Chapter 5

ARTIFICIAL NEURAL NETWORKS

Javid Taheri and Albert Y. Zomaya
The University of Sydney

Artificial Neural Networks have been one of the most active areas of research in computer science during the last fifty years, with periods of intense activity interrupted by episodes of hiatus [1]. The premise for the evolution of the theory of artificial neural networks stems from the basic neurological structure of living organisms. Cells are the most important constituent of these life forms. These cells are connected by *synapses*, which are the links that carry messages between cells. In fact, by using synapses to carry the pulses, cells can activate each other with different threshold values to form a decision or memorize an event.

Inspired by this simplistic vision of how messages are transferred between cells, scientists invented a new computational approach, which became popularly known as Artificial Neural Networks (or Neural Networks for short), and used it extensively to target a wide range of problems in many application areas. Although the shape or configurations of different neural networks may look different at the first glance, the networks themselves are almost similar in structure.

A neural network consists of *cells* and *links*. Cells are the computational part of the network that perform reasoning and generate activation signals for other cells, while links connect the different cells and enable messages to flow among cells. Each link is usually a one-directional connection with a weight that affects the carried message in a certain way. This means that a link receives a value (message) from an input cell, multiplies it by a given weight, and then passes it to the output cell.

In its simplest form, a cell can have three states (of activation), namely, +1 (TRUE), 0, and −1 (FALSE), to represent three states: activation, unknown, and deactivation. Figure 5.1 shows a simple network with two inputs and one output. Table 5.1 gives the output for all possible inputs in such a network. As can be seen, this network simply separates the sample space into two completely individual subspaces.

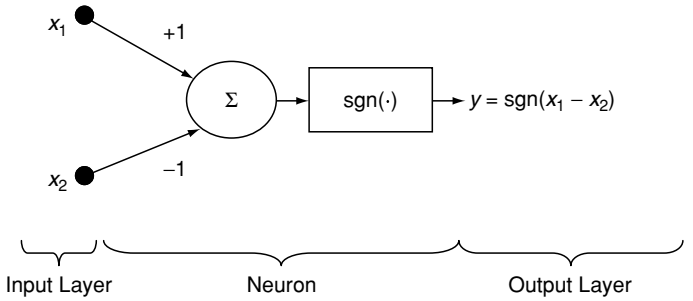

Figure 5.1. *A neural network with two inputs and one output*

Table 5.1 Truth table of the network in Figure 5.1

	x	−1	0	+1
y				
−1		0	−1	−1
0		1	0	−1
+1		1	1	0

1 A GENERIC NEURAL NETWORK

Figure 5.1 shows a simple instant of a neural network. Cells (or neurons) can have more sophisticated structure that can handle complex problems. These neurons can be linear or nonlinear functions with or without biases. Figure 5.2 shows two simple neurons that can have biased and unbiased states.

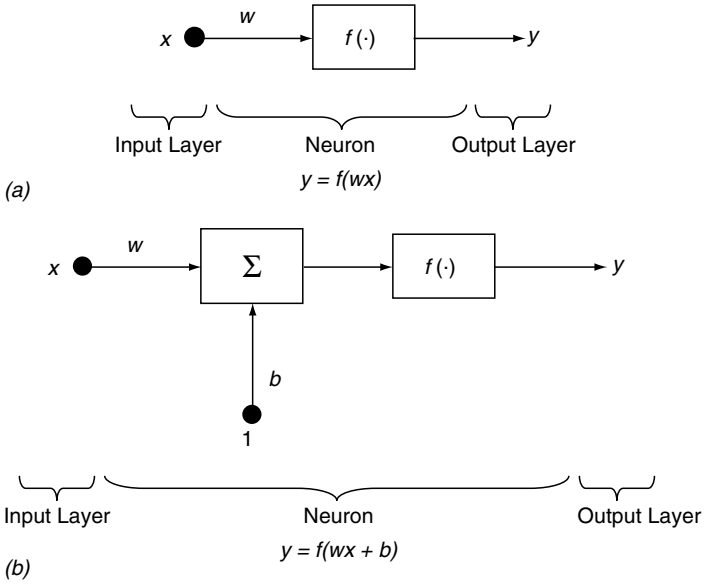

Figure 5.2. *(a) Unbiased and (b) biased structures of a neural network*

1.1 Single-Layer Perceptron

The single-layer perceptron is one of the simplest classes of neural networks [1]. The general overview of this network is shown in Figure 5.3, where the network has n inputs and generates only one output. The input of the function $f(\cdot)$ is actually a linear combination of the network's inputs. In this case, W is a vector of neuron weights, X is the input vector, and y is the only output of the network. These inputs are defined as follows:

$$y = f(W \cdot X + b)$$
$$W = (w_1 \quad w_2 \quad \ldots \quad w_n)$$
$$X = (x_1 \quad x_2 \quad \ldots \quad x_n)^T$$

The above-mentioned basic structure can be extended to produce networks with more than one output. In this case, each output has its own weights and is completely uncorrelated to the other outputs. Figure 5.4 shows such a network, with the following formulas:

$$Y = F(W \cdot X + B)$$

$$W = \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,n} \\ w_{2,1} & & & \\ \cdots & & & \\ w_{m,1} & & \cdots & w_{m,n} \end{bmatrix}$$

$$X = (x_1 \quad x_2 \quad \ldots \quad x_n)^T$$
$$Y = (y_1 \quad y_2 \quad \ldots \quad y_m)^T$$
$$B = (b_1 \quad b_2 \quad \ldots \quad b_m)^T$$
$$F(\cdot) = (f_1(\cdot) \quad f_2(\cdot) \quad \ldots \quad f_m(\cdot))^T$$

where
n: number of inputs
m: number of outputs
W: weighing matrix
X: input vector
Y: output vector
$F(\cdot)$: array of output functions

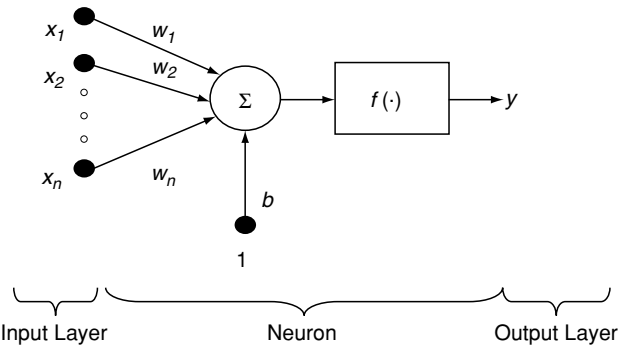

Figure 5.3. *A single-output (single-layer) perceptron*

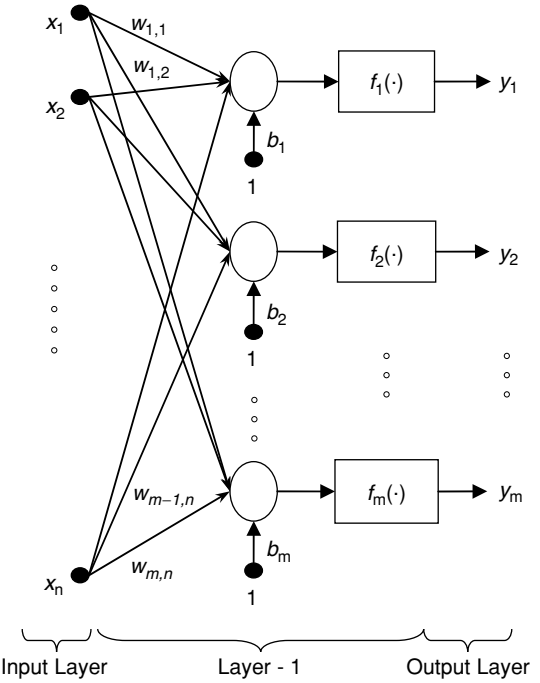

Figure 5.4. *A multioutput single-layer perceptron*

2 MULTILAYER PERCEPTRON

A multilayer perceptron can be simply constructed by concatenating several single-layer perceptron networks. Figure 5.5 shows the basic structure of such a network, which has the following parameters [1]:

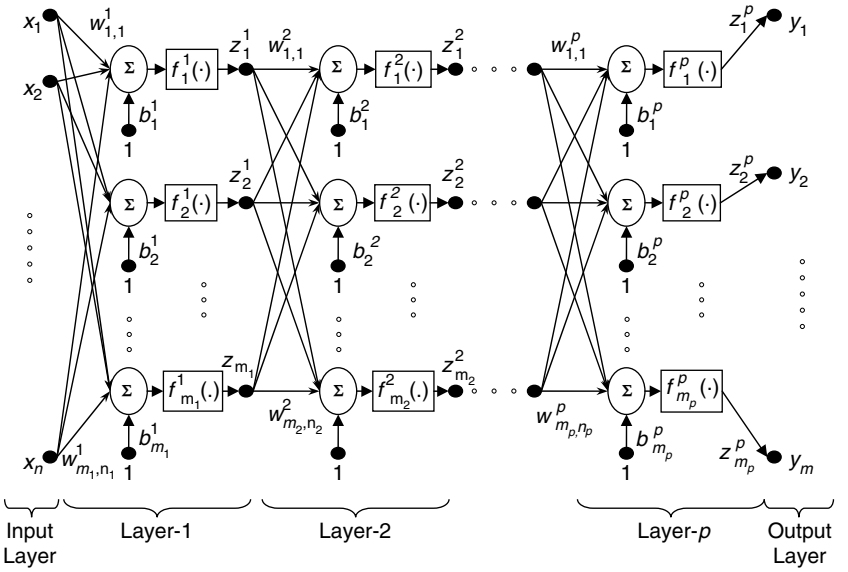

Figure 5.5. *The basic structure of a multilayer neural network*

X: input vector
Y: output vector
n: number of inputs
m: number of outputs
p: total number of layers in the network

while
m_i: number of outputs for the ith layer
n_i: number of inputs for the ith layer

Note that in this network, every internal layer of the network can have its own number of inputs and outputs only by considering the concatenation rule, i.e. $n_i = m_{i-1}$. The output of the first layer is calculated as follows:

$$Z^1 = F^1(W^1 \cdot X + B^1)$$

$$W^1 = \begin{bmatrix} w^1_{1,1} & w^1_{1,2} & \cdots & w^1_{1,n} \\ w^1_{2,1} & & & \\ \cdots & & & \\ w^1_{m_1,1} & & \cdots & w^1_{m_1,n} \end{bmatrix}$$

$$X = (x_1 \quad x_2 \quad \ldots \quad x_n)^T$$
$$B^1 = (b^1_1 \quad b^1_2 \quad \ldots \quad b^1_{m_1})^T$$
$$Z^1 = (z^1_1 \quad z^1_2 \quad \ldots \quad z^1_{m_1})^T$$
$$F^1(\cdot) = (f^1_1(\cdot) \quad f^1_2(\cdot) \quad \ldots \quad f^1_{m_1}(\cdot))^T$$

As a result, the output of the second layer would be

$$Z^2 = F^2\,(W^2 \cdot Z^1 + B^2)$$

$$W^2 = \begin{bmatrix} w^2_{1,1} & w^2_{1,2} & \cdots & w^2_{1,n} \\ w^2_{2,1} & & & \\ \cdots & & & \\ w^2_{m_2,1} & & \cdots & w^2_{m_2,m_1} \end{bmatrix}$$

$$B^2 = (b^2_1 \quad b^2_2 \quad \ldots \quad b^2_{m_2})^T$$
$$Z^2 = (z^2_1 \quad z^2_2 \quad \ldots \quad z^2_{m_2})^T$$
$$F^2(\cdot) = \left(f^2_1(\cdot) \quad f^2_2(\cdot) \quad \ldots \quad f_{m_2}{}^2(\cdot) \right)^T$$

Finally, the last-layer formulation can be given as

$$Y = Z^p = F^p(W^p \cdot Z^{p-1} + B^p)$$

$$W^p = \begin{bmatrix} w^p_{1,1} & w^p_{1,2} & \cdots & w^p_{1,n} \\ w^p_{2,1} & & & \\ \cdots & & & \\ w^p_{m_1,1} & & \cdots & w^p_{m_p,m_{p-1}} \end{bmatrix}$$

$$B^p = (b^p_1 \quad b^p_2 \quad \ldots \quad b^p_{m_p})^T$$

$$Z^p = (z_1^p \quad z_2^p \quad \dots \quad z_{m_p}^p)^T$$
$$F^p(\cdot) = \left(f_1^p(\cdot) \quad f_2^p(\cdot) \quad \dots \quad f_{m_p}^p(\cdot) \right)^T$$

Note that, in such networks, the complexity of the network rises quickly based on the number of layers. Practically experienced, each multilayer perceptron can be evaluated by a single-layer perceptron with a comparatively huge number of nodes.

