4

Supervised Learning Neural Networks

In this chapter, we describe the basic concepts, notation, and basic learning algorithms for supervised neural networks that will be of great use for solving pattern recognition problems in the following chapters of this book. The chapter is organized as follows: backpropagation for feedforward networks, radial basis networks, adaptive neuro-fuzzy inference systems (ANFIS) and applications. First, we give a brief review of the basic concepts of neural networks and the basic backpropagation learning algorithm. Second, we give a brief description of the momentum and adaptive momentum learning algorithms. Third, we give a brief review of the radial basis neural networks. Finally, we end the chapter with a description of the adaptive neuro-fuzzy inference system (ANFIS) methodology. We consider this material necessary to understand the new methods for pattern recognition that will be presented in the final chapters of this book.

4.1 Backpropagation for Feedforward Networks

This section describes the architectures and learning algorithms for adaptive networks, a unifying framework that subsumes almost all kinds of neural network paradigms with supervised learning capabilities. An adaptive network, as the name indicates, is a network structure consisting of a number of nodes connected through directional links. Each node represents a process unit, and the links between nodes specify the causal relationship between the connected nodes. The learning rule specifies how the parameters (of the nodes) should be updated to minimize a prescribed error measure.

The basic learning rule of the adaptive network is the well-known steepest descent method, in which the gradient vector is derived by successive invocations of the chain rule. This method for systematic calculation of the gradient vector was proposed independently several times, by Bryson and Ho (1969), Werbos (1974), and Parker (1982). However, because research on artificial neural networks was still in its infancy at those times, these researchers' early work failed to receive the attention it deserved. In 1986, Rumelhart et al. used the same procedure to find the gradient in a multilayer neural network. Their procedure was called "backpropagation learning rule", a name which is now widely known because the work of Rumelhart inspired enormous interest in research on neural networks. In this section, we introduce Werbos's original backpropagation method for finding gradient vectors and also present improved versions of this method.

4.1.1 The Backpropagation Learning Algorithm

The procedure for finding a gradient vector in a network structure is generally referred to as "backpropagation" because the gradient vector is calculated in the direction opposite to the flow of the output of each node. Once the gradient is obtained, a number of derivative-based optimization and regression techniques are available for updating the parameters. In particular, if we use the gradient vector in a simple steepest descent method, the resulting learning paradigm is referred to as "backpropagation learning rule". Suppose that a given feedforward adaptive network has L layers and layer l ($l = 0, 1, \ldots, L$) has $N(l)$ nodes. Then the output and function of node i [$i = 1, \ldots, N(l)$] in layer l can be represented as $x_{l,i}$ and $f_{l,i}$, respectively, as shown in Fig. 4.1. Since the output of a node depends on the incoming signals and the parameter set of the node, we have the following general expression for the node function $f_{l,i}$:

$$X_{l,i} = f_{l,i}(x_{i-1,1}, \ldots, x_{l-1,N(l-1),}\ \alpha, \beta, \gamma, \ldots) \tag{4.1}$$

where α, β, γ, etc. are the parameters of this node.

Assuming that the given training data set has P entries, we can define an error measure for the pth ($1 \leq p \leq P$) entry of the training data as the sum of the squared errors:

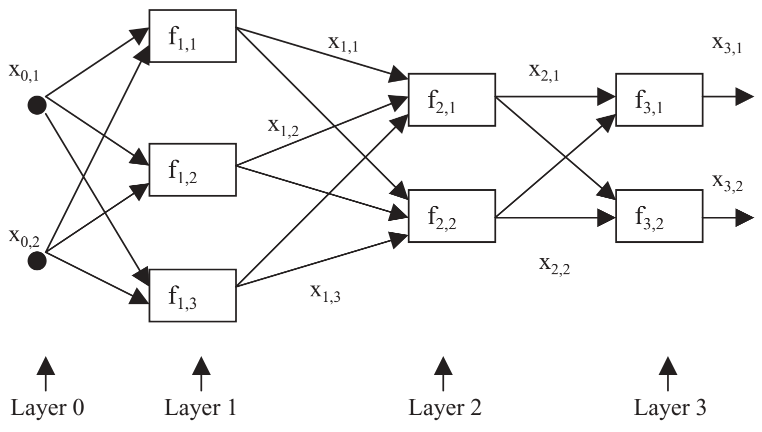


Fig. 4.1. Feedforward adaptive network

$$E_p = \sum_{k=1}^{N(L)} (d_k - x_{L,k})^2 \tag{4.2}$$

where d_k is the kth component of the pth desired output vector and $x_{L,k}$ is the kth component of the actual output vector produced by presenting the pth input vector to the network. Obviously, when E_p is equal to zero, the network is able to reproduce exactly the desired output vector in the pth training data pair. Thus our task here is to minimize an overall error measure, which is defined as $E = \sum E_p$.

We can also define the "error signal" $\varepsilon_{l,i}$ as the derivative of the error measure E_p with respect to the output of the node i in layer l, taking both direct and indirect paths into consideration. Mathematically,

$$\varepsilon_{l,i} = \frac{\partial^+ E_p}{\partial x_{l,i}} \tag{4.3}$$

this expression was called the "ordered derivative" by Werbos (1974). The difference between the ordered derivative and the ordinary partial derivative lies in the way we view the function to be differentiated. For an internal node output $x_{l,i}$, the partial derivative $\partial^+ E_p / \partial x_{l,i}$ is equal to zero, since E_p does not depend on $x_{l,i}$ directly. However, it is obvious that E_p does depend on $x_{l,i}$ indirectly, since a change in $x_{l,i}$ will propagate through indirect paths to the output layer and thus produce a corresponding change in the value of E_p.

The error signal for the ith output node (at layer L) can be calculated directly:

$$\varepsilon_{L,i} = \frac{\partial^+ E_p}{\partial x_{L,i}} = \frac{\partial E_p}{\partial x_{L,i}} \tag{4.4}$$

This is equal to $\varepsilon_{L,i} = -2(d_i - x_{L,i})$ if E_p is defined as in (4.2). For the internal node at the ith position of layer l, the error signal can be derived by the chain rule of differential calculus:

$$\varepsilon_{l,i} = \underbrace{\frac{\partial^+ E_p}{\partial x_{l,i}}}_{\substack{\text{error signal} \\ \text{at layer } l}} = \sum_{m=1}^{N(l+1)} \underbrace{\frac{\partial^+ E_p}{\partial x_{l+1,m}} \frac{\partial f_{l+1,m}}{\partial x_{l,i}}}_{\substack{\text{error signal} \\ \text{at layer } l+1}} = \sum_{m=1}^{M(l+1)} \epsilon_{l+1,m} \frac{\partial f_{l+1,m}}{\partial x_{l,i}} \tag{4.5}$$

where $0 \le l \le L-1$. That is, the error signal of an internal node at layer l can be expressed as a linear combination of the error signal of the nodes at layer $l+1$. Therefore, for any l and i, we can find $\varepsilon_{l,i}$ by first applying (4.4) once to get error signals at the output layer, and then applying (4.5) iteratively until we reach the desired layer l. The underlying procedure is called backpropagation since the error signals are obtained sequentially from the output layer back to the input layer.

The gradient vector is defined as the derivative of the error measure with respect to each parameter, so we have to apply the chain rule again to find the gradient vector. If α is a parameter of the ith node at layer l, we have

$$\frac{\partial^+ E_p}{\partial \alpha} = \frac{\partial^+ E_p}{\partial x_{l,i}} \frac{\partial f_{l,i}}{\partial \alpha} = \varepsilon_{l,i} \frac{\partial f_{l,i}}{\partial \alpha} \tag{4.6}$$

The derivative of the overall error measure E with respect to α is

$$\frac{\partial^+ E}{\partial \alpha} = \sum_{p=1}^{p} \frac{\partial^+ E_p}{\partial \alpha} \tag{4.7}$$

Accordingly, for simple steepest descent (for minimization), the update formula for the generic parameter α is

$$\Delta\alpha = -\eta \frac{\partial^+ E}{\partial \alpha} \tag{4.8}$$

in which η is the "learning rate", which can be further expressed as

$$\eta = \frac{k}{\sqrt{\sum_{\alpha} (\partial E / \partial \alpha)^2}} \tag{4.9}$$

where k is the "step size", the length of each transition along the gradient direction in the parameter space.

There are two types of learning paradigms that are available to suit the needs for various applications. In "off-line learning" (or "batch learning"), the update formula for parameter α is based on (4.7) and the update action takes place only after the whole training data set has been presented-that is, only after each "epoch" or "sweep". On the other hand, in "on-line learning" (or "pattern-by-pattern learning"), the parameters are updated immediately after each input-output pair has been presented, and the update formula is based on (4.6). In practice, it is possible to combine these two learning modes and update the parameter after k training data entries have been presented, where k is between 1 and P and it is sometimes referred to as the "epoch size".

4.1.2 Backpropagation Multilayer Perceptrons

Artificial neural networks, or simply "neural networks" (NNs), have been studied for more than three decades since Rosenblatt first applied single-layer "perceptrons" to pattern classification learning (Rosenblatt, 1962). However, because Minsky and Papert pointed out that single-layer systems were limited and expressed pessimism over multilayer systems, interest in NNs dwindled in the 1970s (Minsky & Papert, 1969). The recent resurgence of interest in the field of NNs has been inspired by new developments in NN learning algorithms (Fahlman & Lebiere, 1990), analog VLSI circuits, and parallel processing techniques (Lippmann, 1987).

