UNIT 2

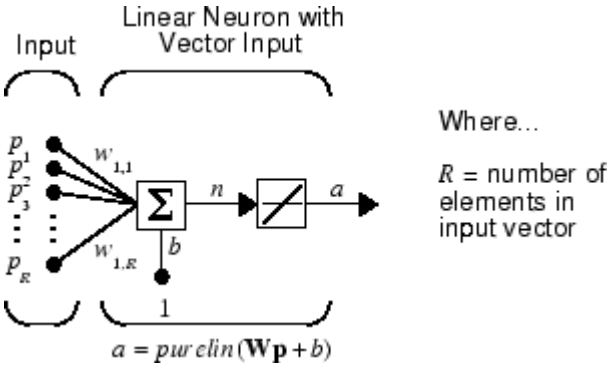
SINGLE LAYER PERCEPTRONS

Adaptive Filtering Problem:

The ADALINE (adaptive linear neuron) networks discussed in this topic are similar to the perceptron, but their transfer function is linear rather than hard-limiting. This allows their outputs to take on any value, whereas the perceptron output is limited to either 0 or 1. Both the ADALINE and the perceptron can solve only linearly separable problems. However, here the LMS (least mean squares) learning rule, which is much more powerful than the perceptron learning rule, is used. The LMS, or Widrow-Hoff, learning rule minimizes the mean square error and thus moves the decision boundaries as far as it can from the training patterns.

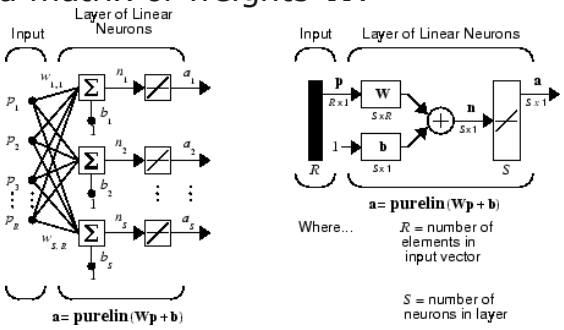
Linear Neuron Model

A linear neuron with *R* inputs is shown below.



Adaptive Linear Network Architecture

The ADALINE network shown below has one layer of *S* neurons connected to *R* inputs through a matrix of weights **W**.

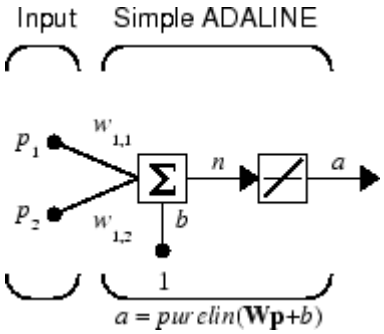


This network is sometimes called a MADALINE for Many ADALINEs. Note that the figure on the right defines an *S*-length output vector **a**.

The Widrow-Hoff rule can only train single-layer linear networks. This is not much of a disadvantage, however, as single-layer linear networks are just as capable as multilayer linear networks. For every multilayer linear network, there is an equivalent single-layer linear network.

Single ADALINE (linearlayer):

Consider a single ADALINE with two inputs. The following figure shows the diagram for this network.



The weight matrix **W** in this case has only one row. The network output is

$$a = purelin(n) = purelin(\mathbf{W}p + b) = \mathbf{W}p + b$$

1. Gradient Descent

Gradient Descent is

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta L(\theta_t)$$

expensive for large d

- **Stochastic Gradie**
  the gradient of a si
  variance in the upda

$$v_t = \beta v_{t-1} + \eta \nabla_\theta L(\theta_t)$$

$$\theta_{t+1} = \theta_t - v_t$$

- **Mini-batch Gradient Descent**: A compromise between batch and SGD, it updates the parameters using a subset of the data. It balances computational efficiency and update stability.

2. Momentum

Momentum is an extension of gradient descent that aims to accelerate convergence by considering the previous updates. It adds a fraction of the previous update to the current update:

Where vt is the velocity and β is the momentum factor (typically set to 0.9).