2.1 Function Representation

Two of the most popular uses of neural networks is to represent (or approximate) functions and model systems. Basically, a neural network would be used to imitate the behavior of a function by generating relatively similar outputs in comparison with the real system (or function) over the same range of inputs.

2.1.1 Boolean Functions

Neural networks were first used to model simple Boolean functions. For example, Figure 5.6 shows how a neural network can be used to model an AND operator, while Figure 5.7 gives the truth table. Note that "1" stands for "TRUE" while "–1" represents a "FALSE" value. The network in Figure 5.6 actually simulates a linear (function) separator, which simply divides the decision space into two parts.

2.1.2 Real-Value Functions

In real-value functions, the network weights must be set so that the network can generate continues outputs of a real system. The generated network is also intended to act as an extrapolator that can generate output data for inputs that are different from the training set.

To clarify this, assume that the data set given in Table 5.2 is produced by a real-world phenomenon (or system). The idea here is for a neural network (Figure 5.8) to regenerate the same data and also be able to produce other values for sets of unforeseen inputs (i.e., extrapolate). Figure 5.9 shows graphically the output of both the system and the neural model.

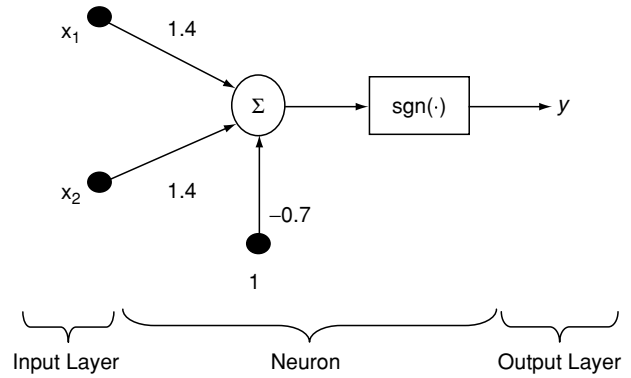

Figure 5.6. *A neural network that implements the logical AND operator*

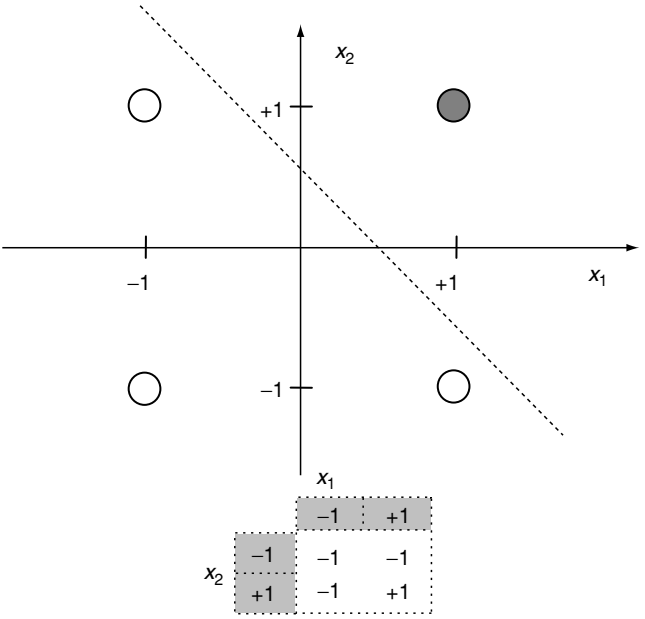

Figure 5.7. *Representation for the network in Figure 5.6*

Table 5.2. Truth table for an instance of a real value function

X_1 / X_2	−3	0	3
−3	0.97	0.43	−3.49
0	1.85	0.28	−1.18
3	4.26	1.0	−1.36

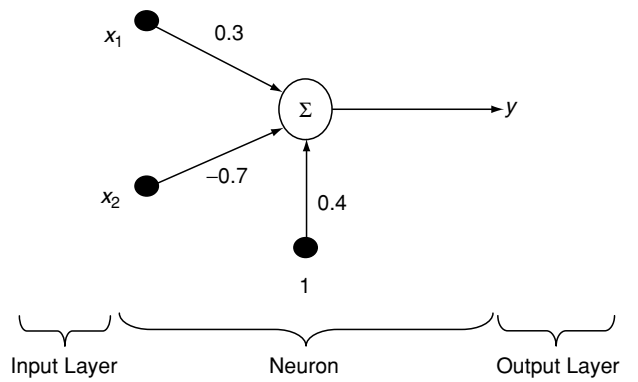

Figure 5.8. *A neural network that implements a simple real function*

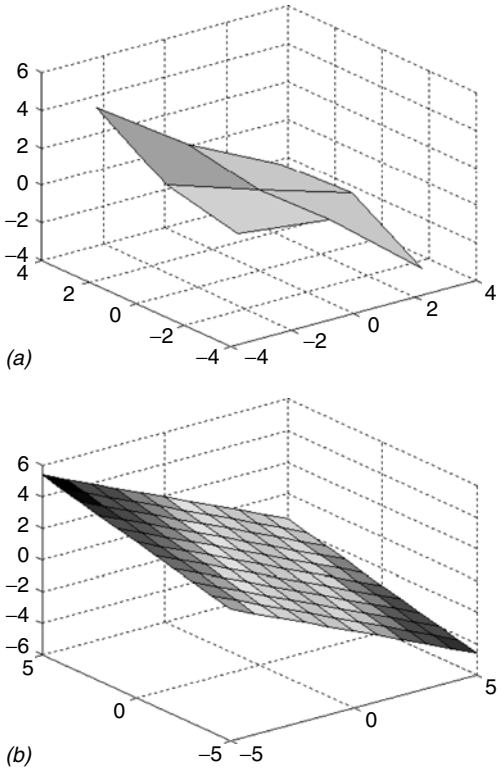

Figure 5.9. *The real values (a) and its corresponding neural model (b)*

3 LEARNING SINGLE-LAYER MODELS

The main, and most important, application of all neural networks is their ability to model a process or learn a behavior of a system. Toward this end, several algorithms have been proposed to train the adjustable parameters of a network (i.e., *W*). Basically, training a neural network to adjust the *Ws* is categorized into two different classes: supervised and unsupervised [2–6].

3.1 Supervised Learning

The main purpose of supervised learning is to "teach" a network to copy the behavior of a system or a function. In this case, there is always a need to have a "training" data set. The network topology and the algorithm with which the network is trained are highly interrelated. In general, a topology of the network is chosen first and then an appropriate training algorithm is used to tune the weights (*W*) [7, 8].

3.1.1. Perceptron Learning

As mentioned earlier, the perceptron is the most basic form of neural networks. Essentially, this network tries to classify input data by mapping it onto a

plane (Figures 5.3 and 5.4). In this approach, to simplify the algorithm, suppose that the network's input is restricted to {+ 1,0, – 1}, while the output can be {+ 1, –1}. The aim of the algorithm is to find an appropriate set of weights, W, by sampling a training set, T, that will capture the mapping that associates each input to an output, i.e.,

$W = (w_0\ w_1 \ldots w_n)$
$T = \{(R^1, S^1), (R^2, S^2), \ldots, (R^L, S^L)\}$

where n is the number of inputs, R^i is the ith input datum, $S^{i'}$ represents the appropriate output for the ith pattern, and L is the size of the training data set. Note that, for the above vector W, w_n is used to adjust the bias in the values of the weights. Perceptron Learning can be summarized as follows:

Step 1: Set all elements of the weighting vector to zero, i.e., $W = (0\ 0 \ldots\ 0)$.
Step 2: Select the training pattern randomly, the kth datum.
Step 3: IF the current W hasn't been classified correctly, i.e., $W.R^k \neq S^k$, THEN modify the weighing vector as follows: $W \leftarrow W + R^k\ S^k$.
Step 4: Repeat steps 1–3 until all data are classified correctly.

The following example is used to demonstrate how this network functions. Assume a network with two inputs and one output used to classify the data of Table 5.3. The different iterations that the network will undergo are as follows:

Iteration	Current W	Choice	OK ?	Action
1	<0 0 0>	T5	NO	W=W–T5
2	<–2 3 –1>	T6	YES	
3	<–2 3 –1>	T4	YES	
4	<–2 3 –1>	T2	YES	
5	<–2 3 –1>	T1	NO	W=W+T1
6	<0 3 0>	T2	YES	
7	<0 3 0>	T1	NO	W=W+T1
	<2 3 1>	Works for all training data; algorithm terminates.		

In this case, the final answer would be $W = [2\ 3\ 1]$. Figure 5.10 shows the network after it converges to the previous answer, while Figure 5.11 graphically shows the output of the network.

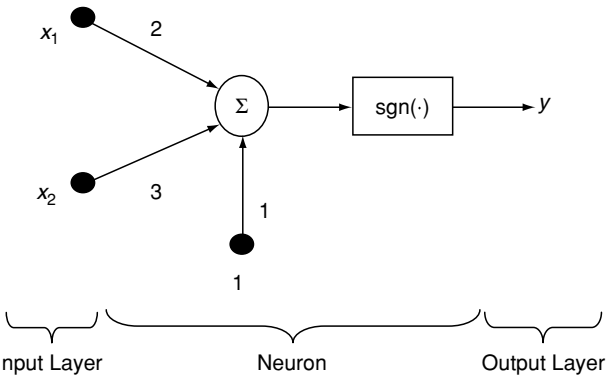

Figure 5.10. *A perceptron neural network for the data in Table 5.3*

Table 5.3. A sample training set for a perceptron

Name	Input		Output
	X_1	X_2	Y
T_1	2	0	1
T_2	1	2	1
T_3	3	4	1
T_4	−3	−2	−1
T_5	2	−3	−1
T_6	−1	−1	−1

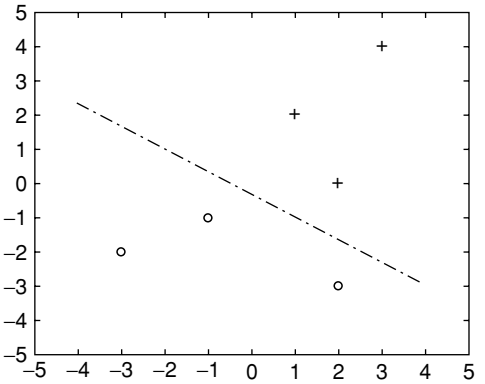

Figure 5.11. *The output of the network of Figure 5.10*

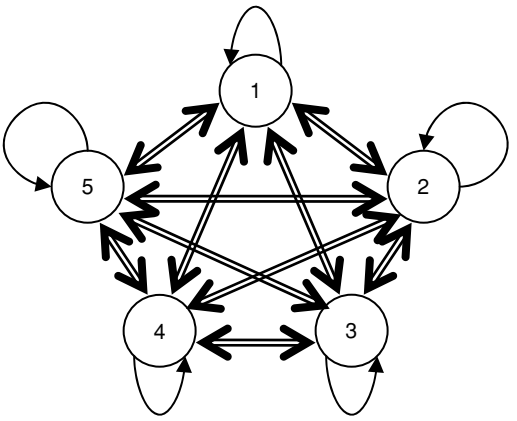

Figure 5.12. *A sample linear auto-associative network with five nodes*

3.2 Linear Auto-Associative Learning

An auto-associative network is another type of network that has some type of memory. In this network, the input and output nodes are basically the same. Hence, when a datum enters the network, it passes through the nodes and converges to the closest memorized datum, which was previously stored in the

network during the training process [1]. Figure 5.12 shows an instance of such network with five nodes.

It is worth noting that the weighing matrix of such network is not symmetrical. That is, $w_{i,j}$, which relate node i to node j, may have different values than $w_{j,i}$. The main key of designing such a network is in the training data set. In this case, the assumption is to have orthogonal or approximately orthogonal training data, i.e.,

$$\langle T_i, T_j \rangle \approx \begin{cases} 0 & i \neq j \\ 1 & i = j \end{cases}$$

where T_i is the ith training data and $\langle \cdot \rangle$ is the inner product of two vectors. Based on the above, the weight matrix for this network is calculated as follows, where $\otimes$ stands for outer product of two vectors:

$$W = \sum_{i=1}^{N} T_i \otimes T_i$$

As can be seen, the main advantage of this network is in its one-shot learning process, accomplished by considering orthogonal data. Note that, even if the input data are not orthogonal in the first place, they can be transferred to a new space by a simple transfer function.

