Release Notes for Neural Network Toolbox™

How to Contact MathWorks

`www.mathworks.com`	Web
`comp.soft-sys.matlab`	Newsgroup
`www.mathworks.com/contact_TS.html`	Technical Support

`suggest@mathworks.com`	Product enhancement suggestions
`bugs@mathworks.com`	Bug reports
`doc@mathworks.com`	Documentation error reports
`service@mathworks.com`	Order status, license renewals, passcodes
`info@mathworks.com`	Sales, pricing, and general information

508-647-7000 (Phone)

508-647-7001 (Fax)

The MathWorks, Inc.
3 Apple Hill Drive
Natick, MA 01760-2098

For contact information about worldwide offices, see the MathWorks Web site.

Release Notes for Neural Network Toolbox™

Contents

R2013a

R2012b

R2012a

R2011b

R2011a

R2010b

R2010a

R2009b

R2009a

R2008b

R2008a

R2007b

R2007a

No New Features or Changes

R2006b

No New Features or Changes

R2006a

R14SP3

No New Features or Changes

R2013a

Version: 8.0.1
New Features: No
Bug Fixes: Yes

R2012b

Version: 8.0
New Features: Yes
Bug Fixes: Yes

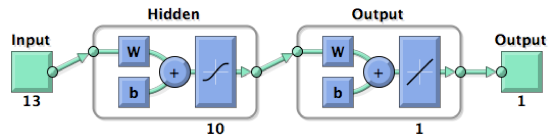
Speed and memory efficiency enhancements for neural network training and simulation

Compatibility Considerations: Yes

The neural network simulation, gradient, and Jacobian calculations are reimplemented with native MEX-functions in Neural Network Toolbox™ Version 8.0. This results in faster speeds, especially for small to medium network sizes, and for long time-series problems.

In Version 7, typical code for training and simulating a feed-forward neural network looks like this:

```
[x,t] = house_dataset;
net = feedforwardnet(10);
view(net)
net = train(net,x,t);
y = net(x);
```



In Version 8.0, the above code does not need to be changed, but calculations now happen in compiled native MEX code.

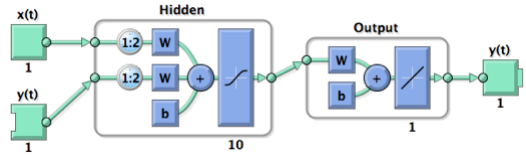
Speedups of as much as 25% over Version 7.0 have been seen on a sample system (4-core 2.8 GHz Intel i7 with 12 GB RAM).

Note that speed improvements measured on the sample system might vary significantly from improvements measured on other systems due to different chip speeds, memory bandwidth, and other hardware and software variations.

The following code creates, views, and trains a dynamic NARX neural network model of a maglev system in open-loop mode.

```
[x,t] = maglev_dataset;
net = narxnet(1:2,1:2,10);
view(net)
[X,Xi,Ai,T] = preparets(net,x,{},t);
```

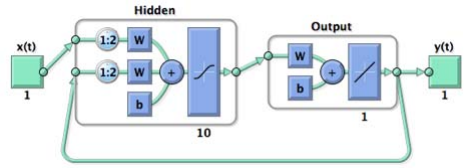
```
net = train(net,X,T,Xi,Ai);
y = net(X,Xi,Ai)
```



The following code measures training speed over 10 training sessions, with the training window disabled to avoid GUI timing interference.

On the sample system, this ran three times (3x) faster in Version 8.0 than in Version 7.0.

```
rng(0)
[x,t] = maglev_dataset;
net = narxnet(1:2,1:2,10);
[X,Xi,Ai,T] = preparets(net,x,{},t);
net.trainParam.showWindow = false;
tic
for i=1:10
  net = train(net,X,T,Xi,Ai);
end
toc
```

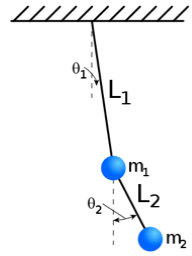


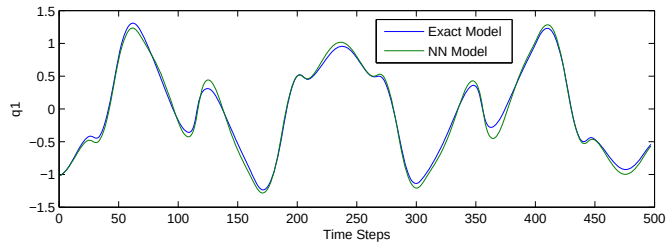
The following code trains the network in closed-loop mode:

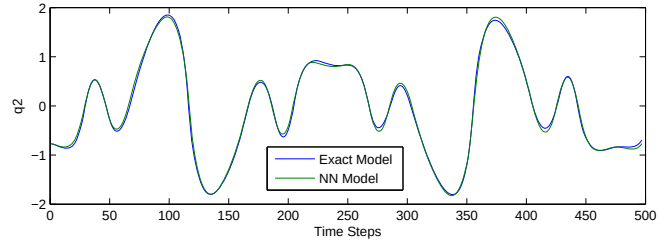
```
[x,t] = maglev_dataset;
net = narxnet(1:2,1:2,10);
net = closeloop(net);
view(net)
[X,Xi,Ai,T] = preparets(net,x,{},t);
net = train(net,X,T,Xi,Ai);
```

For this case, and most closed-loop (recurrent) network training, Version 8.0 ran the code more than one-hundred times (100x) faster than Version 7.0.

A dramatic example of where the improved closed loop training speed can help is when training a NARX network model of a double pendulum. By initially training the network in open-loop mode, then in closed-loop mode with two time step sequences, then three time step sequences, etc., a network has been trained that can simulate the system for 500 time steps in closed-loop mode. This corresponds to a 500 step ahead prediction.







Because of the Version 8.0 MEX speedup, this only took a few hours, as apposed to the months it would have taken in Version 7.0.

