

NI MATRIXx™

SystemBuild™ Neural Network Module

Worldwide Technical Support and Product Information

ni.com

National Instruments Corporate Headquarters

11500 North Mopac Expressway Austin, Texas 78759-3504 USA Tel: 512 683 0100

Worldwide Offices

Australia 1800 300 800, Austria 43 662 457990-0, Belgium 32 (0) 2 757 0020, Brazil 55 11 3262 3599,
Canada 800 433 3488, China 86 21 5050 9800, Czech Republic 420 224 235 774, Denmark 45 45 76 26 00,
Finland 385 (0) 9 725 72511, France 33 (0) 1 48 14 24 24, Germany 49 89 7413130, India 91 80 41190000,
Israel 972 3 6393737, Italy 39 02 413091, Japan 81 3 5472 2970, Korea 82 02 3451 3400,
Lebanon 961 (0) 1 33 28 28, Malaysia 1800 887710, Mexico 01 800 010 0793, Netherlands 31 (0) 348 433 466,
New Zealand 0800 553 322, Norway 47 (0) 66 90 76 60, Poland 48 22 3390150, Portugal 351 210 311 210,
Russia 7 495 783 6851, Singapore 1800 226 5886, Slovenia 386 3 425 42 00, South Africa 27 0 11 805 8197,
Spain 34 91 640 0085, Sweden 46 (0) 8 587 895 00, Switzerland 41 56 2005151, Taiwan 886 02 2377 2222,
Thailand 662 278 6777, Turkey 90 212 279 3031, United Kingdom 44 (0) 1635 523545

For further support information, refer to the *Technical Support and Professional Services* appendix. To comment on National Instruments documentation, refer to the National Instruments Web site at ni.com/info and enter the info code `feedback`.

Important Information

Warranty

The media on which you receive National Instruments software are warranted not to fail to execute programming instructions, due to defects in materials and workmanship, for a period of 90 days from date of shipment, as evidenced by receipts or other documentation. National Instruments will, at its option, repair or replace software media that do not execute programming instructions if National Instruments receives notice of such defects during the warranty period. National Instruments does not warrant that the operation of the software shall be uninterrupted or error free.

A Return Material Authorization (RMA) number must be obtained from the factory and clearly marked on the outside of the package before any equipment will be accepted for warranty work. National Instruments will pay the shipping costs of returning to the owner parts which are covered by warranty.

National Instruments believes that the information in this document is accurate. The document has been carefully reviewed for technical accuracy. In the event that technical or typographical errors exist, National Instruments reserves the right to make changes to subsequent editions of this document without prior notice to holders of this edition. The reader should consult National Instruments if errors are suspected. In no event shall National Instruments be liable for any damages arising out of or related to this document or the information contained in it.

EXCEPT AS SPECIFIED HEREIN, NATIONAL INSTRUMENTS MAKES NO WARRANTIES, EXPRESS OR IMPLIED, AND SPECIFICALLY DISCLAIMS ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. CUSTOMER'S RIGHT TO RECOVER DAMAGES CAUSED BY FAULT OR NEGLIGENCE ON THE PART OF NATIONAL INSTRUMENTS SHALL BE LIMITED TO THE AMOUNT THERETOFORE PAID BY THE CUSTOMER. NATIONAL INSTRUMENTS WILL NOT BE LIABLE FOR DAMAGES RESULTING FROM LOSS OF DATA, PROFITS, USE OF PRODUCTS, OR INCIDENTAL OR CONSEQUENTIAL DAMAGES, EVEN IF ADVISED OF THE POSSIBILITY THEREOF. This limitation of the liability of National Instruments will apply regardless of the form of action, whether in contract or tort, including negligence. Any action against National Instruments must be brought within one year after the cause of action accrues. National Instruments shall not be liable for any delay in performance due to causes beyond its reasonable control. The warranty provided herein does not cover damages, defects, malfunctions, or service failures caused by owner's failure to follow the National Instruments installation, operation, or maintenance instructions; owner's modification of the product; owner's abuse, misuse, or negligent acts; and power failure or surges, fire, flood, accident, actions of third parties, or other events outside reasonable control.

Copyright

Under the copyright laws, this publication may not be reproduced or transmitted in any form, electronic or mechanical, including photocopying, recording, storing in an information retrieval system, or translating, in whole or in part, without the prior written consent of National Instruments Corporation.

National Instruments respects the intellectual property of others, and we ask our users to do the same. NI software is protected by copyright and other intellectual property laws. Where NI software may be used to reproduce software or other materials belonging to others, you may use NI software only to reproduce materials that you may reproduce in accordance with the terms of any applicable license or other legal restriction.

Trademarks

AutoCode™, MATRIXx™, National Instruments™, NI™, ni.com™, SystemBuild™, and Xmath™ are trademarks of National Instruments Corporation. Refer to the *Terms of Use* section on ni.com/legal for more information about National Instruments trademarks.

Other product and company names mentioned herein are trademarks or trade names of their respective companies.

Members of the National Instruments Alliance Partner Program are business entities independent from National Instruments and have no agency, partnership, or joint-venture relationship with National Instruments.

Patents

For patents covering National Instruments products, refer to the appropriate location: **Help»Patents** in your software, the `patents.txt` file on your CD, or ni.com/patents.

WARNING REGARDING USE OF NATIONAL INSTRUMENTS PRODUCTS

(1) NATIONAL INSTRUMENTS PRODUCTS ARE NOT DESIGNED WITH COMPONENTS AND TESTING FOR A LEVEL OF RELIABILITY SUITABLE FOR USE IN OR IN CONNECTION WITH SURGICAL IMPLANTS OR AS CRITICAL COMPONENTS IN ANY LIFE SUPPORT SYSTEMS WHOSE FAILURE TO PERFORM CAN REASONABLY BE EXPECTED TO CAUSE SIGNIFICANT INJURY TO A HUMAN.

(2) IN ANY APPLICATION, INCLUDING THE ABOVE, RELIABILITY OF OPERATION OF THE SOFTWARE PRODUCTS CAN BE IMPAIRED BY ADVERSE FACTORS, INCLUDING BUT NOT LIMITED TO FLUCTUATIONS IN ELECTRICAL POWER SUPPLY, COMPUTER HARDWARE MALFUNCTIONS, COMPUTER OPERATING SYSTEM SOFTWARE FITNESS, FITNESS OF COMPILERS AND DEVELOPMENT SOFTWARE USED TO DEVELOP AN APPLICATION, INSTALLATION ERRORS, SOFTWARE AND HARDWARE COMPATIBILITY PROBLEMS, MALFUNCTIONS OR FAILURES OF ELECTRONIC MONITORING OR CONTROL DEVICES, TRANSIENT FAILURES OF ELECTRONIC SYSTEMS (HARDWARE AND/OR SOFTWARE), UNANTICIPATED USES OR MISUSES, OR ERRORS ON THE PART OF THE USER OR APPLICATIONS DESIGNER (ADVERSE FACTORS SUCH AS THESE ARE HEREAFTER COLLECTIVELY TERMED "SYSTEM FAILURES"). ANY APPLICATION WHERE A SYSTEM FAILURE WOULD CREATE A RISK OF HARM TO PROPERTY OR PERSONS (INCLUDING THE RISK OF BODILY INJURY AND DEATH) SHOULD NOT BE RELIANT SOLELY UPON ONE FORM OF ELECTRONIC SYSTEM DUE TO THE RISK OF SYSTEM FAILURE. TO AVOID DAMAGE, INJURY, OR DEATH, THE USER OR APPLICATION DESIGNER MUST TAKE REASONABLY PRUDENT STEPS TO PROTECT AGAINST SYSTEM FAILURES, INCLUDING BUT NOT LIMITED TO BACK-UP OR SHUT DOWN MECHANISMS. BECAUSE EACH END-USER SYSTEM IS CUSTOMIZED AND DIFFERS FROM NATIONAL INSTRUMENTS' TESTING PLATFORMS AND BECAUSE A USER OR APPLICATION DESIGNER MAY USE NATIONAL INSTRUMENTS PRODUCTS IN COMBINATION WITH OTHER PRODUCTS IN A MANNER NOT EVALUATED OR CONTEMPLATED BY NATIONAL INSTRUMENTS, THE USER OR APPLICATION DESIGNER IS ULTIMATELY RESPONSIBLE FOR VERIFYING AND VALIDATING THE SUITABILITY OF NATIONAL INSTRUMENTS PRODUCTS WHENEVER NATIONAL INSTRUMENTS PRODUCTS ARE INCORPORATED IN A SYSTEM OR APPLICATION, INCLUDING, WITHOUT LIMITATION, THE APPROPRIATE DESIGN, PROCESS AND SAFETY LEVEL OF SUCH SYSTEM OR APPLICATION.

Conventions

The following conventions are used in this manual:

» The » symbol leads you through nested menu items and dialog box options to a final action. The sequence **File»Page Setup»Options** directs you to pull down the **File** menu, select the **Page Setup** item, and select **Options** from the last dialog box.



This icon denotes a note, which alerts you to important information.

bold Bold text denotes items that you must select or click in the software, such as menu items and dialog box options. Bold text also denotes parameter names.

italic Italic text denotes variables, emphasis, a cross-reference, or an introduction to a key concept. Italic text also denotes text that is a placeholder for a word or value that you must supply.

monospace Text in this font denotes text or characters that you should enter from the keyboard, sections of code, programming examples, and syntax examples. This font is also used for the proper names of disk drives, paths, directories, programs, subprograms, subroutines, device names, functions, operations, variables, filenames, and extensions.

monospace bold Bold text in this font denotes the messages and responses that the computer automatically prints to the screen. This font also emphasizes lines of code that are different from the other examples.

Contents

Chapter 1

Introduction

Manual Organization	1-1
About Neural Networks	1-2
Neural Network Module and SystemBuild.....	1-3

Chapter 2

Neural Network Fundamentals

Single Neuron Architecture	2-1
Linear Neuron Learning Rule.....	2-3
Sigmoid Neuron Learning Rule.....	2-4
Threshold Neuron Learning Rule	2-4
Linear-Threshold Neuron Learning Rule	2-4
Learning Rate, Momentum Factor and Epoch Length	2-5
Neuron Layer	2-5
Feedforward Neural Networks.....	2-5
Output Layer Learning Rule.....	2-6
Hidden Layer Learning Rule	2-7
Supported Models.....	2-8
Radial Basis Function Neural Networks.....	2-9
Kohonen Neural Networks	2-10
InStar Network	2-11
OutStar Network.....	2-11
Competitive Network	2-11
Recurrent Neural Networks	2-12

Chapter 3

Neural Network Interface

Starting and Quitting.....	3-1
Create Neural Network Blocks	3-2
Specify Block Options.....	3-3
Specify Model Options.....	3-4
Specify Learning Options.....	3-5
Specify Parameters and Parameter Variables.....	3-6
Instantiation	3-8
Exporting and Importing Neural Network Models.....	3-8

Offline Training.....	3-10
Training Process.....	3-11
Inspect Training Results.....	3-12
Apply Training Results	3-12
Simulation Considerations.....	3-13
Initmode Setting	3-13
Use of Vars = 2 Keyword Option	3-13

Chapter 4 Neural Network Blocks

Number of Inputs to a Neural Network Block	4-1
Block Reference	4-2
Block Parameters.....	4-4
Neuron.....	4-4
Single-Layer.....	4-6
InStar Layer.....	4-8
OutStar Layer.....	4-10
Competitive Layer.....	4-11
Two-Layer.....	4-13
RBFNN	4-15
Sigmoid Hopfield, Signum Hopfield	4-18

Chapter 5 Neural Network Examples

Example: Linear Neuron	5-1
Create Training Data.....	5-1
Set Up a Linear Neuron Model	5-2
Train the Linear Neuron Model Offline.....	5-2
Instantiate a Neural Network Block.....	5-4
Example: RBFNN	5-4
Create Training Data.....	5-4
Set Up an RBFNN Model	5-5
Train an RBFNN Model	5-5
Example: Signum Hopfield Network	5-6
Problem Specification	5-7
Set Up a Signum Hopfield Network Model.....	5-8
Verification by Using the Offline Trainer.....	5-8

Appendix A
Technical Support and Professional Services

Glossary

Index

Introduction

This guide introduces the Neural Network Module (NNM) for the MATRIXx software family. The NNM is a powerful module with applications in areas such as control system design, pattern recognition, signal processing, and information theory.

The Neural Network Module (NNM) lets you define, parameterize, and include a neural network as a SuperBlock in a SystemBuild block diagram. Adding neural network technology to the fully integrated block diagram language of SystemBuild includes the capability to simulate your neural network models, and to generate embedded code for them through AutoCode.

The NNM supports training (offline) and learning (real-time) modes of operation.

Manual Organization

This guide includes the following chapters:

- Chapter 1, *Introduction*, provides a general introduction to neural networks.
- Chapter 2, *Neural Network Fundamentals*, reviews some essential concepts of neural networks.
- Chapter 3, *Neural Network Interface*, describes the NNM user interface.
- Chapter 4, *Neural Network Blocks*, explains the user interface for neural network blocks.
- Chapter 5, *Neural Network Examples*, provides examples and tutorials.

This guide also has a *Glossary* and an *Index*.

About Neural Networks

A *neural network* is a parallel computing architecture (or a computational model) involving a network topology that can be described in terms of its analogy to a directed, weighted graph. The nodes in the graph represent the computing units or neurons, and the directed edges of the graph represent connections from originating neurons to incident neurons, while weights associated with the edges indicate the strengths of the respective connections. If the directed weighted graph representing the neural network topology has at least one cycle, the neural network is called a recurrent network; otherwise, it is called a feedforward network.

Neural networks are especially useful in practice because they can acquire knowledge through a learning process. The procedure used in the learning process is referred to as the learning algorithm. During a learning process, interneuron connection weights are modified in an orderly fashion so as to obtain a desired performance objective. Therefore, the knowledge acquired by the network is stored as interneuron connection weights.

