{-1}\boldsymbol{\mu_i} + \frac{1}{2}\boldsymbol{\mu_i}^T \Sigma_i^{-1}\boldsymbol{x} + \ln P(\omega_i) + c_i \ , \quad (12)$$

where c_i is a constant equal to $-\frac{l}{2}\ln(2\pi) - \frac{1}{2}\ln|\Sigma_i|$.

Clearly, the decision surfaces defined as $g_i(\boldsymbol{x}) - g_j(\boldsymbol{x}) = 0$ are *quadrics* (hyperellipsoids, hyperparaboloids, pairs of hyperplanes, etc.) and the resulting Bayesian classifier is a *quadric classifier*.

Special cases:

I: Assuming that the covariance matrices for all classes are equal to each other, i.e., $\Sigma_i = \Sigma$, $i = 1, 2, \ldots, m$, the terms $-\frac{1}{2}\boldsymbol{x}^T \Sigma_i^{-1}\boldsymbol{x}$ and c_i are the same in all g_i's, thus they can be omitted. In this case, $g_i(\boldsymbol{x})$ becomes a linear function of $\boldsymbol{x}$, and it can be rewritten as:

$$g_i(\boldsymbol{x}) = \boldsymbol{w_i}^T\boldsymbol{x} + w_{i0} \ , \quad (13)$$

where

$$\boldsymbol{w_i} = \Sigma^{-1}\boldsymbol{\mu_i} \quad (14)$$

and

$$w_{i0} = -\frac{1}{2}\boldsymbol{\mu_i}^T \Sigma^{-1}\boldsymbol{\mu_i} + \ln P(\omega_i) \ . \quad (15)$$

In this case, the decision surfaces $g_i(\boldsymbol{x}) - g_j(\boldsymbol{x}) = 0$ are hyperplanes and the Bayesian classifier is a *linear classifier*.

II: Assume, in addition, that all classes are equiprobable, i.e. $P(\omega_i) = 1/m$, $i = 1, 2, \ldots, m$. Then eq. (11) gives

$$g_i(\boldsymbol{x}) = -\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu_i})^T \Sigma^{-1}(\boldsymbol{x}-\boldsymbol{\mu_i}) \equiv -\frac{1}{2}d_m^2 \ , \quad (16)$$

The quantity d_m in the right hand side of the above equation, without the minus sign, is called *Mahalanobis distance*. Thus, in this case, instead of searching for the class with the maximum $g_i(\boldsymbol{x})$, we can equivalently search for the class for which the Mahalanobis distance between the respective mean vector and the input vector $\boldsymbol{x}$ is minimum.

III: In addition to the above asumptions, assume that $\Sigma = \sigma^2 I$, where I is the $l \times l$ identity matrix. In this case, eq. (16) gives

$$g_i(\boldsymbol{x}) = -\frac{1}{2\sigma^2}(\boldsymbol{x} - \boldsymbol{\mu_i})^T(\boldsymbol{x} - \boldsymbol{\mu_i}) \ , \tag{17}$$

or, eliminating the factor $\frac{1}{2\sigma^2}$, which is common for all g_i's, we obtain:

$$g_i(\boldsymbol{x}) = -(\boldsymbol{x} - \boldsymbol{\mu_i})^T(\boldsymbol{x} - \boldsymbol{\mu_i}) \equiv -d_\varepsilon^2 \ . \tag{18}$$

Clearly, searching for the class with the maximum $g_i(\boldsymbol{x})$ is equivalent to searching for the class for which the Euclidean distance, d_ε, between the respective mean vector and the input vector $\boldsymbol{x}$ becomes minimum.

In summary, in the case where each $p(\boldsymbol{x}|\omega_i)$ is a normal distribution, $g_i(\boldsymbol{x})$'s are quadratic functions of $\boldsymbol{x}$ and the decision surface between any two classes is a quadric. If, in addition, all the classes have equal covariance matrices, $g_i(\boldsymbol{x})$'s are linear functions of $\boldsymbol{x}$ and the decision surfaces are hyperplanes.

4 Estimation of Unknown Probability Density Functions

As we saw above, the Bayes decision rule requires the knowledge of $p(\boldsymbol{x}|\omega_i)$'s and $P(\omega_i)$'s, $i = 1, 2, \ldots, m$. However, in practice, this is rarely the case. In the majority of the cases, all we have at our disposal is a finite set of feature vectors from each class known as the *training set*, i.e.,

$$S = \{(\boldsymbol{x_i}, c_i), \boldsymbol{x_i} \in A(\subseteq R^l), c_i \in \{1, 2, \ldots, m\}, i = 1, 2, \ldots, N\} \ , \tag{19}$$

where c_i is the index indicating the class to which $\boldsymbol{x_i}$ belongs.

A commonly used estimate for $P(\omega_i)$ ([30]) is

$$P(\omega_i) = \frac{n_i}{N} \ , \tag{20}$$

where n_i is the number of vectors in the data set that belong to class ω_i.

However, the estimation of $p(\boldsymbol{x}|\omega_i)$ is not so straightforward. One way to attack the problem of estimating $p(\boldsymbol{x}|\omega_i)$ is to assume that it has a given parametric form and to establish certain optimality criteria, the optimization of which will give estimates for the unknmown parameters of the distribution, using only the feature vectors available for class ω_i. If, for example, we assume that $p(\boldsymbol{x}|\omega_i)$ is a normal distribution, the unknown parameters that have to be estimated, may be the mean vector and/or the covariance matrix.

Parametric methods that follow this philosophy are the Maximum Likelihood Parameter Estimation, the Maximum A Posteriori Probability Estimation and the Mixture models (see eg. [8], [22], [30]).

An alternative way for estimating $p(\boldsymbol{x}|\omega_i)$, using the feature vectors available for ω_i, requires no assumptions of parametric models for $p(\boldsymbol{x}|\omega_i)$. The basic idea relies on the approximation of the unknown pdf using the histogram method. Let us consider the one dimansional case ($l = 1$). In this case the data space is divided into intervals (bins) of size h. Then, the probability of x lying in a specific bin is approximated by the frequency ratio, i.e.,

$$P \simeq \frac{k_N}{N} \ , \tag{21}$$

where k_N is the number of data points lying in this bin. As $N \to +\infty$, the frequency ratio converges to the true P. The corresponding pdf value is assumed to be constant throughout the bin and is approximated by

$$\hat{p}(x) = \frac{1}{h}\frac{k_N}{N}, \quad |x - \hat{x}| \leq \frac{h}{2} \ , \tag{22}$$

where $\hat{x}$ is the center of the bin.

Generalizing to the l dimensional case, the bins become l dimensional hypercubes of edge h and $\hat{p}(\boldsymbol{x})$ becomes

$$\hat{p}(\boldsymbol{x}) = \frac{1}{h^l}\frac{k_N}{N} \ , \tag{23}$$

where now k_N is the number of feature vectors lying in the l dimensional hypercube where $\boldsymbol{x}$ belongs.

Let us define $\phi(\boldsymbol{x_i})$ as

$$\phi(\boldsymbol{x_i}) = \begin{cases} 1, if \ |x_{ij}| < \frac{1}{2}, \ for \ j = 1, \ldots, l \\ 0, otherwise \end{cases} \ , \tag{24}$$

where x_{ij} is the jth coordinate of the ith vector. In words, $\phi(\boldsymbol{x_i})$ equals to 1 if $\boldsymbol{x_i}$ lies in the l-dimensional unit hypercube centered at the origin $\mathbf{0}$ and 0 otherwise. Then, eq. (23) may be rewritten in the following more general form

$$\hat{p}(\boldsymbol{x}) = \frac{1}{h^l}\left(\frac{1}{N}\sum_{i=1}^{N}\phi\left(\frac{\boldsymbol{x_i} - \boldsymbol{x}}{h}\right)\right) \ . \tag{25}$$

However, in the above equation, we try to approximate a continuous function ($\hat{p}(\boldsymbol{x})$) in terms of a linear combination of instances of a discontinuous one ($\phi(.)$). Parzen ([23]) generalized (25) by using smooth functions ϕ that satisfy

$$\phi(\boldsymbol{x}) \geq 0 \ ,$$

and

$$\int_{\boldsymbol{x}} \phi(\boldsymbol{x}) d\boldsymbol{x} = 1 \ , \tag{26}$$

and showed that such choices of ϕ lead to legitimate estimates of pdf. A widely used form of ϕ is the Gaussian ([30]).