MEX code is also far more memory efficient. The amount of RAM used for intermediate variables during training and simulation is now relatively constant, instead of growing linearly with the number of samples. In other words, a problem with 10,000 samples requires the same temporary storage as a problem with only 100 samples.

This memory efficiency means larger problems can be trained on a single computer.

Compatibility Considerations

For very large networks, MEX code might fall back to MATLAB® code. If this happens and memory availability becomes an issue, use the `'reduction'` option to implement memory reduction. The reduction number indicates the number of passes to make through the data for each calculation. Each pass calculates with a fraction of the data, and the results are combined after all passes are complete. This trades off lower memory requirements for longer calculation times.

```
net = train(net,x,t,'reduction',10);
y = net(x,'reduction',10);
```

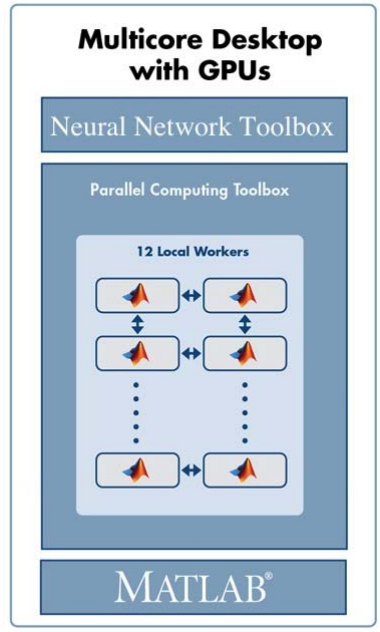
The previous way to indicate memory reduction was to set the `net.efficiency.memoryReduction` property before training:

```
net.efficiency.memoryReduction = N;
```

This continues to work in Version 8.0, but it is recommended that you update your code to use the `'reduction'` option for train and network simulation. Additional name-value pair arguments are the standard way to indicate calculation options.

Speedup of training and simulation with multicore processors and computer clusters using Parallel Computing Toolbox

Parallel Computing Toolbox™ allows Neural Network Toolbox simulation, and gradient and Jacobian calculations to be parallelized across multiple CPU cores, reducing calculation times. Parallelization splits the data among several workers. Results for the whole dataset are combined after all workers have completed their calculations.



Note that, during training, the calculation of network outputs, performance, gradient, and Jacobian calculations are parallelized, while the main training code remains on one worker.

To train a network on the `house_dataset` problem, introduced above, open a local MATLAB pool of workers, then call `train` and `sim` with the new `'useParallel'` option set to `'yes'`.

```
matlabpool open
numWorkers = matlabpool('size')
```

If calling `matlabpool` produces an error, it might be that Parallel Computing Toolbox is not available.

```
[x,t] = house_dataset;
net = feedforwardnet(10);
net = train(net,x,t,'useParallel','yes');
y = sim(net,'useParallel','yes');
```

On the sample system with a pool of four cores, typical speedups have been between 3x and 3.7x. Using more than four cores might produce faster speeds. For more information, see “Parallel and GPU Computing”.

GPU computing support for training and simulation on single and multiple GPUs using Parallel Computing Toolbox

Parallel Computing Toolbox allows Neural Network Toolbox simulation and training to be parallelized across the multiprocessors and cores of a graphics processing unit (GPU).

To train and simulate with a GPU set the `'useGPU'` option to `'yes'`. Use the gpuDevice command to get information on your GPU.

```
gpuInfo = gpuDevice
```

If calling `gpuDevice` produces an error, it might be that Parallel Computing Toolbox is not available.

Training on GPUs cannot be done with Jacobian algorithms, such as `trainlm` or `trainbr`, but it can be done with any of the gradient algorithms such as `trainscg`. If you do not change the training function, it will happen automatically.

```
[x,t] = house_dataset;
net = feedforwardnet(10);
net.trainFcn = 'trainscg';
net = train(net,x,t,'useGPU','yes');
y = sim(net,'useGPU','yes');
```

Speedups on the sample system with an nVidia GTX 470 GPU card have been between 3x and 7x, but might increase as GPUs continue to improve.

You can also use multiple GPUs. If you set both `'useParallel'` and `'useGPU'` to `'yes'`, any worker associated with a unique GPU will use that GPU, and other workers will use their CPU core. It is not efficient to share GPUs between workers, as that would require them to perform their calculations in sequence instead of in parallel.

```
numWorkers = matlabpool('size')
numGPUs = gpuDeviceCount

[x,t] = house_dataset;
```

```
net = feedforwardnet(10);
net.trainFcn = 'trainscg';
net = train(net,x,t,'useParallel','yes','useGPU','yes');
y = sim(net,'useParallel','yes','useGPU','yes');
```

Tests with three GPU workers and one CPU worker on the sample system have seen 3x or higher speedup. Depending on the size of the problem, and how much it uses the capacity of each GPU, adding GPUs might increase speed or might simply increase the size of problem that can be run.

In some cases, training with both GPUs and CPUs can result in slower speeds than just training with the GPUs, because the CPUs might not keep up with the GPUs. In this case, set `'useGPU'` to `'only'` and only GPU workers will be used.

```
[x,t] = house_dataset;
net = feedforwardnet(10);
net = train(net,x,t,'useParallel','yes','useGPU','only');
y = sim(net,'useParallel','yes','useGPU','only');
```

For more information, see “Parallel and GPU Computing”.

Distributed training of large datasets on computer clusters using MATLAB Distributed Computing Server

Besides allowing load balancing, Composite data also allows datasets too large to fit within the RAM of a single computer to be distributed across the RAM of a cluster.

This is done by loading the Composite sequentially. For instance, here the sub-datasets are loaded from files as they are distributed:

```
Xc = Composite;
Tc = Composite;
for i=1:10
  data = load(['dataset' num2str(i)])
  Xc{i} = data.x;
  Tc{i} = data.t;
  clear data
end
```

This technique allows for training with datasets of any size, limited only by the available RAM across an entire cluster.

