Hamilton, M. 1986. Zero-defect software: The elusive goal. *IEEE Spectrum* 23(3):48–53, March.
Hamilton, M. and Zeldin, S. 1976. Higher Order Software—A Methodology for Defining Software. *IEEE Transactions on Software Engineering*, vol. SE-2, no. 1.
McCauley, B. 1993. Software Development Tools in the 1990s. AIS Security Technology for Space Operations Conference, Houston, Texas.
Schindler, M. 1990. *Computer Aided Software Design*. John Wiley & Sons, New York.
The 001 *Tool Suite Reference Manual*, Version 3, Jan. 1993. Hamilton Technologies Inc., Cambridge, MA.

19.3 Neural Networks and Fuzzy Systems

Bogdan M. Wilamowski

19.3.1 Neural Networks and Fuzzy Systems

New and better electronic devices have inspired researchers to build intelligent machines operating in a fashion similar to the human nervous system. Fascination with this goal started when McCulloch and Pitts (1943) developed their model of an elementary computing neuron and when Hebb (1949) introduced his *learning rules*. A decade latter Rosenblatt (1958) introduced the **perceptron** concept. In the early 1960s Widrow and Holf (1960, 1962) developed intelligent systems such as ADALINE and MADALINE. Nillson (1965) in his book *Learning Machines* summarized many developments of that time. The publication of the Mynsky and Paper (1969) book, with some discouraging results, stopped for sometime the fascination with artificial neural networks, and achievements in the mathematical foundation of the **backpropagation** algorithm by Werbos (1974) went unnoticed. The current rapid growth in the area of neural networks started with the Hopfield (1982, 1984) recurrent network, Kohonen (1982) unsupervised training algorithms, and a description of the backpropagation algorithm by Rumelhart et al. (1986).

19.3.2 Neuron cell

A biological neuron is a complicated structure, which receives trains of pulses on hundreds of *excitatory* and *inhibitory* inputs. Those incoming pulses are summed with different weights (averaged) during the time period of *latent summation*. If the summed value is higher than a threshold, then the neuron itself is generating a pulse, which is sent to neighboring neurons. Because incoming pulses are summed with time, the neuron generates a pulse train with a higher frequency for higher positive excitation. In other words, if the value of the summed weighted inputs is higher, the neuron generates pulses more frequently. At the same time, each neuron is characterized by the nonexcitability for a certain time after the firing pulse. This so-called *refractory period* can be more accurately described as a phenomenon where after excitation the threshold value increases to a very high value and then decreases gradually with a certain time constant. The refractory period sets soft upper limits on the frequency of the output pulse train. In the biological neuron, information is sent in the form of frequency modulated pulse trains.

This description of neuron action leads to a very complex neuron model, which is not practical. McCulloch and Pitts (1943) show that even with a very simple neuron model, it is possible to build logic and memory circuits. Furthermore, these simple neurons with thresholds are usually more powerful than typical logic gates used in computers. The McCulloch-Pitts neuron model assumes that incoming and outgoing signals may have only binary values 0 and 1. If incoming signals summed through positive or negative weights have a value larger than threshold, then the neuron output is set to 1. Otherwise, it is set to 0.

$$T = \begin{cases} 1 & \text{if} \quad net \geq T \\ 0 & \text{if} \quad net < T \end{cases} \tag{19.1}$$

where T is the threshold and *net* value is the weighted sum of all incoming signals

$$net = \sum_{i=1}^{n} w_i x_i \tag{19.2}$$

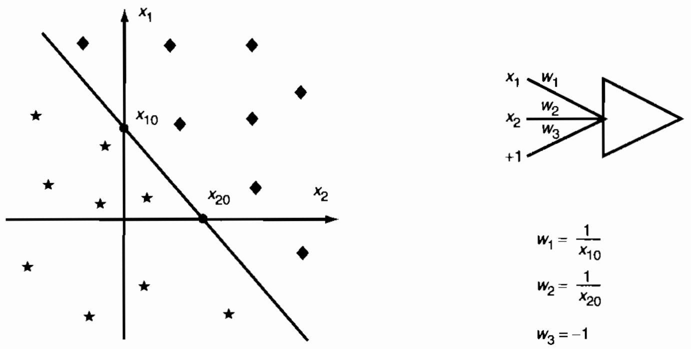

FIGURE 19.18 Illustration of the property of linear separation of patterns in the two-dimensional space by a single neuron.

A single neuron is capable of separating input patterns into two categories, and this separation is linear. For example, for the patterns shown in Fig. 19.18, the separation line is crossing x_1 and x_2 axes at points x_{10} and x_{20}. This separation can be achieved with a neuron having the following weights: $w_1 = 1/x_{10}$, $w_2 = 1/x_{20}$, and $w_3 = -1$. In general for n dimensions, the weights are

$$w_i = \frac{1}{x_{i0}} \quad \text{for} \quad w_{n+1} = -1 \tag{19.8}$$

One neuron can divide only linearly separated patterns. To select just one region in n-dimensional input space, more than $n + 1$ neurons should be used. If more input clusters are to be selected, then the number of neurons in the input (hidden) layer should be properly multiplied. If the number of neurons in the input (hidden) layer is not limited, then all classification problems can be solved using the three-layer network. An example of such a neural network, classifying three clusters in the two-dimensional space, is shown in Fig. 19.19. Neurons in the first hidden layer create the separation lines between input clusters. Neurons in the second hidden layer perform the AND operation, as shown in Fig. 19.13(b). Output neurons perform the OR operation as shown in Fig. 19.13(a), for each category. The linear separation property of neurons makes some problems specially difficult for neural networks, such as exclusive OR, parity computation for several bits, or to separate patterns laying on two neighboring spirals.

The feedforward neural network is also used for nonlinear transformation (mapping) of a multidimensional input variable into another multidimensional variable in the output. In theory, any input-output mapping should be possible if the neural network has enough neurons in hidden layers (size of output layer is set by the number of outputs required). In practice, this is not an easy task. Presently, there is no satisfactory method to define how many neurons should be used in hidden layers. Usually, this is found by the trial-and-error method. In general, it is known that if more neurons are used, more complicated

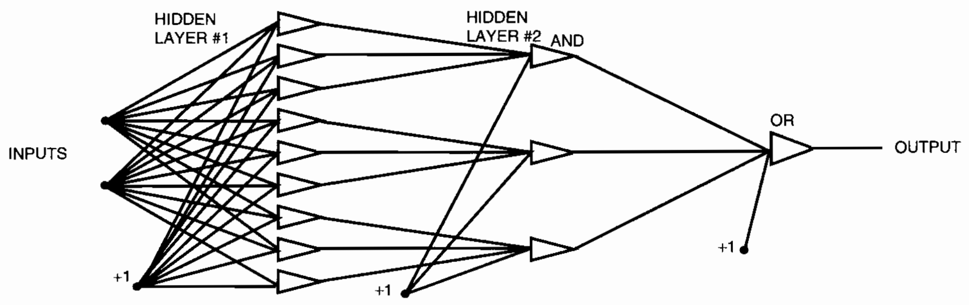

FIGURE 19.19 An example of the three layer neural network with two inputs for classification of three different clusters into one category. This network can be generalized and can be used for solution of all classification problems.

shapes can be mapped. On the other hand, networks with large numbers of neurons lose their ability for generalization, and it is more likely that such networks will also try to map noise supplied to the input.

19.3.4 Learning Algorithms for Neural Networks

Similarly to the biological neurons, the weights in artificial neurons are adjusted during a training procedure. Various learning algorithms were developed, and only a few are suitable for multilayer neuron networks. Some use only local signals in the neurons, others require information from outputs; some require a supervisor who knows what outputs should be for the given patterns, and other unsupervised algorithms need no such information. Common learning rules are described in the following sections.

Hebbian Learning Rule

The Hebb (1949) learning rule is based on the assumption that if two neighbor neurons must be activated and deactivated at the same time, then the weight connecting these neurons should increase. For neurons operating in the opposite phase, the weight between them should decrease. If there is no signal correlation, the weight should remain unchanged. This assumption can be described by the formula

$$\Delta w_{ij} = c x_i o_j \tag{19.9}$$

where

w_{ij} = weight from ith to jth neuron
c = learning constant
x_i = signal on the ith input
o_j = output signal

The training process starts usually with values of all weights set to zero. This learning rule can be used for both soft and hard threshold neurons. Since desired responses of neurons is not used in the learning procedure, this is the **unsupervised learning** rule. The absolute values of the weights are usually proportional to the learning time, which is undesired.

Correlation Learning Rule

The correlation learning rule is based on a similar principle as the Hebbian learning rule. It assumes that weights between simultaneously responding neurons should be largely positive, and weights between neurons with opposite reaction should be largely negative.

Contrary to the Hebbian rule, the correlation rule is the **supervised learning**. Instead of actual response o_j, the desired response d_j is used for the weight change calculation

$$\Delta w_{ij} = c x_i d_j \tag{19.10}$$

This training algorithm usually starts with initialization of weights to zero values.

Instar Learning Rule

If input vectors and weights are normalized, or they have only binary bipolar values (-1 or $+1$), then the *net* value will have the largest positive value when the weights and the input signals are the same. Therefore, weights should be changed only if they are different from the signals

$$\Delta w_i = c(x_i - w_i) \tag{19.11}$$

Note, that the information required for the weight is taken only from the input signals. This is a very local and unsupervised learning algorithm.

