Chapter 3
Artificial Neural Networks

3.1 Introduction

The human brain is like an information processing machine. Information can be seen as signals coming from senses, which then runs through the nervous system. The brain consists of around 100 to 500 billion neurons. These neurons form clusters and networks. Depending on their targets, neurons can be organized hierarchically or layered.

In this chapter we present the basic model of the neuron in order to understand how neural networks work. Then, we offer the classification of these models by their structures and their learning procedure. Finally, we describe several neural models with their respective learning procedures. Some examples are given throughout the chapter.

Biologically, we can find two types of cell groups in the nervous system: the *glial* and *nervous cells*. The nervous cells, also called neurons, are organized in a functional *syncytium* way. This is a complex distribution, which can be imagined as a computer network or a telephone system network. Neurons communicate through an area called a *synapse*. This region is a contact point of two neurons.

The main components of the neurons are the *axon* and the *dendrite*. Each neuron has only one axon, which is divided into multiple exits on a later stage. The number of dendrites may vary and each of them is an extension of the neuron's body, which increases the number of entries of information collection. The axon is the only exit point of the neuron and can be up to 120 mm long.

In a simplified way, a neuron functions in a way where none, one or many electrical impulses are received through its dendrites from axons from other neurons. Those electrical impulses are added in order to have a final potential. This potential must exceed a certain level to have the neuron generate an electrical impulse on its axon. If the level required is not met, then the axon of that neuron does not fire its axon.

In other words, neurons can be divided into *dendrites*, which are channels of input signals, a *core cell* that processes all these signals, and *axons* that transmit output

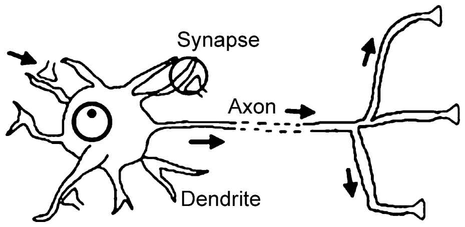


Fig. 3.1 Schematic drawing of a biological neuron

signals of the processed information from dendrites. Figure 3.1 shows a schematic drawing of a neuron.

Why would we want to look inside neurons? The human brain processes information and can react from distinct stimuli. Moreover, the brain can generalize this information to act when new situations are presented. If we are looking inside neurons, we are in a sense searching the notion of how the human brain learns and generalizes information.

At this point, neurons can be modeled as follows. Dendrites are connected to some axons from other neurons, but some links are reinforced when typical actions occur. However, links are not very strong when these channels are not used. Therefore, input signals are weighted by this reinforcement (positively or negatively). All these signals are then summarized and the core cell processes that information. This process is modeled mathematically by an *activation function*. Finally, the result is transmitted by axons and the output signal of the neuron goes to other cells. Figure 3.2 shows this neural model.

The activation function is the characterization of the neurons' activities when input signals stimulate them. Then, the activation function can be any kind of function that describes the neural processes. However, the most common are sigmoidal func-

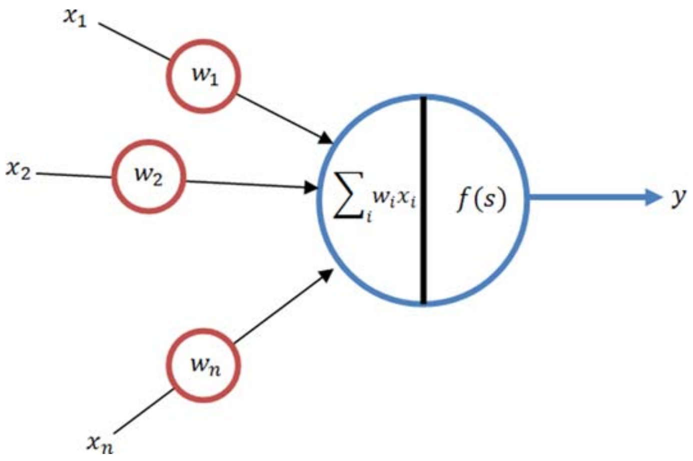


Fig. 3.2 Neural model

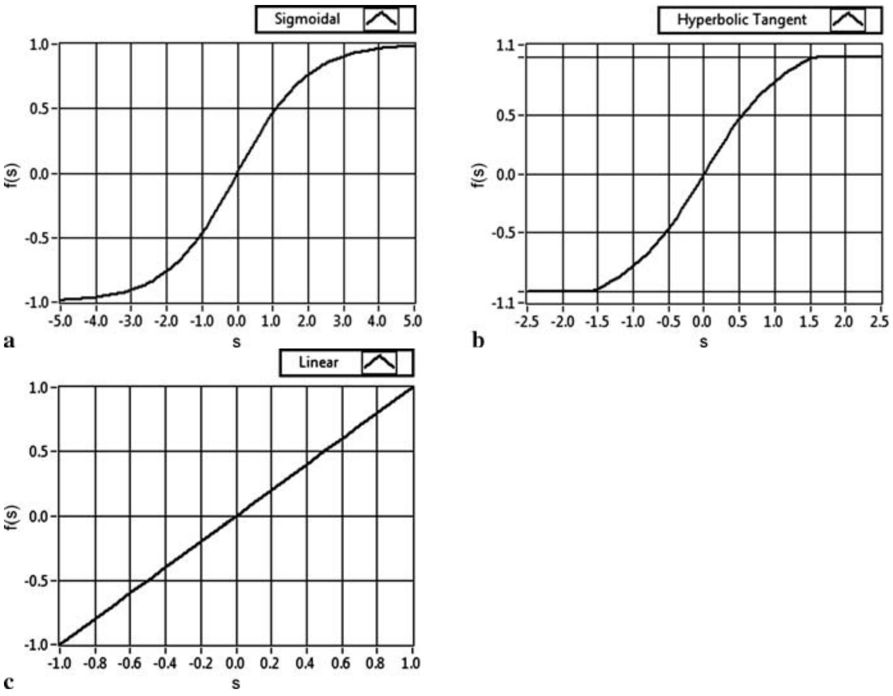


Fig. 3.3a–c Graphs of typical activation functions. **a** Sigmoidal function: $f_s(x) = \frac{2}{e^x+1} - 1$. **b** Hyperbolic tangent function: $f_{\tanh}(x) = \tanh(x)$. **c** Linear function: $f_{\text{linear}}(x) = x$

tion, linear function, and hyperbolic tangent function. Figure 3.3 shows the graphical form of these functions.

These functions can be implemented in LabVIEW with the Intelligent Control Toolkit in the following way. First, open the ICTL and select the *ANN* (artificial neural network) section. We can find all the possibilities in ANN as seen in Figs. 3.4 and 3.5. The activation functions are in ICTL ≫ ANN ≫ Backpropagation ≫ NN

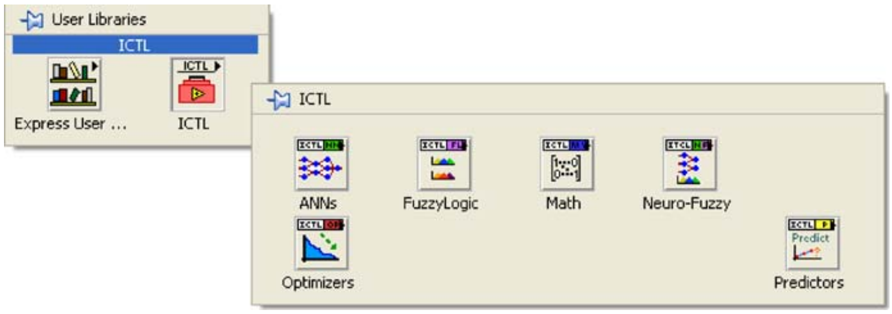


Fig. 3.4 Accessing ANNs library in the ICTL

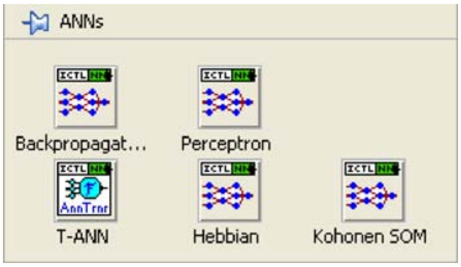


Fig. 3.5 ANNs library in ICTL

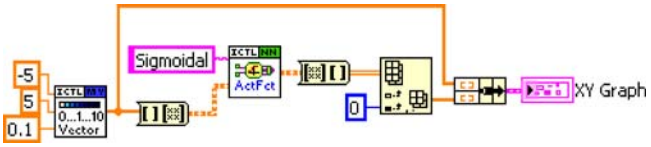


Fig. 3.6 Block diagram of the Sigmoidal function

Methods ≫ **activationFunction.vi**. For instance, we want to plot the sigmoidal function. Then, we can create a VI like that in Fig. 3.6.

At first, we have a **rampVector.vi** that creates an array of elements with values initialized at −5 and runs with a stepsize of 0.1 through to 5. The activation function VI must be a real-valued matrix. This is the reason why the vector is transformed into a matrix. Also, the output of this VI is a matrix. Then, we need to get all values in the first column. Finally, the array in the interval $[-5, 5]$ is plotted against values $f(x)$ for all $x \in [-5, 5]$.

The graph resulting from this process is seen in Fig. 3.3. If we want to plot the other function, the only thing we need to do is change the label *Sigmoidal* to *Hyperbolic Tangent* in order to plot the hyperbolic tangent function, or *User Defined* if we need a linear function. Figure 3.7 shows these labels in the block diagram.

Example 3.1. Let $\boldsymbol{X} = \{0.4, -0.5, 0.2, -0.7\}$ be the input vector and $\boldsymbol{W} = \{0.1, 0.6, 0.2, 0.3\}$ be the weight vector. Suppose a sigmoidal activation function is the processing of the core cell. (a) What is the value of the output signal? (b) What is the value of the output signal if we change the activation function in (3.1), known as symmetrical hard limiting?

$$f(s) = \begin{cases} -1 & s \leq 0 \\ 1 & s > 0 \end{cases} \tag{3.1}$$

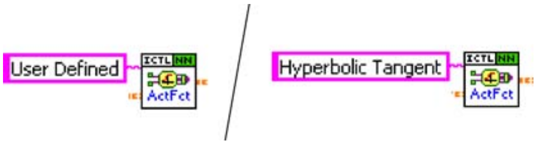
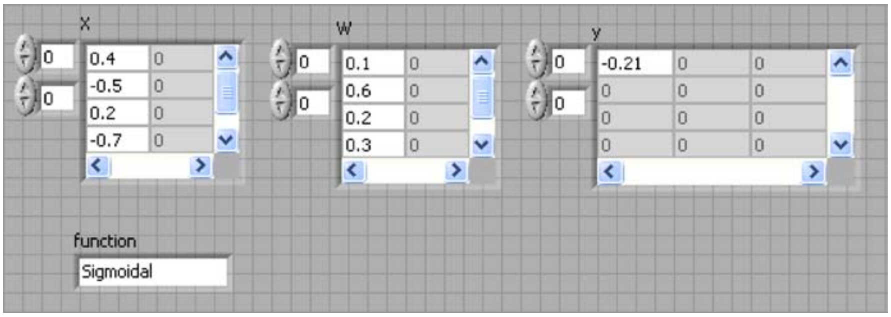


Fig. 3.7 Labels in the activation function VI



X

0	0.4	0
0	-0.5	0
	0.2	0
	-0.7	0

W

0	0.1	0
0	0.6	0
	0.2	0
	0.3	0

y

0	-0.21	0	0
0	0	0	0
	0	0	0
	0	0	0

function

Sigmoidal

Fig. 3.8 Calculations of the output signal

Solution. (a) We need to calculate the inner product of the vector $\boldsymbol{X}$ and $\boldsymbol{W}$. Then, the real-value is evaluated in the sigmoidal activation function.

$$y = f_{\text{sigmoidal}}\left(\sum_i w_i x_i = (0.4)(0.1) + (-0.5)(0.6) + (0.2)(0.2) + (-0.7)(0.3) = -0.43\right) = -0.21 \tag{3.2}$$

This operation can be implemented in LabVIEW as follows. First, we need the NN (neural network) VI located in the path ICTL $\gg$ ANNs $\gg$ Backpropagation $\gg$ NN Methods $\gg$ **neuralNetwork.vi**. Then, we create three real-valued matrices as seen in Fig. 3.8. The block diagram is shown in Fig. 3.9. In view of this block diagram, we need some parameters that will be explained later. At the moment, we are interested in connecting the X-matrix in the *inputs* connector and **W**-matrix in the *weights* connector. The label for the activation function is *Sigmoidal* in this example but can be any other label treated before. The condition 1 in the $L-1$ connector comes from the fact that we are mapping a neural network with four inputs to one output. Then, the number of layers L is 2 and by the condition $L-1$ we get the number 1 in the blue square. The 1D array $\{4, 1\}$ specifies the number of neurons per layer, the input layer (four) and the output layer (one). At the *globalOutputs* the **y**-matrix is connected.

From the previous block diagram of Fig. 3.9 mixed with the block diagram of Fig. 3.6, the connections in Fig. 3.10 give the graph of the sigmoidal function evaluated at –0.43 pictured in Fig. 3.11. Note the connection comes from the **neuralNet-**

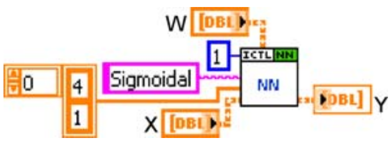


Fig. 3.9 Block diagram of Example 3.1

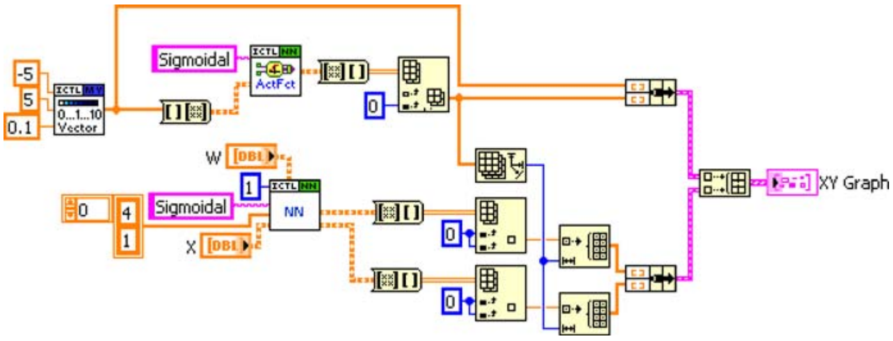


Fig. 3.10 Block diagram for plotting the graph in Fig. 3.11

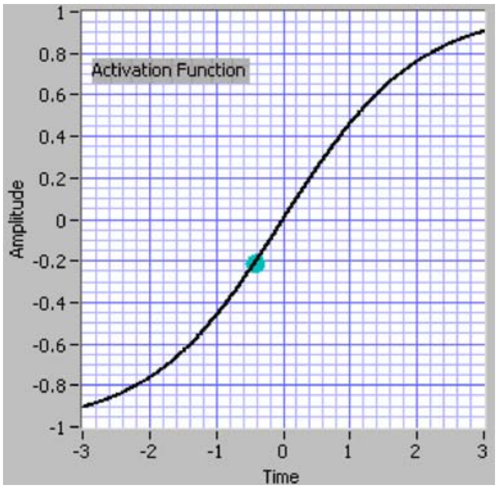


Fig. 3.11 The value −0.43 evaluated at a Sigmoidal function

work.vi at the *sumOut* pin. Actually, this value is the inner product or the sum of the linear combination between $\boldsymbol{X}$ and $\boldsymbol{W}$. This real value is then evaluated at the activation function. Therefore, this is the x-coordinate of the activation function and the y-coordinate is the *globalOutput*. Of course, these two out-connectors are in matrix form. We need to extract the first value at the position $(0, 0)$ in these matrices. This is the reason we use the matrix-to-array transformation and the index array nodes. The last block is an initialize array that creates a 1D array of m elements (sizing from any vector of the sigmoidal block diagram plot) with the value −0.43 for the *sumOut* connection and the value −0.21 for the *globalOutput* link. Finally, we create an array of clusters to plot the activation function in the interval $[-5, 5]$ and the actual value of that function.