Neural networks have been successfully used in a number of applications to do the following:

- Learn an internal representation of the mapping between input and output variables
- Learn plant dynamics
- Implement a controller in the closed-loop system
- Adapt to changes in plant characteristics

During the training session, the weights of the neural network are changed according to the learning algorithm. In other words, learning is basically the process of changing the weights such that the input-output performance of the neural network converges to the target or desired performance. If the target or desired output patterns are available during the training session, the learning process is said to be *supervised*. In the absence of target output patterns, the learning process is said to be *unsupervised*, and some other indirect measure of network performance is applied throughout the training session.

The most common neural network architecture is the *feedforward neural network*. Feedforward neural networks consist of several layers of neurons transforming the input signals to the output pattern. They have been widely used in pattern recognition, classification, and clustering applications. Feedforward networks are usually trained using the Error Back Propagation learning algorithm.

Another type of neural network topology is the *recurrent network*, such as the Hopfield Neural Network; recurrent networks are mostly used in optimization. Recurrent networks are dynamic networks with feedback signals from output to input neurons. These networks are commonly used as associative memories in various applications. Training of recurrent networks is usually complicated and mostly done through stochastic algorithms such as Simulated Annealing.

Competitive learning is another important issue in the design of neural networks. The basic difference between competitive learning and other learning algorithms is competition between neurons through inhibitory connections. The neuron that is the winner of the competition will have its weights reinforced, while the weights of all other neurons are left unchanged or decreased. Competitive learning is utilized in networks such as the Kohonen Neural Networks.

A Radial Basis Function Neural Network (RBFNN) is another class of useful Neural Networks, capable of accurate function approximation. Particularly, RBFNNs can be used in smooth interpolation and generalization applications. An RBFNN is a two-layer network. The first layer of neurons, which have a Gaussian output function, roughly partitions the input space into the clusters. Then the second layer, usually linear, approximates the target function with a linear combination of basis (or Gaussian) functions in the first layer. Thus, an RBFNN can fit a smooth curve to the function to be approximated.

Control systems engineering is a successful field of neural network applications. A neural network can be used as an adaptive controller. It also can be used to capture the dynamics of a complex, nonlinear, time-variant plant. Neural Networks can be trained to copy the inverse dynamics of such plants in order to implement adaptive control systems. Other applications of neural networks in the field of adaptive control include Neural Model Reference Adaptive Control Systems (NMRAS) and self-tuning regulators.

Neural Network Module and SystemBuild

The Neural Network Module provides the user with an intuitive graphical user interface (GUI) to perform the following functions:

- Select the desired network from a predefined library of networks.
- Define appropriate parameters for the selected network.

The selected network can then be instantiated in SystemBuild by the press of a button. To instantiate a network, the SystemBuild Editor must be up and running; the instantiated network is placed in the currently active catalog. Within a SystemBuild model, Neural Network blocks can be readily recognized by their special icon graphics.

A typical instantiated SuperBlock contains a BlockScript block connected to one or more Write to Variable blocks if learning is enabled. The BlockScript blocks contain the algorithmic implementation of the neural network model of choice. The Write to Variable blocks are used to store the network weights and biases, making them available to you after a training simulation. The latter is achieved by using the `vars = 2` option in the simulation command, as described in the *SystemBuild User Guide* and the *MATRIXx Help*.

The instantiated neural network SuperBlocks behave exactly like any other SystemBuild SuperBlock. They are open to the user for inspection, alteration, and customization. This provides an open environment for the design of neural networks. For example, the computational algorithms of neural network blocks, as implemented in BlockScript, are open to experienced users for modification, if necessary.

The Neural Network Block Editor lets you specify SuperBlock attributes during the parameter specification phase. These attributes include input, output, and type (for example, standard procedure and inline procedure). The algorithms supported in the NN module are related to discrete operations, and as a result, instantiation of Neural Network SuperBlocks in continuous SuperBlocks is not recommended.

The SystemBuild Neural Network Module graphical user interface (GUI) has four windows:

- Neural Network Block Editor
- Neural Network Parameter Editor
- Neural Network Catalog
- Neural Network Offline Trainer

These windows are described in Chapter 3, *Neural Network Interface*, and in Chapter 4, *Neural Network Blocks*.

Neural Network Fundamentals

This chapter discusses some important concepts about the Neural Network Module, beginning with a single-neuron network and continuing to more complicated networks.

Single Neuron Architecture

The single neuron is the basic element of a Neural Network and can be considered as a small processing unit performing simple calculations. Figure 2-1 shows a single-neuron network.

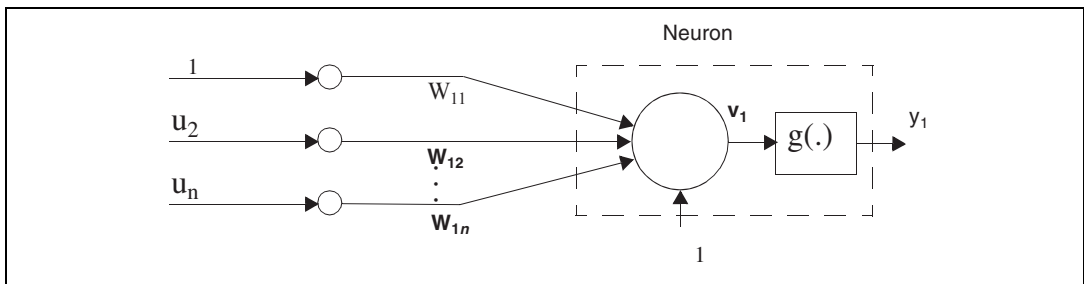


Figure 2-1. Single Neuron

The single neuron has several inputs and one output. The net input or net activation of the neuron can be defined as:

$$v_1 = \sum_{j=1}^n W_{1j}u_j + \theta_1 \quad (2-1)$$

where W_{1j} is the weight associated with the input u_j to the neuron, and θ_1 is the threshold (or bias) of the neuron. Equation 2-1 also can be written as follows:

$$v_1 = \sum_{j=1}^{n+1} W_{1j}u_j \quad (2-2)$$

with

$$\begin{aligned} W_{1(n+1)} &= \theta_1 \\ u_{n+1} &= 1 \end{aligned} \tag{2-3}$$

In other words, the bias can be viewed as the weight to an imaginary input, u_{n+1} , that has one as its fixed value. This kind of notation will be used in all of the learning algorithms that involve biases.

The output of the neuron is a function of its net input or activation. The weights and the threshold can have any real value, including 0.0. The neural output function is defined as:

$$y_1 = g_1(v_1) \tag{2-4}$$

where g_1 is the output function (or the activation function) of the neuron. The supported neural output functions for a single neuron are illustrated in Table 2-1. A single neuron with a threshold activation function is also known as a *Perceptron*.

Table 2-1. Supported Neural Output Functions

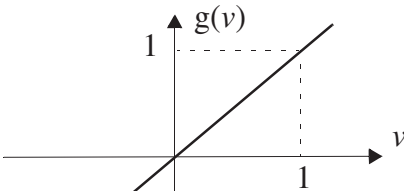
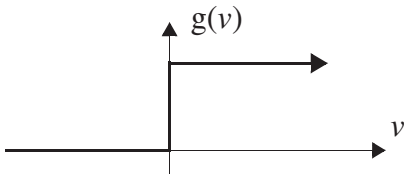
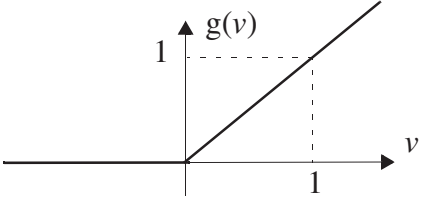
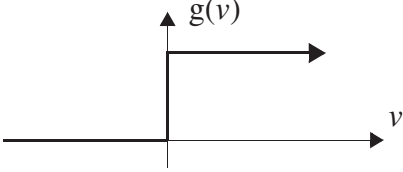
Linear	$g(v) = v$	
Threshold	$g(v) = \begin{cases} 1 & \text{if } v > 0 \\ 0 & \text{if } v \leq 0 \end{cases}$	

Table 2-1. Supported Neural Output Functions (Continued)

Linear Threshold	$g(v) = \begin{cases} v & \text{if } v > 0 \\ 0 & \text{if } v \leq 0 \end{cases}$	
Sigmoid	$g(v) = \frac{1}{1 + e^{-v}}$	

The model of the single neuron given in Table 2-1 has a limited input/output mapping capability. For example, it can only classify linearly separable regions in the space of the input variables.

The SystemBuild NNM supports recursive, supervised learning algorithms for each one of the neural output functions in Table 2-1. These algorithms specify how to update the new values of the weights and the threshold given the old weights and the old threshold, as well as the target or desired output t_1 . Therefore, the network starts with initial weights optionally supplied by the user. The SystemBuild NNM provides the user with the choice to specify the initial weights. The learning algorithms are summarized in the following sections.

Linear Neuron Learning Rule

The learning rule for a single neuron having a linear activation function is as follows:

$$W_{1j}^{(N+1)} = W_{1j}^{(N)} + \eta(1 - \mu)\delta_1 \frac{u_j}{\|\mathbf{u}\|_2} + \mu(W_{1j}^{(N)} - W_{1j}^{(N-1)}) \quad (2-5)$$

where the superscript of W is the index to the sample instance, $\delta_1 = (t_1 - y_1)$ is the difference between the target (or the desired output) and the actual neuron output, η is the learning rate, μ is the momentum factor, and

$$\|\mathbf{u}\|_2^2 = \sum_{j=1}^{n+1} u_j^2 \quad (2-6)$$

with u_{n+1} being an imaginary input having one as its fixed value and the bias as its weight. This learning rule is known as the Normalized Least Mean Square (NLMS) learning rule.

Sigmoid Neuron Learning Rule

The learning rule for a single Sigmoid Neuron is as follows:

$$W_{1j}^{(N+1)} = W_{1j}^{(N)} + \eta(1 - \mu)\delta_1 g_1' u_j + \mu(W_{1j}^{(N)} - W_{1j}^{(N-1)}) \quad (2-7)$$

where $g_1' = y_1(1 - y_1)$ is the derivative of the sigmoid function relative to the net activation. The previous algorithm is known as the Widrow-Hoff or the Least Mean Squares (LMS) learning rule. It is a gradient-based method.

Threshold Neuron Learning Rule

The learning rule applied for Threshold Neurons is known as the Perceptron learning rule:

$$W_{1j}^{(N+1)} = W_{1j}^{(N)} + \eta(1 - \mu)\delta_1 u_j + \mu(W_{1j}^{(N)} - W_{1j}^{(N-1)}) \quad (2-8)$$

The Perceptron learning rule is identical to the Widrow-Hoff learning rule with $g_1' = 1$. However, the Perceptron learning rule is derived from a different concept, because the threshold function and the linear-threshold function are not differentiable at the origin.

Linear-Threshold Neuron Learning Rule

Linear-Threshold Neurons also use the Normalized Least Mean Square (NLMS) method for learning as described in the *Sigmoid Neuron Learning Rule* section.

Learning Rate, Momentum Factor and Epoch Length

All of the previous learning algorithms have been designed to minimize the difference between the output, y_1 , produced by the neural network and the target or desired output, t_1 , by changing the weights W_{1j} in the suitable direction. The convergence rate of these is affected by three parameters—the learning rate, the momentum factor, and the epoch length.

The value of η must be the range of $(0, 2)$ to ensure stability; however, typically we have $0 < \eta < 1$. The momentum factor, satisfying $0 \leq \mu < 1$, has a typical value of 0.9.

The last parameter is the epoch length, which is not shown explicitly in the previous learning algorithms. The epoch length is an integer with its value greater than or equal to one. When it is greater than one, weights are updated in a batch mode. In other words, changes on weights are accumulated for “epoch length” sample periods before they are actually updated. The typical value of the epoch length is either one or the number of data points used for training. Just like momentum, this batch method can be beneficial for certain cases. In general, users need to experiment with those parameters to find out the optimal one for a specific case.

Neuron Layer

After the single-neuron block, the next building block in the SystemBuild NNM is a layer of neurons. The same learning algorithms are applicable for each neuron in the layer. Users can replace the subscript 1 in those learning rules with i , which is the index to neurons.

Feedforward Neural Networks

The simplest type of feedforward neural network consists of layers of neurons, where a neuron in any layer receives input signals only from neurons in the previous layer. Figure 2-2 shows the architecture of a two-layer, feedforward neural network.

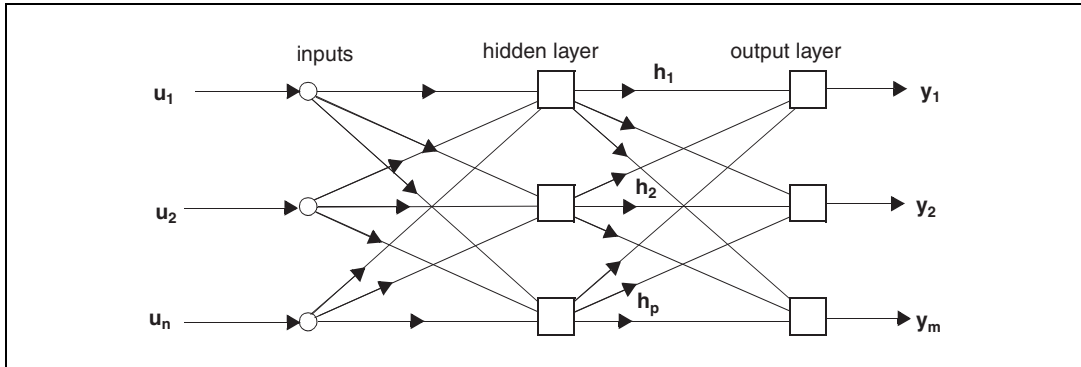


Figure 2-2. Neural Network with One Hidden Layer

The layer that receives external input signals, called the *input layer*, has nodes redistributing the inputs. Therefore, this layer is not counted as a layer of the neural network. The layer that produces the output signals is called the *output layer*. Any other layer in between is called a *hidden layer*. Hidden layers (as well as the output layer) are usually layers of neurons. Adding hidden layers to the network increases the network mapping capabilities.