Winner Takes All (WTA)

The WTA is a modification of the instar algorithm where weights are modified only for the neuron with the highest *net* value. Weights of remaining neurons are left unchanged. Sometimes this algorithm is modified in such a way that a few neurons with the highest net values are modified at the same time. Although this is an unsupervised algorithm because we do not know what are desired outputs, there is a need for "judge" or "supervisor" to find a winner with a largest net value. The WTA algorithm, developed by Kohonen (1982), is often used for automatic clustering and for extracting statistical properties of input data.

Outstar Learning Rule

In the outstar learning rule, it is required that weights connected to a certain node should be equal to the desired outputs for the neurons connected through those weights

$$\Delta w_{ij} = c(d_j - w_{ij}) \tag{19.12}$$

where d_j is the desired neuron output and c is small learning constant, which further decreases during the learning procedure. This is the supervised training procedure because desired outputs must be known. Both instar and outstar learning rules were developed by Grossberg (1969).

Widrow-Hoff LMS Learning Rule

Widrow and Hoff (1960, 1962) developed a supervised training algorithm which allows training a neuron for the desired response. This rule was derived so that the square of the difference between the net and output value is minimized.

$$Error_j = \sum_{p=1}^{P} (net_{jp} - d_{jp})^2 \tag{19.13}$$

where:

$Error_j$ = error for jth neuron
P = number of applied patterns
d_{jp} = desired output for jth neuron when pth pattern is applied
net = given by Eq. (19.2).

This rule is also known as the least mean square (LMS) rule. By calculating a derivative of Eq. (19.13) with respect to w_{ij}, a formula for the weight change can be found,

$$\Delta w_{ij} = cx_i \sum_{p=1}^{P} (d_{jp} - net_{jp}) \tag{19.14}$$

Note that weight change Δw_{ij} is a sum of the changes from each of the individual applied patterns. Therefore, it is possible to correct the weight after each individual pattern was applied. This process is known as *incremental updating*; *cumulative updating* is when weights are changed after all patterns have been applied. Incremental updating usually leads to a solution faster, but it is sensitive to the order in which patterns are applied. If the learning constant c is chosen to be small, then both methods give the same result. The LMS rule works well for all types of activation functions. This rule tries to enforce the *net* value to be equal to desired value. Sometimes this is not what the observer is looking for. It is usually not important what the *net* value is, but it is important if the *net* value is positive or negative. For example, a very large *net* value with a proper sign will result in correct output and in large error as defined by Eq. (19.13) and this may be the preferred solution.

Linear Regression

The LMS learning rule requires hundreds or thousands of iterations, using formula (19.14), before it converges to the proper solution. Using the linear regression rule, the same result can be obtained in only one step.

Considering one neuron and using vector notation for a set of the input patterns $\boldsymbol{X}$ applied through weight vector $\boldsymbol{w}$, the vector of *net* values $\boldsymbol{net}$ is calculated using

$$\boldsymbol{X}\boldsymbol{w} = \boldsymbol{net} \tag{19.15}$$

where

$\boldsymbol{X}$ = rectangular array $(n+1) \times p$
n = number of inputs
p = number of patterns

Note that the size of the input patterns is always augmented by one, and this additional weight is responsible for the threshold (see Fig. 19.3(b)). This method, similar to the LMS rule, assumes a linear activation function, and so the *net* values $\boldsymbol{net}$ should be equal to desired output values $\boldsymbol{d}$

$$\boldsymbol{X}\boldsymbol{w} = \boldsymbol{d} \tag{19.16}$$

Usually $p > n + 1$, and the preceding equation can be solved only in the least mean square error sense. Using the vector arithmetic, the solution is given by

$$\boldsymbol{w} = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{d} \tag{19.17}$$

When traditional method is used the set of p equations with $n + 1$ unknowns Eq. (19.16) has to be converted to the set of $n + 1$ equations with $n + 1$ unknowns

$$\boldsymbol{Y}\boldsymbol{w} = \boldsymbol{z} \tag{19.18}$$

where elements of the $\boldsymbol{Y}$ matrix and the $\boldsymbol{z}$ vector are given by

$$y_{ij} = \sum_{p=1}^{P} x_{ip}x_{jp} \qquad z_i = \sum_{p=1}^{P} x_{ip}d_p \tag{19.19}$$

Weights are given by Eq. (19.17) or they can be obtained by a solution of Eq. (19.18).

Delta Learning Rule

The LMS method assumes linear activation function $\boldsymbol{net} = \boldsymbol{o}$, and the obtained solution is sometimes far from optimum, as is shown in Fig. 19.20 for a simple two-dimensional case, with four patterns belonging to two categories. In the solution obtained using the LMS algorithm, one pattern is misclassified. If error is defined as

$$Error_j = \sum_{p=1}^{P}(o_{jp} - d_{jp})^2 \tag{19.20}$$

Then the derivative of the error with respect to the weight w_{ij} is

$$\frac{d\,Error_j}{dw_{ij}} = 2\sum_{p=1}^{P}(o_{jp} - d_{jp})\frac{df(net_{jp})}{d\,net_{jp}}x_i \tag{19.21}$$

since $o = f(net)$ and the *net* is given by Eq. (19.2). Note that this derivative is proportional to the derivative of the activation function $f'(net)$. Thus, this type of approach is possible only for continuous activation

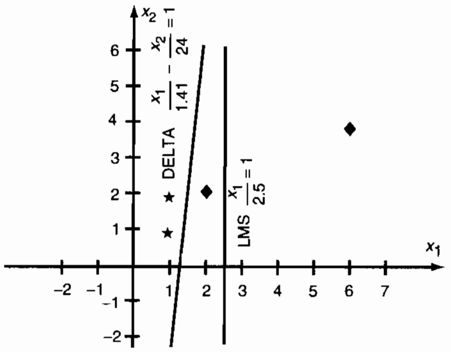

FIGURE 19.20 An example with a comparison of results obtained using LMS and delta training algorithms. Note that LMS is not able to find the proper solution.

functions and this method cannot be used with hard activation functions (19.4) and (19.5). In this respect the LMS method is more general. The derivatives most common continuous activation functions are

$$f' = o(1 - o) \tag{19.22}$$

for the unipolar (Eq. (19.6)) and

$$f' = 0.5(1 - o^2) \tag{19.23}$$

for the bipolar (Eq. (19.7)).

Using the cumulative approach, the neuron weight w_{ij} should be changed with a direction of gradient

$$\Delta w_{ij} = c\, x_i \sum_{p=1}^{P} (d_{jp} - o_{jp}) f'_{jp} \tag{19.24}$$

in case of the incremental training for each applied pattern

$$\Delta w_{ij} = c x_i f'_j (d_j - o_j) \tag{19.25}$$

the weight change should be proportional to input signal x_i, to the difference between desired and actual outputs $d_{jp} - o_{jp}$, and to the derivative of the activation function f'_{jp}. Similar to the LMS rule, weights can be updated in both the incremental and the cumulative methods. In comparison to the LMS rule, the delta rule always leads to a solution close to the optimum. As it is illustrated in Fig. 19.20, when the delta rule is used, all four patterns are classified correctly.

Error Backpropagation Learning

The delta learning rule can be generalized for multilayer networks. Using an approach similiar to the delta rule, the gradient of the global error can be computed with respect to each weight in the network. Interestingly,

$$\Delta w_{ij} = c x_i f'_j E_j \tag{19.26}$$

where

c = learning constant
x_i = signal on the ith neuron input
f'_j = derivative of activation function

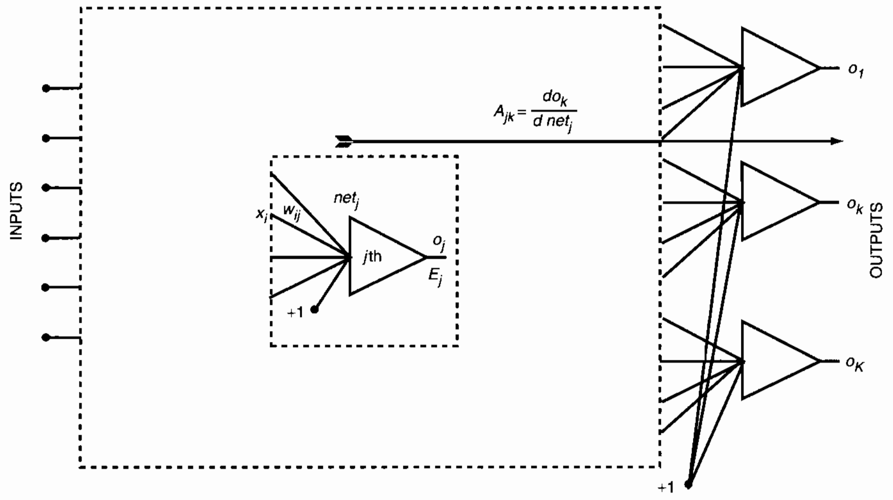

FIGURE 19.21 Illustration of the concept of gain computation in neural networks.