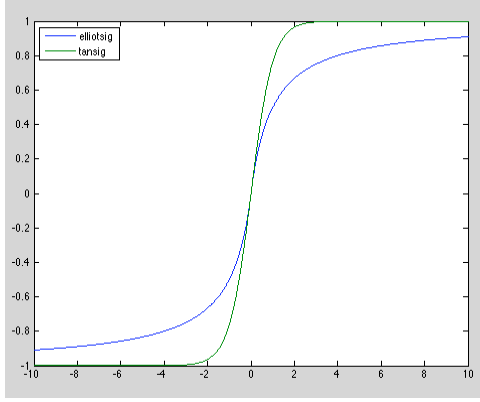
For more information, see “Parallel and GPU Computing”.

Elliot sigmoid transfer function for faster simulation

The new transfer function `elliotsig` calculates its output without using the `exp` function used by both `tansig` and `logsig`. This lets it execute much faster, especially on deployment hardware that might either not support `exp` or which implements it with software that takes many more execution cycles than simple arithmetic operations.

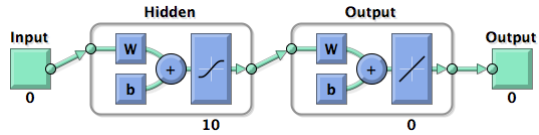
This example displays a plot of `elliotsig` alongside `tansig`:

```
n = -10:0.01:10;
a1 = elliotsig(n);
a2 = tansig(n);
h = plot(n,a1,n,a2);
legend(h,'ELLIOTSIG','TANSIG','Location','NorthWest')
```

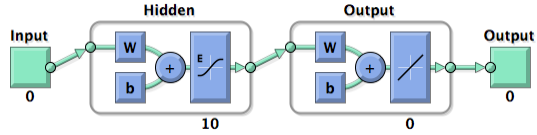


To set up a neural network to use the `elliotsig` transfer function, change each `tansig` layer's transfer function with its `transferFcn` property. For instance, here a network using `elliotsig` is created, viewed, trained, and simulated:

```
[x,t] = house_dataset;
net = feedforwardnet(10);
view(net) % View TANSIG network
```




```
net.layers{1}.transferFcn = 'elliotsig';
view(net) % View ELLIOTSIG network
```




```
net = train(net,x,t);
y = net(x)
```

The `elliotsig` transfer function might be even faster on an Intel® processor.

```
n = rand(1000,1000);
tic, for i=1:100, a = elliotsig(n); end, elliotsigTime = toc
tic, for i=1:100, a = tansig(n); end, tansigTime = toc
speedup = tansigTime / elliotsigTime
```

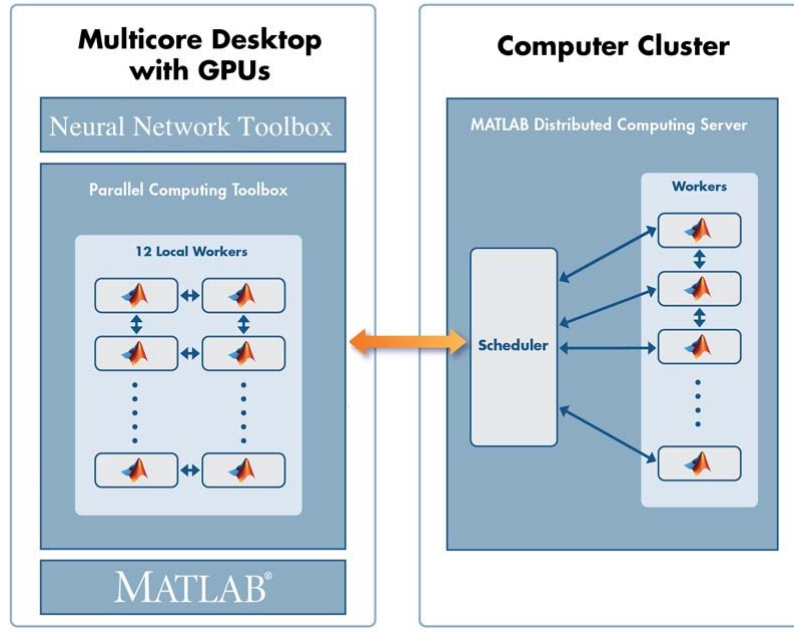
On one system the speedup was almost 3x.

However, because of the different shape, `elliotsig` might not result in faster training than `tansig`. It might require more training steps. For simulation, `elliotsig` is always faster.

For more information, see “Fast Elliot Sigmoid”.

Faster training and simulation with computer clusters using MATLAB Distributed Computing Server

If a MATLAB pool is opened using a cluster of computers, the previous parallel training and simulations happen across the CPU cores and GPUs of all the computers in the pool. For problems with hundreds of thousands or millions of samples, this might result in considerable speedup.



For more information, see “Parallel and GPU Computing”.

Load balancing parallel calculations

When training and simulating a network using the `'useParallel'` option, the dataset is automatically divided into equal parts across the workers. However, if different workers have different speed and memory limitations, it can be helpful to adjust the amount of data sent to each worker, so that the faster workers or those with more memory have proportionally more data.

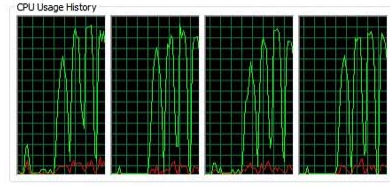
This is done using the Parallel Computing Toolbox function `Composite`. Composite data is data spread across a MATLAB pool of workers.