To demonstrate the use of this network, assume the three-node network of Figure 5.13. In this network, the inputs and outputs of the network are basically same. Also, assume that the data in Table 5.4 need to be stored in the network.

In this case, the training data set is approximately orthogonal, i.e.,

$\langle T_1, T_1 \rangle = 0.9902$
$\langle T_2, T_2 \rangle = 1.0025$
$\langle T_1, T_2 \rangle = -0.0066$

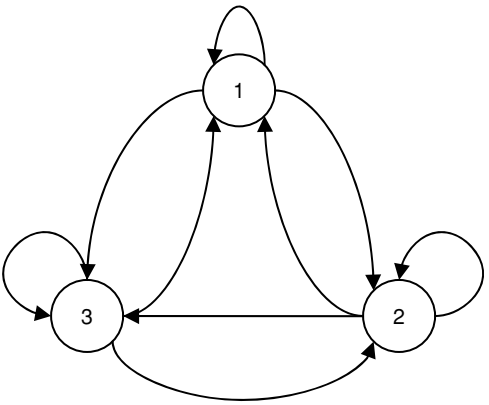

Figure 5.13. *An auto-associative network with three nodes*

Table 5.4. Training data for a sample auto associate network

T_1	<–0.29 0.90 0.31>
T_2	<0.94 0.33 –0.1>

Therefore, the weight matrix would be calculated as follows:

$$W = T_1 \otimes T_1 + T_2 \otimes T_2 =$$

$$\begin{bmatrix} 0.0841 & -0.2610 & -0.0899 \\ -0.2610 & 0.8100 & 0.2790 \\ -0.0899 & 0.2790 & 0.0961 \end{bmatrix} + \begin{bmatrix} 0.8836 & 0.3102 & -0.0940 \\ 0.3102 & 0.1089 & -0.0330 \\ -0.0940 & -0.0330 & 0.0100 \end{bmatrix}$$

$$= \begin{bmatrix} 0.9677 & 0.0492 & -0.1839 \\ 0.0492 & 0.9189 & 0.2460 \\ -0.1839 & 0.2460 & 0.1061 \end{bmatrix}$$

To show how the above network functions, assume that the following data, which are not part of the training data set, are fed into the network: $T = < 0.8 \quad 0.5 \quad 0.33 >$. Figure 5.14 shows how this network converges to an output. Figures 5.14a and 5.14b show the $\| T - T_1 \|$ and $\| T - T_2 \|$ cases, respectively.

3.2.1 Iterative Learning

Iterative learning is another approach that can be used to train a network. In this case, the network's weights are modified smoothly, in contrast to the one-shot learning algorithms. In general, network weights are set to some arbitrary values first, and then training data are fed to the network. In this case, in each training cycle, network weights are modified smoothly. Then the training process proceeds until it achieves an acceptable level of acceptance for the network. However, the training data could be selected sequentially or randomly in each training cycle [9–11].

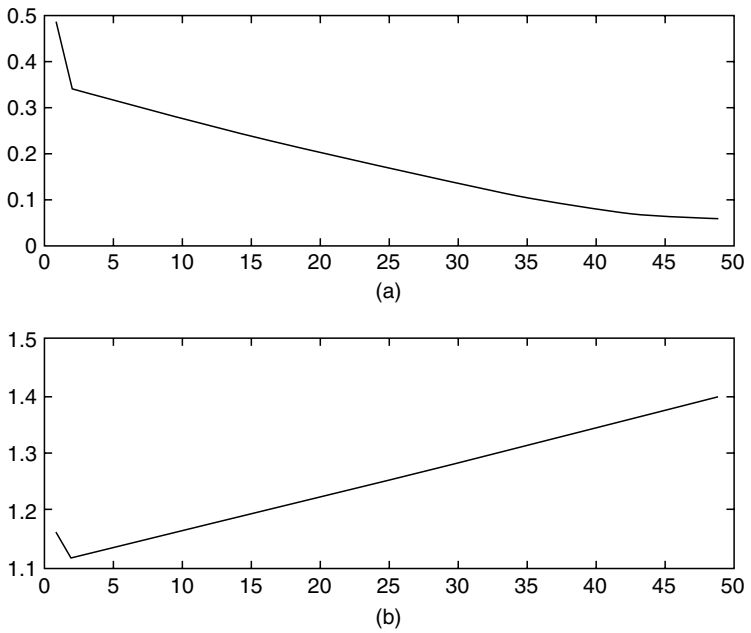

Figure 5.14. *Convergence of the network in Figure 5.13 for the new data set*

3.2.2 Hopfield's Model

A Hopfield neural network is another example of an auto-associative network [1, 12–14]. There are two main differences between this network and the previously described auto-associative network. In this network, self-connection is not allowed, i.e., $w_{i,i} = 0$ for all nodes. Also, inputs and outputs are either 0 or 1. This means that the node activation is recomputed after each cycle of convergence as follows:

$$S_i = \sum_{j=1}^{N} w_{i,j} \cdot u_j(t) \tag{1}$$

$$u'_j = \begin{cases} 1 & if\ S_i \geq 0 \\ 0 & if\ S_i < 0 \end{cases} \tag{2}$$

After feeding a datum into the network, in each convergence cycle, the nodes are selected by a uniform random function, the inputs are used to calculate (1), and then (2) follows to generate the output. This procedure is continued until the network converges.

The proof of convergence for this network uses the notion of *energy*. This means that an energy value is assigned to each state of the network, and through the different iterations of the algorithm, the overall energy is decreased until it reaches a steady state. To show the workings of this network, one can train this network to learn the data set given in Table 5.5. In this case, the weights matrix would be as follows:

$$W = \sum_{i=1}^{N} T_i \otimes T_i = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} + \begin{bmatrix} 1 & 1 & -1 & -1 \\ 1 & 1 & -1 & -1 \\ -1 & -1 & 1 & 1 \\ -1 & -1 & 1 & 1 \end{bmatrix}$$

$$+ \begin{bmatrix} 1 & -1 & -1 & -1 \\ -1 & 1 & 1 & 1 \\ -1 & 1 & 1 & 1 \\ -1 & 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 3 & 1 & -1 & -1 \\ 1 & 3 & 1 & 1 \\ -1 & 1 & 3 & 3 \\ -1 & 1 & 3 & 3 \end{bmatrix} \rightarrow \begin{bmatrix} 0 & 1 & -1 & -1 \\ 1 & 0 & 1 & 1 \\ -1 & 1 & 0 & 3 \\ -1 & 1 & 3 & 0 \end{bmatrix}$$

Now, suppose that the following input is applied to the network:

$$T = \begin{bmatrix} -1 \\ 1 \\ -1 \\ -1 \end{bmatrix}$$

Table 5.5. Training data for a Hopfield neural network

T_1	<1 1 1 1>
T_2	<–1 –1 1 1>
T_3	<1 –1 –1 –1>

In this case, the network output would be

$$W \cdot T = \begin{bmatrix} 3 \\ -3 \\ -1 \\ -1 \end{bmatrix} \rightarrow \begin{bmatrix} 1 \\ -1 \\ -1 \\ -1 \end{bmatrix} \equiv T_3$$

Note that, in this case, the network convergence occurs in only one cycle, although it may need more iteration for other inputs.

3.2.3 Mean Square Error (MSE) Algorithms

MSE algorithms emerged as an answer to the deficiencies experienced by using perceptrons and other simple networks [1, 15]. One of the most important reasons is the inseparability of training data. If the data used to train the network are naturally inseparable, the training algorithm never terminates (Figure 5.15).

The other reason for using this technique is to converge to a better solution. In perceptron learning, the training process terminates right after finding the first answer, regardless of its quality (i.e., sensitivity of the answer). Figure 5.16 shows an example of such a case. Note that, although the answer found by the perceptron algorithm is correct (Figure 5.16a), the answer in (Figure 5.16b) is more robust. Finally, another reason for using MSE algorithms, which is crucial for most neural network algorithms, is speed of convergence.

The MSE algorithm attempts to modify the network weights based on the overall error of all data. In this case, assume that network input and output data are represented by T_i, R_i for $i = 1...N$, respectively. Now the MSE error is defined as follows:

$$E = \frac{1}{N}\sum_{i=1}^{N}\left(W \cdot T_i - R_i\right)^2$$

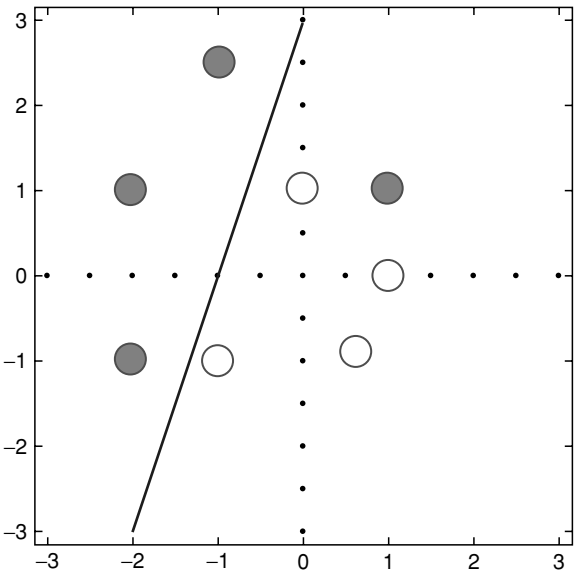

Figure 5.15. *An example of an inseparable training data set*

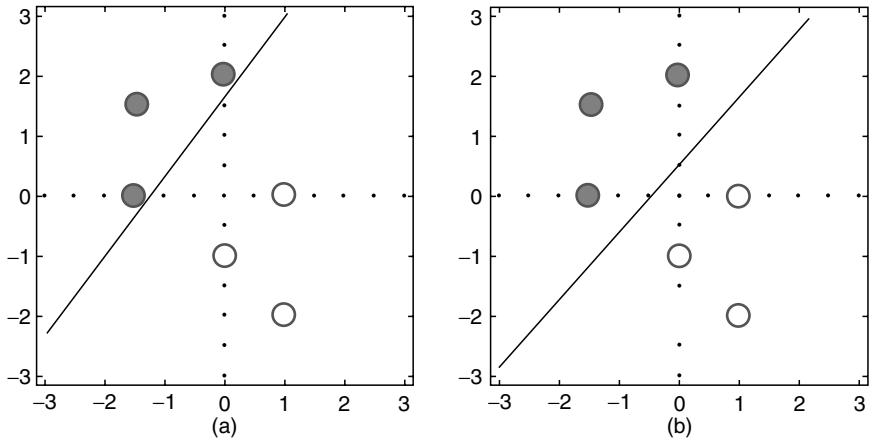

Figure 5.16. *Two classification answers for sample data*

Note that the stated error is the summation of all individual errors for all the training data. In spite of all the advantages of this training technique, there are several disadvantages. For example, the network might not be able to correctly classify the data if they are widely spread apart (Figure 5.17). The other disadvantage is that speed of convergence may completely vary from one set of data to another.

3.2.4 The Widow-Hoff Rule or LMS Algorithm

In the widow-Hoff algorithm, the network weights are modified after each iteration [1, 16]. A training datum is selected randomly, and then the network weights

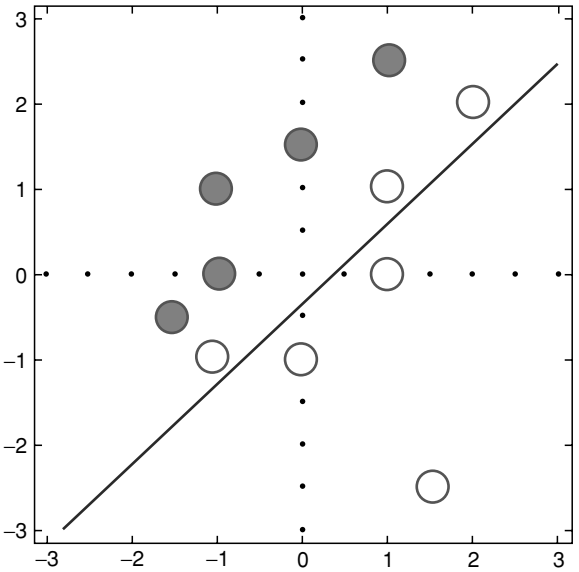

Figure 5.17. *An example of a training data set with spread-out members*

are modified based on the corresponding error. This procedure continues until it converges to the answer. For a randomly selected k^{th} entry in the training data, the error is calculated as follows:

$$\varepsilon = (W \cdot T_k - R_k)^2$$

Now the gradient vector of this error would be

$$\nabla\epsilon = \langle \frac{\partial\epsilon}{\partial W_0} \frac{\partial\epsilon}{\partial W_1} \cdots \frac{\partial\epsilon}{\partial W_N} \rangle$$

Hence,

$$\frac{\partial\epsilon}{\partial W_j} = 2(W \cdot T_k - R_k) \cdot T_k$$

Based on the Widow-Hoff algorithm, the weights should be modified opposite to the direction of the gradient. As a result, the final update formula for the weighting matrix W would be

$$W' = W - \rho \cdot (W \cdot T_k - R_k) \cdot T_k$$

Note that ρ is known as the learning rate and absorbs the multiplier of value 2.