The cumulative error E_j on neuron output is given by

$$E_j = \frac{1}{f'_j} \sum_{k=1}^{K} (o_k - d_k) A_{jk} \tag{19.27}$$

where K is the number of network outputs A_{jk} is the small signal gain from the input of jth neuron to the kth network output, as Fig. 19.21 shows. The calculation of the backpropagating error starts at the output layer and cumulative errors are calculated layer by layer to the input layer. This approach is not practical from the point of view of hardware realization. Instead, it is simpler to find signal gains from the input of the jth neuron to each of the network outputs (Fig. 19.21). In this case, weights are corrected using

$$\Delta w_{ij} = c x_i \sum_{k=1}^{K} (o_k - d_k) A_{jk} \tag{19.28}$$

Note that this formula is general, regardless of if neurons are arranged in layers or not. One way to find gains A_{jk} is to introduce an incremental change on the input of the jth neuron and observe the change in the kth network output. This procedure requires only forward signal propagation, and it is easy to implement in a hardware realization. Another possible way is to calculate gains through each layer and then find the total gains as products of layer gains. This procedure is equally or less computationally intensive than a calculation of cumulative errors in the error backpropagation algorithm.

The backpropagation algorithm has a tendency for oscillation. To smooth the process, the weights increment Δw_{ij} can be modified according to Rumelhart, Hinton, and Wiliams (1986)

$$w_{ij}(n+1) = w_{ij}(n) + \Delta w_{ij}(n) + \alpha \Delta w_{ij}(n-1) \tag{19.29}$$

or according to Sejnowski and Rosenberg (1987)

$$w_{ij}(n+1) = w_{ij}(n) + (1-\alpha)\Delta w_{ij}(n) + \alpha \Delta w_{ij}(n-1) \tag{19.30}$$

where α is the momentum term.

The backpropagation algorithm can be significantly sped up, when, after finding components of the gradient, weights are modified along the gradient direction until a minimum is reached. This process can be carried on without the necessity of a computationally intensive gradient calculation at each step. The new gradient components are calculated once a minimum is obtained in the direction of the previous gradient.

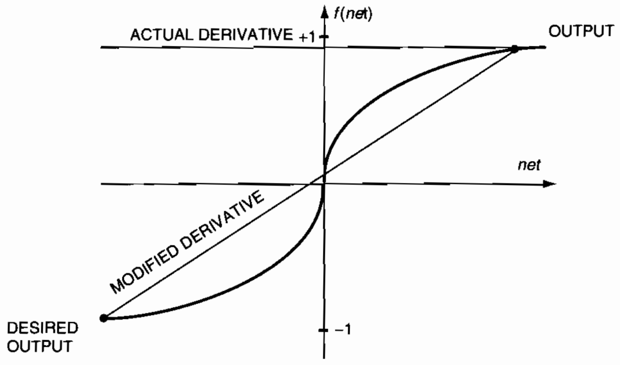

FIGURE 19.22 Illustration of the modified derivative calculation for faster convergency of the error backpropagation algorithm.

This process is only possible for cumulative weight adjustment. One method of finding a minimum along the gradient direction is the *tree step process* of finding error for three points along gradient direction and then, using a parabola approximation, jump directly to the minimum. The fast learning algorithm using the described approach was proposed by Fahlman (1988) and is known as the *quickprop*.

The backpropagation algorithm has many disadvantages, which lead to very slow convergency. One of the most painful is that in the backpropagation algorithm, the learning process almost perishes for neurons responding with the maximally wrong answer. For example, if the value on the neuron output is close to +1 and desired output should be close to −1, then the neuron gain $f'(net) \approx 0$ and the error signal cannot backpropagate, and so the learning procedure is not effective. To overcome this difficulty, a modified method for derivative calculation was introduced by Wilamowski and Torvik (1993). The derivative is calculated as the slope of a line connecting the point of the output value with the point of the desired value, as shown in Fig. 19.22.

$$f_{\text{modif}} = \frac{o_{\text{desired}} - o_{\text{actual}}}{net_{\text{desired}} - net_{\text{actual}}} \tag{19.31}$$

Note that for small errors, Eq. (19.31) converges to the derivative of activation function at the point of the output value. With an increase of system dimensionality, the chances for local minima decrease. It is believed that the described phenomenon, rather than a trapping in local minima, is responsible for convergency problems in the error backpropagation algorithm.

19.3.5 Special Feedforward Networks

The multilayer backpropagation network, as shown in Fig. 19.17, is a commonly used feedforward network. This network consists of neurons with the sigmoid type continuous activation function presented in Fig. 19.16(c) and Fig. 19.16(d). In most cases, only one hidden layer is required, and the number of neurons in the hidden layer are chosen to be proportional to the problem complexity. The number of neurons in the hidden layer is usually found by a trial-and-error process. The training process starts with all weights randomized to small values, and the error backpropagation algorithm is used to find a solution. When the learning process does not converge, the training is repeated with a new set of randomly chosen weights. Nguyen and Widrow (1990) proposed an experimental approach for the two-layer network weight initialization. In the second layer, weights are randomly chosen in the range from −0.5 to +0.5. In the first layer, initial weights are calculated from

$$w_{ij} = \frac{\beta z_{ij}}{\|z_j\|}; \qquad w_{(n+1)j} = \text{random}(-\beta, +\beta) \tag{19.32}$$

where z_{ij} is the random number from -0.5 to $+0.5$ and the scaling factor β is given by

$$\beta = 0.7P^{\frac{1}{N}} \tag{19.33}$$

where n is the number of inputs and N is the number of hidden neurons in the first layer. This type of weight initialization usually leads to faster solutions.

For adequate solutions with backpropagation networks, typically many tries are required with different network structures and different initial random weights. It is important that the trained network gains a generalization property. This means that the trained network also should be able to handle correctly patterns that were not used for training. Therefore, in the training procedure, often some data are removed from the training patterns and then these patterns are used for verification. The results with backpropagation networks often depend on luck. This encouraged researchers to develop feedforward networks, which can be more reliable. Some of those networks are described in the following sections.

Functional Link Network

One-layer neural networks are relatively easy to train, but these networks can solve only linearly separated problems. One possible solution for nonlinear problems presented by Nilsson (1965) and elaborated by Pao (1989) using the functional link network is shown in Fig. 19.23. Using nonlinear terms with initially determined functions, the actual number of inputs supplied to the one-layer neural network is increased. In the simplest case, nonlinear elements are higher order terms of input patterns. Note that the functional link network can be treated as a one-layer network, where additional input data are generated off line using nonlinear transformations. The learning procedure for one-layer is easy and fast. Figure 19.24 shows an XOR problem solved using functional link networks. Note that when the functional link approach is used, this difficult problem becomes a trivial one. The problem with the functional link network is that proper selection of nonlinear elements is not an easy task. In many practical cases, however, it is not difficult to predict what kind of transformation of input data may linearize the problem, and so the functional link approach can be used.

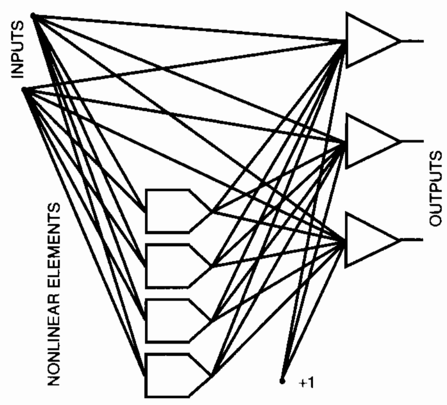

FIGURE 19.23 The functional link network.

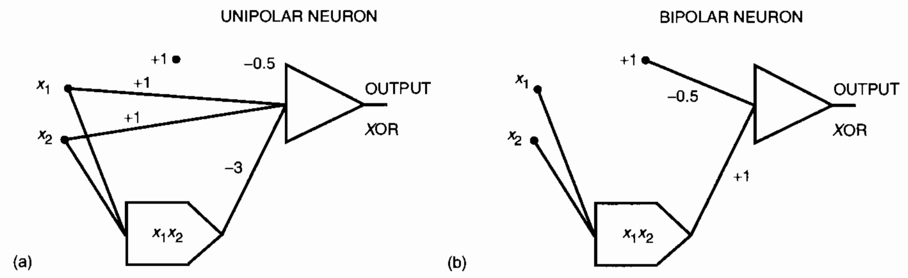

FIGURE 19.24 Functional link networks for solution of the XOR problem: (a) using unipolar signals, (b) using bipolar signals.

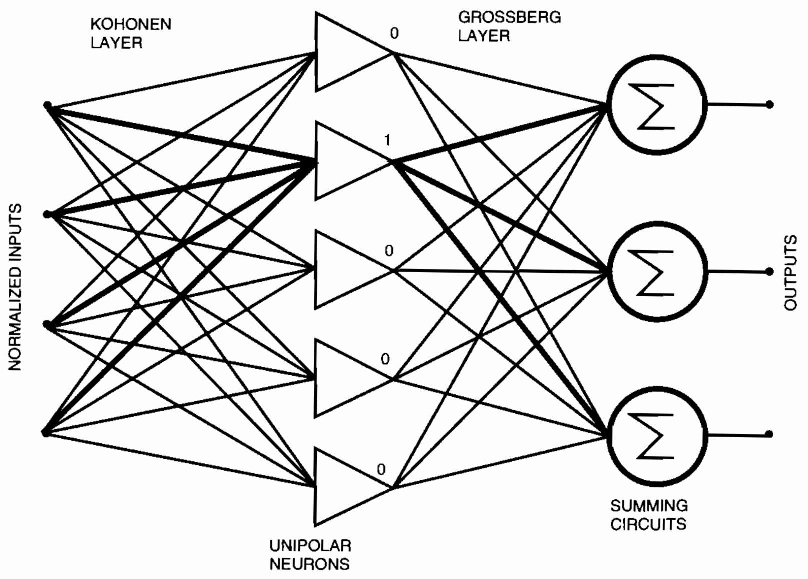

FIGURE 19.25 The counterpropagation network.