Although multilayer neural networks have better mapping capabilities than a single layer neural network, training multilayer neural networks is not as fast or as straightforward as single-layer neural networks. For the single neuron case, the target output is given and the error can easily be calculated in order to generate an updated value of the weights. But in the multilayer case, the target outputs are only given for the output layer and one has no direct information regarding the target output of the neurons in the hidden layers. To overcome this problem, the *back-propagation* (BP) algorithm has been used, based on evaluating the error at the output and propagating it back to the previous layers in order to correct the weights of the hidden layers. The details of the BP algorithm for one hidden layer network for two output functions (sigmoid and linear output functions) are described in the following sections.

Output Layer Learning Rule

If the output layer neurons have linear activation functions, the weight updation rule for the output layer will be the NLMS method:

$$W_{ik}^{(N+1)} = W_{ik}^{(N)} + \eta(1 - \mu)(\delta_{2_i}) \frac{h_k}{\|\mathbf{h}\|_2^2} + \mu(W_{ik}^{(N)} - W_{ik}^{(N-1)}) \quad (2-9)$$

where $\|\mathbf{h}\|_2$ is the 2-norm of vector \mathbf{h} .

$$\|\mathbf{h}\|_2 = \left(\sum_{j=1}^p h_j^2 \right)^{1/2} \quad (2-10)$$

Otherwise, if sigmoid activation functions are used, the weight updation rule for the output layer will be the Widrow-Hoff learning rule:

$$W_{ik}^{(N+1)} = W_{ik}^{(N)} + \eta(1 - \mu)(\delta 2_i)(g 2_i')h_k + \mu(W_{ik}^{(N)} - W_{ik}^{(N-1)}) \quad (2-11)$$

In Equations 2-9 through 2-11, W_{ik} is the weight associated with the connection between the i th output neuron and the k th hidden neuron, $\delta 2_i = (t_i - y_i)$ is the modeling error occurred on the i th output neuron, h_k is the output from the i th hidden neuron, and $g 2_i' = y_i(1 - y_i)$ is the derivative of the sigmoid activation relative to the net activation to the i th output neuron. Typical values for the learning rate, η , and the momentum factor, μ , are discussed in the [Output Layer Learning Rule](#) section. The epoch length, which is not explicitly shown, also can be applied to these learning rules.

The output layer errors (each $\delta 2_i$) can be back-propagated to the hidden layer by using the following equation:

$$\delta 1_k = \sum_{i=1}^m (\delta 2_i)g 2_i' W_{ik} \quad k = 1, 2, \dots, p \quad (2-12)$$

where m is the number of output neurons and p is the number of hidden neurons.

Hidden Layer Learning Rule

Just like the output layer, the learning rule used in the hidden layer depends on the type of activation functions you use. If linear activation functions are used for hidden layer neurons, the learning rule for this layer will be the NLMS:

$$W_{kj}^{(N+1)} = W_{kj}^{(N)} + \eta(1 - \mu)(\delta 1_k) \frac{u_j}{\|\mathbf{u}\|_2} + \mu(W_{kj}^{(N)} - W_{kj}^{(N-1)}) \quad (2-13)$$

Or, if the activation functions are sigmoid functions, the learning rule for the hidden layer is the following Widrow-Hoff rule:

$$W1_{kj}^{(N+1)} = W1_{kj}^{(N)} + \eta(1 - \mu)(\delta 1_k)(g1_k')u_k + \mu(W1_{kj}^{(N)} - W1_{kj}^{(N-1)}) \quad (2-14)$$

In Equations 2-13 and 2-14, $W1_{kj}$ is the weight associated with the connection between the k th hidden neuron and the j th input and $g1_k' = h_k(1 - h_k)$ is the derivative of the sigmoid activation function relative to the net activation to the k th hidden neuron.

Again, errors also can be back-propagated to the inputs by using the following equation:

$$\delta 0_j = \sum_{k=1}^p (\delta 1_k)g1_k' W1_{kj} \quad j = 1, 2, \dots, n \quad (2-15)$$

where p is the number of hidden neurons and n is the number of inputs. The errors propagated to the input level is useful only if the network is connected to other neural networks to make a deeper layer neural network.

Supported Models

The NNM tool supports the following two-layer feedforward neural network models:

- Linear and Linear Network
- Sigmoid and Sigmoid Network
- Sigmoid and Linear Network

The word before *and* describes the type of activation functions for the hidden layer, and the word after *and* describes the type of activation functions for the output layer. For example, a Sigmoid plus Linear Network has sigmoid neurons in the hidden layers and linear neurons in the output layer.

Radial Basis Function Neural Networks

A Radial Basis Function Neural Network (RBFNN) is a special kind of feedforward neural network. It has a hidden layer composed of radial basis functions and an output layer composed of neurons with linear activation functions. For an RBFNN with n inputs, each radial basis function (RBF) is the product of n Gaussian functions:

$$\begin{aligned} h_k &= \prod_{j=1}^n \exp\left(\frac{-(u_j - C_{kj})^2}{2(W1_{kj})^2}\right) \\ &= \exp\left(-\sum_{j=1}^n \frac{(u_j - C_{kj})^2}{2(W1_{kj})^2}\right) \end{aligned} \quad (2-16)$$

where C_{kj} and $W1_{kj}$ are, respectively, the center or mean and the standard deviation of a Gaussian function. The output of an RBFNN is the linear combination of hidden layer outputs:

$$y_i = \sum_{k=1}^p W2_{ik} h_k \quad (2-17)$$

where i is a positive integer ($i = 1, 2, \dots, m$) and $W2_{ik}$ is the weight on the connection between the i th output neuron and the k th RBF. Notice that a RBFNN does not contain any bias.

When given a training data set, you can look at distribution of the data set and subjectively determine RBF centers, C_{kj} , and standard deviations, $W1_{kj}$. One possible solution is to make RBF centers uniformly distributed in the input space and choose a fixed value for all of the standard deviations. Alternatively, you can use the provided MathScript functions `kmeans()` and `rbf_radius()` to calculate RBF centers and standard deviations. The `kmeans()` function uses an improved K-means algorithm to place RBF centers so that the density distribution of those RBF centers will match the density distribution of the input part of a training data set. This method can put RBF centers in reasonable locations. However, you must be aware that the density distribution does not necessarily match the complexity distribution. For more information about MathScript functions, refer to the *Xmath User Guide* or to the *MATRIXx Help*.

After RBF centers and standard deviations are determined, the RBFNN use the Normalized Least Mean Square (NLMS) learning rule to adjust weights of RBF outputs:

$$W_{ik}^{(N+1)} = W_{ik}^{(N)} + \eta(\delta 2_i) \frac{h_k}{\|\mathbf{h}\|_2^2} \quad (2-18)$$

where $\delta 2_i = t_i - y_i$ is the modeling error at the i th RBF output.

Kohonen Neural Networks

Kohonen Neural Networks form a unique class of neural networks which are widely used for pattern recognition. The general architecture of the Kohonen network is depicted in Figure 2-3.

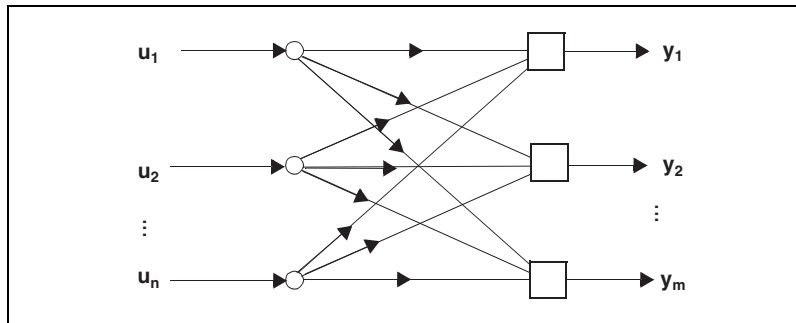


Figure 2-3. Kohonen Neural Network

This section describes linear output functions for the neurons:

$$y_i = \sum_{j=1}^n W_{ij} u_j \quad (2-19)$$

where $i = 1, 2, \dots, m$.

The following sections describe three types of Kohonen Neural Networks: the InStar Layer, the OutStar Layer, and the Competitive Layer. These networks are identical except for their learning algorithms.

InStar Network

The InStar learning algorithm is given by the following equation:

$$W_{ij}^{(N+1)} = W_{ij}^{(N)} + \eta y_i (u_j - W_{ij}^{(N)}) \quad i = 1, 2, \dots, m \quad j = 1, 2, \dots, n \quad (2-20)$$

where η is the learning rate, which the user can specify. It can be seen from Equation 2-20 that the InStar is an unsupervised learning algorithm with no target output. InStar networks are often used to recognize input patterns.

OutStar Network

The OutStar learning algorithm is given by the following equation:

$$W_{ij}^{new} = W_{ij}^{old} + \eta (t_i - W_{ij}^{old}) u_j \quad i = 1, 2, \dots, m \quad j = 1, 2, \dots, n \quad (2-21)$$

where t_i is the i th target output. Therefore, the OutStar learning rule is a supervised learning algorithm; it adjusts the network to recognize output patterns.

Competitive Network

The Kohonen competitive learning rule is based on competition between neurons. The neuron with the highest output is the winner of the competition and is the only neuron where weight is changed. The output of the winning neuron is set to one, and the outputs of other neurons are set to zero. Thus, if k is the index of the winning neuron, the Kohonen competitive learning rule can be expressed as follows:

$$\begin{aligned} W_{kj}^{(N+1)} &= W_{kj}^{(N)} + \eta (u_j - W_{kj}^{(N)}) & j &= 1, 2, \dots, n \\ W_{kj}^{(N+1)} &= W_{kj}^{(N)} & j &= 1, 2, \dots, m; \quad i \neq k \end{aligned} \quad (2-22)$$

At the beginning, the learning rate η is often set to a value close to 0.8. As the learning progresses, η is usually lowered to 0.1 or less for equilibration.

The weights of a neuron correspond to a vector in the R^n input space. Also notice that the weight of a neuron moves toward the input whenever it wins. After the training is complete, you have m weight vectors where distribution resembles that of all input data. Thus, the competitive learning law is essentially the same as the so-called k-means adjustment law.

Recurrent Neural Networks

The feedforward neural network has a forward flow path for the signals from the input to the output with no feedback. This is unlike recurrent networks, where you cannot distinguish separate layers, and each neuron provides an input signal to all other neurons, including itself. In recurrent networks, it also is assumed that the neurons have dynamic characteristics. This feature gives recurrent neural networks extra capabilities. To explain these capabilities, consider the Hopfield Neural Network, which is one of the common recurrent networks, and is implemented in this module.

A Hopfield Neural Network consists of some discrete (or continuous) neurons with the following output function:

$$y_i^{(N+1)} = g(v_i y_i^{(N)}) \quad i = 1, 2, \dots, n \quad (2-23)$$

where n is the number of neurons, $g()$ is the activation function, and v_i is the net activation of the i th neuron:

$$v_i = \sum_{j=1}^n W_{ij} y_j + \theta_i \quad j = 1, 2, \dots, n \quad (2-24)$$

The NNM-supported Hopfield networks include the Sigmoid Hopfield, which has sigmoid activation functions, and the Signum Hopfield, which has a signum-like activation function, as defined by the following equation:

$$y_i^{(N+1)} = \begin{cases} 1 & \text{if } v_i > \theta_i \\ -1 & \text{if } v_i < \theta_i \\ y_i^{(N)} & \text{otherwise} \end{cases} \quad (2-25)$$

Therefore, if v_i is equal to the threshold, θ_i , the current state of a neuron i will be reserved.



Note Equation 2-23 does not take any external inputs; however, the Hopfield network implemented in the NNM tool accepts inputs at the beginning of the simulation as well as the end of each epoch. Inputs read by the Hopfield network replace the current output; therefore, they serve as initial values to the network.

An intensive stability analysis shows that the Hopfield Neural Network, as defined previously, is stable. This means that the neural network potentially has a few equilibrium points that are the final states of the network. Starting from any initial condition, the Neural Network converges to the nearest equilibrium point. This property has been successfully used in content-addressable memory or associative memory. In associative memory, weights of the neural networks are chosen such that the stored data become the equilibrium points of the network. Therefore, the network acts like a content-addressable memory and returns the nearest stored data to the input pattern.

The weights for a Signum Hopfield network can be designed as follows. Let -1 and 1 be the allowed states for each neuron; a Hopfield network with n neurons has 2^n possible states. Let x_i , where $i = 1, 2, \dots, p$, denote the desired stable states; each x_i is an n -vector with each element taking the value -1 or 1 . Then, the following equation defines the weights of the Hopfield network:

$$W = \left(\frac{1}{n}\right) \sum_{i=1}^p \mathbf{x}_i \mathbf{x}_i^T - \left(\frac{p}{n}\right) I \quad (2-26)$$

where I is an identity matrix. With this method, all thresholds are set to zero.

For example, let the stable states for a three-neuron Hopfield network be the following:

$$\begin{aligned} \mathbf{x}_1 &= \{1, -1, 1\}^T \\ \mathbf{x}_2 &= \{-1, 1, -1\}^T \end{aligned} \quad (2-27)$$

Then, the weights of the Hopfield network can be set to the following:

$$\begin{aligned} W &= \frac{1}{3}(\mathbf{x}_1 \mathbf{x}_1^T + \mathbf{x}_2 \mathbf{x}_2^T) - \frac{2}{3}I \\ &= \frac{1}{3} \begin{bmatrix} 0 & -2 & 2 \\ -2 & 0 & -2 \\ 2 & -2 & 0 \end{bmatrix} \end{aligned} \quad (2-28)$$

Any state other than the stable states will converge one of the stable states. Stable states will remain unchanged.