Quite a few NN models have been proposed and investigated in recent years. These NN models can be classified according to various criteria, such as

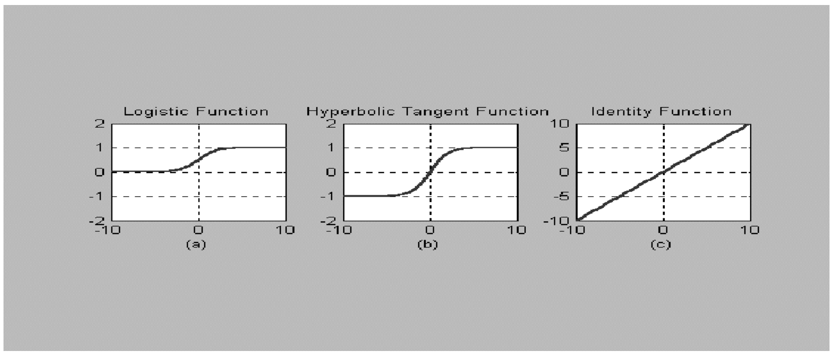


Fig. 4.2. Activation functions for backpropagation MLPs: (**a**) logistic function; (**b**) hyperbolic function; (**c**) identity function

their learning methods (supervised versus unsupervised), architectures (feedforward versus recurrent), output types (binary versus continuous), and so on. In this section, we confine our scope to modeling problems with desired input-output data sets, so the resulting networks must have adjustable parameters that are updated by a supervised learning rule. Such networks are often referred to as "supervised learning" or "mapping networks", since we are interested in shaping the input-output mappings of the network according to a given training data set.

A backpropagation "multilayer perceptron" (MLP) is an adaptive network whose nodes (or neurons) perform the same function on incoming signals; this node function is usually a composite of the weighted sum and a differentiable non-linear activation function, also known as the "transfer function". Figure 4.2 depicts three of the most commonly used activation functions in backpropagation MLPs:

$$\text{Logistic function:} \qquad f(x) = \frac{1}{1+e^{-x}}$$

$$\text{Hyperbolic tangent function:} \qquad f(x) = \tanh(x/2) = \frac{1-e^{-x}}{1+e^{-x}}$$

$$\text{Identity function:} \qquad f(x) = x$$

Both the hyperbolic tangent and logistic functions approximate the signum and step function, respectively, and provide smooth, nonzero derivatives with respect to input signals. Sometimes these two activation functions are referred to as "squashing functions" since the inputs to these functions are squashed to the range [0, 1] or [−1, 1]. They are also called "sigmoidal functions" because their s-shaped curves exhibit smoothness and asymptotic properties.

Backpropagation MLPs are by far the most commonly used NN structures for applications in a wide range of areas, such as pattern recognition, signal processing, data compression and automatic control. Some of the well-known instances of applications include NETtalk (Sejnowski & Rosenberg, 1987), which trained an MLP to pronounce English text, Carnegie Mellon University's ALVINN (Pomerleau, 1991), which used an MLP for steering an autonomous vehicle; and optical character recognition (Sackinger, Boser, Bromley, Lecun & Jackel, 1992). In the following lines, we derive the back-propagation learning rule for MLPs using the logistic function.

The "net input" x of a node is defined as the weighted-sum of the incoming signals plus a bias term. For instance, the net input and output of node j in Fig. 4.3 are

$$\begin{aligned} x_j &= \sum_i w_{ij} x_i + w_j \ , \\ x_j &= f(x_j) = \frac{1}{1+\exp(-x_j)} \ , \end{aligned} \tag{4.10}$$

where x_i is the output of node i located in any one of the previous layers, w_{ij} is the weight associated with the link connecting nodes i and j, and w_j is the bias of node j. Since the weights w_{ij} are actually internal parameters associated with each node j, changing the weights of a node will alter the behavior of the node and in turn alter the behavior of the whole backpropagation MLP.

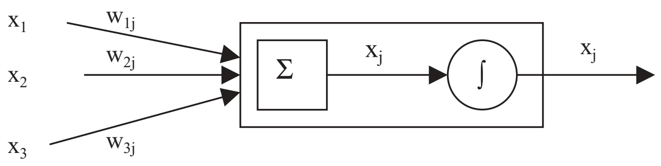


Fig. 4.3. Node j of a backpropagation MLP

Figure 4.4 shows a three-layer backpropagation MLP with three inputs to the input layer, three neurons in the hidden layer, and two output neurons in the output layer. For simplicity, this MLP will be referred to as a 3-3-2 network, corresponding to the number of nodes in each layer.

The "backward error propagation", also known as the "backpropagation" (BP) or the "generalized data rule" (GDR), is explained next. First, a squared error measure for the pth input-output pair is defined as

$$E_p = \sum_k (d_k - x_k)^2 \tag{4.11}$$

where d_k is the desired output for node k, and x_k is the actual output for node k when the input part of the pth data pair presented. To find the gradient vector, an error term ε_i is defined as

$$\varepsilon_i = \frac{\partial^+ E_p}{\partial x_i} \tag{4.12}$$

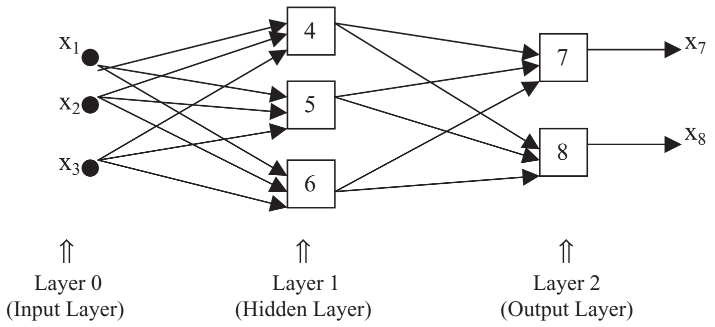


Fig. 4.4. A 3-3-2 backpropagation MLP

By the chain rule, the recursive formula for ε_i can be written as

$$\varepsilon_i = \begin{cases} -2(d_i - x_i)\frac{\partial x_i}{\partial x_i} = -2(d_i - x_i)x_i(1-x_i) & \text{if node } i \text{ is a output node} \\ \frac{\partial x_i}{\partial x_i}\sum_{j,i<j}\frac{\partial^+ E_p}{\partial x_j}\frac{\partial x_j}{\partial x_i} = x_i(1-x_i)\sum_{j,i<j}\varepsilon_j w_{ij} & \text{otherwise} \end{cases} \tag{4.13}$$

where w_{ij} is the connection weight from node i to j; and w_{ij} is zero if there is no direct connection. Then the weight update w_{ki} for on-line (pattern-by-pattern) learning is

$$\Delta w_{ki} = -\eta\frac{\partial^+ E_p}{\partial w_{ki}} = -\eta\frac{\partial^+ E_p}{\partial x_i}\frac{\partial x_i}{\partial w_{ki}} = -\eta\varepsilon_i x_k \tag{4.14}$$

where η is a learning rate that affects the convergence speed and stability of the weights during learning.

For off-line (batch) learning, the connection weight w_{ki} is updated only after presentation of the entire data set, or only after an "epoch":

$$\Delta w_{ki} = -\eta\frac{\partial^+ E}{\partial w_{ki}} = -\eta\sum_p\frac{\partial^+ E_p}{\partial w_{ki}} \tag{4.15}$$

or, in vector form,

$$\Delta \mathbf{w} = -\eta\frac{\partial^+ E}{\partial \mathbf{w}} = -\eta\nabla_{\mathbf{w}}E \tag{4.16}$$

where $E = \sum_p E_p$. This corresponds to a way of using the true gradient direction based on the entire data set.

The approximation power of backpropagation MLPs has been explored by some researchers. Yet there is very little theoretical guidance for determining network size in terms of say, the number of hidden nodes and hidden layers it should contain. Cybenko (1989) showed that a backpropagation MLP, with one hidden layer and any fixed continuous sigmoidal non-linear function, can approximate any continuous function arbitrarily well on a compact set. When used as a binary-valued neural network with the step activation function, a backpropagation MLP with two hidden layers can form arbitrary complex decision regions to separate different classes, as Lippmann (1987) pointed out. For function approximation as well as data classification, two hidden layers may be required to learn a piecewise-continuous function Masters (1993).

Lets consider a simple example to illustrate the backpropagation learning algorithm. We will consider as training data the set of points distributed in the square $[-1, 1] \times [-1, 1]$, which are shown in Fig. 4.5.

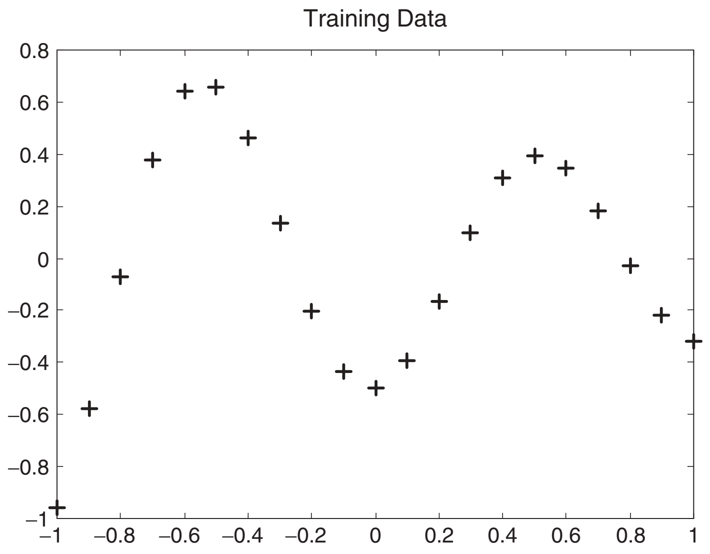


Fig. 4.5. Training data for the backpropagation learning algorithm

We show in Fig. 4.6 the initial approximation of a three layer neural network with 10 neurons in the hidden layer and hyperbolic tangent activation functions. The initial approximation is quite bad because the initial weights of the network are generated randomly. After training with the backpropagation algorithm for 1000 epochs with arrive to the final approximation shown in Fig. 4.7, which has a final sum of squares errors SSE = 0.0264283.