5 Nearest Neighbor Rules

A different kind of rules that are suboptimal in the sense that they lead to error P_e grater than that of the Bayesian clasifier, are the k-nearest neighbor (k-NN) rules. The general idea is the following: for a given vector $\boldsymbol{x}$ we determine the set A of the k closest neighbors of $\boldsymbol{x}$ among the feature vectors of S (see (19)), irrespective to the class where they belong. Let $k_j(A)$, $j = 1, 2, \ldots, m$ be the number of feature vectors in A that belong to class ω_j. We determine, among the $k_j(A)$'s the largest one, say $k_i(A)$, and we assign $\boldsymbol{x}$ to class ω_i. The Euclidean is the most frequently used distance between vectors although other distances may also be used.

A popular rule that belongs to this category is the nearest neighbor (NN) rule, which results from the above general scheme for $k = 1$. According to this rule, a vector $\boldsymbol{x}$ is assigned to the class where its closest vector in S belongs. It has been shown that the probability of error for the NN classifier is no greater than twice the probability of error of a Bayesian classifier, P_B ([8], [7]). Upper bounds for the probability of error are also available for the general case where $k > 1$. It is worth noting that as $k \to +\infty$, the probability of error of the k-NN classifier approaches P_B ([7], [30]).

Note that all the above schemes require at least $O(N)$ operations, which in cases where large data sets are to be utilized require excessive computational effort. One way to avoid this problem, at the cost of loss of information, is to cluster feature vectors of the same class that are close to each other and to use the mean vectors of the resulted clusters as data set. Such an approach reduces significantly the size of the data set.

6 Linear Classifiers

In the sequel, we consider the case of a two-class problem ($m = 2$) and we label the two classes ω_1 and ω_2 as 1 and 0, respectively. In the previous section we discussed how discriminant functions, that stem from pdf's, are related to decision surfaces. From now on we will emancipate from the notion of probability densities and our goal will be to design classifiers that divide the feature space into class regions, via appropriately chosen discriminant functions. These have not necessarily any relation to probability densities and the Bayesian rule. The aim is to estimate a set of unknown parameters of the discriminant function in a way that place the equivalent decision surfaces optimally in the feature space, according to an optimality criterion.

In this section we deal with linear classifiers, that is, with classifiers that implement linear decision surfaces, i.e. hyperplanes. A hyperplane is described as

$$g(\boldsymbol{x}) = \sum_{i=1}^{l} w_i x_i + w_0 = \boldsymbol{w}^T \boldsymbol{x} + w_0 = 0 \ , \tag{27}$$

where $\boldsymbol{w} = [w_1, \ w_2, \ \ldots, w_l]^T$ are the parameters specifying the hyperplane. Note that $(\boldsymbol{w}, w_0)$ define uniquely a single hyperplane.

For the compactness of notation, we embed w_0 into $\boldsymbol{w}$ and we augment the feature vector $\boldsymbol{x}$ with one more coordinate which is equal to 1, i.e., $\boldsymbol{w} = [w_0,\ w_1,\ w_2,\ \ldots, w_l]^T$ and $\boldsymbol{x} = [1,\ x_1,\ x_2,\ \ldots, x_l]^T$. Thus the last equation becomes

$$g(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{x} = 0\ . \tag{28}$$

Our aim is to determine the optimal hyperplane that separates the two classes. Let us define

$$\delta_{\boldsymbol{x}} = \begin{cases} 1, & if\ \boldsymbol{x} \in \omega_2 \\ -1, & if\ \boldsymbol{x} \in \omega_1 \end{cases}\ . \tag{29}$$

Then a vector $\boldsymbol{x}$ is *miclassified (correctly classified)* by the hyperplane $\boldsymbol{w}$ if

$$\delta_{\boldsymbol{x}}(\boldsymbol{w}^T\boldsymbol{x}) > (<)0. \tag{30}$$

Let us now assume that the feature vectors of the two classes that belong to S are lineraly separable, i.e., there exists a hyperplane $\boldsymbol{w}^*$ that leaves all feature vectors of S that belong to class 1 (0) in its positive (negative) half-space. An algorithm that is appropriate for the determination of such a hyperplane that classifies correctly the feature vectors of S, is the *perceptron algorithm* (e.g [26], [8], [20], [30]). This may be viewed as an optimization task as described below.

Let us define the following optimization function

$$J(\boldsymbol{w}) = \sum_{\boldsymbol{x} \in \mathcal{Y}_{\boldsymbol{w}}} \delta_{\boldsymbol{x}}(\boldsymbol{w}^T\boldsymbol{x})\ , \tag{31}$$

where $\mathcal{Y}_{\boldsymbol{w}}$ is the set of all misclassified feature vectors of S by $\boldsymbol{w}$. Clearly, $J(\boldsymbol{w})$ is a non-negative function taking its minimum value when $\mathcal{Y}_{\boldsymbol{w}} = \emptyset$ [3]. Then, the perceptron algorithm may be written as a gradient descent like method that minimizes $J(\boldsymbol{w})$ as follows

$$\boldsymbol{w}(t+1) = \boldsymbol{w}(t) - \rho_t \frac{\partial J(\boldsymbol{w})}{\partial \boldsymbol{w}}|_{\boldsymbol{w}=\boldsymbol{w}(t)}\ . \tag{32}$$

Note, however, that this is not a true gradient descent scheme since $J(\boldsymbol{w})$ is piecewise linear, thus, non differentiable at the points of discontinuity. For the points where $\partial J(\boldsymbol{w})/\partial \boldsymbol{w}$ is defined we can write

$$\frac{\partial J(\boldsymbol{w})}{\partial \boldsymbol{w}} = \sum_{\boldsymbol{x} \in \mathcal{Y}_{\boldsymbol{w}}} \delta_{\boldsymbol{x}}\boldsymbol{x}\ . \tag{33}$$

Substituting the above result to eq. (32), we obtain

$$\boldsymbol{w}(t+1) = \boldsymbol{w}(t) - \rho_t \sum_{\boldsymbol{x} \in \mathcal{Y}_{\boldsymbol{w}(t)}} \delta_{\boldsymbol{x}}\boldsymbol{x}\ . \tag{34}$$

This is the updating equation of the perceptron algorithm. Note that this equation is defined at all points. Typical choices for ρ_t are $\rho_t = c/t$, where c is a constant and $\rho_t = \rho < 2$.

[3] Or in the trivial case where $\boldsymbol{w} = \boldsymbol{0}$.

It can be shown that *if the available feature vectors of the two classes are linearly separable, then the perceptron algorithm converges in a finite number of steps to a* $\boldsymbol{w}^*$ *that separates perfectly these vectors.* However, when the available feature vectors of the two classes are not linearly separable the perceptron algorithm does not converge. A variant of this algorithm that converges (under certain conditions) to an optimal solution (i.e., to a solution with the minimum possible number of misclassified feature vectors) is the *pocket algorithm* ([10]).