Neural Network Interface

The Neural Network Module Graphical User Interface (GUI) provides a convenient and consistent environment for specification and design of neural networks. The Neural Network Module has four windows:

- Neural Network Block Editor
- Neural Network Parameter Editor
- Neural Network Catalog
- Neural Network Offline Trainer

The Neural Network Block Editor window is the first window that appears when you run the Neural Network Module tool. This chapter describes how to start the tool and how to use interface to complete certain tasks, such as creating Neural Network blocks, editing block parameters, and training.

Starting and Quitting

To start the Neural Network Module tool, type the following command in the **Xmath Commands** window:

```
nntool
```

The **Neural Network Block Editor** window appears as shown in Figure 3-1.

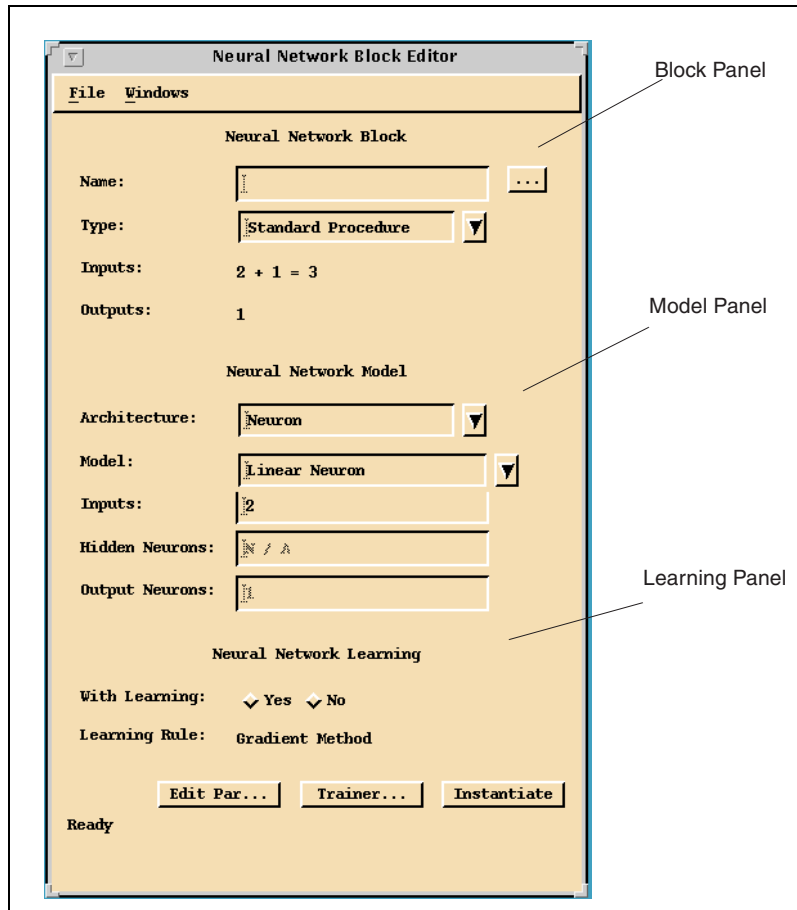


Figure 3-1. Neural Network Block Editor Window

This window has three panels—the **Block Panel**, the **Model Panel**, and the **Learning Panel**. To exit the NNM tool, select **File»Quit**.

Create Neural Network Blocks

The process of creating a neural network block includes five steps:

1. Specify block options.
2. Specify model options.
3. Specify learning options.

4. Specify parameters and parameter variables.
5. Instantiate.

These steps are explained in detail in the following sections.

Specify Block Options

All fields regarding block options are gathered in the **Block Panel**, which is located on the top panel of the block editor as shown in Figure 3-1. The only required field in this panel is **Name**. Notice that a neural network block is actually a Procedure SuperBlock. The Neural Network Module supports the following SuperBlock types:

- Standard Procedure (default)
- Inline Procedure

For a description of Procedure SuperBlocks, refer to the *SystemBuild User Guide*.

The number of inputs and the number of outputs of the neural network block are displayed in the **Block Panel** to the right of the **Inputs** and **Outputs** fields. The number of outputs is always identical to the number of output neurons. However, the number of inputs to a neural network block depends on the following factors:

- The number of inputs to the neural network model
- The learning option
- The type of learning rule (that is, supervised or unsupervised)

Due to the last two factors, a neural network block (SuperBlock) may have one of the following components:

- A neural network model
- A neural network model with an unsupervised learning algorithm
- A neural network model with a supervised learning algorithm

For the first two cases, the neural network block has the same inputs and outputs as the neural network model. For the third case, the inputs of the neural network block are the inputs to the neural network model plus target data items, which are of the same size as the model outputs.

Specify Model Options

Information regarding the neural network model is specified in the **Model Panel**—the middle panel of the Neural Network Block Editor as shown in Figure 3-1. Fields in this panel are as follows:

- **Architecture**—The neural network architecture. Available choices are as follows:
 - Neuron
 - One-Layer Network
 - Two-Layer Network
 - Recurrent Network



Note Inputs to a neural network are not counted as a layer.

- **Model**—This field lists available neural network models for a given neural network architecture. When the architecture is Neuron, for example, the following models are available:
 - Linear Neuron
 - Threshold Neuron
 - Linear-Threshold Neuron
 - Sigmoid Neuron

where the first word in each of these models describes the activation function (or output function) built into the model.



Note A threshold neuron is also known as a Perceptron.

When the architecture type is One-Layer Network, the following models are available:

- Linear Layer
- Threshold Layer
- Linear-Threshold Layer
 - Sigmoid Layer
 - InStar Layer
 - OutStar Layer
 - Competitive Layer

When the architecture type is Two-Layer Network, the possible choices are as follows:

- Linear and Linear Network
- Sigmoid and Sigmoid Network
- Sigmoid and Linear Network
- RBFNN

When the architecture type is Recurrent Network, the possible choices are as follows:

- Sigmoid Hopfield
- Signum Hopfield
- **Inputs**—Number of inputs to the neural network model.
- **Hidden Neurons**—Number of hidden layer neurons. This field displays N/A (for Not Applicable) if the neural network model does not have a hidden layer.
- **Output Neurons**—Number of output layer neurons. When the architecture is Neuron, this field is fixed to one. When the architecture is Recurrent Network, this field has the same value as the **Inputs** field; whenever the value for Inputs or Outputs is changed, the other changes accordingly.

Specify Learning Options

The learning option of a neural network block is set from the **With Learning** checkbox in the **Learning Panel** shown in Figure 3-1. Notice the following:

- Some neural network blocks, such as the Sigmoid Hopfield and Signum Hopfield blocks, do not have a learning algorithm.
- All other neural network blocks are equipped with one learning algorithm, which is displayed in the **Learning Rule** field.
- When the neural network model comes with a supervised learning algorithm and **With Learning** is set to **Yes**, target data is required.
- With the learning option set to **Yes**, weights of a neural network model change as the simulation goes on. Furthermore, only the final weights are available as Xmath variables at the end of the simulation.

Specify Parameters and Parameter Variables

A neural network model and its learning algorithm are implemented as a BlockScript block. Some of the block parameters of the BlockScript block are used internally as static variables. Others, such as network weights and learning options, are accessible by users. You can use the **Neural Network Parameter Editor** shown in Figure 3-2 to specify default values and parameter variables for these block parameters.

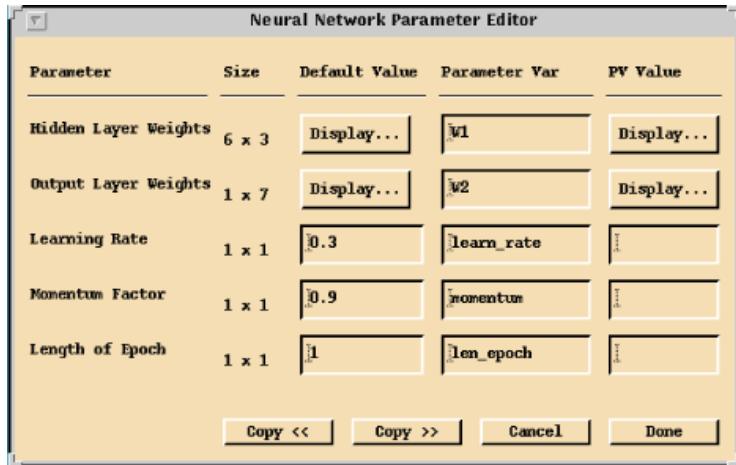


Figure 3-2. Neural Network Parameter Editor Window

To run the parameter editor, do one the following:

- Click **Edit Par** in either the **Neural Network Block Editor** window, as shown in Figure 3-1, or the **Neural Network Offline Trainer** window, as shown in Figure 3-4.
- Select **Windows»Parameter Editor** from the **Neural Network Block Editor** window, as shown in Figure 3-1.

The **Neural Network Parameter Editor** window, as shown in Figure 3-2, has five columns:

- **Parameter**—The label of a block parameter
- **Size**—The size (rows \times columns) of the block parameter. The size for a scalar parameter is displayed as (1 \times 1).
- **Default Value**—Displays the default value of a parameter, if it is a scalar; otherwise, it provides a button that, when clicked, displays the values of a parameter in the **Xmath Log** window.

- **Parameter Var**—The name of the parameter variable.
- **PV Value**—For a scalar parameter, this field displays the value of the parameter variable if the parameter variable is defined in the **Xmath Commands** window; otherwise, this field is left blank. For a matrix or vector parameter, a **Display** button appears in this field. When you click the button, the value of the parameter displays in the **Xmath Commands** window; otherwise, a message reports that the parameter variable is not defined.

The **Neural Network Parameter Editor** window also has four buttons:

- **Copy <<**—Replaces the default value of each parameter with the value of the corresponding parameter variable, if the parameter variable is defined in the Xmath workstation and its size matches that of the parameter.
- **Copy >>**—Copies the value of each parameter into the corresponding parameter variable. If the parameter variable does not exist in the **Xmath Commands** window, a new parameter is created.
- **Cancel**—Closes the parameter editor and cancels all changes. Notice that changes to the values of parameter variables are not recoverable.
- **Done**—Closes the parameter editor and accepts the changes.

When using the **Neural Network Parameter Editor**, notice the following:

- Default values for neural network weights are set at random.
- If a parameter variable is defined in the Xmath workspace, its value is used in the simulation and the default parameter value is ignored.
- It is the user's responsibility to ensure that parameter variables are not shared unintentionally by multiple neural network blocks.
- In addition to using the **Copy <<** button, you can replace the default parameter values by using the following Xmath script command:

```
updatepv NN_BLOCK_NAME
```

where `NN_BLOCK_NAME` is the name of a neural network block. Notice that `updatepv` also can be used to update block parameters in other blocks. To get more information about `updatepv`, type `help updatepv` in the **Xmath Command** window.

Instantiation

To instantiate a neural network block, complete the following steps:

1. Create a discrete-time SuperBlock or select a discrete-time SuperBlock from the catalog of the SuperBlock Editor.
2. Click **Instantiate** in the **Neural Network Block Editor** window.

After you click **Instantiate**, the Neural Network Module tool does the following:

- Creates a neural network Procedure SuperBlock with the name specified in the **Name** field
- If an active SuperBlock exists, creates a neural network block (SuperBlock) that points to the above SuperBlock

For simulation considerations, refer to the [Simulation Considerations](#) section.

Exporting and Importing Neural Network Models

You can export a neural network model in the following ways:

- Instantiate it as a neural network block in the SuperBlock Editor by clicking **Instantiate** in the **Neural Network Block Editor** window.
- Copy it to the Xmath workspace by selecting **File»Copy NN Model to Xmath**.
- Save it as an ASCII file by selecting **File»Save NN Model to File**.

Similarly, you can retrieve a neural network model in the following corresponding ways:

- Read it from the SuperBlock Editor by clicking **...** (the browse button) in the **Block Editor** window. A **Neural Network Catalog** window, as shown in Figure 3-3, appears and lists all neural network blocks currently in the SystemBuild catalog. Click the neural network block that you want to select, and then click **Done**.

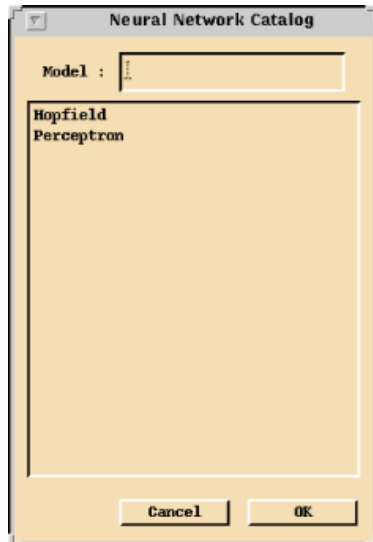


Figure 3-3. Neural Network Catalog Window

- Copy the selected block from the **Xmath Commands** window by selecting **File»Copy NN Model** from Xmath.
- Load the model to a file by selecting **File»Load NN Model to File**.

Offline Training

A neural network model can be trained by the **Neural Network Offline Trainer** as shown in Figure 3-4.