4.1.3 Methods for Speeding Up Backpropagation

One way for to speed up backpropagation is to use the so-called "momentum term" (Rumelhart, Hinton & Williams, 1986):

$$\Delta \mathbf{w} = \eta \nabla_{\mathbf{w}} E + \alpha \Delta \mathbf{w}_{\text{prev}} \tag{4.17}$$

where $\mathbf{w}_{\text{prev}}$ is the previous value of the weights, and the "momentum constant" α, in practice, is usually set to a value between 0.1 and 1. The addition

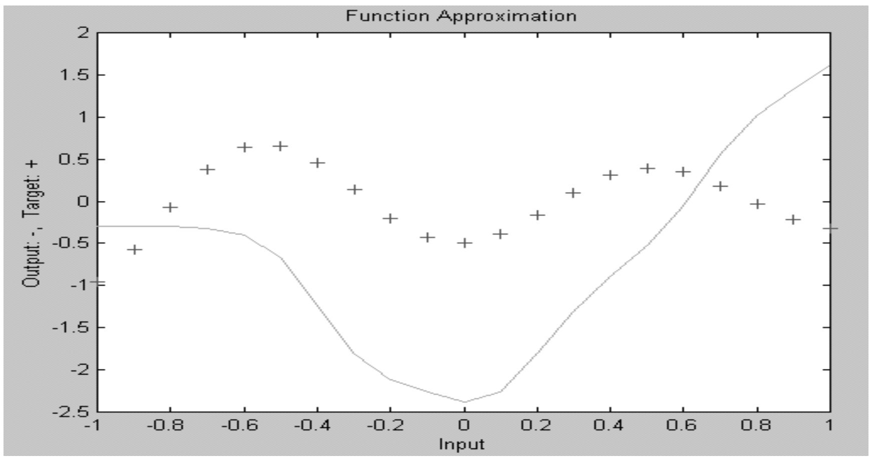


Fig. 4.6. Initial approximation of the backpropagation learning algorithm

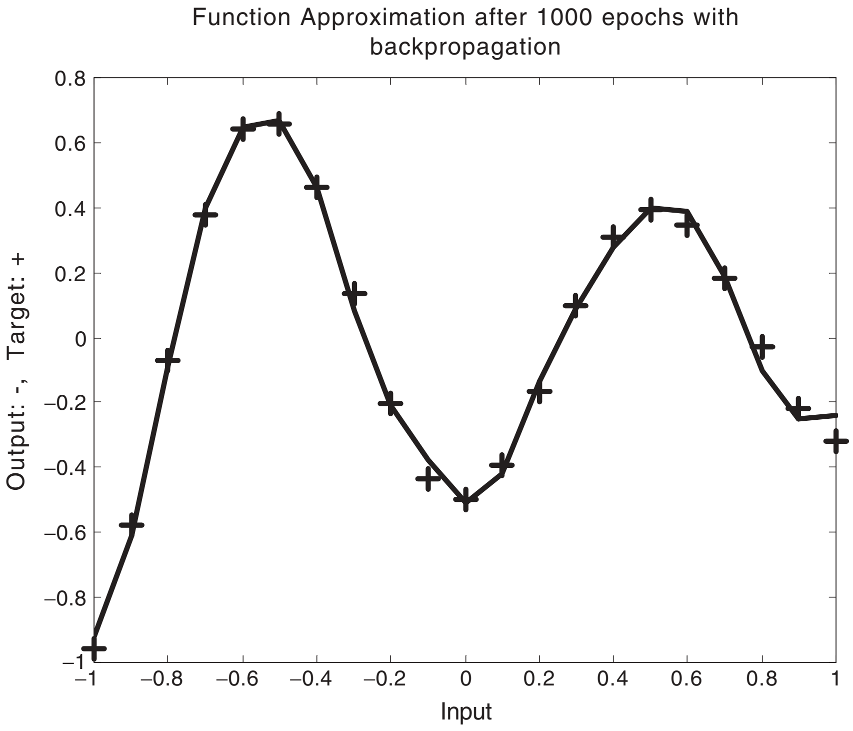


Fig. 4.7. Final approximation backpropagation after 1000 epochs

of the momentum term smoothes weight updating and tends to resist erratic weight changes due to gradient noise or high spatial frequencies in the error surface. However, the use of momentum terms does not always seem to speed up training; it is more or less application dependent.

Another useful technique is normalized weight updating:

$$\Delta \mathbf{w} = -\kappa(\nabla_{\mathbf{w}} E)/(||\nabla_{\mathbf{w}} E||) \tag{4.18}$$

This causes the network's weight vector to move the same Euclidean distance in the weight space with each update, which allows control of the distance κ based on the history of error measures. Other strategies for speeding up backpropagation training include the quick-propagation algorithm Fahlman (1988), backpropagation with adaptive learning rate, backpropagation with momentum and adaptive learning rate, and Levenberg-Marquardt learning algorithm (Jang, Sun & Mizutani, 1997).

Lets consider a simple example to illustrate these methods. We will again use as training data the set of points shown in Fig. 4.5. We will first apply the backpropagation with momentum learning algorithm with the same parameters as before. We show in Fig. 4.8 the initial function approximation and in Fig. 4.9 the final function approximation achieved with backprogration with momentum.

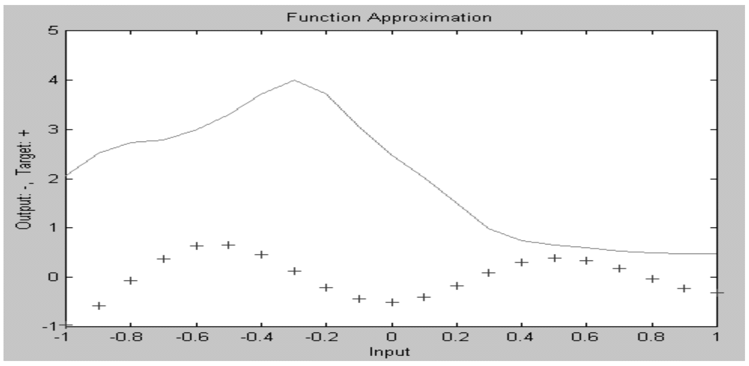


Fig. 4.8. Initial function approximation with backpropagation with momentum

In this case, the final SSE is of 0.0082689, which is lower than the one obtained by simple backpropagation. As a consequence we are achieving a better final approximation with the backpropagation with momentum.

Now we will consider the use of "backpropagation with momentum and adaptive learning rate". In this case, the learning rate is not fixed as in the previous methods, instead it is changed according to the error surface. We will again consider the training data of Fig. 4.5. The initial function approximation is shown in Fig. 4.10. The final function approximation is shown in Fig. 4.11,

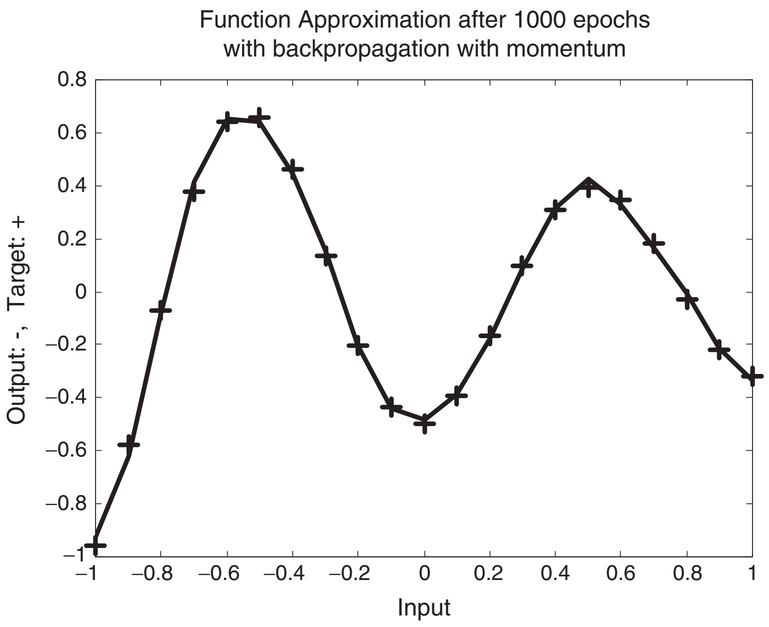


Fig. 4.9. Final function approximation with backpropagation with momentum

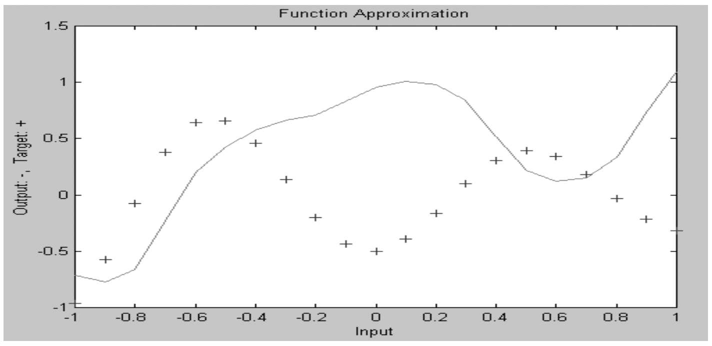


Fig. 4.10. Initial approximation with backpropagation and adaptive learning rate

which is achieved after 1000 epochs with the same network architecture as before.

In this case, the final approximation achieved with the "backpropagation method with adaptive learning rate" is even better because the SSE is of only 0.0045014. This SSE is lower than the ones obtained previously with the other methods.