(b) The inner product is the same as the previous one, −0.43. Then, the activation function is evaluated when this value is fired. So, the output value becomes −1. This is represented in the graph in Fig. 3.12. The activation function for the symmetric hard limiting can be accessed in the path ICTL ≫ ANNs ≫ Perceptron ≫ Trans-

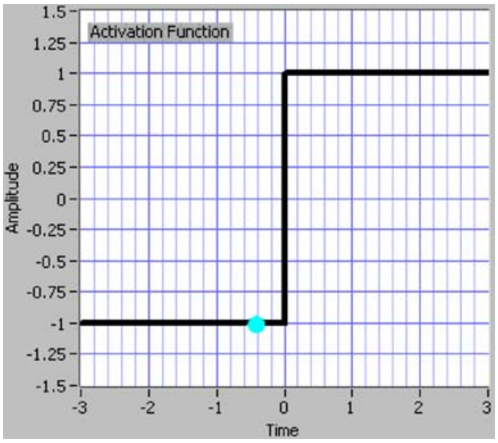


Fig. 3.12 The value −0.43 evaluated at the symmetrical hard limiting activation function

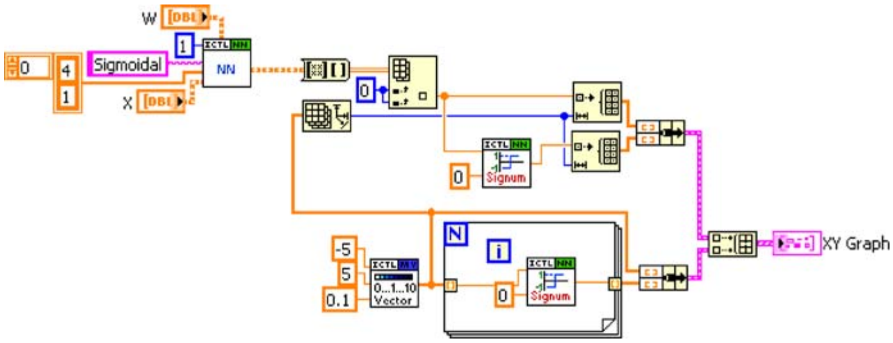


Fig. 3.13 Block diagram of the plot in Fig. 3.12

fer F. ≫ **signum.vi**. The block diagram of Fig. 3.13 shows the next explanation. In this diagram, we see the activation function below the NN VI. It consists of the array in the interval $[-5, 5]$ and inside the for-loop is the symmetric hard limiting function. Of course, the decision outside the **neuralNetwork.vi** comes from the *sumOut* and evaluates this value in a symmetric hard limiting case. □

Neurons communicate between themselves and form a neural network. If we use the mathematical neural model, then we can create an ANN. The basic idea behind ANNs is to simulate the behavior of the human brain in order to define an artificial computation and solve several problems. The concept of an ANN introduces a simple form of biological neurons and their interactions, passing information through the links. That information is essentially transformed in a computational way by mathematical models and algorithms.

Neural networks have the following properties:

1. Able to learn data collection;
2. Able to generalize information;
3. Able to recognize patterns;

4. Filtering signals;
5. Classifying data;
6. Is a massively parallel distributed processor;
7. Predicting and approximating functions;
8. Universal approximators.

Considering their properties and applications, ANNs can be classified as: supervised networks, unsupervised networks, competitive or self-organizing networks, and recurrent networks.

As seen above, ANNs are used to generalize information, but first need to be trained. Training is the process where neural models find the weights of each neuron. There are several methods of training like the backpropagation algorithm used in feed-forward networks. The training procedure is actually derived from the need to minimize errors.

For example, if we are trying to find the weights in a supervised network. Then, we have to have at least some input and output data samples. With this data, by different methods of training, ANNs measure the error between the actual output of the neural network and the desired output. The minimization of error is the target of every training procedure. If it can be found (the minimum error) then the weights that produce this minimization are the optimal weights that enable the trained neural network to be ready for use. Some applications in which ANNs have been used are (general and detailed information found in [1–14]):

Analysis in forest industry. This application was developed by O. Simula, J. Vesanto, P. Vasara and R.R. Helminen in Finland. The core of the problem is to cluster the pulp and paper mills of the world in order to determine how these resources are valued in the market. In other words, executives want to know the competitiveness of their packages coming from the forest industry. This clustering was solved with a Kohonen network system analysis.

Detection of aircraft in synthetic aperture radar (*SAR*) *images.* This application involves real-time systems and image recognition in a vision field. The main idea is to detect aircrafts in images known as SAR and in this case they are color aerial photographs. A multi-layer neural network perceptron was used to determine the contrast and correlation parameters in the image, to improve background discrimination and register the RGB bands in the images. This application was developed by A. Filippidis, L.C. Jain and N.M. Martin from Australia. They use a fuzzy reasoning in order to benefit more from the advantages of artificial intelligence techniques. In this case, neural networks were used in order to design the inside of the fuzzy controllers.

Fingerprint classification. In Turkey, U. Halici, A. Erol and G. Ongun developed a fingerprint classification with neural networks. This approach was designed in 1999 and the idea was to recognize fingerprints. This is a typical application using ANNs. Some people use multi-layer neural networks and others, as in this case, use self-organizing maps. *Scheduling communication systems.* In the Institute of Informatics and Telecommunications in Italy, S. Cavalieri and O. Mirabella developed a multi-layer neural network system to optimize a scheduling in real-time communication systems.

Controlling engine generators. In 2004, S. Weifeng and T. Tianhao developed a controller for a marine diesel engine generator [2]. The purpose was to implement a controller that could modify its parameters to encourage the generator with optimal behavior. They used neural networks and a typical PID controller structure for this application.

3.2 Artificial Neural Network Classification

Neural models are used in several problems, but there are typically five main problems in which ANNs are accepted (Table 3.1). In addition to biological neurons, ANNs have different structures depending on the task that they are trying to solve. On one hand, neural models have different structures and then, those can be classified in the two categories below. Figure 3.14 summarizes the classification of the ANN by their structures and training procedures.
Feed-forward networks. These neural models use the input signals that flow only in the direction of the output signals. Single and multi-layer neural networks are typical examples of that structure. Output signals are consequences of the input signals and the weights involved.
Feed-back networks. This structure is similar to the last one but some neurons have loop signals, that is, some of the output signals come back to the same neuron or neurons placed before the actual one. Output signals are the result of the non-transient response of the neurons excited by input signals.

On the other hand, neural models are classified by their learning procedure. There are three fundamental types of models, as described in the following:

1. *Supervised networks.* When we have some data collection that we really know, then we can train a neural network based on this data. Input and output signals are imposed and the weights of the structure can be found.

Table 3.1 Main tasks that ANNs solve

Task	Description
Function approximation	Linear and non-linear functions can be approximated by neural networks. Then, these are used as fitting functions.
Classification	1. *Data classification.* Neural networks assign data to a specific class or subset defined. Useful for finding patterns. 2. *Signal classification.* Time series data is classified into subsets or classes. Useful for identifying objects.
Unsupervised clustering	Specifies order in data. Creates clusters of data in unknown classes.
Forecasting	Neural networks are used to predict the next values of a time series.
Control systems	Function approximation, classification, unsupervised clustering and forecasting are characteristics that control systems uses. Then, ANNs are used in modeling and analyzing control systems.

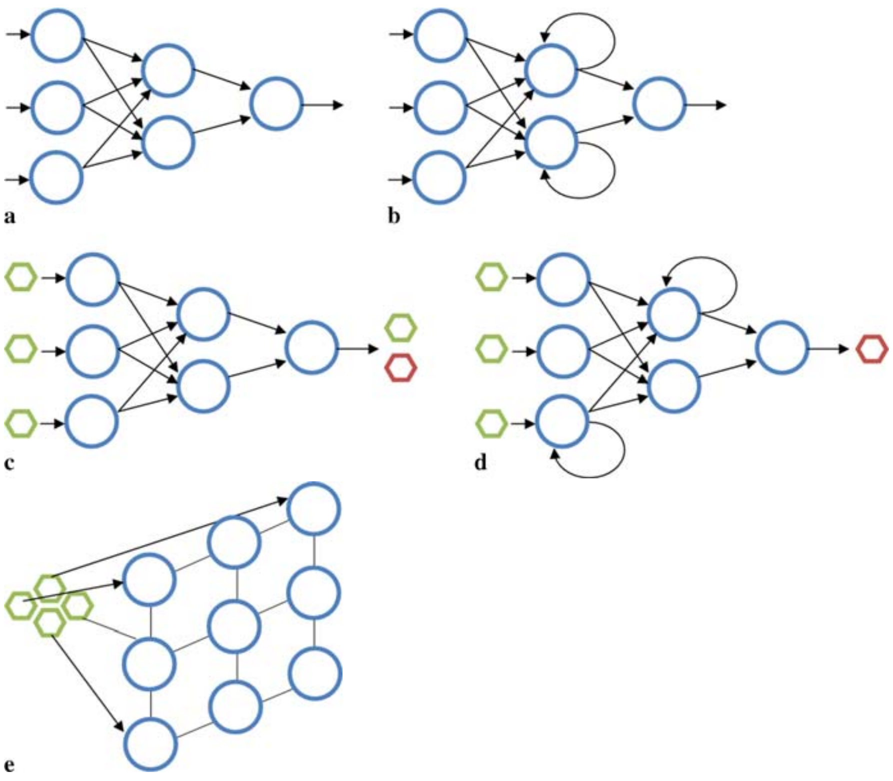


Fig. 3.14a–e Classification of ANNs. **a** Feed-forward network. **b** Feed-back network. **c** Supervised network. **d** Unsupervised network. **e** Competitive or self-organizing network

2. *Unsupervised networks.* In contrast, when we do not have any information, this type of neural model is used to find patterns in the input space in order to train it. An example of this neural model is the Hebbian network.
3. *Competitive or self-organizing networks.* In addition to unsupervised networks, no information is used to train the structure. However, in this case, neurons fight for a dedicated response by specific input data from the input space. Kohonen maps are a typical example.

3.3 Artificial Neural Networks

The human brain adapts its neurons in order to solve the problem presented. In these terms, neural networks shape different architectures or arrays of their neurons. For different problems, there are different structures or models. In this section, we explain the basis of several models such as the perceptron, multi-layer neural networks, trigonometric neural networks, Hebbian networks, Kohonen maps and Bayesian networks. It will be useful to introduce their training methods as well.

3.3.1 Perceptron

Perceptron or threshold neuron is the simplest form of the biological neuron modeling. This kind of neuron has input signals and they are weighted. Then, the activation function decides and the output signal is offered. The main point of this type of neuron is its activation function modeled as a threshold function like that in (3.3). Perceptron is very useful to classify data. As an example, consider the data shown in Table 3.2.

$$f(s) = y = \begin{cases} 0 & s < 0 \\ 1 & s \geq 0 \end{cases} \tag{3.3}$$

We want to classify the input vector $\boldsymbol{X} = \{x_1, x_2\}$ as shown by the target y. This example is very simple and simulates the *AND* operator. Suppose then that weights are $\boldsymbol{W} = \{1, 1\}$ (so-called weight vector) and the activation function is like that given in (3.3). The neural network used is a perceptron. What are the output values for each sample of the input vector at this time?
Create a new VI. In this VI we need a real-value matrix for the input vector $\boldsymbol{X}$ and two 1D arrays. One of these arrays is for the weight vector $\boldsymbol{W}$ and the other is for the output signal **y**. Then, a for-loop is located in order to scan the **X**-matrix row by row. Each row of the **X**-matrix with the weight vector is an inner product implemented with the **sum_weight_inputs.vi** located at ICTL ≫ ANNs ≫ Perceptron ≫ Neuron Parts ≫ **sum_weight_inputs.vi**. The *xi* connector is for the row vector of the X-matrix, the w_{ij} is for the weight array and the *bias* pin in this moment gets the value 0. The explanation of this parameter is given below. After that, the activation function is evaluated at the sum of the linear combination.

We can find this activation function in the path ICTL ≫ ANNs ≫ Perceptron ≫ Transfer F. ≫ **threshold.vi**. The *threshold* connector is used to define in which value the function is discontinued. Values above this threshold are 1 and values below this one are 0. Finally, these values are stored in the output array. Figure 3.15 shows the block diagram and Fig. 3.16 shows the front panel.

Table 3.2 Data for perceptron example

x_1	x_2	y
0.2	0.2	0
0.2	0.8	0
0.8	0.2	0
0.8	0.8	1

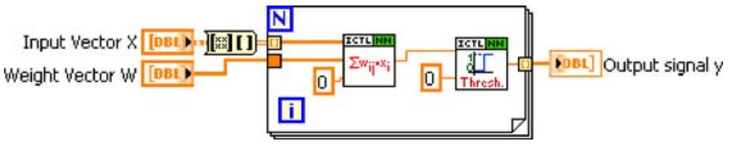
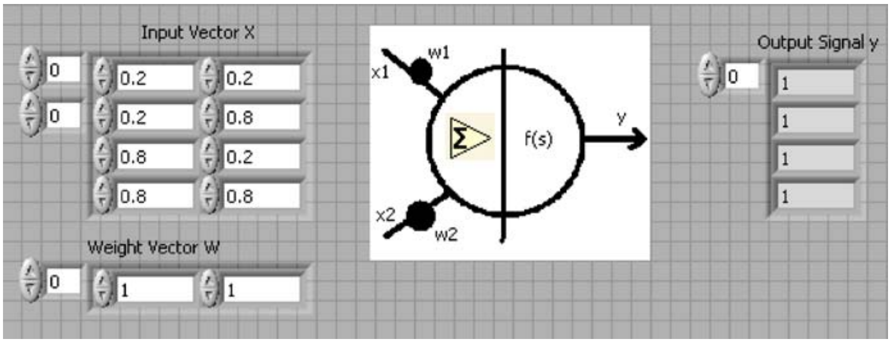


Fig. 3.15 Block diagram for evaluating a perceptron

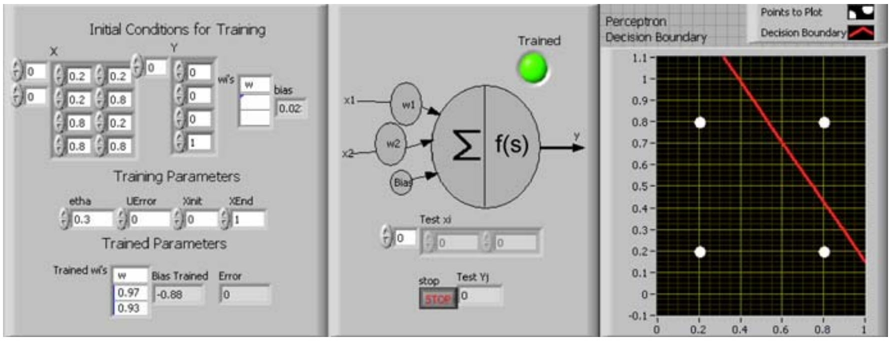


Input Vector X
0 | 0.2 | 0.2
0 | 0.2 | 0.8
0.8 | 0.2
0.8 | 0.8

Weight Vector W
0 | 1 | 1

Output Signal y
0 | 1 | 1 | 1 | 1

Fig. 3.16 Calculations for the initial state of the perceptron learning procedure



Initial Conditions for Training
X: 0 | 0 | 0.2 | 0.2 / 0.2 | 0.8 / 0.8 | 0.2 / 0.8 | 0.8
Y: 0 | 0 | 0 | 0 | 1
wi's: w
bias: 0.02

Training Parameters
etha: 0.3 | UError: 0 | Xinit: 0 | XEnd: 1

Trained Parameters
Trained wi's: w | 0.97 | 0.93
Bias Trained: -0.88
Error: 0

Test xi: 0 | 0 | 0
stop: STOP
Test Yj: 0

Fig. 3.17 Example of the trained perceptron network emulating the *AND* operator

As we can see, the output signals do not coincide with the values that we want. In the following, the training will be performed as a supervised network. Taking the desired output value y and the actual output signal y', the error function can be determined as in (3.4):

$$E = y - y' . \tag{3.4}$$

The rule of updating the weights is in given as:

$$w_{\text{new}} = w_{\text{old}} + \eta E \boldsymbol{X} , \tag{3.5}$$

where w_{new} is the updated weight, w_{old} is the actual weight, η is the learning rate, a constant between 0 and 1 that is used to adjust how fast learning is, and $\boldsymbol{X} = \{x_1, x_2\}$ for this example and in general $\boldsymbol{X} = \{x_1, x_2, \ldots, x_n\}$ is the input vector. This rule applies to every single weight participating in the neuron. Continuing with the example for LabVIEW, assume the learning rate is $\eta = 0.3$, then the updating weights are as in Fig. 3.17.