3.4 Unsupervised Learning

Unsupervised learning networks attempt to cluster input data without the need for the traditional "learn by example" technique that is commonly used for neural networks. Note that clustering applications tend to be the most popular type of applications for which these networks are normally used. The most popular networks in this class are K-means, Kohonen, ART1, and ART2 [17-21].

3.4.1 K-Means Clustering

K-means clustering is the simplest technique used for classifying data. In this technique, a network with a predefined number of clusters is considered, and then each datum is assigned to one of these clusters. This process continues until all data are checked and classified properly. The following algorithm shows how this algorithm is implemented.

Step 1: Consider a network with *K* clusters.
Step 2: Assign all data to one of the above clusters, with respect to the distance from the center of the cluster and each datum.
Step 3: Modify the center of the assigned cluster.
Step 4: Check all data in the network to ensure proper classification.
Step 5: If a datum has to be moved from one cluster to another one, then update the center of both clusters.
Step 6: Repeat steps 4 and 5 until no datum is wrongly classified.

Figure 5.18 shows an example of such a network when applied for data classification with correct and incorrect numbers of clusters. As can be seen, if the number of clusters is properly guessed, then this algorithm can be very effective.

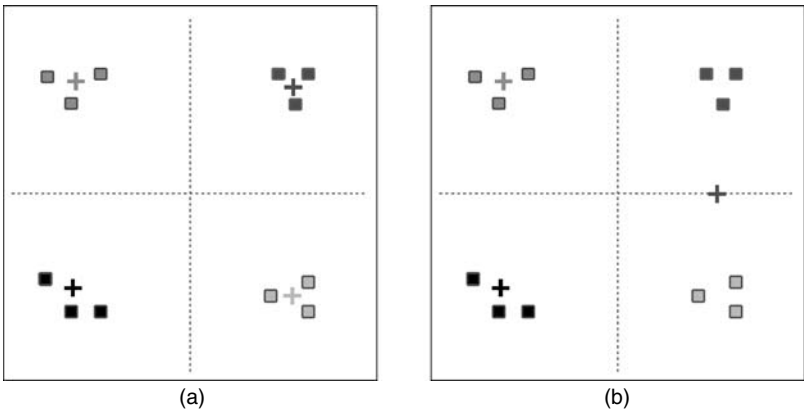

Figure 5.18. *Results of applying a k-means clusterer with (a) appropriate and (b) inappropriate numbers of clusters*

3.4.2 Kohonen Clustering

The Kohoner classification method clusters input data based on a topological representation of the data. The outputs of the network are arranged so that each output has some neighbors. Thus, during the learning process, not only one output but a group of close outputs are modified to classify the data. To clarify the situation, assume that a network is supposed to learn how a set of data is to be distributed in a two-dimensional representation (Figure 5.19).

In this case, each point is a potential output with a predefined neighborhood margin. For example, the cell marked X and eight of its neighbors are given. Therefore, whenever this cell gets selected for an update, all its neighbors are included in the process too. The main principle behind this approach for classifying input data is analogous to principles of biology. In a mammalian brain, all vision, auditory, and tactile sensors are mapped into a number of cell sheets. Therefore, if one of the cells is activated, all cells close to it will be affected, but with different intensity levels.

Now assume that a training data set, T_i for $i = 1...N$, is available and that the network must classify it based on a similarity measure. The main idea here is for

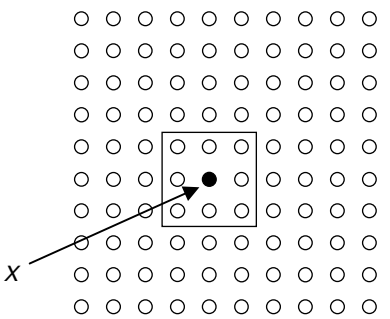

Figure 5.19. *Output topology of a sample Kohonen network*

the network to assign at least one of its output weights to fire for a particular training datum. To achieve this, for each training datum, the closest output is found, and then the corresponding weights are modified in order to get the minimum possible Euclidean distance, i.e. to minimize $\| T_k - W_{m,n} \|$. Another consideration during this training process is the learning rate of the algorithm. In general, at the outset, the network is trained with a fast learning rate, while in the final stages of the training process, the training data hardly change the network weights. The following procedure explains the details of this classifier:

Step 1: Define the algorithm step size $\rho(t) = \left(1 - \frac{t-1}{L}\right)$, where L is the predefined number of iterations.
Step 2: Generate a grid network with the dimension of the input data.
Step 3: Assign all network weights to random data.
Step 4: Select a random training data, T_k.
Step 5: Find the closest output of the network to T_k, and let this be $O_{m,n}$.
Step 6: Modify $O_{m,n}$ and its neighboring weights, with a predefined margin, as follows: $W_{x,y} = W_{x,y} + \rho(t) \cdot (T_k - W_{x,y})$.
Step 7: Set $t \leftarrow t + 1$ and repeat steps 4–7 until $t = L$.

Figure 5.20 shows the random data that need to be classified, while Figure 5.21 shows the end result for a number of iterations.

3.4.3 ART1

This neural classifier, known as *Adaptive Resonance Theory* or *ART*, deals with digital inputs ($T_i \in \{0,1\}$). In this network, each "1" in the input vector represents information, while a "0" entry is considered noise or unwanted information. In ART, there is no predefined number of classes before the start of classification; in fact, the classes are generated during the classification process.

Moreover, each class prototype may include the characteristics of more than a training datum. The basic principle of such a network relies on the similarity factor for data classification. In summary, every time a datum is assigned to a cluster, firstly, the nearest class with this datum is found, and then, if the similar-

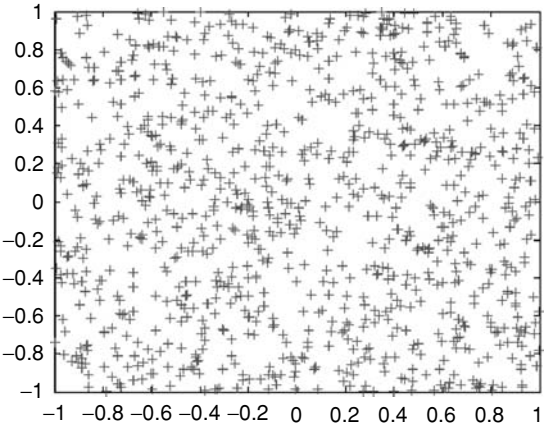

Figure 5.20. *Training data set for a Kohonen network*

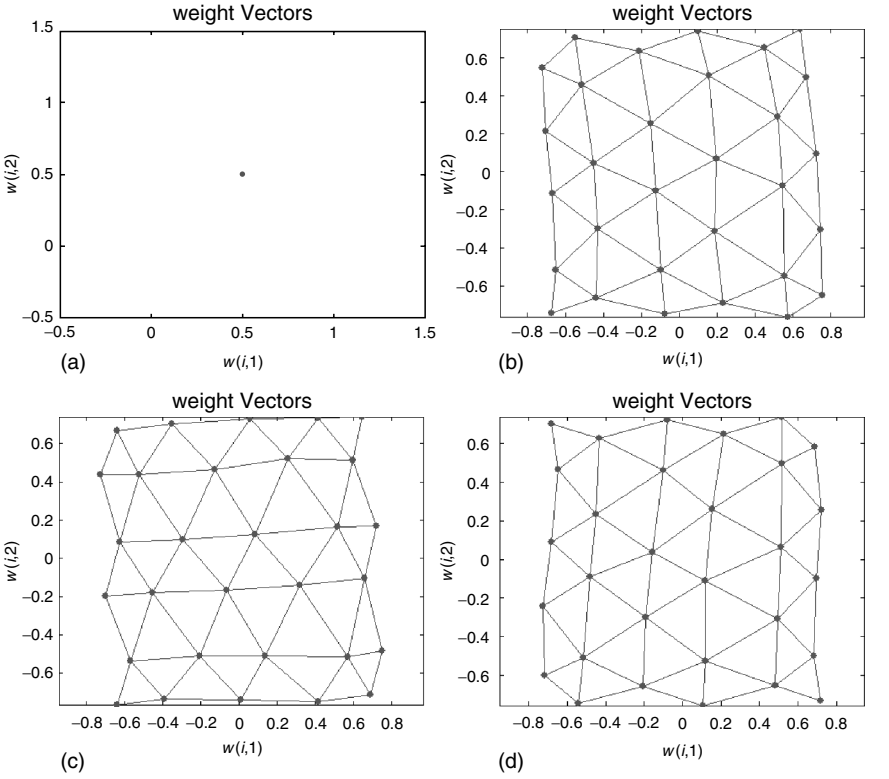

Figure 5.21. *Classification results for the Kohonen network after (a) 0, (b) 10, (c) 20, and (d) 30 iterations*

ity of this datum to the class prototype is more than a predefined value, known as a *vigilance factor*, the datum is assigned to this class and the class prototype is modified to have more similarity with the new data entry [1, 22, 23].

The procedure below shows how this algorithm is implemented. However, the following points need to be noted First.

1. $\|X\|$ is the number of 1s in the vector **X**.
2. $X \cdot Y$ is the number of common 1s between the vectors **X** and **Y**.
3. $X \cap Y$ is the bitwise AND operator applied on vectors **X** and **Y**.

Step 1: Let β be a small number, n be the dimension of the input data, and ρ be the vigilance factor ($0 \le \rho < 1$).

Step 2: Start with no class prototype.

Step 3: Select a training datum by random, T_k.

Step 4: Find the nearest unchecked class prototype, C_i, to this datum by minimizing $\dfrac{C_i \cdot T_k}{\beta + \|C_i\|}$.

Step 5: Test whether C_i is sufficiently close to T_k by verifying $\dfrac{C_i \cdot T_k}{\beta + \|C_i\|} > \dfrac{T_k}{\beta + \rho}$.

Step 6: If C_i is not similar enough, then assign a new class prototype and go to step 3.

Step 7: If it is sufficiently similar, check the vigilance factor: $\frac{C_i \cdot T_k}{\|T_k\|} \geq \rho$

Step 8: If the vigilance factor is exceeded, then modify the class prototype by $C_i = C_i \cap T_k$ and go to step 3.

Step 9: If the vigilance factor is not exceeded, then find another unchecked class prototype (step 4).

Step 10: Repeat steps 3–9 until none of the training data causes any change in class prototypes.

3.4.4 ART2

ART2 is a variation of ART1, with the following differences:

1. Data are considered continuous and not binary.
2. The input data are processed before passing them to the network. Actually, the input data are normalized, and then all elements of the result vector that are below a predefined value are set to zero and the vector is normalized again. The process is used for noise cancellation.
3. When a class prototype is found for a datum, the class prototype vector is moved fractionally toward the selected datum. As a result, contrary to the operation of ART1, the weights are moved smoothly toward a new datum. The main reason for such a modification is to "memorize" previously learnt rules.

The following algorithm demonstrates the working of ART2:

Step 1: Let; n be the dimension of the input data; α be a positive small number given by $\alpha \leq 1/\sqrt{n}$; λ be the normalized factor such that $0 < \lambda < 1/\sqrt{n}$; and ρ be the vigilance factor ($0 \leq \rho < 1$).

Step 2: Process all the training data for $k = 1 \cdot N$ as follows:
Normalize T_k.
Set all elements of T_k to 0 if they all are less or equal to λ.
Normalize T_k again.

Step 3: Start with no class prototype.

Step 4: Select a training datum randomly, T_k.

Step 5: Find the nearest unchecked class prototype, C_i, to this datum by minimizing $C_i \cdot T_k$.

Step 6: Test whether C_i is sufficiently close to T_k by verifying $C_i \cdot T_k \geq \alpha \cdot \sum_j T_j^k$.

Step 7: If C_i is not similar enough, then assign a new class prototype and go to step 4.

Step 8: If it is sufficiently similar, check the vigilance factor: $C_i \cdot T_k \geq \rho$.

Step 9: If the vigilance factor is exceeded, then modify the class prototype by $C_i = \frac{(1-\beta) \cdot C + \beta \cdot T_k}{\|(1-\beta) \cdot C + \beta \cdot T_k\|}$ and go to step 4.