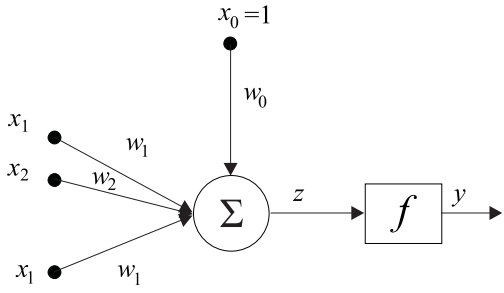

Fig. 2. The perceptron

Let us consider now the structuring element shown in fig. 2, with parameters $w_0,\ w_1,\ \ldots, w_l$. Let also $f(z)$ be the *hard limiter function* defined as

$$f(z) = \begin{cases} 1, \, if \ z \geq 0 \\ 0, \, otherwise \end{cases} . \tag{35}$$

The operation of this element is the following: Once $x_1,\ x_2,\ \ldots, x_l$ are applied to the input nodes, they are weighted by $w_1,\ w_2,\ \ldots, w_l$, respectively, and the results are summed , i.e.,

$$z = \sum_{i=1}^{l} w_i x_i + w_0 = \boldsymbol{w}^{\boldsymbol{T}} \boldsymbol{x} \ , \tag{36}$$

where $\boldsymbol{w} = [w_0,\ w_1, \ldots, w_l]^T$ and $\boldsymbol{x} = [1,\ x_1, \ldots, x_l]^T$. Then, the output of this element is 1 if $z \geq 0$ and 0 otherwise. The w_i's are known as *synaptic weights* or simply *weights.* Also, $\boldsymbol{w}$ and w_0 are called *weight vector* and *threshold*, respectively. Finally, the function f is called *activation function.*

The above structuring element is known as *percepton.* Clearly, *it implements the separation of the feature space by the hyperplane* H *defined by the parameters* $w_i,\ i = 0, \ldots, l$. That is, the perceptron will output 1 if the vector applied to its input nodes lies in the positive side of H and 0 otherwise.

The perceptron, with a generalized activation function f, will be the basic structuring element for most of the more complicated structures, known as *neural networks* that will be discussed next. In this context, it will be known as *neuron* or *node.*

7 Neural Network Classifiers

As it was discussed in the previous section, the perceptron structure can deal with two-class problems, provided that the classes are linearly separable. Two well known linearly separable problems are the AND and OR problems (see fig. 3(a)-(b)). The question that arises now is how often linear separability is met in practice. One has not to go very far to see that even well known simple problems are not linearly separable. Consider for example the XOR problem, where $(0,0)$ and $(1,1)$ are assigned to class 0 and $(0,1)$ and $(1,0)$ are assigned to class 1 (see fig. 3(c)). Clearly, this problem cannot be solved by a single perceptron, since a single perceptron realizes a single hyperplane and there is no hyperplane that separates perfectly the vectors of the two classes.

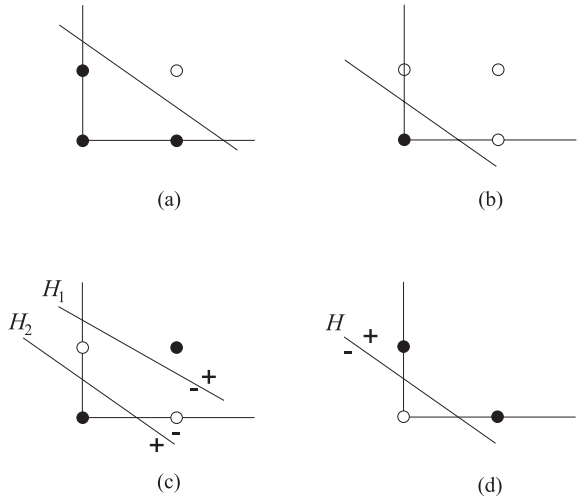

Fig. 3. (a) The AND problem. (b) The OR problem. (c) The XOR problem. (d) The transformed space for the XOR problem. The open (fiiled) circles correspond to class 1 (0)

Table 1. The XOR problem

x_1	x_2	y_1	y_2	y
0	0	0	1	0
0	1	0	0	1
1	0	0	0	1
1	1	1	0	0

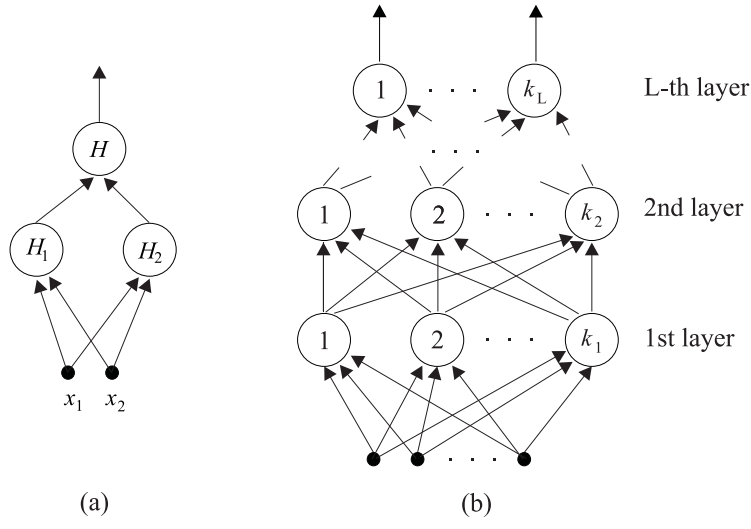

Fig. 4. (a) The network that solves the XOR problem. (b) A generalized neural network architecture

In order to solve this problem two hyperplanes are needed, as shown in fig. 3(c). Taking into account the relative position of each point with respect to each hyperplane, we achieve the transformation of the problem as shown in fig. 3(d) and table 1. That is, y_1 and y_2 take the values 0 or 1 depending on the relative position (+ or −) of the point (x_1, x_2) with respect to the respective hyperplane. Basically, this is a transformation of the original space into the vertices of the unit square. Clearly, a single hyperplane now suffices to separate the two classes in the transformed space (y_1, y_2). Speaking in terms of neurons, we need two neurons to realize the two hyperplanes H_1 and H_2 and a single neuron that combines the outputs of the previous two neurons and realizes a third hyperplane in the (y_1, y_2) space. Thus, the structure shown in fig. 4(a) results, where each neuron has the structure shown in fig. 2 (For simplicity, the neuron thresholds are not shown explicitly). This is known as a *two-layer perceptron*. It consists of two layers of neurons. The first layer consists of two neurons and the second layer of a single neuron. The input layer has as many input neurons as the dimensionality of the input space. This is a special type of a neural network architecture.

In general, a Neural Network consists of L layers of neurons (excluding the input layer) and the rth layer consists of k_r neurons (see fig. 4(b)). The Lth (final) layer is known as the *output layer* and its neurons *output neurons*. All the other layers are called *hidden layers* and the respective neurons *hidden neurons*.

So far we have seen how a two layer neural network can be used to solve the XOR problem. The natural question that now arises is whether there are partitions of the feature space into classes, via hyperplanes, that cannot be solved by a two-layer architecture. Let us consider the example of fig. 5(a), where the shaded regions correspond to class 1 (ω_1) and the rest to class 0 (ω_2). Clearly, a

two layer network that will tackle this problem must have in its first layer as many neurons as the hyperplanes that define the regions in the feature space. Thus, speaking in the spirit of the solution of the XOR problem, each of these neurons realizes one of the hyperplanes and at the same time perform a transformation of the original $\boldsymbol{x}$ space into the (y_1, y_2, y_3) H_3 hypercube space, depending on the position (1 or 0) of $\boldsymbol{x}$ with respect to the respective plane (fig. 5(b)). However, no single hyperplane, realized by the output neuron in the 3-dimensional (y_1, y_2, y_3) space, can separate the vertices of the hypercube that correspond to class 1 from those that correspond to class 0. Thus, this problem cannot be solved by a two-layer network.