This example can be found in ICTL $\gg$ ANNs $\gg$ Perceptron $\gg$ **Example_Perceptron.vi**. At this moment we know the X-matrix or the 2D array, the desired Y-array. The parameter *etha* is the learning rate, and *UError* is the error that we want to have between the desired output signal and the current output for the perceptron. To draw the plot, the interval is $[Xinit, XEnd]$. The weight array and the *bias* are selected, initializing randomly. Finally, the *Trained Parameters* are the values found by the learning procedure.

In the second block of Fig. 3.17, we find the test panel. In this panel we can evaluate any point $\boldsymbol{X} = \{x_1, x_2\}$ and see how the perceptron classifies it. The Boolean LED is on only when a solution is found. Otherwise, it is off. The third panel in Fig. 3.17 shows the graph for this example. The red line shows how the neural network classifies points. Any point below this line is classified as 0 and all the other values above this line are classified as 1.

About the bias. In the last example, the training of the perceptron has an additional element called *bias*. This is an input coefficient that preserves the action of translating the red line displayed by the weights (it is the cross line that separates the elements). If no bias were found at the neuron, the red line can only move around the zero-point. Bias is used to translate this red line to another place that makes possible the classification of the elements in the input space. As with input signals, bias has its own weight. Arbitrarily, the bias value is considered as one unit. Therefore, bias in the previous example is interpreted as the weight of the unitary value.

This can be viewed in the 2D space. Suppose, $\boldsymbol{X} = \{x_1, x_2\}$ and $\boldsymbol{W} = \{w_1, w_2\}$. Then, the linear combination is done by:

$$y = f\left(\sum_i x_i w_i + b\right) = f\left(x_1 w_1 + x_2 w_2 + b\right) . \tag{3.6}$$

Then,

$$f(s) = \begin{cases} 0 & \text{if } -b > x_1 w_1 + x_2 w_2 \\ 1 & \text{if } -b \leq x_1 w_1 + x_2 w_2 \end{cases} . \tag{3.7}$$

Then, $\{w_1, w_2\}$ form a basis of the output signal. By this fact, $\boldsymbol{W}$ is orthogonal to the input vector $\boldsymbol{X} = \{x_1, x_2\}$. Finally, if the inner product of these two vectors is zero then we can know that the equations form a boundary line for the decision process. In fact, the boundary line is:

$$x_1 w_1 + x_2 w_2 + b = 0 . \tag{3.8}$$

Rearranging the elements, the equation becomes:

$$x_1 w_1 + x_2 w_2 = -b . \tag{3.9}$$

Then, by linear algebra we know that the last equation is the expression of a plane, with distance from the origin equal to $-b$. So, b is in fact the deterministic value that translates the line boundary more closely or further away from the zero-point. The angle for this line between the x-axis is determined by the vector $\boldsymbol{W}$. In general, the line boundary is plotted by:

$$x_1 w_1 + \ldots + x_n w_n = -b . \tag{3.10}$$

We can make perceptron networks with the condition that neurons have an activation function like that found in (3.3). By increasing the number of perceptron neurons, a better classification of non-linear elements is done. In this case, neurons form

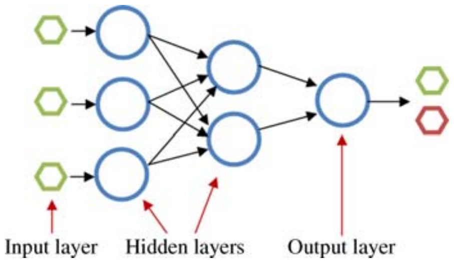


Fig. 3.18 Representation of a feed-forward multi-layer neural network

layers. Each layer is connected to the next one if the network is feed-forward. In another case, layers can be connected to their preceding or succeeding layers. The first layer in known as the *input layer*, the last one is the *output layer*, where the intermediate layers are called *hidden layers* (Fig. 3.18).

The algorithm for training a feed-forward perceptron neural network is presented in the following:

Algorithm 3.1	Learning procedure of perceptron nets
Step 1	Determine a data collection of the input/output signals (x_i, y_i). Generate random values of the weights w_i. Initialize the time $t = 0$.
Step 2	Evaluate perceptron with the inputs x_i and obtain the output signals y'_i.
Step 3	Calculate the error E with (3.4).
Step 4	If error $E = 0$ for every i then *STOP*. Else, update weight values with (3.5), $t \leftarrow t + 1$ and go to *Step 2*.

3.3.2 *Multi-layer Neural Network*

This neural model is quite similar to the perceptron network. However, the activation function is not a unit step. In this ANN, neurons have any number of activation functions; the only restriction is that functions must be continuous in the entire domain.

3.3.2.1 ADALINE

The easiest neural network is the adaptive linear neuron (ADALINE). This is the first model that uses a linear activation function like $f(s) = s$. In other words, the inner product of the input and weight vectors is the output signal of the neuron. More precisely, the function is as in (3.11):

$$y = f(s) = s = w_0 + \sum_{i=1}^{n} w_i x_i \, , \tag{3.11}$$

where w_0 is the bias weight. Thus, as with the previous networks, this neural network needs to be trained. The training of this neural model is called the delta rule. In this case, we assume one input x to a neuron. Thus, considering an ADALINE, the error is measured as:

$$E = y - y' = y - w_1 x \,. \tag{3.12}$$

Looking for the square of the error, we might have

$$e = \frac{1}{2}\left(y - w_1 x\right)^2 \,. \tag{3.13}$$

Trying to minimize the error is the same as the derivative of the error with respect to the weight, as shown in (3.14):

$$\frac{\mathrm{d}e}{\mathrm{d}w} = -Ex \,. \tag{3.14}$$

Thus, this derivative tells us in which direction the error increases faster. The weight change must then be proportional and negative to this derivative. Therefore, $\Delta w = \eta E x$, where η is the learning rate. Extending the updating rule of the weights to a multi-input neuron is show in (3.15):

$$\begin{aligned} w_0^{t+1} &= w_0^t + \eta E \\ w_i^{t+1} &= w_i^t + \eta E x_i \,. \end{aligned} \tag{3.15}$$

A supervised ADALINE network is used if a threshold is placed at the output signal. This kind of neural network is known as a linear multi-layer neural network without saturation of the activation function.

3.3.2.2 General Neural Network

ADALINE is a linear neural network by its activation function. However, in some cases, this activation function is not the desirable one. Other functions are then used, for example, the sigmoidal or the hyperbolic tangent functions. These functions are shown in Fig. 3.3.

In this way, the delta rule cannot be used to train the neural network. Therefore another algorithm is used based on the gradient of the error, called the backpropagation algorithm. We need a pair of input/output signals to train the neural model. This type of ANN is then classified as supervised and feed-forward, because the input signals go from the beginning to the end.

When we are attempting to find the error between the desired value and the actual value, only the error at the last layer (or the output layer) is measured. Therefore, the idea behind the backpropagation algorithm is to retro-propagate the error from the output layer to the input layer through hidden layers. This ensures that a *kind* of proportional error is preserved in each neuron. The updating of the weights can then be done by a variation or delta error, proportional to a learning rate.

First, we divide the process into two structures. One is for the values at the last layer (output layer) and the other values are from the hidden layers to the input layers. In these terms, the updating rule of the output weights is

$$\Delta v_{ji} = \sum_{j} \left(-\eta \delta_j^q z_i^q \right), \tag{3.16}$$

where v_{ji} is the weight linking the ith actual neuron with the jth neuron in the previous layer, and q is the number of the sample data. The other variables are given in (3.17):

$$z_i^q = f\left(\sum_{k=0}^{n} w_{ik} x_k^q \right). \tag{3.17}$$

This value is the input to the hidden neuron i in (3.18):

$$\delta_j^q = \left(o_j^q - y_j^q \right) f'\left(\sum_{k=1}^{m} v_{jk} z_k^q \right). \tag{3.18}$$

Computations of the last equations come from the delta rule. We also need to understand that in hidden layers there are no desired values to compare. Then, we propagate the error to the last layers in order to know how neurons produce the final error. These values are computed by:

$$\Delta^q w_{ik} = -\eta \frac{\partial E^q}{\partial w_{ik}} = -\eta \frac{\partial E^q}{\partial o_i^q} \frac{\partial o_i^q}{\partial w_{ik}}, \tag{3.19}$$

where o_i^q is the output of the ith hidden neuron. Then, $o_i^q = z_i^q$ and

$$\frac{\partial o_i^q}{\partial w_{ik}} = f'\left(\sum_{h=0}^{n} w_{ih} x_h^q \right) x_k^q. \tag{3.20}$$

Now, we obtain the value

$$\delta_i^q = \frac{\partial E^q}{\partial o_i^q} = \sum_{j=1}^{g} \frac{\partial E^q}{\partial o_j^q} \frac{\partial o_j^q}{\partial o_i^q}, \tag{3.21}$$

which is related to the hidden layer. Observe that j is the element of the jth output neuron. Finally, we already know the values $\frac{\partial E^q}{\partial o_j^q}$ and the last expression is:

$$\delta_i^q = f_i'\left(\sum_{k=0}^{n} w_{ik} x_k^q \right) \sum_{j=1}^{p} v_{ij} \delta_j^q. \tag{3.22}$$

Algorithm 3.2 shows the backpropagation learning procedure for a two-layer neural network (an input layer, one hidden layer, and the output layer). This algorithm can be easily extended to more than one hidden layer. The last net is called a multi-layer or n-layer feed-forward neural network. Backpropagation can be thought of as a generalization of the delta rule and can be used instead when ADALINE is implemented.

Algorithm 3.2	Backpropagation
Step 1	Select a learning rate value η. Determine a data collection of q samples of inputs x and outputs y. Generate random values of weights w_{ik} where i specifies the ith neuron in the actual layer and k is the kth neuron of the previous layer. Initialize the time $t = 0$.
Step 2	Evaluate the neural network and obtain the output values o_i.
Step 3	Calculate the error as $E^q(w) = \frac{1}{2}\sum_{i=1}^{p}(o_i^q - y_i^q)^2$.
Step 4	Calculate the delta values of the output layer: $\delta_i^q = f_i'(\sum_{k=1}^{n} v_{ik} z_k)(o_i^q - y_i^q)$. Calculate the delta values at the hidden layer as: $\delta_i^q = f_i'(\sum_{k=0}^{n} w_{ik} x_k^q)\sum_{j=1}^{p} v_{ij}\delta_j^q$.
Step 5	Determine the change of weights as $\Delta w_{ik}^q = -\eta\delta_i^q o_k^q$ and update the parameters with the next rule $w_{ik}^q \leftarrow w_{ik}^q + \Delta w_{ik}^q$.
Step 6	If $E \leq e$ min where e min is the minimum error expected then *STOP*. Else, $t \leftarrow t + 1$ and go to *Step 2*.

Example 3.2. Consider the points in $\Re^2$ as in Table 3.3. We need to classify them into two clusters by a three-layer feed-forward neural network (with one hidden layer). The last column of the data represents the target $\{0, 1\}$ of each cluster. Consider the learning rate to be 0.1.

Table 3.3 Data points in $\Re^2$

Point	X-coordinate	Y-coordinate	Cluster
1	1	2	0
2	2	3	0
3	1	1	0
4	1	3	0
5	2	2	0
6	6	6	1
7	7	6	1
8	7	5	1
9	8	6	1
10	8	5	1

Solution. First, we have the input layer with two neurons; one for the x-coordinate and the second one for the y-coordinate. The output layer is simply a neuron that must be in the domain $[0, 1]$. For this example we consider a two-neuron hidden layer (actually, there is no analytical way to define the number of hidden neurons).

Table 3.4 Randomly initialized weights

Weights between the first and second layers	Weights between the second and third layers
0.0278	0.0004
0.0148	0.0025
0.0199	
0.0322	

We need to consider the following parameters:

Activation function:	Sigmoidal
Learning rate:	0.1
Number of layers:	3
Number of neurons per layer:	$2-2-1$

Other parameters that we need to consider are related to the stop criterion:

Maximum number of iterations:	1000
Minimum error or energy:	0.001
Minimum tolerance of error:	0.0001

In fact, the input training data are the two columns of coordinates. The output training data is the last column of cluster targets. The last step before the algorithm will train the net is to initialize the weights randomly. Consider as an example, the randomizing of values in Table 3.4.

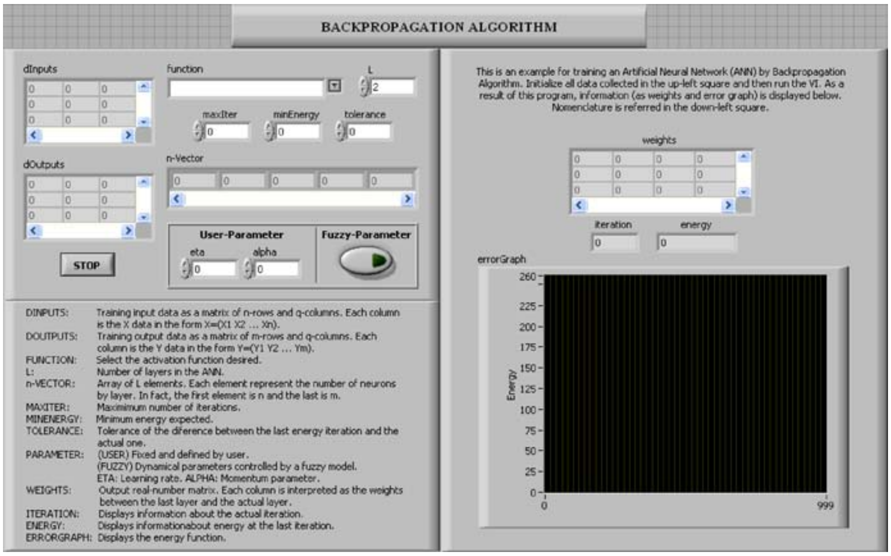
According to the above parameters, we are able to run the backpropagation algorithm implemented in LabVIEW. Go to the path ICTL ≫ ANNs ≫ Backpropagation ≫ **Example_Backpropagation.vi**. In the front panel, we can see the window shown in Fig. 3.19. Desired input values must be in the form of (3.23):

$$X = \begin{bmatrix} x_1^1 & \dots & x_1^m \\ \vdots & \ddots & \vdots \\ x_n^1 & \dots & x_n^m \end{bmatrix}, \tag{3.23}$$

where $x^j = \{x_1^j, \dots, x_n^j\}^T$ is the column vector of the jth sample with n elements. In our example, $x^j = \{X^j, Y^j\}$ has two elements. Of course, we have 10 samples of that data, so $j = 1, \dots, 10$. The desired input data in the matrix looks like Fig. 3.20. The desired output data must also be in the same form as (3.23).