Step 10: If the vigilance factor is not exceeded, then try to find another unchecked class prototype (step 5).

Step 11: Repeat steps 3–9 until none of the training data causes any change in class prototypes.

3.5 Learning in Multiple-Layer Models

As mentioned earlier, multilayer neural networks consist of several concatenated single-layer networks [1, 24–26]. The inner layers, known as hidden layers, may have different number of inputs and outputs. Because of this added complexity, the training process becomes more involved. This section presents two of the most popular multilayer neural networks.

3.6 Back-Propagation Algorithm

The back-propagation algorithm is one of the most powerful and reliable techniques that can be used to adjust network weights. The main principle of this approach is to use the gradient information of a cost function to modify the network's weights.

However, the use of such an approach to train multilayer networks is a little different from applying it to single-layer networks. In general, multilayer networks are much harder to train than single-layer ones. In fact, convergence of such networks is much slower and very error sensitive.

In this approach, an input is presented to the network and allowed to "forward" propagate through the network. The output is calculated, and then the output is compared with a "desired" output (from the training set) and an error is calculated. This error is then propagated "backward" into the network, and the different weights are updated accordingly. To simplify the description of this algorithm, consider a network with a single hidden layer (and two layers of weights), as shown in Figure 5.22. In relation to this network, the following definitions apply. Of course, the same definitions can be easily extended to larger networks.

T_i, R_i for $i = 1...L$: The training set of input and outputs, respectively.
N, S, M: The size of the input, hidden, and output layers, respectively.
W^1: Network weights from the input layer to the hidden layer.
W^2: Network weights from the hidden layer to the output layer.
X, Z, Y: Input and output of the hidden layer and the network output, respectively.
$F^1(\cdot)$: Array of network functions for the hidden layer.
$F^2(\cdot)$: Array of network functions for the output layer.
These definitions lead to the following formulas:

$$Z = F^1(W^1 \cdot X)$$

$$W^1 = \begin{bmatrix} w^1_{1,1} & w^1_{1,2} & \cdots & w^1_{1,n} \\ w^1_{2,1} & & & \\ \cdots & & & \\ w^1_{s,1} & & \cdots & w^1_{s,n} \end{bmatrix}$$

$$X = (x_1 \quad x_2 \quad \ldots \quad x_n)^T$$
$$Z = (z_1 \quad z_2 \quad \ldots \quad z_s)^T$$
$$F^1(\cdot) = (f^1_1(\cdot) \quad f^1_2(\cdot) \quad \ldots \quad f^1_s(\cdot))^T$$

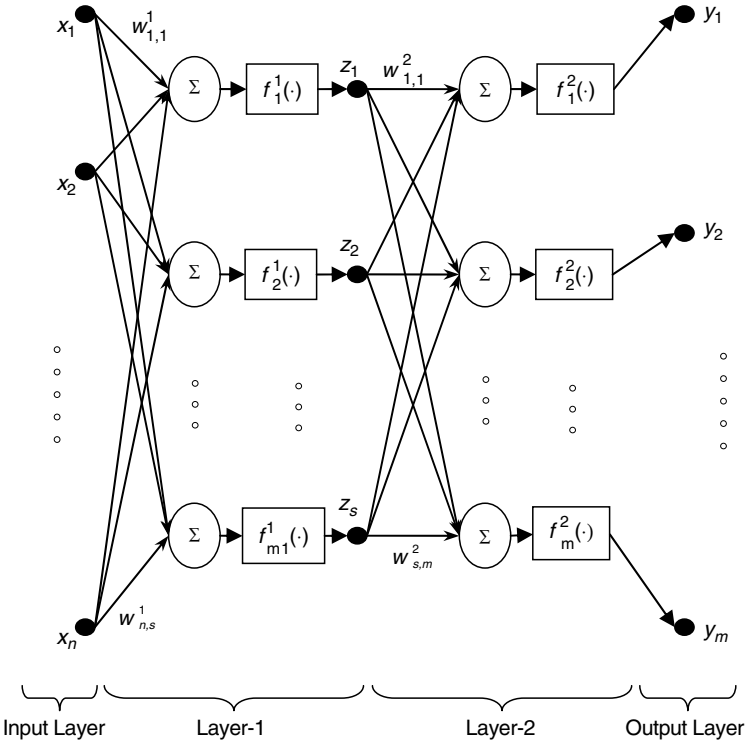

Figure 5.22. *A two-layer neural network*

$$Y = F^2\,(W^2 \cdot Z)$$

$$W^2 = \begin{bmatrix} w^2_{1,1} & w^2_{1,2} & \cdots & w^2_{1,s} \\ w^2_{2,1} & & & \\ \cdots & & & \\ w^2_{m,1} & & \cdots & w^2_{m,s} \end{bmatrix}$$

$$F^2(\cdot) = (f_1^2(\cdot) \quad f_2^2(\cdot) \quad \ldots \quad f_m^2(\cdot))^T$$

Now assume that the cost function for this optimization process is defined as

$$E = \frac{1}{2L}\sum_{k=1}^{L}\sum_{j=1}^{M}\left(R_j^{(k)} - Y_j^{(k)}\right)^2$$

where $Y_j^{(k)}$ and $R_j^{(k)}$ are the actual and desired outputs of the network, respectively. In this case, k represents the kth training datum and j is the jth output.

In this case, the following formulas represent the details:

$$Y_j^{(k)} = f_j^2\left(\sum_{s=1}^{S} w_{j,s}^2 \cdot Z_s^{(k)}\right) \equiv f_j^2\left(net2_j^{(k)}\right)$$

$$Z_s^{(k)} = f_s^1\left(\sum_{i=1}^{N} w_{s,i}^1 \cdot x_i^{(k)}\right) \equiv f_s^1\left(net1_s^{(k)}\right)$$

Now, the main cost function can be rewritten as follows:

$$E = \frac{1}{2L}\sum_{k=1}^{L}\sum_{j=1}^{M}\left(R_j^{(k)} - f_j^2\left(\sum_{s=1}^{S} w_{j,s}^2 \cdot f_s^1\left(\sum_{i=1}^{N} w_{s,i}^1 \cdot x_i^{(k)}\right)\right)\right)^2$$

Based on the gradient algorithm, with the assumption that all functions in the network are derivable, the following would apply for the output layer:

$$\frac{\partial E}{\partial w_{f,g}^2} = -\frac{1}{L}\sum_{k=1}^{L}\left(R_f^{(k)} - Y_f^{(k)}\right)\frac{\partial Y_f^{(k)}}{\partial w_{f,g}^2}$$

$$= -\frac{1}{L}\sum_{k=1}^{L}\left(R_f^{(k)} - Y_f^{(k)}\right)\frac{\partial f_f^2\left(net2_f^{(k)}\right)}{\partial net2_f^{(k)}}\frac{\partial net_f^{(k)}}{\partial w_{f,g}^2}$$

$$= -\frac{1}{L}\sum_{k=1}^{L}\lambda_{2_f}^{(k)} \cdot Z_g^{(k)}$$

where

$$\lambda_{2_f}^{(k)} = (R_f^{(k)} - Y_f^{(k)}) \cdot f_f^2(net_f^{(k)})$$

The gradient formulas for the hidden layer would be as follows:

$$\frac{\partial E}{\partial w_{f,g}^1} = \frac{1}{L}\sum_{k=1}^{L}\frac{\partial E}{\partial Z_f^{(k)}}\frac{\partial Z_f^{(k)}}{\partial w_{f,g}^1}$$

$$= -\frac{1}{L}\sum_{k=1}^{L}\sum_{j=1}^{M}\left(R_j^{(k)} - Y_j^{(k)}\right)\frac{\partial f^2\left(net_j^{(k)}\right)}{\partial net_j^{(k)}} w_{j,f}^2 \frac{\partial Z_j^{(k)}}{\partial w_{f,g}^1}$$

$$= -\frac{1}{L}\sum_{k=1}^{L}\sum_{j=1}^{M}\lambda_{2_j}^{(k)} \cdot w_{j,f}^2 \frac{\partial f^1\left(net_f^{(k)}\right)}{\partial net_f^{(k)}} \cdot X_g^{(k)}$$

$$= -\frac{1}{L}\sum_{k=1}^{L}\lambda_{1_f}^{(k)} \cdot X_g^{(k)}$$

where

$$\lambda_{1_f}^{(k)} = \dot{f}_f^1\left(net_f^{(k)}\right)\sum_{j=1}^{M} w_{j,f}^2 \cdot \lambda_{2_j}^{(k)}$$

To summarize the above technique, the back-propagation algorithm for a two-layer network can be derived as follows:

Step 1: Create a network with a predefined number of nodes in the hidden layer, and random weights for all links.

Step 2: Select a *k*th entry consisting of an input and desired output.

Step 3: Compute $net1_i^{(k)}$ and $Z_i^{(k)}$ for $i = 1... S$:

$$net1_i^{(k)} = \sum_{r=1}^{N} = w_{r,i}^1 \cdot X_r^{(k)}, Z_i^{(k)} = f_i^1(net1_i^{(k)})$$

followed by $net2_j^{(k)}$ and $Y_j^{(k)}$ for $j = 1...M$:

$$net2_i^{(k)} = \sum_{i=1} = w_{i,j}^2 \cdot Z_i^{(k)}, Y_j^{(k)} = f_j^2(net2_j^{(k)})$$

Step 4: Computer $\lambda_1 i^{(k)}$ and $\lambda_2 j^{(k)}$ for $i = 1 ... S$ and $j = 1...M$:

$$\lambda_{2_j}^{(k)} = \left(R_J^{(k)} - Y_J^{(k)}\right) f_j^2\left(net2_j^{(k)}\right)$$

$$\lambda_{1_i}^{(k)} = f_i^1\left(net1_i^{(k)}\right) \sum_{j=1}^{M} w_{i,j}^2 \cdot \lambda_{2_j}^{(k)}$$

Step 5: Calculate the gradient over the input batch:

$$\frac{\partial E}{\partial w_{i,j}^2} = \frac{\partial E}{\partial w_{i,j}^2} + \lambda_{2_j}^{(k)} \cdot Z_i^{(k)}$$

$$\frac{\partial E}{\partial w_{i,j}^1} = \frac{\partial E}{\partial w_{i,j}^1} + \lambda_{1_j}^{(k)} \cdot X_i^{(k)}$$

Step 6: Repeat steps 2–5 for all training data.
Step 7: Update the network weights as follows:

$$w_{i,j}^1(new) \leftarrow w_{i,j}^1(old) - \frac{\mu}{L} \frac{\partial E}{\partial w_{i,j}^1}$$

$$w_{i,j}^2(new) \leftarrow w_{i,j}^2(old) - \frac{\mu}{L} \frac{\partial E}{\partial w_{i,j}^2}$$

Step 8: Repeat steps 2–7 until a predefined accuracy measure is reached for the network.

3.7 Radial Basis Functions

The Radial Basis Function (RBF) neural network is another popular multilayer neural network [27–31]. The RBF network consists of two layers, one hidden layer and one output layer. In this network, the hidden layer is implemented by radial activation functions while the output layer is simply a weighted sum of the hidden-layer outputs.

The RBF neural network is able to model complex mappings, which perceptron neural networks can only accomplish by means of multiple hidden layers. The outstanding characteristics of such a network makes it applicable for a variety of applications, such as function interpolation [32, 33], chaotic time series modeling [34, 35], system identification [36–38], control systems [39, 40], channel equalization [41–43], speech recognition [44, 45], image restoration [46, 47], motion estimation [48], pattern classification [49], and data fusion [50].