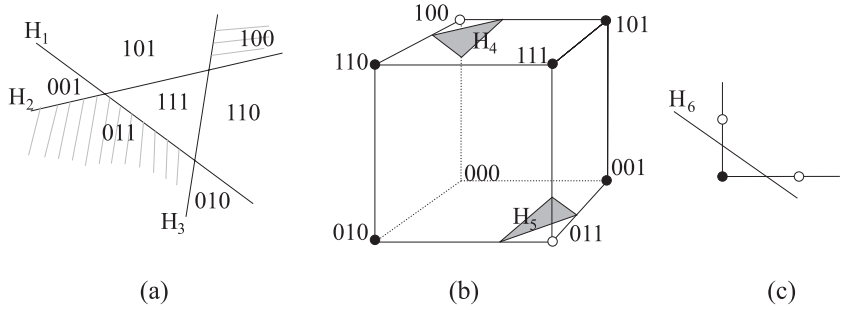

Fig. 5. (a) Classes separated by hyperplanes. (b)-(c) The transformed spaces (see text for explanation). The open (filled) circles correspond to class 1 (0)

However, this problem can be solved by a three layer network as follows. Having transformed the original problem to that of fig. 5(b), we can use two neurons to realize the hyperplanes H_4 and H_5 in the transformed 3-dimensional space. H_4 (H_5) leaves the vertex $(1, 0, 0)$ ($(0, 1, 1)$) to its positive side and all the others to its negative side. Thus, we achieve a further transformation as shown in fig. 5(c), where class 0 (ω_2) is represented by a single vertex in the two-dimensional hypercube (square). Clearly, the problem now in this transformed space (fig. 5(c)) is linearly separable. The network that implements the given partition is shown in fig. 6.

The above construction (which by no means is optimal with respect to the number of neurons used) can be applied to any partition of the input space in two classes, where each class is a union of polyhedral sets. A *polyhedral set* is a union of half-spaces. For example, in fig. 5(a) we have three hyperplanes (lines in R^2) that form seven non-intersected polyhedral sets.

In summary, we can say that *the first layer neurons form hyperplanes, the second layer neurons form regions, and the third layer neurons form classes.*

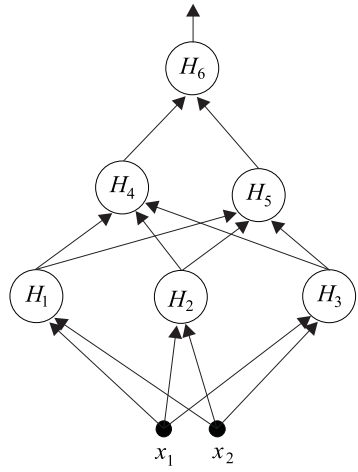

Fig. 6. The network implementing the partition shown in fig. 5(a)

7.1 Choosing the Size of the Network

Having established the capabilities of neural networks in a more theoretical context, one has to find ways for utilizing them in real world situations, where, in most of the times, the only available information is a finite set of training vectors S. More specifically, we need to determine the right size of a network, as well as all the values of the weights connecting its neurons and the neuron thresholds. The size of the network plays a very important role. *The right size for the network should be large enough to be able to learn the differences between the different classes and small enough to be unable to distinguish the differences between the feature vectors of the same class.*

There are three basic directions to approach this problem.

- *Constructive techniques:* The idea here is to begin with a small neural network architecture (often unable to solve the problem at hand) and to augment it, utilizing the available training data set, until a specific criterion is satisfied. One such criterion (not necessarily the best one) is the resulting architecture to classify correctly *all* the available training vectors.
  In this case, the size of the network as well as the values of its weights are adjusted simultaneously. Some algorithms of this category are the Divide and Conquer algorithm ([5], [17]), the Cascade Correlation algorithm ([9]), the Constructive Learning by Specialization algorithm([25]), the Tilling algorihm ([19]) etc.
- *Prunning techniques:* Here one begins with a very large architecture and gradually reduces it by prunning weights or even neurons when certain predefined criteria are met. This procedure is repeated until a specific termination criterion is satisfied (see eg., [18], [11], [25], [28], [30]).

– *Analytical methods:* In this case, the size of the network is estimated a priori, using statistical or algebraic methods, and then, an appropriate algorithm is employed in order to determine the values of the weights and the thresholds of the network. Note that the network structure is kept unaltered when esitmation of the weights and the thresholds takes place.

7.2 The Back Propagation Algorithm

By far, the most well known and widely used learning algorithm to estimate the values of the synaptic weights and the thresholds, is the so called *Back Propagation (BP) algorithm* and its variants ([33], [27], [30]). According to this training method, the size (number of layers and number of nodes in each layer) of the network remains fixed. In general, the procedure of determining the values of the unknown parameters of a network is known as *training of the network*.

The BP algorithm relies on a training data set S (see (19)). Note that we can use various representations for labeling the respective class of each vector of the training set. One representation that is frequently used in practice is the following:

$$\boldsymbol{y(i)} = [\overbrace{0, \ldots, 0, \underbrace{1}_{c_i}, 0, \ldots, 0}^{m}] \ , \tag{37}$$

where $\boldsymbol{y(i)}$ is the vector having zeros everywhere except at position c_i, that corresponds to the class to which the ith training vector belongs, where it has 1.

The BP algorithm is a gradient descent scheme that determines a local minimum of the cost function which is adopted. The most popular cost function is the following:

$$J = \sum_{i=1}^{N} \mathcal{E}(i) \ , \tag{38}$$

where

$$\mathcal{E}(i) = \frac{1}{2} \sum_{p=1}^{k} e_p^2(i) = \frac{1}{2} \sum_{p=1}^{k} (\hat{y}_p(i) - y_p(i))^2 \ , \tag{39}$$

with $\hat{y}_p(i)$ being the true output of the pth output node of the network when the ith feature vector is fed to the network and k is the number of neurons of the output layer. Note that, in general, y_p and $\hat{y}_p$ are different and are known as the desired and true outputs, respectively.

In the sequel we adopt the representation of classes described above. Thus, we have $k = m$.

The gradient descent optimization procedure requires the computation of the gradient of the cost function with respect to the unknown parameters. However, differentiation of J is not possible, since $\hat{y}_p(i)$ is expressed via the hard limiter function, which is not a differentiable function. One way to overcome this difficulty is to use a smooth function that approximates the hard limiter function,

which will be continuous and differentiable. One such function is the *logistic function*, shown in fig. 7, which is defined as

$$f(x) = \frac{1}{1 + \exp(-ax)} \ , \tag{40}$$

where a is the slope parameter. The larger the value of a, the more the logistic function approaches the hard limiter. Clearly, this function takes values between 0 and 1. This function belongs to a broader class of functions known as *squashing functions*, whose output is limited in a continuous real number interval. Other such functions are also possible (e.g., [30]).