The term $y^j = \{y_1^j, \dots, y_r^j\}^T$ is the column of the jth sample with r elements. In our example, we have $y^j = \{C^j\}$, where C is the corresponding value of the cluster. In fact, we need exactly $j = 1, \dots, 10$ terms to solve the problem. This matrix looks like Fig. 3.21.

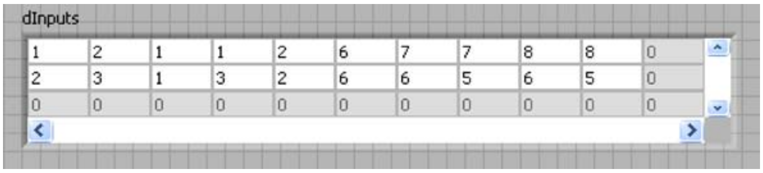
In the *function* value we will select *Sigmoidal*. In addition, L is the number of layers in the neural network. We treated a three-layer neural network, so $L = 3$. The



This is an example for training an Artificial Neural Network (ANN) by Backpropagation Algorithm. Initialize all data collected in the up-left square and then run the VI. As a result of this program, information (as weights and error graph) is displayed below. Nomenclature is referred in the down-left square.

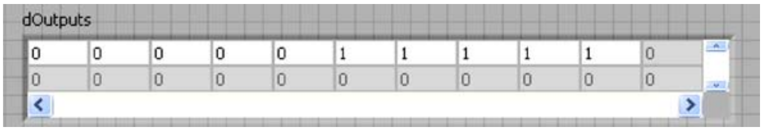
DINPUTS:	Training input data as a matrix of n-rows and q-columns. Each column is the X data in the form X=(X1 X2 ... Xn).
DOUTPUTS:	Training output data as a matrix of m-rows and q-columns. Each column is the Y data in the form Y=(Y1 Y2 ... Ym).
FUNCTION:	Select the activation function desired.
L:	Number of layers in the ANN.
n-VECTOR:	Array of L elements. Each element represent the number of neurons by layer. In fact, the first element is n and the last is m.
MAXITER:	Maximimum number of iterations.
MINENERGY:	Minimum energy expected.
TOLERANCE:	Tolerance of the diference between the last energy iteration and the actual one.
PARAMETER:	(USER) Fixed and defined by user. (FUZZY) Dynamical parameters controlled by a fuzzy model. ETA: Learning rate. ALPHA: Momentum parameter.
WEIGHTS:	Output real-number matrix. Each column is interpreted as the weights between the last layer and the actual layer.
ITERATION:	Displays information about the actual iteration.
ENERGY:	Displays informationabout energy at the last iteration.
ERRORGRAPH:	Displays the energy function.

Fig. 3.19 Front panel of the backpropagation algorithm


dInputs

1	2	1	1	2	6	7	7	8	8	0
2	3	1	3	2	6	6	5	6	5	0
0	0	0	0	0	0	0	0	0	0	0

Fig. 3.20 Desired input data


dOutputs

0	0	0	0	0	1	1	1	1	1	0
0	0	0	0	0	0	0	0	0	0	0

Fig. 3.21 Desired output data

n-vector is an array in which each of the elements represents the number of neurons per layer. Indeed, we have to write the array *n-vector* $= \{2, 2, 1\}$. Finally, *maxIter* is the maximum number of iterations we want to wait until the best answer is found. *minEnergy* is the minimum error between the desired output and the actual values derived from the neural network.

Tolerance is the variable that controls the minimum change in error that we want in the training procedure. Then, if one of the three last values is reached, the procedure will stop. We can use crisp parameters of fuzzy parameters to train the network, where *eta* is the learning rate and *alpha* is the momentum parameter.

As seen in Fig. 3.19, the right window displays the result. *Weights* values will appear until the process is finished and there are the coefficients of the trained neural

Table 3.5 Trained weights

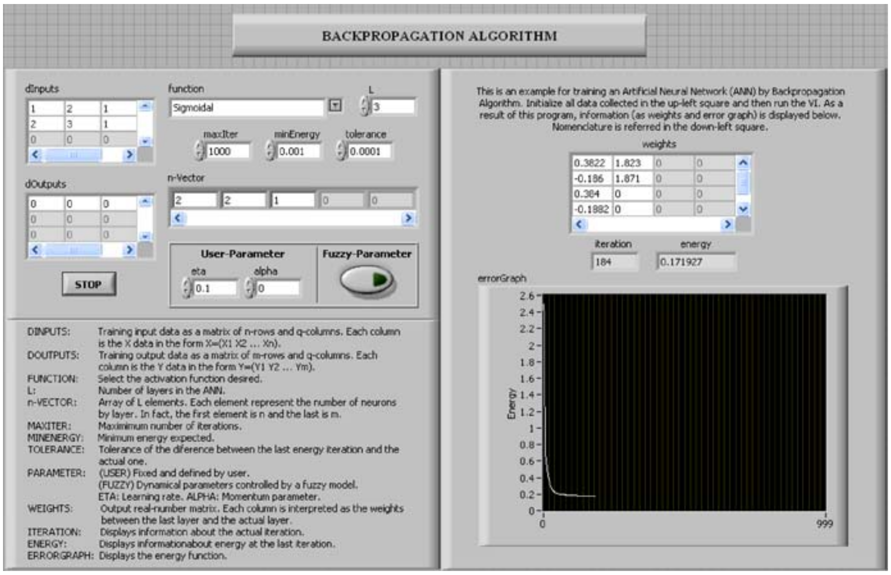
Weights between the first and second layers	Weights between the second and third layers
0.3822	1.8230
−0.1860	1.8710
0.3840	
−0.1882	

network. The *errorGraph* shows the decrease in the error value when the actual output values are compared with the desired output values. The real-valued number appears in the *error* indicator. Finally, the *iteration* value corresponds to the number of iterations completed at the moment.

With those details, the algorithm is implemented and the training network (or the weights) is shown in Table 3.5 (done in 184 iterations and reaching the local minima at 0.1719). The front panel of the algorithm looks like Fig. 3.22.

In order to understand what this training has implemented, there are graphs of this classification. In Fig. 3.23, the first graph is the data collection, and the second graph shows the clusters. If we see a part of the block diagram in Fig. 3.24, only the input data is used in the three-layer neural network. To show that this neural network can generalize, other data different from the training collection is used. Looking at Fig. 3.25, we see the data close to the training zero-cluster. □

When the learning rate is not selected correctly, the solution might be trapped in local minima. In other words, minimization of the error is not reached. This can be


BACKPROPAGATION ALGORITHM

dInputs

1	2	1
2	3	1
0	0	0

dOutputs

0	0	0
0	0	0
0	0	0

function: Sigmoidal

L: 3

maxIter: 1000 minEnergy: 0.001 tolerance: 0.0001

n-Vector

2	2	1	0	0

User-Parameter: eta 0.1 alpha 0

Fuzzy-Parameter

STOP

DINPUTS: Training input data as a matrix of n-rows and q-columns. Each column is the X data in the form X=(X1 X2 ... Xn).
DOUTPUTS: Training output data as a matrix of m-rows and q-columns. Each column is the Y data in the form Y=(Y1 Y2 ... Ym).
FUNCTION: Select the activation function desired.
L: Number of layers in the ANN.
n-VECTOR: Array of L elements. Each element represent the number of neurons by layer. In fact, the first element is n and the last is m.
MAXITER: Maximum number of iterations.
MINENERGY: Minimum energy expected.
TOLERANCE: Tolerance of the diference between the last energy iteration and the actual one.
PARAMETER: (USER) Fixed and defined by user.
(FUZZY) Dynamical parameters controlled by a fuzzy model.
ETA: Learning rate. ALPHA: Momentum parameter.
WEIGHTS: Output real-number matrix. Each column is interpreted as the weights between the last layer and the actual layer.
ITERATION: Displays information about the actual iteration.
ENERGY: Displays informationabout energy at the last iteration.
ERRORGRAPH: Displays the energy function.

This is an example for training an Artificial Neural Network (ANN) by Backpropagation Algorithm. Initialize all data collected in the up-left square and then run the VI. As a result of this program, information (as weights and error graph) is displayed below. Nomenclature is referred in the down-left square.

weights

0.3822	1.823	0	0
-0.186	1.871	0	0
0.384	0	0	0
-0.1882	0	0	0

iteration: 184 energy: 0.171927

Fig. 3.22 Implementation of the backpropagation algorithm

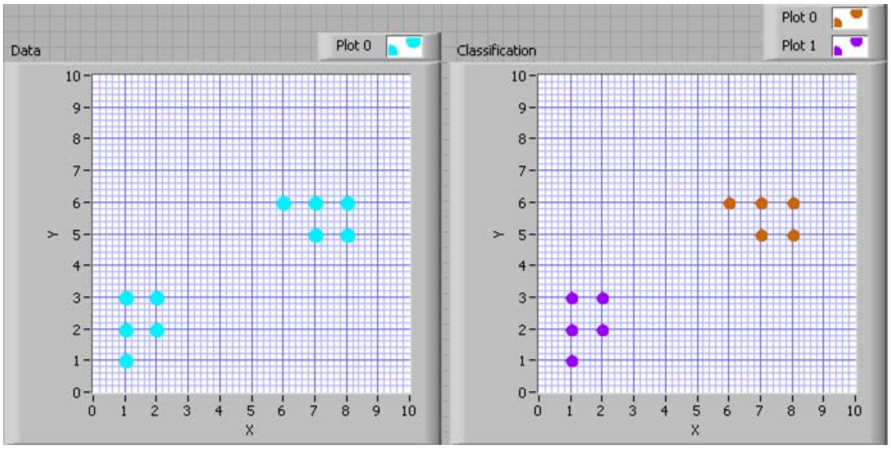
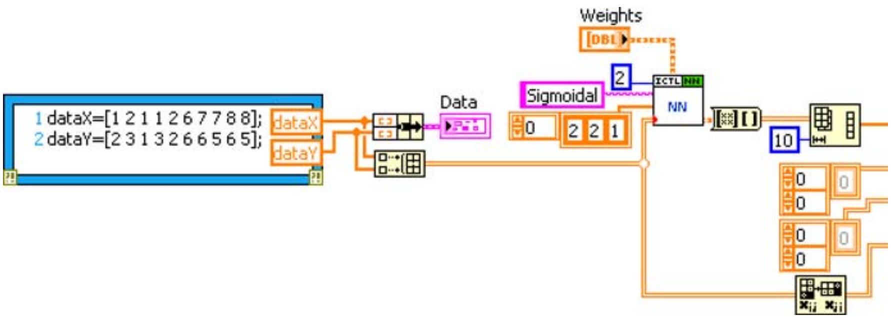


Fig. 3.23 The *left* side shows a data collection, and the *right* shows the classification of that data



```
dataX=[1 2 1 1 2 6 7 7 8 8];
dataY=[2 3 1 3 2 6 6 5 6 5];
```

Fig. 3.24 Partial view of the block diagram in classification data, showing the use of the neural network

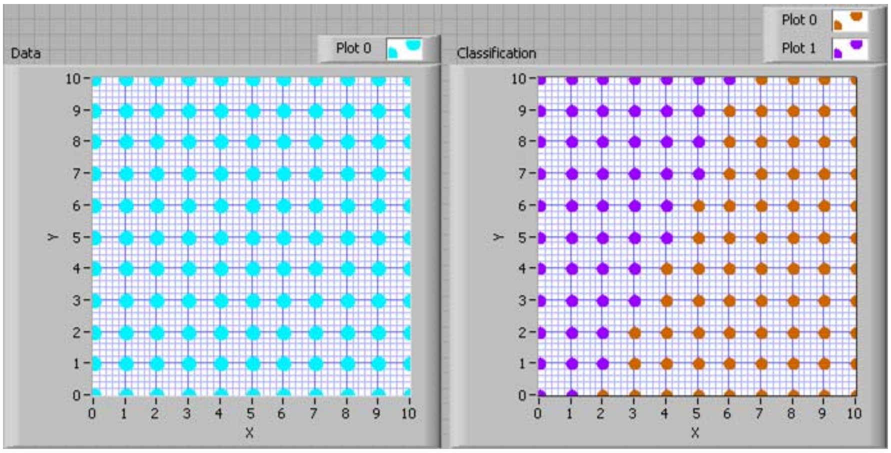


Fig. 3.25 Generalization of the data classification

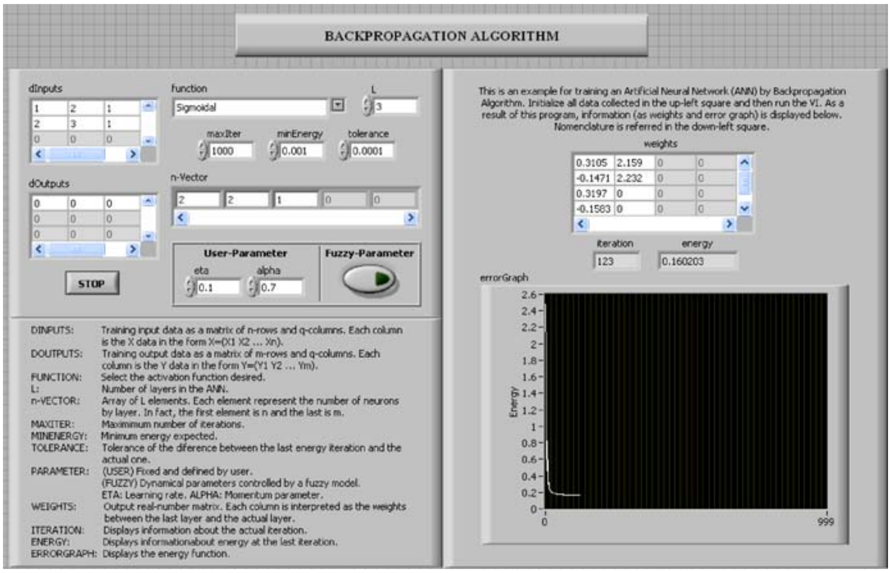
partially solved if the learning rate is decreased, but time grows considerably. One solution is the modification of the backpropagation algorithm by adding a momentum coefficient. This is used to try to get the tending of the solution in the weight space. This means that the solution is trying to find and follow the tendency of the previous updating weights. That modification is summarized in Algorithm 3.3, which is a rephrased version of Algorithm 3.2 with the new value.

Algorithm 3.3	Backpropagation with momentum parameter
Step 1	Select a learning rate value η and momentum parameter α. Determine a data collection of q samples of inputs x and outputs y. Generate random values of weights w_{ik} where i specifies the ith neuron in the actual layer and k is the kth neuron of the previous layer. Initialize the time $t = 0$.
Step 2	Evaluate the neural network and obtain the output values o_i.
Step 3	Calculate the error as $E^q(w) = \frac{1}{2}\sum_{i=1}^{p}(o_i^q - y_i^q)^2$.
Step 4	Calculate the delta values of the output layer: $\delta_i^q = f_i'(\sum_{k=1}^{n} v_{ik} z_k)(o_i^q - y_i^q)$. Calculate the delta values at the hidden layer as: $\delta_i^q = f_i'(\sum_{k=0}^{n} w_{ik} x_k^q)\sum_{j=1}^{p} v_{ij}\delta_j^q$.
Step 5	Determine the change of weights as $\Delta w_{ik}^q = -\eta\delta_i^q o_k^q$ and update the parameters with the next rule: $w_{ik}^q \leftarrow w_{ik}^q + \Delta w_{ik}^q + \alpha\left(w_{ik\_act}^q - w_{ik\_last}^q\right)$ where w_{act} is the actual weight and $w_{last}$is the previous weight.
Step 6	If $E \le e$ min where e min is the minimum error expected then *STOP*. Else, $t \leftarrow t + 1$ and go to *Step 2*.