We will now consider the more complicated problem of forecasting the prices of onion and tomato in the U.S. market. The time series for the prices of these consumer goods show very complicated dynamic behavior, and for

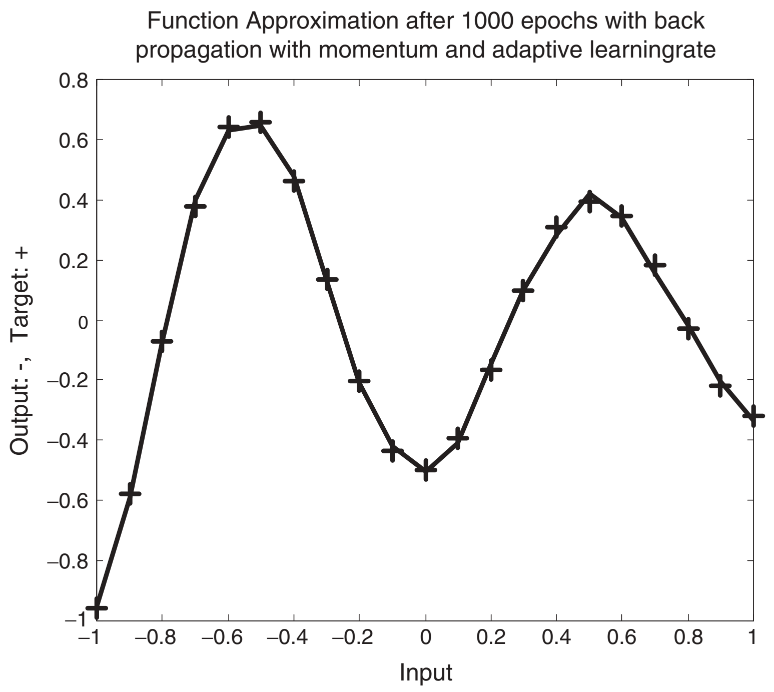


Fig. 4.11. Final approximation with backpropagation and adaptive learning rate

this reason it is interesting to analyze and predict the future prices for these goods. We show in Table 4.1 the time series of onion prices in the U.S. market from 1994 to 2000. We also show in Fig. 4.12 the time series of onion prices in the same period, to give an idea of the complex dynamic behavior of this time series.

We will apply both the neural network approach and the linear regression one to the problem of forecasting the time series of onion prices. Then, we will compare the results of both approaches to select the best one for forecasting.

Table 4.1. Time series of onion prices in the U.S. Market for 1994–2000 period

Month	1994–1995	1995–1996	1996–1997	1997–1998	1998–1999	1999–2000
October	4.13	7.30	6.60	5.63	8.74	6.89
November	4.20	7.50	6.69	5.72	4.66	6.93
December	4.38	7.60	6.30	4.80	8.11	5.63
January	5.16	8.21	6.10	4.75	6.19	3.92
February	5.14	6.00	5.84	4.83	5.97	4.59
March	4.65	7.86	4.73	4.75	4.86	3.76
April	4.89	5.58	5.38	4.75	5.62	3.19

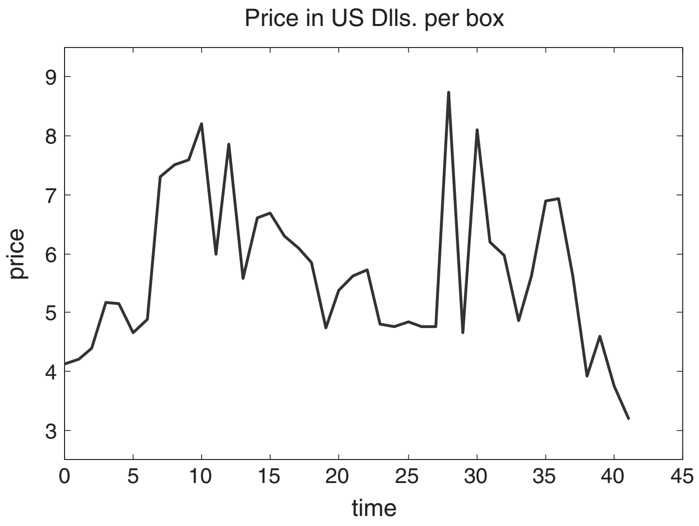


Fig. 4.12. Price in US Dollars per box of onion from October 1994 to April 2000

We also show in Table 4.2 the time series of tomato prices in the U.S. Market from 1994 to 1999.

First, we will describe the results of applying neural networks to the time series of onion prices (Table 4.1). The data given in Table 4.1 was used as training data for feedforward neural networks with two different learning algorithms. The adaptive learning backpropagation with momentum, and the Levenberg-Marquardt training algorithms were used for the neural networks.

We show in Fig. 4.13 the result of training a three layer (85 nodes in the hidden layer) feedforward neural network with the Levenberg-Marquardt learning algorithm. In Fig. 4.13, we can see how the neural network approximates very well the real time series of onion prices over the relevant period of time. Actually, we have seen that the approximating power for the Levenberg-Marquardt learning algorithm is better.

Table 4.2. Time series of tomato prices in the U.S. Market for the 1994–1999 period

Month	1994	1995	1996	1997	1998	1999
June	28.40	32.80	26.80	27.30	25.70	27.80
July	23.30	17.10	23.50	25.40	43.10	20.30
August	27.40	12.70	20.60	25.40	20.40	22.50
September	19.10	17.20	22.40	23.20	26.60	25.30
October	25.70	20.20	27.60	23.30	43.10	18.90
November	28.80	22.70	31.60	41.10	35.80	20.30

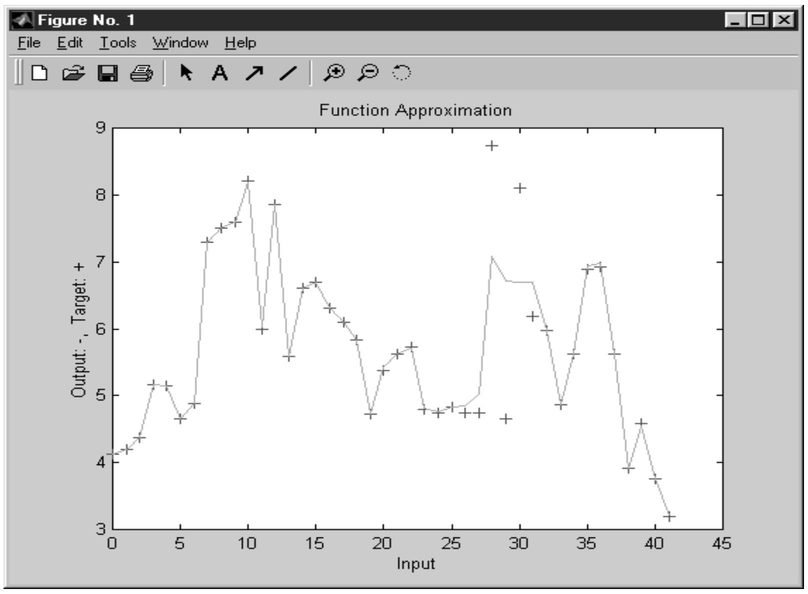


Fig. 4.13. Neural network for onion prices with the Levenberg-Marquardt algorithm

Now we describe the results of applying neural networks to the time series of tomato prices (Table 4.2). We show in Fig. 4.14 the result of training a three layer (85 nodes in hidden layer) neural network with the Levenberg-Marquardt learning algorithm. In Fig. 4.14, we can see how the neural network approximates very well the time series of tomato prices.

We also show in Fig. 4.15 the results of forecasting tomato prices from 2000 to 2010 using the neural network with the Levenberg-Marquardt learning algorithm. Predictions are considered very good by experts at the beginning (first three years or so) but after that they may be not so good. In any case, the results are better than the ones obtained with classical regression methods.

We summarize the above results using neural networks, and the results of using linear regression models in Table 4.3. From Table 4.3 we can see very clearly the advantage of using neural networks for simulating and forecasting the time series of prices. Also, we can conclude from this table that the Levenberg-Marquardt training algorithm is better than backpropagation with momentum. Of course, the reason for this may be that the Levenberg-Marquardt training algorithm, having a variable learning rate is able to adjust to the complicated behavior of the time series. Finally, we have to mention that the problem of time series analysis can be considered as one of pattern recognition, because we are basically finding or learning the patterns in data. For this reason, it is very important to be able to use neural networks in this type of problems. We will concentrate in this book more on pattern

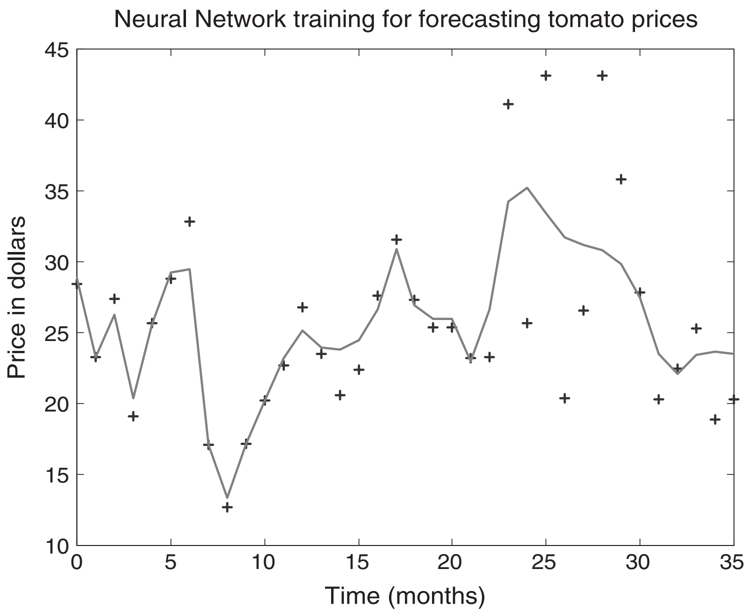


Fig. 4.14. Neural network for tomato prices with the Levenberg-Marquardt algorithm

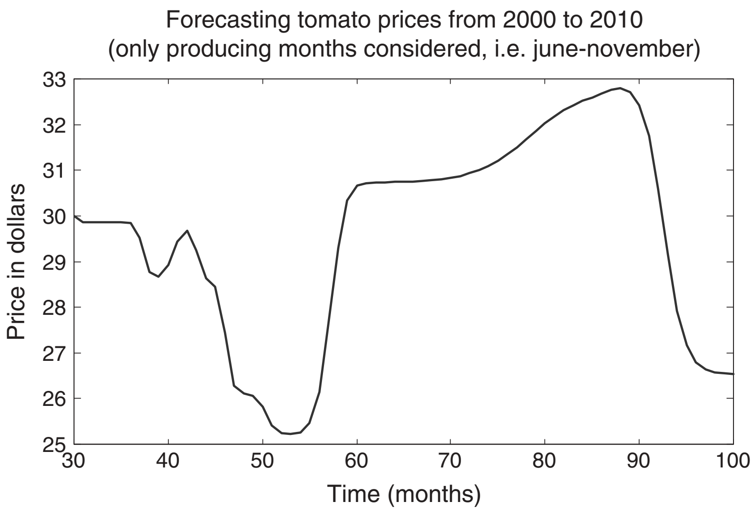


Fig. 4.15. Forecasting tomato prices from 2000 to 2010 with the neural network

Table 4.3. Summary of results for the forecasting methods

	Training Data		Validation with Future Data (2000)	
	SSE Tomato	SSE Onion	SSE Tomato	SSE Onion
NN LM	0.019	0.00039	0.025	0.00055
NN BPM	0.100	0.00219	0.150	0.00190
LR AR(1)	0.250	0.05000	0.551	0.08501
LR AR(2)	0.200	0.01000	0.511	0.03300

recognition for images, but time series analysis and data mining are also important areas of application for intelligent techniques.

