What is the NeuralBuilder?

The NeuralBuilder is a sophisticated neural network builder that sends commands to NeuroSolutions to automatically construct a fully-functional neural network. The object-oriented simulation environment of NeuroSolutions gives the user an unprecedented flexibility to construct neural network simulations. However, flexibility and power require a substantial amount of knowledge about neural networks. The NeuralBuilder aids the user by encapsulating the network building rules and reducing the user decisions down to an easy, step-by-step procedure.

Much of the construction effort necessary to build neural networks with NeuroSolutions becomes transparent to the user. There is a wide range of conventional neural models to choose from. When a model is selected, the user is lead through a series of panels containing the configuration parameters for the model. Each parameter has a default value that can be overwritten. After completing all the panels, the utility makes calls to NeuroSolutions to automatically construct the network according to the specifications.

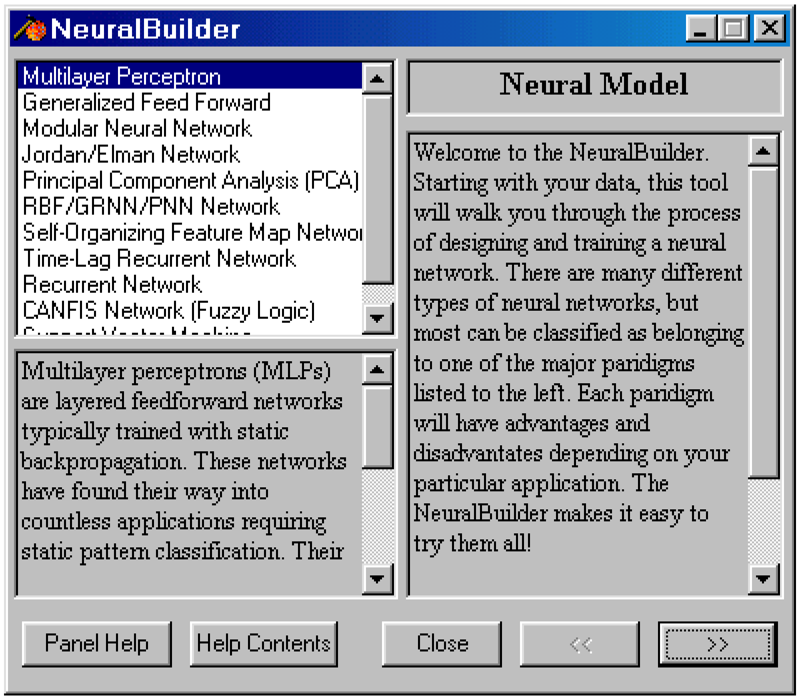
Supported Models

For each model there are configuration instructions, a summary of advantages and disadvantages, examples of typical applications, and a short summary of the theory. The following models are presently supported by the NeuralBuilder:

1. Multilayer Perceptrons (MLPs)
2. Generalized Feedforward MLPs
3. Modular Feedforward Networks
4. Radial Basis Functions (RBFs) Networks
5. Jordan/Elman Networks
6. Principal Component Analysis (PCA) Networks
7. Self-Organizing Feature Map (SOFM) Networks
8. Time Lagged Recurrent Networks (TLRNs)
9. General Recurrent Networks
10. CANFIS Networks (Fuzzy Logic)
11. Support Vector Machines (SVM)

Design Steps of the NeuralBuilder

By default, the NeuralBuilder is accessible from the main menu of NeuroSolutions. Select the NeuralBuilder item from the Tools menu of NeuroSolutions. The NeuralBuilder panel will appear in the center of the display with a list of the supported neural models (see Supported Models), a brief description of the selected model, and brief instructions on making a selection (see figure below). Select the desired neural model by single-clicking on the item within the list.



Multilayer perceptrons (MLPs) are layered feedforward networks typically trained with static backpropagation. These networks have found their way into countless applications requiring static pattern classification. Their

Welcome to the NeuralBuilder. Starting with your data, this tool will walk you through the process of designing and training a neural network. There are many different types of neural networks, but most can be classified as belonging to one of the major paridigms listed to the left. Each paridigm will have advantages and disadvantantes depending on your particular application. The NeuralBuilder makes it easy to try them all!

Opening panel of the NeuralBuilder

If you are uncertain which neural model to apply to your problem, it is recommended that you start with a simple model by selecting the multilayer perceptron (MLP). The "Multilayer Perceptrons" section of this chapter contains an explanation of the power of this model and some alternatives that may provide a solid start for enhancements. If you want to know more about neural networks, it is recommended that you read the "Introduction to Neural Computation" chapter of the NeuroSolutions on-line manual. The "Concepts" chapter of this manual covers the details and overall structure of NeuroSolutions.

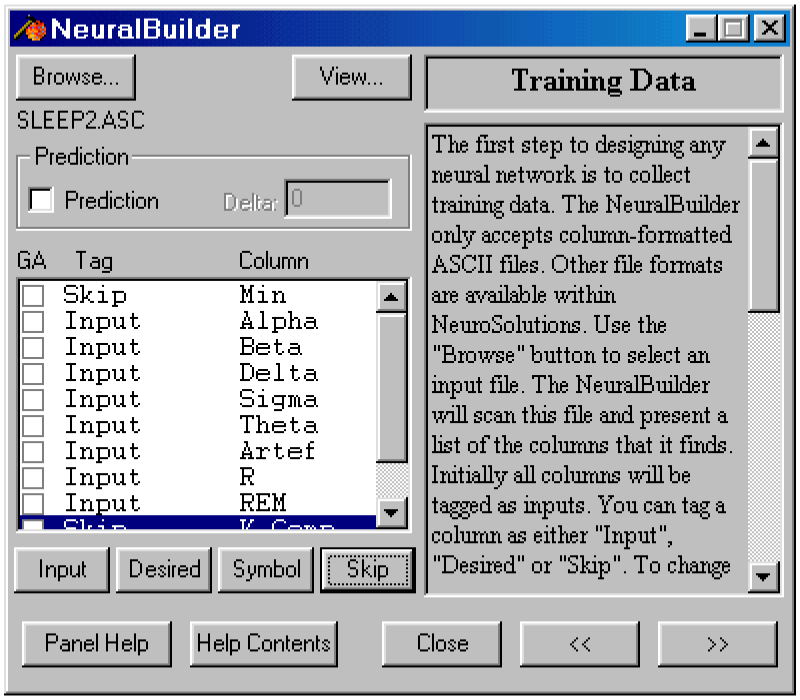
The 7 basic steps of neural network construction are:

- Step 1: Input/desired data file selection
- Step 2: Network Analysis
- Step 3: Neural Topology
- Step 4: Layer configuration
- Step 5: Simulation control
- Step 6: Data Display
- Step 7: Simulation

The NeuralBuilder will sequentially follow these steps. You will simply need to click on the >> button of the active panel to go through the building process. If you change your mind, you can go back by clicking on the << button. To leave the NeuralBuilder at any time just click on the Close button (see figure above). Each neural model can have unique parameters to set, so steps 3, 4 and 5 are dependent on the chosen model. Design steps 1, 2, 6 and 7 are common to all of the models.

Step 1: Input/Desired Data File Selection

This step links NeuroSolutions to your data stored on the computer file system (see figure below). All the models currently supported by the NeuralBuilder use some form of supervised learning. Hence, the user is required to supply an input file and a desired response file. From the Training Data panel, pressing the Browse button will open the Windows file browser, which allows you to specify the input/desired data from the file system. You have basically two options: either the input and desired data are in the same file, or they exist in two independent files.



SLEEP2.ASC

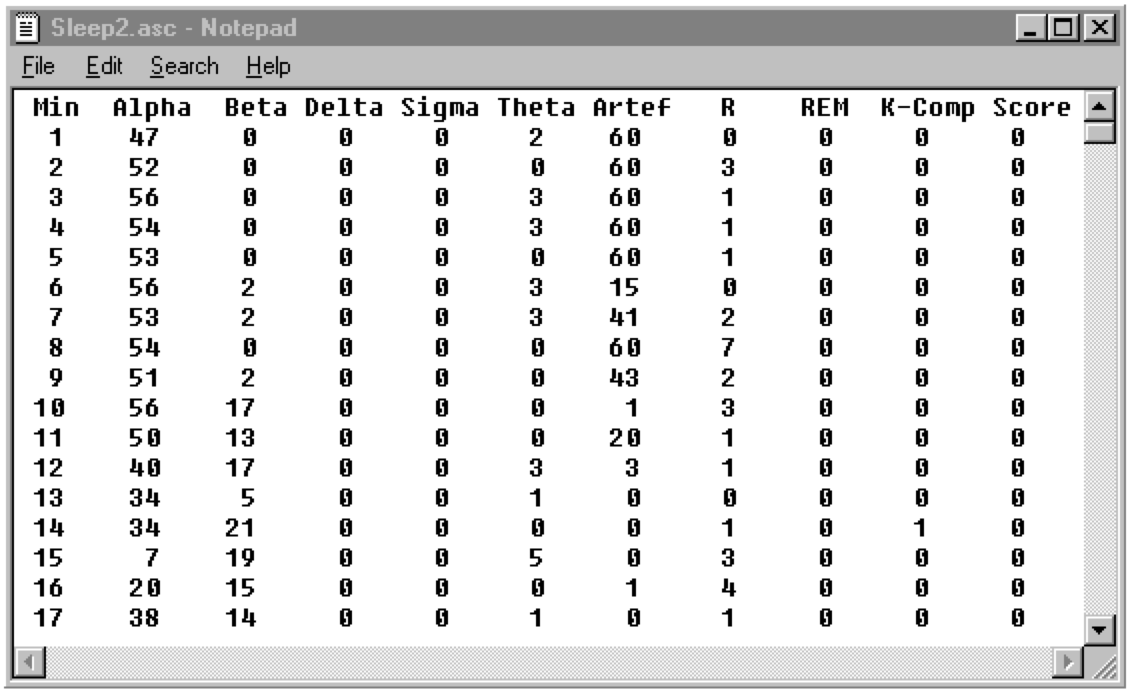
Delta: 0

GA	Tag	Column
☐	Skip	Min
☐	Input	Alpha
☐	Input	Beta
☐	Input	Delta
☐	Input	Sigma
☐	Input	Theta
☐	Input	Artef
☐	Input	R
☐	Input	REM

The first step to designing any neural network is to collect training data. The NeuralBuilder only accepts column-formatted ASCII files. Other file formats are available within NeuroSolutions. Use the "Browse" button to select an input file. The NeuralBuilder will scan this file and present a list of the columns that it finds. Initially all columns will be tagged as inputs. You can tag a column as either "Input", "Desired" or "Skip". To change

Training Data panel of the NeuralBuilder

The NeuralBuilder requires the data files to be ASCII text, arranged into columns. The data elements can be integers, floating point numbers (decimal point or exponential notation), or symbols (non-numeric). The first row in the file is assumed to be the column titles (which are used to tag the training data), and the data starts at the second line. Either spaces or tabs can be used as data delimiters. The figure below shows an example data file.



```
Min  Alpha  Beta Delta Sigma Theta Artef   R   REM  K-Comp Score
 1    47     0     0     0     2    60    0     0     0     0
 2    52     0     0     0     0    60    3     0     0     0
 3    56     0     0     0     3    60    1     0     0     0
 4    54     0     0     0     3    60    1     0     0     0
 5    53     0     0     0     0    60    1     0     0     0
 6    56     2     0     0     3    15    0     0     0     0
 7    53     2     0     0     3    41    2     0     0     0
 8    54     0     0     0     0    60    7     0     0     0
 9    51     2     0     0     0    43    2     0     0     0
10    56    17     0     0     0     1    3     0     0     0
11    50    13     0     0     0    20    1     0     0     0
12    40    17     0     0     3     3    1     0     0     0
13    34     5     0     0     1     0    0     0     0     0
14    34    21     0     0     0     0    1     0     1     0
15     7    19     0     0     5     0    3     0     0     0
16    20    15     0     0     0     1    4     0     0     0
17    38    14     0     0     1     0    1     0     0     0
```

Column-formatted ASCII file used by the NeuralBuilder

This organization was chosen to be compatible with text files produced by spreadsheet programs, and also to enable the NeuralBuilder to automatically configure the input/output network layers for you. The NeuralBuilder assumes that each numeric input file column is a network input, so it assigns an input processing element (PE) to it. Likewise, an output PE will be assigned to each column specified as a desired output.

Specify the input data file in the Windows file browser by selecting (single-clicking) the file you want open and clicking on the OK button. After making your selection, the NeuralBuilder displays the name of the selected file. The box labeled with the headers Tag and Column will display the column titles read from the opened data file (see Training Data Panel and Column-formatted ASCII figures above).

Note that all columns are configured as inputs. You can change this assignment by simply double-clicking on the corresponding entry. The input label will toggle to "Desired". Alternatively, you can select the column title and press the Desired button. By doing this, you are specifying that the desired response is to be read from the same file as the input.

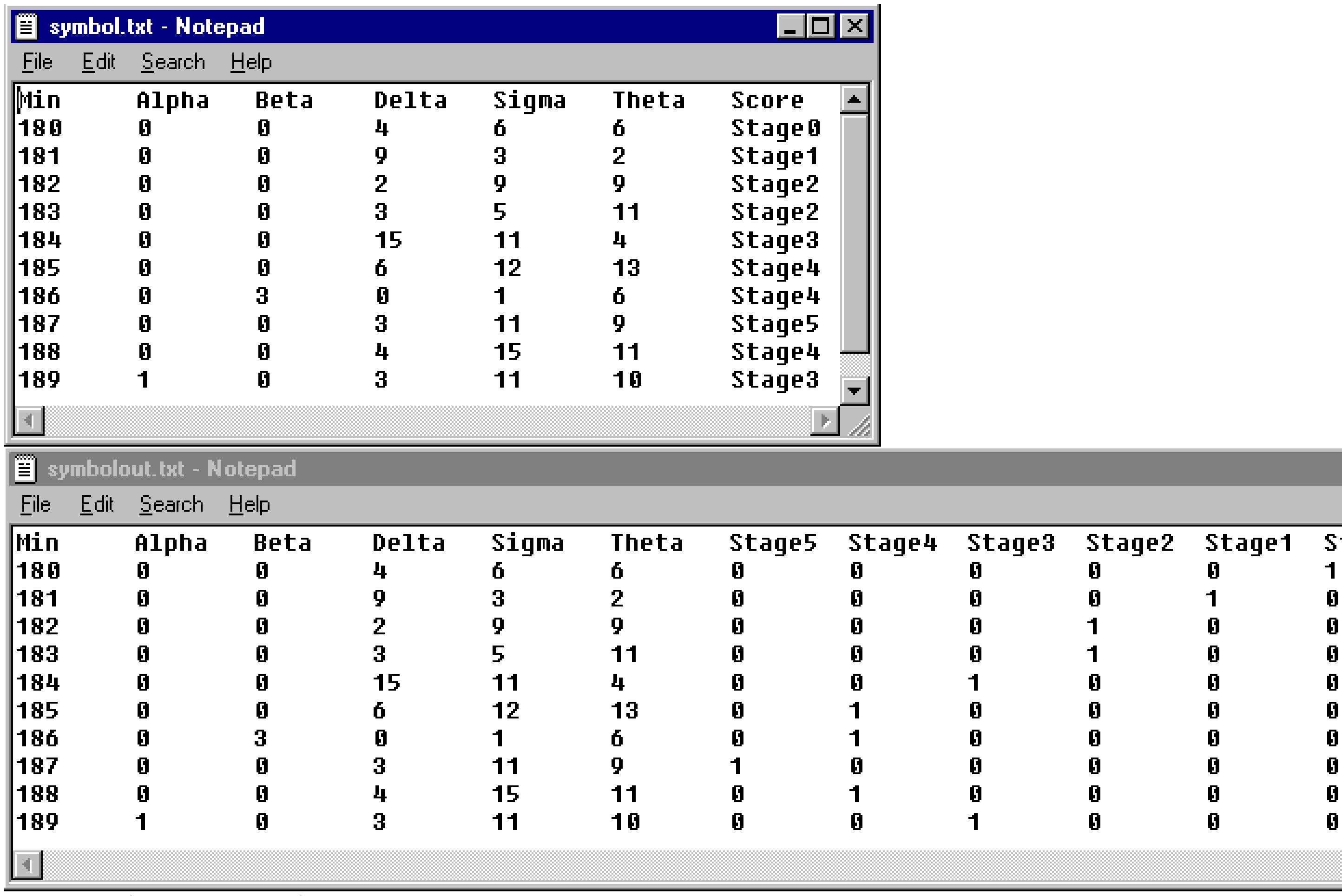
You can have the system ignore the data from a column by double-clicking on a column tagged as Desired, or by selecting this item and clicking the Skip button. Double-clicking on a skipped column toggles that column back to being an input.

If you are unsure whether an input should be skipped or not, check the GA check box next to that input. Each input tagged as GA (optimize using a genetic algorithm) will be selected (tagged as Input) or de-selected (tagged as Skip) based on the performance of multiple training runs. The combination of inputs that produces the lowest error across these training runs will be used for the final model.

Another option within this panel is contained within the Prediction box. When the Prediction switch is set, the simulation mode is switched from classification to prediction. In prediction, the desired response is the input advanced by a given (user specified) number of data samples. The number of samples to advance is specified in the Delta field. The default value is one sample prediction. The Desired button is changed to a Prediction button, and the columns are now tagged as either Input, Prediction or Skip.

A given column selected for prediction will be used by the network as both an input and a desired response. The difference is that the desired response is advanced by Delta samples. Note that by setting this panel to prediction mode, you have specified that the desired response is to be read from the same file as the input. Therefore, the NeuralBuilder will not give the option to select a separate file for the desired response.

Recall that each column represents one channel (PE) of input data. The exception to this rule is when a column contains symbolic instead of numeric data. A column is specified as symbolic by selecting an item marked as Input, Desired or Predict, and then pressing the Symbol button. Note that the item is now tagged with a "(S)". A new column will be created for each unique symbol contained within a symbolic column. The figure below shows an example of a file with a symbolic column and how the symbols are expanded into multiple columns.

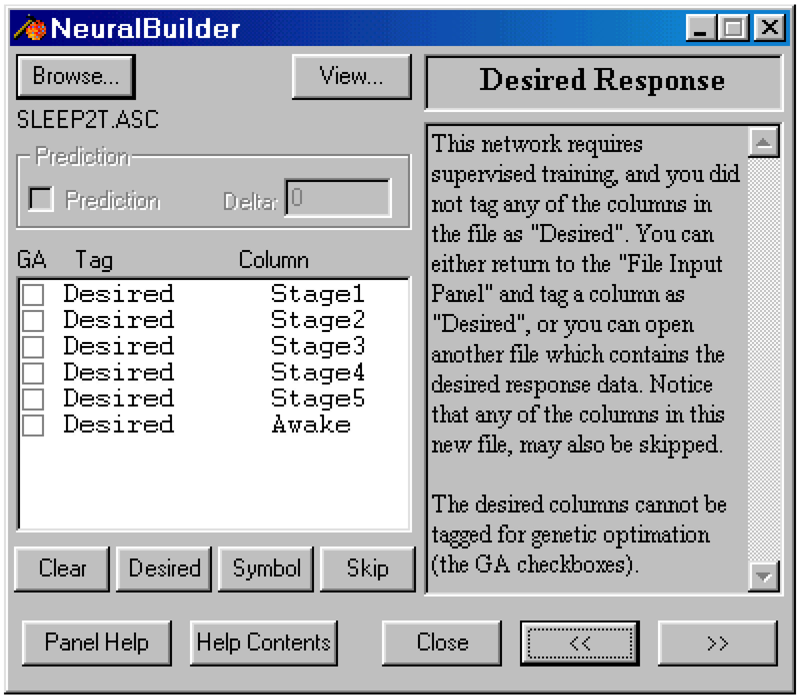


```
Min     Alpha   Beta    Delta   Sigma   Theta   Score
180     0       0       4       6       6       Stage0
181     0       0       9       3       2       Stage1
182     0       0       2       9       9       Stage2
183     0       0       3       5       11      Stage2
184     0       0       15      11      4       Stage3
185     0       0       6       12      13      Stage4
186     0       3       0       1       6       Stage4
187     0       0       3       11      9       Stage5
188     0       0       4       15      11      Stage4
189     1       0       3       11      10      Stage3
```

```
Min     Alpha   Beta    Delta   Sigma   Theta   Stage5  Stage4  Stage3  Stage2  Stage1  S
180     0       0       4       6       6       0       0       0       0       0       1
181     0       0       9       3       2       0       0       0       0       1       0
182     0       0       2       9       9       0       0       0       1       0       0
183     0       0       3       5       11      0       0       0       1       0       0
184     0       0       15      11      4       0       0       1       0       0       0
185     0       0       6       12      13      0       1       0       0       0       0
186     0       3       0       1       6       0       1       0       0       0       0
187     0       0       3       11      9       1       0       0       0       0       0
188     0       0       4       15      11      0       1       0       0       0       0
189     1       0       3       11      10      0       0       1       0       0       0
```

Illustration of the expansion of one symbolic column into six numeric columns

If you are not in prediction mode and there are no columns tagged as Desired (given that there is at least one column tagged as Input), then the next panel (after pressing the >> button) will be the Desired Response panel (see figure below).



Desired Response

GA	Tag	Column
☐	Desired	Stage1
☐	Desired	Stage2
☐	Desired	Stage3
☐	Desired	Stage4
☐	Desired	Stage5
☐	Desired	Awake

This network requires supervised training, and you did not tag any of the columns in the file as "Desired". You can either return to the "File Input Panel" and tag a column as "Desired", or you can open another file which contains the desired response data. Notice that any of the columns in this new file, may also be skipped.

The desired columns cannot be tagged for genetic optimation (the GA checkboxes).

Desired Response panel of the NeuralBuilder

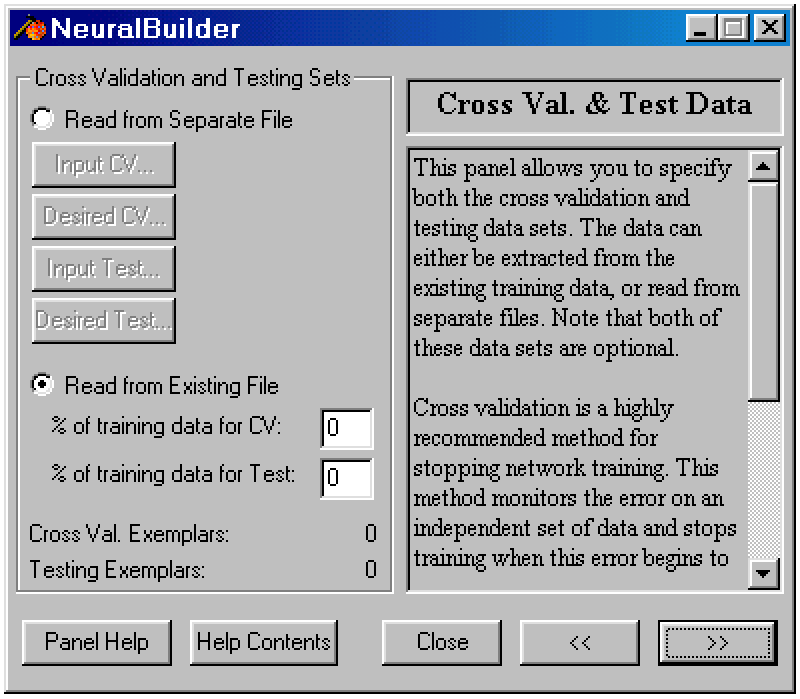
In this panel, the mechanics are basically the same as the previous panel. The NeuralBuilder expects the same file columns and organization as discussed for the input file. The file is opened using the Browse button, but now the default tag for the columns is Desired. Note that you cannot specify any column of this file as input data and the ability to optimize the column selection using a genetic algorithm (GA) is not available.

You should know that the NeuralBuilder automatically normalizes the input data for you. This is either in the range between 0 and 1 or between -1 and 1 depending on the type of non-linearity selected in the Layer panels. This is covered in more detail under Step4: Layer Configuration.   This normalization can be overridden after the network is constructed.

Step 2

Step 2: Cross Validation and Test Data

The cross validation set is used to determine the level of generalization produced by the training set. Cross validation is executed in concurrence with the training of the network. Every so often, the network weights are frozen, the cross validation data is fed through the network, and the results are reported. The stop criteria of the controller can be based on the error of the cross validation set instead of the training set to insure this generalization (see Step 5: Simulation Control).



% of training data for CV: 0

% of training data for Test: 0

Cross Val. Exemplars: 0

Testing Exemplars: 0

Cross Val. & Test Data

This panel allows you to specify both the cross validation and testing data sets. The data can either be extracted from the existing training data, or read from separate files. Note that both of these data sets are optional.

Cross validation is a highly recommended method for stopping network training. This method monitors the error on an independent set of data and stops training when this error begins to

Cross Validation Data panel of the NeuralBuilder

The cross validation set may be extracted from the existing file(s) specified in the previous panels, or it may be contained within separate files. In the case that the cross validation set is to be extracted from the training file(s), you simply specify the percentage of exemplars to extract. The NeuralBuilder calculates the number of exemplars used for cross validation and displays this value at the bottom of the panel.

If the cross validation set data comes from separate files, click on the Read from Separate Files radio button. If your training data is divided into two files (an input file and a desired response file), then you must specify two additional files to be used for cross validation. For the case where the input and the desired response of the training set are contained within the same file, you only need to specify one file for cross validation. The files are selected using the Input and Desired buttons in the same way as the Browse button was used in the previous panel.

The testing set is used to test the performance of the network. Once the network has been trained, the weights are then frozen, the testing set is fed into the network, and the network output is compared with the desired output. The testing set is specified in the same manner as the cross validation set.

You may configure your network with both, one or neither of these data sets. Setting the percentage field to zero specifies that the data set is to be excluded.