Example 3.3. Train a three-layer feed-forward neural network using a 0.7 momentum parameter value and all data used in Example 3.2.
Solution. We present the final results in Table 3.6 and the algorithm implemented in Fig. 3.26. We find the number of iterations to be 123 and the local minima 0.1602, with a momentum parameter of 0.7. This minimizes in some way the number of iterations (decreasing the time processing at the learning procedure) and the local minima is smaller than when no momentum parameter is used. □

Table 3.6 Trained weights for feed-forward network

Weights between the first and second layers	Weights between the second and third layers
0.3822	1.8230
−0.1860	1.8710
0.3840	
−0.1882	



dInputs

1	2	1
2	3	1
0	0	0

dOutputs

0	0	0
0	0	0
0	0	0

function: Sigmoidal
L: 3
maxIter: 1000
minEnergy: 0.001
tolerance: 0.0001

n-Vector

2	2	1	0	0

eta: 0.1
alpha: 0.7

DINPUTS:	Training input data as a matrix of n-rows and q-columns. Each column is the X data in the form X=(X1 X2 ... Xn).
DOUTPUTS:	Training output data as a matrix of m-rows and q-columns. Each column is the Y data in the form Y=(Y1 Y2 ... Ym).
FUNCTION:	Select the activation function desired.
L:	Number of layers in the ANN.
n-VECTOR:	Array of L elements. Each element represent the number of neurons by layer. In fact, the first element is n and the last is m.
MAXITER:	Maximmum number of iterations.
MINENERGY:	Minimum energy expected.
TOLERANCE:	Tolerance of the diference between the last energy iteration and the actual one.
PARAMETER:	(USER) Fixed and defined by user. (FUZZY) Dynamical parameters controlled by a fuzzy model. ETA: Learning rate. ALPHA: Momentum parameter.
WEIGHTS:	Output real-number matrix. Each column is interpreted as the weights between the last layer and the actual layer.
ITERATION:	Displays information about the actual iteration.
ENERGY:	Displays informationabout energy at the last iteration.
ERRORGRAPH:	Displays the energy function.

This is an example for training an Artificial Neural Network (ANN) by Backpropagation Algorithm. Initialize all data collected in the up-left square and then run the VI. As a result of this program, information (as weights and error graph) is displayed below. Nomenclature is referred in the down-left square.

weights

0.3105	2.159	0	0
-0.1471	2.232	0	0
0.3197	0	0	0
-0.1583	0	0	0

iteration: 123
energy: 0.160203

Fig. 3.26 Implementation of the backpropagation algorithm with momentum parameter

3.3.2.3 Fuzzy Parameters in the Backpropagation Algorithm

In this section we combine the knowledge about fuzzy logic and ANNs. In this way, the main idea is to control the parameters of learning rate and momentum in order to get fuzzy values and then evaluate the optimal values for these parameters.

We first provide the fuzzy controllers for the two parameters at the same time. As we know from Chap. 2 on fuzzy logic, we evaluate the error and the change in the error coefficients from the backpropagation algorithm. That is, after evaluating the error in the algorithm, this value enters the fuzzy controller . The change in the error is the difference between the actual error value and the last error evaluated.

Input membership functions are represented as the normalized domain drawn in Figs. 3.27 and 3.28. Fuzzy sets are *low positive* (LP), *medium positive* (MP), and *high positive* (HP) for error value E. In contrast, fuzzy sets for change in error *CE* are *low negative* (LN), *medium negative* (MN), and *high negative* (HN). Figure 3.29 reports the fuzzy membership functions of change parameter $\Delta\beta$ with fuzzy sets *low negative* (LN), *zero* (ZE), and *low positive* (LP). Tables 3.7 and 3.8 have the fuzzy associated matrices (FAM) to imply the fuzzy rules for the learning rate and momentum parameter, respectively.

In order to access the fuzzy parameters, go to the path ICTL $\gg$ ANNs $\gg$ Backpropagation $\gg$ **Example_Backpropagation.vi**. As with previous examples, we can obtain better results with these fuzzy parameters. Configure the settings of this VI except for the learning rate and momentum parameter. Switch on the *Fuzzy-Parameter* button and run the VI. Figure 3.30 shows the window running this configuration.

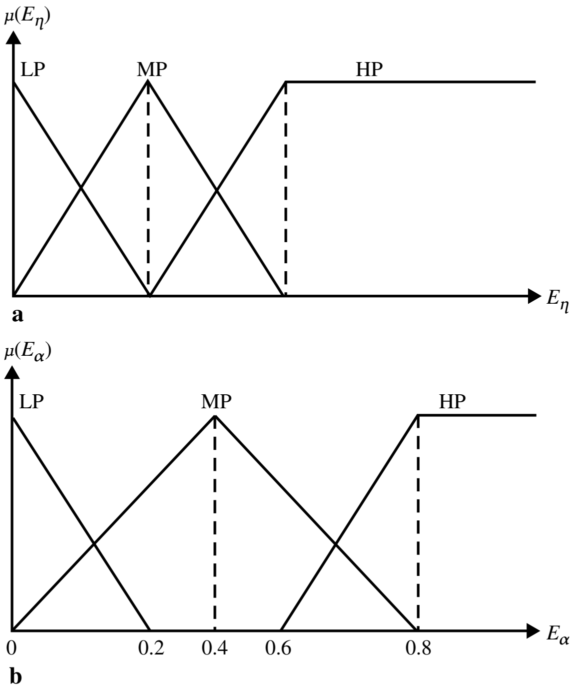


Fig. 3.27a,b Input membership functions of error. **a** Error in learning parameter. **b** Error in momentum parameter

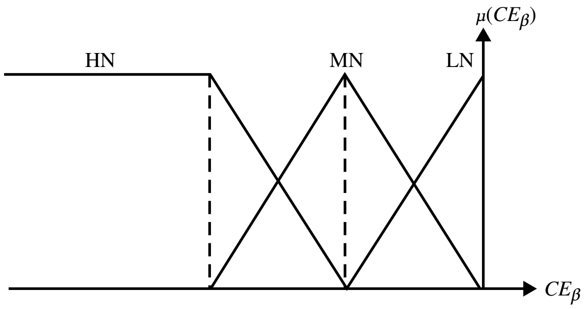


Fig. 3.28 Input membership functions of change in error

Table 3.7 Rules for changing the learning rate

$E\backslash CE$	LN	MN	HN
LP	ZE	ZE	LN
MP	LP	ZE	ZE
HP	LP	LP	ZE

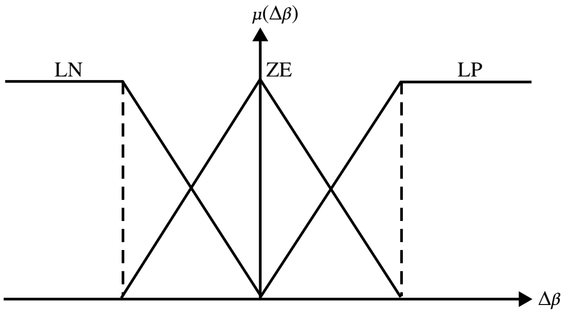
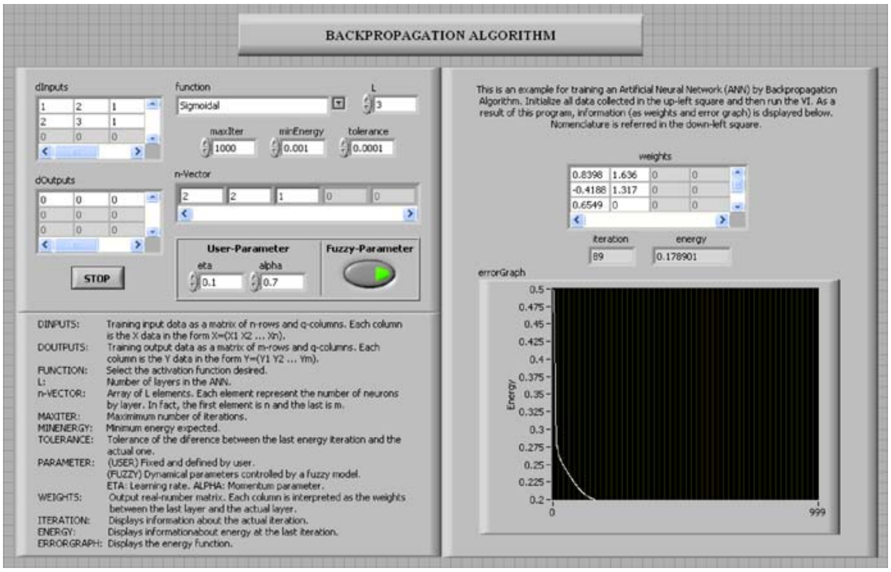


Fig. 3.29 Output membership functions of change in the parameter selected

Table 3.8 Rules for changing the momentum parameter

$E\backslash CE$	LN	MN	HN
LP	ZE	LN	LN
MP	ZE	LN	LN
HP	LP	ZE	ZE


BACKPROPAGATION ALGORITHM

dInputs

1	2	1
2	3	1
0	0	0

dOutputs

0	0	0
0	0	0
0	0	0

function: Sigmoidal
L: 3
maxIter: 1000
minEnergy: 0.001
tolerance: 0.0001

n-Vector

2	2	1	0	0

User-Parameter
eta: 0.1
alpha: 0.7

Fuzzy-Parameter

STOP

DINPUTS:	Training input data as a matrix of n-rows and q-columns. Each column is the X data in the form X=(X1 X2 ... Xn).
DOUTPUTS:	Training output data as a matrix of m-rows and q-columns. Each column is the Y data in the form Y=(Y1 Y2 ... Ym).
FUNCTION:	Select the activation function desired.
L:	Number of layers in the ANN.
n-VECTOR:	Array of L elements. Each element represent the number of neurons by layer. In fact, the first element is n and the last is m.
MAXITER:	Maximimum number of iterations.
MINENERGY:	Minimum energy expected.
TOLERANCE:	Tolerance of the diference between the last energy iteration and the actual one.
PARAMETER:	(USER) Fixed and defined by user. (FUZZY) Dynamical parameters controlled by a fuzzy model. ETA: Learning rate. ALPHA: Momentum parameter.
WEIGHTS:	Output real-number matrix. Each column is interpreted as the weights between the last layer and the actual layer.
ITERATION:	Displays information about the actual iteration.
ENERGY:	Displays informationabout energy at the last iteration.
ERRORGRAPH:	Displays the energy function.

This is an example for training an Artificial Neural Network (ANN) by Backpropagation Algorithm. Initialize all data collected in the up-left square and then run the VI. As a result of this program, information (as weights and error graph) is displayed below. Nomenclature is referred in the down-left square.

weights

0.8398	1.636	0	0
-0.4188	1.317	0	0
0.6549	0	0	0

iteration: 89
energy: 0.178901

Fig. 3.30 Backpropagation algorithm with parameter adjusted using fuzzy logic

3.3.3 Trigonometric Neural Networks

In the previous neural networks, we saw that supervised and feed-forward neural models need to be trained by iterative methods. This situation increases the time of convergence of the learning procedure. In this section, we introduce a trigonometric-based neural network.

First, as we know, a Fourier series is used to approximate a periodic function $f(t)$ with constant period T. It is well known that any function can be approximated by a Fourier series, and so this type of network is used for periodic signals.

Consider a function as in (3.24):

$$\begin{aligned} f(t) &= \frac{1}{2}a_0 + a_1 \cos \omega_0 t + a_2 \cos 2\omega_0 t + \ldots + b_1 sen\omega_0 t + b_2 sen 2\omega_0 t + \ldots \\ f(t) &= \frac{1}{2}a_0 + \sum_{n=1}^{\infty} [a_n \cos(n\omega_0 t) + b_n sen(n\omega_0 t)] \\ f(t) &= C_0 + \sum_{n=1}^{\infty} C_n \cos(n\omega_0 t - \Theta n) . \end{aligned} \tag{3.24}$$

Looking at the neural networks described above, this series is very similar to the mathematical neural model when the activation function is linear:

$$y = x_0 + \sum_{i=1}^{n} w_i x_i . \tag{3.25}$$

Comparing (3.24) and (3.25), we see that they are very close in form, except for the infinite terms of the sum. However, this is not a disadvantage. On the contrary, if we truncate the sum to N terms, then we produce an error in the approximation. This is clearly helpful in neural networks because we do not need them to be memorized.

Thus, a trigonometric neural network (T-ANN) is a Fourier-based net . Figure 3.30 shows this type of neural model. As we might suppose, T-ANN are able to compute with cosine functions or with sine functions. This selection is arbitrary.

Considering its learning procedure, a Fourier series can be solved analytically by employing least square estimates (LSE). This process means that we want to find coefficients that preserve the minimum value of the function

$$S(a_0, a_1, \ldots, a_n) = \sum_{i=1}^{m} \left[y_i - \left(\frac{1}{2}a_0 + \sum_{k=1}^{\infty} a_k \cos(k\omega_0 x_i) \right) \right]^2 , \tag{3.26}$$

where ω_0 is the fundamental frequency of the series, x_i is the ith input data and y_i is the ith value of the desired output. Then, we need the first derivative of that function, which is:

$$\frac{\delta S}{\delta a_p} = \sum_{i=1}^{m} \left[y_i - \left(\frac{1}{2}a_0 + \sum_{k=1}^{\infty} a_k \cos(k\omega_0 x_i) \right) \cos(n\omega_0 x) \right] = 0, \quad \forall p \geq 1 . \tag{3.27}$$

This is a system of linear equations that can be viewed as:

$$\begin{bmatrix} \frac{1}{2}m & \cdots & \sum_{i=1}^{m}\cos(p\omega_0 x_i) \\ \frac{1}{2}\sum_{i=1}^{m}\cos(\omega_0 x_i) & \cdots & \sum_{i=1}^{m}\cos(\omega_0 x_i)\cos(p\omega_0 x_i) \\ \vdots & \ddots & \sum_{i=1}^{m}\cos(p\omega_0 x_i)\cos(p\omega x_i) \\ \frac{1}{2}\sum_{i=1}^{m}\cos(p\omega_0 x_i) & \cdots & \sum_{i=1}^{m}\cos^2(p\omega_0 x_i) \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^{m} y_i \\ \sum_{i=1}^{m} y_i\cos(\omega_0 x_i) \\ \vdots \\ \sum_{i=1}^{m} y_i\cos(p\omega_0 x_i) \end{bmatrix}. \quad (3.28)$$

Then, we can solve this system for all coefficients. At this point, p is the number of neurons that we want to use in the T-ANN . In this way, if we have a data collection of the input/output desired values, then we can compute analytically the coefficients of the series or what is the same, the weights of the net. Algorithm 3.4 is proposed for training T-ANNs ; eventually, this procedure can be computed with the backpropagation algorithm as well.