3.7.1 Network Topology

The main topology of this network is as shown in Figure 5.23. Many functions were introduced for possible use in the hidden layer; however, radial functions (Gaussian) remain the most effective to use for data or pattern classification. The Gaussian functions are defined as follows:

$$\Phi_j(X) = \exp\left[-(X - \mu_j)^T \Gamma_j^{-1} (X - \mu_j)\right]$$

where $j = 1,2, \ldots, L$, L represents the number of nodes in the hidden layer, X is the input vector, μ_j and Γ_j are the mean vector and covariance matrix of the j^{th} Gaussian function, respectively. In some approaches, a polynomial term is

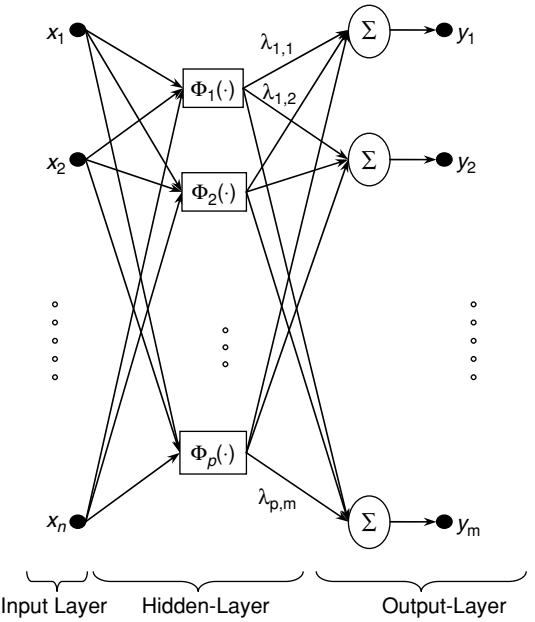

Figure 5.23. *The basic structure of an RBF network*

appended to the above expression, while in others the functions are normalized to the sum of all Gaussian components as in the Gaussian mixture estimation. Geometrically, a radial basis function in this network represents a bump in the N-dimensional space where N is the number of entries (input vector size). In this case, the μ_j represents the location of this bump in the space and Γ_j models its shape. The output layer of this network is a linear combination of the hidden layer outputs, as follows:

$$Y_k(X) = \sum_{i=1}^{L} \lambda_{i,k} \cdot \Phi_i(X)$$

where L is the number of outputs of the hidden layer, Y_k is the kth output, and $\lambda_{i,k}$ is a linear factor (connection weight) from the ith hidden layer output to the kth network output. In the classification application, the actual output of the network is usually limited by a sigmoid function to be between 0 and 1:

$$Z_k = \frac{1}{1 + \exp\left[-Y_k(X)\right]}$$

3.7.2 Training Algorithms

Because of the nonlinear behavior of this network, the training procedure of the RBF network (as in multilayer networks) is approached in a completely different manner from that of single-layer networks. In this network, the aim is to find the center and variance factor of all hidden-layer Gaussian functions as well as the optimal weights for the linear output layer. In this case, the following cost function is usually considered to be the main network objective:

$$Min\left(\sum_{i=0}^{N}\left(\left[Y(T_i)-R_i\right]^T\cdot\left[Y(T_i)-R_i\right]\right)\right)$$

where N is the number of inputs in the training data set, $Y(X)$ is the output of the network for input X, and $\langle T_k, R_k\rangle$ is the k^{th} training data pair. So the actual output of the network is a combination of a nonlinear computation followed by a linear operation. Therefore, finding an optimal set of weights for hidden layers and output layer parameters is hardly achievable.

In this case, several approaches were used to find the optimal set of weights; however, none of these can provide any guarantees that optimality can be achieved. For example, many approaches suggest that the hidden-layer parameters are set randomly and that the training procedure is just carried on for the output linear components. In contrast, in some other cases, the radial basis functions are homogenously distributed over the sample space before the output linear weights are found. However, the back-propagation algorithm seems to be the most suitable approach for training such a network.

Note that, in this approach, numerous iterations might be needed to converge to a suitable answer, so the probability of getting stuck in local minima during training process is unavoidable. To solve this problem, several algorithms were suggested that use another classification technique, such as K-means [51], to guess the initial location of the radial functions in the space. However, the following approach is mentioned as one of the simplest but most powerful approaches to tune these sensitive weights. The main idea of this technique is similar to that of Kohonen's training routine. In this method, the parameters of the radial basis functions are set randomly, and for each training datum, the closest radial basis function in the hidden layer is selected and modified. The following algorithm shows the details:

Step 1: Generate an RBF neural network with a predefined number of Gaussian functions in the hidden layer, namely, L functions.

Step 2: Set the center of these Gaussian functions randomly, and, set their variance using a predefined fixed value.

Step 3: Select a random datum from the training data, T_k.

Step 4: Find the closest center of these Gaussian function with reference to the following criterion: $\underset{1<j<P}{Min}\left(\|T_k-\mu_j\|\right)$, where μ_j is the center of the jth Gaussian function, μ^*.

Step 5: Modify the center of the closest function as $\mu^* = \mu^* + \gamma(T_k - \mu^*)$, where γ is an arbitrary small positive value as the learning rate.

Step 6: Repeat Steps 3–5 until all radial functions are almost fixed in space.

Step 7: Calculate the output-layer linear weights by Least Mean Squares or any other optimization routine.

Note that, during step 3–5, when the centers of the radial functions are calculated, some radial functions might get close to each other during the process, while others will never be affected. Therefore, an RBF neural network with an excessive number of radial functions is usually generated first; then, after the result generated by steps 3–5 is examined, the optimal number of radial functions is guessed and the training algorithm is restarted. Figure 5.24 shows an RBF network with two radial functions in the hidden layer that managed to learn the behavior of a given data set.

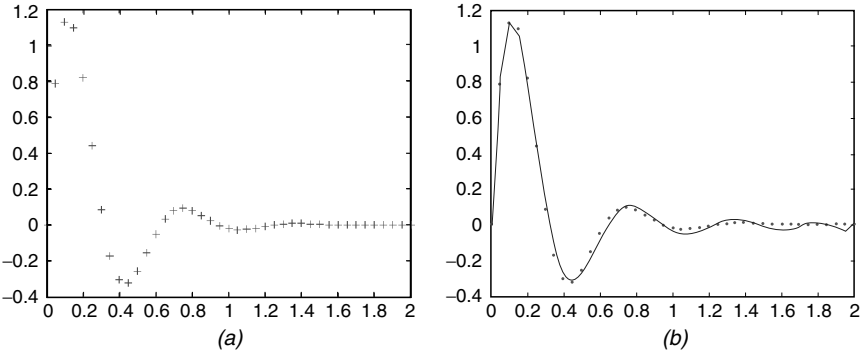

Figure 5.24. *(a) A typical data set for an RBF neural network (b) corresponding network output after training*

4 LEARNING VECTOR QUANTIZATION

The Learning Vector Quantization (LVQ) network is another popular classification technique that can be used for data clustering [52–54]. In this technique, a two-layer network topology is used to classify the data, which can be used in either supervised or unsupervised training.

In the supervised mode, a training data set is used to adjust the network parameters (Figure 5.25). The first layer of this network follows the "winner-takes-all" routine, while the second layer is linear. Basically, in the winner-takes-all topology, the input vector is fed to the layer, and based on the layer outputs, only one of the outputs is set to "1" and the rest to "0." The basic topology of this layer is usually the distance between network weights and the input datum. This process is followed by a comparator to evaluate the outputs of different nodes and to emphasize the largest element by setting it to "1" and the others to "0."

4.1 Learning Algorithm

Because of the nonlinear nature of the first layer, the training of such a network requires certain considerations. The back-propagation and other learning methods that try to adjust network parameters by using gradient information of the cost function cannot be readily applied here. However, a similar formulation is used here with some modifications.

The main idea of the LVQ algorithm is based on a simple rule. A number of hidden nodes with random weights are set as the first layer to build subclasses, and then some of these subclasses are merged together to make up the final network output. In this case, the final classes are assumed to have approximately equal number of subclasses (Figure 5.26).

The other key element of this technique is the use of fixed subclass assignments during the whole process of training. In fact, the second-layer weight matrix is set only once, and the training process modifies the parameters of the

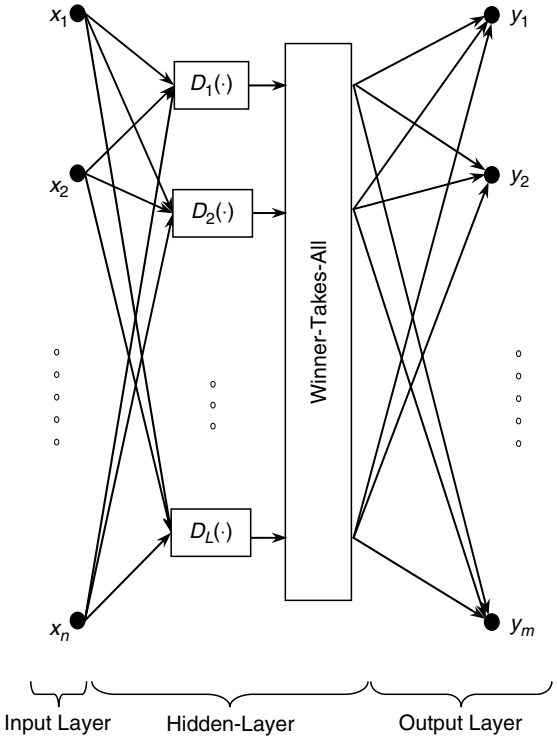

Figure 5.25. *Basic structure of a LVQ network*

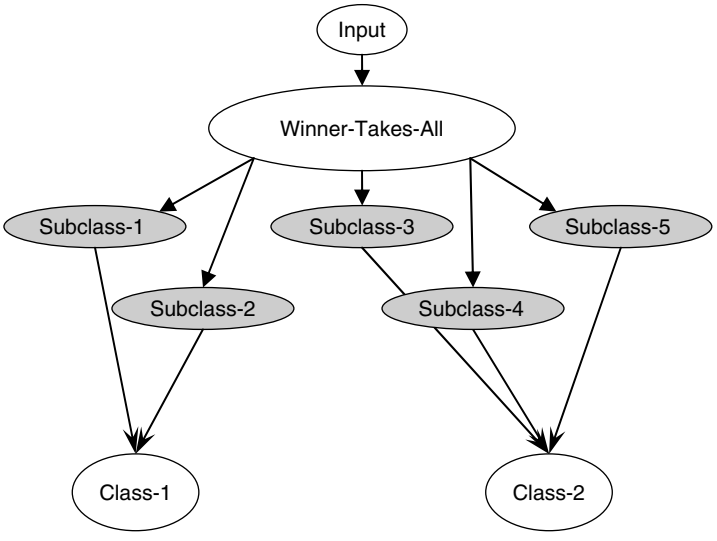

Figure 5.26. *Subclass association of an LVQ network*

first layer only. Based on this classification technique, the weight matrix of the output layer has only a "1" entry in each column, to distinguish the final class, and several 1s in each row to assign several subclasses to a class. Now each training data entry is fed into the network, and its corresponding output is calculated, based on the current network parameters. In this case, the parameters of the nodes in the first layer that won (or lost) the competition are slightly moved toward (or away from) this datum if the network correctly (incorrectly) classifies this entry.

The following algorithm describes the overall procedure and the $\langle T_k, R_k \rangle$ represents the kth training entry (input, output).

Step 1: Set random weights for all L nodes in the first layer.

Step 2: Assign different subclasses to the same class randomly and homogenously, because these weight won't be changed during the training process.

Step 3: Select a random training datum, T_k.

Step 4: Find the neuron in the first layer that is closest to this input, i^*.

Step 5: Calculate the final output answer for the input vector.

Step 6: If the network output is the same as the desired value, R_k, then adjust the weights of the i^* nodes of the first layer as follows: $w_{i^*} \leftarrow w_{i^*} + \lambda(T_k - w_{i^*})$.

Step 7: If the network output is not the same as the desired value, R_k, then adjust the weights of the i^* nodes of the first layer as follows: $w_{i^*} \leftarrow w_{i^*} - \lambda(T_k - w_{i^*})$.

Step 8: Repeat steps 3–7 until all data are correctly classified.

Note that λ is the learning rate, which can be set to a constant or modified during the training process.

To clarify this procedure, assume that the training data set is as follows:

$$\langle T, R \rangle = \left\{ \left\langle \begin{bmatrix} 0 \\ 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} \right\rangle, \left\langle \begin{bmatrix} 0 \\ 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} \right\rangle, \left\langle \begin{bmatrix} 1 \\ 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \end{bmatrix} \right\rangle, \left\langle \begin{bmatrix} 1 \\ 1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} \right\rangle \right\}$$

Also, assume that the weights matrix of the first and second layer is set as follows:

$$W^1 = \begin{bmatrix} 0.1 & 0.2 \\ 0.2 & 0.3 \\ 0.3 & 0.1 \\ 0.2 & 0.2 \end{bmatrix} \text{ and } W^2 = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

Now, suppose the second training datum is selected to train the network. Therefore, the output of the first layer would be

$$A(1) = \begin{bmatrix} \|[0.1 \quad 0.2] - [0 \quad 1]\| \\ \|[0.2 \quad 0.3] - [0 \quad 1]\| \\ \|[0.3 \quad 0.1] - [0 \quad 1]\| \\ \|[0.2 \quad 0.2] - [0 \quad 1]\| \end{bmatrix} = \begin{bmatrix} 0.8062 \\ 0.7280 \\ 0.9487 \\ 0.8246 \end{bmatrix} \xrightarrow{compete} \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

and the output of the network would be

$$A(2) = W^2 \cdot A(1) = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \neq \begin{bmatrix} 1 \\ 1 \end{bmatrix} = R_2$$

Note that the network output is different from the desired output. Therefore, the weighing vector of the second node of the first layer is modified as follows:

$$w_3 = \begin{bmatrix} 0.3 \\ 0.1 \end{bmatrix} - 0.2 \cdot \left(\begin{bmatrix} 1 \\ 1 \end{bmatrix} - \begin{bmatrix} 0.3 \\ 0.1 \end{bmatrix} \right) = \begin{bmatrix} 0.16 \\ -0.08 \end{bmatrix}$$

The final weight matrix for the first layer would be

$$W^1 \leftarrow \begin{bmatrix} 0.1 & 0.2 \\ 0.16 & -0.08 \\ 0.3 & 0.1 \\ 0.2 & 0.2 \end{bmatrix}$$

This procedure should be continued until all the data have been correctly classified.