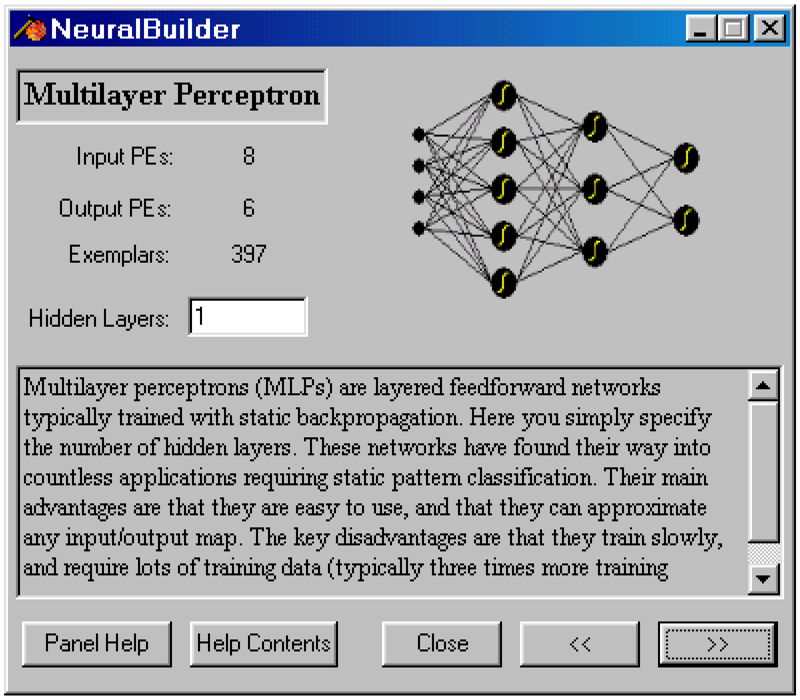
The NeuralBuilder assumes that the structure of your training, testing and cross validation data files are the same. First of all, the column headings selected within the training file(s) must also exist within the cross validation file(s) and testing files. If you are using separate files for the input and desired response, you should verify that there is the same number of exemplars contained within each one.

Step 3

Step 3: Neural Topology

The next panel is used to define the topology specifics based on the neural model selected earlier. Since this panel is dependent on the neural model, just a brief overview of the panel will be given here (see figure below). See Supported Models for specifics on each of the supported models.

This panel displays the name of the selected neural model, a block diagram showing the relevant aspects, and a synopsis of the selections made thus far. The number of input PEs and output PEs was obtained according to your selection of input columns and desired columns. If these numbers are not correct, then you may click the << button to make changes to one of the previous panels.



NeuralBuilder

Multilayer Perceptron

Input PEs: 8

Output PEs: 6

Exemplars: 397

Hidden Layers: 1

Multilayer perceptrons (MLPs) are layered feedforward networks typically trained with static backpropagation. Here you simply specify the number of hidden layers. These networks have found their way into countless applications requiring static pattern classification. Their main advantages are that they are easy to use, and that they can approximate any input/output map. The key disadvantages are that they train slowly, and require lots of training data (typically three times more training

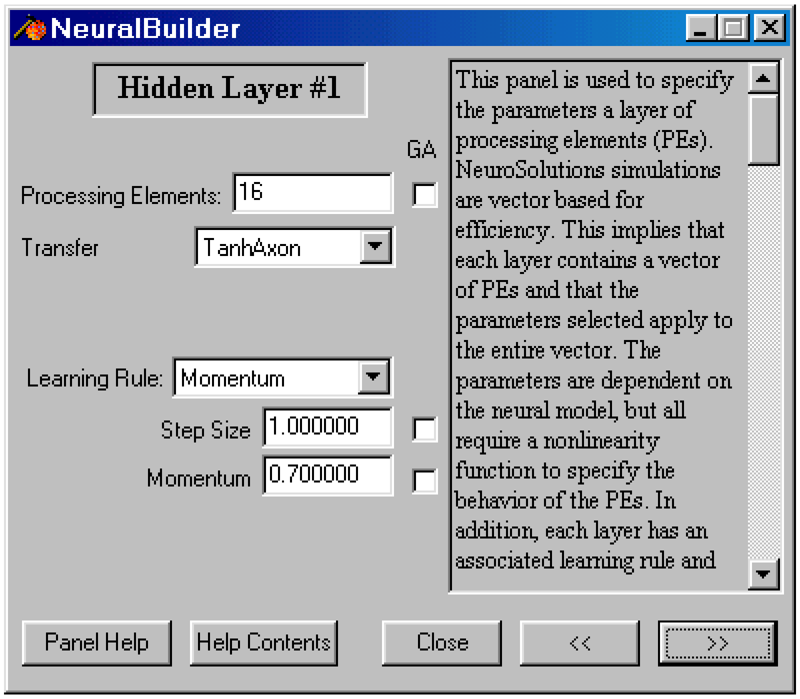
Multilayer Perceptron panel of the NeuralBuilder

For all neural models, this panel is also used to specify the number of hidden layers comprising the topology. The NeuralBuilder will create this many additional Hidden Layer panels, which are described in the next section. Note that hidden layers are optional and that a zero may be entered in the Hidden Layers field.

Step 4

Step 4: Layer Configuration

The Layer panels are used to specify the parameters for each of the hidden layers, as well as the output layer. NeuroSolutions simulations are vector based for efficiency. This implies that each layer contains a vector of PEs and that the parameters selected apply to the entire vector. The parameters are dependent on the neural model, but all require a nonlinearity function to specify the behavior of the PEs. In addition, each layer has an associated learning rule and learning parameters. The number of PEs and learning parameters are entered in the corresponding fields. The learning rule and nonlinearity are selected from a list of options contained within pull-down menus (see figure below).



Processing Elements: 16
Transfer: TanhAxon
Learning Rule: Momentum
Step Size: 1.000000
Momentum: 0.700000

This panel is used to specify the parameters a layer of processing elements (PEs). NeuroSolutions simulations are vector based for efficiency. This implies that each layer contains a vector of PEs and that the parameters selected apply to the entire vector. The parameters are dependent on the neural model, but all require a nonlinearity function to specify the behavior of the PEs. In addition, each layer has an associated learning rule and

Layer panel of the NeuralBuilder

The table below summarizes the types of transfer functions that are available in NeuroSolutions. See the on-line documentation of NeuroSolutions for more detailed information on the PE types contained within the various axon components. If you find that the menu does not contain all of these selections, it means that it found the missing component(s) to be inappropriate (or incompatible) for that neural model.

Axon Selections

PE Transfer Function	Description
Axon	stores input
BiasAxon	adds a bias
LinearAxon	adds a bias and scales
LinearTanhAxon	piecewise linear (-1/+1)
LinearSigmoidAxon	piecewise linear (0/1)
TanhAxon	hyperbolic tangent (-1/+1)
SigmoidAxon	sigmoid (0/1)
SoftMaxAxon	sum to one

Each one of these axon components applies a static map to the data it receives. The map can be linear or nonlinear, or it can normalize the input to the PE.
Learning from the data is the essence of neurocomputing. Every PE that has an adaptive parameter must change

it according to some pre-specified procedure. Backpropagation is by far the most common form of learning (see Multilayer Perceptrons and the on-line documentation of NeuroSolutions for a more in depth discussion). Here it is sufficient to say that the weights are changed based on their previous value and a correction term. The learning rule is the means by which the correction term is specified. Once the particular rule is selected, the user must still specify how much correction should be applied to the weights, referred to as the learning rate. If the learning rate is too small, then learning takes a long time. On the other hand, if it is set too high, then the adaptation diverges and the weights are unusable.

The table below shows the options contained within the Learning Rule pull-down menu. The NeuralBuilder selects the momentum component as default. In searching with the momentum component there are two parameters to be selected: the step size and the momentum. The NeuralBuilder provides a default value for the learning rates. Be ready to modify this selection if you see that learning is unstable or very slow.

Learning Rule

Name	Description
Step	Gradient Information
Momentum	Gradient and Weight Change (Momentum)
Quickprop	Gradient and Rate of Change of Gradient
DeltaBarDelta	Adaptive Step Sizes for Gradient plus Momentum
Conjugate Gradient	Second Order method for Gradient

The default setting for the Processing Elements, Step Size and Momentum are generally a good starting point. However, if you would like to try to find the optimum setting for one or more of these parameters, check the corresponding GA check box. Each parameter tagged as GA (optimize using a genetic algorithm) will be set based on the performance of multiple training runs. The combination of parameter settings that produces the lowest error across these training runs will be used for the final model.

After specifying the parameters for the hidden layers, the next panel is the Output Layer panel. This panel is similar to the Hidden Layer panels, except that the number of PEs is fixed. Recall that this number is determined by the number of columns selected as your desired response.

Step 5

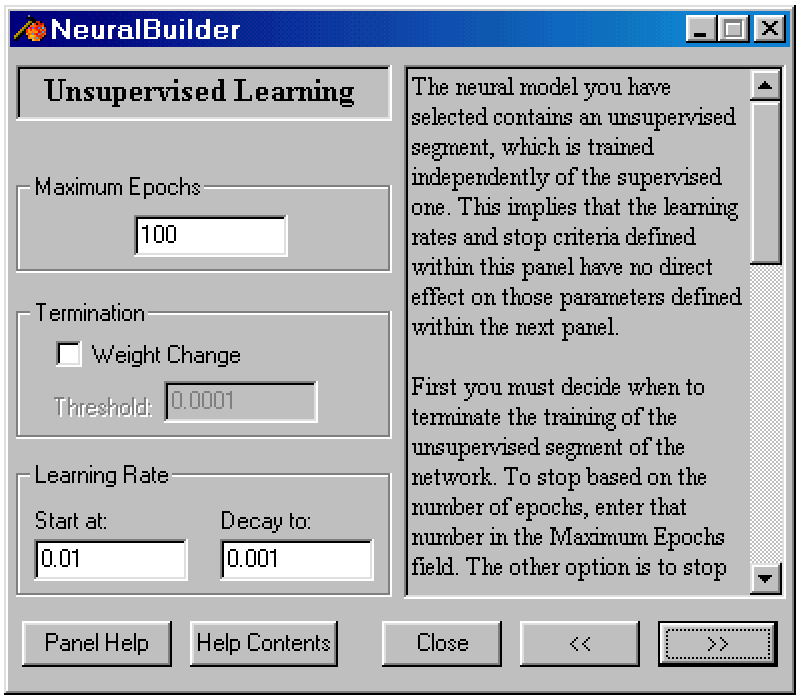
Step 5: Simulation Control

The next step in the configuration of your network is the simulation control. Here you will select how to train the network and when to stop the simulations. There are separate panels for supervised and unsupervised learning. Some neural models require both panels but most (such as the MLP) require only the Supervised Learning Control panel.

Unsupervised Learning

If the selected neural model contains an unsupervised layer, then the Unsupervised Learning panel is displayed next. Note that for the hybrid networks of the NeuralBuilder (those that contain both a supervised and an unsupervised segment), the unsupervised segment is trained independently of the supervised one. This implies that the learning rates and stop criteria defined within this panel have no direct effect on those parameters defined within the Supervised Learning Control panel.

First you must decide when to terminate the training of the unsupervised segment of the network. To stop based on the number of epochs, enter that number in the Maximum Epochs field. The other option is to stop the unsupervised training based on the change in the weights. First set the Maximum Epochs to a high value so that the training will not stop prematurely. Then Activate the termination based on the Weight Change. This specifies that the training will terminate when all of the weights change by less than the specified value from one epoch to the next. This parameter can be changed from its default value of 0.0001 (see figure below).


NeuralBuilder

Unsupervised Learning

Maximum Epochs: 100

Termination: Weight Change [ ] Threshold: 0.0001

Learning Rate: Start at: 0.01 Decay to: 0.001

The neural model you have selected contains an unsupervised segment, which is trained independently of the supervised one. This implies that the learning rates and stop criteria defined within this panel have no direct effect on those parameters defined within the next panel.

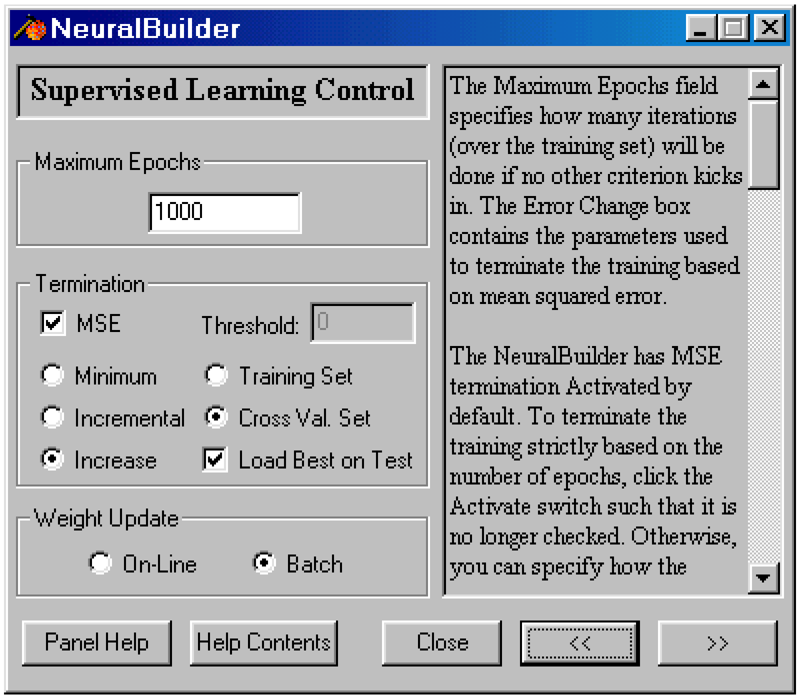
First you must decide when to terminate the training of the unsupervised segment of the network. To stop based on the number of epochs, enter that number in the Maximum Epochs field. The other option is to stop

Unsupervised Learning Control panel of the NeuralBuilder

The other unsupervised parameters specify the learning rate (the amount to change the weights between epochs). Unsupervised learning usually performs best when the learning rate starts out high and then gradually decays during training. This allows the network to find an approximate solution quickly, and then focus on the details of the problem. The two parameters define the starting point of the learning rate and the value that the learning rate is to decay to (provided that it runs for the maximum number of epochs).

Supervised Learning

Similar to the Unsupervised Learning panel, the stop criteria for the supervised training of the network must be specified. The Maximum Epochs field specifies how many iterations (over the training set) will be done if no other criterion kicks in. The Error Change box contains the parameters used to terminate the training based on mean squared error (see figure below).



Supervised Learning Control

Maximum Epochs: 1000

Termination

[✓] MSE Threshold: 0

( ) Minimum ( ) Training Set

( ) Incremental [✓ Cross Val. Set]

[✓ Increase] [✓] Load Best on Test

Weight Update

( ) On-Line [✓ Batch]

The Maximum Epochs field specifies how many iterations (over the training set) will be done if no other criterion kicks in. The Error Change box contains the parameters used to terminate the training based on mean squared error.

The NeuralBuilder has MSE termination Activated by default. To terminate the training strictly based on the number of epochs, click the Activate switch such that it is no longer checked. Otherwise, you can specify how the

Supervised Learning Control panel of the NeuralBuilder

The NeuralBuilder has MSE termination activated by default. To terminate the training strictly based on the number of epochs, click the MSE switch such that it is no longer checked. Otherwise, you can specify how the training terminates as a function of the desired error level. There are three functions to choose from, but the first two are the most applicable to the MSE of the training set. The Minimum function terminates when the MSE drops below the specified Threshold. The Incremental function terminates when the change in MSE from one iteration to the next is less than the threshold. Note that the default threshold changes when you select between the two functions. As expected, the default Incremental error is much smaller than the Minimum error.

The other option of MSE termination is to base the stop criteria on the cross validation set (using Cross Validation from the Network Analysis panel) instead of the training set. As mentioned earlier, this tends to be a good indicator of the level of generalization that the network has achieved. Increase is the default function when using the cross validation set for MSE termination. This stops the network when the MSE of the cross validation set begins to increase. This is an indication that the network has begun to overtrain. Overtraining is when the network simply memorizes the training set and is unable to generalize the problem. The other two stopping functions described above can be applied to the cross validation set as well as the training set.

The weights of the best network (the one with the lowest MSE) are automatically saved by default. By setting the "Load Best on Test" switch, these weights will automatically be loaded into the network before the "Testing" set is fed through the network.

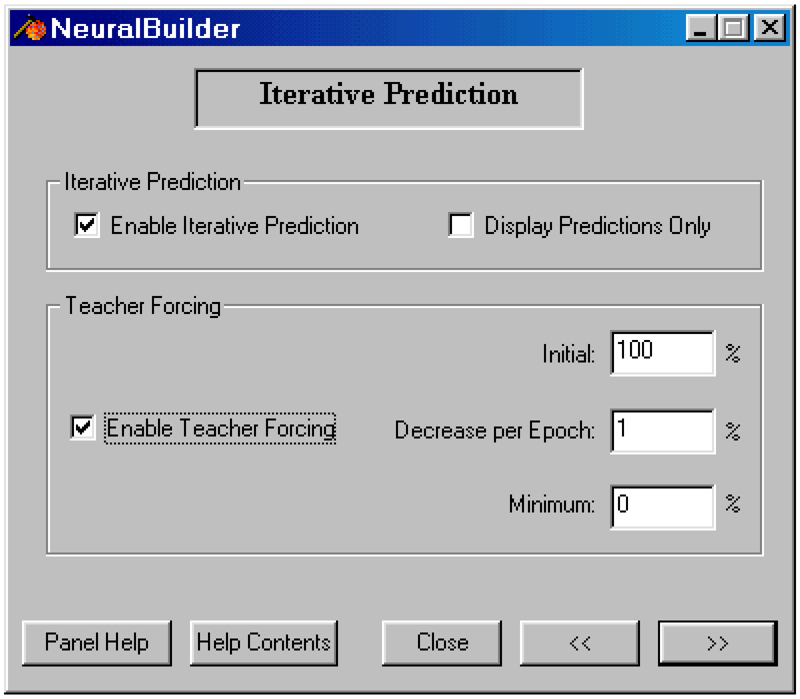
The Supervised Learning Control panel also specifies when the weights are updated. On-Line learning updates the weights after the presentation of each exemplar. In contrast, Batch learning updates the weights after the presentation of the entire training set.

Iterative Prediction

There is a small subset of problems that are best modeled using a method called iterative prediction. This procedure feeds the first sample of each exemplar from the input file and then obtains the remaining input exemplars from the network output. The most common way to train an iterative prediction network is with teacher forcing. This algorithm feeds *x* samples from the input file and the remaining *y* samples from the network output, where *x*+*y* is the number of samples per exemplar (z).

The Iterative Predition panel will be displayed after the Supervised panel if the following conditions are met:

1. The neural model is either "Time-Lag Recurrent" or "General Recurrent".
2. The "Predict" switch is set (from the Training Data panel).
3. All columns are selected as either "Skip" or "Predict" (none as "Input").
4. The "Delta" is greater than 1.



Iterative Prediction

[✓] Enable Iterative Prediction [ ] Display Predictions Only

Teacher Forcing

[✓] Enable Teacher Forcing

Initial: 100 %

Decrease per Epoch: 1 %

Minimum: 0 %

Supervised Learning Control panel of the NeuralBuilder

Check the "Enable Iterative Prediction" switch to use iterative prediction, otherwise you may skip to the next panel. The "Display Predictions Only" configures the probes to only display the last sample of each exemplar (the predicted value). Leaving this box unchecked will display all of the intermediate predictions in the exemplar.

To train using teacher forcing, check the "Enable Teacher Forcing" box. By default, the training starts out by forcing all (100%) of the input samples (i.e., $x = z$ and $y = 0$). Each epoch the number of forced samples is reduced by 1% of the total number of samples per exemplar (i.e., x = 0.99 * z and y = 0.01 * z). Eventually, there will be no forced inputs (0%) such that it will only read the first sample from the input file and the remaining inputs from the network output (as with straight iterative prediction).

For more detailed information on iterative prediction and teacher forcing, review the NeuroSolutions documentation for the DynamicControl and BackDynamicControl components.

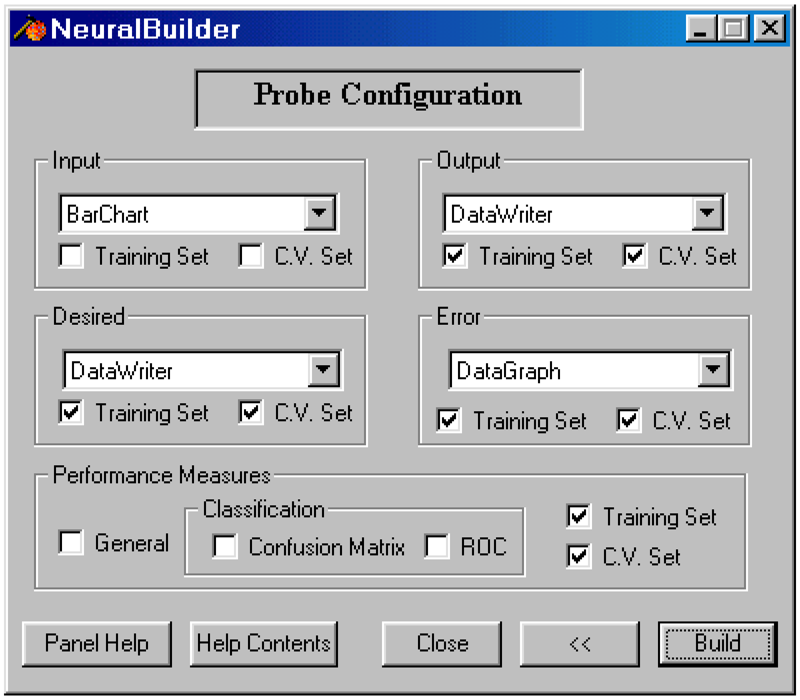
Step 6

Step 6: Probe Configuration

One of the advantages of NeuroSolutions is its extensive probing ability. Within this panel you can select five common network points that you may want to probe. At each of the selected points, you can choose the probe that is most appropriate for the data at that point (see figure below). Note that you are not restricted to these five probing points. Once you have constructed the network, you can attach additional probe components.

By default, the NeuralBuilder is configured to probe the error curve (using the DataGraph), the network output (using the DataWriter) and the desired output (using the DataWriter). The DataWriter and DataGraph have an additional feature in that it is configured to display both the network output and the desired output in the same window so that you can easily compare the values.

The bottom section of the panel contains check boxes for three MatrixViewer probes, which provide various performance measures. The General performance probe displays the Mean Squared Error (MSE), the Normalized Mean Squared Error (NMSE), the Correlation Coefficient (r), the Percent Error, the Akaike Information Criterion (AIC), and the Minimum Description Length (MDL). The Confusion Matrix probe shows the percentage of exemplars classified correctly for each output class. The Receiver Operating Characteristics (ROC) probe shows the ratio of detections and false alarms for a range of output thresholds. For more information on these probes, see the NeuroSolutions documentation under the "Confusion Matrix", "ROC", and "Performance Measures" access points of the Criterion family of components.


NeuralBuilder

Probe Configuration

Input: BarChart — [ ] Training Set [ ] C.V. Set

Output: DataWriter — [✓ Training Set] [✓ C.V. Set]

Desired: DataWriter — [✓ Training Set] [✓ C.V. Set]

Error: DataGraph — [✓ Training Set] [✓ C.V. Set]

Performance Measures: [ ] General — Classification: [ ] Confusion Matrix [ ] ROC — [✓ Training Set] [✓ C.V. Set]

Probe Configuration panel of the NeuralBuilder

The table below summarizes the probes listed within each of the pull-down menus:

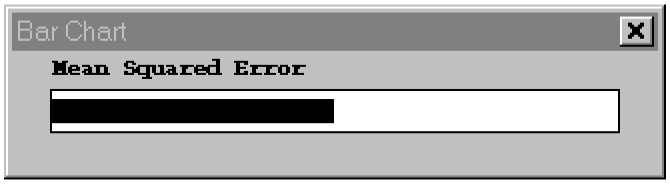
Probes

Name	Description
BarChart	qualitative (size of bars)
Hinton	qualitative (size of squares)
ImageViewer	qualitative (intensity level)
MatrixViewer	quantitative (numeric - view only)
MatrixEditor	quantitative (numeric - editable)
MegaScope	qualitative (graph over time)
DataGraph	qualitative & quantitative (graph over

	time w/ labeled axis)
DataWriter	quantitative (writes data to a file)

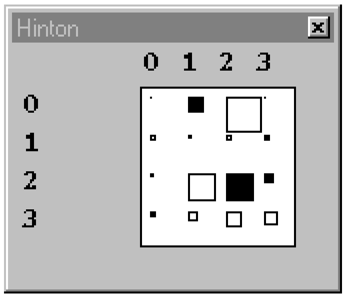
Static probes display instantaneous data, meaning that only a single time step is represented. Temporal probes display the data across a "window" of time. All probes listed above are static with the exception of the MegaScope, which is temporal. Below is a brief summary of the probes. For a complete description of the probe components, see on-line documentation for NeuroSolutions.

The *BarChart* provides qualitative information in the form of a column-oriented display of information (see figure below). It is very useful for static classification problems when making a comparison between the output and the desired response.



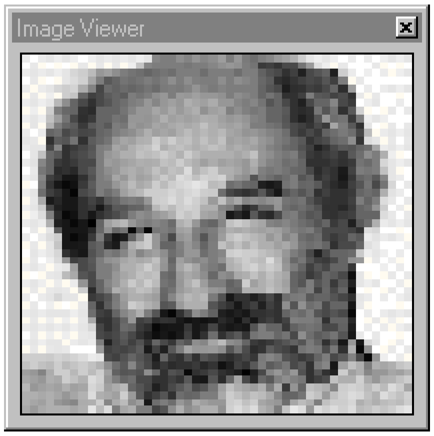
Display window of BarChart probe

The *Hinton* diagram is also a qualitative display that provides a global view of a matrix (normally the weight matrix). The values are represented by squares whose size is associated with the magnitude and whose color represents the sign (black is negative, white is positive).



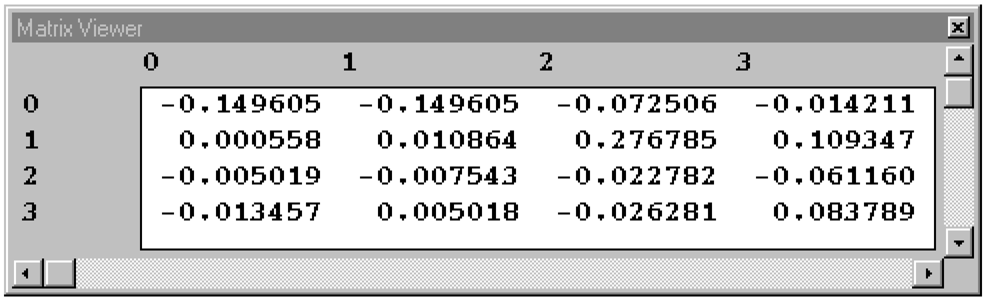
Display window of Hinton probe

The *ImageViewer* displays a matrix of values as intensity levels (white corresponds to one and black corresponds to zero). The primary purpose of this probe is to display images such as bmp files (see figure below), but it can also be used as an alternative to other qualitative probes.