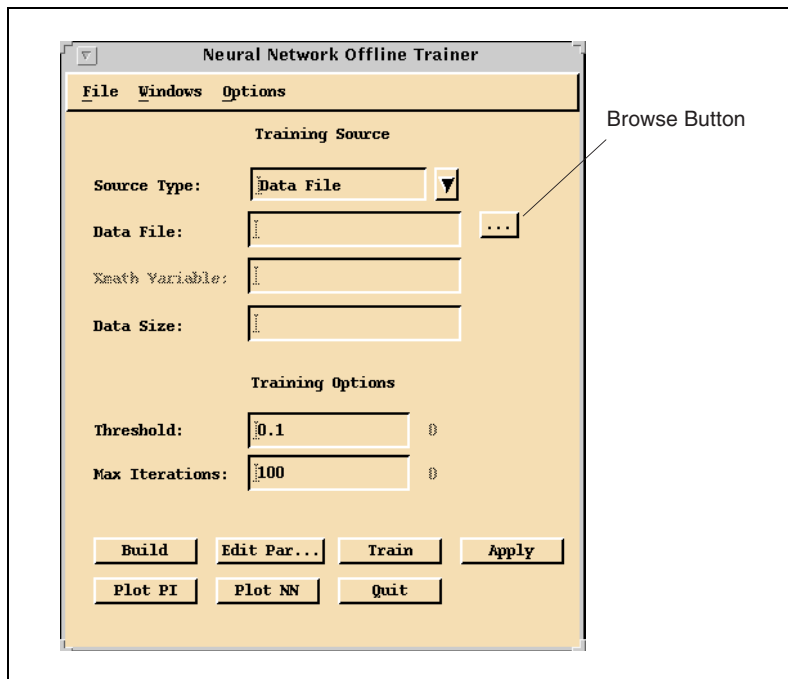


Figure 3-4. Neural Network Offline Trainer Window

The trainer is an integral part of the Neural Network Module tool. Before you can start the training process, you must provide training data, which can be saved either in a data file or an Xmath variable.

The number of columns in the training data depends on the type of learning rule. Let n denote the number of model inputs and m denote the number of model outputs.

- If the neural network model has a supervised learning algorithm, the training data must have $n + m$ columns; the first n columns hold input data and the last m columns hold target data.
- If the neural network model has an unsupervised learning algorithm, such as the competitive learning rule and the InStar learning rule, the number of rows of the training data must be n . In other words, no target data is required.

Training Process

After the training data is prepared, you can complete the offline training process by completing the following steps:

1. Set up or retrieve a Neural Network model by using the Neural Network Block Editor.
2. Click **Trainer** in the **Block Editor** window to invoke the Neural Network Offline Trainer. The neural network model must have a name before the Trainer can be opened.
3. Specify the type of training data by clicking either **Data File** or **Xmath Variable** in the **Source Type** field.
4. If you clicked the **Source Type** of **Data File**, type the name of the training data file in the **Data File** field. As an alternative, you can click **...** (the browse button) on the right end of the Data File, and a **File Selection** window appears.
5. If you clicked the **Source Type** of **Xmath Variable**, type the name of the Xmath matrix variable that contains the training data. The variable must reside in the `main.` partition.
6. If the learning rule is supervised, type the value of the desired threshold in the **Threshold** field. The performance index (PI), which is defined as the root-mean-squares of approximation errors, is evaluated against the threshold.
7. Type the maximum number of iterations in the **Max Iterations** field. Available training data is used once per iteration.
8. Examine training parameters, such as Learning Rate and Length of Epoch, through the parameter editor. The parameter editor can be invoked by clicking **Edit Par.**
9. Click **Build** to build a Training SuperBlock. The Training SuperBlock could have the following components:
 - **WaveForm Block**—Generates training data as specified in the training source, either a file or an Xmath matrix.
 - **Neural Network Block**—An underscore is added to the name of the neural network block so that any change to this block will not affect the original one directly.
 - **Terminator Block**—A BlockScript block that evaluates the performance of approximation.
 - **Stop Block**—When this block is triggered by the Terminator block, the training process stops.



Note The Terminator Block and the Stop Block appear only when the learning rule is supervised.

10. Click **Train** to start the training. The training stops when the number of iterations has reached the specified maximum number or, if the learning rule is supervised, when the performance index is less than the threshold.



Note If the neural network model is changed, or if training data items are changed, rebuild the Training SuperBlock by clicking **Build** again. If applicable, you also may want to change the parameter value or the parameter variable value of **Length of Epoch** to the length of the training data or a fraction of it, so performance evaluation is more consistent. For example, if the length (or rows) of the training data is 10, consider setting **Length of Epoch** to 10 or 5, but not 7. Be aware that, for most neural network models with supervised learning, neural network weights are updated at the end of each epoch while the performance evaluation is made for each iteration.

Inspect Training Results

Users can inspect training results as follows:

- Click **Plot PI** and the graph of performance index versus number of iterations appears.
- Click **Plot NN** to plot the final output from the trained Neural Network model, in solid lines, and the training data, in markers.
- Final weights are saved as Xmath variables. For example, if the parameter variable for hidden layer weights is W1, then the final values for the hidden layer are saved as w1_. You can enter the variable name in the **Xmath Commands** window to check the value.

Apply Training Results

You can apply satisfactory training results by completing the following steps:

1. Click **Apply**.
2. Click **Quit**.
3. Click **Instantiate** in the **Block Editor** window.

Otherwise, click the **Apply** button if you want to keep the current status. When you click **Apply**, values of final weights are copied into parameter variables; therefore, the final weights can be used for further training. You can do the following to improve training results:

- Set a different learning rate.
- Set a different epoch length.
- Increase the maximum number of iterations.
- Provide different training data.
- Change the structure of the neural network model. For example, add more neurons to the hidden layer.
- Try a different neural network model. For example, replace a Sigmoid Neuron with a Linear Neuron or replace a Sigmoid plus Linear Network with an RBFNN.

Simulation Considerations

The following sections describe the initialization mode for AutoCode simulation and how to capture revised weights from a training simulation.

Initmode Setting

For consistency with AutoCode, the `initmode` keyword in SystemBuild must be set equal to 0 whenever you are running with neural network blocks and AutoCode generated code. Setting `actiming = 1` has the effect of setting `initmode = 0`. If `initmode = 0`, signal transients at startup time are not suppressed. The default value for `initmode` is 3, which invokes a series of block executions before startup, automatically cleaning up startup transients.

Use of Vars = 2 Keyword Option

If the learning option is turned on, at the end of training the acquired knowledge of a neural network is factored into the updated weights. Thus, you need access to these weights. To provide such access, simulate your designs with the `vars = 2` keyword option.

During the simulation the updated weights are continually written into variables with original names plus an appended underscore character; for example, updated weights are stored in the `myweight_` file as they are derived from the `myweight` file.

The `vars` keyword allows the final value of these updated weights to be transferred to Xmath. The default partition for these weights is the main partition unless a partition is specified when the original parameter is named. For more details on the `vars = 2` keyword option, refer to the *SystemBuild User Guide*.

Neural Network Blocks

The Neural Network Module is implemented as a set of SuperBlock blocks under SystemBuild. Each block implements a neural network function, most of which correspond to the algorithms explained in Chapter 2, *Neural Network Fundamentals*. Most of the blocks also have the option to include a learning algorithm. The available blocks are as follows:

- **Neuron**—Including Linear Neuron, Threshold Neuron, Linear-Threshold Neuron, and Sigmoid Neuron
- **One-Layer Network**—Including Linear Layer, Threshold Layer, Linear-Threshold Layer, Sigmoid Layer, InStar Layer, OutStar Layer, and Competitive Layer
- **Two-Layer Network**—Including Linear and Linear Network, Sigmoid and Sigmoid Network, Sigmoid and Linear Network, and RBF and Linear Network
- **Recurrent Network**—Including Sigmoid Hopfield and Signum Hopfield

Neural Network Module blocks are implemented using the National Instruments BlockScript language. For a complete description of the BlockScript language, refer to the *BlockScript User Guide*. For documentation on SuperBlocks, refer to the *SystemBuild User Guide* or to the *MATRIXx Help*.

Number of Inputs to a Neural Network Block

The number of inputs to a neural network block depends on whether the block is instantiated with a supervised learning algorithm or not. Without a supervised learning algorithm, the inputs to a neural network block are the same as the inputs of the neural network model. Otherwise, supervised learning requires target inputs as well as data inputs, where the total number of inputs to a network is calculated by adding the number of inputs for the input layer to the number of target outputs. Users only need to specify the number of inputs for the input layer. The Neural Network Module uses this number and the number of outputs to deduce the number of target inputs.

As a result, the neural network SuperBlocks instantiated with supervised learning would automatically have a total number of inputs equal to the number of input layer inputs plus the number of target inputs. The inputs to the SuperBlock are arranged so that the neural network layer inputs come first. For an illustration of Input Layer Inputs and Target Inputs, refer to Figure 4-1. Notice that inputs to the single neuron block have three neural network inputs and one target input.

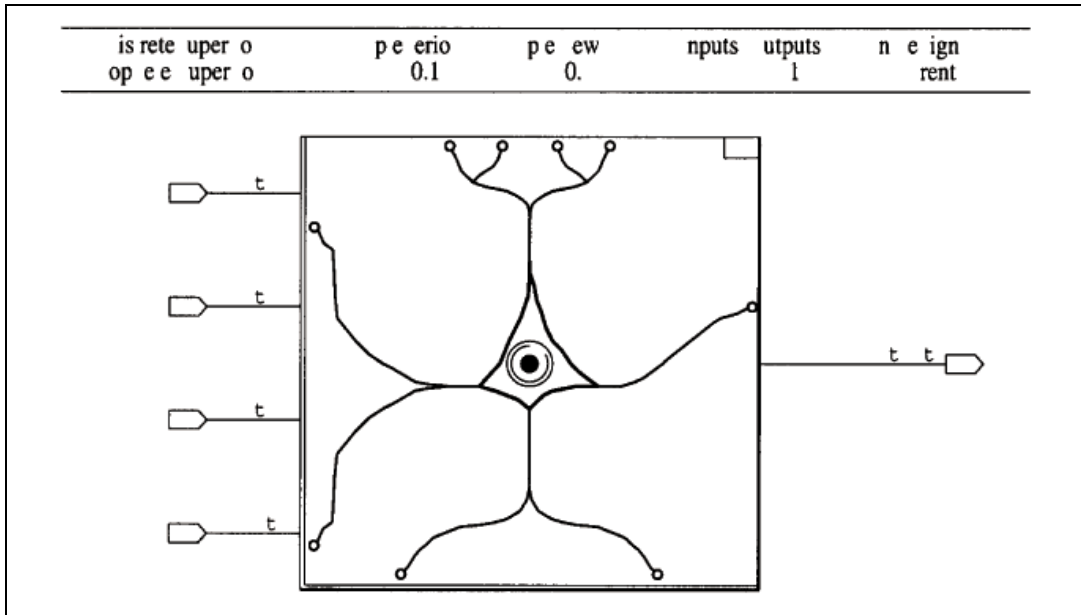


Figure 4-1. Single Neuron Block with a Supervised Learning Algorithm

Block Reference

For a summary of the available block calls by category, refer to Table 4-1.

Table 4-1. Summary of Available Blocks

Block Name	Description	Section
Neuron		
Linear Neuron	A single neuron with a linear activation function.	<i>Neuron</i>
Threshold Neuron	A single neuron with a threshold activation function.	<i>Neuron</i>

Table 4-1. Summary of Available Blocks (Continued)

Block Name	Description	Section
Linear-Threshold Neuron	A single neuron with a linear-threshold activation function.	<i>Neuron</i>
Sigmoid Neuron	A single neuron with a sigmoid activation function.	<i>Neuron</i>
One-Layer Network		
Linear Layer	A one-layer neural network with a linear activation function.	<i>Single-Layer</i>
Threshold Layer	A one-layer neural network with a threshold activation function.	<i>Single-Layer</i>
Linear-Threshold Layer	A one-layer neural network with a linear-threshold activation function.	<i>Single-Layer</i>
Sigmoid Layer	A one-layer neural network with a sigmoid activation function.	<i>Single-Layer</i>
InStar Layer	Trains a layer of linear neurons using the InStar learning rule.	<i>InStar Layer</i>
OutStar Layer	Trains a layer of linear neurons using the OutStar learning rule.	<i>OutStar Layer</i>
Competitive Layer	Defines a layer of neurons with a competitive output layer. This neural network block can be embedded with the competitive learning algorithm if it is instantiated with learning.	<i>Competitive Layer</i>
Two-Layer Network		
Linear and Linear Network	A two-layer neural network with a linear activation function in both layers.	<i>Two-Layer</i>
Sigmoid and Sigmoid Network	A two-layer neural network with a sigmoid activation function in both layers.	<i>Two-Layer</i>
Sigmoid and Linear Network	A two-layer neural network with a sigmoid activation function in the hidden layer and a linear activation function in the output layer.	<i>Two-Layer</i>

Table 4-1. Summary of Available Blocks (Continued)

Block Name	Description	Section
RBFNN	This block defines a radial basis function neural network (RBFNN) with linear activation functions. A learning algorithm is embedded if it is instantiated with learning.	<i>RBFNN</i>
Recurrent Network		
Sigmoid Hopfield	Simulates a Hopfield neural network with sigmoid activation functions.	<i>Sigmoid Hopfield, Signum Hopfield</i>
Signum Hopfield	Simulates a Hopfield neural network with signum activation functions.	<i>Sigmoid Hopfield, Signum Hopfield</i>

Block Parameters

Each neural network block contains a BlockScript block. The parameters of the BlockScript block are regarded as the parameters of the neural network block. This chapter describes only those parameters that you can change. Other unlisted block parameters are used as static variables and their values cannot be changed by the user.

The name of a neural network block parameter may be different from that shown in the **Parameter Editor** window. For example, the block parameters `EpochLength` and `LearnRate` are shown as **Length of Epoch** and **Learn Rate**, respectively. However, with the explanation given under each block parameter, users can identify the corresponding fields in the **Parameter Editor** window.

Neuron

The neuron block includes the Linear Neuron, the Threshold Neuron, the Linear-Threshold Neuron, and the Sigmoid Neuron algorithms.

Purpose This block defines a single neuron. A learning algorithm is embedded in the neuron if the neuron is instantiated with learning.

Input u : (vector)

The input vector to the single neuron is as follows:

$$u = [u_1 \ u_2 \ \dots \ u_n] \quad (4-1)$$

where u_j is the j th input. Neuron inputs take the first n input pins of the neural network block.

t : (scalar)

The target t takes the last input pin of the single neuron block. This parameter appears only if the neuron is instantiated with learning.