5 NEURAL NETWORK APPLICATIONS

This section briefly reviews a number of application areas in which neural networks have been used effectively. This is by no means an exhaustive list of applications.

5.1 EXPERT SYSTEMS

One popular application is the use of neural networks as expert systems. Several definitions have been presented to clearly distinguish this kind of systems from other approaches [55–57]. Generally, an expert system is defined as a system than can imitate the action of a human being for a given process. This definition does not restrict the design of such systems by traditional Artificial Intelligence approaches. Therefore, a variety of such systems can be built by using Fuzzy Logic, Neural Networks, and Neuro-Fuzzy techniques. In most of these systems, there is always a knowledge-based component that holds information about the behavior of the system as simple rules followed by operators (usually in Fuzzy Systems) or a large database collected from the system performance that a neural network can be trained to emulate (Figure 5.27).

5.2 NEURAL CONTROLLERS

Neural controllers are a specific class of expert systems that deal with the process of regulating a liner or nonlinear system (Figure 5.28). There are two methods to train such system, namely, supervised and unsupervised. In the super-

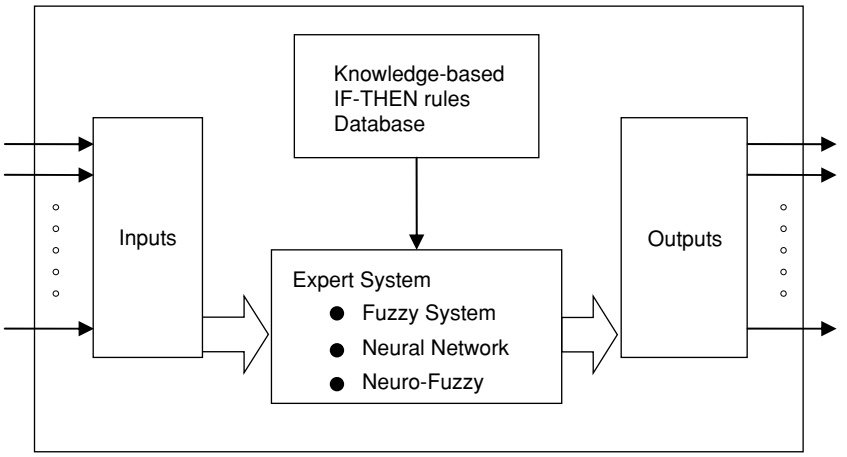

Figure 5.27. *A generic expert system*

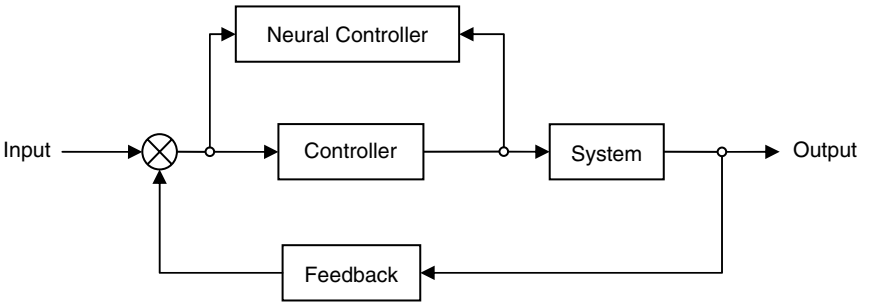

Figure 5.28. *A neural network controller*

vised approach, another controller usually exists, and the neural controller is trained to imitate its behavior. In this case, the neural controller is connect in parallel to the other controller, and during the process, by sampling inputs and outputs, the network is trained to generate similar outputs for similar inputs of the real controller. This process is known as *online training*. In contrast, in the case of offline training, a database of the real-controller inputs and outputs can be employed to train the network [58-60].

6 DECISION MAKERS

In the specific class of decision makers, which can also be viewed as an expert system, a neural network is used to make critical decisions in unexpected situations. One such application is popular in financial markets such as stock market applications. One of the main characteristics of such systems that distinguish them from simple expert systems is their stability. In fact, these systems must be able to produce acceptable output for untrained situations. Therefore, a sufficiently rich data set must be used for the training process [61–63] (see Figure 5.29).

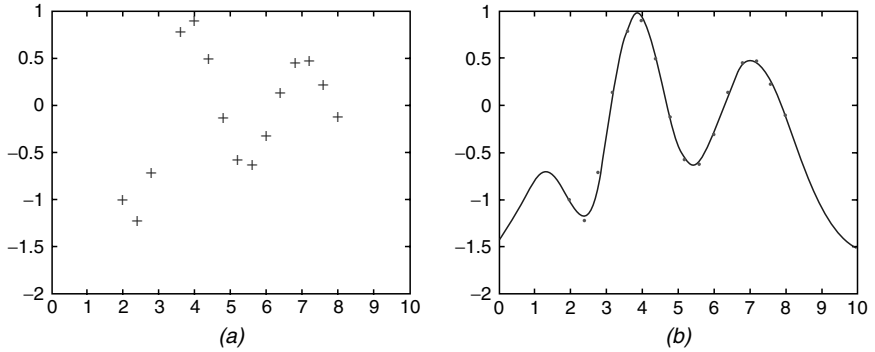

Figure 5.29. *(a) Sample data and (b) network output after training*

7 ROBOT PATH PLANNING

Another complex scenario in which neural networks have been used with some promise is that of robot path planning. In this case, the robot tries to navigate its way to reach a target location. The situation can be made more complicated by adding obstacles in the environment or even other mobile robots. Normally, this situation is modeled as an optimization problem in which some cost function is minimized (e.g., minimize the distance that the robot needs to travel) while satisfying certain constraints (e.g., no collisions) [64–66] (see Figures 5.30 and 5.31).

8 ADAPTIVE NOISE CANCELLATION

Neural networks have been used very effectively to filter noise. In this case, the target signal (in the training set) is the nonnoisy signal that the input should be

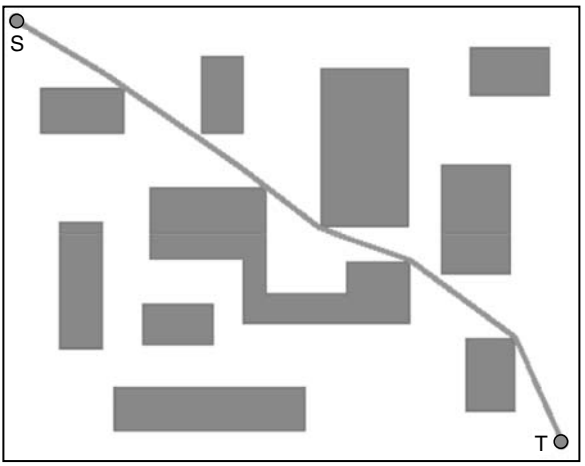

Figure 5.30. *An optimal path for a sample robot work space*

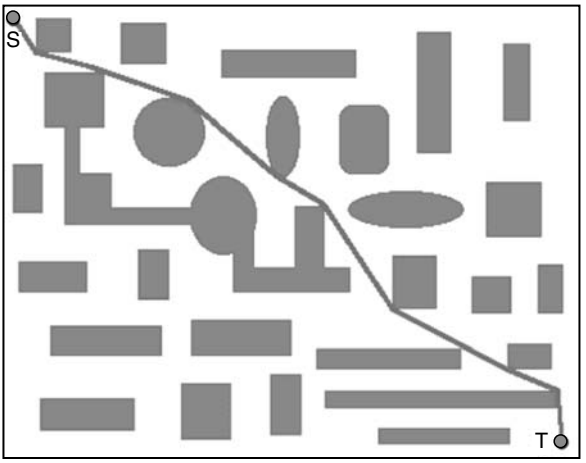

Figure 5.31. *A robot work space with deep U-traps*

generating. The network must learn how to imitate the noise and in the process manage to neutralize it. Many approaches have been introduced in the literature over the years, and some of these have been deployed in real environments [67–69]. An example is provided in Figures 5.32 and 5.33.

9 CONCLUSION

In this chapter, a general overview of artificial neural networks has been presented. These networks vary in their sophistication from the very simple to the more complex. As a result, their training techniques vary as well as their capabilities and suitability for certain applications. Neural networks have

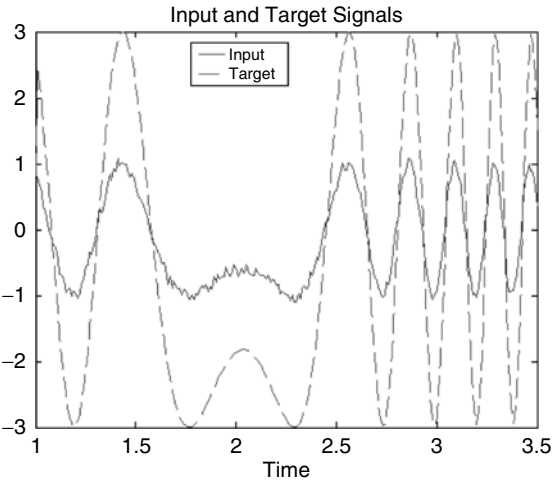

Figure 5.32. *Input and target signals for a noise-cancellation neural network*

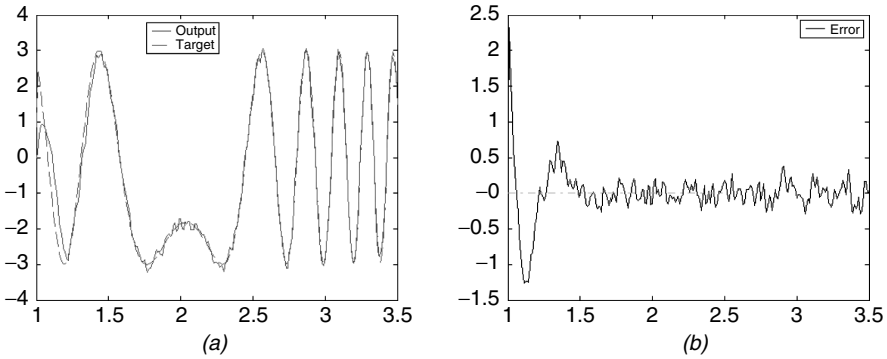

Figure 5.33. *Final performance and error signal of the noise-cancellation neural network*

attracted a lot of interest over the last few decades, and it is expected they will be an active area of research for years to come. Undoubtedly, more robust neural techniques will be introduced in the future that could benefit a wide range of complex applications.

REFERENCES

[1] S. I. Gallant (1993): *Neural Network Learning and Expert Systems*, MIT Press.

[2] N.B. Karayiannis and A.N. Venetsanopoulos, (1993) Efficient learning algorithms for neural networks (ELEANNE), *IEEE Transactions on Systems, Man and Cybernetics*, *23*(5), 1372–1383.

[3] M.H. Hassoun and D.W. Clark (1988): An adaptive attentive learning algorithm for single-layer neural networks, in *Proceedings of the IEEE International Conference on Neural Networks*, *1*, 431–440.

[4] M.E. Ulug (1994): A single layer fast learning fuzzy controller/filter: Neural Networks, in *Proceedings of the IEEE World Congress on Computational Intelligence*, *3*, 1662–1667.

[5] N.B. Karayiannis and A.N. Venetsanopoulos (1992): Fast learning algorithms for neural networks, *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, *39*(7), 453–474.

[6] T. Hrycej (1991): Back to single-layer learning principles, in *Proceedings of the International Joint Conference on Neural Networks*, Seattle, *2*, 945.

[7] M.J. Healy (1991): A logical architecture for supervised learning: Neural Networks, in *Proceedings of the IEEE International Joint Conference on Neural Networks*, *1*, 190–195.