Display window of ImageViewer probe

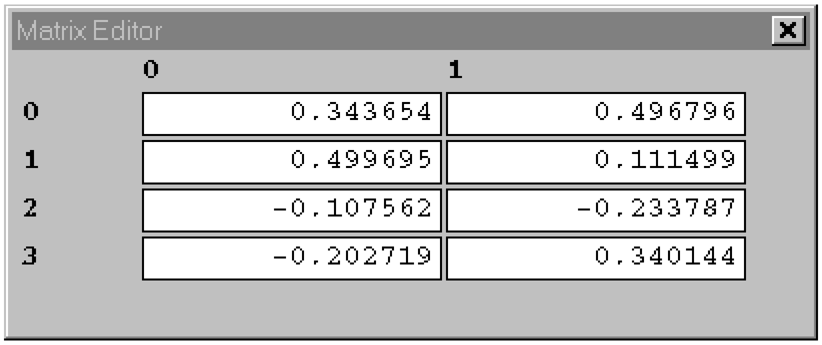
The *MatrixViewer* provides quantitative (numeric) information of the data being probed (see figure below). You can use it to obtain the value of any internal network variable.


Matrix Viewer

	0	1	2	3
0	-0.149605	-0.149605	-0.072506	-0.014211
1	0.000558	0.010864	0.276785	0.109347
2	-0.005019	-0.007543	-0.022782	-0.061160
3	-0.013457	0.005018	-0.026281	0.083789

Display window of the MatrixViewer probe

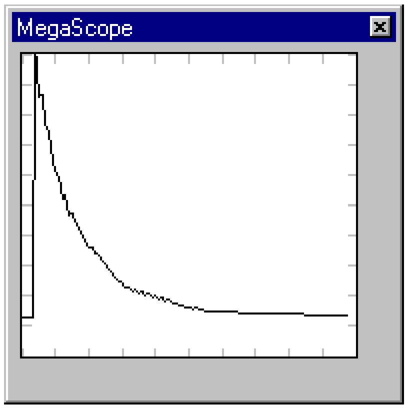
The *MatrixEditor* is similar in appearance to the MatrixViewer, but it has a very important additional function. It allows the user to modify the values of the internal network variables. For example, you may want to modify the weights or inject a specific pattern to find out the response. You simply change the values within the probe window and re-start the simulation. Note that this probe slows down the simulations significantly more than the MatrixViewer.


Matrix Editor

	0	1
0	0.343654	0.496796
1	0.499695	0.111499
2	-0.107562	-0.233787
3	-0.202719	0.340144

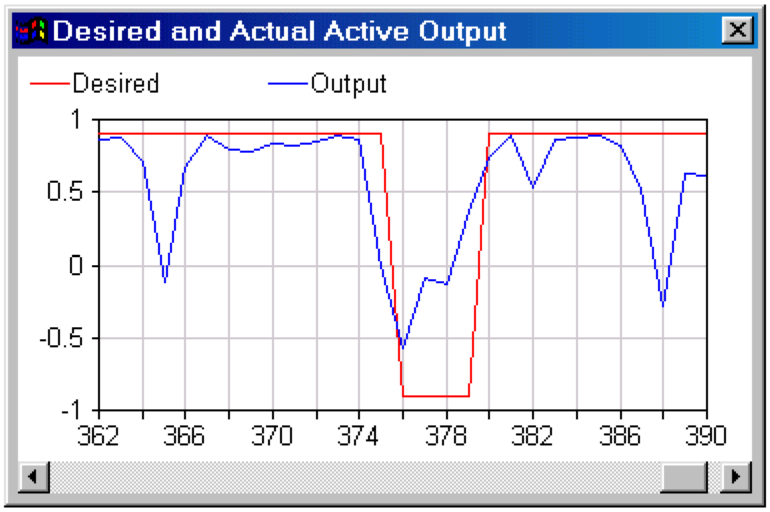
Display window of the MatrixEditor probe

The *MegaScope* is like a multichannel oscilloscope in that it displays amplitude versus time (see figure below). It is useful for displaying the learning curve (the evolution of the mean square error during learning). It is a necessity when working with temporal problems such as adaptive signal processing.



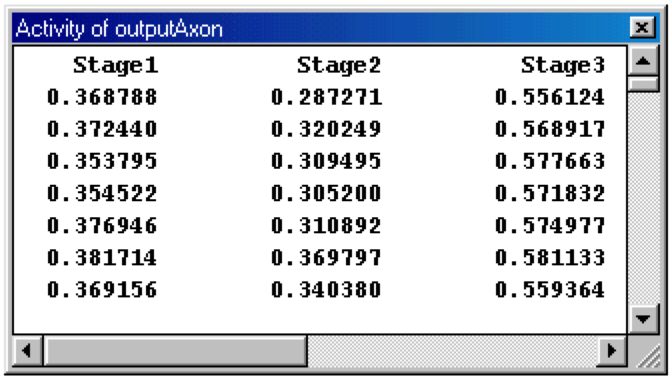
MegaScope window used to display the learning curve

The *DataGraph* is a graphing tool that displays amplitude versus time (see figure below). It is useful for displaying the learning curve (the evolution of the mean square error during learning) as well as the network output vs. desired output.



DataGraph window used to display the output vs. desired data.

The *DataWriter* provides quantitative (numeric) information of the data being probed. It differs from the MatrixViewer, in that it displays each new set of data in a new row, instead of replacing the previous data with the new data. This probe can also be used to write the probed data to a file.



Activity of outputAxon

Stage1	Stage2	Stage3
0.368788	0.287271	0.556124
0.372440	0.320249	0.568917
0.353795	0.309495	0.577663
0.354522	0.305200	0.571832
0.376946	0.310892	0.574977
0.381714	0.369797	0.581133
0.369156	0.340380	0.559364

DataWriter window used to display the network output over time.

Step 7

Step 7: Simulation

The final step is started by clicking on the Build button once you have specified the probes. The NeuralBuilder constructs the network as per your specifications. Watch as the components are brought to the breadboard one at a time, interconnected and configured. Eventually, you will learn to construct networks component by component, allowing you to build many more neural models than those supported by the NeuralBuilder. To understand the iconic representation of the neural components, refer to the on-line documentation of NeuroSolutions. Here just a brief description is presented.

Note that only the display window for the error probe(s) is displayed. You can double-click on the probe icons to open the display windows for the other probes you have selected. The probes are labeled according to the column headings extracted from the data files.

Observation of the probes, in particular the probe monitoring the mean square error, is crucial for fine tuning the learning process. You should be aware that computing cycles are consumed by the graphical animations of the probes. Only the probes that are opened will slow down the simulation. A compromise should be reached between speed of simulation and the speed of the animations. Each of the probes has a parameter for setting the refresh rate of the display. If simulation speed is important, then this rate should be set low so that the probe does not consume a high percentage of the processing cycles. See the NeuroSolutions on-line documentation for a complete description of the probe parameters.

To start the simulation, press the Start button from the Control Toolbar. Once you have trained the network and are satisfied with the performance, you should save the breadboard to a file. This will save all of the components, connections, parameters and (optionally) trained weights of the network. This trained neural network can then be loaded back into NeuroSolutions at a later time.

If the simulation was not successful, then the parameters and/or the topology itself may need adjusting. One approach is to return to the NeuralBuilder and change the settings within one or more of the panels. This may be the best approach if you are new to NeuroSolutions and the topology (e.g., the number of layers) needs to be modified. If it is simply a matter of parameter adjustment, then these changes should be made directly from the breadboard using the Inspector window. The following section contains a brief summary of the procedure for manipulating a component's parameters. Other basic concepts of the package are briefly summarized as well. See the NeuroSolutions on-line documentation for a complete discussion of these topics.

Planes of a Network

NeuroSolutions simulations can be broken down to three basic planes: the activation plane, the backpropagation plane and the gradient search plane. The activation plane is responsible for the forward activation of the network (i.e., producing an output for a given input). The backprop plane is responsible for backpropagating the error used for backpropagation learning. The gradient search plane is responsible for updating the weights based on the activation and the error at each weight.

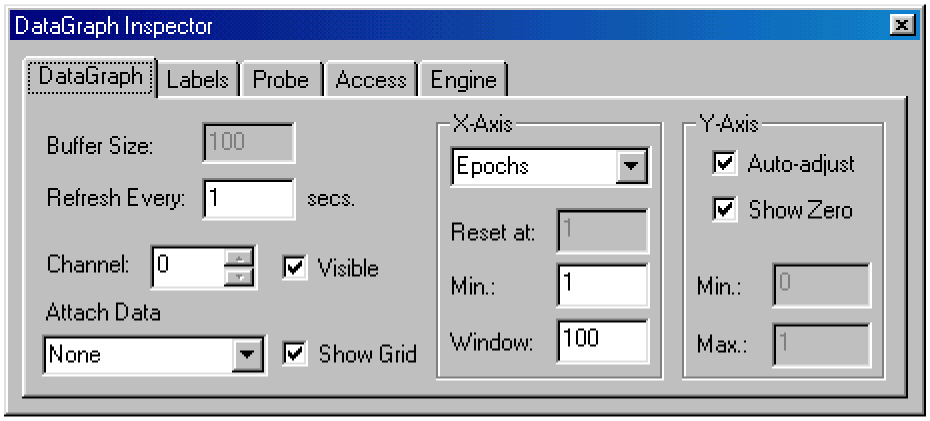
Access Points

Each component understands a common communication protocol for accessing the data of other components. Components can share their data with other components by providing one or more access points.

The most common use of access points is for probing the internal network data. By attaching a probe component to an access point of another component on the breadboard, that component’s data can then be displayed.

Inspector

The Inspector window is the interface for viewing and modifying the parameters of the components. To display the Inspector window, select the Inspector item from the View menu (at the top of the main window of NeuroSolutions).

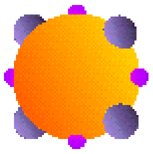


Buffer Size: 100
Refresh Every: 1 secs.
Channel: 0
Visible ✓
Attach Data: None
Show Grid ✓
X-Axis
Epochs
Reset at: 1
Min.: 1
Window: 100
Y-Axis
Auto-adjust ✓
Show Zero ✓
Min.: 0
Max.: 1

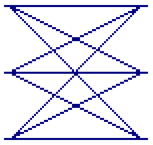
DataGraph inspector

When you select (single-click on) a component on the breadboard, the corresponding inspector is displayed within the Inspector window. The parameters are presented and can be modified by typing in new values or clicking on the controls. Not all of the parameters are visible at one time. At the top of a component's inspector are labeled tab keys. Selecting one of these keys will switch to a different "page" of parameters. Some of these pages will be common to several components. The components and their parameters are explained in detail in the on-line documentation for NeuroSolutions. This chapter will only address the parameters of the most important components.

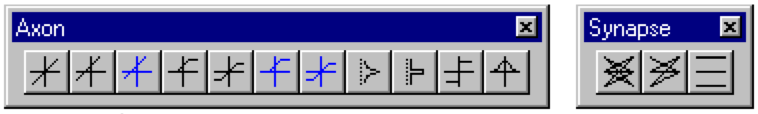
Activation Plane



The basic building blocks for the activation plane are the Axon and Synapse families. The Axon implements the nonlinearity, while the Synapse



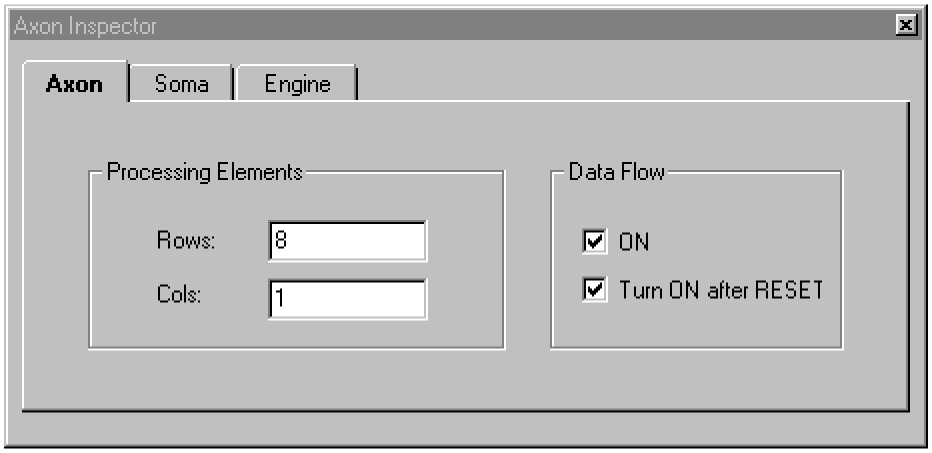
implements the sum of products of the McCulloch-and-Pitts neuron. The Axon contains the biases, while the Synapse contains all of the weights. Each axon represents a vector (layer) of PEs and each synapse represents a matrix of weights.



Axon and Synapse palettes

The nonlinearity that an axon implements is determined by the particular type of axon selected from the palette. To change this nonlinearity you would need to replace this component with a new one. One way to do this is to return to the NeuralBuilder and change the Transfer Function setting(s) within the Layer panel(s).

The number of PEs within an axon can be viewed by selecting the Axon tab key within its inspector. Edit the number in the Rows field to modify the axon's size. Note that there is also a Columns field within this inspector. There are cases when the vector of PEs is best interpreted as a matrix (e.g. an image), in which case this field would be used.



Axon Inspector

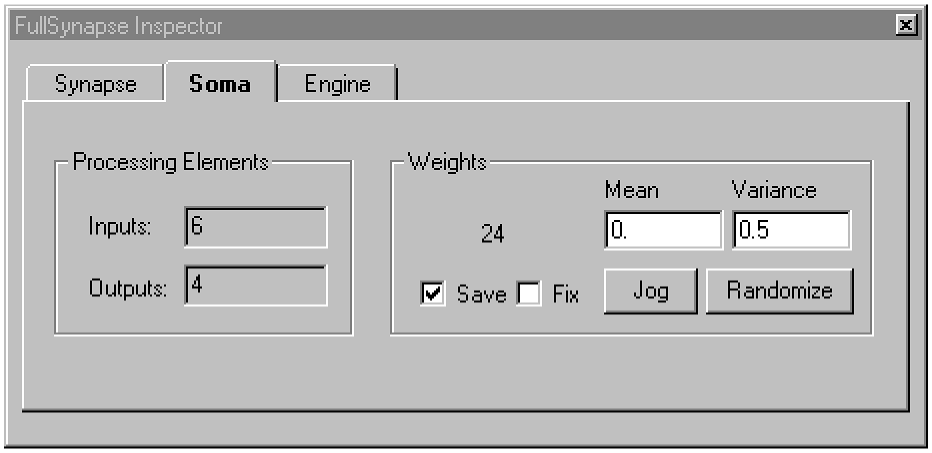
Axon | Soma | Engine

Processing Elements
Rows: 8
Cols: 1

Data Flow
☑ ON
☑ Turn ON after RESET

Axon inspector

The size of a synapse is determined by the size of the axons that it is connecting. For the standard FullSynapse component, the number of weights is the product of the number of PEs in the axon at its input with the number of PEs in the axon at its output. The number of weights can be viewed from the Soma page of the synapse's inspector.



Inputs: 6

Outputs: 4

Mean: 0.

Variance: 0.5

24

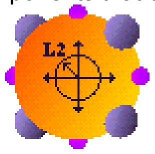
[✓ Save] Fix

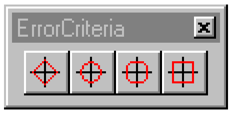
FullSynapse Inspector

Saving the Weights

The Save and Fix switches within the Soma page of the Axon and Synapse inspectors control how the weights are treated (see figure). When you save a breadboard to a file, the weights of a given component will be saved only if this switch is checked (the default).

The Fix switch protects the component weights from being modified by the global commands of the controller (i.e., Reset, Randomize and Jog). This switch is useful when you want to set the values of a component's weight matrix (using the MatrixEditor) and have those values stay fixed while the rest of the network is trained. Note that a GradientSearch component can still alter these weights.

The ErrorCriteria family consists of components that compute error used by the backpropagation plane and the gradient search plane. The L2Criterion  (accessed using the second button from the left in the figure below) is most commonly used to generate the mean squared error (MSE) of the activation plane.

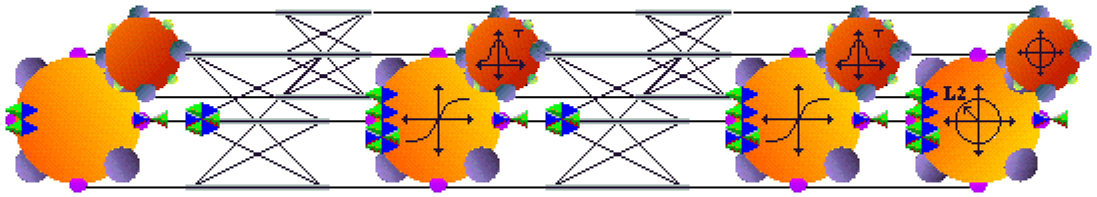


ErrorCriteria palette

The ErrorCriteria component can be configured to automatically save the weights that produced the lowest MSE during the training. These weights are not stored with the breadboard, but as a separate ASCII weight file (*.nsw). The network's controller (StaticControl or DynamicControl) is the component used to manually save the breadboard weights to a file or to load a set of stored weights into the breadboard.

Backpropagation Plane

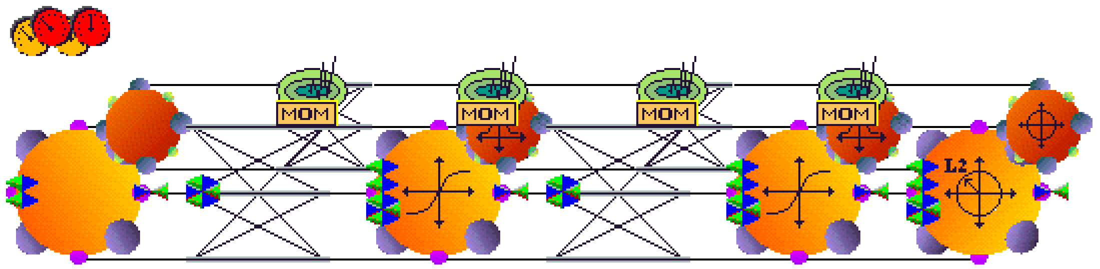
The backpropagation plane is represented by smaller, but similar, component icons that lay on top of the activation components. Normally, you will not need to select these components from the Backprop palette because they are created automatically using either the NeuralBuilder or the BackStaticControl inspector (see Changing the Learning Rule).



Activation and Backprop planes

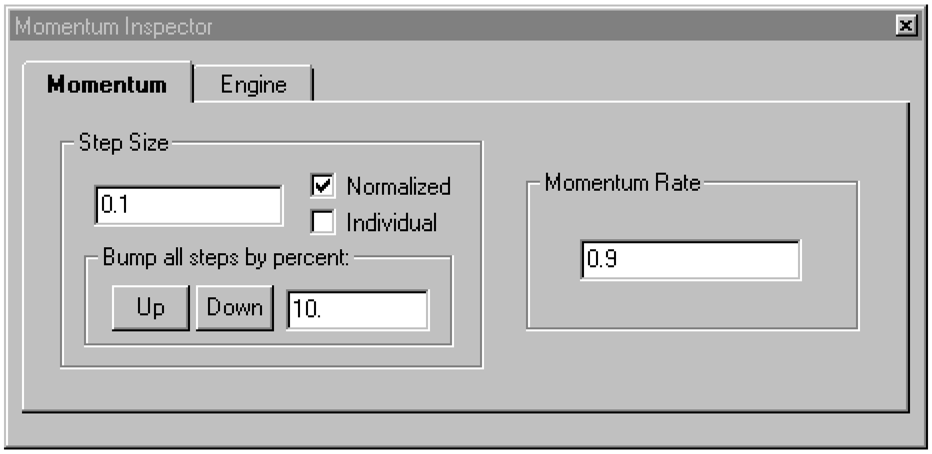
Changing the Learning Rates

On top of the backpropagation plane you will find the gradient search components. Note that only those components with adaptive coefficients (weights or biases) can have a corresponding gradient search component attached.



Activation, Backprop, and Gradient Search planes

Selecting one of the gradient search components will display that component's learning rate within the Inspector window (see figure below). These are the parameters that need changing if the learning is too slow or if the training diverges. Note that changing these parameters only affects the learning rate for the selected component. To set the same learning rate for all of the gradient search components, you must first select the group by clicking on all of the gradient search icons while holding down the Shift key. Any parameter changes made within the Inspector window are now applied to all components of the group.


Momentum Inspector

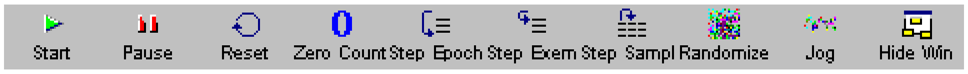
Momentum | Engine

Step Size: 0.1
[✓] Normalized
[ ] Individual
Bump all steps by percent: 10.

Momentum Rate: 0.9

Momentum inspector

Control Menu & Toolbar Commands



Description:

This toolbar allows you to perform global data flow operations on the network. Open this toolbar by selecting "Control" from the "Toolbars" menu. These control commands can also be found within the "Tools" menu.

Toolbar Commands:

Start
Begins an experiment defined by the Control component on the breadboard. If this button is disabled, then the experiment has run to completion and the network must either be reset (see below) or the epochs must be increased (see the documentation for the StaticControl inspector within the help for NeuroSolutions).

Pause
Pauses the simulation after finishing the current epoch.

Reset
Resets the experiment by resetting the epoch and exemplar counters, and randomizing the network weights. Note that when the Learning switch from the Static property page of the controller is off, the weights are not randomized when the network is reset

Zero Counters
Sets the Epoch and Exemplar counters to zero without resetting the network.

Step Epoch
Runs the network for the duration of one epoch.

Step Exemplar
Runs the network for the duration of one exemplar.

Step Sample
Runs the network for the duration of one sample.

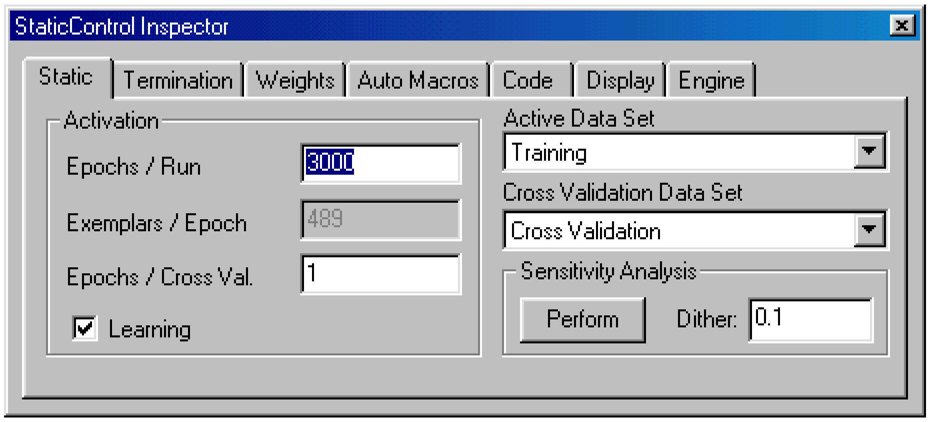
Randomize
Randomizes the network weights. The mean and variance of the randomization is defined within the Soma property page of each component that has adaptable weights.

Jog
Alters all network weights by a random value. The variance of the randomization is defined within the Soma property page of each component that has adaptable weights.

Hide Windows
When this button is selected (pressed down), all display windows are hidden from view. When the button is de-selected (popped up), the display windows are restored to their original state.

Static Simulation Control

The icons shown in the upper-left corner of the Activation, Backprop, and Gradient Search planes Figure are the controllers. Their job is to orchestrate the firing of data throughout the breadboard. The orange icon represents the controller for the forward activation plane, the StaticControl. It is used to select the number of Epochs/Run, which is the number of training set iterations (epochs) that will be performed before stopping the training. The number of Exemplars/Epoch is automatically set by the File components. This is the number of patterns (exemplars) that constitute the data set (an epoch). The number of Epochs between Cross Validation checks can also be specified using the controller.


StaticControl Inspector

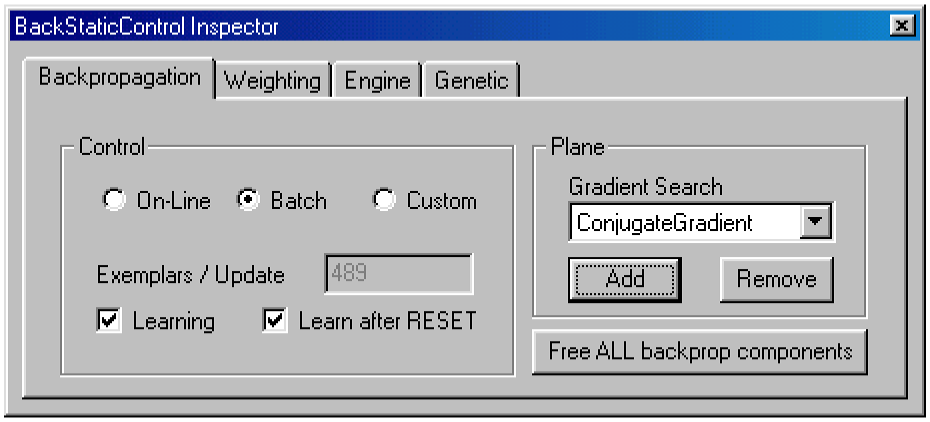
Static | Termination | Weights | Auto Macros | Code | Display | Engine

Activation
Epochs / Run: 3000
Exemplars / Epoch: 489
Epochs / Cross Val.: 1
☑ Learning

Active Data Set: Training
Cross Validation Data Set: Cross Validation
Sensitivity Analysis
Perform
Dither: 0.1

StaticControl inspector

The BackStaticControl is the base controller for the backpropagation plane. It is used to specify the type of learning, batch or on-line. On-line learning updates the weights after every exemplar (i.e., the Exemplar/Update field is set to 1). Batch learning updates the weights after the entire training set has been presented (i.e. the Exemplars/Update field is set to the number of Exemplars/Epoch specified within the StaticControl inspector).