For instance, if a MATLAB pool is open with four workers, data can be distributed as follows:

```
[x,t] = house_dataset;
Xc = Composite;
Tc = Composite;
Xc{1} = x(:,   1:150); % First  150 samples of x
Tc{1} = x(:,   1:150); % First  150 samples of t
Xc{2} = x(:, 151:300); % Second 150 samples of x
Tc{2} = x(:, 151:300); % Second 150 samples of t
Xc{3} = x(:, 301:403); % Third  103 samples of x
Tc{3} = x(:, 301:403); % Third  103 samples of t
Xc{4} = x(:, 404:506); % Fourth 103 samples of x
Tc{4} = x(:, 404:506); % Fourth 103 samples of t
```

When `train` is called, the `'useParallel'` option is not needed, as `train` automatically trains in parallel upon getting the Composite data.

```
net = train(net,Xc,Tc);
```



If you want workers 1 and 2 to use GPU devices 1 and 2, while workers 3 and 4 use CPUs, set up data for workers 1 and 2 using `nndata2gpu` inside an `spmd` clause.

```
spmd
  if labindex <= 2
    Xc = nndata2gpu(Xc);
    Tc = nndata2gpu(Tc);
  end
end
```

The function `nndata2gpu` takes a neural network matrix or cell array time series data and converts it to a properly sized gpuArray on the worker's GPU. This involves transposing the matrices, padding the columns so their first elements are memory aligned, and combining matrices, if the data was a cell array of matrices. To reverse process outputs returned after simulation with gpuArray data, use `gpu2nndata` to convert back to a regular matrix or a cell array of matrices.

As with `'useParallel'`, the data type removes the need to specify `'useGPU'`. Training and simulation automatically recognize that two of the workers have gpuArray data and employ their GPUs accordingly.

```
net = train(net,Xc,Tc);
```

This way, any variation in speed or memory limitations between workers can be accounted for by putting differing numbers of samples on those workers.

For more information, see "Parallel and GPU Computing".

Summary and fallback rules of computing resources used from train and sim

The convention used for computing resources requested by options `'useParallel'` and `'useGPU'` is that if the resource is available it will be used. If it is not, calculations still occur accurately, but without that resource. Specifically:

1. If `'useParallel'` is set to `'yes'`, but no MATLAB pool is open, then computing occurs in the main MATLAB thread and is not distributed across workers.
2. If `'useGPU'` is set to `'yes'`, but there is not a suppported GPU device selected, then computing occurs on the CPU.
3. If `'useParallel'` and `'useGPU'` are set to `'yes'`, each worker uses a GPU if it is the first worker with a particular supported GPU selected, or uses a CPU core otherwise.
4. If `'useParallel'` is set to `'yes'` and `'useGPU'` is set to `'only'`, then only the first worker with a supported GPU is used, and other workers are not used. However, if no GPUs are available, calculations revert to parallel CPU cores.

Set the `'showResources'` option to `'yes'` to check what resources are actually being used, as opposed to requested for use, when training and simulating.

```
[x,t] = house_dataset;
net = feedforwardnet(10);

net2 = train(net,x,t,'showResources','yes');
y = net2(x,'showResources','yes');

Computing Resources:
MEX on PCWIN64

net2 = train(net,x,t,'useParallel','yes','showResources','yes');
y = net2(x,'useParallel','yes','showResources','yes');

Computing Resources:
```

```
  Worker 1 on Computer1, MEX on PCWIN64
  Worker 2 on Computer1, MEX on PCWIN64
  Worker 3 on Computer1, MEX on PCWIN64
  Worker 4 on Computer1, MEX on PCWIN64

net2 = train(net,x,t,'useGPU','yes','showResources','yes');
y = net2(x,'useGPU','yes','showResources','yes');

Computing Resources:
GPU device 1, TypeOfCard

net2 = train(net,x,t,'useParallel','yes','useGPU','yes',...
                                                'showResources','yes');
y = net2(x,'useParallel','yes','useGPU','yes','showResources','yes');

Computing Resources:
  Worker 1 on Computer1, GPU device 1, TypeOfCard
  Worker 2 on Computer1, GPU device 2, TypeOfCard
  Worker 3 on Computer1, MEX on PCWIN64
  Worker 4 on Computer1, MEX on PCWIN64

net2 = train(net,x,t,'useParallel','yes','useGPU','only',...
                                                 'showResources','yes');
y = net2(x,'useParallel','yes','useGPU','only','showResources','yes');

Computing Resources:
  Worker 1 on Computer1, GPU device 1, TypeOfCard
  Worker 2 on Computer1, GPU device 2, TypeOfCard
```

Updated code organization

Compatibility Considerations: Yes

The code organization for data processing, weight, net input, transfer, performance, distance and training functions are updated. Custom functions of these kinds need to be updated to the new organization.

In Version 8.0 the related functions for neural network processing are in package folders, so each local function has its own file.

For instance, in Version 7.0 the function `tansig` contained a large switch statement and several local functions. In Version 8.0 there is a root function `tansig`, along with several package functions in the folder `/toolbox/nnet/nnet/nntransfer/+tansig/`.

```
+tansig/activeInputRange.m
+tansig/apply.m
+tansig/backprop.m
+tansig/da_dn.m
+tansig/discontinuity.m
+tansig/forwardprop.m
+tansig/isScalar.m
+tansig/name.m
+tansig/outputRange.m
+tansig/parameterInfo.m
+tansig/simulinkParameters.m
+tansig/type.m
```

Each transfer function has its own package with the same set of package functions. For lists of processing, weight, net input, transfer, performance, and distance functions, each of which has its own package, type the following:

```
help nnprocess
help nnweight
help nnnetinput
help nntransfer
help nnperformance
help nndistance
```

The calling interfaces for training functions are updated for the new calculation modes and parallel support. Normally, training functions would

not be called directly, but indirectly by train, so this is unlikely to require any code changes.

Compatibility Considerations

Due to the new package organization for processing, weight, net input, transfer, performance and distance functions, any custom functions of these types will need to be updated to conform to this new package system before they will work with Version 8.0.

See the main functions and package functions for `mapminmax`, `dotprod`, `netsum`, `tansig`, `mse`, and `dist` for examples of this new organization. Any of these functions and its package functions may be used as a template for new or updated custom functions.

Due to the new calling interfaces for training functions, any custom backpropagation training function will need to be updated to work with Version 8.0. See `trainlm` and `trainscg` for examples that can be used as templates for any new or updated custom training function.