Feedforward Version of the Counterpropagation Network

The *counterpropagation network* was originally proposed by Hecht-Nilsen (1987). In this section a modified feedforward version as described by Zurada (1992) is discussed. This network, which is shown in Fig. 19.25, requires numbers of hidden neurons equal to the number of input patterns, or more exactly, to the number of input clusters. The first layer is known as the Kohonen layer with unipolar neurons. In this layer only one neuron, the winner, can be active. The second is the Grossberg outstar layer. The Kohonen layer can be trained in the unsupervised mode, but that need not be the case. When binary input patterns are considered, then the input weights must be exactly equal to the input patterns. In this case,

$$net = \boldsymbol{x}^t\boldsymbol{w} = [n - 2HD(\boldsymbol{x}, \boldsymbol{w})] \tag{19.34}$$

where

n = number of inputs
$\boldsymbol{w}$ = weights
$\boldsymbol{x}$ = input vector
$HD(\boldsymbol{w}, \boldsymbol{x})$ = *Hamming distance* between input pattern and weights

For a neuron in the input layer to be reacting just for the stored pattern, the threshold value for this neuron should be

$$w_{(n+1)} = -(n - 1) \tag{19.35}$$

If it is required that the neuron must also react for similar patterns, then the threshold should be set to $w_{n+1} = -[n - (1 + HD)]$, where HD is the Hamming distance defining the range of similarity. Since for a given input pattern only one neuron in the first layer may have the value of 1 and remaining neurons have 0 values, the weights in the output layer are equal to the required output pattern.

The network, with unipolar activation functions in the first layer, works as a lookup table. When the linear activation function (or no activation function at all) is used in the second layer, then the network also can be considered as an analog memory. For the address applied to the input as a binary vector, the stored set of analog values, as weights in the second layer, can be accurately recovered. The feedforward counterpropagation network may also use analog inputs, but in this case all input data should be normalized.

$$\boldsymbol{w}_i = \hat{\boldsymbol{x}}_i = \frac{\boldsymbol{x}_i}{\|\boldsymbol{x}_i\|} \tag{19.36}$$

The counterpropagation network is very easy to design. The number of neurons in the hidden layer is equal to the number of patterns (clusters). The weights in the input layer are equal to the input patterns, and the weights in the output layer are equal to the output patterns. This simple network can be used for rapid prototyping. The counterpropagation network usually has more hidden neurons than required. However, such an excessive number of hidden neurons are also used in more sophisticated feedforward networks such as the *probabilistic neural network* (PNN) Specht (1990) or the *general regression neural networks* (GRNN) Specht (1992).

WTA Architecture

The winner takes all network was proposed by Kohonen (1988). This is basically a one-layer network used in the unsupervised training algorithm to extract a statistical property of the input data (Fig. 19.26(a)). At the first step, all input data are normalized so that the length of each input vector is the same and, usually, equal to unity (Eq. (19.36)). The activation functions of neurons are unipolar and continuous. The learning process starts with a weight initialization to small random values. During the learning process the weights are changed only for the neuron with the highest value on the output-the winner.

$$\Delta \boldsymbol{w}_w = c(\boldsymbol{x} - \boldsymbol{w}_w) \tag{19.37}$$

where

$\boldsymbol{w}_w$ = weights of the winning neuron
$\boldsymbol{x}$ = input vector
c = learning constant

Usually, this single-layer network is arranged into a two-dimensional layer shape, as shown in Fig. 19.26(b). The hexagonal shape is usually chosen to secure strong interaction between neurons. Also, the algorithm is modified in such a way that not only the winning neuron but also neighboring neurons are allowed for the weight change. At the same time, the learning constant c in Eq. (19.37) decreases with the distance from the winning neuron. After such a unsupervised training procedure, the Kohonen layer is able to organize data into clusters. Output of the Kohonen layer is then connected to the one- or two-layer feedforward network with the error backpropagation algorithm. This initial data organization in the WTA layer usually leads to rapid training of the following layer or layers.

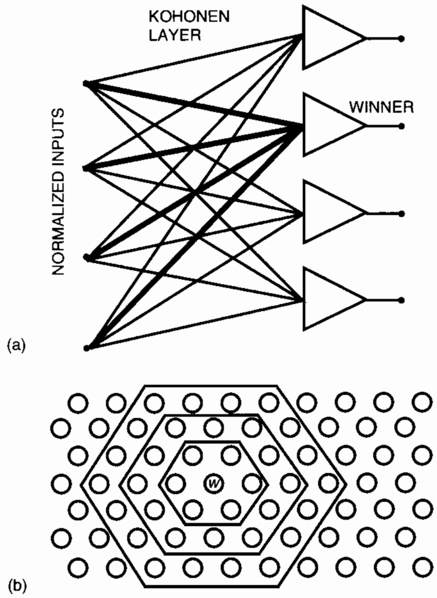

FIGURE 19.26 A winner takes all architecture for cluster extracting in the unsupervised training mode: (a) Network connections, (b) single-layer network arranged into a hexagonal shape.

Cascade Correlation Architecture

The cascade correlation architecture was proposed by Fahlman and Lebiere (1990). The process of network building starts with a one-layer neural network and hidden neurons are added as needed. The network architecture is shown in Fig. 19.27. In each training step, a new hidden neuron is added and its weights are adjusted to maximize the magnitude

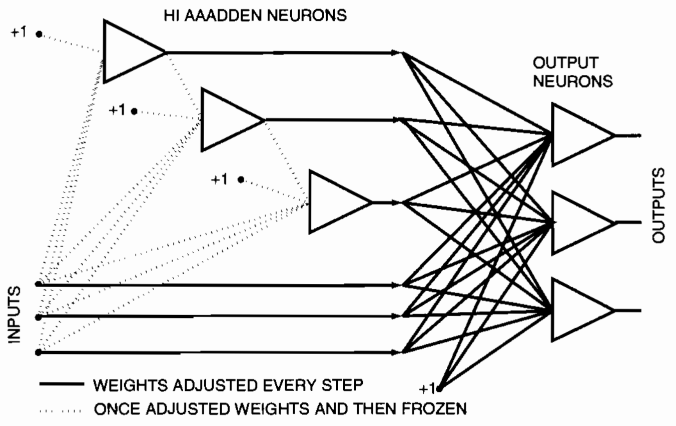
HI AAADDEN NEURONS
+1
+1
+1
OUTPUT NEURONS
OUTPUTS
INPUTS
+1
WEIGHTS ADJUSTED EVERY STEP
ONCE ADJUSTED WEIGHTS AND THEN FROZEN

FIGURE 19.27 The cascade correlation architecture.

of the correlation between the new hidden neuron output and the residual error signal on the network output to be eliminated. The correlation parameter S must be maximized.

$$S = \sum_{o=1}^{O} \left| \sum_{p=1}^{P} (V_p - \bar{V})(E_{po} - \bar{E}_o) \right| \tag{19.38}$$

where

O = number of network outputs
P = number of training patterns
V_p = output on the new hidden neuron
E_{po} = error on the network output

$\bar{V}$ and $\bar{E}_o$ are average values of V_p and E_{po}, respectively. By finding the gradient, $\delta S/\delta w_i$, the weight adjustment for the new neuron can be found as

$$\Delta w_i = \sum_{o=1}^{O} \sum_{p=1}^{P} \sigma_o (E_{po} - \bar{E}_o) f_p' x_{ip} \tag{19.39}$$

where

σ_o = sign of the correlation between the new neuron output value and network output
f_p' = derivative of activation function for pattern p
x_{ip} = input signal

The output neurons are trained using the delta or quickprop algorithms. Each hidden neuron is trained just once and then its weights are frozen. The network learning and building process is completed when satisfactory results are obtained.

Radial Basis Function Networks

The structure of the radial basis network is shown in Fig. 19.28. This type of network usually has only one hidden layer with special neurons. Each of these neurons responds only to the inputs signals close to the stored pattern. The output signal h_i of the ith hidden neuron is computed using formula

$$h_i = \exp\left(-\frac{\|\mathbf{x} - \mathbf{s}_i\|^2}{2\sigma^2}\right) \tag{19.40}$$

where

$\boldsymbol{x}$ = input vector

$\boldsymbol{s}_i$ = stored pattern representing the center of the i cluster

σ_i = radius of the cluster

Note that the behavior of this "neuron" significantly differs form the biological neuron. In this "neuron," excitation is not a function of the weighted sum of the input signals. Instead, the distance between the input and stored pattern is computed. If this distance is zero then the neuron responds with a maximum output magnitude equal to one. This neuron is capable of recognizing certain patterns and generating output signals that are functions of a similarity. Features of this neuron are much more powerful than a neuron used in the backpropagation networks. As a consequence, a network made of such neurons is also more powerful.