BackStaticControl Inspector

Backpropagation | Weighting | Engine | Genetic

Control
○ On-Line ● Batch ○ Custom
Exemplars / Update: 489
☑ Learning ☑ Learn after RESET

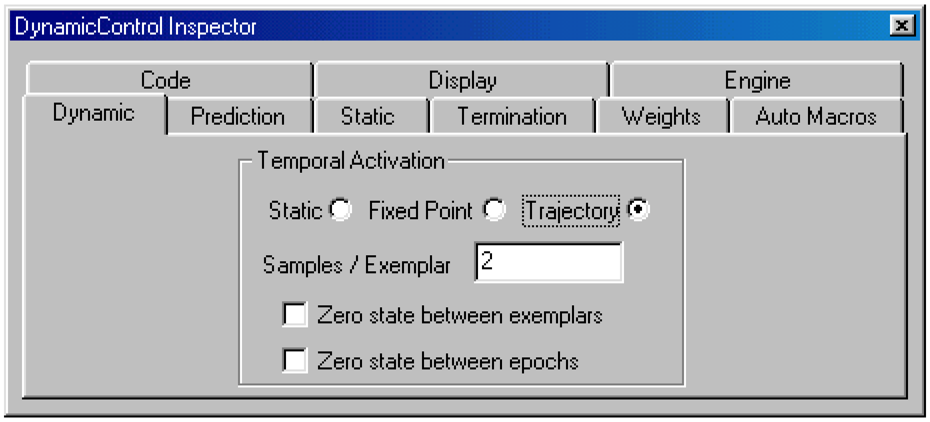
Plane
Gradient Search: ConjugateGradient
Add
Remove
Free ALL backprop components

BackStaticControl inspector

Dynamic Simulation Control

Networks used to solve temporal problems require a dynamic learning algorithm; NeuroSolutions uses the backpropagation through time (BPTT) algorithm. The controllers required for this type of learning are the DynamicControl and the BackDynamicControl. Note that these controllers are represented differently than their static counterparts (there are three dials instead of two).

The DynamicControl inspector (see figure below) has two additional parameters from its static counterpart. The three radio buttons at the top specify the type of learning: Static (standard backpropagation), Fixed-Point (recurrent backpropagation) and Trajectory (backpropagation through time). Setting the Static switch is equivalent to using the StaticControl component used by the other neural models. Recurrent backpropagation is demonstrated in the on-line documentation for NeuroSolutions and is not covered here. The default setting for the dynamic control is Trajectory learning.


DynamicControl Inspector

Code | Display | Engine
Dynamic | Prediction | Static | Termination | Weights | Auto Macros

Temporal Activation

Static ○ Fixed Point ○ Trajectory ●

Samples / Exemplar: 2

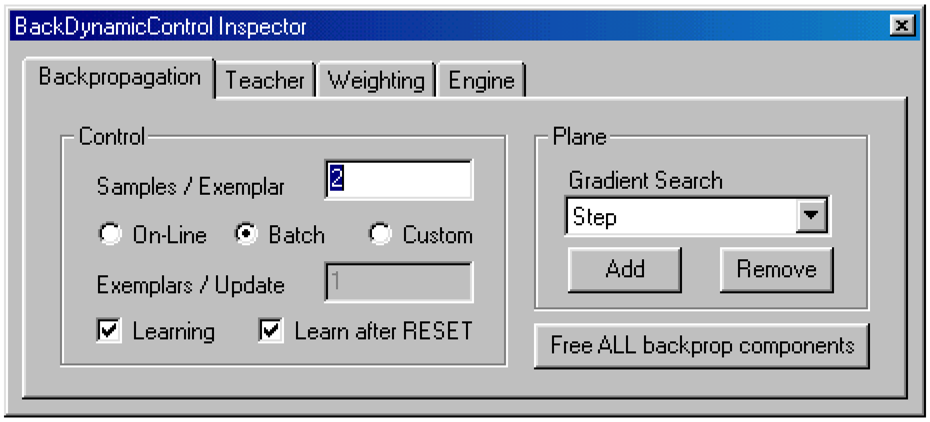
☐ Zero state between exemplars

☐ Zero state between epochs

DynamicControl inspector

The Samples/Exemplar parameter specifies how many samples of data represent one exemplar. One exemplar is now a time sequence. For example, suppose that your are trying to train a network to recognize spoken words from an audio signal. Each word might consist of 100 samples from that signal, so this parameter would be set to 100. The Exemplars/Epoch field would specify the number of words in your training set. Note, however, that the Samples/Exemplar must evenly divided into the number of Exemplars, or the simulation will not run.

Given the Samples/Exemplar defined in the DynamicControl, the BackDynamicControl inspector (see figure below) is used to specify how many of these samples are used to backpropagate the error through time. This parameter is also labeled Samples/Exemplar and must be no greater than that specified within the DynamicControl inspector. Note that if this parameter is set to 1, the learning is equivalent to static backpropagation. Normally, it should be set to the same value as for the forward DynamicControl.


BackDynamicControl Inspector

Backpropagation | Teacher | Weighting | Engine

Control

Samples / Exemplar: 2

○ On-Line ● Batch ○ Custom

Exemplars / Update: 1

☑ Learning ☑ Learn after RESET

Plane

Gradient Search: Step

Add | Remove

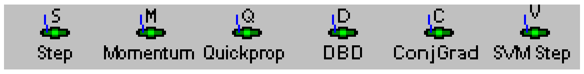
Free ALL backprop components

BackDynamicControl inspector

Changing the Learning Rule

The Momentum is the default learning rule of the NeuralBuilder. The learning rules of the layers can be changed by returning to the Layer panels and changing the appropriate pull-down menu. The learning rules can also be changed directly from the breadboard.

The figure below shows the GradientSearch palette. To change an individual learning rule, remove the old gradient search component from the breadboard and stamp a new component in its place.



Gradient Search palette

The other option is to make the change directly from the backpropagation controller. Within the Inspector for the BackStaticControl component there is a box labeled Plane. This is used to add and remove the backprop and gradient search components (the learning). Suppose that you want to change the gradient search components because the learning is too slow. Click the Remove button to clear all the components of the backprop and gradient search families from the breadboard. Choose a new gradient search procedure from the pull-down menu. Click on the Add button to re-create the backprop and gradient search planes. Note that the learning parameters will be set to the defaults and will likely need to be adjusted.

Testing the Network

Once you have trained the network so that the error is down to an acceptable level, the next step is to freeze the weights and test the network by feeding it a new data set (the test set). Assuming that you have specified a test set within the Cross Validation and Testing Data panel, you need to follow a few simple steps:

1. From the inspector of the StaticControl (or DynamicControl), switch the Active Data Set to Testing.
2. Double-click on the DataWriter probe attached to the desired File component (the default setting) to open its display window.
3. Run the network and observe the output within the DataWriter window. By default, both the network output and desired output will be displayed side-by-side.

There is also a utility called the TestingWizard, which will allow you to easily test the network, even if you did not specify a test set within the NeuralBuilder. This utility can be launched by clicking the "Testing" toolbar button of NeuroSolutions, or by selecting "TestingWizard" under the Tools menu.

Creating New Probes

Additional probes can be attached to the network using the Probes palette. Make a selection from this palette and move the cursor to the component on the breadboard that you would like to probe. Note that if the component does not accept the probe or the cursor is over the breadboard, the cursor will change to a forbidden sign. The cursor should now be represented by a stamp icon, indicating that this component will accept the attachment of the selected probe. Single-click to stamp a new probe and attach it to the component. See the on-line documentation of NeuroSolutions for complete instructions on stamping components onto the breadboard.

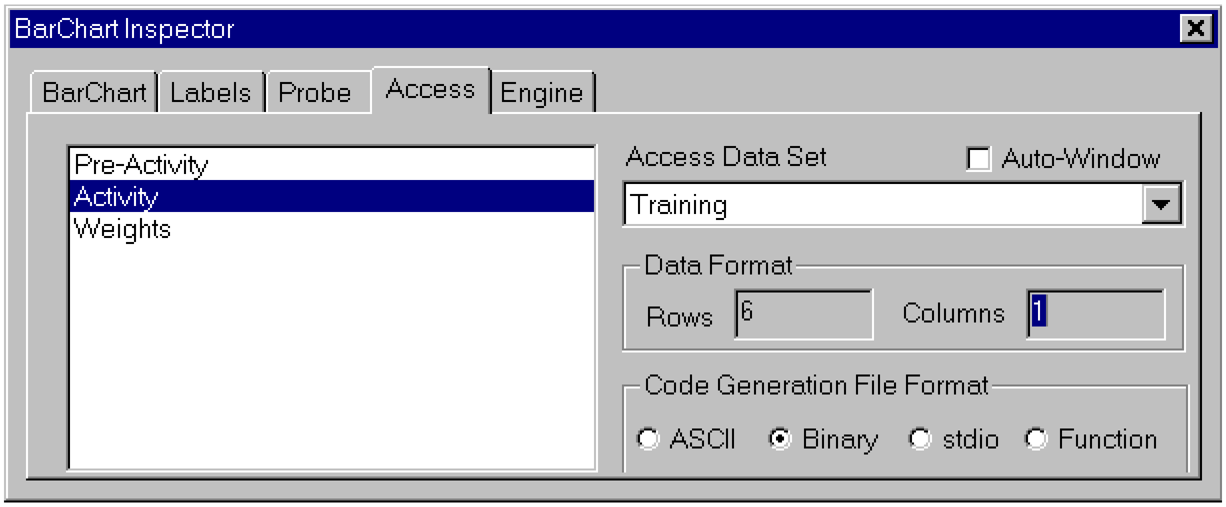
The cursor remains in stamping mode if you use the right mouse button to stamp the components. Stamping with the left mouse will return the program back to selection mode. The Selection Cursor button (the one with an arrow for an icon) from the main toolbar can also be used to set the program back to selection mode.

Now you may select the new probe on the breadboard to bring up its corresponding inspector. Double-clicking on the probe icon will display the probe's window.

Selecting the Data to Probe

In order to avoid the mistake of probing different data than what was intended, it is important to understand the difference between the various access points. Select the Access tab of the probe's inspector. Select the data set that you would like to monitor from the Access Data Set list.

The list on the left side of the page contains the access points available based on the component that the probe is attached to (see figure below). Select from this list to change the access point to attach the probe to (i.e., select what data to probe).



Access property page of a component's inspector

Below is a brief summary of the primary access points of the most common components:

Activity

- Data at the output of the component
- Commonly used with: Axon or Synapse

Pre-Activity

- Data at the input of the component
- Commonly used with: Axon

Weights

- Values of the connection matrix (synapse) or bias vector (axon).
- Commonly used with: Axon or Synapse

Cost

- Instantaneous error of the network
- Commonly used with: ErrorCriteria

Average Cost

- Error of the network, averaged since the last weight update
- Commonly used with: ErrorCriteria

Stacked Access

- Same data that the component below is attached to
- Commonly used with: File or another Probe

The Data Display panel of the NeuralBuilder provides six options for probe placement. Below is a summary of these most common access points.

Input

- Component: Input Axon (left-most)
- Access Point: Pre-Activity

Output

- Component: Output Axon (left of error criteria)
- Access Point: Activity

Desired

- Component: Desired File (right-most)
- Access Point: Stacked Access

Error

- Component: ErrorCriteria
- Access Point: Average Cost

Bias

- Component: Axon (specified by Layer field)
- Access Point: Weights

Weights

- Component: Synapse (specified by Layer field)
- Access Point: Weights

Static Probes

Static probes display instantaneous data, meaning that only a single time step is represented. Each probe has a set of unique parameters, which are specific to the way in which it displays the data (e.g., colors, sizes and data normalization).

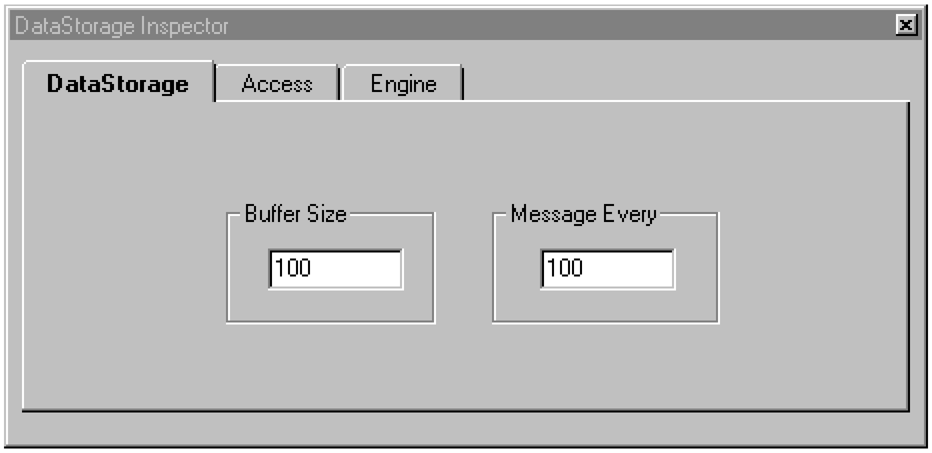
The static probes contain a Display Every field within their inspector. This parameter is very important because it specifies the refresh rate of the probe's display. This value is set to 1 by default, meaning that the display is updated after every sample. The problem is that this consumes a lot of processing cycles that could otherwise be used for the network training. The NeuralBuilder closes all of the probes except those attached to the Average Cost in order to minimize this overhead.

One way to reduce overhead of the probe display is to set the Display Every field to the number of Exemplars/Epoch plus one. This way, the display is refreshed after every epoch instead of every exemplar. The "plus one" term is added so that the same exemplar is not displayed over and over again. For example, a training set with 4 exemplars is presented to the network four times (4 epochs for a total of 16 presentations). A probe is attached and set to display after every 5 exemplars. The probe displays the 1st, 6th, 11th and 16th presentations. This corresponds to the 1st, 2nd, 3rd and 4th exemplars of the training set.

Temporal Probes

The temporal probes such as the MegaScope and XYScatterPlot are a little more complex because they handle data over time. Temporal probes do not attach to the access points of the breadboard directly as the static probes do. Temporal probes can only attach to a DataStorage component (or another component stacked on top of a DataStorage). The DataStorage component (the barrel icon) can attach to any component that a static probe can attach to.

The DataStorage provides a circular buffer used to store data across a window in time. Its inspector (see figure below) contains the parameters for both the size of the buffer (the length of the window) and how often the attached component (i.e., the temporal probe) updates its data (i.e., re-displays). This second parameter has the same functionality as the Display Every field of the static probe.


DataStorage Inspector

DataStorage | Access | Engine

Buffer Size: 100

Message Every: 100

DataStorage inspector

The MegaScope has many more parameters than any of the static probes. Double-click the MegaScope icon to open its display window and show its inspector. The MegaScope Sweep is used to specify the scale of the horizontal axis. The higher the Samples/Division, the higher the resolution and the more samples displayed at once. The MegaScope inspector page provides controls for the vertical scale and horizontal and vertical offsets. These can be set to control individual channels or all channels. The Autoset Channels button attempts to adjust these selections based on the current data. The MegaScope window can be scrolled and/or resized to display all of the data stored in the circular buffer.

Note that the DataGraph component can do most of what the MegaScope can do, and it is much easier to configure. In addition, it does not require the DataStorage component.

Saving Data to Files

The DataWriter probe can be used to create ASCII or binary files that contain the data that passes through the access point of the attached component. For example, you may want to store the history of the error over the course of a simulation. Place a DataWriter on the Average Cost access point of the ErrorCriteria component and set the Dump Raw Data to File switch from its inspector. This will open a file selection window to specify the file name to save to. Once the simulation is run, the error data will be dumped to this file.

Neural Models

To exemplify the neural models supported by the NeuralBuilder, each section contains configuration instructions for the same real world problem—sleep staging. See A Prototype Problem for an introduction to this problem and its corresponding data. These examples are indented within each section to distinguish them from the reference text.

A lot can be learned about the problem solving capabilities of a given model when its performance is compared with other similar models. Each will need to be configured differently and some will solve the problem more efficiently than others. The overall goal is to allow you, the user, to pick the best model for a given problem. There are no hard and fast rules for making this selection; it usually requires experience and some trial and error.

A Prototype Problem

Sleep staging is a quantitative measure to evaluate sleep. Sleep disorders are becoming quite common, probably due to the stress of modern living. Sleep is not a uniform process. The brain goes through well-defined patterns of activity that have been catalogued by researchers. Insomnia is a disruption of this normal pattern, and can be diagnosed by analyzing sleep patterns. Normally these patterns are divided into five stages plus awake (sleep stage 0). Sleep staging is a time consuming and extremely expensive task, because the expert must score every minute of a multichannel tracing (1,200 feet of paper) recorded during the whole night. For these reasons, there is great interest in automating this procedure.

In order to score sleep automatically, it is necessary to measure specific waveforms in the brain (alpha, beta, sigma spindles, delta, theta waves) along with additional indicators (two rapid eye movements -- REM 1 and 2, and muscle artifact).

This example illustrates the type of problem that is best solved by a neural network. After much training the human expert is able to classify the sleep stages based on the input signals, but it would be impossible for that person to come up with an algorithm to automate the process. A neural network is able to perform such a classification by extracting information from the data, without any prior knowledge.

The table below shows a segment of the brain wave sensor data. The first column contains the time, in minutes, of each reading, and the next eight columns contain the eight sensor readings. The last column is the sleep stage, scored by the sleep researcher, for each minute of the experiment.

Sleep Staging Data

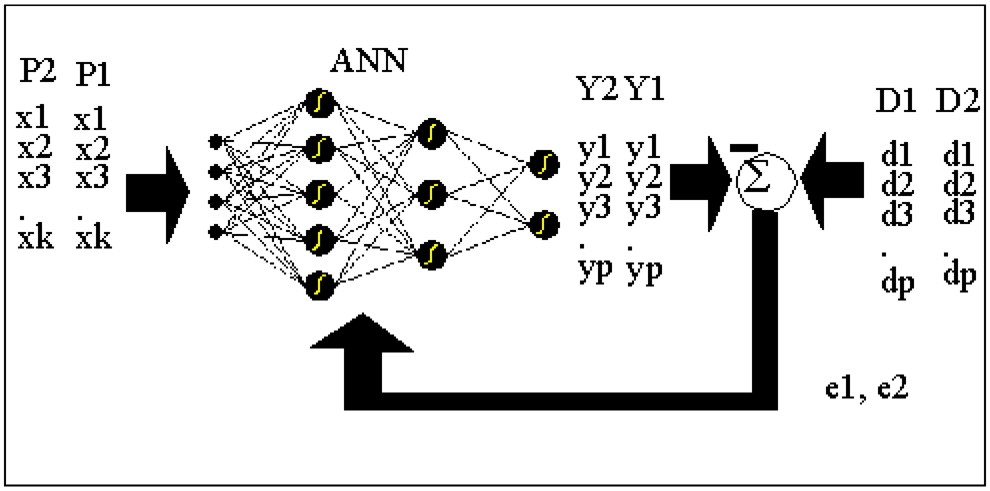
Min	α	β	Δ	δ	θ	MA	REM 1	REM 2	Stage
37	0	0	25	1	16	0	6	0	1
38	0	0	25	2	14	0	1	0	1
39	0	0	29	3	13	0	4	0	2
40	0	0	29	1	8	0	5	0	1
41	0	0	32	2	8	0	2	0	0
42	0	0	29	1	8	0	1	1	1

The problem is to find the best mapping from the input patterns (the eight sensors) to the desired response (one of six sleep stages). The neural network will produce from each set of inputs a set of outputs. Given a random set of initial weights, the outputs of the network will be very different from the desired classifications. As the network is trained, the weights of the system are continually adjusted to incrementally reduce the difference between the output of the system and the desired response. This difference is referred to as the error and can be measured in different ways. The most common measurement is the mean squared error (MSE). The MSE is the sum of the squares of the difference between each output PE and the true sleep stage (desired output).

This simple example illustrates the basic ingredients required in neural computation. The network requires input data and a desire response to each input. The more data presented to the network, the better its performance will be. Neural networks take this input-output data, apply a learning rule and extract information from the data. Unlike other technologies that try to model the problem, ANNs learn from the input data and the error (See figure below). The network tries to adjust the weights to minimize the error. Therefore, the weights embody all of the information extracted during learning.

Essential to this learning process is the repeated presentation of the input-output patterns. If the weights change too fast, the conditions previously learned will be rapidly forgotten. If the weights change too slowly, it will take a long time to learn complicated input-output relations. The rate of learning is problem dependent and must be judiciously chosen.

Each PE in the ANN will simply produce a nonlinear weighted sum of inputs. A good network output (i.e. a response with small error) is the right combinations of each individual PE response. Learning seeks to find this combination. In so doing, the network is discovering patterns in the input data that can solve the problem.



Inputs, outputs and desired response of an MLP

It is interesting that these basic principles are very similar to the ones used by biological intelligence. Information is gained and structured from experience, without explicit formulation. This is one of the exciting aspects of neural computation. These are probably the same principles utilized by evolution to construct intelligent beings. Like biological systems, ANNs can solve difficult problems that are not mathematically formulated. The systematic application of the learning rule guides the system to find the best possible solution.

Multilayer Perceptrons

Multilayer perceptrons (MLPs) are feedforward neural networks trained with the standard backpropagation algorithm. They are supervised networks so they require a desired response to be trained. They learn how to transform input data into a desired response, so they are widely used for pattern classification. Most neural network applications involve MLPs.

Advantages

MLPs are very powerful pattern classifiers. With one or two hidden layers they can approximate virtually any input-output map. They have been shown to approximate the performance of optimal statistical classifiers in difficult problems. They efficiently use the information contained in the input data.

Disadvantages

MLPs are static classifiers; i.e., the input-output map depends only on the present input. If we want to process temporal data, each time sample has to be fed to a different input, requiring very large networks.

They need lots of input data. As a general rule of thumb, there should be at least 3 times more exemplars than network weights (free parameters). Training can be slow, and setting the parameters can be tricky for difficult problems.

Configuration

The configuration of any neural topology within the NeuralBuilder requires a seven step procedure. All of the steps have already been explained above and the MLP does not have any unique parameters.

MLP Example
MLP Hints
MLP Theoretical Summary

MLP Example

From the Utilities menu select NeuralBuilder. From the NeuralBuilder window select Multilayer Perceptron. Note that the text below the selection box provides a description of the selected neural model. The text to the right provides a description of this first panel. Each panel of the NeuralBuilder has a similar text box.

Click the forward button ⇨ to switch to the Training Data panel. This panel is used to select the file containing the input data. This same file may also contain the desired output data. Click the Browse button to display the file Open panel. Find the NeuroSolutions directory and select the file sleep2.asc.

The Training Data Panel figure shows the NeuralBuilder after making this file selection. Note that there are eleven columns in the file, all of them tagged as Input. See A Prototype Problem for a description of the file's contents. The first column and last two columns will not be used. Select the corresponding items (min, K-Comp, and Score) and press the skip button. The remaining eight columns are the eight brain wave sensors described earlier.

For this example, the desired signal is contained within a separate file. Click on the forward button of the NeuralBuilder to display the Desired Response panel. Click the Browse button and select the file sleep2t.asc. Notice that all of the columns are tagged as Desired. These outputs correspond to the six stages of sleep scored by the human expert. The MLP now has the input-output pairs needed to score sleep.

Click on the forward button again. The Network Analysis panel is used for cross validation and sensitivity analysis of the training. This panel will be used later on in this experiment. Click on the forward button again to bypass this panel.

The Multilayer Perceptron panel is used to set the parameters that are specific to this neural model. The number of inputs, outputs and exemplars are computed from the input data files. The only parameter to set for the MLP is the number of hidden layers. This can be left at the default of 1. More hidden layers will also solve the problem, but at the expense of longer training times and less generalization.

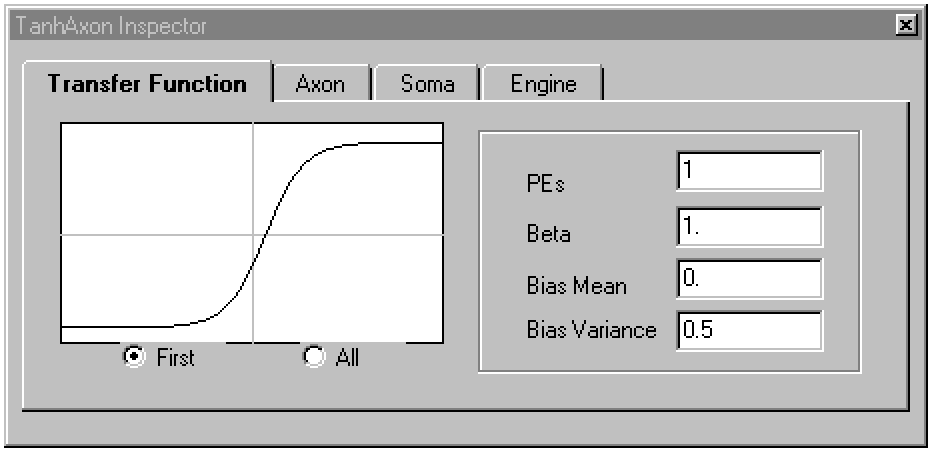
Click on the forward button to display the Hidden Layer #1 panel. This panel is used to specify the number of processing elements (PEs), the type of nonlinearity, the type of learning rule, and the learning parameters of the first hidden layer. By leaving the selections at their defaults, the network will have 16 hidden PEs with a hyperbolic tangent nonlinearity. The training will use momentum learning with a Step Size of 1.0 and Momentum of 0.7. Note that the step size is normalized (i.e., divided by the number of exemplars/update). Change the number of Processing Elements to 8 and switch to the next panel.

The Output Layer panel is the same as the previous panel except that the number of PEs is fixed to the number of outputs (6). Note that the default Step Size is one magnitude smaller (set to 0.1) than the previous layer. This is because the error attenuates as it is backpropagated through the network. Since the error is largest towards the output of the network, the output layer requires a smaller step size than that of the hidden layer in order to balance the weight updates. Click on the forward button.

Next is the Supervised Learning panel. Leave the Maximum Epochs (number of training iterations) set to 1,000. However, the network may learn the problem (i.e., have a small error) in fewer iterations that this. The default configuration will terminate the training when the error falls below 0.01. The default method for updating the weights is Batch learning. Leave the settings at the defaults and switch to the next panel.

The Probe Configuration panel is used to select the probes to attach to the network. By default, there are three BarChart probes attached to the input, output and desired response and a MatrixViewer used to display the error. Once again, these settings can be left at the defaults. Notice that instead of the forward button there is a Build button. Click on this button to construct the network (see figure).

It is instructive to compare the specifications that you entered in the panels with the final network constructed by the NeuralBuilder. Note that there is only one hidden layer (the second axon from the left). Clicking on this TanhAxon , the inspector window displays the number of PEs and nonlinearity type specified from the NeuralBuilder (see figure below).