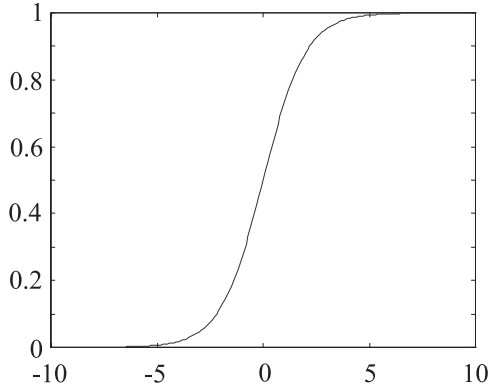

Fig. 7. The logistic function

An important property of a neuron with sigmoid activation function is that *it gives the same output for all points lying on a hyperplane parallel to the hyperplane implemented by the neuron.* This can be easily verified as follows. Let H be the hyperplane $z = \boldsymbol{w}^T\boldsymbol{x}$ implemented by the neuron with $\boldsymbol{w} = [w_0,\ w_1,\ \ldots, w_l]^T$, and H_1 be the hyperplane $z = \boldsymbol{w_1}^T\boldsymbol{x}$, parallel to H at distance d. Let $\boldsymbol{x'}$ be a point on H_1. Then, the distance between $\boldsymbol{x'}$ and H is

$$d(\boldsymbol{x'}, H) = d = \frac{\boldsymbol{w}^T\boldsymbol{x'}}{\sqrt{\sum_{i=1}^{l} w_i}}^{4} \ . \tag{41}$$

or

$$\boldsymbol{w}^T\boldsymbol{x'} = d\sqrt{\sum_{i=1}^{l} w_i} \ . \tag{42}$$

[4] A negative distance means that the input vector $\boldsymbol{x}$ lies in the negative half-space of H.

Since the right hand side term of the last equality is constant, for all points in H_1, and taking into account that the output of the node is equal to $1/(1 + \exp(-a(\boldsymbol{w}^T \boldsymbol{x}')))$, the claim follows.

Because of the above property, we say that the output of such a neuron is of *global nature*.

Note that what has been said, concerning the classification capabilities of neural networks consisted of neurons with hard limiter activation functions, in the case of neurons with logistic activation functions is valid only approximately for large values of a.

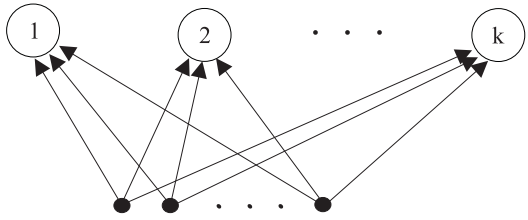

Fig. 8. A one-layer network

Let us now return to the BP algorithm. First, we will derive it for a one layer network (see fig. 8). In this case, $\hat{y}_p(i)$ is given as

$$\hat{y}_p(i) = f(z_p(i)) = f(\sum_{j=0}^{l} w_{pj} x_j(i)) = f(\boldsymbol{w_p}^T \boldsymbol{x}(i)) \ , \tag{43}$$

where w_{pj} denotes the weight from the jth input neuron to the pth output neuron, w_{p0} is the threshold of the pth neuron, $\boldsymbol{w_p} = [w_{p0},\ w_{p1}, \ldots, w_{pl}]^T$ is the weight vector of the pth neuron, $x_j(i)$ is the jth coordinate of the ith input vector and $\boldsymbol{x}(i) = [1,\ x_1(i), \ldots, x_l(i)]^T$ is the input vector. Also, $z_p(i)$ is the weighed sum of the inputs to the pth output neuron.

The basic equation of the BP algorithm, at the tth iteration step, is

$$\boldsymbol{w_p}(t+1) = \boldsymbol{w_p}(t) + \Delta \boldsymbol{w_p}(t+1), \qquad p = 1, \ldots, k \ , \tag{44}$$

where

$$\Delta \boldsymbol{w_p}(t+1) = -\rho \frac{\partial J(\boldsymbol{w_p})}{\partial \boldsymbol{w_p}} |_{\boldsymbol{w_p} = \boldsymbol{w_p}(t)} \ . \tag{45}$$

Let us define

$$\delta_p(i) = \frac{\partial \mathcal{E}(i)}{\partial z_p(i)} \tag{46}$$

and

$$e_p(i) = \hat{y}_p(i) - y_p(i) \ . \tag{47}$$

It is easy to see that

$$\delta_p(i) = \frac{\partial \mathcal{E}(i)}{\partial z_p(i)} = (\hat{y}_p(i) - y_p(i)) f'(z_p(i)) = e_p(i) f'(z_p(i)) \ . \tag{48}$$

Also

$$\frac{\partial z_p(i)}{\partial \boldsymbol{w_p}} = \boldsymbol{x}(i) \ . \tag{49}$$

Combining eqs. (44), (45), (38), (46), (49) we finally obtain

$$\boldsymbol{w_p}(t+1) = \boldsymbol{w_p}(t) - \rho[\sum_{i=1}^{N} \delta_p(i) \boldsymbol{x}(i)] \ , \tag{50}$$

where $\delta_p(i)$ is given by eq. (48).

Let us now state explicitly the steps of the BP algorithm for the one layer case.

- Initialize the weight vectors $\boldsymbol{w_p}$, $p = 1, \ldots, k$
- Repeat
  - For $i = 1$ to N
    - * Present $\boldsymbol{x}(i)$ to the network
    - * For each neuron compute $\hat{y}_p(i)$, $e_p(i)$ and $\delta_p(i)$, $p = 1, \ldots, k$
  - End { For }
  - Use eq. (50) to update the weight vectors $\boldsymbol{w_p}$, $p = 1, \ldots, k$
- Until a specific termination criteron is met

The above algorithm is also known as the *delta rule* [Rummelhart]. Moreover, if $f(.)$ is the identity function, the delta rule coincides with the well known LMS rule [31], [32].

Let us now move to the more general case of multilayer neural networks, i.e., networks with more than one layers. In this framework we adopt the following notation: L is the number of layers of the network (excluding, of course, the input layer), k_q is the number of neurons in the qth layer, $q = 1, \ldots, L$, $\boldsymbol{w_j^q}$ is the weight vector for the jth neuron in the qth layer, w_{ji}^q is the weight connecting the jth neuron of the qth layer with the ith neuron of the previous layer, $y_j^q(i)$ is the output of the jth neuron in the qth layer, when $\boldsymbol{x}(i)$ is the input to the network, $\boldsymbol{y^q}(i)$ is the output vector of the qth layer. Here, the convention that $\boldsymbol{y^0}(i) = \boldsymbol{x}(i)$ has been adopted. Also, we define

$$z_j^q(i) = \boldsymbol{w_j^{qT}} \boldsymbol{y^{q-1}}(i) \tag{51}$$

and

$$e_j^L = \hat{y}_j^L(i) - y_j^L(i) \ . \tag{52}$$

Note that the threshold w_{j0}^q of each neuron in any layer is embedded in $\boldsymbol{w_j^q}$, i.e., we define $\boldsymbol{w_j^q} = [w_{j0}^q, \ w_{j1}^q, \ldots, w_{jk_{q-1}}^q]^T$ and
$\boldsymbol{y^{q-1}}(i) = [1, \ y_1^{q-1}(i), \ldots, y_{k_{q-1}}^{q-1}(i)]^T$.

Using the above notation, the updating equation (50) is written as

$$\boldsymbol{w_p^q}(t+1) = \boldsymbol{w_p^q}(t) - \rho[\sum_{i=1}^{N} \delta_p^q(i)\boldsymbol{y^{q-1}}(i)] \ . \tag{53}$$

The terms $\delta_m^L(i)$ for the output layer neurons are computed as in the one layer case and in this context we can write

$$\delta_p^L(i) = (\hat{y}_p^L(i) - y_p(i))f'(z_p^L(i)) = e_p^L(i)f'(z_p^L(i)) \ . \tag{54}$$

The problem now is the computation of the terms $\delta_m^q(i)$ for $q < L$. In this case we use the chain rule in differentiation, which, after some manipulations, gives (see e.g. [30])

$$\delta_p^q(i) = e_p^q(i)f'(z_p^q(i)) \ , \tag{55}$$

with

$$e_p^q(i) = \sum_{s=1}^{k_{q+1}} \delta_s^{q+1}(i)w_{sp}^{q+1} \ . \tag{56}$$

Note that $e_p^q(i)$ for $q < L$ is not a true error, as the one given in eq. (52), but it is defined as such for notational uniformity.