3. Nesterov Accelerated Gradient (NAG)

NAG is a variant of momentum that improves the convergence speed by making a correction based on an estimated future position of the parameters:

$$v_t = \beta v_{t-1} + \eta \nabla_\theta L(\theta_t - \beta v_{t-1})$$

$$\theta_{t+1} = \theta_t - v_t$$

Linear Least Square Filters/ Least Mean Square Algorithm:

Like the perceptron learning rule, the least mean square error (LMS) algorithm is an example of supervised training, in which the learning rule is provided with a set of examples of desired network behavior.

$$\{\mathbf{p}_1, \mathbf{t}_1\}, \{\mathbf{p}_2, \mathbf{t}_2\}, \ldots \{\mathbf{p}_Q, \mathbf{t}_Q\}$$

Here $\mathbf{p}_q$ is an input to the network, and $\mathbf{t}_q$ is the corresponding target

the network output is compared to the target. The error is calculated as the

the network output. The goal is to minimize the average of the sum of these errors.

$$mse = \frac{1}{Q}\sum_{k=1}^{Q} e(k)^2 = \frac{1}{Q}\sum_{k=1}^{Q} t(k) - a(k))^2$$

The LMS algorithm adjusts the weights and biases of the ADALINE so as to minimize this mean square error.

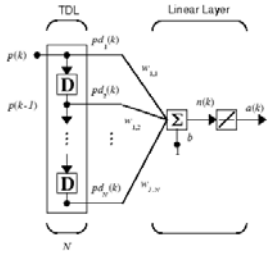
Fortunately, the mean square error performance index for the ADALINE network is a quadratic function. Thus, the performance index will either have one global minimum, a weak minimum, or no minimum, depending on the characteristics of the input vectors. Specifically, the characteristics of the input vectors determine whether or not a unique solution exists.

$$\mathbf{W}(k+1) = \mathbf{W}(k) + 2\alpha\mathbf{e}(k)\mathbf{p}^T(k)$$

$$\mathbf{b}(k+1) = \mathbf{b}(k) + 2\alpha\mathbf{e}(k)$$

Adaptive Filtering (adapt)

The ADALINE network, much like the perceptron, can only solve linearly separable problems. It is, however, one of the most widely used neural

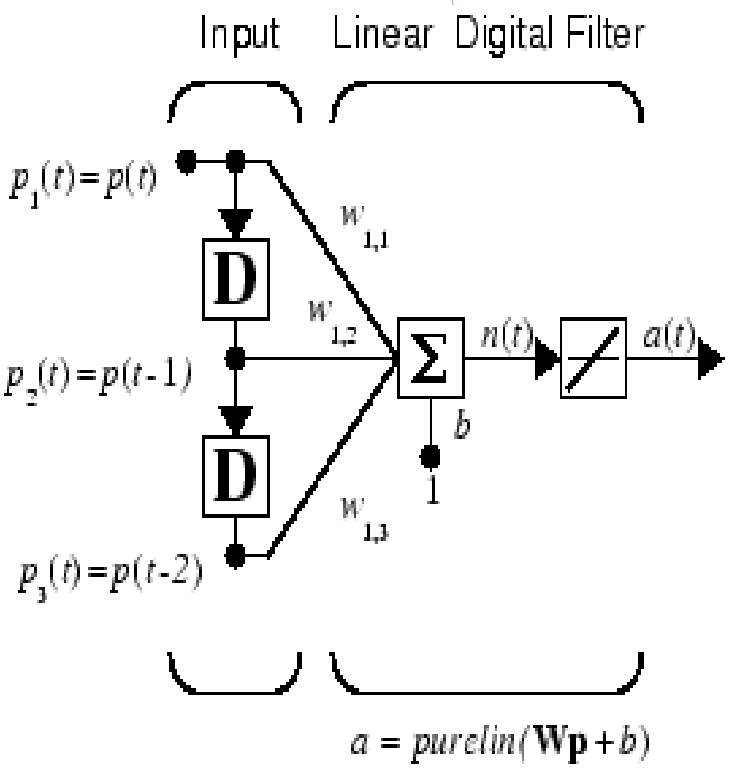


The output of the filter is given by

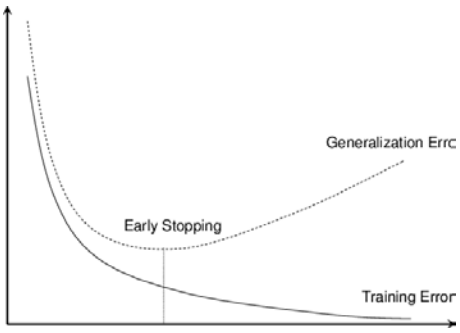
$$a(k) = purelin(\mathbf{W}\mathbf{p} + b) = \sum_{i=1}^{R} w_{1,i}a(k-i+1) + b$$