TanhAxon Inspector

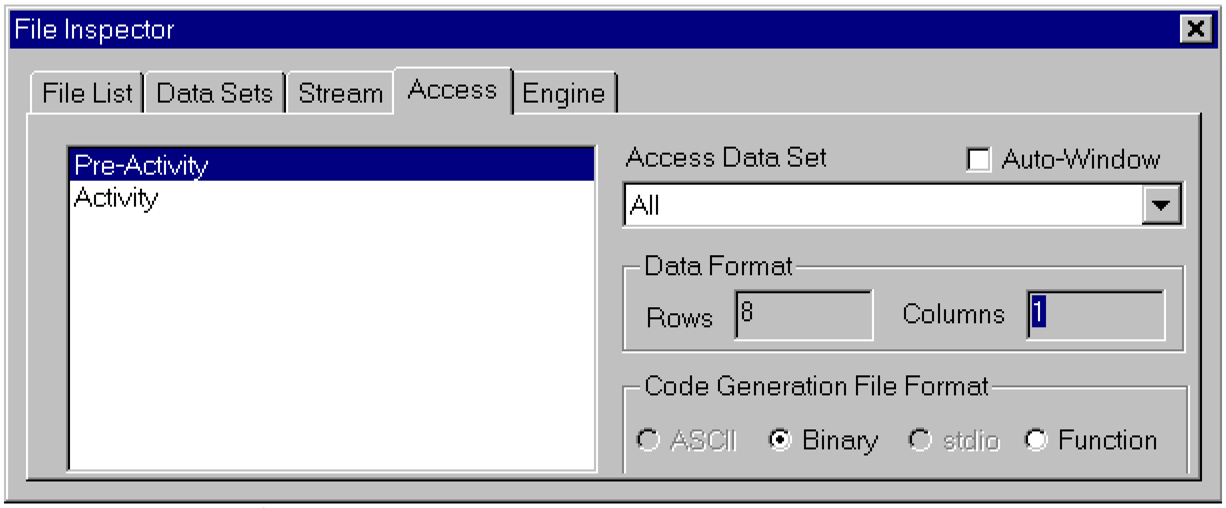
PEs: 1
Beta: 1.
Bias Mean: 0.
Bias Variance: 0.5
(● First ○ All)

TanhAxon Inspector

Click on the Momentum components to verify the learning rate settings. Note the difference in step sizes between the components attached to the hidden layer and those attached to the output layer. Click on the Axon component

at the input to verify that it has 8 PEs, corresponding to the 8 brain wave signals. Click on the File component

at the input and observe its inspector. It displays the file name, the type of file (column-formatted ASCII) and the data set. Click on the Stream tab button to display the details about the data read from the file and placed onto the data stream. Click on the Access tab button to observe that the data from the input file is being injected into the Pre-Activity access point of the Axon (see figure below).


File Inspector

Pre-Activity
Activity

Access Data Set: All
Auto-Window: ☐
Data Format
Rows: 8
Columns: 1
Code Generation File Format: ○ ASCII [✓ Binary] ○ stdio ○ Function

Access property page of the File inspector

To the right of the TanhAxon at the output is a L2criterion , which is used to compute the mean square error (MSE). Attached to this component are a File, a MatrixViewer

and a ThresholdTransmitter

. The File component is attached at the Desired Response access point and contains the desired output data (sleep2t.asc). The difference between the output and the desired output is used to compute the error of the system. Note that the number of samples in this file are the same as the input file, such that each minute of sensor data has a corresponding sleep stage score.

The ThresholdTransmitter is used to terminate training once the MSE has dropped below the specified threshold (0.01). This component is very general and can be used to send a number of messages to one or more components when the variable being monitored crosses the specified threshold. In this case, the variable is the mean squared error located at the Average Cost access point of the L2Criterion. The recipient of this message is specified under the Transmitter property page. Select the StaticControl from the Receivers List. Note that the Actions List indicates that there is a connection made to the Stop Network action (message). In summary, when the MSE drops below 0.01, the ThresholdTransmitter sends a message to the StaticControl component to Stop the network.

The MatrixViewer component is a probe used to display the numerical value of the error. It is stacked on top of the ThresholdTransmitter (using the Stacked access point), so it is monitoring the same value (the Average Cost).

The Control Toolbar is used to initiate and terminate the training. Click on the Start button to begin the simulation. Observe the data displayed in the bar charts. Note that every time the display changes, an entire epoch (397 samples) has been presented to the network. This is because the probes are configured to update their displays after every 398 samples. This is done so that a different input—output pair is displayed after every epoch (instead of showing the first one over and over).

Notice that the lengths of the bars displayed for the desired and the output are very different in the beginning. The error starts out large, but decreases rather quickly. After 100 iterations the error should be down to about 0.07. You can run the simulations for another 50 epochs, but the error will only drop by about 0.01. To verify that the network has learned the task, single-step through the data by clicking the Exemplar button from the StaticControl inspector. This will fire one exemplar at a time through the network. Notice that the BarChart at the output and the desired response BarChart coincide most of the time.

Should you continue training? With the normalization of -/+1, one error in 397 corresponds roughly to a MSE of 0.005. So this means you have about 10 misclassified minutes out of the entire night's worth of sleep data. You may want to try to improve this figure by training longer (another 500 epochs), but this is probably a reasonable place to stop the training. The real concern is to determine how well the network performs with data that it has not yet seen. For this you will need to specify a cross validation set.

Return to the Cross Validation panel of the NeuralBuilder and turn on the MSE switch. Select the file sleep1.asc as the cross validation file at the Input and sleep1t.asc as the Desired cross validation file. This data set contains sleep data from a different subject and will be used to determine how generalized the training from the first subject is. Build the network again.

Notice that the NeuralBuilder built the same network, but now there are twice as many probes as before. Open the inspector for each of the File components and observe that there are now two data sets defined instead of one. Press the Start button (from the Control toolbar) to restart the simulation. The network is being trained on the files sleep2.asc and sleep2t.asc, as before. After every epoch of training, the cross validation data is fired through the network (without updating the weights) and the results are displayed within the cross validation probes. You should observe that the error of the training set drops much more quickly than that of the cross validation set.

It is important to note that the error in the cross validation set may start to increase, even though the training set error is still going down. The explanation for this is that the network has begun to overtrain. The best value for the MSE of the cross validation set should be around 0.27 after about 150 epochs. Note that the training set error is much lower (around 0.07). After this point the training set error continues to decrease, but the cross validation set error will monotonically increase. This means that the gains in the training set learning are due to fine running of the training set data, resulting in a poorer generalization of the problem.

It is not a surprise that the cross validation set error is much higher than the training set error. This is simply an indication that sleep tokens (input signals and corresponding sleep stages) from one individual are not enough to generalize the population. The answer is to prepare a larger training set with the sleep tokens of several individuals.

MLP Hints

Start with a small network with one or even no hidden layers. Training time increases exponentially with the size of the network, so always keep things simple. More hidden layers slow the training due to an attenuation of the backpropagation error as it goes through each layer.

Momentum learning is the recommended method for training. Faster methods such as the Delta-Bar-Delta or Quickprop have extra parameters to be controlled and require extra knowledge and practice.

The learning curve (the mean squared error across time) provides a good gauge of the training progression and can be used as a basis for parameter adjustments. Observe the learning curve during learning by attaching a MegaScope at the error criteria component. If the curve is almost flat, try increasing the momentum and/or step size. If you observe that the error oscillates (repeatedly rises and falls) then it is likely that the momentum is set too high. If the network blows up (i.e., the error continually increases to a large value) then it is likely that the step size is set too high. Note that if this happens, you will need to Reset the network (from the StaticControl panel) to restore the weights to small values.

The other basic parameters of a MLP are the number of PEs and the number of layers. If adjusting these parameters do not produce an acceptable error level, then you may need to base your topology on a different neural model. Below are some suggestions of things to try first.

- Normalize your training data.
- Use the tanh nonlinearity (TanhAxon) instead of the logistic function (SigmoidAxon).
- Normalize the desired signal to be just below the output nonlinearity rail voltages (e.g., For the TanhAxon, use desired signals of +/- 0.9 instead of +/- 1).
- Set the step size higher towards the input. For example, for a one hidden layer MLP set the step size to 0.05 in the synapse between the input and hidden layer, and 0.01 in the synapse between the hidden and output layer.
- Use a more sophisticated learning method (e.g., quickprop or delta bar delta).
- Increase the ratio of training patterns over weights. You can expect the performance of your MLP in the cross validation set to be limited by the relation $N>W/\varepsilon$, where N is the number of training epochs, W the number of weights and ε the performance error. You should train until the mean square error is less than $\varepsilon/2$.

MLP Theoretical Summary

Multilayer perceptrons are an extension of Rosenblatt's perceptron, a device that was invented in the '50s for optical character recognition. The perceptron only had an input and an output layer (each with multiple processing elements). It was shown that the perceptron would only solve pattern recognition problems where the classes could be separated by hyperplanes (an extension of a plane for more than two dimensions). A lot of problems in practice do not fit this description. Multilayer perceptrons (MLPs) extend the perceptron with hidden layers, i.e., layers of processing elements that are not connected to the external world.

There are two important characteristics of the multilayer perceptron. First, its processing elements (PEs) are nonlinear. The nonlinearity function must be smooth (the logistic function and the hyperbolic tangent are the most widely utilized). Second, they are massively (fully) interconnected such that any element of a given layer feeds all the elements of the next layer.

The perceptron and the multilayer perceptron are trained with error correction learning, which means that the desired response for the system must be known. This is normally the case with pattern recognition. Error correction learning works in the following way: From the system response at PE i at iteration n, , and the desired response for a given input pattern an instantaneous error is defined by:

$$e_i(n) = d_i(n) - y_i(n)$$

Using the theory of gradient descent learning, each weight in the network can be adapted by correcting the present value of the weight with a term that is proportional to the present input at the weight and the present error at the weight:

$$w_{ij}(n+1) = w_{ij}(n) + \eta\,\delta_i(n)x_j(n)$$

The local error can be directly computed from ei(n) at the output PE or can be computed as a weighted sum of errors at the internal PEs. The constant h is called the step size. This procedure is called the backpropagation algorithm.

Backpropagation computes the sensitivity of the output with respect to each weight in the network, and modifies each weight by a value that is proportional to the sensitivity. The beauty of the procedure is that it can be implemented with local information and is efficient because it requires just a few multiplications per weight. However, since it is a gradient descent procedure and only uses the local information, it can get caught in a local minimum. Moreover, the procedure is a little noisy since we are using a poor estimate of the gradient, so the convergence can be slow.

Momentum learning is an improvement to the straight gradient descent in the sense that a memory term (the past increment to the weight) is utilized to speed up and stabilize convergence. In momentum learning the equation to update the weights becomes:

$$w_{ij}(n+1) = w_{ij}(n) + \eta\,\delta_i(n)x_j(n) + \alpha(w_{ij}(n) - w_{ij}(n-1))$$

where a is the momentum. Normally a should be set between 0.1 and 0.9.

The training can be implemented in two ways: Either we present a pattern and update the weights (on-line learning); or we present all the patterns in the input file (an epoch), store the weight update for each pattern, and then update the weights with the average weight update (batch learning). They are equivalent theoretically, but the former sometimes has advantages in tough problems (ones with many similar input-output pairs).

To start backpropagation, an initial value needs to be loaded for each weight (normally a small random value), and proceed until some stopping criteria is met. The three most common criteria are: The number of iterations, the mean square error of the training set, and the mean squared error of the cross validation set. Cross validation is the more powerful of the three since it stops the training at the point of optimal generalization (i.e., the error of the cross validation set is minimized). To implement cross validation, one must put aside a small part of the training data (10% is recommended) and use it to determine how well the trained network is learning. When the performance starts to degrade in the cross validation set, the training should be stopped.

Generalized Feedforward Networks

Generalized feedforward nets are a special case of multilayer perceptrons such that connections can jump over one or more layers.

Advantages

In theory, a MLP can solve any problem that a generalized feedforward network can solve. In practice, however, the generalized feedforward networks often solve the problem much more efficiently. A classic example of this is the two spiral problem. Without describing the problem, it suffices to say that a standard MLP requires hundreds of times more epochs of training than the generalized feedforward (for the same size network). The advantage of the generalized FF network is in the ability to project activities forward by bypassing layers. The result is that the training of the layers closer to the input become much more efficient.

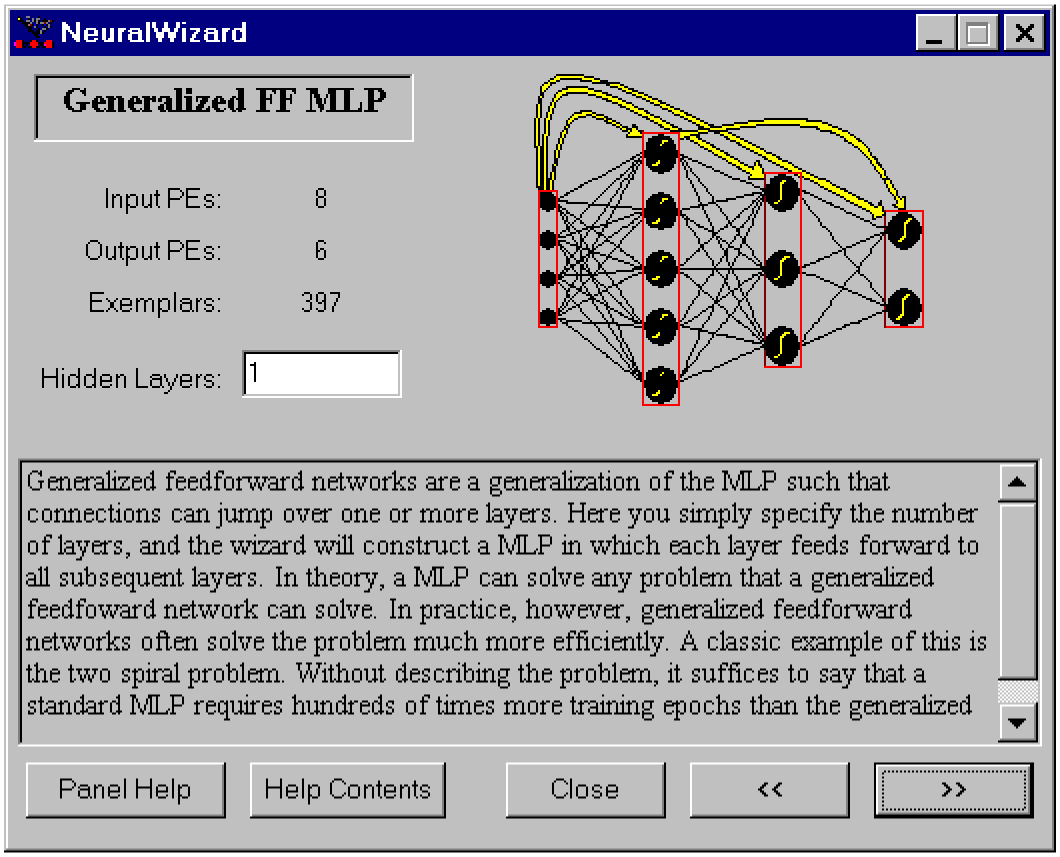
Disadvantages

The same disadvantages of the MLP apply to the Generalized Feedforward networks.

Alternative Topologies

Multilayer Perceptrons and Modular Feedforward Networks.

Configuration



Input PEs: 8

Output PEs: 6

Exemplars: 397

Hidden Layers: 1

Generalized feedforward networks are a generalization of the MLP such that connections can jump over one or more layers. Here you simply specify the number of layers, and the wizard will construct a MLP in which each layer feeds forward to all subsequent layers. In theory, a MLP can solve any problem that a generalized feedfoward network can solve. In practice, however, generalized feedforward networks often solve the problem much more efficiently. A classic example of this is the two spiral problem. Without describing the problem, it suffices to say that a standard MLP requires hundreds of times more training epochs than the generalized

Generalized FF MLP panel of the NeuralBuilder

The Generalized Feedforward nets are an extension of the MLP in the sense that signal paths from the input can cross over hidden layers. Hence, the only choice to make is the number of hidden layers.

Generalized Feedforward Example

Generalized Feedforward Hints
Generalized Feedforward Theoretical Summary

Generalized Feedforward Example

From the first panel of the NeuralBuilder select the Generalized Feed Forward model and click on the forward button ⇨. From the Training Data panel, select the file sleep2.asc and skip channels Min, K-Comp, and Score. Switch to the Desired Response panel and select the file sleep2t.asc. From the Testing Data panel, select the files sleep1.asc and sleep1t.asc as your Input and Desired cross validation set. Switching to the Generalized FF MLP panel, observe that the number of hidden layers defaults to 1. Switch to the Hidden Layer #1 panel and set the number of Processing Elements to 8, leave the PE Transfer function set to the hyperbolic tangent (TanhAxon), leave the Step Size at the default (0.8) and set the Momentum to 0.7. Configure the Output Layer to have the same Momentum. Switch to the Supervised Learning Control panel. Keep the Maximum Epochs at 1,000 and set the MSE termination so that the network stops when the Test Set error Increases by more than 0. Note that this threshold uses a smoothing filter to prevent the network from stopping prematurely due to temporary oscillations in the error. This is highly dependent on the learning rates, however, and it is still possible for a network to stop too early. If this happens, try running the network again and/or lowering the learning rates.

Switch to the next panel, keep the default probes and Build the network. Observing the constructed network, note that there is an extra connection between the input layer and the output layer. This increases the number of weights in the network. Select the FullSynapse ⇨ at the top and verify that it has 48 weights (from the Soma property page of the inspector window). Note that the learning rate for the Momentum component attached to these weights is the same as that attached to the FullSynapse at the output layer.

Run the network by clicking on the Start button of the Control toolbar. The learning is not as stable as for the MLP (the MSE oscillates), but notice that the error goes down about as fast. After 100 epochs the error is about 0.1 and the cross validation error is about 0.3. The simulation should terminate after about 150-200 epochs. Just before termination, the training error was still decreasing but the cross validation error began to rise. This is the point when the best generalization has been obtained. To observe the phenomenon of overtraining, select the ThresholdTransmitter and change the value of the Threshold from 0 to 100 (from the inspector window), or simply remove this component from the breadboard. Run the network for a few epochs and observe the falling training error and the rising cross validation error.

In comparison to the MLP, this topology does not offer any advantage for this particular problem.

Generalized Feedforward Hints

As with the MLP, the learning rates should be lower towards the output. This needs to be taken into account when setting the learning rates for the synapses that bypass the hidden layers. For example, the learning rate of the synapse which connects the input directly to the output should be roughly the same as the synapse which connects the last hidden layer to the output. The reason is that they are both extracting linear features from the data. Since the learning for the linear portion of the data happens much faster than for the non-linear portion, this can skew the performance.

Generalized Feedforward Theoretical Summary

These networks are simply an extension to the Multilayer Perceptrons, so the theory and the learning rules are the same.

Modular Feedforward Networks

As the name indicates, the modular feedforward networks are special cases of MLPs, such that layers are segmented into modules. This tends to create some structure within the topology, which will foster specialization of function in each sub-module. In biology, modular networks are very common.

Advantages

In contrast to the MLP, modular feedforward networks do not have full interconnectivity between the layers. Therefore, a smaller number of weights are required for the same size network (the same number of PEs). This tends to speed the training and reduce the number of examples needed to train the network to the same degree of accuracy.

Disadvantages

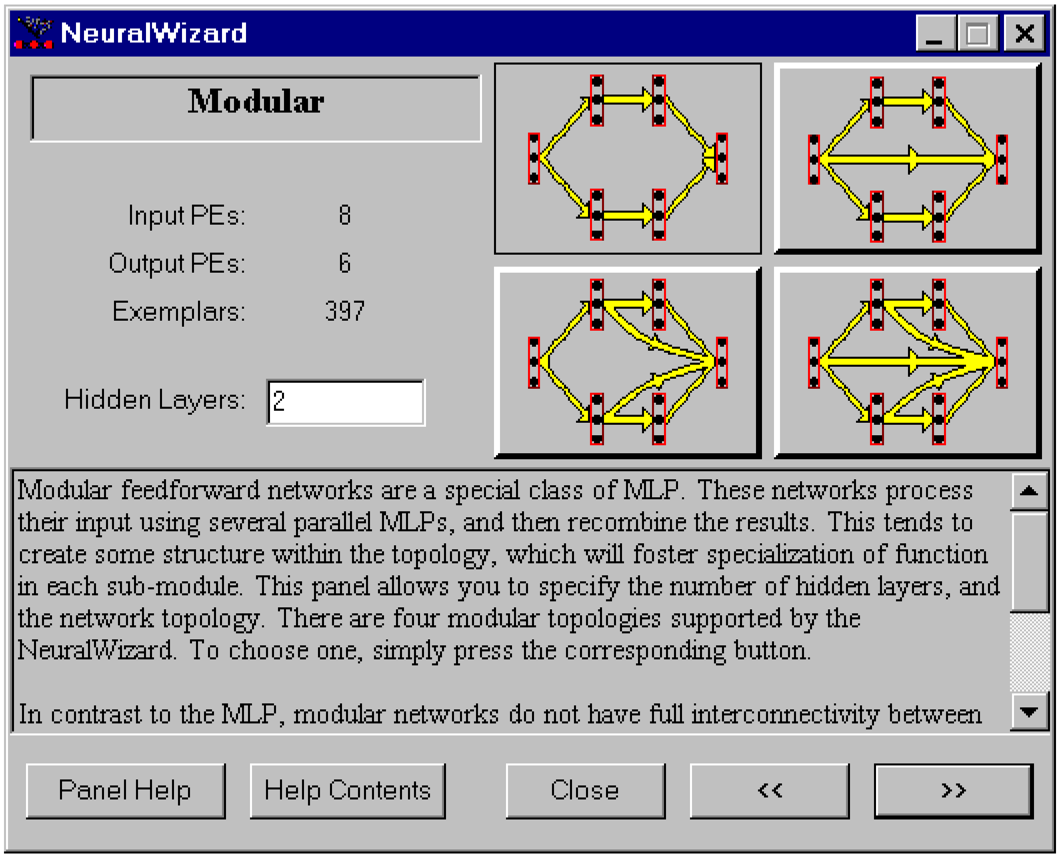
There are many ways to segment a MLP into modules. It is unclear how to best design the modular topology based on the data. There is no guarantee that each module is specializing its training on a unique portion of the data.

Alternative topologies

Multilayer Perceptrons and Generalized Feedforward Networks.

Configuration

From the Modular panel (see figure below), you can configure the number of layers and the specific topology (i.e., how the connections are made). There are four possible topologies to choose from.



Input PEs: 8
Output PEs: 6
Exemplars: 397
Hidden Layers: 2

Modular feedforward networks are a special class of MLP. These networks process their input using several parallel MLPs, and then recombine the results. This tends to create some structure within the topology, which will foster specialization of function in each sub-module. This panel allows you to specify the number of hidden layers, and the network topology. There are four modular topologies supported by the NeuralWizard. To choose one, simply press the corresponding button.

In contrast to the MLP, modular networks do not have full interconnectivity between

Modular panel of the NeuralBuilder

The Hidden Layer panels are the same as for the MLP, except that you can configure the nonlinearity and the number of PEs for both the top and bottom halves of the layer.

Modular Feedforward Hints
Modular Feedforward Theoretical Summary

Modular Feedforward Hints

See the hints for the Multilayer Perceptrons and Generalized Feedforward Networks.

Modular Feedforward Theoretical Summary

These networks are simply an extension to the Multilayer Perceptron, so the theory and the learning rules are the same. The only aspect to note is that a modular network requires fewer weights versus a MLP with the same number of PEs. This will decrease the number of required training patterns, according to the rule of thumb explained under MLP Hints. A modular network will generally train faster than a MLP, due to the fact that it has "short-cut" connections to the output, aiding in the weight adaptation for the hidden and input layers.

Radial Basis Function Networks

Radial basis function (RBF) networks have a static Gaussian function as the nonlinearity for the hidden layer PEs. The output PEs are normally linear. The Gaussian function responds only to a small region of the input space where the Gaussian is centered.

The key to a successful implementation of these networks is to find suitable centers for the Gaussian functions. This can be done with supervised learning, but an unsupervised approach usually produces better results. For this reason, NeuroSolutions implements RBF networks as a hybrid supervised-unsupervised topology.

The simulation starts with the training of an unsupervised layer. Its function is to derive the Gaussian centers and the widths from the input data. These centers are encoded within the weights of the unsupervised layer using competitive learning. During the unsupervised learning, the widths of the Gaussians are computed based on the centers of their neighbors. The output of this layer is derived from the input data weighted by a Gaussian mixture.

Once the unsupervised layer has completed its training, the supervised segment then sets the centers of Gaussian functions (based on the weights of the unsupervised layer) and determines the width (standard deviation) of each Gaussian based on the centers of its neighbors. The NeuralBuilder configures the supervised segment as a linear combiner, although any supervised topology (such as a MLP) may be used. The supervised segment uses the weighted input instead of the input data read from the data file.

There is a special case of the RBF where the number of cluster centers is equal to the number of exemplars. In this case the network does not require any training and all the weights can be set analytically. These networks are called Generalized Regression or Probabilisic Nets, depending on whether the desired outputs are continuous or discrete, respectively. They should only be used when the number of exemplars is small (<100) and the data is so scattered that clustering is ill-defined.

Advantages

The advantage of the radial basis function network is that it finds the input to output map using local approximators. Each one of these local pieces is weighted linearly at the output of the network. Since they have fewer weights, these networks train extremely fast and require fewer training samples.

Disadvantages

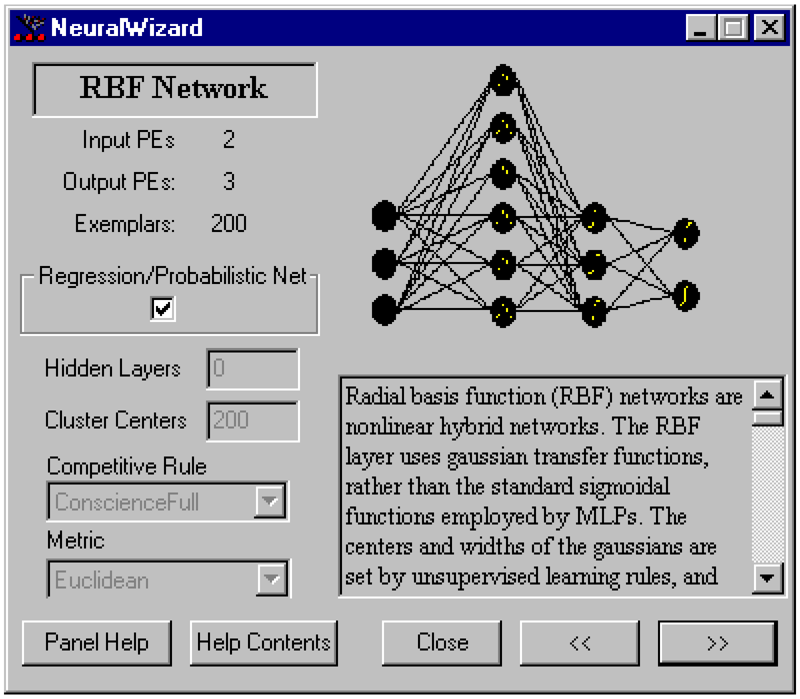
A problem may require a lot of radial basis functions to cover a very large dimensionality space.