4.2 Radial Basis Function Networks

Locally tuned and overlapping receptive fields are well-known structures that have been studied in regions of the cerebral cortex, the visual cortex, and others. Drawing on knowledge of biological receptive fields, Moody and Darken (1989) proposed a network structure that employs local receptive fields to perform function mappings. Similar schemes have been proposed by Powell (1987) and many others in the areas of "interpolation" and "approximation theory"; these schemes are collectively call radial basis function approximations. Here we will call the neural network structure the "radial basis function network" or RBFN.

Figure 4.16 shows a schematic diagram of a RBFN with four receptive field units; the activation level of the ith receptive field unit (or hidden unit) is

$$w_i = R_i(\mathbf{x}) = R_i(||\mathbf{x} - \mathbf{u}_i||/\sigma_i), i = 1, 2, \dots, H \,, \tag{4.19}$$

where $\mathbf{x}$ is a multidimensional input vector, $\mathbf{u}_i$ is a vector with the same dimension as $\mathbf{x}$, H is the number of radial basis functions (or, equivalently, receptive field units), and $R_i(\ )$ is the ith radial basis function with a single maximum at the origin. There are no connection weights between the input layer and the hidden layer. Typically, $R_i(\ )$ is a Gaussian function

$$R_i(\mathbf{x}) = \exp\left[-(||\mathbf{x} - \mathbf{u}_i||^2)/2\sigma_i^2\right] \tag{4.20}$$

or a logistic function

$$R_i(\mathbf{x}) = 1/[1 + \exp\left[(||\mathbf{x} - \mathbf{u}_i||^2)/\sigma_i^2\right] \tag{4.21}$$

Thus, the activation level of radial basis function wi computed by the ith hidden unit is maximum when the input vector $\mathbf{x}$ is at the center $\mathbf{u}_i$ of that unit.

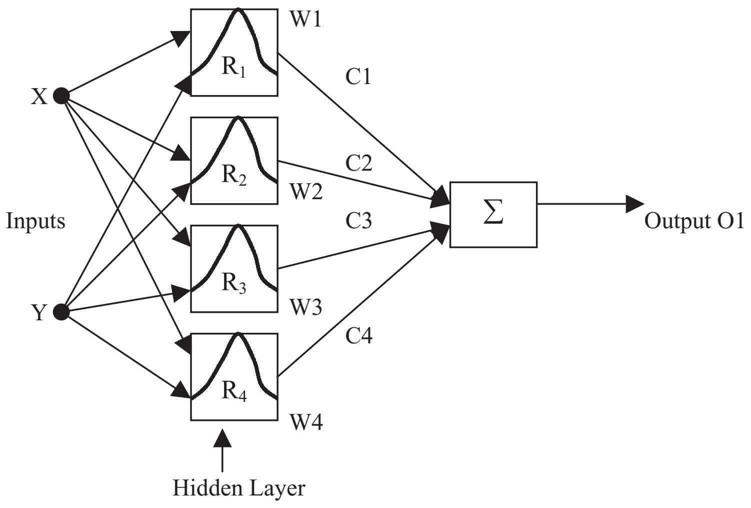


Fig. 4.16. Single-output RBFN that uses weighted sum

The output of an RBFN can be computed in two ways. In the simpler method, as shown in Fig. 4.12, the final output is the weighted sum of the output value associated with each receptive field:

$$d(\mathbf{x}) = \sum_{i=1}^{H} c_i w_i = \sum_{i=1}^{H} c_i R_i(\mathbf{x}) \tag{4.22}$$

where c_i is the output value associated with the ith receptive field. We can also view c_i as the connection weight between the receptive field i and the ouput unit. A more complicated method for calculating the overall output is to take the weighted average of the output associated with each receptive field:

$$d(\mathbf{x}) = \left(\sum_{i=1}^{H} c_i w_i\right) \Bigg/ \left(\sum_{i=1}^{H} w_i\right) \tag{4.23}$$

Weighted average has a higher degree of computational complexity, but it has the advantage that points in the areas of overlap between two or more receptive fields will have a well-interpolated overall output between the outputs of the overlapping receptive fields.

For representation purposes, if we change the radial basis function $R_i(x)$ in each node of layer 2 in Fig. 4.16 to its normalized counterpart $R_i(x)/\sum_i R_i(x)$, then the overall output is specified by (4.23). A more explicit representation is the shown in Fig. 4.17, where the division of the weighted sum $(\sum_{i=1} c_i\ w_i)$ by the activation total $(\sum_{i=1} w_i)$ is indicated in the division node in the last layer. Of course, similar figures can be drawn for two inputs or more in a RBFN network. We can appreciate from these figures the architecture of this

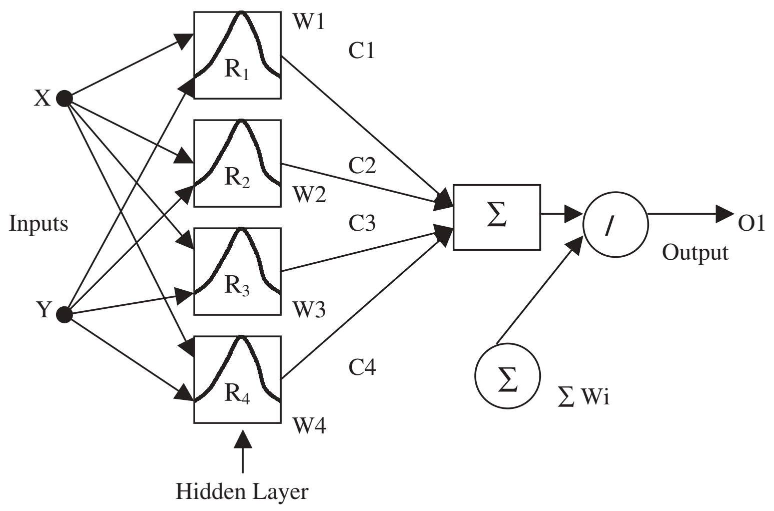


Fig. 4.17. Single-output RBFN that uses weighted average

type of neural networks. As a consequence we can see the difference between RBFN neural networks and MLP networks.

Moody-Darken's RBFN may be extended by assigning a linear function to the output function of each receptive field-that is, making c_i a linear combination of the input variables plus a constant:

$$c_i = \mathbf{a}_i^T \mathbf{x} + b_i \tag{4.24}$$

where $\mathbf{a}_i$ is a parameter vector and b_i is a scalar parameter. An RBFN's approximation capacity may be further improved with supervised adjustments of the center and shape of the receptive field (or radial basis) functions (Lee & Kil, 1991). Several learning algorithms have been proposed to identify the parameters ($\mathbf{u}_i$, σ_i, and c_i) of an RBFN. Besides using a supervised learning scheme alone to update all modifiable parameters, a variety of sequential training algorithms for RBFNs have been reported. The receptive field functions are first fixed, and then the weights of the output layer are adjusted. Several schemes have been proposed to determine the center positions ($\mathbf{u}_i$) of the receptive field functions. Lowe (1989) proposed a way to determine the centers based on standard deviations of the training data. Moody and Darken (1989) selected the centers $\mathbf{u}_i$ by means of data clustering techniques that assume that similar input vectors produce similar outputs; σ_i's are then obtained heuristically by taking the average distance to the several nearest neighbors of u_i's. Once the non-linear parameters are fixed and the receptive fields are frozen, the linear parameters (i.e., the weights of the output layer) can be updated using either the least squares method or the gradient method.

Using (4.24), extended RBFN response is identical to the response produced by the first-order Sugeno (type-1) fuzzy inference system described in

Chap. 2, provided that the membership functions, the radial basis functions, and certain operators are chosen correctly. While the RBFN consists of radial basis functions, the Sugeno fuzzy system contains a certain number of membership functions. Although the fuzzy system and the RBFN were developed on different bases, they are essentially rooted in the same grounds. Just as the RBFN enjoys quick convergence, the fuzzy system can evolve to recognize some features in a training data set.

Assuming that there is no noise in the training data set, we need to estimate a function $d(.)$ that yields exact desired outputs for all training data. This task is usually called an "interpolation" problem, and the resulting function $d(.)$ should pass through all of the training data points. When we use an RBFN with the same number of basis functions as we have training patterns, we have a so-called "interpolation RBFN", where each neuron in the hidden layer responds to one particular training input pattern.