The above two equations provide the term $\delta_p^q(i)$ in terms of all $\delta_p^{q+1}(i)$'s of the next layer. Thus, for a given input vector $\boldsymbol{x}(i)$, we compute the output vector $\boldsymbol{y^L}(i)$ and the terms $\delta_p^L(i)$'s. Then we move one layer back, in order to compute $\delta_p^{L-1}(i)$'s in terms of $\delta_p^L(i)$'s, through eqs. (55) and (56). Continuing this process backwards we can compute the terms $\delta_p^q(i)$ for all neurons and then we use eq. (53) to update the weight vectors. It is this way of moving backwards, in order to compute the terms $\delta_p^q(i)$, that gives to the Back Propagation algorithm its name.

Let us state now explicitly the BP algorithm

- Initialize the weight vectors $\boldsymbol{w_p^q} = \boldsymbol{w_p^q}(0)$, $p = 1, \ldots, k_q$, $q = 1, \ldots, L$.
- Repeat
  - For $i = 1$ to N
    - Present the vector $\boldsymbol{x}(i)$ to the network
    - Compute $\hat{y}_p^L(i)$, $p = 1, \ldots, k_L$
    - Compute $\delta_p^L(i)$, $p = 1, \ldots, k_L$, using eq. (54)
    - For $q = L - 1$ to 1
      - Compute $\delta_p^q(i)$, $p = 1, \ldots, k_q$ using eqs. (55) and (56)
    - End
  - End
  - Update the weight vectors of the network using eq. (53)
- Until a specific termination criterion is met

The time needed for the presentation of all training feature vectors to the network and the computation of the $\delta_p^q(i)$'s, $i = 1, \ldots, N$, i.e., the time required for the execution of the statements in the for - loop once, is known as an *epoch*.

Some comments about the BP algorithm are now in order.

- Usually, the number of the output neurons in a neural network, used for classification, equals to the number of classes involved in the problem at hand, i.e. each neuron corresponds to a class. This implies that the representation given in (37) for the class indices is employed.
- The BP algorithm is a nonlinear optimization task due to the nonlinearity of the activation functions of the neurons of the network. This implies that the cost function J has local minima and the algorithm may converge to one of them. However, a local minimum is not necessarily a bad solution as we will see soon.
- As a termination criterion we may require that the value of J falls below a prespecified threshold or that the gradient of J is close enough to zero. The last criterion indicates convergence to a local minimum.
- The quantity ρ, which is known as *learning rate*, plays a very important part to the convergence of the algorithm. It should be sufficiently small, in order to guarantee convergence, but not very small, so that to slow down the convergence procedure. In regions of the weight space where broad minima (A in fig. 9) are encountered, ρ should be larger in order to accelerate convergence. On the other hand, in regions with very narrow minima (B in fig. 9), small ρ should be used in order to avoid oscillation around the local minimum.

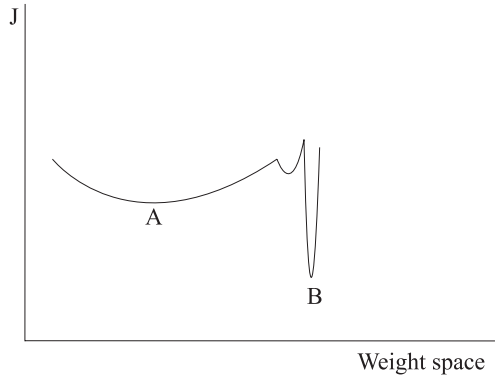

Fig. 9. A broad minimum (A) and a narrow minimum (B)

- In the BP version presented above, the updating of the network weights takes place once all feature vectors have been presented to the network. This mode of learning is called *batch mode*. A different mode arises if we update the weights of the network each time a new vector is presented to the network. This mode is called *pattern mode* of operation. In practice, in

the pattern mode of operation, for each epoch, the sequence of vectors in which they are presented to the network is randomized.

Training in the batch mode exhibits a smoother convergence behavior. On the other hand, training in the pattern mode offers the algorithm the possibility of escaping from a local minimum ([30]).

- In practice, it is common that the change of the cost function gradient between successive iteration steps is not smooth but oscillatory, which results in slow convergence. One way to speed up convergence is to use a momentum term that smooths out the oscillatory behavior and speeds up convergence. Then, the correction term $\Delta \boldsymbol{w}_{\boldsymbol{p}}^{\boldsymbol{q}}$ is no more a function of the $\partial J/\partial \boldsymbol{w}_{\boldsymbol{p}}^{\boldsymbol{q}}$ only, but it also depends on the correction term of the previous step via the following relation

$$\Delta \boldsymbol{w}_{\boldsymbol{p}}^{\boldsymbol{q}}(new) = a\Delta \boldsymbol{w}_{\boldsymbol{p}}^{\boldsymbol{q}}(old) - \rho \sum_{i=1}^{N} \delta_p^q(i) y^{q-1}(i) \ , \tag{57}$$

  where a is a constant called *momentum factor*, which usually takes values between 0.1 and 0.8.

  It is worth noting that due to the above relation, the whole history of the correction steps is accumulated into $\Delta \boldsymbol{w}_{\boldsymbol{p}}^{\boldsymbol{q}}(old)$.

  It can be shown (see [30]) that the use of the momentum term is equivalent to the increase of the learning rate at regions in the weight space where the gradient of J is close to zero.
- An alternative way to accelerate convergence of the BP algorithm is to adapt the learning rate ρ using heuristic rules (see e.g. [30]). One set of such rules is the following: if $J(t)/J(t-1) < 1 (> c > 1)$ then increase (decrease) ρ by a factor r_i (r_d). Otherwise, if $1 \leq J(t)/J(t-1) \leq c$, we leave ρ unchanged. Typical values for the constants r_i, r_d and c are 1.05, 0.7 and 1.04, respectively. Other acceleration schemes are also possible (see e.g. [6], [12])
- After the completion of the training phase, the parameters of the network are frozen to their converged values and the network is now ready for classification. Adopting the repesentation given in (37), the classification of an unknown vector $\boldsymbol{x}$ into one of the m classes, depends on which output neuron gets the maximum value (this is expected to be close to one and all the others close to zero).
- An important aspect of the type of training discusssed above, i.e., using the representation given in eq. (37) and the cost function given by (38) and (39), is that the outputs $\hat{y}_p$ of the network, *corresponding to the optimal weights*, are estimates, in the least square sense, of $P(\omega_p|\boldsymbol{x})$.
- More complex training algorithms, such as Newton type schemes, which involve the computation of the Hessian matrix of J with respect to $\boldsymbol{w}_{\boldsymbol{p}}^{\boldsymbol{q}}$'s, have also been suggested in order to speed up convergence (see e.g. [16], [2], [14], [3], [21], [4]).
- Different cost function choices are also possible. If for example we assume that y_p take binary values (0 or 1) and they are independent to each other,

and $\hat{y}_p$ are interpreted as estimates of the a posteriori probabilites $P(\omega_p|\boldsymbol{x})$, we may define the so called *cross-entropy function* as

$$J = -\sum_{i=1}^{N}\sum_{p=1}^{k_L}(y_p(i)\ln\hat{y}_p(i) + (1-y_p(i))\ln(1-\hat{y}_p(i))) \ . \tag{58}$$

Computation of the δ terms is independent of the cost function used and is performed in the BP spirit.

– It can be shown that if we adopt the cross-entropy cost function and binary values for $y_p^L(i)$'s, the outputs $\hat{y}_p$ of the network, corresponding to the optimal weights, are also estimates of $P(\omega_p|\boldsymbol{x})$.

7.3 Training Aspects

In the last section we commented on the termination criteria of the BP algorithm. However, a problem that may arise in practice is that of *overtraining*. If this happens, the network learns the peculiarities of the specific training data set very well and fails to capture the underlying general structure of the problem. As a consequence, the behavior of the network is expected to be rather poor on new data, "unknown" to the network.