If the input signal is the same as a pattern stored in a neuron, then this neuron responds with 1 and remaining neurons have 0 on the output, as is illustrated in Fig. 19.28. Thus, output signals are exactly equal to the weights coming out from the active neuron. This way, if the number of neurons in the hidden layer is large, then any input/output mapping can be obtained. Unfortunately, it may also happen that for some patterns several neurons in the first layer will respond with a nonzero signal. For a proper approximation, the sum of all signals from the hidden layer should be equal to one. To meet this requirement, output signals are often normalized as shown in Fig. 19.28.

The radial-based networks can be designed or trained. Training is usually carried out in two steps. In the first step, the hidden layer is usually trained in the unsupervised mode by choosing the best patterns for cluster representation. An approach, similar to that used in the WTA architecture can be used. Also in this step, radii σ_i must be found for a proper overlapping of clusters.

The second step of training is the error backpropagation algorithm carried on only for the output layer. Since this is a supervised algorithm for one layer only, the training is very rapid, 100–1000 times faster than in the backpropagation multilayer network. This makes the radial basis-function network very attractive. Also, this network can be easily modeled using computers, however, its hardware implementation would be difficult.

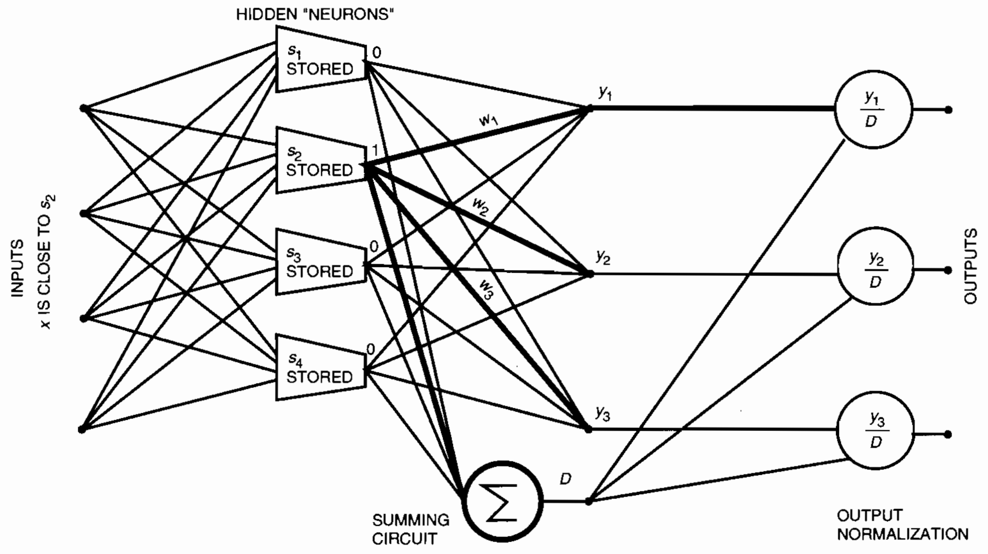

FIGURE 19.28 A typical structure of the radial basis function network.

19.3.6 Recurrent Neural Networks

In contrast to feedforward neural networks, with **recurrent networks** neuron outputs can be connected with their inputs. Thus, signals in the network can continuously circulate. Until recently, only a limited number of recurrent neural networks were described.

Hopfield Network

The single-layer recurrent network was analyzed by Hopfield (1982). This network, shown in Fig. 19.29, has unipolar hard threshold neurons with outputs equal to 0 or 1. Weights are given by a symmetrical square matrix **W** with zero elements ($w_{ij} = 0$ for $i = j$) on the main diagonal. The stability of the system is usually analyzed by means of the *energy function*

$$E = -\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} W_{ij} v_i v_j \tag{19.41}$$

It has been proved that during signal circulation the energy E of the network decreases and the system converges to the stable points. This is especially true when the values of system outputs are updated in the asynchronous mode. This means that at a given cycle, only one random output can be changed to the required values. Hopfield also proved that those stable points which the system converges can be programmed by adjusting the weights using a modified Hebbian rule,

$$\Delta w_{ij} = \Delta w_{ji} = (2v_i - 1)(2v_j - 1)c \tag{19.42}$$

Such memory has limited storage capacity. Based on experiments, Hopfield estimated that the maximum number of stored patterns is $0.15N$, where N is the number of neurons.

Later the concept of energy function was extended by Hopfield (1984) to one-layer recurrent networks having neurons with continuous activation functions. These types of networks were used to solve many optimization and linear programming problems.

Autoassociative Memory

Hopfield (1984) extended the concept of his network to autoassociative memories. In the same network structure as shown in Fig. 19.29, the bipolar hard-threshold neurons were used with outputs equal to −1

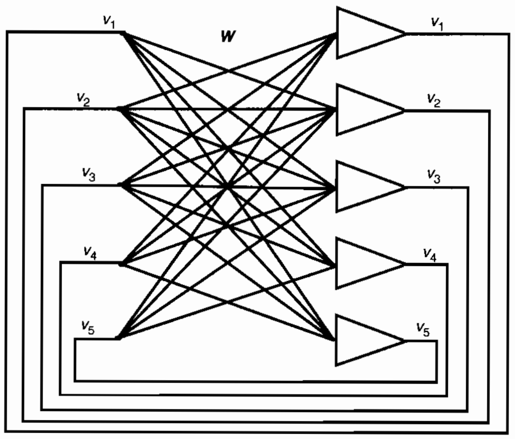

FIGURE 19.29 A Hopfield network or autoassociative memory.

or $+1$. In this network, pattern $\boldsymbol{s}_m$ are stored into the weight matrix $\boldsymbol{W}$ using the autocorrelation algorithm

$$\boldsymbol{W} = \sum_{m=1}^{M} \mathbf{s}_m \mathbf{s}_m^T - M\boldsymbol{I} \tag{19.43}$$

where M is the number of stored pattern and $\boldsymbol{I}$ is the unity matrix. Note that $\boldsymbol{W}$ is the square symmetrical matrix with elements on the main diagonal equal to zero (w_{ji} for $i = j$). Using a modified formula (19.42), new patterns can be added or subtracted from memory. When such memory is exposed to a binary bipolar pattern by enforcing the initial network states, after signal circulation the network will converge to the closest (most similar) stored pattern or to its complement. This stable point will be at the closest minimum of the energy

$$E(\boldsymbol{v}) = -\frac{1}{2}\boldsymbol{v}^T \mathbf{W} \boldsymbol{v} \tag{19.44}$$

Like the Hopfield network, the autoassociative memory has limited storage capacity, which is estimated to be about $M_{\max} = 0.15N$. When the number of stored patterns is large and close to the memory capacity, the network has a tendency to converge to spurious states, which were not stored. These spurious states are additional minima of the energy function.

Bidirectional Associative Memories (BAM)

The concept of the autoassociative memory was extended to bidirectional associative memories (BAM) by Kosko (1987, 1988). This memory, shown in Fig. 19.30, is able to associate pairs of the patterns $\boldsymbol{a}$ and $\boldsymbol{b}$. This is the two-layer network with the output of the second layer connected directly to the input of the first layer. The weight matrix of the second layer is $\boldsymbol{W}^T$ and $\boldsymbol{W}$ for the first layer. The rectangular weight matrix $\boldsymbol{W}$ is obtained as a sum of the cross-correlation matrixes

$$\boldsymbol{W} = \sum_{m=1}^{M} \boldsymbol{a}_m \boldsymbol{b}_m \tag{19.45}$$

where M is the number of stored pairs, and $\boldsymbol{a}_m$ and $\boldsymbol{b}_m$ are the stored vector pairs. If the nodes $\boldsymbol{a}$ or $\boldsymbol{b}$ are initialized with a vector similar to the stored one, then after signal circulations, both stored patterns $\boldsymbol{a}_m$ and $\boldsymbol{b}_m$ should be recovered. The BAM has limited memory capacity and memory corruption problems similar to the autoassociative memory. The BAM concept can be extended for association of three or more vectors.

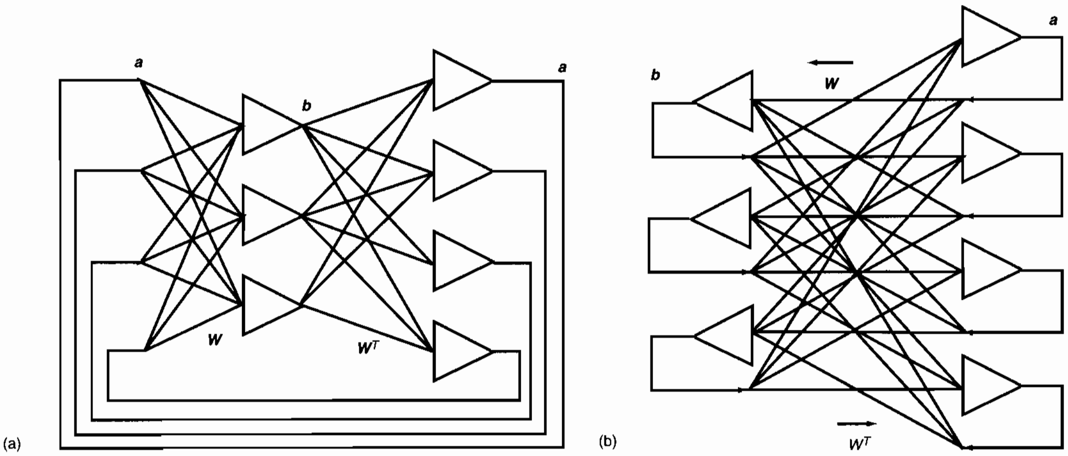

FIGURE 19.30 An example of the bi-directional autoassociative memory: (a) drawn as a two-layer network with circulating signals, (b) drawn as two-layer network with bi-directional signal flow.