Adaptive Filter Example

First, define a new linear network using linear layer


Input
Linear Digital Filter
$p_1(t)=p(t)$
$p_2(t)=p(t-1)$
$p_3(t)=p(t-2)$
D
D
$W'_{1,1}$
$W'_{1,2}$
$W'_{1,3}$
Σ
$n(t)$
$a(t)$
b
1
$a = purelin(\mathbf{W}\mathbf{p}+b)$

Learning Curves:



Definition: Learning curves visualize the relationship between a model's performance and the amount of training data or the number of training iterations.

Components:

- **Training Curve:** Shows the model's performance on the training dataset.
- **Validation (or Test) Curve:** Shows the model's performance on a separate validation or test dataset.

Performance Metrics: The performance metric used depends on the task (e g accuracy for classification mean squared error for regression)

Learning curves in neural networks are powerful diagnostic tools that reveal how well a model is learning from the training data and how it generalizes to unseen data. They plot the training and validation (or test) performance as a function of the training set size or the number of training iterations (epochs). Analyzing these curves helps us understand if a model is suffering from issues like overfitting, under fitting, or if more data would be beneficial.

2. **Types of Learning Curves and Interpretations:**

- **Ideal Learning Curve:**
  - The training and validation curves converge to a high performance level.
  - There's a small gap between the two curves.
  - This indicates that the model is learning effectively and generalizing well.
- **High Bias (Under fitting):**
  - Both the training and validation curves plateau at a low performance level.
  - The curves are close together, indicating that the model is not capturing the underlying patterns in the data.
  - **Causes:**
    - The model is too simple (e.g., too few layers, too few

neurons).
- The features are not informative enough.

High Variance (Overfitting):

- The training curve reaches a very high performance level, while the validation curve plateaus at a significantly lower level.
- There's a large gap between the two curves, indicating that the model is memorizing the training data but not generalizing well.
- **Causes:**
  - The model is too complex (e.g., too many layers, neurons).
  - The training data is noisy.
  - Insufficient training data.
  -

Learning rate schedules adjust the learning rate based on predefined

rules or functions, enhancing convergence and performance. Some common methods include:

- **Step Decay**: The learning rate decreases by a specific factor at designated epochs or after a fixed number of iterations.
- **Exponential Decay**: The learning rate is reduced exponentially over time, allowing for a rapid decrease in the initial phases of training.
- **Polynomial Decay**: The learning rate decreases polynomial over time, providing a smoother reduction.

3. Adaptive Learning Rate

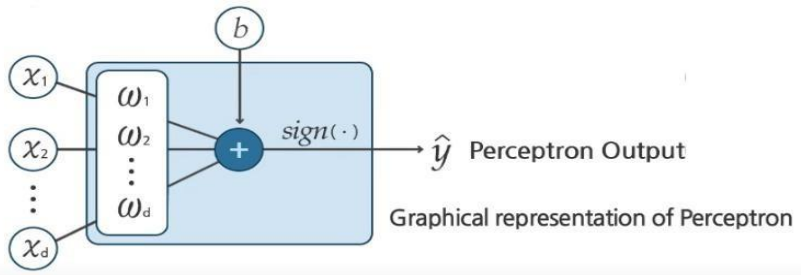
Adaptive learning rates dynamically adjust the learning rate based on the model's performance and the gradient of the cost function. This approach can lead to optimal results by adapting the learning rate depending on the steepness of the cost function curve:

- **AdaGrad**: This method adjusts the learning rate for each parameter individually based on historical gradient information, reducing the learning rate for frequently updated parameters.
- **RMSprop**: A variation of AdaGrad, RMSprop addresses overly aggressive learning rate decay by maintaining a moving average of squared gradients to adapt the learning rate effectively.