R2012a

Version: 7.0.3
New Features: No
Bug Fixes: Yes

R2011b

Version: 7.0.2
New Features: No
Bug Fixes: Yes

R2011a

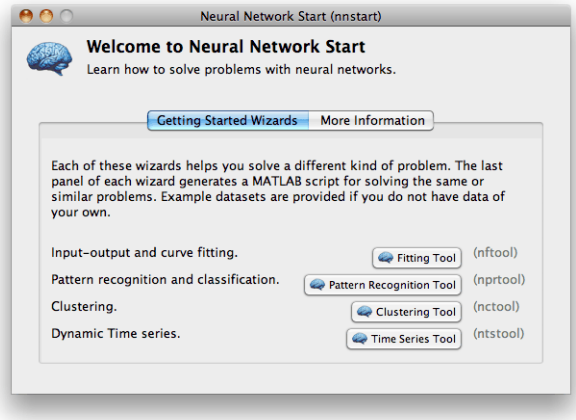
Version: 7.0.1
New Features: No
Bug Fixes: Yes

R2010b

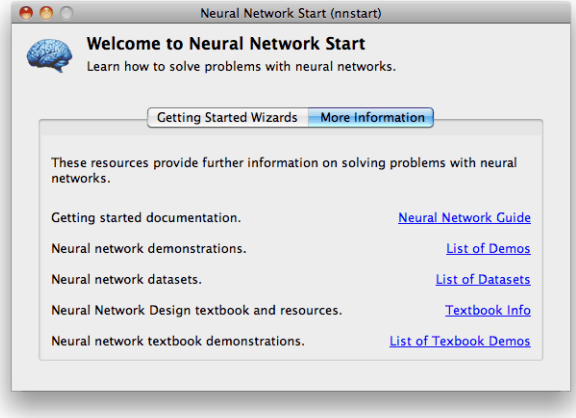
Version: 7.0
New Features: Yes
Bug Fixes: Yes

New Neural Network Start GUI

The new `nnstart` function opens a GUI that provides links to new and existing Neural Network Toolbox GUIs and other resources. The first panel of the GUI opens four "getting started" wizards.

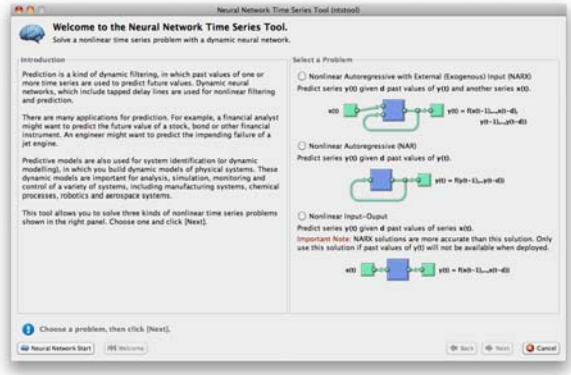


The second panel provides links to other toolbox starting points.

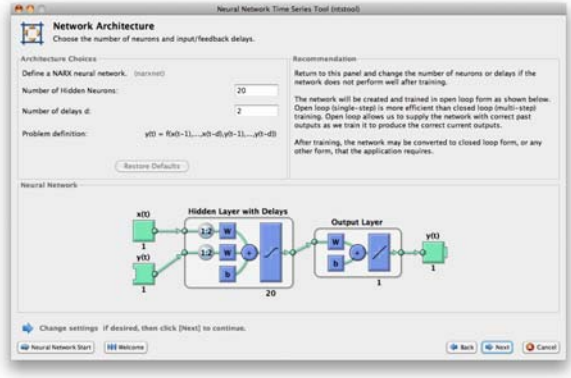


New Time Series GUI and Tools

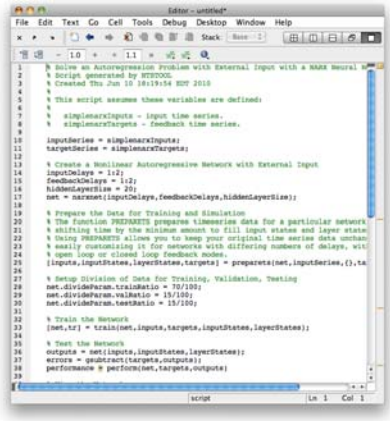
The new `ntstool` function opens a wizard GUI that allows time series problems to be solved with three kinds of neural networks: NARX networks (neural auto-regressive with external input), NAR networks (neural auto-regressive), and time delay neural networks. It follows a similar format to the neural fitting (`nftool`), clustering (`nctool`), and pattern recognition (`nprtool`) tools.



Network diagrams shown in the Neural Time Series Tool, Neural Training Tool, and with the `view(net)` command, have been improved to show tap delay lines in front of weights, the sizes of inputs, layers and outputs, and the time relationship of inputs and outputs. Open loop feedback outputs and inputs are indicated with matching tab and indents in their respective blocks.



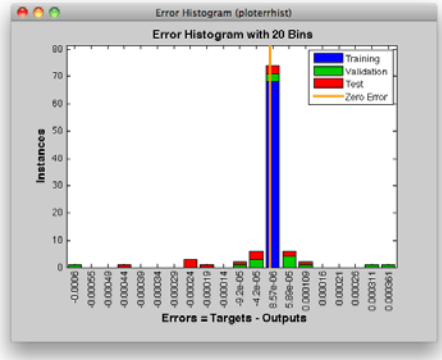
The Save Results panel of the Neural Network Time Series Tool allows you to generate both a Simple Script, which demonstrates how to get the same results as were obtained with the wizard, and an Advanced Script, which provides an introduction to more advanced techniques.



The Train Network panel of the Neural Network Time Series Tool introduces four new plots, which you can also access from the Network Training Tool and the command line.