19.3.7 Fuzzy Systems

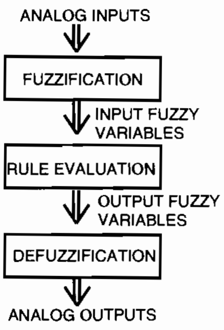

FIGURE 19.31 The block diagram of the fuzzy controller.

The main applications of neural networks are related to the nonlinear mapping of n-dimensional input variables into m-dimensional output variables. Such a function is often required in control systems, where for specific measured variables certain control variables must be generated. Another approach for nonlinear mapping of one set of variables into another set of variables is the *fuzzy controller*. The principle of operation of the fuzzy controller significantly differs from neural networks. The block diagram of a fuzzy controller is shown in Fig. 19.31. In the first step, analog inputs are converted into a set of fuzzy variables. In this step, for each analog input, 3–9 fuzzy variables typically are generated. Each fuzzy variable has an analog value between zero and one. In the next step, a fuzzy logic is applied to the input fuzzy variables and a resulting set of output variables is generated. In the last step, known as *defuzzification*, from a set of output fuzzy variables, one or more output analog variables are generated, which are used as control variables.

Fuzzification

The purpose of fuzzification is to convert an analog variable input into a set of fuzzy variables. For higher accuracy, more fuzzy variables will be chosen. To illustrate the fuzzification process, consider that the input variable is the temperature and is coded into five fuzzy variables: cold, cool, normal, warm, and hot. Each fuzzy variable should obtain a value between zero and one, which describes a *degree of association* of the analog input (temperature) within the given fuzzy variable. Sometimes, instead of the term degree of association, the term *degree of membership* is used. The process of fuzzification is illustrated in Fig. 19.32. Using Fig. 19.32 we can find the degree of association of each fuzzy variable with the given temperature. For example, for a temperature of 57°F, the following set of fuzzy variables is obtained: (0, 0.5, 0.2, 0, 0), and for $T = 80$°F it is (0, 0, 0.25, 0.7, 0). Usually only one or two fuzzy variables have a value other than zero. In the example, trapezoidal functions are used for calculation of the degree of association. Various different functions such as triangular or Gaussian, can also be used, as long as the computed value is in the range from zero to one. Each membership function is described by only three or four parameters, which have to be stored in memory.

For proper design of the fuzzification stage, certain practical rules should be used:

- Each point of the input analog variable should belong to at least one and no more than two membership functions.
- For overlapping functions, the sum of two membership functions must not be larger than one. This also means that overlaps must not cross the points of maximum values (ones).
- For higher accuracy, more membership functions should be used. However, very dense functions lead to frequent system reaction and sometimes to system instability.

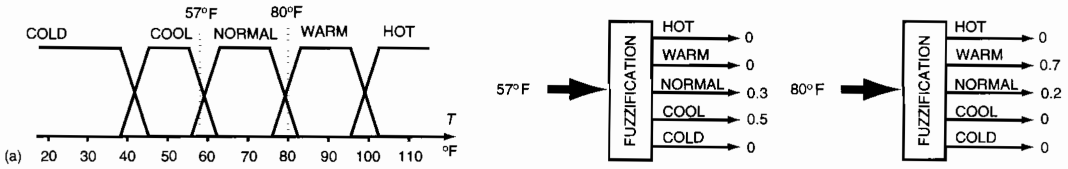

FIGURE 19.32 Fuzzification process: (a) typical membership functions for the fuzzification and the defuzzification processes, (b) example of converting a temperature into fuzzy variables.

Rule Evaluation

Contrary to Boolean logic where variables can have only binary states, in fuzzy logic all variables may have any values between zero and one. The fuzzy logic consists of the same basic $\wedge$ - AND, $\vee$-OR, and NOT operators

$$\begin{aligned} A \wedge B \wedge C \quad &\Longrightarrow \min\{A, B, C\}\text{—smallest value of } A \text{ or } B \text{ or } C \\ A \vee B \vee C \quad &\Longrightarrow \max\{A, B, C\}\text{—largest value of } A \text{ or } B \text{ or } C \\ \bar{A} \quad &\Longrightarrow 1\ 1 - A \qquad \text{—one minus value of } A \end{aligned}$$

For example $0.1 \wedge 0.7 \wedge 0.3 = 0.1, 0.1 \vee 0.7 \vee 0.3 = 0.7$, and $\bar{0.3} = 0.7$. These rules are also known as *Zadeh* AND, OR, and NOT operators (Zadeh, 1965). Note that these rules are true also for classical binary logic.

Fuzzy rules are specified in the *fuzzy table* as it is shown for a given system. Consider a simple system with two analog input variables x and y, and one output variable z. The goal is to design a fuzzy system generating z as $f(x, y)$. After fuzzification, the analog variable x is represented by five fuzzy variables: x_1, x_2, x_3, x_4, x_5 and an analog variable y is represented by three fuzzy variables: y_1, y_2, y_3. Assume that an analog output variable is represented by four fuzzy variables: z_1, z_2, z_3, z_4. The key issue of the design process is to set proper output fuzzy variables z_k for all combinations of input fuzzy variables, as is shown in the table in Fig. 19.33. The designer has to specify many rules such as if inputs are represented by fuzzy variables x_i and y_j, then the output should be represented by fuzzy variable z_k. Once the fuzzy table is specified, the fuzzy logic computation proceeds in two steps. First each field of the fuzzy table is filled with intermediate fuzzy variables t_{ij}, obtained from AND operator $t_{ij} = \min\{x_i, y_j\}$, as shown in Fig. 19.33(b). This step is independent of the required rules for a given system. In the second step, the OR (max) operator is used to compute each output fuzzy variable z_k. In the given example in Fig. 19.33, $z_1 = \max\{t_{11}, t_{12}, t_{21}, t_{41}, t_{51}\}$, $z_2 = \max\{t_{13}, t_{31}, t_{42}, t_{52}\}$, $z_3 = \max\{t_{22}, t_{23}, t_{43}\}$, $z_4 = \max\{t_{32}, t_{34}, t_{53}\}$. Note that the formulas depend on the specifications given in the fuzzy table shown in Fig. 19.33(a).

	y_1	y_2	y_3
x_1	z_1	z_1	z_2
x_2	z_1	z_3	z_3
x_3	z_1	z_3	z_4
x_4	z_2	z_3	z_4
x_5	z_2	z_3	z_4

(a)

	y_1	y_2	y_3
x_1	t_{11}	t_{12}	t_{13}
x_2	t_{21}	t_{22}	t_{23}
x_3	t_{31}	t_{32}	t_{33}
x_4	t_{41}	t_{42}	t_{43}
x_5	t_{51}	t_{52}	t_{53}

(b)

FIGURE 19.33 Fuzzy tables: (a) table with fuzzy rules, (b) table with the intermediate variables t_{ij}.

Defuzzification

As a result of fuzzy rule evaluation, each analog output variable is represented by several fuzzy variables. The purpose of defuzzification is to obtain analog outputs. This can be done by using a membership function similar to that shown in Fig. 19.32. In the first step, fuzzy variables obtained from rule evaluations are used to modify the membership function employing the formula

$$\mu_k^*(z) = \min\{\mu_k(z), z_k\} \tag{19.46}$$

For example, if the output fuzzy variables are: 0, 0.2, 0.7, 0.0, then the modified membership functions have shapes shown by the thick line in Fig. 19.34. The analog value of the z variable is found as a *center of gravity* of modified membership functions $\mu_k^*(z)$,

$$z_{\text{analog}} = \frac{\left(\sum_{k=1}^{n} \int_{-\infty}^{+\infty} \mu_k{}^*(z) z dz\right)}{\left(\sum_{k=1}^{n} \int_{-\infty}^{+\infty} \mu_k{}^*(z) dz\right)} \tag{19.47}$$

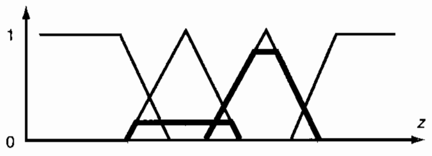

FIGURE 19.34 Illustration of the defuzzification process.

In the case where shapes of the output membership functions $\mu_k(z)$ are the same, the equation can be simplified to

$$z_{\text{analog}} = \frac{\left(\sum_{k=1}^{n} z_k z c_k\right)}{\left(\sum_{k=1}^{n} z_k\right)} \tag{19.48}$$

where

n = number of membership functions of z_{analog} output variable
z_k = fuzzy output variables obtained from rule evaluation
zc_k = analog values corresponding to the center of kth membership function.

Equation (19.47) is usually too complicated to be used in a simple microcontroller based system; therefore, in practical cases, Eq. (19.48) is used more frequently.