A way out of the problem of overtraining is to use apart from the training set at hand, an additional data set, known as *test set*, which will be used to measure the performance of the trained neural network. Thus, we split the set of all available data into two sets, the training set and the test set. Then, we plot the evolution of the error on the training and the test set, during the training phase, versus the number of epochs. The point where overtraining becomes noticable is the one where the error on the test set starts to increase [5] (see fig. 10). However, it must be pointed out that this method requires the existence of a large data set, which is not available for all applications.

Avoiding overtraining is one prerequisite for good *generalization*, i.e., for good performance to feature vectors that do not belong to the training set. Another factor affecting the generalization properties is the size of the network. Small networks (with respect to the number of parameters) are often unable to solve the problem at hand, while large networks tend to learn the details of the training data set leading to poor generalization performance.

7.4 Radial Basis Function (RBF) Networks

Up to now we considered only neurons of the structure shown in fig. 2, that implement the function $y = f(\boldsymbol{w}^T\boldsymbol{x})$, where f may be a hard limiter or a sigmoid function. In this section we consider networks whose neurons implement functions of the form

$$y = f(||\boldsymbol{x} - \boldsymbol{c_i}||) \ , \tag{59}$$

[5] Of course, we do not compute the error on the test set at each epoch but after the completion of a predetermined number of epochs.

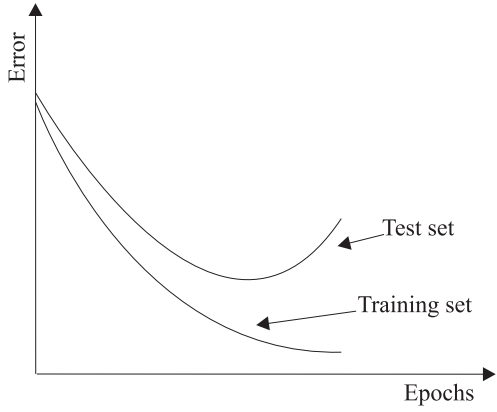

Fig. 10. Plot of the error on the training and the test set versus the number of epochs. The point where the test error starts to increase, indicates the beginning of overtraining

where $\boldsymbol{c_i}$ is a vector in R^l. Typical choices for f are

$$f(\boldsymbol{x}) = \exp(-\frac{1}{2\sigma_i^2}||\boldsymbol{x} - \boldsymbol{c_i}||^2) \ , \tag{60}$$

and

$$f(\boldsymbol{x}) = \frac{\sigma^2}{\sigma^2 + ||\boldsymbol{x} - \boldsymbol{c_i}||^2} \ , \tag{61}$$

with the Gaussian being the most commonly used form in practice.

In words,the output of each neuron is determined by the distance between the input vector $\boldsymbol{x}$ and the vector $\boldsymbol{c_i}$, associated with the corresponding node. Clearly, a neuron of this kind gives the same response for all the points lying on a hypersphere centered at $\boldsymbol{c_i}$. This is the reason why these neurons are called *radial basis function neurons*. Moreover, note that for the above choices of f, the output of the neuron decreases as $||\boldsymbol{x} - \boldsymbol{c_i}||$ increases. Thus the action of such nodes is of a *local nature*.

Radial Basis Function Networks usually consist of two layers of neurons. The first layer consists of k RBF neurons and the second layer of linear output neurons, i.e., neurons as those shown in fig. 2, with f being the identity function. A typical architecture of an RBF network is shown in fig. 11.

Let us now see how an RBF network works through an example. Consider the data set shown in fig. 12(a). The points denoted by open circles belong to class ω_1, while the rest belong to class ω_2. Let us choose $k = 2$ and RBF neurons with centers $\boldsymbol{c_1} = (0,\ 0)$ and $\boldsymbol{c_2} = (1,\ 1)$. The activation function of these neurons is the one given in eq. (60). Also, let $\sigma_i = 1/2$, $i = 1, 2$. Then $\boldsymbol{y} = [\exp(-||\boldsymbol{x} - \boldsymbol{c_1}||^2),\ \exp(-||\boldsymbol{x} - \boldsymbol{c_2}||^2)]^T$ denotes the mapping implemented by the above choice of RBF neurons. Thus, the data points are mapped to a two-dimensional space as follows: the points $(0,\ 0)$ and $(1,\ 1)$ are mapped to the points $(1,\ 0.135)$ and $(0.135,\ 1)$, respectively, the points $(0,\ 1)$ and $(1,\ 0)$ are

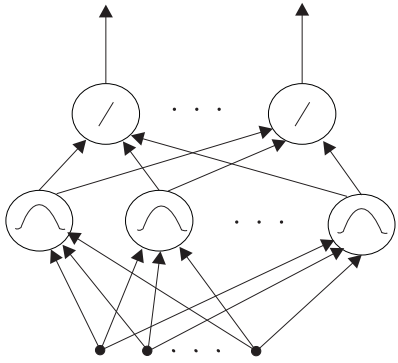

Fig. 11. A typical architecture of an RBF network

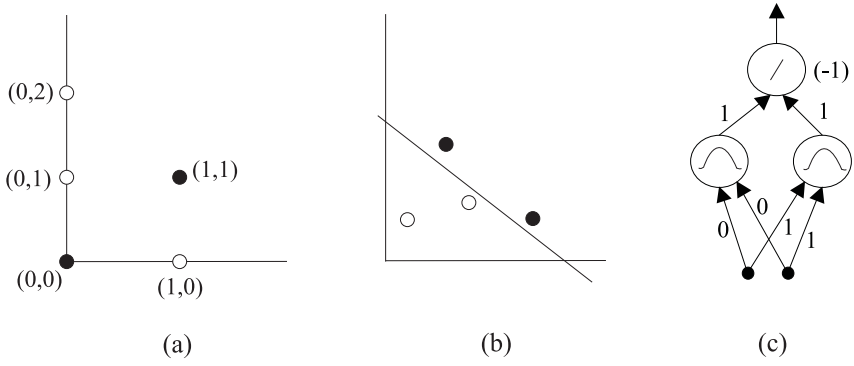

Fig. 12. (a) Set up of the example. The open (filled) circles correspond to class 1 (0). (b) The transformed space. (c) The RBF network implementing the desired partition. The threshold of the output node is shown in parenthesis

both mapped to the point (0.368, 0.368) and, finally, the point (0, 2) is mapped to the point (0.018, 0.136). This mapping is shown in fig. 12(b). Note that the two classes are linearly separable in the transformed space and the straight line

$$g(\boldsymbol{y}) = y_1 + y_2 - 1 = 0 \ , \tag{62}$$

is one possible solution. The corresponding RBF network is shown in fig. 12(c).

Training of the RBF Networks. In the above toy example, we chose arbitrarily the parameters of the RBF network, i.e., the centers $\boldsymbol{c_i}$ and the variances σ_i for each hidden layer neuron and the weights w_j for the output layer neuron. The question that arises now is how can we estimate these parameters in a real world problem. In the sequel we consider the case where we have a single output neuron.