Algorithm 3.4	T-ANNs
Step 1	Determine input/output desired samples. Specify the number of neurons N.
Step 2	Evaluate weights C_i by LSE.
Step 3	*STOP.*

Example 3.4. Approximate the function $f(x) = x^2 + 3$ in the interval $[0, 5]$ with: (a) 5 neurons, (b) 10 neurons, (c) 25 neurons. Compare them with the real function. *Solution.* We need to train a T-ANN and then evaluate this function in the interval $[0, 5]$. First, we access the VI that trains a T-ANN following the path ICTL $\gg$ ANNs $\gg$ T-ANN $\gg$ **entrenaRed.vi**. This VI needs the **x**-vector coordinate, **y**-vector coordinate and the number of neurons that the network will have.

In these terms, we have to create an array of elements between $[0, 5]$ and we do this with a stepsize of 0.1, by the **rampVector.vi**. This array evaluates the function $x^2 + 3$ with the program inside the for-loop in Fig. 3.31. Then, the array coming from the **rampVector.vi** is connected to the x pin of the **entrenaRed.vi**, and the array coming from the evaluated **x**-vector is connected to the y pin. Actually, the pin n is available for the number of neurons. Then, we create a control variable for neurons because we need to train the network with a different number of neurons.

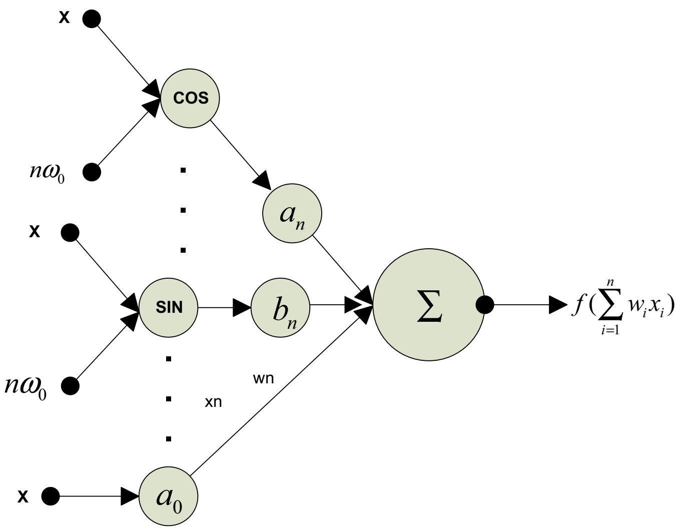


Fig. 3.31 T-ANN model

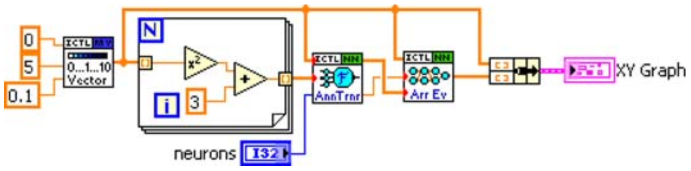


Fig. 3.32 Block diagram of the training and evaluating T-ANN

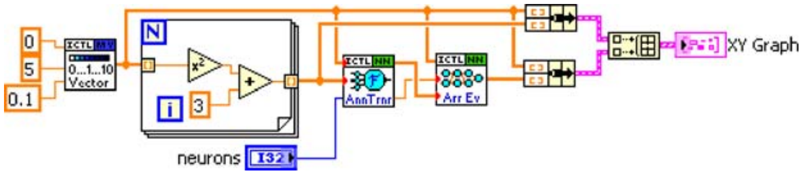


Fig. 3.33 Block diagram for plotting the evaluating T-ANN against the real function

This VI is then connected to another VI that returns the values of a T-ANN. This last node is found in the path ICTL ≫ ANNs ≫ T-ANN ≫ **Arr_Eval_T-ANN.vi**. This receives the coefficients that were the result of the previous VI named *T-ANN Coeff* pin connector. The *Fund Freq* connector is referred to the fundamental frequency of the trigonometric series ω_0. This value is calculated in the **entrenaRed.vi**. The last pin connector is referred to as *Values*. This pin is a 1D array with the values in the x-coordinate, which we want to evaluate the neural network. The result of this VI is the output signal of the T-ANN by the pin *T-ANN Eval*. The block diagram of this procedure is given in Fig. 3.32.

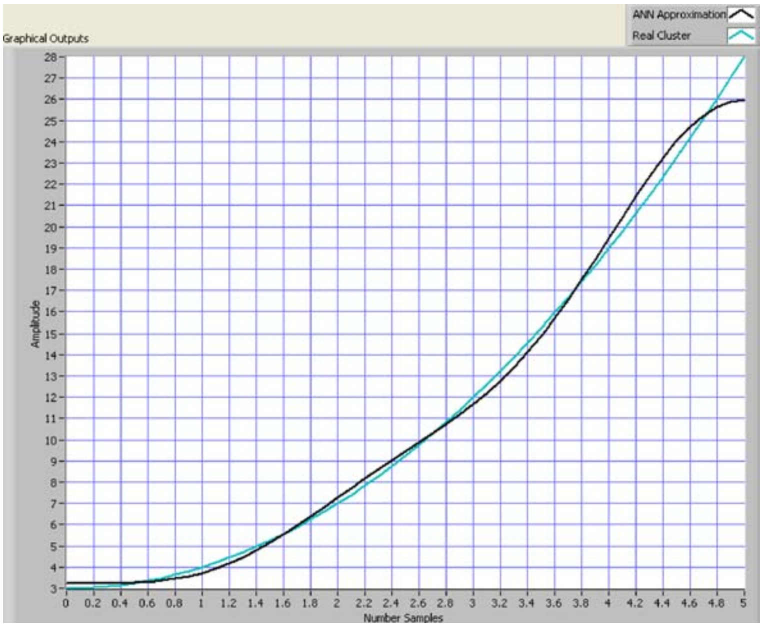


Fig. 3.34 Approximation function with T-ANN with 5 neurons

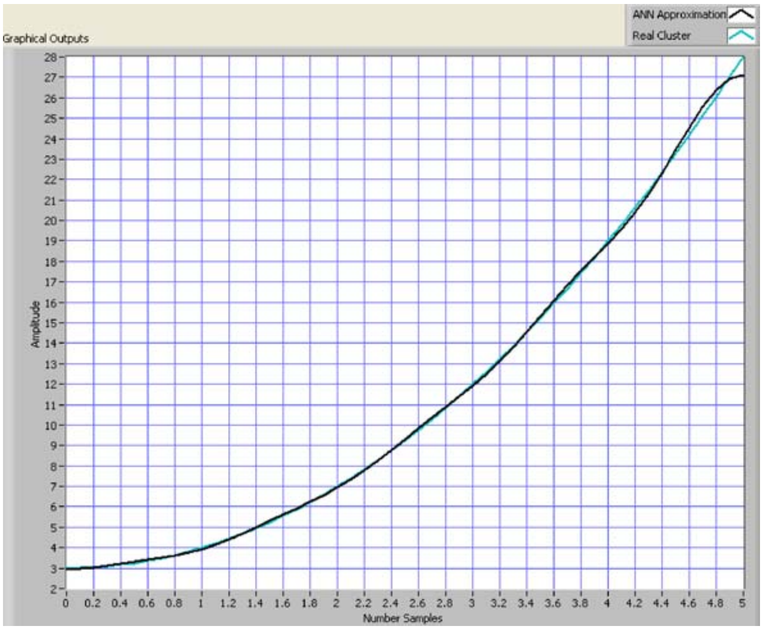


Fig. 3.35 Approximation function with T-ANN with 10 neurons

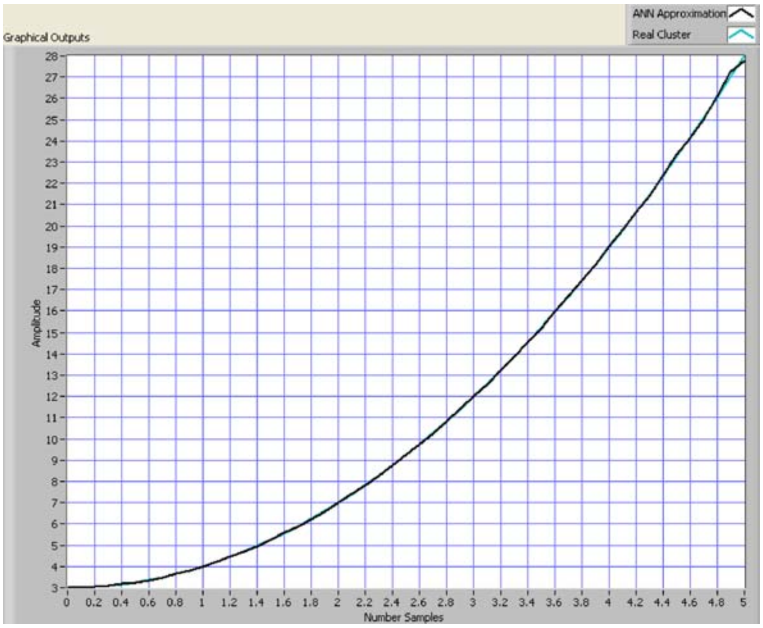


Fig. 3.36 Approximation function with T-ANN with 25 neurons

To compare the result with the real value we create a cluster of two arrays, one comes from the **rampVector.vi** and the other comes from the output of the for-loop. Figure 3.33 shows the complete block diagram. As seen in Figs. 3.34–3.36, the larger the number of neurons, the better the approximation. To generate each of these graphs, we only vary the value of neurons. □

3.3.3.1 Hebbian Neural Networks

A Hebbian neural network is an unsupervised and competitive net. As unsupervised networks, these only have information about the input space, and their training is based on the fact that the weights store the information. Thus, the weights can only be reinforced if the input stimulus provides sufficient output values. In this way, weights only change proportionally to the output signals. By this fact, neurons compete to become a dedicated reaction of part of the input. Hebbian neural networks are then considered as the first self-organizing nets .

The learning procedure is based on the following statement pronounced by Hebb: As A becomes more efficient at stimulating B during training, A sensitizes B to its stimulus, and the weight on the connection from A to B increases during training as B becomes sensitized to A.

Steven Grossberg then developed a mathematical model for this sentence, given in (3.29):

$$w_{AB}^{\text{new}} = w_{AB}^{\text{old}} + \beta x_B x_A \,, \tag{3.29}$$

where w_{AB} is the weight between the interaction of two neurons A and B, x_i is the output signal of the ith neuron, and $x_B x_A$ is the so-called Hebbian learning term. Algorithm 3.5 introduces the Hebbian learning procedure.

Algorithm 3.5	Hebbian learning procedure
Step 1	Determine the input space.
	Specify the number of iterations $iterNum$ and initialize $t = 0$.
	Generate small random values of weights w_i.
Step 2	Evaluate the Hebbian neural network and obtain the outputs x_i.
Step 3	Apply the updating rule (3.29).
Step 4	If $t = iterNum$ then *STOP*.
	Else, go to *Step 2*.

These types of neural models are good when no desired output values are known. Hebbian learning can be applied in multi-layer structures as well as feed-forward and feed-back networks.

Example 3.5. There are points in the following data. Suppose that this data is some input space. Apply Algorithm 3.5 with a forgotten factor of 0.1 to train a Hebbian network that approximates the data presented in Table 3.9 and Fig. 3.37.

Table 3.9 Data points for the Hebbian example

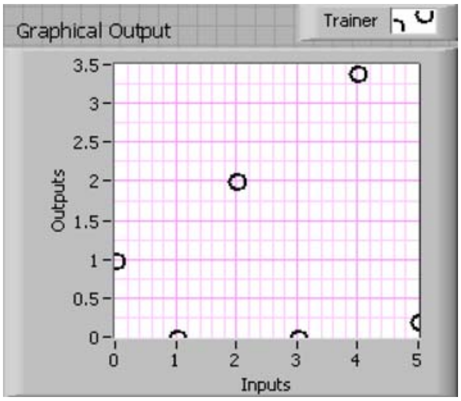
X-coordinate	Y-coordinate
0	1
1	0
2	2
3	0
4	3.4
5	0.2

Solution. We consider a 0.1 of the learning rate value. The forgotten factor α is applied with the following equation:

$$w_{AB}^{\text{new}} = w_{AB}^{\text{old}} - \alpha w_{AB}^{\text{old}} + \beta x_B x_A \,. \tag{3.30}$$

We go to the path ICTL $\gg$ ANNs $\gg$ Hebbian $\gg$ **Hebbian.vi**. This VI has input connectors of the y-coordinate array, called x pin, which is the array of the desired values, the forgotten factor a, the learning rate value b, and the *Iterations* variable.

Fig. 3.37 Input training data



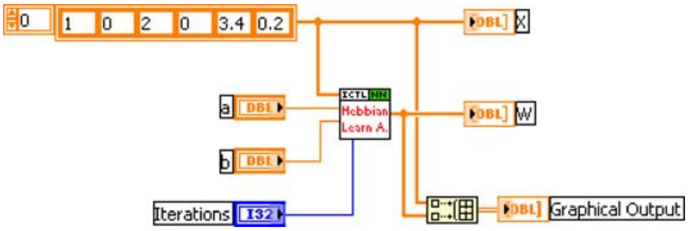
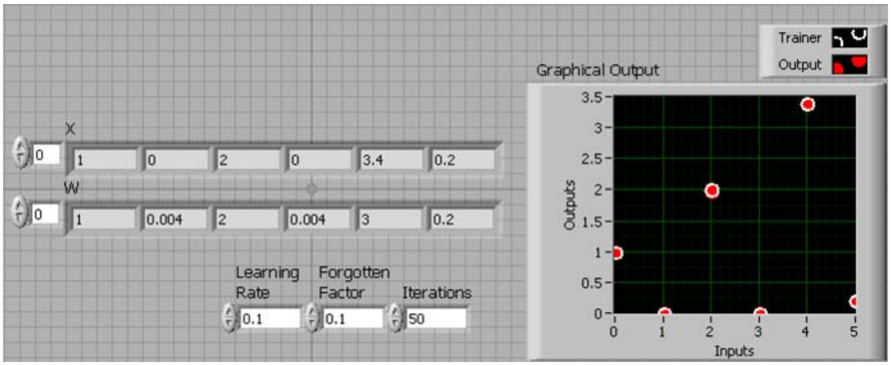


Fig. 3.38 Block diagram for training a Hebbian network

This last value is selected in order to perform the training procedure by this number of cycles. The output of this VI is the weight vector, which is the y-coordinate of the approximation to the desired values. The block diagram for this procedure is shown in Fig. 3.38.

Then, using Algorithm 3.5 with the above rule with forgotten factor, the result looks like Fig. 3.39 after 50 iterations. The vector $\boldsymbol{W}$ is the y-coordinate approximation of the y-coordinate of the input data. Figure 3.39 shows the training procedure. □



X: 0 | 1 | 0 | 2 | 0 | 3.4 | 0.2

W: 0 | 1 | 0.004 | 2 | 0.004 | 3 | 0.2

Learning Rate: 0.1

Forgotten Factor: 0.1

Iterations: 50

Fig. 3.39 Result of the Hebbian process in a neural network

3.3.4 Kohonen Maps

Kohonen networks or self-organizing maps are a competitive training neural network aimed at ordering the mapping of the input space. In competitive learning, we normally have distributed input $x = x(t) \in \Re^n$, where t is the time coordinate, and a set of reference vectors $\boldsymbol{m}_i = \boldsymbol{m}_i(t) \in \Re^n, \forall i = 1, \ldots, k$. The latter are initialized randomly. After that, given a metric $d(x, m_i)$ we try to minimize this function to find a reference vector that best matches the input. The best reference vector is named $\boldsymbol{m}_c$ (the winner) where c is the best selection index. Thus, $d(x, \boldsymbol{m}_c)$ will be the minimum metric. Moreover, if the input x has a density function $p(x)$, then, we can minimize the error value between the input space and the set of reference vectors, so that all $\boldsymbol{m}_i$ can represent the form of the input as much as possible. However, only an iterative process should be used to find the set of reference vectors.