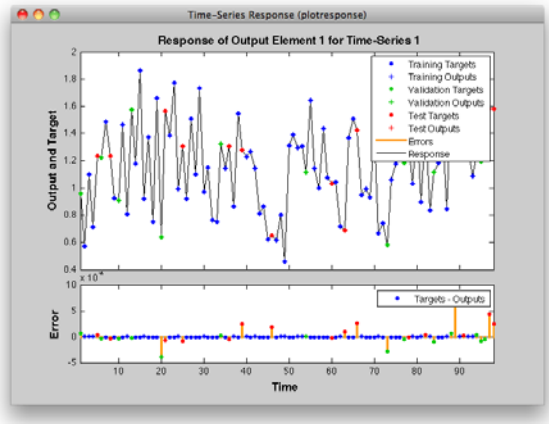
The error histogram of any static or dynamic network can be plotted.

```
plotresponse(errors)
```



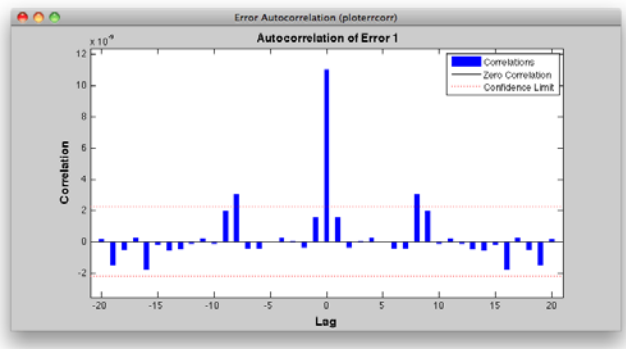
The dynamic response can be plotted, with colors indicating how targets were assigned to training, validation and test sets across timesteps. (Dividing data by timesteps and other criteria, in addition to by sample, is a new feature described in “New Time Series Validation” on page 38.)

```
plotresponse(targets,outputs)
```



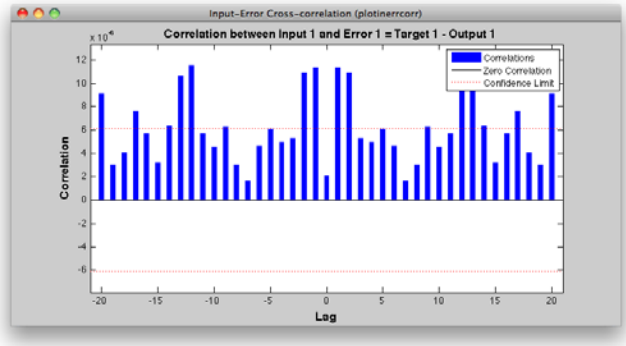
The autocorrelation of error across varying lag times can be plotted.

```
ploterrcorr(errors)
```



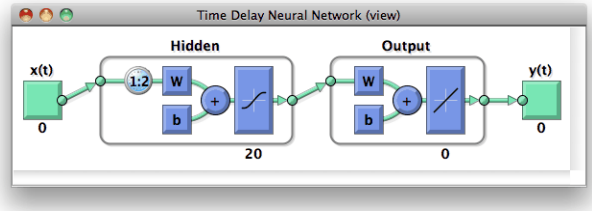
The input-to-error correlation can also be plotted for varying lags.

```
plotinerrcorr(inputs,errors)
```



Simpler time series neural network creation is provided for NARX and time-delay networks, and a new function creates NAR networks. All the network diagrams shown here are generated with the command `view(net)`.

```
net = narxnet(inputDelays, feedbackDelays, hiddenSizes,
feedbackMode, trainingFcn
net = narnet(feedbackDelays, hiddenSizes, feedbackMode,
trainingFcn)
net = timedelaynet(inputDelays, hiddenSizes, trainingFcn)
```



Several new data sets provide sample problems that can be solved with these networks. These data sets are also available within the `ntstool` GUI and the command line.

```
[x, t] = simpleseries_dataset;
[x, t] = simplenarx_dataset;
[x, t] = exchanger_dataset;
[x, t] = maglev_dataset;
[x, t] = ph_dataset;
```

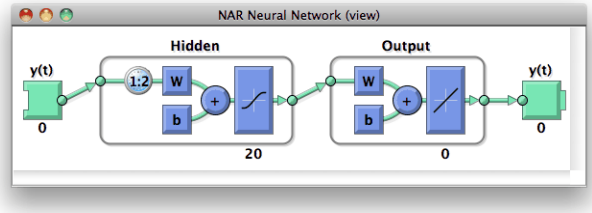
```
[x, t] = pollution_dataset;
[x, t] = refmodel_dataset;
[x, t] = robotarm_dataset;
[x, t] = valve_dataset;
```

The `preparets` function formats input and target time series for time series networks, by shifting the inputs and targets as needed to fill initial input and layer delay states. This function simplifies what is normally a tricky data preparation step that must be customized for details of each kind of network and its number of delays.

```
[x, t] = simplenarx_dataset;
net = narxnet(1:2, 1:2, 10);
[xs, xi, ai, ts] = preparets(net, x, {}, t);
net = train(net, xs, ts, xi, ai);
y = net(xs, xi, ai)
```

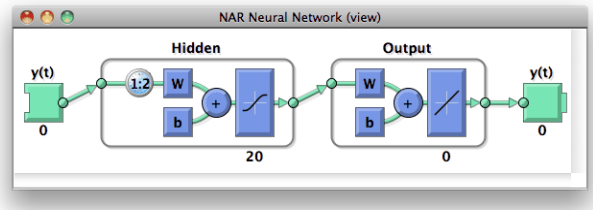
The output-to-input feedback of NARX and NAR networks (or custom time series network with output-to-input feedback loops) can be converted between open- and closed-loop modes using the two new functions `closeloop` and `openloop`.

```
net = narxnet(1:2, 1:2, 10);
net = closeloop(net)
net = openloop(net)
```



The total delay through a network can be adjusted with the two new functions `removedelay` and `adddelay`. Removing a delay from a NARX network which has a minimum input and feedback delay of 1, so that it now has a minimum delay of 0, allows the network to predict the next target value a timestep ahead of when that value is expected.

```
net = removedelay(net)
net = adddelay(net)
```



The new function `catsamples` allows you to combine multiple time series into a single neural network data variable. This is useful for creating input and target data from multiple input and target time series.

```
x = catsamples(x1, x2, x3);
t = catsamples(t1, t2, t3);
```