Output y : (scalar)

The neuron output.

Parameters W : (vector)

Weights and a threshold (or bias) are as follows:

$$W = [W_{11} \ W_{12} \ \dots \ W_{1n} \ \theta_1] \quad (4-2)$$

where W_{1j} is the weight of the connection between the neuron and the j th input, n is the number of neuron inputs, and θ_1 is the threshold of the neuron. If instantiated with a learning algorithm, the values of W change at the end of each epoch.

LearnRate: (scalar)

The learning rate. This parameter is available only if the neural network block is instantiated with learning. Its typical value is between 0 and 1.

Momentum: (scalar)

The momentum factor. This parameter is available only if the neural network block is instantiated with learning. Its typical value is 0.9.

EpochLength: (scalar)

The length of epoch. This parameter is available only if the neural network block is instantiated with learning. The weights and the threshold are updated at the end of each epoch. `EpochLength` is usually chosen to be the length of the training data set.

Remarks

- The activation function type is indicated by the first word in the name of the single neuron block. For example, Sigmoid Neuron is a neuron with the sigmoid activation function.
- If learning is enabled, the Widrow-Hoff learning algorithm (or the Least Mean Squares method) is applied to Linear Neurons and Sigmoid Neurons; the Perceptron learning rule is applied to Threshold Neurons and Linear-Threshold Neurons.
- If learning is enabled and `vars = 2` is used as a `sim` keyword, the final values of W are written into the **Xmath Commands** window after simulation.

Single-Layer

The layer block includes the Linear Layer, the Threshold Layer, the Linear-Threshold Layer, and the Sigmoid Layer single-layer neural networks.

Purpose This block defines a one-layer neural network. A learning algorithm is embedded in the neural network if it is instantiated with learning.

Input u : (vector)

The inputs to the one-layer neural network are as follows:

$$u = [u_1 \ u_2 \ \dots \ u_n] \quad (4-3)$$

where u_j is the j th input. Network inputs take the first n input pins of the neural network block.

t : (vector)

The target is expressed as follows:

$$t = [t_1 \ t_2 \ \dots \ t_m] \quad (4-4)$$

where t_i is the i th target and m is the number of neurons (or outputs) of the one-layer neural network. Target data takes the last m input pins of the neural network block. Target inputs are required only if the neural network block is instantiated with learning.

Output

y : (vector)

The neuron outputs the following:

$$y = [y_1 \ y_2 \ \dots \ y_m] \quad (4-5)$$

where y_i is the output of the i th output neuron and m is the number of neurons ($i = 1, 2, \dots, m$).

Parameters

W : (matrix)

Weights and corresponding thresholds (or biases) are as follows:

$$W = \begin{bmatrix} W_{11} & W_{12} & \dots & W_{1n} & \theta_1 \\ W_{21} & W_{22} & \dots & W_{2n} & \theta_2 \\ \dots & \dots & \dots & \dots & \dots \\ W_{m1} & W_{m2} & \dots & W_{mn} & \theta_m \end{bmatrix} \quad (4-6)$$

where W_{ij} is the weight of the connection between the i th neuron and the j th input, and θ_j is the threshold of the i th neuron. If instantiated with a learning algorithm, the values of W change at the end of each epoch.

LearnRate: (scalar)

The learning rate. This parameter is available only if the neural network block is instantiated with learning. Its value is typically between 0 and 1.

Momentum: (scalar)

The momentum factor. This parameter is available only if the neural network block is instantiated with learning. Its typical value is 0.9.

EpochLength: (scalar)

The length of epoch. This parameter is available only if the neural network block is instantiated with learning. The weights and the threshold are updated at the end of each epoch. `EpochLength` is usually chosen to be the length of the training data set.

Remarks

- The activation function type is indicated by the first word in the name of the neural network block. For example, a Sigmoid Layer block is a layer of neurons with sigmoid activation functions.
- If learning is enabled, the Widrow-Hoff learning algorithm (or the Least Mean Squares method) is applied to Linear Layer and Sigmoid Layer; the Perceptron learning rule is applied to Threshold Layer and Linear-Threshold Layer.
- If learning is enabled and `vars = 2` is used as a `sim` keyword, the final values of W are written into the **Xmath Commands** window after simulation.

InStar Layer

Purpose This block defines a layer of neurons. This neural network block can be embedded with the InStar learning algorithm if it is instantiated with learning. With the learning algorithm, each neuron can learn an input pattern.

Input u : (vector)

The inputs to the neural network are as follows:

$$u = [u_1 u_2 \dots u_n] \quad (4-7)$$

Output y : (vector)

The neural network outputs are as follows:

$$y = [y_1 y_2 \dots y_m] \quad (4-8)$$

where y_i is the output of the i th output neuron and m is the number of neurons ($i = 1, 2, \dots, m$).

Parameters W : (matrix)

Weights are as follows:

$$W = \begin{bmatrix} W_{11} & W_{12} & \dots & W_{1n} \\ W_{21} & W_{22} & \dots & W_{2n} \\ \dots & \dots & \dots & \dots \\ W_{m1} & W_{m2} & \dots & W_{mn} \end{bmatrix} \quad (4-9)$$

where W_{ij} is the weight of the connection between the i th neuron and the j th input.

LearnRate: (scalar)

The learning rate. This parameter is available only if the neural network block is instantiated with learning. Its value is typically between 0 and 1.

Remarks

- The activation function of an InStar Layer is linear and no bias is applied to each neuron:

$$y_i = \sum_{j=1}^n W_{ij} u_j, \quad i = 1, 2, \dots, m \quad (4-10)$$

- The InStar learning algorithm is an unsupervised. Therefore, no target data is required.
- If learning is enabled and `vars = 2` is used as a `sim` keyword, the final values of W are written into the **Xmath Commands** window after simulation.
- The InStar learning algorithm is shown in Equation 2-22.

OutStar Layer

Purpose This block defines a layer of neurons. This neural network block may be embedded with the OutStar learning algorithm if it is instantiated with learning. With the learning algorithm, the OutStar Layer is able to learn output patterns.

Input u : (vector)

The inputs to the neural network have the following form:

$$u = [u_1 u_2 \dots u_n] \quad (4-11)$$

where u_j is the j th input and n is the number of inputs. Neural network inputs take the first n input pins of the neural network block.

t : (vector)

The target is defined as follows:

$$t = [t_1 t_2 \dots t_m] \quad (4-12)$$

where t_i is the i th target and m is the number of neurons (or outputs) of the one-layer neural network. Target data takes the last m input pins of the neural network block. Target inputs are required only if the neural network block is instantiated with learning.

Output y : (vector)

The output vector has the following form:

$$y = [y_1 y_2 \dots y_m] \quad (4-13)$$

where y_i is the output of the i th output neuron and m is the number of neurons ($i = 1, 2, \dots, m$).

Parameters W : (matrix)

Weights of neurons are as follows:

$$W = \begin{bmatrix} W_{11} & W_{12} & \cdots & W_{1n} \\ W_{21} & W_{22} & \cdots & W_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ W_{m1} & W_{m2} & \cdots & W_{mn} \end{bmatrix} \quad (4-14)$$

where W_{ij} is the weight of the connection between the i th neuron and the j th input.

LearnRate: (scalar)

The learning rate. This parameter is available only if the neural network block is instantiated with learning. Its value is typically between 0 and 1.

Remarks

- The activation function of an OutStar Layer is linear and no bias is applied to each neuron:

$$y_i = \sum_{j=1}^n W_{ij}u_j, \quad (i = 1, 2, \dots, m) \quad (4-15)$$

- If learning is enabled and `vars = 2` is used as a `sim` option for `sim`, the final values of W are written into the **Xmath Commands** window after simulation.
- The OutStar learning rule is shown as Equation 2-21.

Competitive Layer

Purpose This block defines a layer of neurons with a competitive output layer. This neural network block may be embedded with the competitive learning algorithm if it is instantiated with learning. With the learning algorithm, a Competitive Layer works like a K-means method. At the end of learning, the density distribution of the weight vectors will be similar to that of input data.

Input u : (vector)

The inputs to the neural network have the following form:

$$u = [u_1 u_2 \dots u_n] \quad (4-16)$$

where y_i is the output of the i th output neuron and m is the number of neurons ($i = 1, 2, \dots, m$).

Output y : (vector)

The output vector has the following form:

$$y = [y_1 y_2 \dots y_m] \quad (4-17)$$

where y_i is the output of the i th output neuron and m is the number of neurons ($i = 1, 2, \dots, m$).

here y_j is the j th output and m is the number of outputs.

Parameters W : (matrix)

Weights of neurons are as follows:

$$W = \begin{bmatrix} W_{11} & W_{12} & \dots & W_{1n} \\ W_{21} & W_{22} & \dots & W_{2n} \\ \dots & \dots & \dots & \dots \\ W_{m1} & W_{m2} & \dots & W_{mn} \end{bmatrix} \quad (4-18)$$

where W_{ij} is the weight of the connection between the i th neuron and the j th input.

LearnRate: (scalar)

The learning rate. This parameter is available only if the neural network block is instantiated with learning. Its value is typically between 0 and 1.

Remarks

- The competitive learning algorithm is an unsupervised algorithm. Therefore, no target data is required.
- Given the weight vector, $\{W_{i1}, W_{i2}, \dots, W_{in}\}$, the neuron has the shortest distance to the input is the winner.
- The output of the winner is 1; the output of any other neuron is 0.
- If learning is enabled, only the weights of the winner will be tuned.
- If learning is enabled and `vars = 2` is used as a keyword option for `sim`, the final values of W are written into the **Xmath Commands** window after simulation.

Two-Layer

The two-layer block includes the Linear and Linear Network, Sigmoid and Sigmoid Network, and Sigmoid and Linear Network algorithms.

Purpose This block defines a two-layer neural network. A learning algorithm is embedded in the neural network if it is instantiated with learning.

Input u : (vector)

The inputs to the two-layer neural network are as follows:

$$u = [u_1 \ u_2 \ \dots \ u_n] \quad (4-19)$$

where u_j is the j th input. Network inputs take the first n input pins of the neural network block.

t : (vector)

The target is expressed as follows:

$$t = [t_1 \ t_2 \ \dots \ t_m] \quad (4-20)$$

where t_i is the i th target and m is the number of output neurons. Target data takes the last m input pins of the neural network block. Target inputs are required only if the neural network block is instantiated with learning.

Output y : (vector)

The neural network outputs the following:

$$y = [y_1 \ y_2 \ \dots \ y_m] \quad (4-21)$$

where y_i is the output of the i th output neuron and m is the number of neurons ($i = 1, 2, \dots, m$).

Parameters $W1$: (matrix)

The hidden layer weights and thresholds (or biases) are as follows:

$$W1 = \begin{bmatrix} W1_{11} & W1_{12} & \dots & W1_{1n} & \theta1_1 \\ W1_{21} & W1_{22} & \dots & W1_{2n} & \theta1_2 \\ \dots & \dots & \dots & \dots & \dots \\ W1_{p1} & W1_{p2} & \dots & W1_{pn} & \theta1_p \end{bmatrix} \quad (4-22)$$

where $W1_{kj}$ is the weight of the connection between the k th neuron and the j th input, and $\theta1_k$ is the threshold of the k th hidden layer neuron.

$W2$: (matrix)

The hidden layer weights and thresholds (or biases) are as follows:

$$W2 = \begin{bmatrix} W2_{11} & W2_{12} & \dots & W2_{1p} & \theta2_1 \\ W2_{21} & W2_{22} & \dots & W2_{2p} & \theta2_2 \\ \dots & \dots & \dots & \dots & \dots \\ W2_{m1} & W2_{m2} & \dots & W2_{mp} & \theta2_m \end{bmatrix} \quad (4-23)$$

where $W2_{kj}$ is the weight of the connection between the k th hidden neuron and the j th hidden neuron, and $\theta2_k$ is the threshold of the k th output neuron.

LearnRate: (scalar)

The learning rate. This parameter is available only if the neural network block is instantiated with learning. Its value is typically between 0 and 1.

Momentum: (scalar)

The momentum factor. This parameter is available only if the neural network block is instantiated with learning. Its typical value is 0.9.

EpochLength: (scalar)

The length of epoch. This parameter is available only if the neural network block is instantiated with learning. The weights and thresholds are updated at the end of each epoch. `EpochLength` is usually chosen to be the length of the training data set.

- The activation function type is indicated by the name of the neural network block. For example, the Sigmoid and Linear Network name indicates that sigmoid activation functions are used for the hidden layer and linear activation functions are used for the output layer.
- If learning is enabled, the learning algorithm for each neural network block is the back-propagation algorithm.
- If learning is embedded and `vars = 2` is used as a `sim` keyword, the final values of W are written into the **Xmath Commands** window after simulation.

RBFNN

Purpose This block trains a Radial Basis Function Neural Network (RBFNN) to learn a set of patterns. For RBFNN, a learning algorithm is embedded in the neural network if it is instantiated with learning.

Input u : (vector)

The inputs to the one-layer neural network are as follows:

$$u = [u_1 \ u_2 \ \dots \ u_n] \quad (4-24)$$

where u_j is the j th input. Network inputs take the first n input pins of the neural network block.

t : (vector)

The target is expressed as follows:

$$t = [t_1 \ t_2 \ \dots \ t_m] \quad (4-25)$$

where t_i is the i th target and m is the number of output neurons. Target data takes the last m input pins of the neural network block. Target inputs are required only if the neural network block is instantiated with learning.

Output

y : (vector)

The neural network outputs the following:

$$y = [y_1 \ y_2 \ \dots \ y_m] \quad (4-26)$$

where y_i is the output of the i th output neuron and m is the number of neurons ($i = 1, 2, \dots, m$).