Let $\hat{y}(i)$ be the output of the network and $y(i)$ the desired output, when $\boldsymbol{x}(i)$ is the input and $\phi(.)$ a differentiable function (for example $\phi(z) = z^2$). One way to determine the parameters of the network is to define a cost function of the form

$$J = \sum_{i=1}^{N} \phi(e(i)) \ , \tag{63}$$

where $e(i) = \hat{y}(i) - y(i)$, and to use a gradient descent method to determine the above parameters, i.e.,

$$w_j(t+1) = w_j(t) - \rho_1 \frac{\partial J}{\partial w_j}|_t \ , \quad j = 0, 1, \ldots, k \tag{64}$$

$$\boldsymbol{c_i}(t+1) = \boldsymbol{c_i}(t) - \rho_2 \frac{\partial J}{\partial \boldsymbol{c_i}}|_t \ , \quad j = 1, \ldots, k \tag{65}$$

$$\sigma_i(t+1) = \sigma_i(t) - \rho_3 \frac{\partial J}{\partial \sigma_i}|_t \ , \quad j = 1, \ldots, k \ , \tag{66}$$

where t is the current iteration step. However, the computational complexity of such an approach is excessively high. To overcome this problem other ways have been suggested. One such way is to select the centers corresponding to the hidden layer neurons in a way representative of the distribution of the data set. This can be achieved by assigning a center to every region in the feature space which is dense in feature vectors. These regions may be unravelled using appropriate clustering algorithms (see e.g., [13], [8], [1], [30]). Then, we assign a center to each one of these regions [6] and we compute the variance σ_i of each such region.

After the parameters of the first layer neurons have been determined, we define the following data set

$$S' = \{(\boldsymbol{z_j}, y_j), \ \boldsymbol{z_j} = [f^1(\boldsymbol{x_j}), \ldots, f^k(\boldsymbol{x_j})]^T, \ \boldsymbol{x_j} \in S, \ j = 1, \ldots, N\} \ , \tag{67}$$

where $f^i(\boldsymbol{x_j})$ is the output of the ith first layer neuron of the network. Then, based on the new S', we determine w_j's using the delta rule for one layer networks described above.

Alternative ways for the training of RBF networks have also been suggested (e.g., [29], [24], [34], [15], [30]).

8 Concluding Remarks

In this chapter a number of basic classifiers were considered and discussed, such as the Bayesian classifier, classifers based on Parzen windows and nearest neighbor concepts, the perceptron and neural networks. The goal was to present these methods in a way appropriate for the newcomer in the field.

[6] In most of the cases, the center is computed as the mean of the data vectors in the corresponding region.

References

1. Anderberg M. R., *Cluster Analysis for Applications*, Academic Press, 1973.
2. Barnard E., "Optimization for training neural networks", *IEEE Transactions on Neural Networks*, Vol. 3(2), pp. 232-240, 1992.
3. Battiti R., "First and second order methods for learning: Between steepest descent and Newton's method", *Neural Computation*, Vol. 4, pp. 141-166, 1992.
4. Bishop C. M., *Neural Networks for Pattern Recognition*, Oxford University Press, 1995.
5. Bose N. K., Lianf P., *Neural Network Fundamentals with Graphs, Algorithms and Applications*, McGraw-Hill, 1996.
6. Cichocki A., Unbehauen R., *Neural Networks for Optimization and Signal Processing*, John Wiley, 1993.
7. Devroye L., Gyorfi L., Lugosi G. A., *A Probabilistic Theory of Pattern Recognition*, Springer-Verlag, 1996.
8. Duda R., Hart P. E., *Pattern Classification and Scene Analysis*, John Wiley, 1973.
9. Fahlman S. E., Lebiere C., "The cascade-correlation learning architecture", in *Advances in Neural Information Processing Systems, 2* (Touretzky D. S., ed.), pp. 524-532, Morgan Kaufmann, San Mateo, CA, 1990.
10. Gallant S. I., "Perceptron based learning algorithms", *IEEE Transactions on Neural Networks*, Vol., 1(2), pp. 179-191, 1990.
11. Hassibi B., Stork D. G., Woff G. J., "Optimal brain surgeon and general network pruning", *Proceedings IEEE Conference in Neural Networks*, Vol. 1, pp. 293-299, Sna Francisco, 1993.
12. Jacobs R. A., "Increased rates of convergence through learning rate of adaptation", *Neural Networks*, Vol. 2, pp. 359-366, 1988.
13. Jain A. K., Dubes R. C., *Algorithms for Clustering Data*, Prentice Hall, 1998.
14. Johansson E. M., Dowla F. U., Goodman D.M., "Backpropagation learning for multilayer feedforward neural networks using conjugate gradient method", *International Journal of Neural Systems*, Vol. 2(4), pp. 291-301,1992.
15. Karayiannis N. B., Mi G. W., "Growing radial basis neural networks. Merging supervisedand unsupervised learning with network growth techniques", *IEEE Transactions on Neural Networks*, Vol. 8(6) pp. 1492-1506, 1997.
16. Kramer A. H., Sangiovanni-Vincentelli A., "Efficient parallel learning algorithms for neural networks", in *Advances in Neural Information Processing Systems 3* (Lippmann R. P., Moody J., Touretzky D. S., eds.), pp. 684-692, Morgan Kaufmann, San Mateo, CA, 1991.
17. Koutroumbas K., Pouliakis A., Kalouptsidis N., "Divide and conquer algorithms for constructing neural network architectures", EUCIPCO '98, Rhodes, 1998.
18. Le Cun Y., Denker J. S., Solla S. A., "Optimal brain damage", in *Advances in Neural Information Systems 2* (Touretzky D. S. ed.), pp. 598-605, Morgan Kaufmann, San Mateo, CA, 1990.
19. Mezard M., Nadal J. P., "Learning in feedforward layered networks: The tilling algorithm", *Journal of Physics*, Vol. A 22, pp. 2191-2203,1989.
20. Minsky M. L., Papert S.A., *Perceptrons*, expanded edition, MIT Press, Cambridge, MA, 1988.
21. Palmieri F., Datum M., Shah A., Moiseff A., "Sound localization with a neural network trained with the multiple extended Kalman algorithm", *International JointConference on Neural Networks*, Vol. 1, pp. 125-131, Seattle, 1991.

22. Papoulis A., *Probability, Random Variables and Stochastic Processes*, 3rd ed., McGraw-Hill, 1991.
23. Parzen E., "On the estimation of a probability density function and mode", *Ann. Math.m Stat.*, Vol. 33, pp. 1065-1076, 1962.
24. Platt J., "A resource allocating network for function interpolation", *Neural Computation*, Vol. 3, pp. 213-225, 1991.
25. Refenes A., Chen L., "Analysis of methods for optimal network construction", *University College London Report*, CC30/080:DCN, 1991.
26. Rosenblatt F., "The perceptron: A probabilistic model for information storage and organization in the brain", *Psychological Review*, Vol. 65, pp. 386-408, 1958.
27. Rumelhart D. E., McLelland J. L., *Parallel Distributed Processing*, Cambridge MA: MIT Press, 1986.
28. Russell R., "Pruning algorithms: A survey", *IEEE Transactions on Neural Networks*, Vol. 4(5), pp. 740-747, 1993.
29. Scholkopf B., Sung K.-K., Burges C. J. C., Girosi F., Niyogi P., Poggio T., Vapnik V., "Comparing support vector machnies with Gaussian kernels to RBF classifiers", *IEEE Transactions on Signal Processing*, Vol. 45(11), pp. 2758-2766, 1997.
30. Theodoridis S., Koutroumbas K., *Pattern Recognition*, Academic Press, 1998.
31. Widrow B., Hoff M. E. Jr, "Adaptive switching circuits", *IREWESCON Convention Record*, pp. 96-104, 1960.
32. Widrow B., Lehr M. A., "30 years of adaptive neural networks: Perceptron, madaline and backpropagation", *Proceedings of the IEEE*, Vol. 78(9), pp. 1415-1442, 1990.
33. Werbos P.J., "Beyond regression: New tools for prediction and analysis in the behavioral sciences", Ph.D. Thesis, Harvard University, Cambridge, MA, 1974.
34. Yingwei L., Sundarajan N., Saratihandram P., "Performance evaluation of a sequential minimal RBF network learning algorithm", *IEEE Transactions on Leural Networks*, Vol. 9(2), pp. 308-318, 1998.