In the case where the time series are not the same length, the shorter time series can be padded with NaN values. This will indicate “don’t care” or equivalently “don’t know” input and targets, and will have no effect during simulation and training.

```
x = catsamples(x1, x2, x3, 'pad')
t = catsamples(t1, t2, t3, 'pad')
```

Alternatively, the shorter series can be padded with any other value, such as zero.

```
x = catsamples(x1, x2, x3, 'pad', 0)
```

There are many other new and updated functions for handling neural network data, which make it easier to manipulate neural network time series data.

```
help nndatafun
```

New Time Series Validation

Normally during training, a data set's targets are divided up by sample into training, validation and test sets. This allows the validation set to stop training at a point of optimal generalization, and the test set to provide an independent measure of the network's accuracy. This mode of dividing up data is now indicated with a new property:

```
net.divideMode = 'sample'
```

However, many time series problems involve only a single time series. In order to support validation you can set the new property to divide data up by timestep. This is the default setting for NARXNET and other time series networks.

```
net.divideMode = 'time'
```

This property can be set manually, and can be used to specify dividing up of targets across both sample and timestep, by all target values (i.e., across sample, timestep, and output element), or not to perform data division at all.

```
net.divideMode = 'sampletime'
net.divideMode = 'all'
net.divideMode = 'none'
```

New Time Series Properties

Time series feedback can also be controlled manually with new network properties that represent output-to-input feedback in open- or closed-loop modes. For open-loop feedback from an output from layer `i` back to input `j`, set these properties as follows:

```
net.inputs{j}.feedbackOutput = i
net.outputs{i}.feedbackInput = j
net.outputs{i}.feedbackMode = 'open'
```

When the feedback mode of the output is set to `'closed'`, the properties change to reflect that the output-to-input feedback is now implemented with internal feedback by removing input `j` from the network, and having output properties as follows:

```
net.outputs{i}.feedbackInput = [];
net.outputs{i}.feedbackMode = 'closed'
```

Another output property keeps track of the proper closed-loop delay, when a network is in open-loop mode. Normally this property has this setting:

```
net.outputs{i}.feedbackDelay = 0
```

However, if a delay is removed from the network, it is updated to 1, to indicate that the network's output is actually one timestep ahead of its inputs, and must be delayed by 1 if it is to be converted to closed-loop form.

```
net.outputs{i}.feedbackDelay = 1
```

New Flexible Error Weighting and Performance

Compatibility Considerations: Yes

Performance functions have a new argument list that supports error weights for indicating which target values are more important than others. The `train` function also supports error weights.

```
net = train(net, x, t, xi, ai, ew)
perf = mse(net, x, t, ew)
```

You can define error weights by sample, output element, time step, or network output:

```
ew = [1.0 0.5 0.7 0.2];      % Weighting errors across 4 samples
ew = [0.1; 0.5; 1.0];        % ... across 3 output elements
ew = {0.1 0.2 0.3 0.5 1.0};  % ... across 5 timesteps
ew = {1.0; 0.5};             % ... across 2 network outputs
```

These can also be defined across any combination. For example, weighting error across two time series (i.e., two samples) over four timesteps:

```
ew = {[0.5 0.4], [0.3 0.5], [1.0 1.0], [0.7 0.5]};
```

In the general case, error weights can have exactly the same dimension as targets, where each target has an associated error weight.

Some performance functions are now obsolete, as their functionality has been implemented as options within the four remaining performance functions: `mse`, `mae`, `sse`, and `sae`.

The regularization implemented in `msereg` and `msnereg` is now implemented with a performance property supported by all four remaining performance functions.

```
% Any value between the default 0 and 1.
net.performParam.regularization
```

The error normalization implemented in `msne` and `msnereg` is now implemented with a normalization property.

```
% Either 'normalized', 'percent', or the default 'none'.
```

```
net.performParam.normalization
```

A third performance parameter indicates whether error weighting is applied to square errors (the default for `mse` and `sse`) or the absolute errors (`mae` and `sae`).

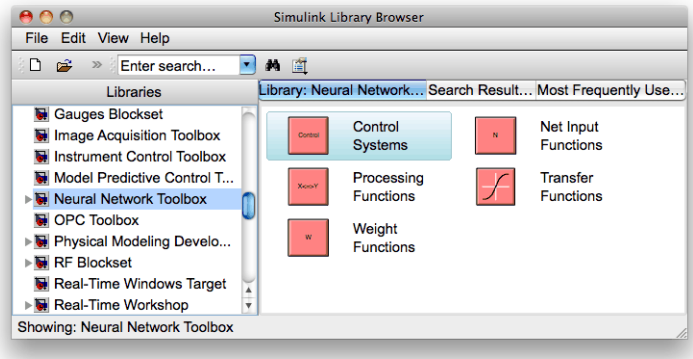
```
net.performParam.squaredWeighting  % true or false
```

Compatibility Considerations

The old performance functions and old performance arguments lists continue to work as before, but are no longer recommended.

New Real Time Workshop and Improved Simulink Support

Neural network Simulink® blocks now compile with Real Time Workshop® and are compatible with Rapid Accelerator mode.

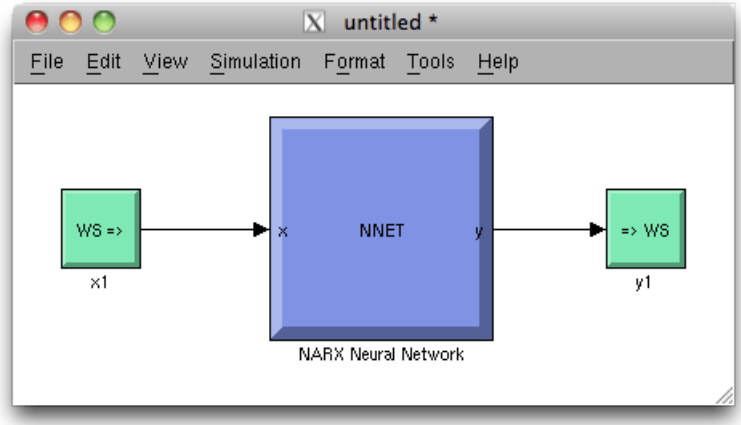


`gensim` has new options for generating neural network systems in Simulink.

```
Name - the system name
SampleTime - the sample time
InputMode - either port, workspace, constant, or none.
OutputMode - either display, port, workspace, scope, or none
SolverMode - either default or discrete
```