Parameters

$W1$: (matrix)

The hidden layer weights are as follows:

$$W1 = \begin{bmatrix} W1_{11} & W1_{12} & \dots & W1_{1n} \\ W1_{21} & W1_{22} & \dots & W1_{2n} \\ \dots & \dots & \dots & \dots \\ W1_{p1} & W1_{p2} & \dots & W1_{pn} \end{bmatrix} \quad (4-27)$$

where $W1_{kj}$ is the weight of the connection between the k th radial basis function (RBF) and the j th input. Notice that no thresholds (or biases) are applied to radial basis functions.

$W2$: (matrix)

The output layer weights are as follows:

$$W2 = \begin{bmatrix} W2_{11} & W2_{12} & \dots & W2_{1p} \\ W2_{21} & W2_{22} & \dots & W2_{2p} \\ \dots & \dots & \dots & \dots \\ W2_{m1} & W2_{m2} & \dots & W2_{mp} \end{bmatrix} \quad (4-28)$$

where W_{2kj} is the weight of the connection between the i th output neuron and the k th radial basis function.

C : (matrix)

The radial basis functions have the following centers:

$$C = \begin{bmatrix} C_{11} & C_{12} & \dots & C_{1n} \\ C_{21} & C_{22} & \dots & C_{2n} \\ \dots & \dots & \dots & \dots \\ C_{p1} & C_{p2} & \dots & C_{pn} \end{bmatrix} \quad (4-29)$$

where the k th row, $\{C_{k1}C_{k2}\dots C_{kn}\}$, is the center of the k th radial basis function.

LearnRate: (scalar)

The learning rate. This parameter is available only if the neural network block is instantiated with learning. Its value is typically between 0 and 1. The adjustments of RBF centers are not affected by the learning rate.

EpochLength: (scalar)

The length of epoch. This parameter is available only if the neural network block is instantiated with learning. The weights and RBF centers are updated at the end of each epoch. *EpochLength* is usually chosen to be the length of the training data set.

Remarks

- The hidden layer of an RBFNN is composed of radial basis functions and the output layer is composed of neurons with linear activation functions.
- If learning is enabled, the Normalized Least Mean Square (NLMS) method will be used to adjust the output weights (W_2).
- Hidden layer weights (W_1) and RBF centers (C) will not be changed during learning. Instead the user can either set values intuitively (for example, by observing the distribution of the training data) or by use of the `kmean()` and `rbfradius()` MathScript functions that come with the NNM distribution.

- If learning is embedded and `vars = 2` is used as a `sim` keyword, the final values of $W1$, $W2$, and C are written into the **Xmath Commands** window after simulation.

Sigmoid Hopfield, Signum Hopfield

Purpose This block simulates a Hopfield Neural Network.

Input u : (vector)

The inputs to the one-layer neural network are as follows:

$$u = [u_1 \ u_2 \ \dots \ u_n] \quad (4-30)$$

where n is the number of neurons. The number of inputs is the same as the number of outputs.

Output y : (vector)

The neural network outputs the following:

$$y = [y_1 \ y_2 \ \dots \ y_n] \quad (4-31)$$

where n is the number of neurons. The number of inputs is the same as the number of outputs.

Parameters $W1$: (matrix)

The hidden layer weights and thresholds (or biases) are as follows:

$$W1 = \begin{bmatrix} W1_{11} & W1_{12} & \dots & W1_{1n} & \theta_1 \\ W1_{21} & W1_{22} & \dots & W1_{2n} & \theta_2 \\ \dots & \dots & \dots & \dots & \dots \\ W1_{n1} & W1_{n2} & \dots & W1_{nn} & \theta_p \end{bmatrix} \quad (4-32)$$

where W_{ij} is the weight of the connection between the i th neuron and the j th neuron, and θ_j is the threshold of the j th neuron.

EpochLength: (scalar)

The length of epoch. This parameter is available only if the neural network block is instantiated with learning. The weights and thresholds are updated at the end of each epoch. `EpochLength` is usually chosen to be the length of the training data set.

Remarks

- For the definition of Hopfield networks, refer to the [Recurrent Neural Networks](#) section of Chapter 2, [Neural Network Fundamentals](#).
- The Sigmoid Hopfield network has sigmoid activation functions and the Signum Hopfield network has signum activation functions.
- Hopfield networks do not have a learning rule available.

Neural Network Examples

This chapter provides examples on how to create and train different neural network models. This includes the following examples:

- Linear Neuron
- RBFNN
- Signum Hopfield Network

Example: Linear Neuron

This example shows how to create a Linear Neuron block with one input and one output. It also shows how to train this neuron offline so that it can approximate a line in two-dimensional space. The basic process for this example is as follows:

1. Create training data.
2. Set up a Linear Neuron model.
3. Train the Linear Neuron model.
4. Instantiate the Linear Neuron block.

Create Training Data

The training data set chosen for this example is a noisy line, which can be generated with the following Xmath commands:

```
m = -0.8;  
c = 0.2;  
u = [-1:0.2:1]';  
noise = 0.3 * (rand(u) - 0.5);  
yd = m * u + c + noise;  
lin11 = [u,yd];
```

where m is the slope, c is an offset, $noise$ is uniform white noise, yd is the target (or the desired output) of the linear neuron, and $lin11$ is the training data set, which has a column of input data and a column of target data.

Alternatively, users can type the following MathScript command to generate the same training data set:

```
maketrd "lin11";
```

This command generates a variable called `main.lin11` in the **Xmath Commands** window.

Set Up a Linear Neuron Model

To set up a Linear Neuron block, complete the following steps:

1. Launch the Neural Network Module tool by typing `nnTool` in the **Xmath Commands** window.
2. In the **Name** field, type a name— for example, `nn1`.
3. Select **Type»Standard Procedure**.
4. Select **Architecture»Neuron**.
5. Select **Model»Linear Neuron**.
6. In the **Inputs** field, type `1`. A Linear Neuron does not have any hidden layers and the number of outputs is fixed to `1`.

Train the Linear Neuron Model Offline

To train the Linear Neuron model offline, complete the following steps:

1. In the **Neural Network Block Editor** window, click the **Trainer** button.
2. For the **Source Type** field, select **Xmath Variable**.
3. In the **Xmath Variable** field, type in `lin11`.
4. In the **Threshold** field, type in `0.1`.
5. In the **Max Iterations** field, type in `100`.
6. Click **Build** to create a Linear Neuron model for training.
7. Click **Edit Par**.
8. In the **Parameter Editor** window, complete the following steps:
 - a. Set **Learning Rate** to `0.3` by changing either the default value or the parameter variable (PV) value. If PV is defined, the default value is ignored.
 - b. Set **Momentum Factor** to `0.9`.
 - c. Set **Length of Epoch** to `11`.
 - d. Click **Done** to validate these changes.
9. Click **Train** to start the training.
10. Click **Plot PI** to plot the performance index versus the number of iterations. The results are shown in Figure 5-1.

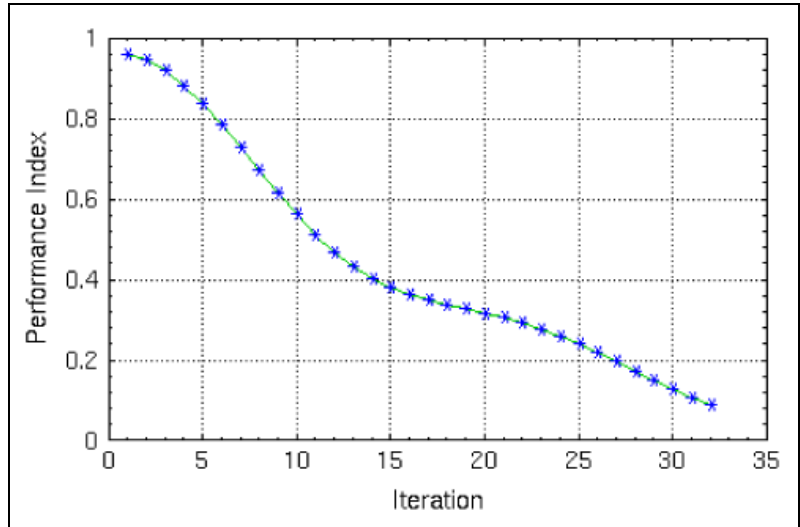


Figure 5-1. Performance Index versus Number of Iterations

11. Click **Plot NN** to plot the target data and the final output from the neural network block. The results are shown in Figure 5-2.

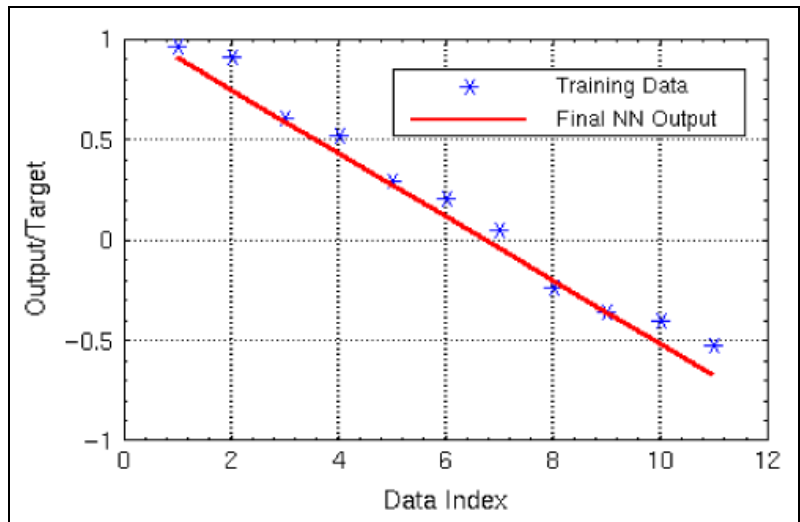


Figure 5-2. Target Data and Final Output from the Linear Neuron

12. Click **Apply** so that the neuron weight matrix in the memory and in the **Xmath Commands** window are updated.
13. Click **Quit** to close the **Offline Trainer** window.

Instantiate a Neural Network Block

To create a Linear Neuron block, complete the following steps:

1. Select or create a discrete-time SuperBlock and make it the active SuperBlock.
2. Click **No** for **With Learning** in the **Neural Network Block Editor** window.
3. Click **Instantiate**.

Example: RBFNN

This section explains how to create a Radial Basis Function Neural Network (RBFNN) to fit data sampled from a `sinc()` function. The basic process for this example is as follows:

1. Create training data.
2. Set up an RBFNN model.
3. Train the RBFNN model.

Create Training Data

The function used for generating the training data is defined as follows:

$$f(x) = \frac{\sin(8x + \varepsilon)}{8x + \varepsilon} \quad (5-1)$$

This function is also known as a *sinc* function. Training data based on this function can be generated easily by running the following MathScript function:

```
maketrnd "sinc11"
```

This function creates an Xmath variable called `sinc11`, where `11` indicates that the training data is for a model having one input and one output. A plot of the `sinc11` variable is shown in Figure 5-3.

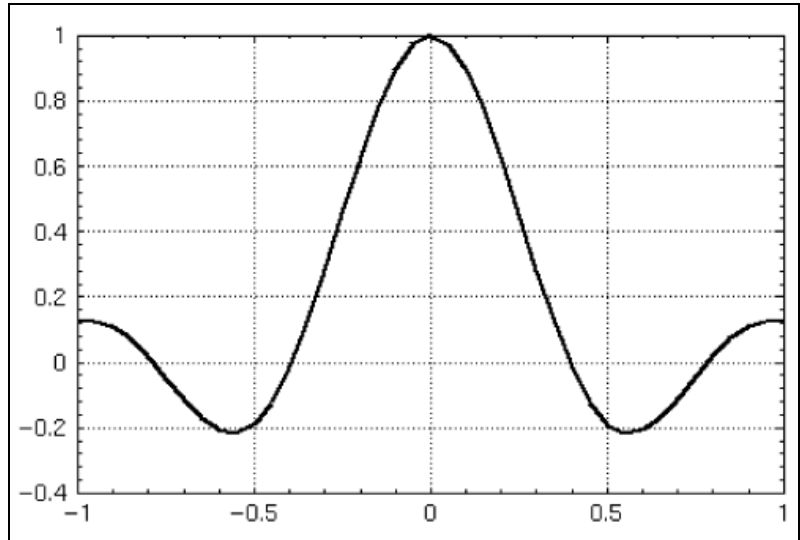


Figure 5-3. Training Data for a RBFNN Model

Set Up an RBFNN Model

To set up the RBFNN model, complete the following steps:

1. If the Neural Network Module is not running, open it by typing `mntool` in the **Xmath Commands** window.
2. In the **Name** field, type `rbfnn`.
3. Select **Type»Standard Procedure**.
4. Select **Architecture»Two-Layer Network**.
5. Select **Model»RBFNN**.
6. In the **Inputs** field, type 1.
7. In the **Hidden Neurons** field, type 10.
8. In the **Outputs** field, type 1.

Train an RBFNN Model

1. Click **Trainer** from the **Neural Network Block Editor** window. This brings up the **Neural Network Offline Trainer** window.
2. In the **Source Type** field, click **Xmath Variable**.
3. In the **Xmath Variable** field, type `sinc11`.
4. In the **Threshold** field, type 0.01.
5. In the **Max Iterations** field, type 100.

6. Click **Build**. A dialog box appears asking you if you want to update RBF centers and standard deviations. Click **Yes**.
7. Click **Edit Par** and set the **Learning Rate** to 0.8 and **Length of Epoch** to 1.
8. Click **Train** to start the training. When the training is finished, a dialog box will appear showing the final performance index and the number of iterations used.
9. Click **Plot PI** to plot the history of the performance index.
10. Click **Plot NN** to plot the final output of the neural network model as well as the training data.

A plot of the final output is shown in Figure 5-4.