Lets consider application of the RBFN network to the same example of Fig. 4.5. We will use a two layer RBFN network with 3 neurons in the hidden layer and weighted sum to calculate the output. We show in Fig. 4.18 the Gaussian radial basis function used in the network. Figure 4.19 illustrates the application of weighted sum to achieve the approximation of the training data. Figure 4.20 shows the final approximation achieved with the RBFN network, which is very good. The final SSE is of only 0.002, which is smaller than the one obtained by any of the previous methods. We can conclude that the RBFN network gives the best approximation to the training data of Fig. 4.5.

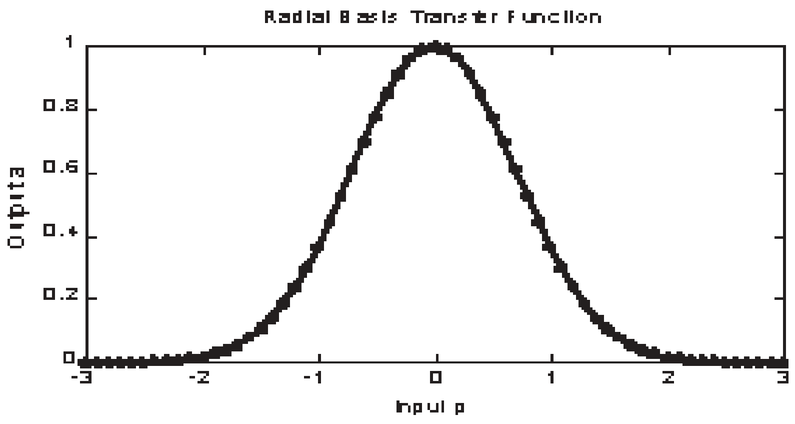


Fig. 4.18. Gaussian radial basis function

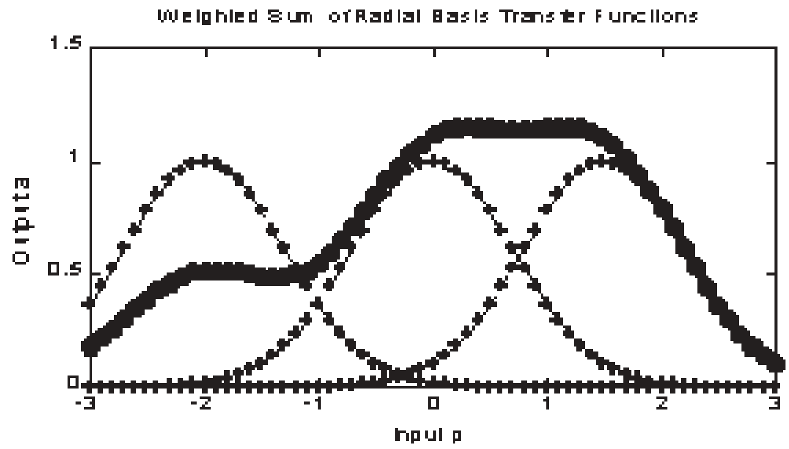


Fig. 4.19. Weighted sum of the three Gaussian functions of the RBFN network

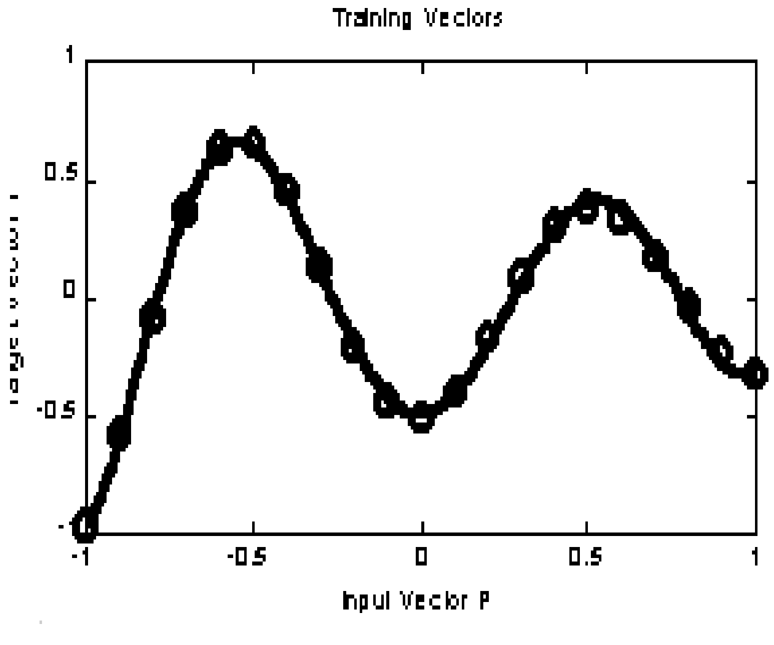


Fig. 4.20. Final function approximation achieved with the RBFN network

4.3 Adaptive Neuro-Fuzzy Inference Systems

In this section, we describe a class of adaptive networks that are functionally equivalent to fuzzy inference systems (Kosko, 1992). The architecture is referred to as ANFIS, which stands for "adaptive network-based fuzzy inference system". We describe how to decompose the parameter set to facilitate the hybrid learning rule for ANFIS architectures representing both the Sugeno and Tsukamoto fuzzy models.

4.3.1 ANFIS Architecture

A fuzzy inference system consists of three conceptual components: a fuzzy rule base, which contains a set of fuzzy if-then rules; a database, which defines the membership functions used in the fuzzy rules; and a reasoning mechanism, which performs the inference procedure upon the rules to derive a reasonable output or conclusion (Kandel, 1992). For simplicity, we assume that the fuzzy inference system under consideration has two inputs x and y and one output z. For a first-order Sugeno fuzzy model (Sugeno & Kang, 1988), a common rule set with two fuzzy if-then rules is the following:

$$\text{Rule 1: If } x \text{ is } A_1 \text{ and } y \text{ is } B_1\text{, then } f_1 = p_1x + q_1y + r_1 \ ,$$
$$\text{Rule 2: If } x \text{ is } A_2 \text{ and } y \text{ is } B_2\text{, then } f_2 = p_2x + q_2y + r_2 \ ,$$

Figure 4.21(a) illustrates the reasoning mechanism for this Sugeno model; the corresponding equivalent ANFIS architecture is as shown in Fig. 4.21(b), where nodes of the same layer have similar functions, as described next. (Here we denote the output of the ith node in layer l as $0_{l,i}$).

Layer 1: Every node i in this layer is an adaptive node with a node function

$$\begin{aligned} 0_{l,i} &= \mu_{Ai}(x), \text{ for } i = 1,\ 2\ , \\ 0_{l,i} &= \mu_{Bi-2}(y), \text{ for } i = 3,\ 4\ , \end{aligned} \tag{4.25}$$

where x (or y) is the input to node i and A_i (or B_{i-2}) is a linguistic label (such as "small" or "large") associated with this node. In other words, $0_{l,i}$ is the membership grade of a fuzzy set A and it specifies the degree to which the given input x (or y) satisfies the quantifier A. Here the membership function for A can be any appropriate parameterized membership function, such as the generalized bell function:

$$\mu_A(x) = \frac{1}{1 + |(x - c_i)/a_i|^{2bi}} \tag{4.26}$$

where $\{a_i, b_i, c_i\}$ is the parameter set. As the values of these parameters change, the bell-shaped function varies accordingly, thus exhibiting various forms of membership functions for a fuzzy set A. Parameters in this layer are referred to as "premise parameters".

Layer 2: Every node in this layer is a fixed node labeled Π, whose output is the product of all incoming signals:

$$0_{2,i} = w_i = \mu_{Ai}(x)\mu_{Bi}(y), i = 1, 2 \tag{4.27}$$

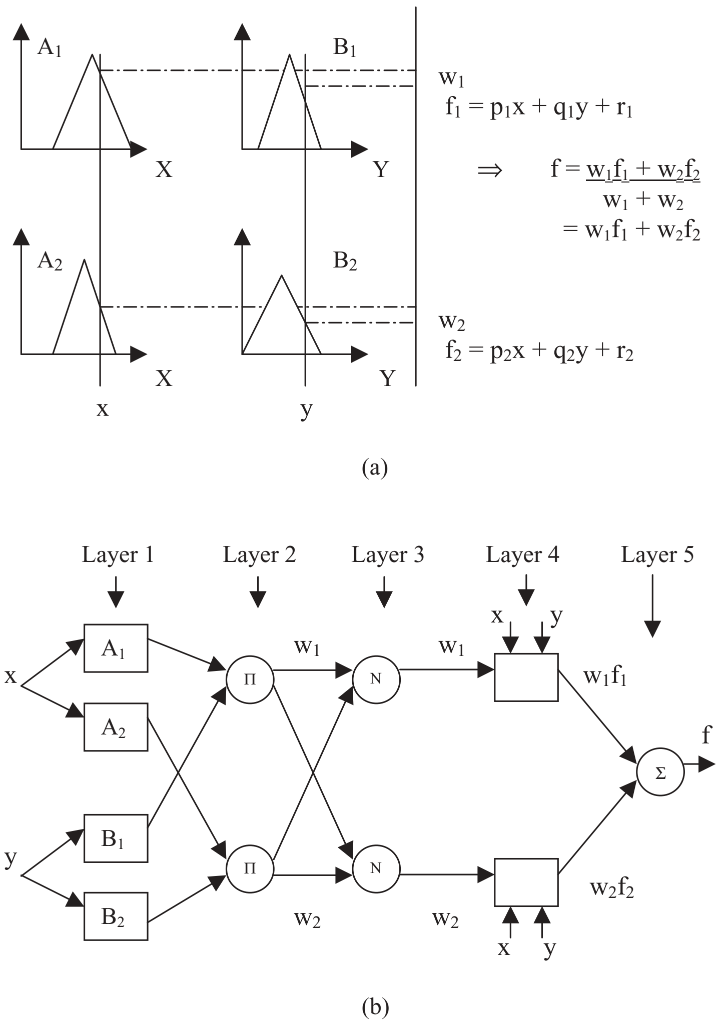


Fig. 4.21. (**a**) A two-input Sugeno fuzzy model with 2 rules; (**b**) equivalent ANFIS architecture (adaptive nodes shown with a square and fixed nodes with a circle)

Each node output represents the firing strength of a fuzzy rule.