Alternative Topologies

In principle MLPs can be used to solve the same problems as the radial basis function networks. But the MLP has many more weights requiring more training exemplars and yielding a slower convergence. The "Principal Component Analysis Networks" network may be a good alternative.

Generalized Regression / Probabilistic Net Configuration

If a Generalized Regression / Probabilistic Net is chosen, the number of hidden layers will automatically be set to 0, the number of cluster centers will be set to the number of exemplars, and the metric will be set to "Euclidean". Once built, this network does not require training, since all of the network weights are determined analytically from the data by the Wizard.   However, for convenience, the backprop plane is left in place but with learning turned off. The Neural Wizard calculates and sets all the Gaussian widths (variances) to the same value based on a standard formula involving the number of exemplars and the dimension of the input. This value determines the smoothness of the interpolation between Gaussian centers. You can change this value by placing a MatrixEditor on top of the GaussianAxon, setting it's access point to "width", and editing the values.


NeuralWizard

RBF Network

Input PEs 2

Output PEs: 3

Exemplars: 200

Regression/Probabilistic Net [✓]

Hidden Layers 0

Cluster Centers 200

Competitive Rule ConscienceFull

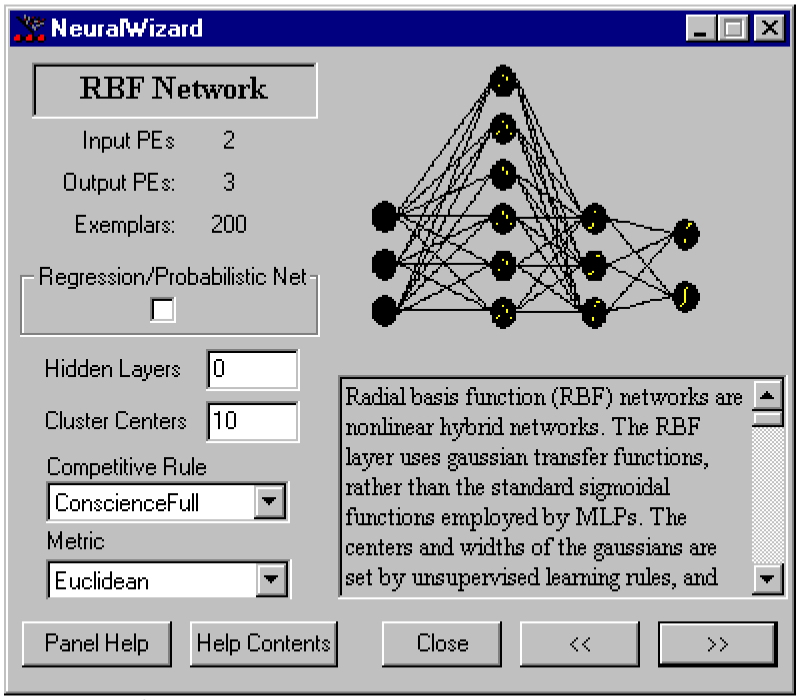
Metric Euclidean

Radial basis function (RBF) networks are nonlinear hybrid networks. The RBF layer uses gaussian transfer functions, rather than the standard sigmoidal functions employed by MLPs. The centers and widths of the gaussians are set by unsupervised learning rules, and

RBF panel of the NeuralBuilder for Generalized Regression/Probabilistic Nets

Supervised/Unsupervised Configuration

Based on the theory, the supervised segment of the network only needs to produce a linear combination of the output at the unsupervised layer. This is the default (0 Hidden Layers at the RBF Network panel (see figure below) and a linear PE Transfer Function at the Output Layer panel). Hidden Layers can be added to make the supervised segment a MLP instead of a simple linear perceptron.


NeuralWizard

RBF Network

Input PEs 2

Output PEs: 3

Exemplars: 200

Regression/Probabilistic Net [ ]

Hidden Layers 0

Cluster Centers 10

Competitive Rule ConscienceFull

Metric Euclidean

Radial basis function (RBF) networks are nonlinear hybrid networks. The RBF layer uses gaussian transfer functions, rather than the standard sigmoidal functions employed by MLPs. The centers and widths of the gaussians are set by unsupervised learning rules, and

RBF panel of the NeuralBuilder

The number of Gaussians is entered using the Prototype Neurons field. It is impossible to suggest an appropriate number of Gaussians, because it is problem dependent. We know that the number of patterns in the training set affects the number of centers (more patterns imply more Gaussians), but this is mediated by the dispersion of the clusters. If the data is very well clustered, then few Gaussians are needed. On the other hand, if the data is scattered, many more Gaussians are required for good performance. Once again, the idea is to start small, test the performance (MSE), and increase the Gaussians if needed.

The selections for the learning address two peculiarities of competitive learning. Competitive learning has an intrinsic metric. You can choose from the dot product metric, box car and Euclidean metrics. Provided that the inputs and weight vectors are normalized, the two are equivalent. The dot product measures the angle between the present input and the weight vector. The Euclidean metric measures the difference between the two vectors such that it preserves distances in the input space.

Competitive learning also keeps an intrinsic probability distribution of the input data. It has the drawback that some PEs may never fire, while others may always win the competition. To avoid these extremes, we can include a "conscience" mechanism that keeps a count on how often a PE wins the competition, and enforces a constant winning rate across the PEs. If you want to place the centers of the Gaussians with a conscience mechanism (the default) use the ConscienceFull component, otherwise select StandardFull from the Competitive Rule pull-down menu. Use of StandardFull component will place more Gaussians in areas of higher sample density, paying less attention to areas with fewer data samples.

Recall that the RBF is a hybrid supervised-unsupervised network. Each of these two segments of the network are controlled separately. The Unsupervised Learning Control panel is inserted between the Output Layer panel and the Supervised Control Panel. This panel is used to specify how long to train the unsupervised layer. Once the epoch count reaches the Maximum Epochs the unsupervised learning terminates and control is passed to the supervised segment. This unsupervised learning may terminate before it reaches this maximum if all of the weights change by less than the amount listed within the Termination box (provided that the Weight Change switch is set).

The learning rate should Start At a high value (but below the value that makes the network diverge) to find a reasonable solution quickly. As the unsupervised learning progresses, this rate should slowly Decay To a lower limit. Note that the value of the termination criterion for the weight change should be set below (10 to 50%) of the value entered in the "Decay to" field.

RBF Example
RBF Hints
RBF Theoretical Summary

RBF Example

Note that this network is only available in the Users version or above. If you are running a lower-level version, then you can still run this example in Evaluation mode.

Select Radial Basis Function Network from the first NeuralBuilder panel. Select sleep2.asc as the training input file, skipping the Min, K-Comp and Score columns. Select sleep2t.asc as the training desired file. Skip the Testing Data panel for now (by turning off the Perform Validation switch).

From the RBF Network panel (see figure), enter 0 as the number of Hidden Layers. This implies that the supervised layer is a simple perceptron, which will linearly discriminate the outputs of the Gaussians. The Prototype Neurons field specifies the number of Gaussians (i.e., the number of PEs in the GaussianAxon component). Set this field to 12. Use the competitive with conscience (ConscienceFull) rule to find the centers and widths of the Gaussians. The centers should be based on the Euclidean metric. Keep the parameter settings of the Output Layer panel at the defaults.

The RBF network needs to adapt the centers and variances (widths) of the clusters before performing the classification. This is done during the unsupervised phase of the training. From the Unsupervised Learning panel, enter 150 as the Maximum Epochs. This will be the actual number of training epochs for the unsupervised phase since you have not Activated the termination based on the Weight Change. The default step size for the competitive learning Starts at 0.01 and linearly Decays to 0.001. Keep these default values and switch to the Supervised Learning panel. The supervised learning begins after the 150 epochs of unsupervised learning has completed. Keep the default settings, switch to the next panel, and Build the network.

For this RBF network, the component connected to the input Axon is the ConscienceFull . This component implements competitive learning with a conscience to find the centers and widths of the Gaussians contained within the next component, the GaussianAxon. The GaussianAxon component is initially represented by a cracked icon

to signify that the supervised segment of the network is inactive. Once the unsupervised segment has completed its training, this icon will change to that of the GaussianAxon

.

Run the network. The simulation quickly goes through the unsupervised training phase (the first 150 epochs) and then the training of the perceptron begins. The error decays very steadily since there are no hidden layers. The error after 350 iterations (200 of them supervised) should be about 0.10. This performance is not quite as good as that achieved with the previous examples. Try choosing different sizes for the RBF layer (number of Prototype Neurons) in an attempt to improve the performance. Many centers may not necessarily help the classification. Since the algorithm is adapting the centers and the widths of the Gaussians, too many centers will make the widths of the Gaussians smaller than required and will produce poor performance.

To test the generalization of this network, go to the Cross Validation Data panel and select the file sleep1.asc as the Input and sleep1t.asc as the Desired. From the Supervised Learning panel, set the MSE Termination to trigger when there is an Increase of more than 0 of the Cross Validation Set error. Build and run the network again.

After a total of 350 iterations, the error decreases to 0.1 for the training set and 0.3 for the cross validation set. The cross validation set error will not begin to increase for several hundred more epochs, after which the network would terminate automatically. The error in the cross validation set is rather disappointing.

One thing to try is to expand the perceptron to a single hidden layer MLP. Change the Hidden Layers field of the RBF Network panel from 0 to 1. Specify 8 Processing Elements for the hidden layer. Build and run the network. Each epoch takes longer because of the extra weights, but the cross validation set performance improved to an error of 0.28 after 350 iterations. Try decreasing the number of PEs in the RBF layer (i.e., from 12 to 8) to possibly further improve the generalization.

RBF Hints

Use the generalized regression/probabilistic nets only for small data sets (<100 exemplars) where the cluster centers are ill-defined.

In order for an RBF network to train efficiently, the Gaussian functions should all have widths that are of the same order of magnitude (e.g., they should all be between 3 and 30 and not between 3 and 3000). To verify this, place a MatrixViewer on the Width access point of the GaussianAxon. If these widths vary a large amount (i.e., by more than one order of magnitude), then the number of Gaussians may need adjusting (see below). Two other sources of problems are the parameters used for the competitive learning and the computation of the Gaussian widths.

The default unsupervised learning rule used by the NeuralBuilder is "competitive learning with a conscience". From the ConscienceFull inspector, the Gamma parameter is used to adjust the level of "conscience". If two or more centers converge closely together, then the Gamma parameter may be set too high. If one or more of the centers diverge away from the input data, this parameter may be set too low. Note that a Gamma setting of 0 is equivalent to the standard competitive learning rule. Note that this component often works best when the gamma starts out at a high value and is linearly decremented to a low value.

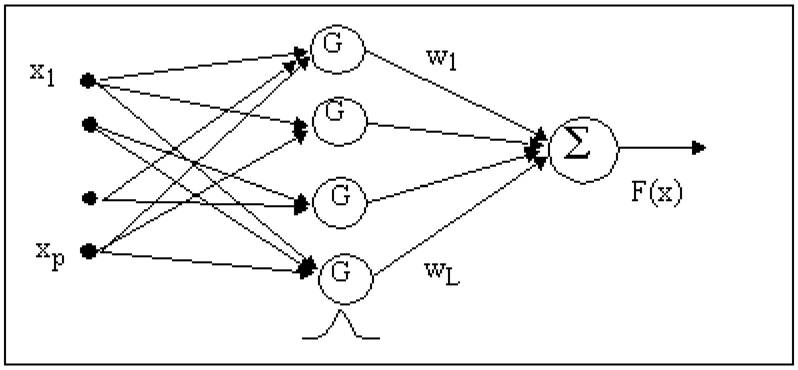
The widths of the Gaussians are determined by the centers of the nearest neighbors and the parameter P from the GaussianAxon inspector. This parameter sets the number of nearest neighbors that are averaged together when computing the widths. If this is set low and there are clusters of centers that are relatively close together, then the resulting widths will often be too small (filtering out important data). If P is set high, then many of the neighbors will be averaged together and the resulting widths may be too high (blending the Gaussians together).

RBF Theoretical Summary

Radial basis functions (RBF) networks solve the mapping problem by local pieces, unlike the MLPs. A layer of special hidden units made up of Gaussian functions are placed judiciously on the input data space. Then the output of these Gaussians is linearly weighted to produce the desired response.

This solution should work if the centers of the Gaussians are appropriately positioned within the data space and their widths are properly chosen. In order to estimate the positions of each radial basis function and its variance (width), an unsupervised technique such as competitive learning is used. The input space is discretized into clusters and the size of each is obtained from the structure of the input data.

The output weights are obtained using supervised learning. The convergence is usually fast since the output units are linear. Note that the GaussianAxon is initially displayed as a cracked component. This is a visual indication that the learning is first restricted to the left part (the unsupervised segment) of the network. After the unsupervised layer has finished training, the unsupervised weights are fixed and the supervised segment begins training. This is indicated by the removal of the crack from the GaussianAxon. This approach tends to improve the training times.



Radial basis function network

In the special case where the number of cluster centers is exactly equal to the number of exemplars, and the output is linear, the network becomes known as a generalized regression or probabilistic net, depending on whether the network targets are continuous or represent probabilities, respectively. In this special case, all the network weights can be calculated analytically from the data. If x(n) is the input training set, and d(n) is the desired output, then for some new input x0, the output y is given by:

$$y = \frac{\sum_{n=1}^{N} d(n) Exp\left[-\frac{[x_0 - x(n)]^2}{2\sigma^2}\right]}{\sum_{n=1}^{N} Exp\left[-\frac{[x_0 - x(n)]^2}{2\sigma^2}\right]}$$

Generalized regression equation

The variance for all Gaussian centers is the same and is set according to the equation,

$$\sigma = cN^{-E/a} \qquad 0 \le E \le 1$$

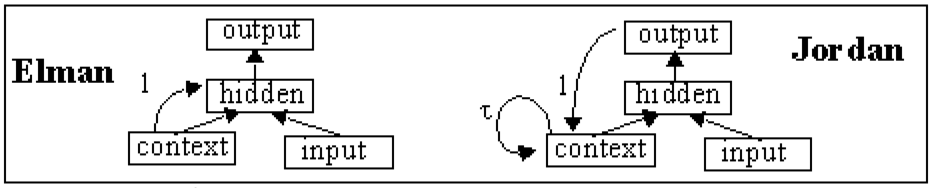
Generalized regression variance equation

where "c" and "E" are constants and "a" is the dimensionality of the input.

Jordan and Elman Networks

Jordan and Elman networks extend the multilayer perceptron with context units, which are PEs that remember past activity. Context units are required when learning patterns over time (i.e., when the past value of the network influences the present processing). In the Elman network, the output of the hidden PEs from the previous time step are copied to the context units (see figure below). In the Jordan network, the output of the network is copied to the context units. In addition, the context units are locally recurrent (i.e., they feedback onto themselves). The local recurrence decreases the values by a multiplicative constant τ (time constant) as they are fed back. This constant determines the memory depth (i.e., how long a given value fed to the context unit will be "remembered").

One can treat the context units as input units, just as if they were obtained from an external source such as a file. Since the recurrent connections within the context units are fixed, static backpropagation is used to train these networks. Note that if the recurrent connections were adaptive, then backpropagation through time would be required.



Block diagrams of Jordan and Elman neural models

Advantages

The previous neural models can only solve static problems. Temporal problems are ones where the previous value of the input affects the current output. The Jordan and Elman networks can solve temporal problems by processing information over time using recurrent connections. Other neural models can solve temporal problems using memory units (see Time Lagged Recurrent Networks).

The Jordan network is slightly more versatile than the Elman network because it can retain older past information due to the locally recurrent connections of the context units. Context units (neurons that self excite) are very common in the brain.

Disadvantages

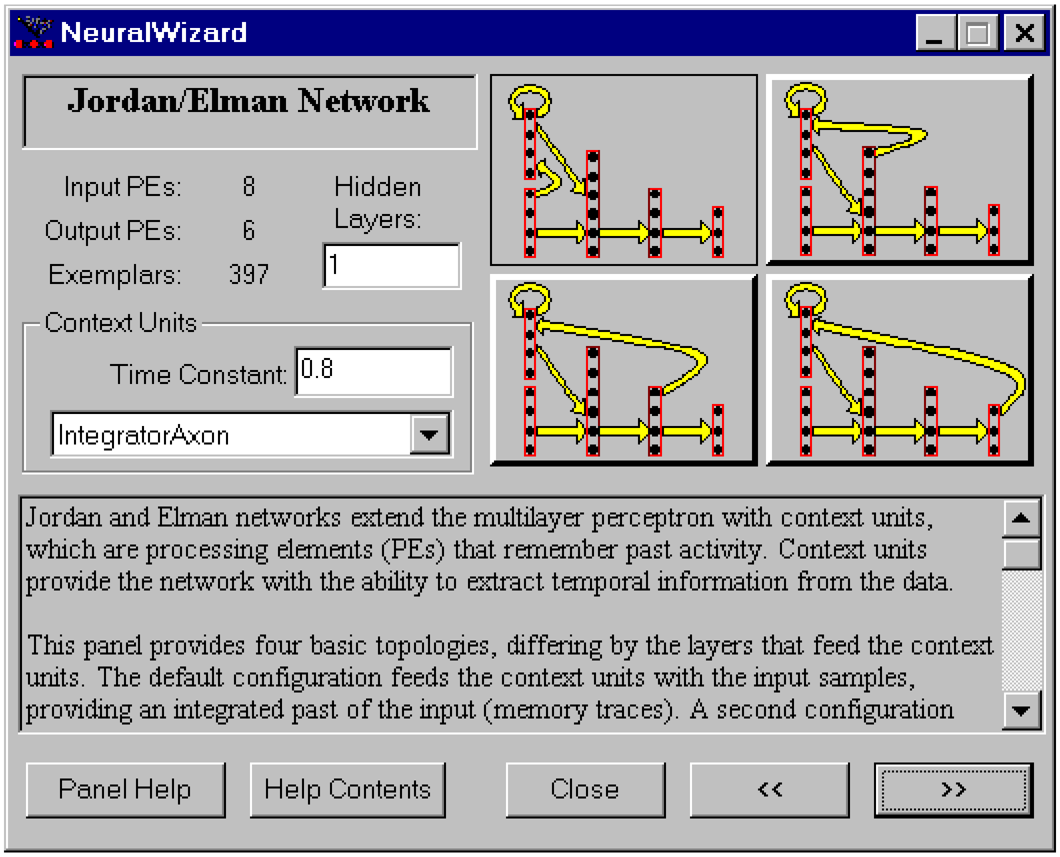
Both of these nets are constrained in their ability to handle time. The time constant of the Jordan network is fixed and often difficult to set optimally for a given problem. Moreover, the past is always exponentially attenuated, which may not be very representative of the problem.

Alternative topologies

An alternate topology would be a MLP such that each input channel is extended to D PEs. The D past samples of each input channel would be used to compose the MLP output. Notice that this requires huge neural topologies that are very difficult to train (lots of input data and long training times). See Time Lagged Recurrent Networks for more versatile temporal topologies.

Configuration

The unique parameters of the Jordan/Elman networks involve the configuration of the context units. The Jordan/Elman Network panel provides four basic topologies, differing by the layers that feed the context units (see figure below).



Jordan and Elman networks extend the multilayer perceptron with context units, which are processing elements (PEs) that remember past activity. Context units provide the network with the ability to extract temporal information from the data.

This panel provides four basic topologies, differing by the layers that feed the context units. The default configuration feeds the context units with the input samples, providing an integrated past of the input (memory traces). A second configuration

Jordan/Elman panel of the NeuralBuilder

The default configuration feeds the context units with the input samples, providing an integrated past of the input (memory traces). A second configuration creates memory traces from the first hidden layer (as proposed by Elman). A third possibility is to use the past of the last hidden layer activations as input to the context units. The final choice is to use the past of the output layer to create the memory traces, as proposed by Jordan.

The context unit remembers the past of its inputs using what has been called a recency gradient, i.e., the unit forgets the past with an exponential decay. This means that events that just happened are stronger than the ones that have occurred further in the past. The context unit controls the forgetting factor through the time constant. Useful values are between 0 and 1. A value of 1 is useless in the sense that only the past is factored in. On the other extreme, a value of zero means that only the present time is factored in (i.e., there is no self-recurrent connection). The closer the value is to 1, the longer the memory depth and the slower the "forgetting" factor.

The pull-down menu within the Context Units box is used to select the transfer function of the context units. There are linear and nonlinear context units, as well as linear and nonlinear integrators (see table below). The integrators are the same as context units except that they normalize the input based on the time constant τ.

Context unit types

Type	Description
IntegratorAxon	linear integrator
SigmoidIntegratorAxon	saturating integrator (0/1)
TanhIntegratorAxon	saturating integrator (-1/+1)
ContextAxon	linear decay
SigmoidContextAxon	saturating context decay (0/1)
TanhContextAxon	saturating context decay (-1/1)

The number of PEs in the context layer is defined by the number of PEs in the layer that feeds the context layer (i.e., the network will assign one context unit per input connection). As with the other neural models, the number of Hidden Layers must be defined. Note that if the are 0 hidden layers then the 1st and 2nd topologies are equivalent, as are the 3rd and the 4th. If there is 1 hidden layer, then the 2nd and the 3rd topologies are equivalent. With 2 or more hidden layers, all 4 topologies are unique.

Jordan/Elman Example
Jordan/Elman Hints
Jordan/Elman Theoretical Summary

Jordan/Elman Example

Note that this network is only available in the Users version or above. If you are running lower-level version, then you can still run this example in Evaluation mode.

Select Jordan/Elman Network from the first NeuralBuilder panel. Select sleep2.asc as the input training file, skipping the Min, K-Comp and Score columns. Select sleep2t.asc as the desired training file. From the Testing Data panel, select sleep1.asc and sleep1t.asc as the Input and Desired files respectively.

From the Jordan/Elman Network panel (see figure), keep the default setting of 1 Hidden Layer. Note that there are four different topologies to choose from. Select the upper-left button to specify that the memory units are fed by the input layer. The Time Constant parameter specifies the memory depth of the context units. Set this to 0.4. The memory should be implemented using the IntegratorAxon. Set the number of PEs in the hidden layer to 8. Set the Momentum to 0.7 for both the hidden and the output layer. Verify that the MSE termination is activated for the cross validation set (set the Increase radio button and the Threshold to 0). Build and run the network.

After 100 epochs, the training set error should drop to around 0.09 and the cross validation set error should be near 0.21. Sometime before the next 100 epochs, the simulation should stop when the cross validation set error begins to rise. At this point, the training error should be around 0.07 and the cross validation error down around 0.19. This is the best performance when compared with the previous examples.

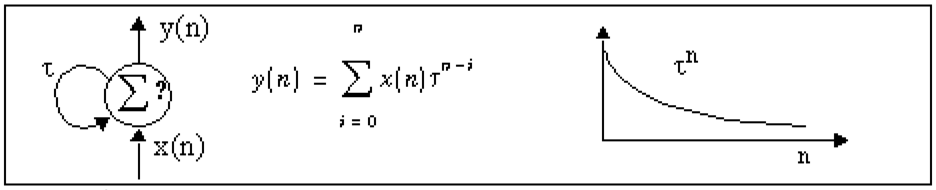
The changes between the sleep stages occur with a predictable rhythm. This rhythm is contained within the temporal (time) information of the signals. The reason that the Jordan network was able to outperform the others is that it was able to extract this temporal information. The previous examples used static neural models and were unable to capture this information.

Jordan/Elman Hints

It is difficult to give an idea of the number of context units and their time constants, since their number is problem dependent. The time constant τ should be set to 1 - 1/D (D>0), where D is the number of time samples to "remember" the patterns (memory depth).

Jordan/Elman Theoretical Summary

The theory of neural networks with context units can only be analyzed mathematically for the case of linear PEs. In this case, the context unit is nothing but a very simple lowpass filter. A lowpass filter creates an output that is a weighted (average) value of some of its more recent past inputs. In the case of the Jordan context unit, the output is obtained by summing the past values multiplied by the scalar τ^n as shown in the figure below.



Theory of the context unit

An impulse function x(n) outputs a 1 at n=0 and a 0 for n>0. Given this impulse function, the output of the context unit is τ^1 at n=1,

τ^2 at n=2, etc. This is the reason these context units are called memory units, because they "remember" past events. The time constant τ should be less than 1, otherwise the context unit response gets progressively larger (unstable). The Jordan network combines past values of the context units with the present inputs to obtain the present network output. One disadvantage of these nets is that the weighting over time is kind of inflexible since one can only control the time constant τ (i.e., the exponential decay). Moreover, a small change in τ is reflected in a large change in the weighting (due to the exponential relationship between time constant and amplitude). Since the optimal memory depth is usually unknown, the choice of τ can be problematic without a mechanism to adapt it.

Principal Component Analysis Networks

As with the Radial Basis Function Networks, Principal Component Analysis (PCA) networks are a mixture of unsupervised and supervised networks. Principal component analysis is a linear procedure to find the direction in input space where most of the energy of the input lies. In other words, PCA performs feature extraction. The projections of these components correspond to the eigenvalues of the input covariance matrix. The unsupervised segment of the network performs the feature extraction and the supervised segment of the network performs the (linear or nonlinear) classification of these features using a MLP.

The principal component analysis is performed first, and then the MLP is trained. The reason for this is that the PCA network trains faster when it does not have to share computing resources with the MLP. There is no point in training the MLP until the eigenvalues are stable.

Advantages

Principal component analysis is a well known method of orthogonalizing data. It converges very fast and the theoretical method is well understood. Since the features are orthogonal, the MLP is able to train easily. There are usually fewer features extracted than there are inputs, so the unsupervised segment provides a means of data reduction.