19.3.8 Design Example

Consider the design of a simple fuzzy controller for a sprinkler system. The sprinkling time is a function of humidity and temperature. Four membership functions are used for the temperature, three for humidity, and three for the sprinkle time, as shown in Fig. 19.35. Using intuition, the fuzzy table can be developed, as shown in Fig. 19.36(a).

Assume a temperature of 60°F and 70% humidity. Using the membership functions for temperature and humidity the following fuzzy variables can be obtained for the temperature: (0, 0.2, 0.5, 0), and for

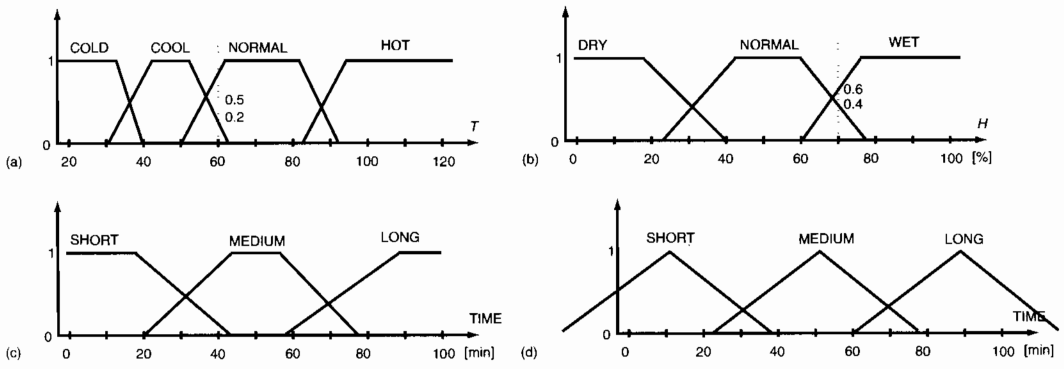

FIGURE 19.35 Membership functions for the presented example: (a) and (b) are membership functions for input variables, (c) and (d) are two possible membership functions for the output variable.

the humidity: (0, 0.4, 0.6). Using the min operator, the fuzzy table can be now filled with temporary fuzzy variables, as shown in Fig. 19.36(b). Note that only four fields have nonzero values. Using fuzzy rules, as shown in Fig. 19.36(a), the max operator can be applied in order to obtain fuzzy output variables: short $\rightarrow o_1 = \max\{0, 0, 0.2, 0.5, 0\} = 0.5$, medium $\rightarrow o_2 = \max\{0, 0, 0.2, 0.4, 0\} = 0.4$, long $\rightarrow o_3 = \max\{0, 0\} = 0$. Using Eq. (19.47) and Fig. 19.35(c) a sprinkle time of 28 min is determined. When the simplified approach is used with Eq. (19.46) and Fig. 19.35(d), then sprinkle time is 27 min.

(a)

	DRY	NORMAL	WET
COLD	M	S	S
COOL	M	M	S
WARM	L	M	S
HOT	L	M	S

(b)

	DRY	NORMAL	WET
	0	0.4	0.6
COLD 0	0	0	0
COOL 0.2	0	0.2	0.2
WARM 0.5	0	0.4	0.5
HOT 0	0	0	0

FIGURE 19.36 Fuzzy tables: (a) fuzzy rules for the design example, (b) fuzzy temporary variables for the design example.

19.3.9 Genetic Algorithms

The success of the artificial neural networks encouraged researchers to search for other patterns in nature to follow. The power of genetics through evolution was able to create such sophisticated machines as the human being. Genetic algorithms follow the evolution process in nature to find better solutions to some complicated problems. The foundations of genetic algorithms are given by Holland (1975) and Goldberg (1989). After initialization, the steps *selection, reproduction with a crossover*, and *mutation* are repeated for each generation. During this procedure, certain strings of symbols, known as chromosomes, evaluate toward a better solution. The genetic algorithm method begins with coding and an initialization. All significant steps of the genetic algorithm will be explained using a simple example of finding a maximum of the function $(\sin^2 x - 0.5x)^2$ (Fig. 19.37) with the range of x from 0 to 1.6. Note that in this range, the function has a global maximum at $x = 1.309$, and a local maximum at $x = 0.262$.

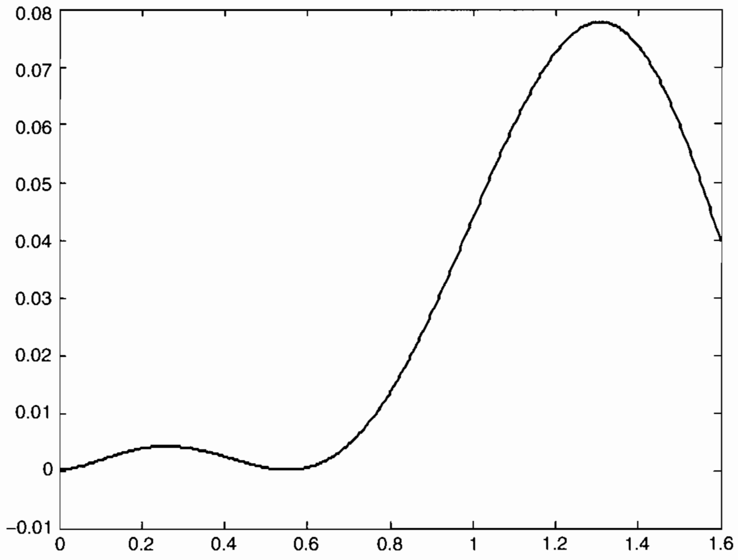

FIGURE 19.37 Function $(\sin^2 x - 0.5x^2)$ for which the minimum must be sought.

TABLE 19.3 Initial Population

String number	String	Decimal value	Variable value	Function value	Fraction of total
1	101101	45	1.125	0.0633	0.2465
2	101000	40	1.000	0.0433	0.1686
3	010100	20	0.500	0.0004	0.0016
4	100101	37	0.925	0.0307	0.1197
5	001010	10	0.250	0.0041	0.0158
6	110001	49	1.225	0.0743	0.2895
7	100111	39	0.975	0.0390	0.1521
8	000100	40	0.100	0.0016	0.0062
Total				0.2568	1.0000

Coding and Initialization

At first, the variable x has to be represented as a string of symbols. With longer strings, the process usually converges faster, so the fewer the symbols for one string field that are used, the better. Although this string may be the sequence of any symbols, the binary symbols 0 and 1 are usually used. In our example, six bit binary numbers are used for coding, having a decimal value of $40x$. The process starts with a random generation of the initial population given in Table 19.3.

Selection and Reproduction

Selection of the best members of the population is an important step in the genetic algorithm. Many different approaches can be used to rank individuals. In this example the ranking function is given. Highest rank has member number 6, and lowest rank has member number 3. Members with higher rank should have higher chances to reproduce. The probability of reproduction for each member can be obtained as a fraction of the sum of all objective function values. This fraction is shown in the last column of Table 19.3. Note that to use this approach, our objective function should always be positive. If it is not, the proper normalization should be introduced at first.

Reproduction

The numbers in the last column of Table 19.3 show the probabilities of reproduction. Therefore, most likely members number 3 and 8 will not be reproduced, and members 1 and 6 may have two or more copies. Using a random reproduction process, the following population, arranged in pairs, could be generated

101101 $\rightarrow$ 45	110001 $\rightarrow$ 49	100101 $\rightarrow$ 37	110001 $\rightarrow$ 49
100111 $\rightarrow$ 39	101101 $\rightarrow$ 45	110001 $\rightarrow$ 49	101000 $\rightarrow$ 40

If the size of the population from one generation to another is the same, two parents should generate two children. By combining two strings, two other strings should be generated. The simplest way to do this is to split in half each of the parent strings and exchange substrings between parents. For example, from parent strings 010100 and 100111, the following child strings will be generated 010111 and 100100. This process is known as the *crossover*. The resultant children are

101111 $\rightarrow$ 47	110101 $\rightarrow$ 53	100001 $\rightarrow$ 33	110000 $\rightarrow$ 48
100101 $\rightarrow$ 37	101001 $\rightarrow$ 41	110101 $\rightarrow$ 53	101001 $\rightarrow$ 41

In general, the string need not be split in half. It is usually enough if only selected bits are exchanged between parents. It is only important that bit positions are not changed.

TABLE 19.4 Population of Second Generation

String number	String	Decimal value	Variable value	Function value	Fraction of total
1	010111	47	1.175	0.0696	0.1587
2	100100	37	0.925	0.0307	0.0701
3	110101	53	1.325	0.0774	0.1766
4	010001	41	1.025	0.0475	0.1084
5	100001	33	0.825	0.0161	0.0368
6	110101	53	1.325	0.0774	0.1766
7	110000	48	1.200	0.0722	0.1646
8	101001	41	1.025	0.0475	0.1084
Total				0.4387	1.0000

Mutation

In the evolutionary process, reproduction is enhanced with mutation. In addition to the properties inherited from parents, offspring acquire some new random properties. This process is known as mutation. In most cases mutation generates low-ranked children, which are eliminated in the reproduction process. Sometimes, however, the mutation may introduce a better individual with a new property. This prevents the process of reproduction from degeneration. In genetic algorithms, mutation usually plays a secondary role. For very high-levels of mutation, the process is similar to random pattern generation, and such a searching algorithm is very inefficient. The mutation rate is usually assumed to be at a level well below 1%. In this example, mutation is equivalent to the random bit change of a given pattern. In this simple case, with short strings and a small population, and with a typical mutation rate of 0.1%, the patterns remain practically unchanged by the mutation process. The second generation for this example is shown in Table 19.4.