At each iteration, vectors are actualized by the following equation:

$$\boldsymbol{m}_i(t+1) = \begin{cases} \boldsymbol{m}_i(t) + \alpha(t) \cdot d[x(t), m_i(t)] & i = c \\ \boldsymbol{m}_i(t) & i \neq c \end{cases}, \tag{3.31}$$

where $\alpha(t)$ is a monotonically decreasing function with scalar values between 0 and 1. This method is known as vector quantization (VQ) and looks to minimize the error, considering the metric as a Euclidean distance with r-power:

$$E = \int \|x - \boldsymbol{m}_c\|^r \, p(x) \, \mathrm{d}x \,. \tag{3.32}$$

On the other hand, years of studies on the cerebral cortex have discovered two important things: (1) the existence of specialized regions, and (2) the ordering of these regions. Kohonen networks create a competitive algorithm based on these facts in order to adjust specialized neurons into subregions of the input space, and if this input is ordered, specialized neurons also perform an ordering space (mapping). A typical Kohonen network N is shown in Fig. 3.40.

If we suppose an n-dimensional input space $\boldsymbol{X}$ is divided into subregions x_i, and a set of neurons with a d-dimensional topology, where each neuron is associated to a n-dimensional weight m_i (Fig. 3.40), then this set of neurons forms a space N. Each subregion of the input will be mapped by a subregion of the neuron space. Moreover, mapped subregions will have a specific order because input subregions have order as well.

Kohonen networks emulate the behavior described above, which is defined in Algorithm 3.6.

As seen in the previous algorithm, VQ is used as a basis. To achieve the goal of ordering the weight vectors, one might select the winner vector and its neighbors to approximate the interesting subregion. The number of neighbors v should be a monotonically decreasing function with the characteristic that at the first iteration the network will order uniformly, and then, just the winner neuron will be reshaped to minimize the error.

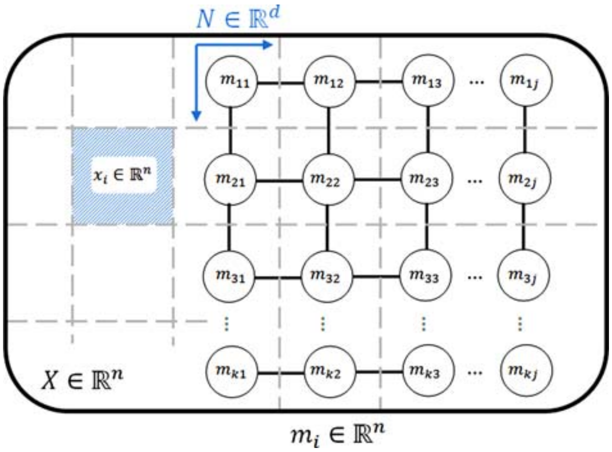


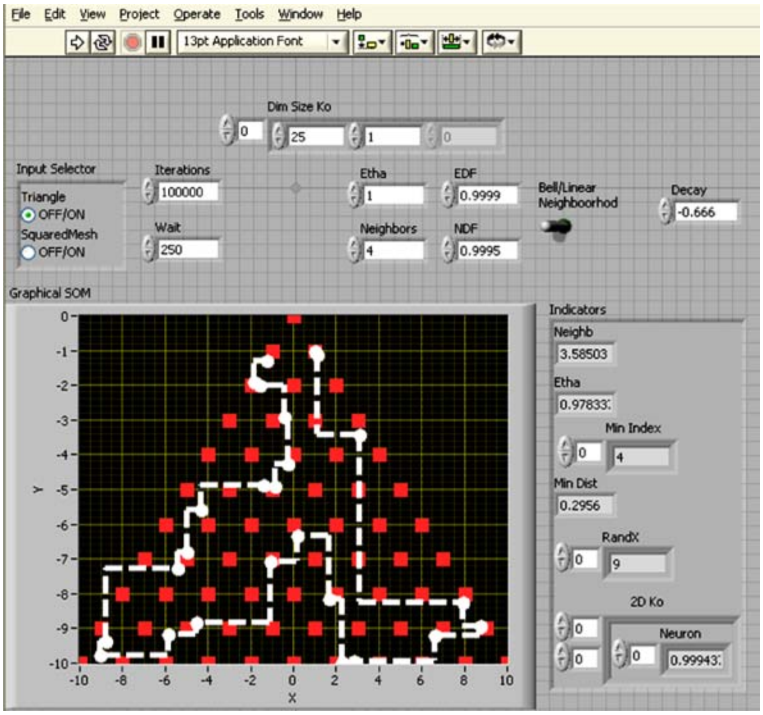
Fig. 3.40 Kohonen network N approximating the input space $\boldsymbol{X}$

Algorithm 3.6	Kohonen learning procedure
Step 1	Initialize the number of neurons and the dimension of the Kohonen network. Associate a weight vector m_i to each neuron, randomly.
Step 2	Determine the configuration of the neighborhood N_c of the weight vector considering the number of neighbors v and the neighborhood distribution $v(c)$.
Step 3	Randomly, select a subregion of the input space $x(t)$ and calculate the Euclidean distance to each weight vector.
Step 4	Determine the winner weight vector m_c (the minimum distance defines the winner) and actualize each of the vectors by (3.31) which is a discrete-time notation.
Step 5	Decrease the number of neighbors v and the learning parameter α.
Step 6	Use a statistical parameter to determine the approximation between neurons and the input space. If neurons approximate the input space then *STOP*. Else, go to *Step 2*.

Moreover, the training function or learning parameter will be decreased. Figure 3.41 shows how the algorithm is implemented. Some applications of this kind of network are: pattern recognition, robotics, control process, audio recognition, telecommunications, etc.

Example 3.6. Suppose that we have a square region in the interval $x \in [-10, 10]$ and $y \in [-10, 10]$. Train a 2D-Kohonen network in order to find a good approximation to the input space.

Solution. This is an example inside the toolkit, located in ICTL $\gg$ ANNs $\gg$ Kohonen SOM $\gg$ **2DKohonen_Example.vi**. The front panel is the same as in Fig. 3.42, with the following sections.



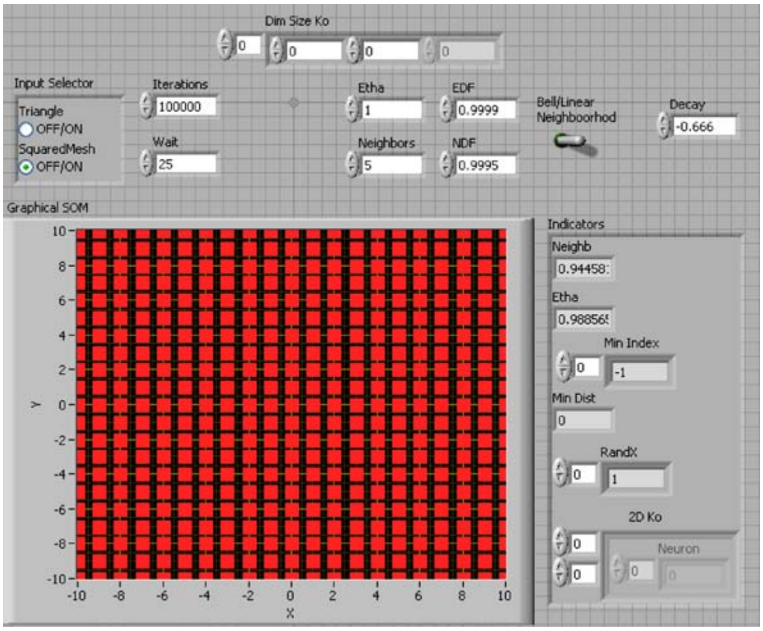
Dim Size Ko
0 25 1 0

Input Selector
Triangle [✓ OFF/ON]
SquaredMesh OFF/ON

Iterations 100000
Wait 250
Etha 1
Neighbors 4
EDF 0.9999
NDF 0.9995
Bell/Linear Neighboorhod
Decay -0.666

Indicators
Neighb 3.58503
Etha 0.97833
Min Index 0 4
Min Dist 0.2956
RandX 0 9
2D Ko 0 0
Neuron 0 0.99943

Fig. 3.41 One-dimensional Kohonen network with 25 neurons (*white dots*) implemented to approximate the triangular input space (*red subregions*)



Dim Size Ko
0 0 0 0

Input Selector
Triangle OFF/ON
SquaredMesh [✓ OFF/ON]

Iterations 100000
Wait 25
Etha 1
Neighbors 5
EDF 0.9999
NDF 0.9995
Bell/Linear Neighboorhod
Decay -0.666

Indicators
Neighb 0.94458
Etha 0.98856
Min Index 0 -1
Min Dist 0
RandX 0 1
2D Ko 0 0
Neuron 0 0

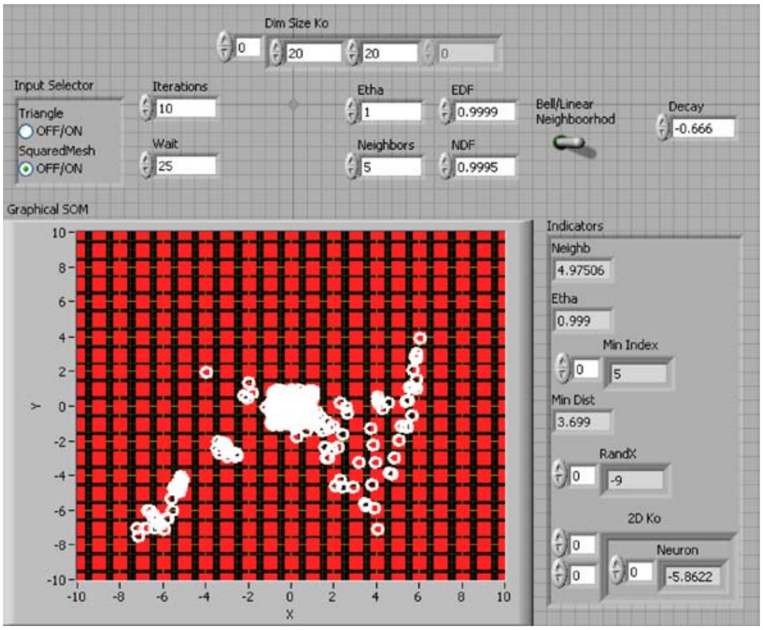
Fig. 3.42 Front panel of the 2D-Kohonen example

We find the input variables at the top of the window. These variables are *Dim Size Ko*, which is an array in which we represent the number of neurons per coordinate system. In fact, this is an example of a 2D-Kohonen network, and the dimension of the Kohonen is 2. This means that it has an x-coordinate and a y-coordinate. In this case, if we divide the input region into 400 subregions, in other words, we have an interval of 20 elements per 20 elements in a square space, then we may say that we need 20 elements in the x-coordinate and 20 elements in the y-coordinate dimension. Thus, we are asking for the network to have 400 nodes.

Etha is the learning rate, *EDF* is the learning rate decay factor, *Neighbors* represents the number of neighbors that each node has and its corresponding *NDF* or neighbor decay factor. *EDF* and *NDF* are scalars that decrease the value of *Etha* and *Neighbors*, respectively, at each iteration. After that we have the *Bell/Linear Neighborhood* switch. This switches the type of neighborhood between a bell function and a linear function. The value *Decay* is used as a factor of fitness in the bell function. This has no action in the linear function.

On the left side of the window is the *Input Selector*, which can select two different input regions. One is a triangular space and the other is the square space treated in this example. The value *Iterations* is the number of cycles that the Kohonen network takes to train the net. *Wait* is just a timer to visualize the updating network.

Finally, on the right side of the window is the *Indicators* cluster. It rephrases values of the actual *Neighbor* and *Etha*. *Min Index* represents the indices of the winner node. *Min Dist* is the minimum distance between the winner node and the


Dim Size Ko: 0, 20, 20, 0

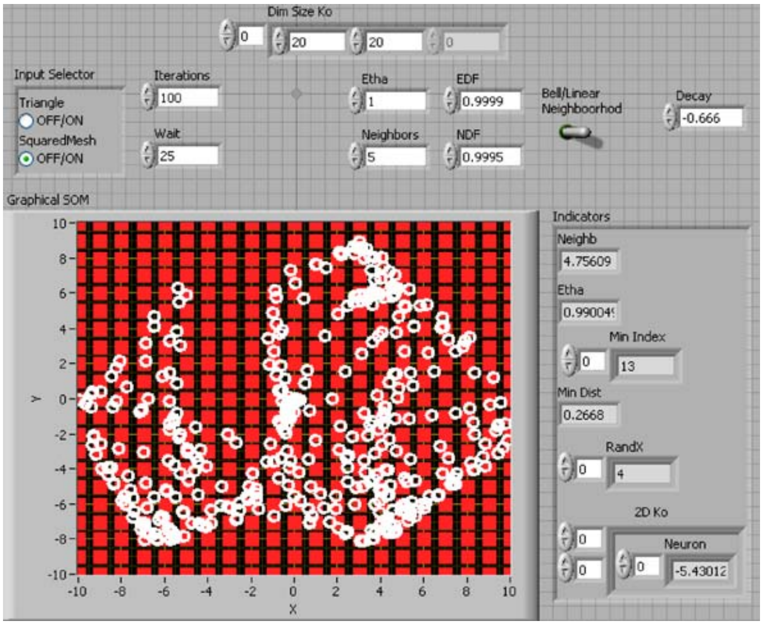
Input Selector
Triangle OFF/ON
SquaredMesh [✓ OFF/ON]

Iterations: 10
Wait: 25
Etha: 1
EDF: 0.9999
Neighbors: 5
NDF: 0.9995
Bell/Linear Neighboorhod
Decay: -0.666

Graphical SOM

Indicators
Neighb: 4.97506
Etha: 0.999
Min Index: 0, 5
Min Dist: 3.699
RandX: 0, -9
2D Ko: 0, 0
Neuron: 0, -5.8622

Fig. 3.43 The 2D-Kohonen network at 10 iterations



Dim Size Ko: 0 | 20 | 20 | 0

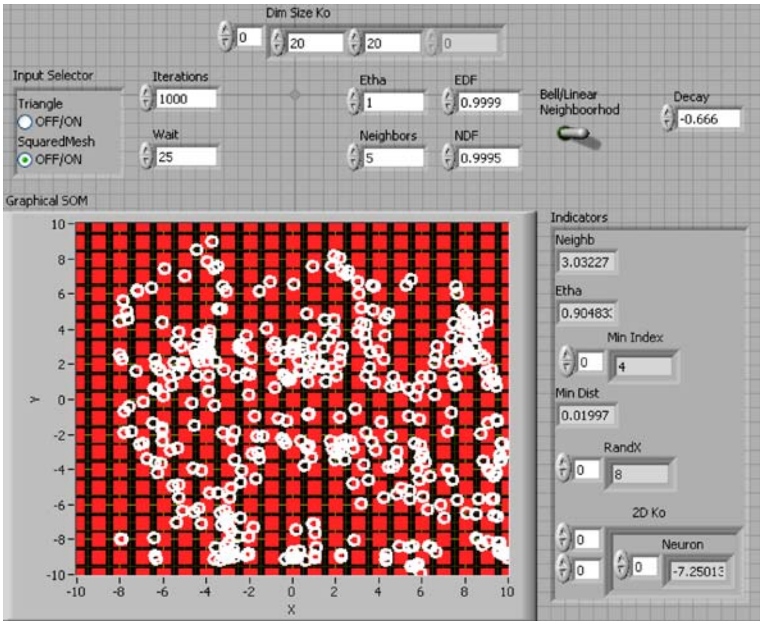
Input Selector
Triangle
OFF/ON
SquaredMesh
[✓ OFF/ON]