Disadvantages

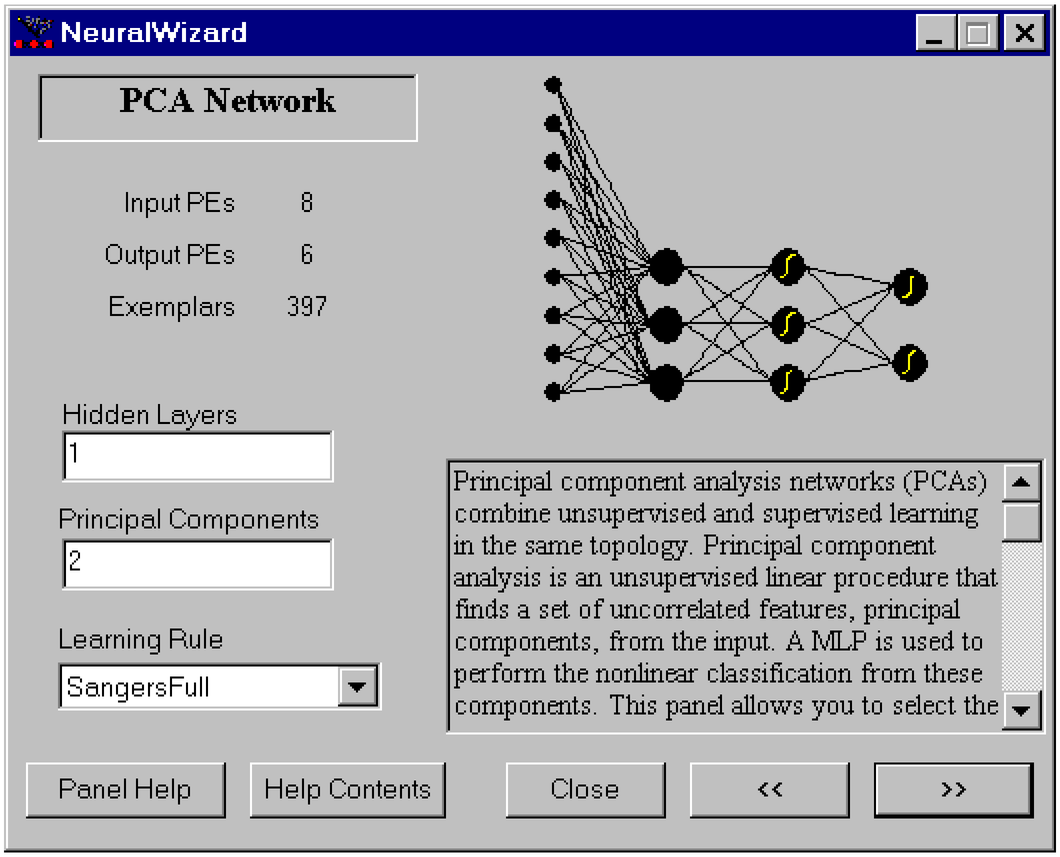
The most discriminant features are not always the features that have the largest eigenvalues. For these cases, the PCA is suboptimal and the choice of the number of features to extract is rather arbitrary.

Alternative topologies

Radial Basis Function Networks and Self-Organizing Feature Map Networks.

Configuration

The PCA Network panel is used to set the number of Principle Components to extract and the unsupervised Learning Rule to use for the feature extraction. (see figure below). The number of Hidden Layers of the MLP is selected from this panel as well.



Input PEs 8
Output PEs 6
Exemplars 397

Hidden Layers: 1

Principal Components: 2

Learning Rule: SangersFull

Principal component analysis networks (PCAs) combine unsupervised and supervised learning in the same topology. Principal component analysis is an unsupervised linear procedure that finds a set of uncorrelated features, principal components, from the input. A MLP is used to perform the nonlinear classification from these components. This panel allows you to select the

PCA panel of the NeuralBuilder

Recall that PCA is a data reduction method, which condenses the input data down to a few principal components. As with any data reduction method, there is the possibility of losing important input information. The number of principal components selected will be a compromise between training efficiency (few PCA components) and accurate results (a large number of PCA components). It is not possible to provide a general formula for selecting an appropriate number of principal components for a given application. See the hints below on how to adjust this parameter based on the results of the unsupervised training.

The learning rule choices for PCA are Sangers and Ojas. They are both normalized implementations of the Hebbian learning rule. Straight Hebbian learning must be utilized with care, since it may become unstable. For this reason, it is not given as an option within this panel. The two more robust choices are Oja's and Sanger's implementations of the Hebbian principle. Between the two, Sanger's is preferred for PCA because it naturally orders the PCA components by magnitude. This also provides an easy way to decide if the number of PCA components is reasonable—simply check the ratio of the first component magnitude versus the last component magnitude.

Since this network is a hybrid supervised-unsupervised network, the control parameters must be set for both the supervised and the unsupervised segments of the network. See Radial Basis Function Networks for instructions on setting the Unsupervised Learning Control parameters.

PCA Example
PCA_Hints
PCA Theoretical Summary

PCA Example

Note that this network is only available in the Users version or above. If you are running a lower-level version, then you can still run this example in Evaluation mode.

Select Principal Component Analysis Network from the first NeuralBuilder panel. Select sleep2.asc as the input training file, skipping the Min, K-Comp and Score columns. Select sleep2t.asc as the desired training file. Skip the Testing Data panel for now (by turning the Perform Validation switch off).

From the PCA Network panel (see figure) enter 0 as the number of Hidden Layers. The number of PEs at the output of the unsupervised component is specified using the Principal Components field. Enter 5 in this field and use the default learning rule (SangersFull).

The output layer is used to combine the features extracted by the unsupervised segment. Note that this topology is only able to find linear separable patterns from the 5 components (features). Keep the parameters of the Output Layer panel at the defaults.

From the Unsupervised Learning Control panel, enter 150 as the Maximum Epochs. The unsupervised learning rate should Start at 0.01 and Decay to 0.0001. Keep the defaults for the rest of the panels, then build and run the network.

Notice that the first part of the learning is unsupervised. During the first 150 epochs, the unsupervised segment trains while the supervised segment is disabled (indicated by the cracked axon). When the epochs counter reaches 150, the MatrixViewer starts displaying error of the supervised training. Note the difference in speed between the two modes.

After about 500 epochs (350 of them supervised), the error drops to about 0.11. Try increasing the number of principal components from 5 to 7 (change the number of PEs in the middle Axon). This change will require a higher learning rate. Select the two Momentum components (click on the first, hold down the Shift key and then click on the second) and change the Momentum from 0.5 to 0.7. Run the network. The performance should have improved (to an error of 0.1 after 500 epochs).

Note that it is useless to have more than eight components, since the input data is eight dimensional. Eight principal components would simply provide a rotation of the data set over the eigenvectors of the input correlation function.

Return to the Cross Validation Data panel and select sleep1.asc and sleep1t.asc as the Input and Desired files respectively. Switch to the PCA Network panel and change the supervised segment from a linear combiner to a MLP (by setting the Hidden Layers to 1). Keep the Principle Components set to 5. Note that this topology is the same demonstrated in the MLP example, but the input has been projected by the PCA layer. In theory, this network should learn a little faster than the original MLP, since the input data becomes orthogonalized.

Set the number of Processing Elements in the hidden layer to 8. Specify that the termination of the supervised training be based on an Increase of the cross validation set error. Build and run the network.

Let the network run until it terminates automatically (when the cross validation set error begins to increase). This should occur after about 500 epochs. Expect a training set error of about 0.07 and a cross validation set error of 0.22. This means that the PCA decomposition is creating a representation that seems to generalize better than the unprocessed data. The training is also faster because there are only 5 inputs into the MLP instead of 8.

PCA Hints

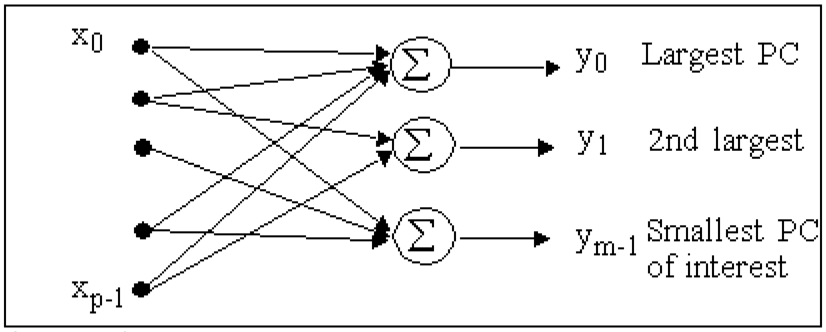
Deciding on the number of features to extract is important to achieve the desired data reduction without losing important information. The Sanger's rule produces an ordered set of eigenvectors. For many problems, there is a large decrease of the eigenvalue amplitude for the high-order components. After training the unsupervised segment, place a MatrixViewer on the axon to the right of the SangersFull component. The PE that displays this large decrease of amplitude should give the natural boundary for the features; there is no need for any more PEs after this one.

PCA Theoretical Summary

The fundamental problem in pattern recognition is defining the data features that are important for the classification (feature extraction). The goal is to transform the input samples into a new space (the feature space) such that the information about the samples is kept, but the dimensionality is reduced. This makes the classification job much easier.

Principal component analysis (PCA) is such a technique. From the input space, it finds an orthogonal set of P directions where the input data has the largest energy, and extracts P projections from these directions in an ordered fashion. The first principal component is the projection, which has the largest value (think of the projections as the shadow of the data clusters in each direction), while the Pth principal component has the smallest value. If the largest projections are extracted, then the most significant information about the input data is kept.

Principal component analysis is normally done by analytically solving an eigenvalue problem of the input correlation matrix. Sanger and Oja demonstrated that PCA can be accomplished by a linear, single layer neural network trained with a modified Hebbian learning rule. Sanger's rule is recommended, since it orders the projections.



Ordering of the principal components

It is interesting to note that this segment of the network computes the eigenvectors of the input's correlation function without ever computing the correlation function itself. The outputs of the PCA layer are therefore related to the eigenvalues and can be used as input features to the supervised segment for classification. Since many of these eigenvalues are usually small, only the M (M<P) largest values need to be kept. This speeds up training even more.

Principal component analysis can be used for data compression, producing the M most significant linear features. When used in conjunction with a multilayer perceptron (MLP) to perform classification, the separability of the classes is not always guaranteed. If the classes are not sufficiently separated, the PCA will extract the largest projections while the separability could be contained within some of the smaller projections.

Another problem with linear PCA networks is evident when the input data contains outliers. Outliers are individual pieces of data that are far removed from the data clusters (i.e., noise). They tend to distort the estimation of the eigenvectors and create skewed data projections.

The importance of PCA analysis is that the number of inputs for the MLP classifier can be significantly reduced. This results in a reduction of the number of required training patterns and a reduction in the training times of the classifier.

Self-Organizing Feature Map Networks

Self-organizing feature maps (SOFM) transform the input of arbitrary dimension into a one or two dimensional discrete map subject to a topological (neighborhood preserving) constraint. The feature maps are computed using Kohonen unsupervised learning. The output of the SOFM can be used as input to a supervised classification neural network such as the MLP.

Advantages

This network's key advantage is the clustering produced by the SOFM which reduces the input space into representative features using a self-organizing process. Hence the underlying structure of the input space is kept, while the dimensionality of the space is reduced.

Disadvantages

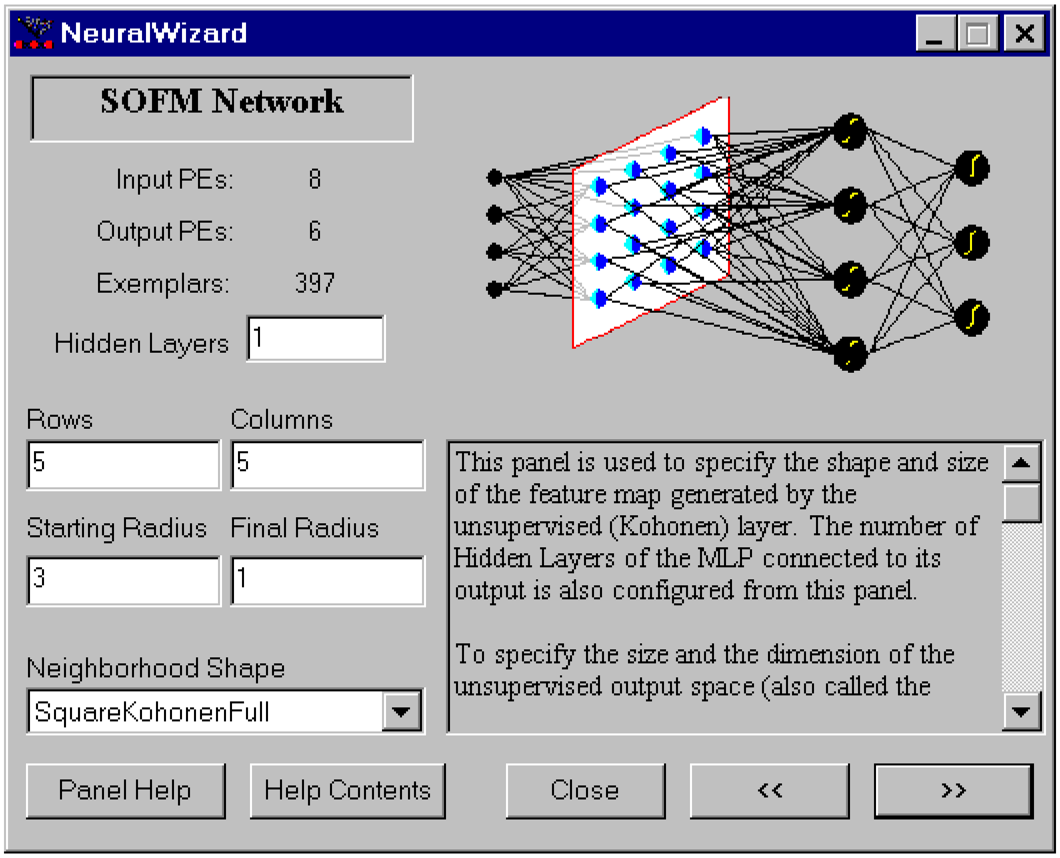
It is difficult to measure the performance of the SOFM, resulting in an absence of a stop criterion. This may impact the stability of the map.

Alternative topologies

Radial Basis Function Networks and Principal Component Analysis Networks.

Configuration

The SOFM Network panel is used to specify the shape and size of the feature map generated by the unsupervised (Kohonen) layer (see figure below). The number of Hidden Layers of the MLP connected to its output is also configured from this panel.



Input PEs: 8

Output PEs: 6

Exemplars: 397

Hidden Layers: 1

Rows: 5

Columns: 5

Starting Radius: 3

Final Radius: 1

Neighborhood Shape: SquareKohonenFull

This panel is used to specify the shape and size of the feature map generated by the unsupervised (Kohonen) layer. The number of Hidden Layers of the MLP connected to its output is also configured from this panel.

To specify the size and the dimension of the unsupervised output space (also called the

SOFM panel of the NeuralBuilder

To specify the size and the dimension of the unsupervised output space (also called the neural field), enter the number of Rows and Columns of the 2D neural field. For a 1D neural field (i.e., LineKohonen) the length of the line (number of PEs) is the product of the values entered. Increasing the number of PEs will improve the resolution, but also increase the training times. Since the SOFM normally trains slowly, the size of the Kohonen layer should be kept as small as possible. The shape of the neighborhood is also important because it defines organizational rules within the neural field. The Neighborhood Shapes available are summarized in the table below. Experiment with each of these neighborhoods to find out which provides the best results for your data.

Kohonen Neighborhoods

Type	Description
Square	2D space, 8 neighbors
Diamond	2D space, 4 neighbors
Line	1D neighborhood

The final shape of the feature map is heavily dependent on the size of the initial and final neighborhoods. These quantities are labeled as Starting and Final Radius. The complete neighborhood is the default value for the initial radius. This has been shown to preserve the probability distribution of the input data, but makes learning very slow. You may want use an initial neighborhood that is 70% of the linear size of the 2D field. The final radius should be a small value, normally one or two PEs wide. A radius of 1 corresponds to a single PE neighborhood. A radius of 2 corresponds to the nearest neighbors of the winning PE. Its shape depends upon the specified neighborhood: in the diamond, only the four nearest neighbors (up, down, left, right) are considered. In the square neighborhood, all of the adjacent PEs (8) are included.

The SOFM network is a hybrid network, in the sense that unsupervised and supervised learning are utilized. See Radial Basis Function Networks for configuration instructions for the Unsupervised Learning Control panel.

SOFM Example
SOFM Hints
SOFM Theoretical Summary

SOFM Example

Note that this network is only available in the Users version or above. If you are running a lower-level version, then you can still run this example in Evaluation mode.

Select Self Organizing Feature Map Network from the first NeuralBuilder panel. Select sleep2.asc as the input training file, skipping the Min, K-Comp and Score columns. Select sleep2t.asc as the desired training file. From the Testing Data panel, select the files sleep1.asc and sleep1t.asc as your Input and Desired cross validation set.

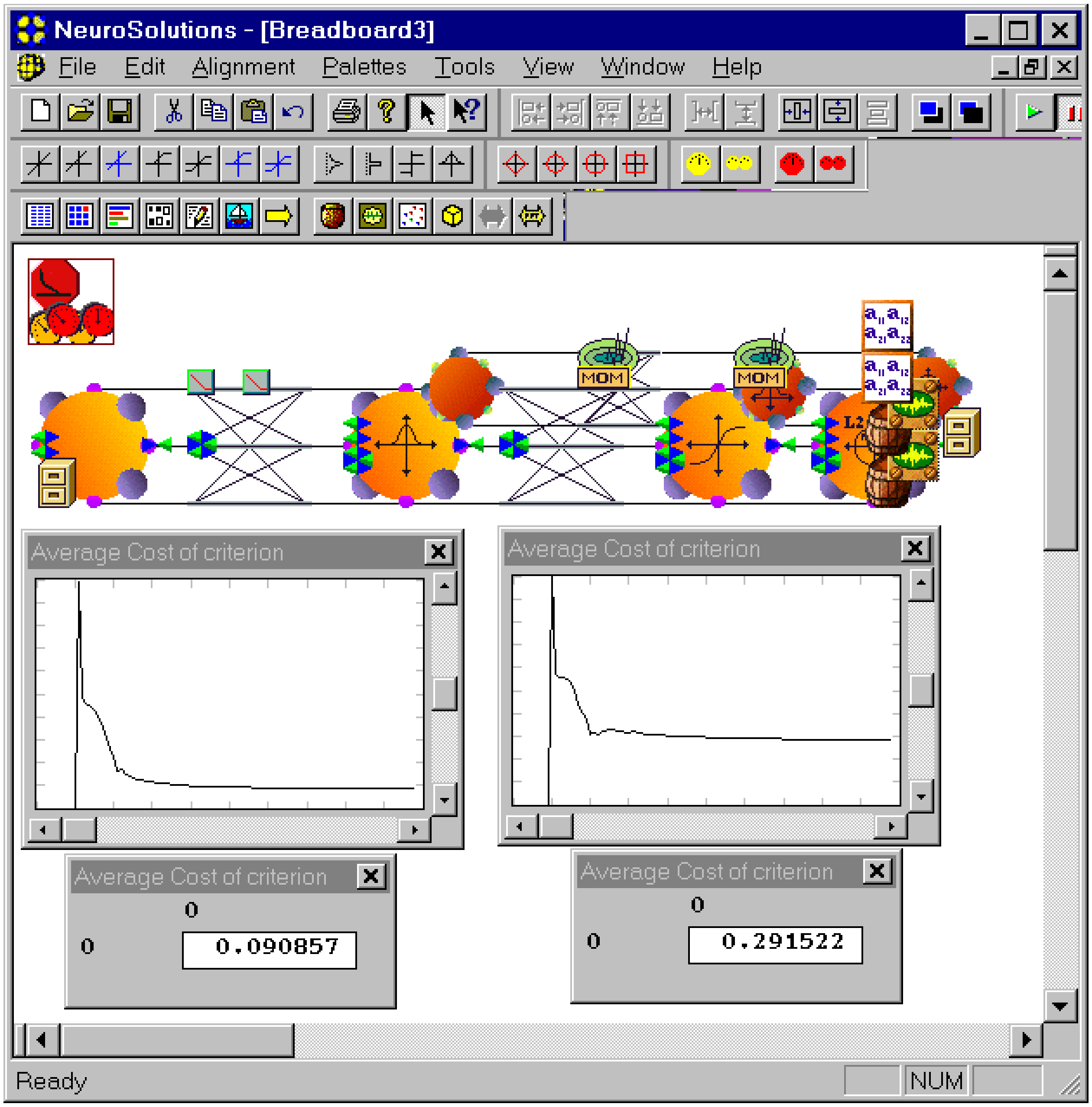
From the SOFM Network panel (see figure) enter 0 as the number of Hidden Layers. Specify a 5x5 (5 Rows and 5 Columns) Kohonen layer that has a square neighborhood (SquareKohonen). The neighborhood should have a Starting Radius of 5 and linearly decay to a Final Radius of 1.

Use all of the defaults for the Output Layer panel. From the Unsupervised Learning Control panel, enter 150 as the Maximum Epochs and use the default learning rates.

As you will see from the learning curves, the network should not be terminated based on a rise of the cross validation set MSE. For this reason, deactivate the MSE Termination. Keep the remaining parameters of the Supervised Learning Control panel at the defaults. From the Data Display panel, select to view Error of the cross validation set and training set with a MegaScope. Build the network.

In order to view the numerical values of the cross validation and training errors, stack two MatrixViewers on top of the DataStorage components, as shown in the figure below. By default, these MatrixViewers are both configured to display the training error. Select one of them and switch to the Access property page of the inspector window. Change the Access Data Set to Cross Validation.

Observe that the input layer is followed by the SquareKohonen component. The output of this component is fed to a GaussianAxon. This component is the input to the supervised segment, which in this case is a simple perceptron. The widths of the Gaussians are fixed and the centers are determined from outputs of the SquareKohonen. Start the simulation.


NeuroSolutions - [Breadboard3]
File Edit Alignment Palettes Tools View Window Help
MOM
MOM
L2
Average Cost of criterion
Average Cost of criterion
Ready
NUM

Average Cost of criterion

```
          0
0    0.090857
```

Average Cost of criterion

```
          0
0    0.291522
```

Zero-layer SOFM network with training and cross validation learning curves

As with the other examples using a hybrid network, the unsupervised segment trains for 150 iterations, then those weights are frozen and the supervised segment begins its training (indicated by the cracked axon icon changing to the icon for the GaussianAxon). After 250 epochs, the training set error should be just below 0.1 and the cross validation set error should be close to 0.25.

Try substituting the perceptron with a one hidden layer MLP (by changing the number of Hidden Layers to 1). Set the number of PEs in the hidden layer to 8 and leave the remaining settings unchanged. After running the network for 250 epochs, you should find a slight improvement in the cross validation set performance (0.24) and the training set performance (0.07).

SOFM Hints

The initial values for the SOFM should all be different small values to brake any symmetry. Both the neighborhood radius and the learning rate should decay over the course of training. The Learning Rate should Start At a large value (at least 0.1) and Decay by at least a factor of 10. Typically the unsupervised training will require about 1,000 epochs. The Starting neighborhood parameter should be large (the whole map) and decay (linearly) until a Final Radius of 1 (a single PE). The MLP should be trained after the SOFM has converged.

SOFM Theoretical Summary

As stated previously, one of the most important issues in pattern recognition is feature extraction. An alternative to the principal components analysis approach is the self-organizing feature map.

The ideas of SOFM are rooted in competitive networks. Competitive networks use linear PEs and a competitive learning rule. There is only one winning PE for each input pattern (i.e., the PE whose weights are closest to the input pattern). With competitive learning, only the weights of the winning node get updated (winner-take-all). Kohonen proposed a slight modification of this principle, referred to as the self-organizing feature map. Instead of updating only the winning PE, the neighboring PE weights are also updated with a smaller learning rate.

During the Kohonen learning process, neighborhood (topological) relationships are created in which the spatial locations correspond to features of the input data. It can be shown that the data points that are similar in input space are mapped to small neighborhoods in Kohonen’s SOFM layer. This SOFM layer contains information about the probability density function of the input data. Our brain has several known topographic maps, namely the visual and auditory cortex.

The SOFM layer can be a one or two dimensional lattice, and the size of the network provides the resolution for the lattice. The SOFM layer receives as input the data points, and organizes its outputs in discrete clusters that are related by a distance metric (the user can choose either a Euclidean, box car or dot product metric). In order to guarantee that all PEs have a chance to compete independently of the relative frequency of occurrence of input data samples, a conscience mechanism is necessary.

Adaptive learning rates and neighborhood sizes are implemented to form local neighborhoods in the beginning, then stabilize and fine tune the map in the later stages of learning. These issues are very difficult to study theoretically, so heuristics have to be used in the specification of these adaptive parameters.

Once the SOFM stabilizes, its output can be used for classification (using a supervised segment) or as a vector quantizer. The vector quantization application allows the clustering of the input data in similar groups. This is in itself a very important application as a "smart switch", but also has obvious applications for data compression.

Time Lagged Recurrent Networks

TLRNs are MLPs extended with short term memory structures that have local recurrent connections. The TLRN is a very appropriate model for processing temporal (time-varying) information. Examples of temporal problems include time series prediction, system identification and temporal pattern recognition. The training algorithm used with TLRNs is more advanced than standard backpropagation.

Advantages

The main advantage of TLRNs is the smaller network size required to learn temporal problems when compared to MLPs that use extra inputs to represent the past samples (equivalent to time delay neural networks). An added advantage of TLRNs is their low sensitivity to noise. The recurrence of the TLRN provides the advantage of an adaptive memory depth (i.e., it finds the best duration to represent the input signal's past).

From a system identification point of view, TLRNs implement nonlinear moving average (NMA) models. With global feedback from the output to the hidden layer, they can be extended to nonlinear ARMA (autoregressive moving average) models. These nonlinear models can be used for optimal control applications, surpassing the performance of their linear counterparts.