Note that two identical highest ranking members of the second generation are very close to the solution $x = 1.309$. The randomly chosen parents for the third generation are

$$\begin{array}{llll} 010111 \rightarrow 47 & 110101 \rightarrow 53 & 110000 \rightarrow 48 & 101001 \rightarrow 41 \\ 110101 \rightarrow 53 & 110000 \rightarrow 48 & 101001 \rightarrow 41 & 110101 \rightarrow 53 \end{array}$$

which produces the following children

$$\begin{array}{llll} 010101 \rightarrow 21 & 110000 \rightarrow 48 & 110001 \rightarrow 49 & 101101 \rightarrow 45 \\ 110111 \rightarrow 55 & 110101 \rightarrow 53 & 101000 \rightarrow 40 & 110001 \rightarrow 49 \end{array}$$

The best result in the third population is the same as in the second one. By careful inspection of all strings from the second or third generation, it may be concluded that using crossover, where strings are always split in half, the best solution 110100 $\rightarrow$52 will never be reached, regardless of how many generations are created. This is because none of the population in the second generation has a substring ending with 100. For such crossover, a better result can be only obtained due to the mutation process, which may require many generations. Better results in the future generation also can be obtained when strings are split in random places. Another possible solution is that only randomly chosen bits are exchanged between parents.

The genetic algorithm almost always leads to a good solution, but sometimes many generations are required. This solution is usually close to global maximum, but not the best. In the case of a smooth function the gradinet methods are converging much faster and to a better solution. GA are much slower, but more robust.

Defining Terms

Backpropagation: Training technique for multilayer neural networks.
Bipolar neuron: Neuron with output signal between -1 and $+1$.

Feedforward network: Network without feedback.
Perceptron: Network with hard threshold neurons.
Recurrent network: Network with feedback.
Supervised learning: Learning procedure when desired outputs are known.
Unipolar neuron: Neuron with output signal between 0 and +1.
Unsupervised learning: Learning procedure when desired outputs are unknown.

References

Fahlman, S.E. 1988. Faster-learning variations on backpropagation: an empirical study. *Proceedings of the Connectionist Models Summer School,* eds. Touretzky, D. Hinton, G., and Sejnowski, T. Morgan Kaufmann, San Mateo, CA.

Fahlman, S.E. and Lebiere, C. 1990. The cascade correlation learning architecture. *Adv. Ner. Inf. Proc. Syst.,* 2, D.S. Touretzky, ed. pp. 524–532. Morgan, Kaufmann, Los Altos, CA.

Goldberg, D.E. 1989. *Genetic Algorithm in Search, Optimization and Machine Learning.* Addison-Wesley, Reading, MA.

Grossberg, S. 1969. Embedding fields: a theory of learning with physiological implications. *Journal of Mathematical Psychology* 6:209–239.

Hebb, D.O. 1949. *The Organization of Behivior, a Neuropsychological Theory.* John Wiley, New York.

Hecht-Nielsen, R. 1987. Counterpropagation networks. *Appl. Opt.* 26(23):4979–4984.

Hecht-Nielsen, R. 1988. Applications of counterpropagation networks. *Neural Networks* 1:131–139.

Holland, J.H. 1975. *Adaptation in Natural and Artificial Systems.* University. of Michigan Press, Ann Arbor, MI.

Hopfield, J.J. 1982. Neural networks and physical systems with emergent collective computation abilities. *Proceedings of the National Academy of Science* 79:2554–2558.

Hopfield, J.J. 1984. Neurons with graded response have collective computational properties like those of two-state neurons. *Proceedings of the National Academy of Science* 81:3088–3092.

Kohonen, T. 1988. The neural phonetic typerater. *IEEE Computer* 27(3):11–22.

Kohonen, T. 1990. The self-organized map. *Proc. IEEE* 78(9):1464–1480.

Kosko, B. 1987. Adaptive bidirectional associative memories. *App. Opt.* 26:4947–4959.

Kosko, B. 1988. Bidirectional associative memories. *IEEE Trans. Sys. Man, Cyb.* 18:49–60.

McCulloch, W.S. and Pitts, W.H. 1943. A logical calculus of the ideas imminent in nervous activity. *Bull. Math. Biophy.* 5:115–133.

Minsky, M. and Papert, S. 1969. *Perceptrons.* MIT Press, Cambridge, MA.

Nilsson, N.J. 1965. *Learning Machines: Foundations of Trainable Pattern Classifiers.* McGraw-Hill, New York.

Nguyen, D. and Widrow, B. 1990. Improving the learning speed of 2-layer neural networks, by choosing initial values of the adaptive weights. *Proceedings of the International Joint Conference on Neural Networks* (San Diego), CA, June.

Pao, Y.H. 1989. *Adaptive Pattern Recognition and Neural Networks.* Addison-Wesley, Reading, MA.

Rosenblatt, F. 1958. The perceptron: a probabilistic model for information storage and organization in the brain. *Psych. Rev.* 65:386–408.

Rumelhart, D.E., Hinton, G.E., and Williams, R.J. 1986. Learning internal representation by error propagation. *Parallel Distributed Processing.* Vol. 1, pp. 318–362. MIT Press, Cambrige, MA.

Sejnowski, T.J. and Rosenberg, C.R. 1987. Parallel networks that learn to pronounce English text. *Complex Systems* 1:145–168.

Specht, D.F. 1990. Probalistic neural networks. *Neural Networks* 3:109–118.

Specht, D.F. 1992. General regression neural network. *IEEE Trans. Neural Networks* 2:568–576.

Wasserman, P.D. 1989. *Neural Computing Theory and Practice.* Van Nostrand Reinhold, New York.

Werbos, P. 1974. Beyond regression: new tools for prediction and analysis in behavioral sciences. Ph.D. diss., Harvard Universtiy.

Widrow, B. and Hoff, M.E., 1960. Adaptive switching circuits. 1960 IRE Western Electric Show and Convention Record, Part 4 (Aug. 23):96–104.

Widrow, B. 1962. Generalization and information storage in networks of adaline Neurons. In *Self-organizing Systems,* Jovitz, M.C., Jacobi, G.T. and Goldstein, G. eds. pp. 435–461. Sparten Books, Washington, D.C.

Wilamowski, M. and Torvik, L. 1993. Modification of gradient computation in the backpropagation algorithm. *ANNIE'93—Artificial Neural Networks in Engineering.* November 14–17, 1993, St. Louis, Missou.; also in Dagli, C.H. ed. 1993. *Intelligent Engineering Systems Through Artificial Neural Networks* Vol. 3, pp. 175–180. ASME Press, New York.

Zadeh, L.A. 1965. Fuzzy sets. *Information and Control* 8:338–353.

Zurada, J. M. 1992. *Introduction to Artificial Neural Systems.* West Publishing Company, St. Paul. MN.

19.4 Machine Vision

David A. Kosiba and Rangachar Kasturi

19.4.1 Introduction

Machine vision, also known as computer vision, is the scientific discipline whereby explicit, meaningful descriptions of physical objects from the world around us are constructed from their images. Machine vision produces measurements or abstractions from geometrical properties and comprises techniques for estimating features in images, relating feature measurements to the geometry of objects in space, and interpreting this geometric information. This overall task is generally termed *image understanding.*

The goal of a machine vision system is to create a model of the real world from images or sequences of images. Since images are two-dimensional projections of the three-dimensional world, the information is not directly available and must be recovered. This recovery requires the inversion of a many-to-one mapping. To reclaim this information, however, knowledge about the objects in the scene and projection geometry is required.

At every stage in a machine vision system decisions must be made requiring knowledge of the application or goal. Emphasis in machine vision systems is on maximizing the automatic operation at each stage, and these systems should use knowledge to accomplish this. The knowledge used by the system includes models of features, image formation, objects, and relationships among objects. Without explicit use of knowledge, machine vision systems can be designed to work only in a very constrained environment for limited applications. To provide more flexibility and robustness, knowledge is represented explicitly and is directly used by the system. Knowledge is also used by the designers of machine vision systems in many implicit as well as explicit forms. In fact, the efficacy and efficiency of a system is usually governed by the quality of the knowledge used by the system. Difficult problems are often solvable only by identifying the proper source of knowledge and appropriate mechanisms to use it in the system.

Relationship to Other Fields

Machine vision is closely related to many other disciplines and incorporates various techniques adopted from many well-established fields, such as physics, mathematics, and psychology. Techniques developed from many areas are used for recovering information from images. In this section, we briefly describe some very closely related fields.

Image processing techniques generally transform images into other images; the task of information recovery is left to a human user. This field includes topics such as image enhancement, image compression, and correcting blurred or out-of-focus images (Gonzalez and Woods, 1982). On the other hand, machine vision algorithms take images as inputs but produce other types of outputs, such as representations for the object contours in an image or the motion of objects in a series of images. Thus, emphasis in machine vision is on recovering information automatically, with minimal interaction with a human. Image processing algorithms are useful in the early stages of a machine vision system. They are usually used to enhance particular information and suppress noise.