Layer 3: Every node in this layer is a fixed node labeled N. The ith node calculates the ratio of the ith rule's firing strength to the sum of all rules' firing strengths:

$$0_{3,i} = w_i = w_i/(w_1 + w_2), i = 1,\ 2\ . \tag{4.28}$$

For convenience, outputs of this layer are called "normalized firing strengths".

Layer 4: Every node i in this layer is an adaptive node with a node function

$$0_{4,i} = w_i f_i = w_i(p_i x + q_i y + r_i) \; , \tag{4.29}$$

where w_i is a normalized firing strength from layer 3 and $\{p_i, q_i, r_i\}$ is the parameter set of this node. Parameters in this layer are referred to as "consequent parameters".

Layer 5: The single node in this layer is a fixed node labeled Σ, which computes the overall output as the summation of all incoming signals:

$$\text{overall output} = 0_{5,i} = \sum_i w_i f_i = \frac{\sum_i w_i f_i}{\sum_i w_i} \tag{4.30}$$

Thus we have constructed an adaptive network that is functionally equivalent to a Sugeno fuzzy model. We can note that the structure of this adaptive network is not unique; we can combine layers 3 and 4 to obtain an equivalent network with only four layers. In the extreme case, we can even shrink the whole network into a single adaptive node with the same parameter set. Obviously, the assignment of node functions and the network configuration are arbitrary, as long as each node and each layer perform meaningful and modular functionalities.

The extension from Sugeno ANFIS to Tsukamoto ANFIS is straightforward, as shown in Fig. 4.22, where the output of each rule (f_i, $i = 1, 2$) is induced jointly by a consequent membership function and a firing strength.

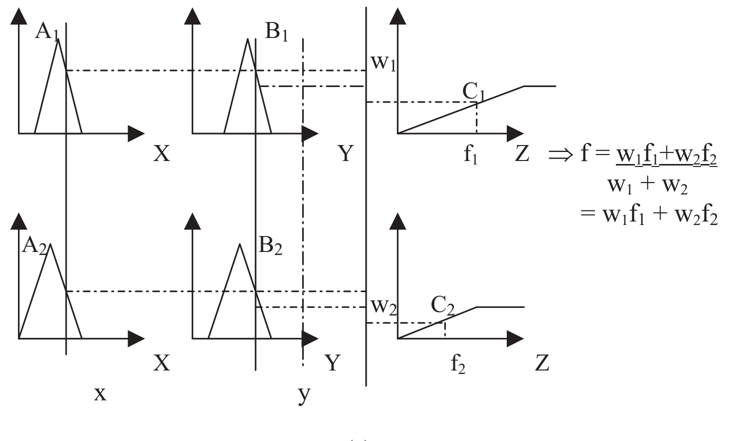
4.3.2 Learning Algorithm

From the ANFIS architecture shown in Fig. 4.21(b), we observe that when the values of the premise parameters are fixed, the overall output can be expressed as a linear combination of the consequent parameters. Mathematically, the output f in Fig. 4.21(b) can be written as

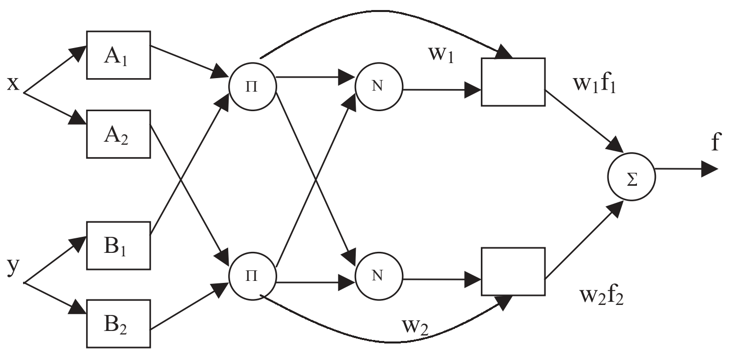
$$\begin{aligned} f &= \frac{w_1 f_1}{w_1 + w_2} + \frac{w_2 f_2}{w_1 + w_2} \\ &= w_1(p_1 x + q_1 y + r_1) + w_2(p_2 x + q_2 y + r_2) \\ &= (w_1 x)p_1 + (w_1 y)q_1 + (w_1)r_1 + (w_2 x)p_2 + (w_2 y)q_2 + (w_2)r_2 \end{aligned} \tag{4.31}$$

which is linear in the consequent parameters p_1, q_1, r_1, p_2, q_2, and r_2. From this observation, we can use a hybrid learning algorithm for parameter estimation in this kind of models (Jang, 1993). More specifically, in the forward pass of the hybrid learning algorithm, node outputs go forward until layer 4 and the consequent parameters are identified by the least-squares method. In the backward pass, the error signals propagate backward and the premise parameters are updated by gradient descent.

It has been shown (Jang, 1993) that the consequent parameters identified in this manner are optimal under the condition that the premise parameters are fixed. Accordingly, the hybrid approach converges much faster since



(a)



(b)

Fig. 4.22. (**a**) A two-input Tsukamoto fuzzy model with two rules; (**b**) equivalent ANFIS architecture

it reduces the search space dimensions of the original pure backpropagation method. For Tsukamoto ANFIS, this can be achieved if the membership function on the consequent part of each rule is replaced by a piecewise linear approximation with two consequent parameters.

As we discussed earlier, under certain minor conditions, an RBFN is functionally equivalent to a fuzzy system, and thus to ANFIS. This functional equivalence provides a shortcut for better understanding both ANFIS and RBFNs in the sense that development in either literature cross-fertilize the other (Jang, Sun & Mizutani, 1997).

Finally, we have to mention that it has been shown that the ANFIS methodology can be viewed as universal approximator (Jang, Sun & Mizutani, 1997). More specifically, it has been shown that when the number of rules is not restricted, a zero-order Sugeno model has unlimited approximation power for matching any non-linear function arbitrarily well on a compact set. This fact is intuitively reasonable. However, the mathematical proof can be made by showing that ANFIS satisfies the well-known Stone-Weierstrass theorem (Kantorovich & Akilov, 1982).

We will now show a simple example to illustrate the ANFIS methodology. We will use as training data the set of points shown in Fig. 4.5 We will use a network of 20 nodes, 4 rules, 4 Gaussian membership functions, and 16 unknown parameters. The complete network is shown in Fig. 4.23, in which we can clearly see all the details mentioned above. We have to mention that the ANFIS methodology is been used here to obtain a first order Sugeno model. In Fig. 4.24, we can appreciate the rate of convergence of ANFIS as the error is plotted against the number of epochs. From this figure it is clear that ANFIS can achieve a comparable error (with the previous methods in this chapter) in only 20 epochs, which is a lot less than the 1000 epochs required by the networks presented before (for the same example). In Fig. 4.25 we can see the final function approximation achieved by the ANFIS method, which is very good. In Fig. 4.26 we show the non-linear surface of the final fuzzy

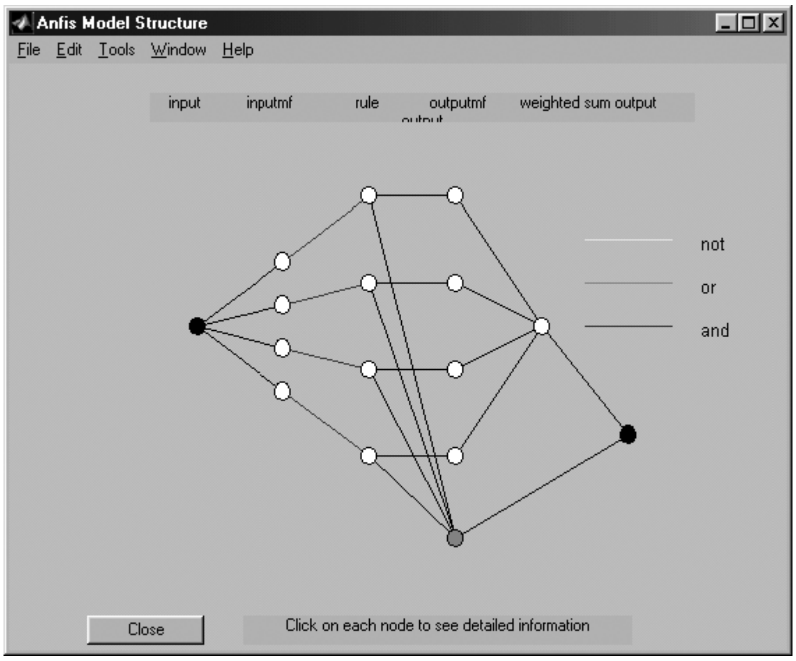
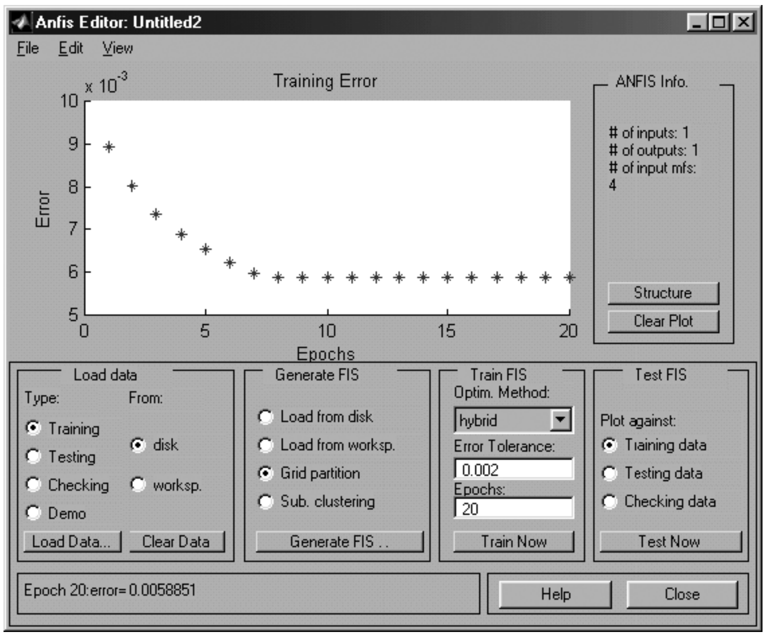


Fig. 4.23. Architecture of network for the ANFIS method



# of inputs: 1
# of outputs: 1
# of input mfs:
4

Type: [✓ Training] Testing Checking Demo
From: [✓ disk] worksp.