Disadvantages

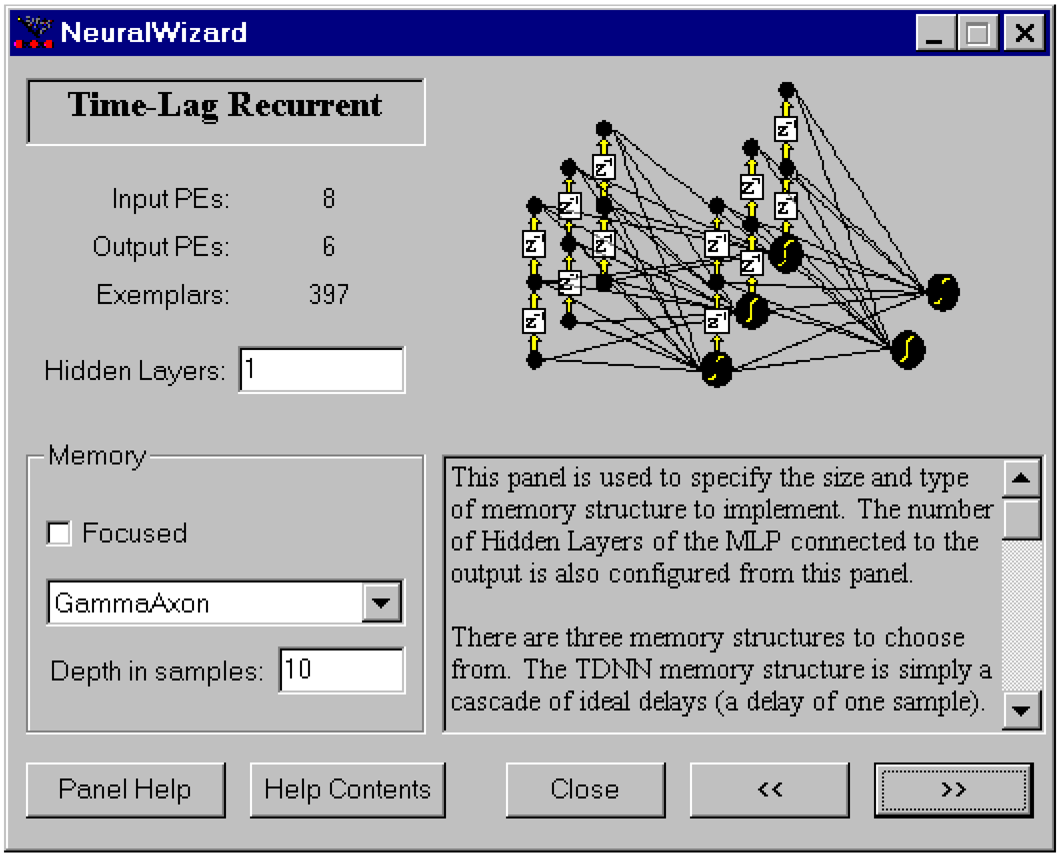
The recurrent adaptation of the weights is nonlinear, so the training can get caught in local minima. Another disadvantage is that straight backpropagation cannot be used for training. The backpropagation through time (BPTT) algorithm is quite complex and requires a lot of memory. NeuroSolutions hides most of the complexity from the user and provides an efficient implementation of this algorithm.

Alternative topologies

Jordan and Elman Networks. A Multilayer Perceptron extended with a tapped delay line can also be utilized, but will require a much larger network.

Configuration

The Time-Lag Recurrent panel is used to specify the size and type of memory structure to implement (see figure below). The number of Hidden Layers of the MLP connected to the output is also configured from this panel.


NeuralWizard

Time-Lag Recurrent

Input PEs: 8

Output PEs: 6

Exemplars: 397

Hidden Layers: 1

Memory

☐ Focused

GammaAxon

Depth in samples: 10

This panel is used to specify the size and type of memory structure to implement. The number of Hidden Layers of the MLP connected to the output is also configured from this panel.

There are three memory structures to choose from. The TDNN memory structure is simply a cascade of ideal delays (a delay of one sample).

TLRN panel of the NeuralBuilder

The table below summarizes the types of Memory structures available. The TDNN memory structure is simply a cascade of ideal delays (a delay of one sample). The gamma memory is a cascade of leaky integrators shown in the Jordan and Elman Networks. The Laguarre memory is slightly more sophisticated than the gamma memory in that it orthogonalizes the memory space. This is useful when working with large memory kernels.

Memory types

Type	Description
TDNNAxon	tap delay line
GammaAxon	gamma memory
LaguarreAxon	orthogonal gamma

The Focused topology only includes the memory kernels in the input layer. This way, only the past of the input is remembered. If the Focused switch is not set, the hidden layers' PEs will also be equipped with memory kernels.

The Depth in Samples parameter (D) is used to compute the number of taps (T) contained within the memory structure(s) of the network. The number of taps within the input memory layer is dependent on the type of memory structure used. For the TDNNAxon, the number of input taps T0 is equal to the depth D. The formula for the other two memory types is T0 = 2D/3. The number of taps for the memory structures at hidden layer n is computed (for all memory types) by the formula Tn = T0/2*n. This is only used as a starting point for the memory depth, since the depth will be adapted by the network.

Recurrent networks must be trained using a dynamic learning algorithm. NeuroSolutions uses the so-called backpropagation through time (BPTT) algorithm. The dynamic controllers are required for this type of learning.

TLRN Hints
TLRN Theoretical Summary

TLRN Hints

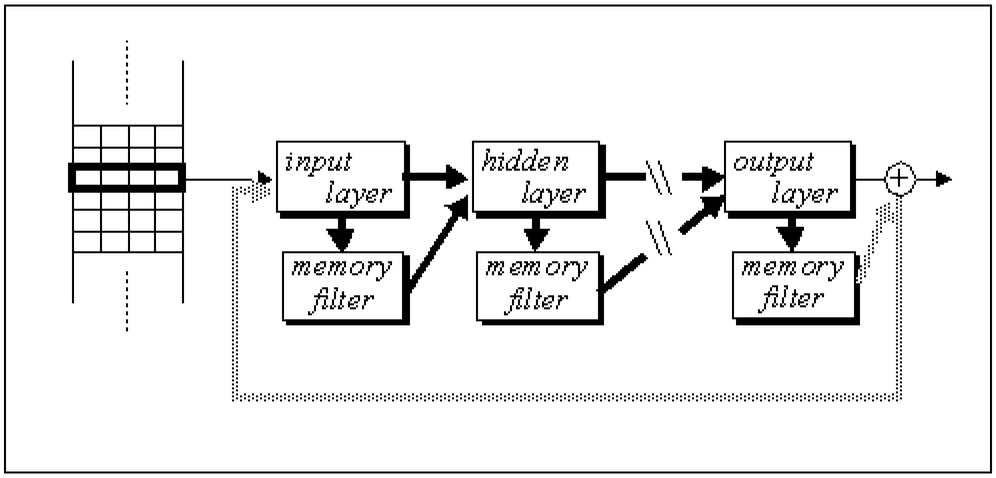
The memory layer connected to the input provides a natural way of storing the past values of the input data. You should begin with a small memory and increase the size as needed. As the network trains, the network adapts the memory depth based on the input data and the number of taps. The memory parameter should be adapted with a step size that is an order of magnitude smaller than the step sizes used for the reset of the network.

If learning seems slow, go to the dynamic controllers, switch to on-line learning, and set the number of Samples/Exemplar (both forward and backward) to a smaller number. Note, however, that this number must evenly divide into the total number of Exemplars.

TLRN Theoretical Summary

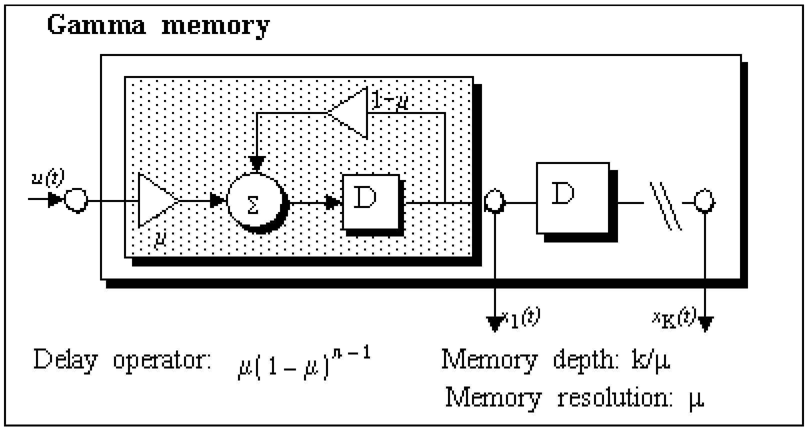
TLRNs with the memory layer confined to the input can be thought of as an input preprocessor. The information is represented across time instead of simply across the static input patterns. Given a signal in time (such as a time series of financial data, or a signal coming from a sensor monitoring an industrial process), the network must process that signal to determine where in time the relevant information lies. The term signal processing is used here in a general sense; it can be substituted for prediction, identification of dynamics, or classification.

A brute force approach is to use a long time window. This method does not work in practice because it creates very large networks that are difficult to train (particularly if the data is noisy). TLRNs are a very good alternative to this brute force approach (see figure below). Another class of models that have adaptive memory are the General Recurrent Networks. These networks are more difficult to train and require a more advanced knowledge of neural network theory.



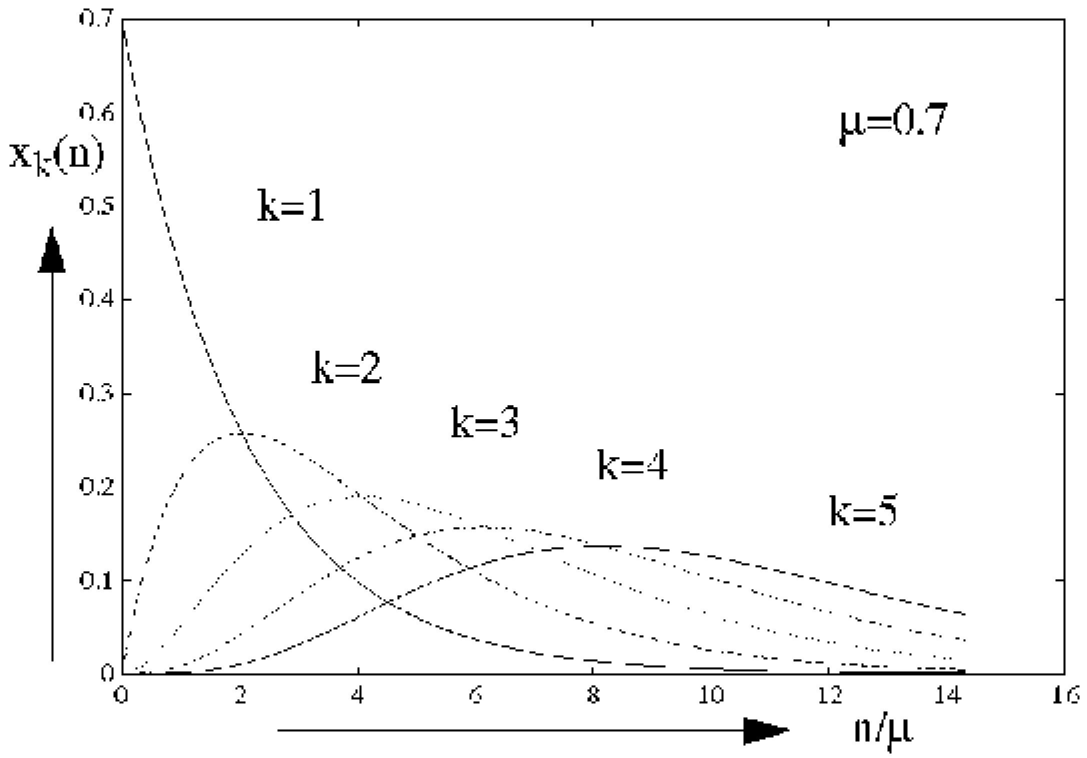
Block diagram of a Time Lagged Recurrent Network

The most studied TLRN network is the gamma model (see figure below). The gamma model is characterized by a memory structure that is a cascade of leaky integrators, i.e. an extension of the context unit of the Jordan network.



Block diagram of the Gamma memory structure

Note that the signal at tap k is a smoothed version of the input that holds the voltage of a past event, creating a memory. The impulse responses of the different taps are shown in the figure below.



Impulse responses of each tap of a 5-tap Gamma memory

Note that the point in time where the response has a peak is approximately given by k/m, where m is the feedback parameter. This means that the neural network can control the depth of the memory by changing the value of the feedback parameter, instead of requiring a topological change in the number of taps. The parameter m can be adapted using gradient descent procedures, just like the other parameters in the network. Since this parameter is recursive, a more powerful learning rule needs to be applied. NeuroSolutions implements backpropagation through time (BPTT) for the adaptation process.

The NeuralBuilder offers a choice of memory structures. The Laguarre memory is an improvement over the gamma memory since it trains faster. It also provides signals at the taps that are not attenuated in amplitude, but appropriately shifted in time.

General Recurrent Networks

General recurrent networks (GRN's) are to temporal data as multi-layer perceptrons (MLP's) are to static data. They are categorized by a layer that feeds back upon itself using adaptable weights. If all of the layer's axon's feed back their output, then the network is fully recurrent, otherwise it is called partially recurrent.

Advantages

The main advantage of GRN's is that they have a potentially unlimited memory depth and thus, as previously stated, can actually capture the dynamics of the system that produced a temporal signal. This distinguishes them from Time Lagged Recurrent Networks, where the memory depth is also adaptable, but has an effective upper limit due to loss of resolution.

Disadvantages

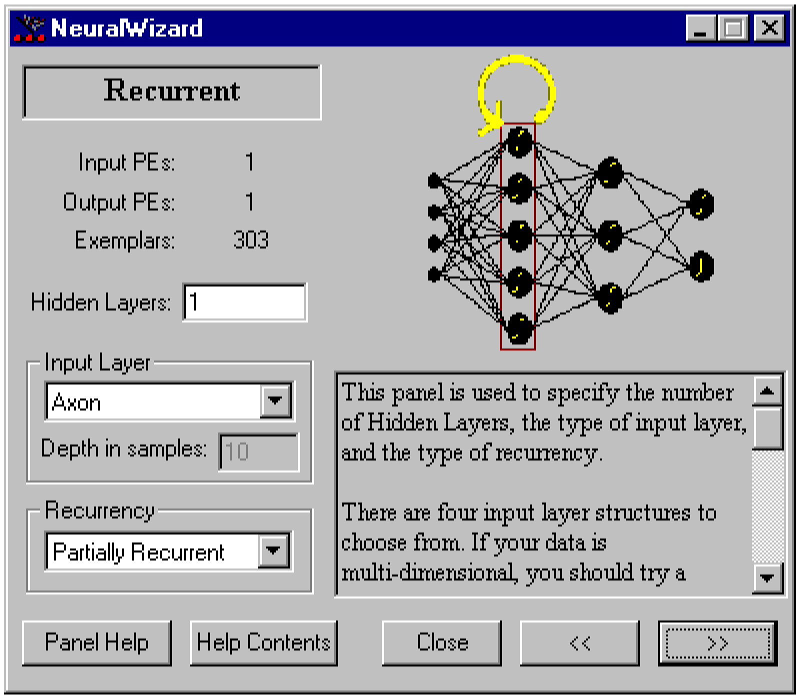
The primary disadvantage of GRN's is that they can become unstable during training. This is because their feedback is adaptable, and they can evolve to an unstable non-linear pole. This distinguishes them from Jordan and Elman Networks, where the feedback is fixed. Also, Bengio showed that GRN's have trouble learning long term relationships because the gradient over time in the back propagation through time (BPTT) algorithm decays exponentially. The BPTT algorithm is also quite complex and requires a lot of memory. NeuroSolutions hides most of the complexity from the user and provides an efficient implementation of this algorithm.

Alternative topologies

Jordan and Elman Networks and Time Lagged Recurrent Networks

Configuration

The General Recurrent panel is used to specify the size and type of the recurrent structure to implement (see figure below). The number of Hidden Layers is also configured from this panel.



Input PEs: 1
Output PEs: 1
Exemplars: 303
Hidden Layers: 1
Input Layer: Axon
Depth in samples: 10
Recurrency: Partially Recurrent

This panel is used to specify the number of Hidden Layers, the type of input layer, and the type of recurrency.

There are four input layer structures to choose from. If your data is multi-dimensional, you should try a

General recurrent panel of the NeuralBuilder

The table below summarizes the two structures available. The fully recurrent network utilizes full feedback to itself of the axon layer immediately following the input layer. The partially recurrent network includes the fully recurrent structure, but adds a synapse connection from the input layer to the layer immediately following the recurrent layer.

Recurrent network types

Type	Description
Fully Recurrent	full state feedback
Partially Recurrent	partial state feedback

Recurrent networks must be trained using a dynamic learning algorithm. NeuroSolutions uses the so-called backpropagation through time (BPTT) algorithm, which is implemented by the dynamic controllers.

General Recurrent Hints
General Recurrent Theoretical Summary

General Recurrent Hints

Start with a partially recurrent network, and use a linear axon for the layer immediately following the recurrent layer. Our own experiments have indicated that this architecture is much less likely to go unstable during training. Recurrent networks also often go unstable when they are overtrained. Once the mean square error has dropped to an acceptable level, stop the training.

If learning seems slow, go to the dynamic controllers, switch to on-line learning, and set the number of Samples/Exemplar (both forward and backward) to a smaller number. Note, however, that this number must evenly divide into the total number of Exemplars.

General Recurrent Theoretical Summary

Recurrent networks are fundamentally different from feed-forward networks. Even with a constant input, they do not necessarily settle to a constant output. They can exhibit limit cycles and even chaotic behavior. Nevertheless, Li [1992] showed that GRN's are a universal approximator of a differentiable trajectory. This means that given a sufficient number of neurons, GRN's can capture the dynamics imbedded within any signal.

CANFIS Networks (Fuzzy Logic)

CANFIS stands for coactive neuro-fuzzy inference systems. This model was proposed by J.-S. R. Jang, C.-T. Sun, and E. Mizutani in Neuro-Fuzzy and Soft Computing, published by Prentice Hall in 1997. The CANFIS model integrates fuzzy inputs with a neural network to quickly solve poorly defined problems. Fuzzy inference systems are also valuable as they combine the explanatory nature of rules (membership functions) with the power of "black box" neural networks.

Advantages

Using fuzzy rules as a preprocessor to a neural network allows one to incorporate human knowledge to perform inferencing and decision making. The CANFIS model optimizes the fuzzy rules (membership function parameters) with backpropagation, so the human knowledge is not required. This "fuzzification" makes the neural network's job easier by characterizing inputs that are not easily discretized, often resulting in a better overall model.

Disadvantages

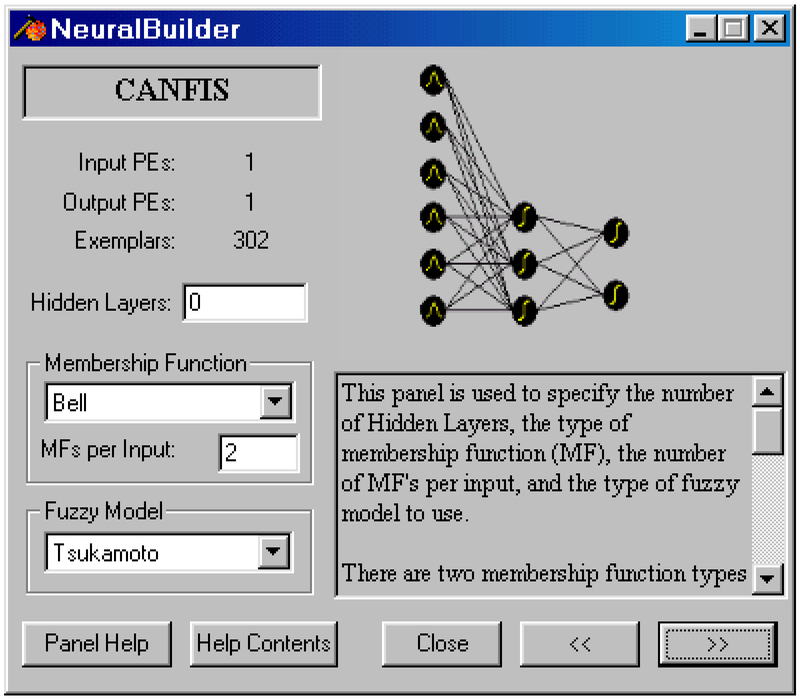
The CANFIS model is more computationally intensive than most other models.

Alternative topologies

Radial Basis Function Networks

Configuration

The CANFIS panel is used to specify the type of fuzzy membership function, the number of functions and the variant of the CANFIS model. The number of Hidden Layers is also configured from this panel.



Input PEs: 1
Output PEs: 1
Exemplars: 302
Hidden Layers: 0
Membership Function: Bell
MFs per Input: 2
Fuzzy Model: Tsukamoto

This panel is used to specify the number of Hidden Layers, the type of membership function (MF), the number of MF's per input, and the type of fuzzy model to use.

There are two membership function types

CANFIS panel of the NeuralBuilder

The two fuzzy membership functions available are the bell-shaped curve and the gaussian-shaped curve. The bell curve is a little more flexible, since it has 3 free parameters to adjust, versus two parameters for the gaussian.

The "MFs per Input" field specifies the number of membership functions assigned to each network input. These membership functions are then combined together to perform the inferencing operation. For small to medium-sized data sets, this number will generally be between 2 and 4.

There are two variants of CANFIS networks to choose from. The Tsukamoto fuzzy model is simpler and runs faster, but the TSK fuzzy model (also known as the Sugeno fuzzy model) is generally more popular.

CANFIS Hints
CANFIS Theoretical Summary

CANFIS Hints

Start with a small number of membership functions per input, and then rebuild the network with a larger number to see if the results improve. It is generally best to configure this parameter within the NeuralBuilder rather than try to change the value after the network is built, since there are dependent parameters in other components that should also be changed.

CANFIS Theoretical Summary

The theory behind fuzzy inference systems and the CANFIS model is too in depth to cover in this documentation. Those interested should read Neuro-Fuzzy and Soft Computing by J.-S. R. Jang, C.-T. Sun, and E. Mizutani.

Support Vector Machine (SVM)

In NeuroSolutions, Support Vector Machines (SVMs) are implemented using the kernel Adatron algorithm. The kernel Adatron maps inputs to a high-dimensional feature space, and then optimally separates data into their respective classes by isolating those inputs that fall close to the data boundaries. Therefore, the kernel Adatron is especially effective in separating sets of data that share complex boundaries. SVMs are generally only useful for classification problems.

Advantages

SVMs have produced excellent results in many practical classification problems.

Disadvantages

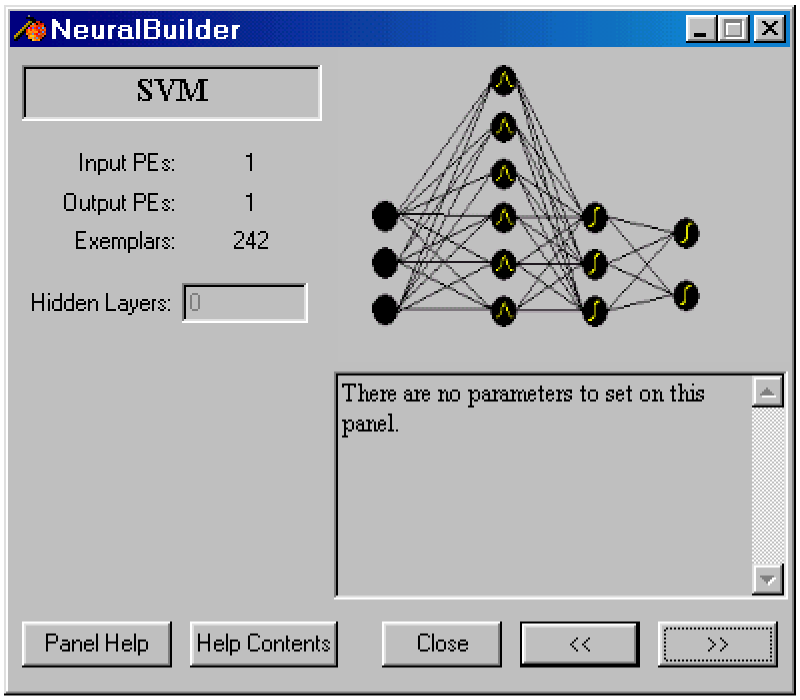
SVMs assign one Gaussian function for each input exemplar in the training set. This makes SVMs impractical for large training sets (greater than 1000 exemplars).

Alternative topologies

Radial Basis Function Networks

Configuration

There are no parameters specific to the SVM that need to be configured.


NeuralBuilder

SVM

Input PEs: 1
Output PEs: 1
Exemplars: 242

Hidden Layers: 0

There are no parameters to set on this panel.

SVM panel of the NeuralBuilder

Support Vector Machine Hints
Support Vector Machine Summary

Support Vector Machine Hints

Since SVMs are used primarily for classification, you will likely want to include a confusion matrix probe to display the classification performance during training. If the network is training too slowly, you can change the step size of the SVMStep gradient search component. You may also want to experiment with various training set sizes, since this will adjust the size of the network (by adjusting the number of PEs in the GaussianAxon).

Support Vector Machine Theoretical Summary

The SVMInputSynapse and GaussianAxon components apply a gaussian of appropriate width at each input, which essentially projects the inputs into a high-dimensional feature space. The kernel Adatron is then able to identify the boundary inputs (referred to as support vectors), which are the important features for classification. The SVMOutputSynapse, SVMStep, and SVML2Criterion components train the alpha and bias values corresponding to each gaussian in order to determine the support vectors. These support vectors allow the network to rapidly converge on the data boundaries and consequently classify the inputs.

More extensive theoretical coverage of Support Vector Machines can be found in section 5.8 (pp. 261-269) of Neural and Adaptive Systems: Fundamentals Through Simulations, by Principe, Euliano, and Lefebvre (John Wiley & Sons, 2000).

Summary of NeuralBuilder Examples

These experiments used different neural models to solve the same problem using the same data. This provides a way to compare the differences in network behavior. The table below contains the summary of results from the experiments. Keep in mind that this is not a scientific comparison. Only one or two simulations were run for each experiment and the experiments themselves were not necessarily balanced (i.e., different number of PEs and different stop criteria). Your results will likely differ due to the unique values of the initial weights.

Summary of experimental results from examples

Neural Model	Training MSE	Testing MSE	Supervised Epochs
MLP	0.07	0.27	150
General. FF	0.10	0.30	175
Jordan	0.07	0.19	175
PCA	0.07	0.22	350
RBF	0.10	0.28	200
SOFM	0.07	0.24	100

It is interesting to note that the MLP, Jordan, PCA, and SOFM achieved the best training set performance, but that the Jordan outperformed the other three for the cross validation set. This is illustrative of the fact that what is learned in the training set is only part of the story in neural network applications.

The Jordan network is the only model of the six that uses time information. In sleep staging there is information contained in the sleep stage transitions that none of the other models can capture. Topologies with memory can be very useful when dealing with this type of real world data.

The second best cross validation set performer was the PCA network. The reason that the PCA generalizes well is that it uses all the information from the input to derive new projections of the data. These projections capture general information about the data clusters. There are other cases when PCA does not perform as well, such as when classes are intermixed.

There are also other comparisons that could have been made. For instance, the size of the network will often affect the results. A larger number of PEs may improve the training set performance but degrade the cross validation set performance due to the lack of generalization. Many other comparisons could be drawn, but the goal of these examples is to illustrate the use of the NeuralBuilder and the care that one should take when making parameter choices.

Remember that this comparison is only for one isolated data set. A neural model that performs poorly on this data set may outperform the others given a different problem. This comparison has shown that neural models that contain memory are best suited for problems that contain temporal information.

These examples have only provided a brief overview of the power available within NeuroSolutions. It is suggested that the interested user work through the Tutorials chapter of the NeuroSolutions on-line documentation, where a much more in depth coverage of the components is given. Knowledge of the package at the component level is fundamental to making appropriate component and parameter selections.

{ewl RoboEx32.dll, WinHelp2000, }