- **Adam**: Combining concepts from both AdaGrad and RMSprop, Adam incorporates adaptive learning rates and momentum to accelerate convergence.

PERCEPTION –CONVERGENCE THEOREM:

- The Perceptron Convergence Theorem is a fundamental concept in machine learning, showing how a simple algorithm, the perceptron, can learn to classify items accurately. It's like a basic building block for understanding how computers make decisions, much like our brains handle simple choices. In this article, we will delve into the details of the perceptron convergence theorem, its significance, and its implications for neural network design.
- This theo machine learning t faces in photos to itforward idea, we today's intelligent



Graphical representation of Perceptron

.

Here are the key components and concepts related to a perceptron:

e is

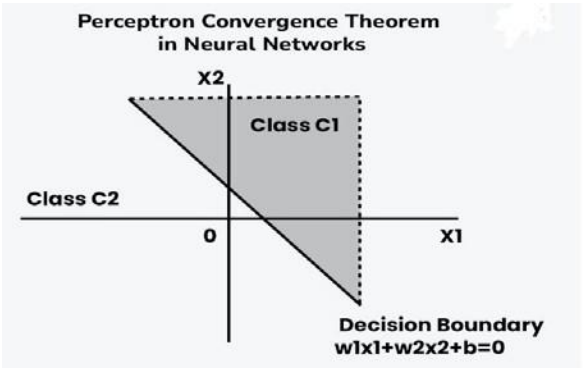
h

$$y_i(w \cdot x_i + b) \geq \gamma$$

where w is the weight vector and b is the bias.

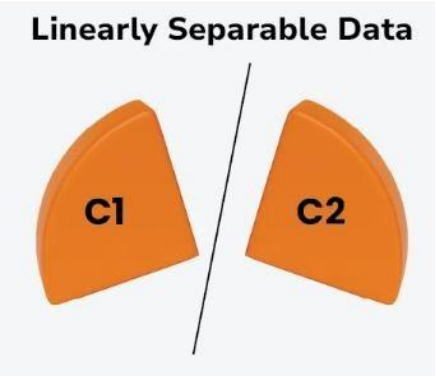
negative exa

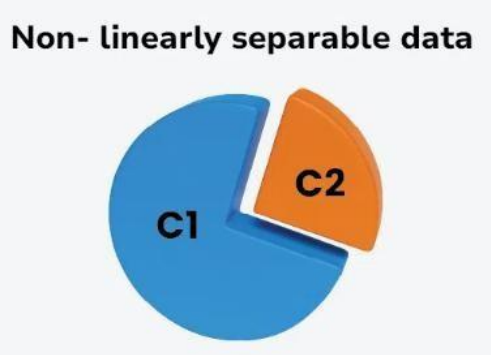
2. **Non-zero M**
training exan



Implications for Theorem

- **Convergence:** For linearly separable data, the perceptron algorithm will always find a hyperplane that correctly classifies the training examples. For example- classes C1 and C2 can be classified by a hyperplane defined as:
- **Separable Data:**] algorithm will not indefinitely.
  1. **Linearly Separa** perfectly divide c two dimensions) higher dimensio that can clearly s the other class w





  2. **Non-linearly sepa** be divided into tw

$$w_i = w_i + \Delta w_i$$

$$b = b + \Delta b$$

Workings of Perceptron Convergence Algorithm

1. **Initialization:** Start by setting the weights and bias to small random values. These weights determine the importance of each input feature.
2. **Input:** Feed an input data point (a vector of features) into the perceptron.
3. **Weighted Sum:** Calculate the weighted sum of the input features plus the bias. This is done by multiplying each input feature by its corresponding weight and adding them all up, along with the bias.
4. **Activation Function:** Apply the activation function to the weighted sum to get the output. In a simple perceptron, this is usually a step function that outputs 1 if the weighted sum is positive, and -1 (or 0) if it's negative.
5. **Prediction:** Compare the perceptron's output (prediction) to the actual label of the input data point (either 1 or -1).
6. **Update Weights:** If the prediction is incorrect, adjust the weights and bias. The weights are updated to reduce the error, moving the decision boundary closer to correctly classifying the input. The update rule is:

where

- $\Delta w_i = \eta * (y - y') * x_i$
- $\Delta b = \eta * (y - y')$
- η is learning rate parameter
- y is the actual label
- y' is the predicted label

7. **Repeat:** Repeat steps 2-6 for all data points in the training set. This process is called an epoch. Multiple epochs are performed until the perceptron correctly classifies all training data points or a maximum number of epochs is reached.