Load from disk
Load from worksp.
[✓ Grid partition]
Sub. clustering

Optim. Method: hybrid
Error Tolerance: 0.002
Epochs: 20

Plot against: [✓ Training data] Testing data Checking data

Epoch 20:error= 0.0058851

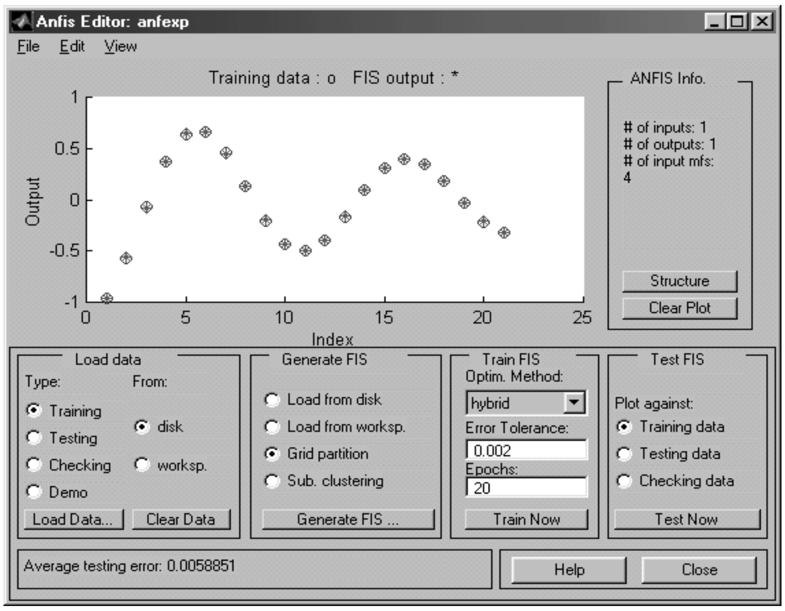
Fig. 4.24. Convergence of ANFIS (final SSE = 0.0058851)

system obtained by ANFIS. In Fig. 4.27, we show the use of ANFIS with specific values. In this case, the "rule viewer" of the Fuzzy Logic Toolbox of MATLAB is used to obtain these results. Finally, we show in Fig. 4.28 the membership functions for the input variable of ANFIS.

Finally, we have to say that the use of the ANFIS methodology is facilitated in the MATLAB programming language because it is already available in the Fuzzy Logic Toolbox. For this reason, all of the results shown before were obtained very easily using this tool of MATLAB.

4.4 Summary

In this chapter, we have presented the main ideas underlying supervised neural networks and the application of this powerful computational theory to general problems in function approximation. We have discussed in some detail the backpropagation learning algorithm for feedforward networks, radial basis function neural networks, and the integration of fuzzy logic techniques to neural networks to form powerful adaptive neuro-fuzzy inference

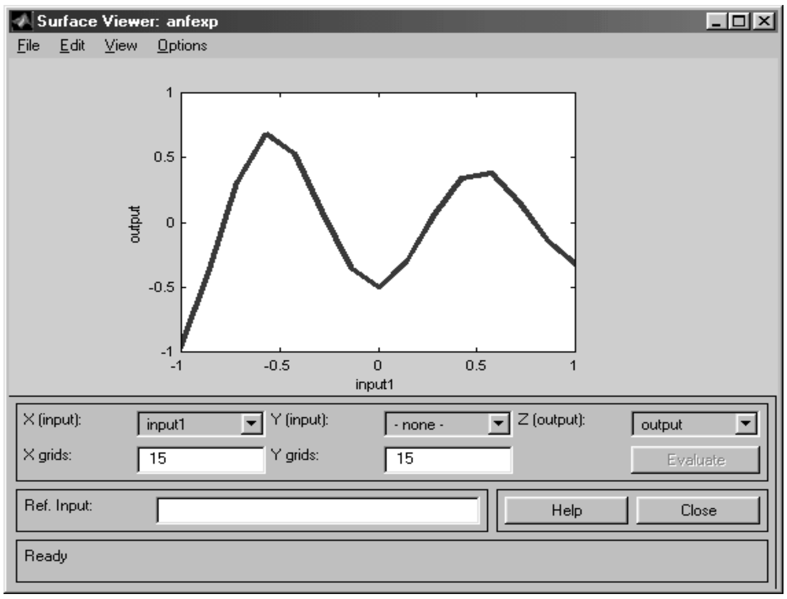


# of inputs: 1
# of outputs: 1
# of input mfs:
4

Type: [✓ Training] Testing Checking Demo
From: [✓ disk] worksp.
[✓ Grid partition] Load from disk Load from worksp. Sub. clustering
Optim. Method: hybrid
Error Tolerance: 0.002
Epochs: 20
Plot against: [✓ Training data] Testing data Checking data

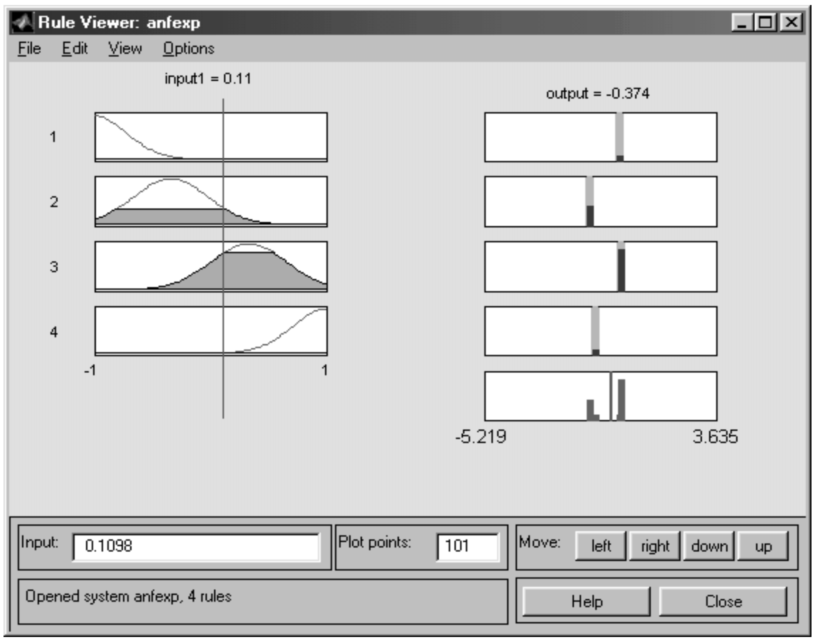
Average testing error: 0.0058851

Fig. 4.25. Final function approximation achieved by the ANFIS method



X (input): input1
Y (input): - none -
Z (output): output
X grids: 15
Y grids: 15
Ref. Input:
Ready

Fig. 4.26. Non-linear surface obtained by the ANFIS method

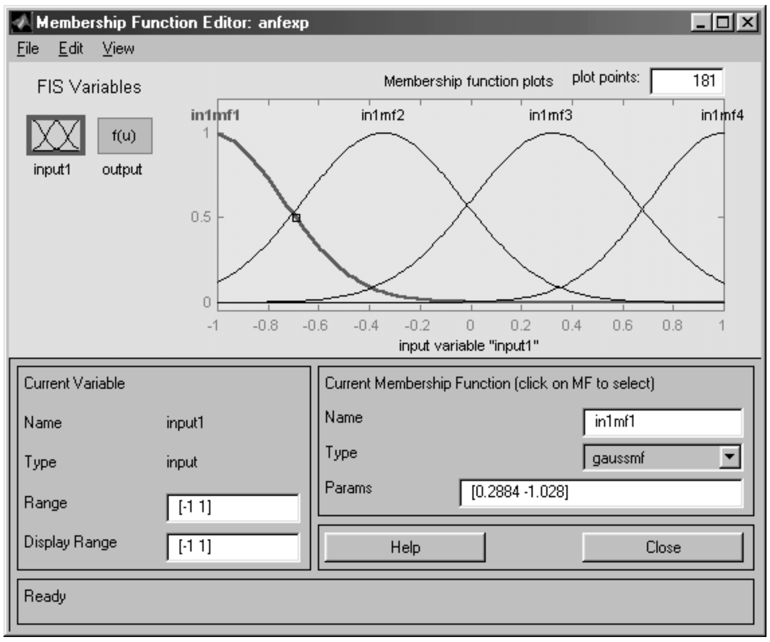


Input: 0.1098

Plot points: 101

Opened system anfexp, 4 rules

Fig. 4.27. Use of ANFIS to calculate specific values with the rule viewer



plot points: 181

Current Variable

Name input1

Type input

Range [-1 1]

Display Range [-1 1]

Current Membership Function (click on MF to select)

Name in1mf1

Type gaussmf

Params [0.2884 -1.028]

Ready

Fig. 4.28. Membership functions for the input variable of ANFIS

systems. In the following chapters, we will show how supervised neural network techniques (in conjunction with other techniques) can be applied to solve real world complex problems in intelligent pattern recognition. This chapter will serve as a basis for the new hybrid intelligent methods that will be described in the chapters at the end of this book.