[8] R.D. Brandt and L. Feng (1996): Supervised learning in neural networks without feedback network, in *Proceedings of the IEEE International Symposium on Intelligent Control*, pp. 86–90.

[9] Y. Gong and P. Yan (1995): Neural network based iterative learning controller for robot manipulators, in *Proceedings of the IEEE International Conference on Robotics and Automation*, *1*, 569–574.
[10] S. Park and T. Han (2000): Iterative inversion of fuzzified neural networks, *IEEE Transactions on Fuzzy Systems*, *8*(3), 266–280.
[11] X. Zhan, K. Zhao, S. Wu, M. Wang, and H. Hu (1997): Iterative learning control for nonlinear systems based on neural networks, in *Proceedings of the IEEE International Conference on Intelligent Processing Systems*, *1*, 517–520.
[12] C.J. Chen, A.L. Haque, and J.Y. Cheung (1992): An efficient simulation model of the Hopfield neural networks, in *Proceedings of the International Joint Conference on Neural Networks*, *1*, 471–475.
[13] G. Galan-Marin and J. Munoz-Perez (2001): Design and analysis of maximum Hopfield networks, *IEEE Transactions on Neural Networks*, *12* (2), 329–339.
[14] N.M. Nasrabadi and W. Li (1991): Object recognition by a Hopfield neural network, *IEEE Transactions on Systems, Man and Cybernetics*, *21* (6), 1523–1535.
[15] J. Xu, X. Zhang, and Y. Li (2001): Kernel MSE algorithm: a unified framework for KFD, LS-SVM and KRR, in *Proceedings of the International Joint Conference on Neural Networks*, *2*, 1486–1491.
[16] T. Hayasaka, N. Toda, S. Usui, and K. Hagiwara (1996): On the least square error and prediction square error of function representation with discrete variable basis, in *Proceedings of the Workshop on Neural Networks for Signal Processing*, *6*, 72–81. IEEE Signal Processing Society.
[17] D.-C. Park (2000): Centroid neural network for unsupervised competitive learning, *IEEE Transactions on Neural Networks*, *11*(2), 520–528.
[18] W. Pedrycz and J. Waletzky (1997): Neural-network front ends in unsupervised learning, *IEEE Transactions on Neural Networks*, *8*(2), 390–401.
[19] D.-C. Park (1997): Development of a neural network algorithm for unsupervised competitive learning, in *Proceedings of the International Conference on Neural Networks*, *3*, 1989–1993.
[20] K.-R. Hsieh and W.-T. Chen (1993): A neural network model which combines unsupervised and supervised learning, *IEEE Transactions on Neural Networks*, *4* (2), 357–360.
[21] A.L. Dajani, M. Kamel, and M.I. Elmastry (1990): Single layer potential function neural network for unsupervised learning, in *Proceedings of the International Joint Conference on Neural Networks*, *2*, 273–278.
[22] M. Georgiopoulos, G.L. Heileman, and J. Huang (1991): Properties of learning in ART1, in *Proceedings of the IEEE International Joint Conference on Neural Networks*, *3*, 2671–2676.
[23] G.L. Heileman, M. Georgiopoulos, and J. Hwang (1994): A survey of learning results for ART1 networks, in the *Proceedings of the IEEE International Conference on Neural Networks*, IEEE World Congress on Computational Intelligence, *2*, 1222–1225.
[24] J. Song and M.H. Hassoun (1990): Learning with hidden targets, in the *Proceedings of the International Joint Conference on Neural Networks*, *3*, 93–98.

[25] H.K. Kwan (1991): Multilayer feedbackward neural networks, in *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, *2*, 1145–1148.

[26] J.F. Shepanski (1988): Fast learning in artificial neural systems: multilayer perceptron training using optimal estimation, in *Proceedings of the IEEE International Conference on Neural Networks*, *1*, 465–472.

[27] N.B. Karayiannis and M.M. Randolph-Gips (2003): On the construction and training of reformulated radial basis function neural networks, *IEEE Transactions on Neural Networks*, *14* (4), 835–846.

[28] J.A. Leonard and M.A. Kramer (1991): Radial basis function networks for classifying process faults, *IEEE Control Systems Magazine*, *11*(3), 31–38.

[29] R. Li, G. Lebby, and S. Baghavan (2002): Performance evaluation of Gaussian radial basis function network classifiers, *SoutheastCon, 2002, Proceedings IEEE*, pp. 355–358.

[30] F. Heimes and B. van Heuveln (1998): The normalized radial basis function neural network, in *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics*, *2*, 1609–1614.

[31] R.J. Craddock and K. Warwick (1996): Multi-layer radial basis function networks. An extension to the radial basis function, in the *Proceedings of the IEEE International Conference on Neural Networks*, *2*, 700–705.

[32] J.C. Carr, W.R. Fright and R.K. Beatson (1997): Surface interpolation with radial basis functions for medical imaging, *IEEE Transactions on Medical Imaging*, *16*(1), 96–107.

[33] M.A. Romyaldy Jr. (2000): Observations and guidelines on interpolation with radial basis function network for one dimensional approximation problem, in the *Proceedings of the 26th Annual Conference of the IEEE Industrial Electronics Society*, *3*, 2129–2134.

[34] H. Leung, T. Lo, and S. Wang, (2001): Prediction of noisy chaotic time series using an optimal radial basis function neural network, *IEEE Transactions on Neural Networks*, *12*(5), 1163–1172.

[35] R. Katayama, Y. Kajitani, K. Kuwata, and Y. Nishida (1993): Self generating radial basis function as neuro-fuzzy model and its application to nonlinear prediction of chaotic time series, in a *Proceedings of the Second IEEE International Conference on Fuzzy Systems*, pp. 407–414.

[36] K. Warwick and R. Craddock (1996): An introduction to radial basis functions for system identification. A comparison with other neural network methods, in the *Proceedings of the 35th IEEE Decision and Control Conference*, *1*, 464–469.

[37] Y. Lu, N. Sundararajan and P. Saratchandran (1996): Adaptive nonlinear system identification using minimal radial basis function neural networks, in *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, *6*, 3521–3524.

[38] S. Tan, J. Hao, and J. Vandewalle (1995): A new learning algorithm for RBF neural networks with applications to nonlinear system identification, in *Proceedings of the IEEE International Symposium on Circuits and Systems*, *3*, 1708–1711.

[39] T. Ibayashi, T. Hoya, and Y. Ishida (2002): A model-following adaptive controller using radial basis function networks, in *Proceedings of the International Conference on Control Applications*, *2*, 820–824.

[40] P.K. Dash, S. Mishra and G. Panda (2000): A radial basis function neural network controller for UPFC, *IEEE Transactions on Power Systems*, *15*(4), 1293–1299.

[41] J. Deng, S. Narasimhan, and P. Saratchandran (2002): Communication channel equalization using complex-valued minimal radial basis function neural networks, *IEEE Transactions on Neural Networks*, *13*(3), 687–696.

[42] J. Lee, C.D. Beach, and N. Tepedelenlioglu (1996): Channel equalization using radial basis function network, in *Proceedings of the IEEE International Conference on Neural Networks*, *4*, 1924–1928.

[43] J. Lee, C.D. Beach, and N. Tepedelenlioglu (1996): Channel equalization using radial basis function network, in *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, *3*, 1719–1722.

[44] R. Sankar and N.S. Sethi (1997): Robust speech recognition techniques using a radial basis function neural network for mobile applications, in *Proceedings of IEEE Southeastcon*, pp. 87–91.

[45] H. Ney (1991): Speech recognition in a neural network framework: discriminative training of Gaussian models and mixture densities as radial basis functions, in *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, *1*, 573–576.

[46] I. Cha and S.A. Kassam (1994): Nonlinear image restoration by radial basis function networks, in *Proceedings of the IEEE International Conference on Image Processing*, *2*, 580–584.

[47] I. Cha and S.A. Kassam (1996): Nonlinear color image restoration using extended radial basis function networks, in *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, *6*, 3402–3405.

[48] A.G. Bors and I. Pitas (1998): Optical flow estimation and moving object segmentation based on median radial basis function network, *IEEE Transactions on Image Processing*, *7* (5), 693–702.

[49] D. Gao and G. Yang (2002): Adaptive RBF neural networks for pattern classifications, in *Proceedings of the International Joint Conference on Neural Networks*, *1*, 846–851.

[50] C. Fan, Z. Jin, J. Zhang, and W. Tian (2002): Application of multisensor data fusion based on RBF neural networks for fault diagnosis of SAMS, in *Proceedings of the 7th International Conference on Control, Automation, Robotics and Vision*, *3*, 1557–1562.

[51] J.T. Tou and R.C. Gonzalez (1974), *Pattern Recognition*, Reading, MA, Addison-Wesley.

[52] Z.-P. Lo, Y. Yu, and B. Bavarian (1992): Derivation of learning vector quantization algorithms, in *Proceedings of the International Joint Conference on Neural Networks*, *3*, 561–566.

[53] P. Burrascano (1991): Learning vector quantization for the probabilistic neural network, *IEEE Transactions on Neural Networks*, *2*(4), 458–461.

[54] N.B. Karayiannis and M.M. Randolph-Gips (2003): Soft learning vector quantization and clustering algorithms based on non-Euclidean norms: multinorm algorithms, *IEEE Transactions on Neural Networks*, *14*(1), 89–102.

[55] L. Medsker (1994): Design and development of hybrid neural network and expert systems, in *Proceedings of the IEEE International Conference on Neural Networks*, IEEE World Congress on Computational Intelligence, *3*, 1470–1474.

[56] M.S. Kurzyn (1993): Expert systems and neural networks: a comparison, Artificial Neural Networks and Expert Systems, in *Proceedings of the First International Two-Stream Conference on Neural Networks*, New Zealand, pp. 222–223.

[57] A.V. Hudli, M.J. Palakal and M.J. Zoran (1991): A neural network based expert system model, in *Proceedings of the Third International Conference on Tools for Artificial Intelligence*, pp. 145–149.

[58] W.-Y. Wang, C.-Y. Cheng and Y.-G. Leu (2004): An online GA-based output-feedback direct adaptive fuzzy-neural controller for uncertain nonlinear systems, in *IEEE Transactions on Systems, Man and Cybernetics, Part B*, *34*(1), 334–345.

[59] Y. Zhang, P.-Y. Peng and Z.-P. Jiang (2000): Stable neural controller design for unknown nonlinear systems using backstepping, *IEEE Transactions on Neural Networks*, *11*(6), 1347–1360.

[60] A.L. Nelson, E. Grant and G. Lee (2003): Developing evolutionary neural controllers for teams of mobile robots playing a complex game, in *Proceedings of the IEEE International Conference on Information Reuse and Integration*, pp. 212–218.

[61] L. Rothrock (1992): Modeling human perceptual decision-making using an artificial neural network, in *Proceedings of the International Joint Conference on Neural Networks*, *2*, 448–452.

[62] S. Mukhopadhyay and H. Wang (1999): Distributed decomposition architectures for neural decision-makers, in *Proceedings of the 38th IEEE Conference on Decision and Control*, *3*, 2635–2640.

[63] G. Rogova, P. Scott, and C. Lolett (2002): Distributed reinforcement learning for sequential decision making, in *Proceedings of the Fifth International Conference on Information Fusion*, *2*, 1263–1268.

[64] J. Taheri and N. Sadati, (2003): Fully modular online controller for robot navigation in static and dynamic environments, in *Proceedings of the 2003 IEEE International Symposium on Computational Intelligence in Robotics and Automation*, *1*, 163–168.

[65] N. Sadati and J. Taheri (2002): Genetic algorithm in robot path planning problem in crisp and fuzzified environments, in *Proceedings of the IEEE International Conference on Industrial Technology*, *1*, 175–180.

[66] N. Sadati and J. Taheri (2002): Solving robot motion planning problem using Hopfield neural network in a fuzzified environment, in *Proceedings of IEEE International Conference on Fuzzy Systems*, *2*, 1144–1149.

[67] R. Bambang (2002): Active noise cancellation using recurrent radial basis function neural networks, in *Proceedings of the Asia-Pacific Conference on Circuits and Systems*, *2*, 231–236.

[68] C.K. Chen and T.-D. Chiueh (1996): Multilayer perceptron neural networks for active noise cancellation, in *Proceedings of the IEEE International Symposium on Circuits and Systems*, *3*, 523–526.

[69] L. Tao and H.K. Kwan (1999): A neural network method for adaptive noise cancellation, circuits and systems, in *Proceedings of the IEEE International Symposium on Circuits and Systems*, *5*, 567–570.