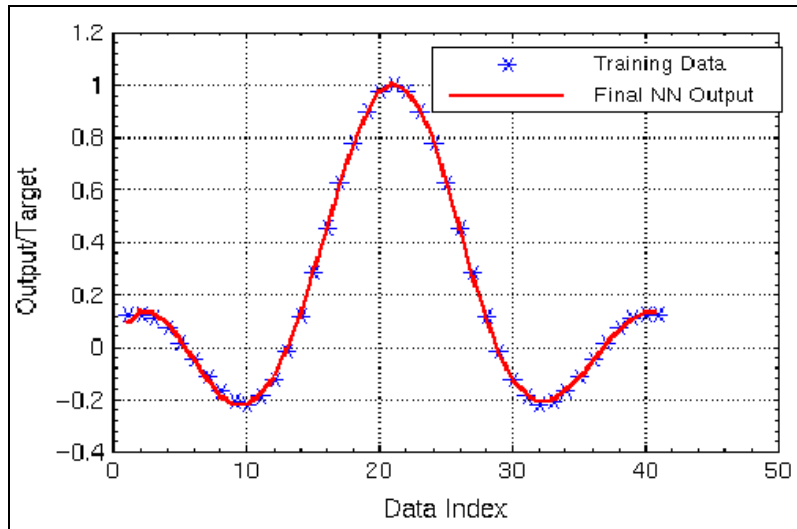


Figure 5-4. Training Data for an RBFNN Model

Example: Signum Hopfield Network

In this example we show how to create a Signum Hopfield Network for pattern recognition. The basic process for this example is as follows:

1. Specify a problem.
2. Set up a Signum Hopfield Network model.
3. Verify it using the Offline Trainer.

Problem Specification

The goal of the problem is to create a Signum Hopfield Network to recognize the patterns 0 and 1 shown in Figure 5-5.

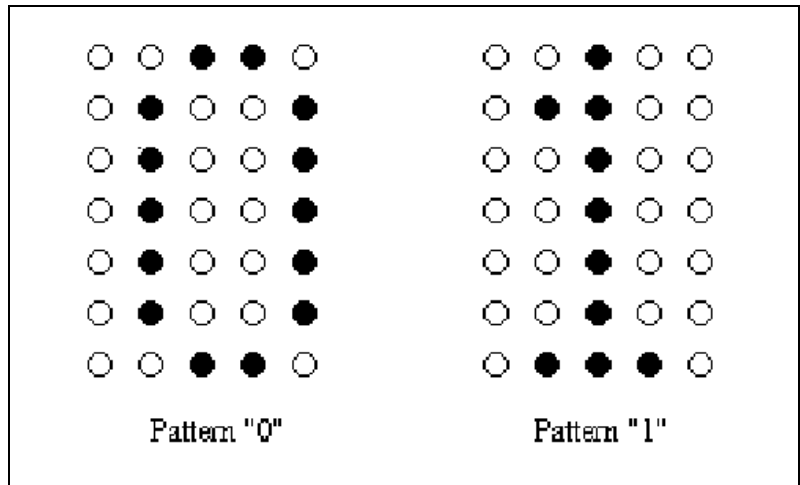


Figure 5-5. Two Patterns to be Recognized by a Signum Hopfield Network

Each pattern has 7 rows and 5 columns. Therefore, we need to create a Hopfield Network having 35 neurons. Each neuron has two possible states: -1 indicates an empty cell and $+1$ indicates a filled cell. Furthermore, there must be a one-on-one mapping between pattern cells to neurons. One possible solution is to number neurons from 1 to 35 then associate the cell on the i th row and j th column with neuron number $(i - 1) * 5 + j$. The two stable states can be denoted as the following:

$$\begin{aligned}
 x1 &= [-1, -1, +1, +1, -1, \\
 &\quad -1, +1, -1, -1, +1, \\
 &\quad -1, +1, -1, -1, +1, \\
 &\quad -1, +1, -1, -1, +1, \\
 &\quad -1, +1, -1, -1, +1, \\
 &\quad -1, +1, -1, -1, +1, \\
 &\quad -1, -1, +1, +1, -1]'; \\
 x2 &= [-1, -1, +1, -1, -1, \\
 &\quad -1, +1, +1, -1, -1, \\
 &\quad -1, -1, +1, -1, -1, \\
 &\quad -1, -1, +1, -1, -1,
 \end{aligned}$$

```
-1, -1, +1, -1, -1,
-1, -1, +1, -1, -1,
-1, +1, +1, +1, -1]';
```

After we have the stable vectors, the desired network weights can be calculated as:

```
W1 = (x1*x1' + x2*x2')/3 - (2/3)*eye(35,35);
```

You also can use the following MathScript to generate the weight matrix:

```
maketrd "hop7x5"
```

which creates an Xmath variable called `main.W1`. In addition, `maketrd()` also creates an Xmath variable `main.hop7x5`, which is a row vector has 35 elements representing a randomly generated pattern. This pattern can be displayed by running the following MathScript command:

```
hoppattern hop7x5, 7, 5
```

You will use `main.hop7x5` later to test the Hopfield Network.

Set Up a Signum Hopfield Network Model

To set up a Signum Hopfield Network model, complete the following steps:

1. If the Neural Network Module is not running, open it by typing `nntool` in the **Xmath Commands** window.
2. In the **Name** field, type `hopfield`, or any other name.
3. Select **Architecture»Recursive Network**.
4. Select **Model»Signum Hopfield**.
5. In the **Inputs** field, type `35`, and then type `Return`. The **Outputs** field will be updated automatically.

Verification by Using the Offline Trainer

1. Click **Trainer** in the **Neural Network Block Editor** window. This opens the **Neural Network Offline Trainer** window.
2. In the **Source Type** field, click **Xmath Variable**.
3. In the **Xmath Variable** field, type `hop7x5`.
4. In the **Max Iterations** field, type `3`. The **Threshold** field will be ignored.
5. Click **Build**. A dialog box appears asking you if you want to update RBF centers and standard deviations. Click **Yes**.

6. Click **Edit Par.** Set **Length of Epoch** to 3 and assert that the parameter variable for **Weights** is $w1$. Click **Done** in the **Parameter Editor** window to validate the changes.
7. Click **Train** to start the testing.

Even though you are using the Offline Training, there is no actual training. The weights of the Hopfield network are fixed during the simulation process. For this case, the Offline Trainer simply provide a testing environment. Results of the testing can be displayed by typing the following MathScript command:

```
hoppattern _nntool.y, 7, 5
```

where `_nntool.y` is the Xmath variable used to save the simulation output.

Pattern 1

```
@ @
  @
@ @@
@ @
  @ @
@ @
  @
```

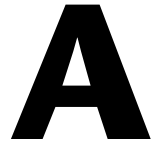
Pattern 2

```
@@
@ @
@ @
@ @
@ @
@ @
@@
```

Pattern 3

```
@@
@ @
@ @
@ @
@ @
@ @
@@
```

Pattern 1 is a distorted pattern fed in to the Hopfield network. After one sample period, the Hopfield network restores the pattern to 0, which is a stable pattern. A stable pattern will remain stable, as shown in Pattern 3.



Technical Support and Professional Services

Visit the following sections of the National Instruments Web site at ni.com for technical support and professional services:

- **Support**—Online technical support resources at ni.com/support include the following:
 - **Self-Help Resources**—For answers and solutions, visit the award-winning National Instruments Web site for software drivers and updates, a searchable KnowledgeBase, product manuals, step-by-step troubleshooting wizards, thousands of example programs, tutorials, application notes, instrument drivers, and so on.
 - **Free Technical Support**—All registered users receive free Basic Service, which includes access to hundreds of Application Engineers worldwide in the NI Discussion Forums at ni.com/forums. National Instruments Application Engineers make sure every question receives an answer.

For information about other technical support options in your area, visit ni.com/services or contact your local office at ni.com/contact.
- **Training and Certification**—Visit ni.com/training for self-paced training, eLearning virtual classrooms, interactive CDs, and Certification program information. You also can register for instructor-led, hands-on courses at locations around the world.
- **System Integration**—If you have time constraints, limited in-house technical resources, or other project challenges, National Instruments Alliance Partner members can help. To learn more, call your local NI office or visit ni.com/alliance.

If you searched ni.com and could not find the answers you need, contact your local office or NI corporate headquarters. Phone numbers for our worldwide offices are listed at the front of this manual. You also can visit the Worldwide Offices section of ni.com/niglobal to access the branch office Web sites, which provide up-to-date contact information, support phone numbers, email addresses, and current events.

Glossary

A

associative memory Neural networks with associative learning and retrieval of information are usually referred to as associative (neural) memories.

B

back-propagation An algorithm that has been used, based on evaluating the error at the output and propagating it back to the previous layers in order to correct the weights of the hidden layers.

C

competitive learning Based on competition between neurons through inhibitory connections, the neuron that is the winner of the competition will have its weights reinforced while the weights of all other neurons are left unchanged or decreased. Competitive learning is utilized in networks such as the Kohonen Neural Networks.

D

data flow A channel between processes where data can flow. Also referred to as a connection.

F

feedforward neural network Consists of several layers of neurons transforming the input signals to the output pattern. Frequently used in pattern recognition, classification, and clustering applications.

N

neural network	A parallel computing architecture (or a computational model) involving a network topology that can be described in terms of its analogy to a directed, weighted graph.
NLMS	Normalized Least Mean Square method.
NMRAS	Neural Model Reference Adaptive Control Systems.

P

Perceptron learning rule	The learning rule originally created for Perceptrons.
--------------------------	-------------------------------------------------------

R

RBF	Radial Basis Function.
RBFNN	Radial Basis Function Neural Network. A class of neural network capable of accurate function approximation.
recurrent networks	Dynamic networks with feedback signals from output to input neurons; mostly used in pattern recognition.

S

supervised learning process	When the target or desired output patterns are available during a training session. In the absence of target output patterns, the learning process is said to be <i>unsupervised</i> , and some other indirect measure of network performance is applied throughout the training session.
-----------------------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

U

unsupervised learning process	When the target or desired output patterns are not available during the training session. In this case, some other indirect measure of network performance is applied throughout the training session.
-------------------------------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

X

Xmath workspace

A working space used to store Xmath partitions and variables. The contents of the workspace can be displayed by using the Xmath command `who`.

Index

A

- actiming SystemBuild keyword, 3-13
- adaptive controller, 1-3
- associative memory, 2-13
- AutoCode, 3-13
 - code generator, 1-1

B

- back-propagation algorithm, 2-6
- Block Panel, 3-3
- BlockScript language, 4-1

C

- code generation, 1-1
- commands, list of, 4-4
- competitive network, 2-11
- competitive output layer, 4-12
- content addressable memory, 2-13
- conventions used in the manual, *iv*
- curve fitting, RBFNN, 1-3

D

- data flow, G-1
- Declaration of Conformity (NI resources), A-1
- diagnostic tools (NI resources), A-1
- display manipulation functions, 4-2
- documentation
 - conventions used in the manual, *iv*
 - NI resources, A-1
- drivers (NI resources), A-1

E

- EpochLength parameter, 4-17
- examples (NI resources), A-1
- export, 3-8
- extending the block library, 4-10

F

- feedforward neural network, 1-2, 2-5, 2-12
- function calls
 - display manipulation, 4-2
 - list of, 4-2

G

- Gaussian output function, 1-3
- generated code, 1-1
- GUI, 3-1

H

- help, technical support, A-1
- hidden layer (neural network), 2-6
- Hopfield
 - block, 4-18
 - Neural Network, 1-3, 2-12

I

- import, 3-8
- initmode SystemBuild keyword, 3-13
- input layer (neural network), 2-6
- instantiate, 3-8
- InStar
 - block, 4-8
 - layer, 4-8

- learning algorithm, 2-11, 4-8
- learning rule, 4-9
- Network, 2-11

instrument drivers (NI resources), A-1

K

KnowledgeBase, A-1
Kohonen Neural Network, 2-10

L

learning

- algorithms, 2-5
- defined, 1-2

Learning Panel, 3-5
LearnRate parameter, 4-8, 4-11, 4-12, 4-17
Linear and Linear Network, 4-13
Linear Layer, 4-6, 4-13
Linear Neuron, 4-4, 5-1
Linear-Threshold Layer, 4-6, 4-13
Linear-Threshold Neuron, 4-4

M

Model Panel, 3-4

N

National Instruments support and services, A-1
network topology, 1-2, G-2
Neural Model Reference Adaptive Control Systems (NMRAS), 1-3
Neural Network

- Block Editor, 3-1
- Catalog, 3-8
- examples, 5-1
- Graphical User Interface, 3-1
- models, 1-1

- Module (NNM), 1-1
- Offline Trainer, 3-10
- Parameter Editor, 3-6

Neural Networks, in SystemBuild, 1-3
NI support and services, A-1
nntool, 3-1

O

Oneneuron block, 4-4
output layer (neural network), 2-6
OutStar

- block, 4-10
- layer, 4-10
- learning algorithm, 2-11
- learning rule, 4-10
- Network, 2-11

P

Perceptron learning rule, 2-4
programming examples (NI resources), A-1

R

radial basis function (RBF), 4-16
Radial Basis Function Neural Network block (RBFNN), 1-3, 4-15
RBFNN, 4-15
recurrent neural network, 1-2, 2-12

S

Sigmoid and Linear Network, 4-13
Sigmoid and Sigmoid Network, 4-13
Sigmoid Hopfield, 4-18
Sigmoid Layer, 4-6, 4-13
Sigmoid Neuron, 4-4
Signum Hopfield, 4-18
single neurons, 2-1
software (NI resources), A-1

starting the NNM, 3-1
supervised training, 1-2, G-2
support, technical, A-1

T

technical support, A-1
Threshold Layer, 4-6, 4-13
Threshold Neuron, 4-4
training
 process, 1-2
 supervised, 1-2
 unsupervised, 1-2
training and certification (NI resources), A-1
troubleshooting (NI resources), A-1

U

unsupervised training, 1-2, G-2

V

vars = 2 keyword option, 1-4, 3-13

W

Web resources, A-1

X

Xmath Commands window, 4-6