Relation between Perceptron and Bayes Classifier for a Gaussian Environment:

The perceptron has a certain relationship to a classical classifier known as the Bayes

classifier. When the environment is Gaussian, the Bayes classifier reduces to a linear classifier, this is the same form taken by the perceptron.

In the Bayes classifier, we minimize the average risk, denoted by $\mathcal{R}$. For a two-class problem, represented by classes c1 and c2, the average risk is defined by,

$$\mathcal{R} = c_{11}p_1 \int_{\mathcal{H}_1} p_{\mathbf{X}}(\mathbf{x}|\mathcal{C}_1)d\mathbf{x} + c_{22}p_2 \int_{\mathcal{H}_2} p_{\mathbf{X}}(\mathbf{x}|\mathcal{C}_2)d\mathbf{x} + c_{21}p_1 \int_{\mathcal{H}_2} p_{\mathbf{X}}(\mathbf{x}|\mathcal{C}_1)d\mathbf{x} + c_{12}p_2 \int_{\mathcal{H}_1} p_{\mathbf{X}}(\mathbf{x}|\mathcal{C}_2)d\mathbf{x} \quad (1.23)$$

Where

p_i = *prior probability* that the observation vector **x** (representing a realization of the random vector **X**) corresponds to an object in class $\mathcal{C}_1$, with $i = 1, 2$, and $p_1 + p_2 = 1$

c_{ij} = cost of deciding in favor of class $\mathcal{C}_i$ represented by subspace $\mathcal{X}_i$ when

2. All values of the observation vector **x** for which the integrand is positive should be excluded from subset $\mathcal{X}_1$ (i.e., assigned to class $\mathcal{C}_2$), for the integral would then make a positive contribution to the risk $\mathcal{R}$.
3. Values of **x** for which the integrand is zero have no effect on the average risk $\mathcal{R}$ and may be assingned arbitrarily. We shall assume that these points are assigned to subset $\mathcal{X}_2$ (i.e., class $\mathcal{C}_2$).

On this basis, we may now formulate the Bayes classifier to minimize the Risk if,

$$p_1(c_{21} - c_{11}) f_{\mathbf{X}}(\mathbf{x}|\mathscr{C}_1) > p_2(c_{12} - c_{22}) f_{\mathbf{X}}(\mathbf{x}|\mathscr{C}_2)$$

$$\frac{p_{\mathbf{X}}(\mathbf{x}|\mathscr{C}_1)}{\quad} \quad \frac{p_2(c_{12} - c_{22})}{\quad}$$

The covariance matrix $\mathbf{C}$ is nondiagonal, whic
classes $\mathscr{C}_1$ and $\mathscr{C}_2$ are *correlated*. It is assumed t
matrix $\mathbf{C}^{-1}$ exists.

With this background, we may express the conditional probability density function of $\mathbf{X}$ as the multivariate Gaussian distribution

$$p_{\mathbf{X}}(\mathbf{x}|\mathscr{C}_i) = \frac{1}{(2\pi)^{m/2}(\det(\mathbf{C}))^{1/2}} \exp\left(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu}_i)^T \mathbf{C}^{-1}(\mathbf{x}-\boldsymbol{\mu}_i)\right), \quad i = 1, 2 \qquad (1.30)$$

where m is the dimensionality of the observation vector $\mathbf{x}$.