Iterations: 100
Wait: 25
Etha: 1
Neighbors: 5
EDF: 0.9999
NDF: 0.9995
Bell/Linear Neighboorhod
Decay: -0.666

Graphical SOM

Indicators
Neighb: 4.75609
Etha: 0.99004(
Min Index: 0 | 13
Min Dist: 0.2668
RandX: 0 | 4
2D Ko: 0 | 0
Neuron: 0 | -5.43012

Fig. 3.44 The 2D-Kohonen network at 100 iterations



Dim Size Ko: 0 | 20 | 20 | 0

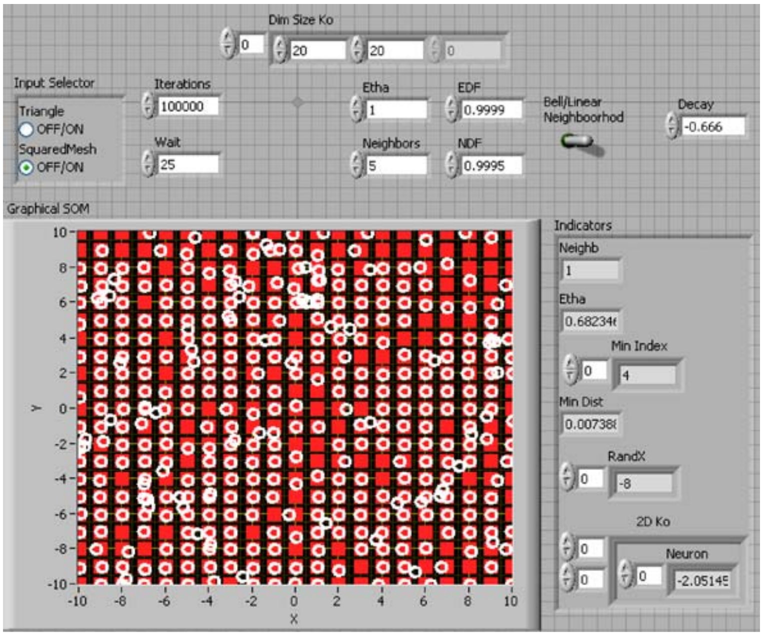
Input Selector
Triangle
OFF/ON
SquaredMesh
[✓ OFF/ON]

Iterations: 1000
Wait: 25
Etha: 1
Neighbors: 5
EDF: 0.9999
NDF: 0.9995
Bell/Linear Neighboorhod
Decay: -0.666

Graphical SOM

Indicators
Neighb: 3.03227
Etha: 0.90483(
Min Index: 0 | 4
Min Dist: 0.01997
RandX: 0 | 8
2D Ko: 0 | 0
Neuron: 0 | -7.2501(

Fig. 3.45 The 2D-Kohonen network at 1000 iterations


Dim Size Ko 0 20 20 0

Input Selector
Triangle
[OFF/ON]
SquaredMesh
[✓ OFF/ON]

Iterations 100000
Wait 25
Etha 1
Neighbors 5
EDF 0.9999
NDF 0.9995
Bell/Linear Neighboorhod
Decay -0.666

Graphical SOM

Indicators
Neighb 1
Etha 0.68234(
Min Index 0 4
Min Dist 0.00738(
RandX 0 -8
2D Ko 0 0
Neuron 0 -2.0514

Fig. 3.46 The 2D-Kohonen network at 10 000 iterations

close subregion. *RandX* is the subregion selected randomly. *2D Ko* is a cluster of nodes with coordinates. Figures 3.42–3.46 represent the current configuration of the 2D-Kohonen network with five neighbors and one learning rate at the initial conditions, with values of 0.9999 and 0.9995 for *EDF* and *NDF*, respectively. The training was done by a linear function of the neighborhood. □

3.3.5 Bayesian or Belief Networks

This kind of neural model is a directed acyclic graph (DAG) in which nodes have random variables. Basically, a DAG consists of nodes and deterministic directions between links. A DAG can be interpreted as an adjacency matrix in which 0 elements mean no links between two nodes, and 1 means a linking between the ith row and the jth column.

This model can be divided into polytrees and cyclic graphs. Polytrees are models in which the evidence nodes or the input nodes are at the top, and the children are below the structure. On the other hand, cyclic models are any kind of DAG, when going from one node to another node that has at least another path connecting these points. Figure 3.47 shows examples of these structures. For instance, we only consider polytrees in this chapter.

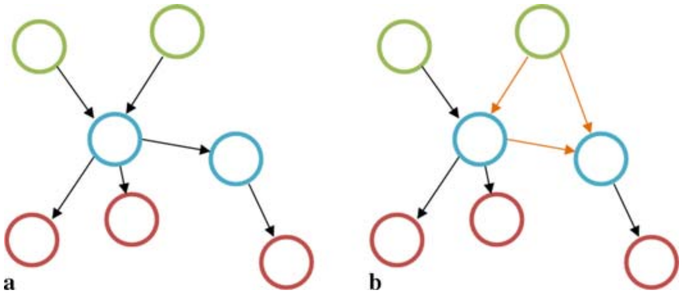


Fig. 3.47a,b Bayesian or belief networks. **a** A polytree. **b** A cyclic structure

Bayesian networks or belief networks have a node V_i that is conditionally independent from a subset of nodes that are not descendents of V_i given its parents $P(V_i)$. Suppose that we have $V_1, \ldots, V_k$ nodes of a Bayesian network and they are conditionally independent. The joint probability of all nodes is:

$$p(V_1, \ldots, V_k) = \prod_{i=1}^{k} p(V_i | P(V_i)) . \tag{3.33}$$

These networks are based on tables of probabilities known as conditional probability tables (CPT), in which the node is related to its parents by probabilities.

Bayesian networks can be trained by some algorithms, such as the expectation-maximization (EM) algorithm or the gradient-ascent algorithm. In order to understand the basic idea of training a Bayesian network, a gradient-ascent algorithm will be described in the following.

We are looking to maximize the likelihood hypothesis $\ln P(D|h)$ in which P is the probability of the data D given hypothesis h. This maximization will be performed with respect to the parameters that define the CPT. Then, the expression derived from this fact is:

$$\frac{\partial \ln P(D|h)}{\partial w_{ij}} = \sum_{d \in D} \frac{P(Y_i = y_{ij}, U_i = u_{ik} | d)}{w_{ijk}} \tag{3.34}$$

where y_{ij} is the j-value of the node Y_i, U_i is the parent with the k-value u_{ik}, w_{ijk} is the value of the probability in the CPT relating y_{ij} with u_{ik}, and d is a sample of the training data D. In Algorithm 3.7 this training is described.

Example 3.7. Figure 3.48 shows a DAG. Represent this graph in an adjacency matrix (it is a cyclic structure).
Solution. Here, we present the matrix in Fig. 3.49. Graph theory affirms that the adjacency matrix is unique. Therefore, the solution is unique. □

Example 3.8. Train the network in Fig. 3.48 for the data sample shown in Table 3.10. Each column represents a node. Note that each node has targets $Y_i = \{0, 1\}$.

Algorithm 3.7 Gradient-ascent learning procedure for Bayesian networks

```
Step 1    Generate a CPT with random values of probabilities.
          Determine the learning rate $\eta$.
Step 2    Take a sample $d$ of the training data $D$ and determine the probability on
          the right-hand side of (3.34).
Step 3    Update the parameters with
          $w_{ijk} \leftarrow w_{ijk} + \eta \sum_{d \in D} \frac{P(Y_i = y_{ij}, U_i = u_{ik} | d)}{w_{ijk}}$.
Step 4    If $CPT_t = CPT_{t-1}$ then STOP.
          Else, go to Step 2 until reached.
```

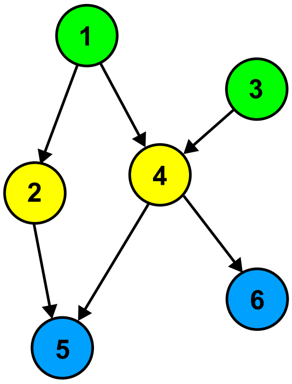
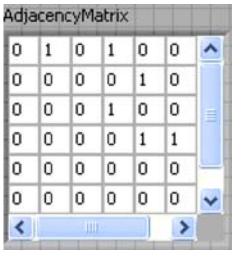


Fig. 3.48 DAG with evidence nodes 1 and 3, and query nodes 5 and 6. The others are known as hidden nodes

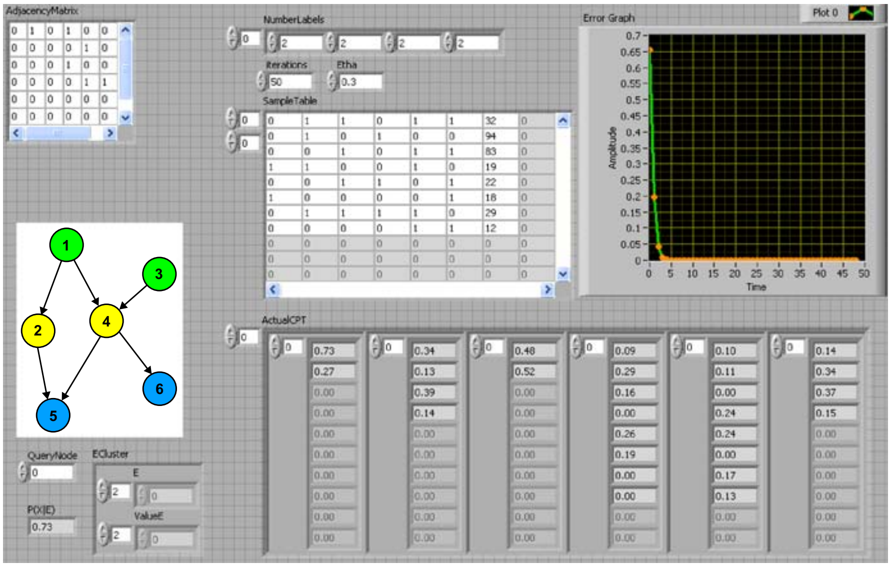


0	1	0	1	0	0
0	0	0	0	1	0
0	0	0	1	0	0
0	0	0	0	1	1
0	0	0	0	0	0
0	0	0	0	0	0

Fig. 3.49 Adjacency matrix for the DAG in Fig. 3.48

Table 3.10 Bayesian networks example

Node 1	Node 2	Node 3	Node 4	Node 5	Node 6	Frequency
0	1	1	0	1	1	32
0	1	0	1	0	0	94
0	0	1	0	1	1	83
1	1	0	0	1	0	19
0	0	1	1	0	1	22
1	0	0	0	0	1	18
0	1	1	1	1	0	29
0	0	0	0	1	1	12



AdjacencyMatrix

0	1	0	1	0	0
0	0	0	0	1	0
0	0	0	1	0	0
0	0	0	0	1	1
0	0	0	0	0	0
0	0	0	0	0	0

NumberLabels: 0; 2, 2, 2, 2

Iterations: 50   Etha: 0.3

SampleTable: 0, 0

0	1	1	0	1	1	32	0
0	1	0	1	0	0	94	0
0	0	1	0	1	1	83	0
1	1	0	0	1	0	19	0
0	0	1	1	0	1	22	0
1	0	0	0	0	1	18	0
0	1	1	1	1	0	29	0
0	0	0	0	1	1	12	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

ActualCPT: 0; 0, 0, 0, 0, 0, 0

0.73	0.34	0.48	0.09	0.10	0.14
0.27	0.13	0.52	0.29	0.11	0.34
0.00	0.39	0.00	0.16	0.00	0.37
0.00	0.14	0.00	0.00	0.24	0.15
0.00	0.00	0.00	0.26	0.24	0.00
0.00	0.00	0.00	0.19	0.00	0.00
0.00	0.00	0.00	0.00	0.17	0.00
0.00	0.00	0.00	0.00	0.13	0.00
0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00

QueryNode: 0

P(X|E): 0.73

ECluster: E: 2, 0; ValueE: 2, 0

Fig. 3.50 Training procedure of a Bayesian network

Solution. This example is located at ICTL $\gg$ ANNs $\gg$ Bayesian $\gg$ **Bayes_Example.vi**. Figure 3.50 shows the implementation of Algorithm 3.7. At the top-left side of the window, we have the adjacency matrix in which we represent the DAG as seen in Example 3.7. Then, *NumberLabels* represents all possible labels that the related node can have. In this case, we have that all nodes can only take values between 0 or 1, then each node has two labels. Therefore, the array is $NumberLabels = \{2, 2, 2, 2, 2, 2\}$. *Iterations* is the same as in the other examples. *Etha* is the learning rate in the gradient-ascent algorithm. *SampleTable* comes from experiments and measures the frequency that some combination of nodes is fired. In this example, the table is the sample data given in the problem.

The *Error Graph* shows how the measure of error is decreasing when time is large. Finally, *ActualCPT* is the result of the training procedure and it is the CPT of the Bayesian network. For instance, we choose a value of learning rate that equals 0.3 and 50 iterations to this training procedure. As we can see, the training needs around five iterations to obtain the CPT. This table contains the training probabilities that relate each node with its immediate parents. □

References

1. Lakhmi J, Rao V (1999) Industrial applications of neural networks. CRC, Boca Raton, FL
2. Weifeng S, Tianhao T (2004) CMAC Neural networks based combining control for marine diesel engine generator. IEEE Proceedings of the 5th World Congress on Intelligent Control, Hangzshou, China, 15–19 June 2004

3. Ananda M, Srinivas J (2003) Neural networks. Algorithms and applications. Alpha Sciences International, Oxford
4. Samarasinghe S (2007) Neural networks for applied sciences and engineering. Auerbach, Boca Raton, FL
5. Irwin G, et al. (1995) Neural network applications in control. The Institution of Electrical Engineers, London
6. Mitchell T (1997) Machine learning. McGraw-Hill, Boston
7. Kohonen T (1990) The self-organizing map. Proceedings of the IEEE 78(9):1464–1480
8. Rojas R (1996) Neural networks. Kohonen networks, Chap 15. Springer, Berlin Heidelberg New York, pp 391–412
9. Veksler O (2004) Lecture 18: Pattern recognition. University of Western Ontario, Computer Science Department. http://courses.media.mit.edu/2004fall/mas622j. Accessed on 10 March 2009
10. Jensen F (2001) Bayesian networks and decision graphs. Springer, Berlin Heidelberg New York
11. Nilsson N (2001) Inteligencia artificial, una nueva síntesis. McGraw-Hill, Boston
12. Nolte J (2002) The human brain: an introduction to its functional anatomy. Mosby, St. Louis, MO
13. Affi A, Bergman R (2005) Functional neuroanatomy text and atlas. McGraw-Hill, Boston
14. Nguyen H, et al. (2003) A first course in fuzzy and neural control. Chapman & Hall/CRC, London
15. Ponce P (2004) Trigonometric Neural Networks internal report. ITESM-CCM, México City

Futher Reading

Hertz J, Krogh A, Lautrup B, Lehmann T (1997) Nonlinear backpropagation: doing backpropagation without derivatives of the activation function. Neural Networks, IEEE Transactions 8:1321–1327

Loh AP, Fong KF (1993) Backpropagation using generalized least squares. Neural Networks, IEEE International Conference 1:592–597

McLauchlan LLL, Challoo R, Omar SI, McLauchlan RA (1994) Supervised and unsupervised learning applied to robotic manipulator control. American Control Conference, 3:3357–3358

Taji K, Miyake T, Tamura H (1999) On error backpropagation algorithm using absolute error function Systems, Man, and Cybernetics. 1999. IEEE SMC '99 Conference Proceedings, IEEE International Conference 5:401–406