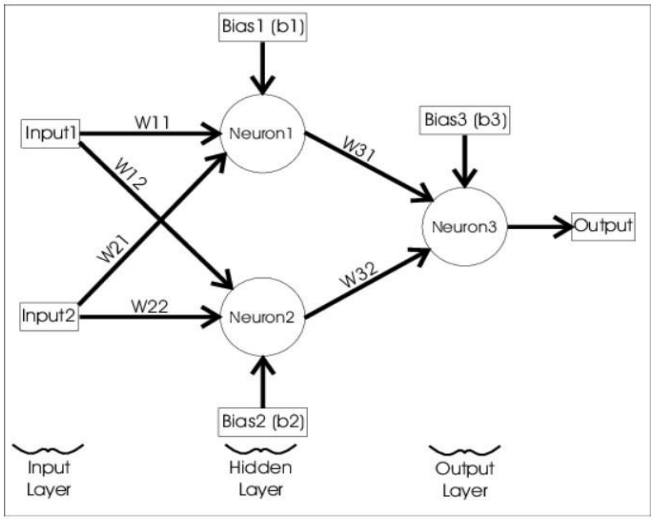
We further assume the following:

vector x to class C1. Otherwise, assign it to class C2. The operation of the Bayes classifier for the Gaussian environment described herein is analogous to that of the perceptron in that they are both linear classifiers.

MULTI LAYER PERCEPTRON

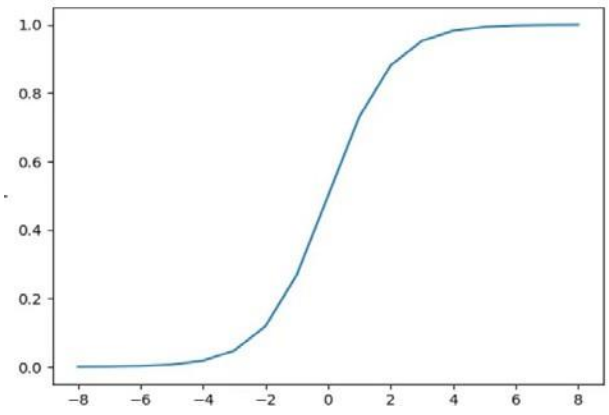
Back Propagation Algorithm XOR Problem

The neural network needs to produce two different decision planes to linearly separate the input data based on the output patterns. This is achieved by using the concept of *hidden layers*. The neural network will consist of one input layer with two nodes (X1,X2); one hidden layer with



i.e. ∑wi xi

3. The outp
which is as
is a continu

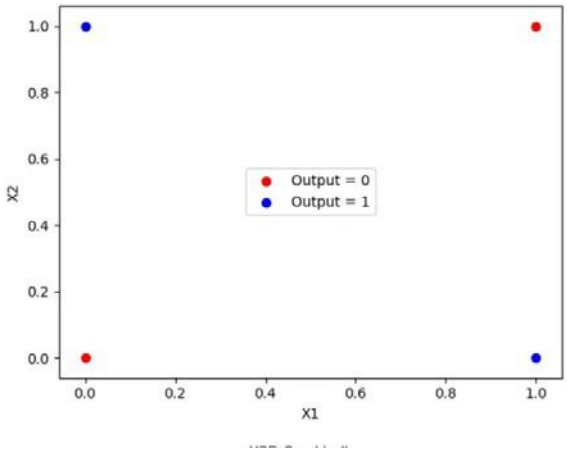


Implementing logic gates using neural networks help understand the mathematical computation by which a neural network processes its inputs to arrive at a certain output. This neural network will deal with the XOR logic problem. An XOR (exclusive OR gate) is a digital logic gate that gives a true output only when both its inputs differ from each other. The truth table for an XOR gate is shown below:

X1	X2	Y
0	0	0
0	1	1
1	0	1
1	1	0

Truth Table for XOR

The goal of the neural network is to classify the input patterns according to the above truth table. If the input
patterns are plotted according to their outputs, it is seen that these points are not linearly

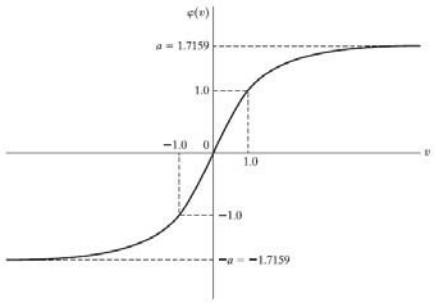


It is often said that the design of a neural network using the back-propagation algorithm is more of an art than a science, in the sense that many of the factors involved in the design are the results of one's own personal experience. There is some truth in this statement. Nevertheless, there are methods that will significantly improve the back-propagation algorithm's performance, as described here:

1. *Stochastic versus batch update.* As mentioned previously, the stochastic (sequential) mode of back-propagation learning (involving pattern-by-pattern updating) is computationally faster than the batch mode. This is especially true when the

training data sample is large and highly redundant. (Highly redundant data pose computational problems for the estimation of the Jacobian required for the batch update.)

2. *Maximizing information content.* As a general rule, every training example pre-

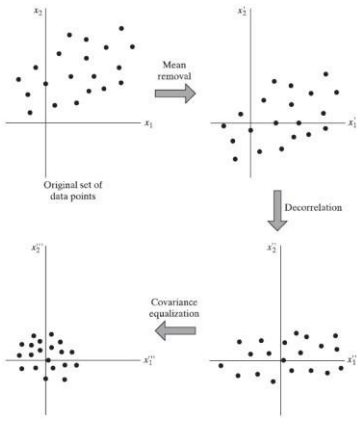


4. *Target values.* It is important that the target values (desired response) be chosen within the range of the sigmoid activation function. More specifically, the desired response d_j for neuron j in the output layer of the multilayer perceptron should be *offset* by some amount ε away from the limiting value of the sigmoid activation function, depending on whether the limiting value is positive or negative. Otherwise, the back-propagation algorithm tends to drive the free parameters of the network to infinity and thereby slow down the learning process by driving the hidden neurons into saturation. To be specific, consider the hyperbolic tangent function of Fig. 4.10. For the limiting value $+a$, we set

$$d_j = a - \varepsilon$$

and for the limiting value of $-a$, we set

$$d_j = -a + \varepsilon$$



In theory, for an M-class *classification problem* in which the union of the M distinct classes forms the entire input space, we need a total of M outputs to represent all possible classification decisions, as depicted in Fig. 4.12. In this figure the vector $\mathbf{x}_j$ denotes the jth *prototype* (i.e., unique sample) of an m-dimensional random vector $\mathbf{x}$ to be classified by a multilayer perceptron. The kth of M possible classes to which $\mathbf{x}$ can belong is denoted by $\mathscr{C}_k$. Let y_{kj} be the kth output of the network produced in response to the prototype $\mathbf{x}_j$, as shown by

$$y_{k,j} = F_k(\mathbf{x}_j), \qquad k = 1, 2, \ldots, M$$

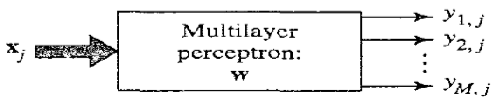


FIGURE 4.12 Block diagram of a pattern classifier.

where the function $F_k(\cdot)$ defines the mapping learned by the network from the input to the kth output. For convenience of presentation, let

$$\mathbf{y}_j = [y_{1,j}, y_{2,j}, \ldots, y_{M,j}]^T$$

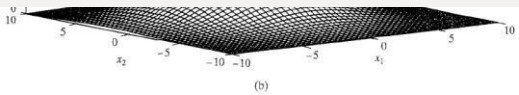
$$[F_1(\cdot), F_2(\cdot), \ldots, F_M(\cdot)]^T$$

$$\begin{bmatrix} \vdots \\ 0 \end{bmatrix}$$

It is tempting to suppose that a multilayer perceptron classifier trained with the back-propagation algorithm on a finite set of independently and identically distributed (i.i.d.) examples may lead to an asymptotic approximation of the underlying *a posteriori* class probabilities. This property may be justified on the following grounds (White, 1989a; Richard and Lippmann, 1991):

Computer Experiment: Feature Detection:

In this section we use a computer experiment to illustrate the learning behavior of a multilayer perceptron used as a pattern classifier. The objective of the experiment is to distinguish between two classes of "overlapping," two-dimensional, Gaussian-distributed patterns labeled 1 and 2. Let $\mathscr{C}_1$ and $\mathscr{C}_2$ denote the set of events for which a random vec-



(a) Probability density function $f_x(\mathbf{x}|\mathscr{C}_1)$; (b) Probability density function $f_x(\mathbf{x}|\mathscr{C}_2)$.