For instance, here a NARX network is created and set up in MATLAB to use workspace inputs and outputs.

```
[x, t] = simplenarx_dataset;
net = narxnet(1:2, 1:2, 10);
[xs, xi, ai, ts] = preparets(net, x, {}, t);
net = train(net, xs, ts, xi, ai);
net = closeloop(net);
[sysName, netName] = gensim(net, 'InputMode', 'workspace', ...
          'OutputMode', 'workspace', 'SolverMode', 'discrete');
```



Simulink neural network blocks now allow initial conditions for input and layer delays to be set directly by double-clicking the neural network block. `setsiminit` and `getsiminit` provide command-line control for setting and getting input and layer delays for a neural network Simulink block.

```
setsiminit(sysName, netName, net, xi, ai);
```

New Documentation Organization and Hyperlinks

The User's Guide has been rearranged to better focus on the workflow of practical applications. The Getting Started section has been expanded.

References to functions throughout the online documentation and command-line help now link directly to their function pages.

```
help feedforwardnet
```

The command-line output of neural network objects now contains hyperlinks to documentation. For instance, here a feed-forward network is created and displayed. Its command-line output contains links to network properties, function reference pages, and parameter information.

```
net = feedforwardnet(10);
```

Subobjects of the network, such as inputs, layers, outputs, biases, weights, and parameter lists also display with links.

```
net.inputs{1}
net.layers{1}
net.outputs{2}
net.biases{1}
net.inputWeights{1, 1}
net.trainParam
```

The training tool `nntraintool` and the wizard GUIs `nftool`, `nprtool`, `nctool`, and `ntstool`, provide numerous hyperlinks to documentation.

New Derivative Functions and Property

New functions give convenient access to error gradient (of performance with respect to weights and biases) and Jacobian (of error with respect to weights and biases) calculated by various means.

```
staticderiv - Backpropagation for static networks
bttderiv - Backpropagation through time
fpderiv - Forward propagation
num2deriv - Two-point numerical approximation
num5deriv - Five-point numerical approximation
defaultderiv - Chooses recommended derivative function for the network
```

For instance, here you can calculate the error gradient for a newly created and configured feedforward network.

```
net = feedforwardnet(10);
[x, t] = simplefit_dataset;
net = configure(net, x, t);
d = staticderiv('dperf_dwb', net, x, t)
```

Improved Network Creation

Compatibility Considerations: Yes

New network creation functions have clearer names, no longer need example data, and have argument lists reduced to only the arguments recommended for most applications. All arguments have defaults, so you can create simple networks by calling network functions without any arguments. New networks are also more memory efficient, as they no longer need to store sample input and target data for proper configuration of input and output processing settings.

```
% New function
net = feedforwardnet(hiddenSizes, trainingFcn)

% Old function
net = newff(x,t,hiddenSizes, transferFcns, trainingFcn, ...
      learningFcn, performanceFcn, inputProcessingFcns, ...
      outputProcessingFcns, dataDivisionFcn)
```

The new functions (and the old functions they replace) are:

`feedforwardnet` (`newff`)
`cascadeforwardnet` (`newcf`)
`competlayer` (`newc`)
`distdelaynet` (`newdtdnn`)
`elmannet` (`newelm`)
`fitnet` (`newfit`)
`layrecnet` (`newlrn`)
`linearlayer` (`newlin`)
`lvqnet` (`newlvq`)
`narxnet` (`newnarx`, `newnarxsp`)
`patternnet` (`newpr`)
`perceptron` (`newp`)
`selforgmap` (`newsom`)
`timedelaynet` (`newtdnn`)

The network's inputs and outputs are created with size zero, then configured for data when `train` is called or by optionally calling the new function `configure`.

```
net = configure(net, x, t)
```

Unconfigured networks can be saved and reused by configuring them for many different problems. `unconfigure` sets a configured network's inputs and outputs to zero, in a network which can later be configured for other data.

```
net = unconfigure(net)
```

Compatibility Considerations

Old functions continue working as before, but are no longer recommended.

Improved GUIs

The neural fitting `nftool`, pattern recognition `nprtool`, and clustering `nctool` GUIs have been updated with links back to the `nnstart` GUI. They give the option of generating either simple or advanced scripts in their last panel. They also confirm with you when closing, if a script has not been generated, or the results not yet saved.

Improved Memory Efficiency

Compatibility Considerations: Yes

Memory reduction, the technique of splitting calculations up in time to reduce memory requirements, has been implemented across all training algorithms for both gradient and network simulation calculations. Previously it was only supported for gradient calculations with `trainlm` and `trainbr`.

To set the memory reduction level, use this new property. The default is 1, for no memory reduction. Setting it to 2 or higher splits the calculations into that many parts.

```
net.efficiency.memoryReduction
```

Compatibility Considerations

The `trainlm` and `trainbr` training parameter MEM_REDUC is now obsolete. References to it will need to be updated. Code referring to it will generate a warning.

Improved Data Sets

All data sets in the toolbox now have help, including example solutions, and can be accessed as functions:

```
help simplefit_dataset
[x, t] = simplefit_dataset;
```

See help for a full list of sample data sets:

```
help nndatasets
```

Updated Argument Lists

Compatibility Considerations: Yes

The argument lists for the following types of functions, which are not generally called directly, have been updated.

The argument list for training functions, such as `trainlm`, `traingd`, etc., have been updated to match `train`. The argument list for the adapt function `adaptwb` has been updated. The argument list for the layer and network initialization functions, `initlay`, `initnw`, and `initwb` have been updated.

Compatibility Considerations

Any custom functions of these types, or code which calls these functions manually, will need to be updated.

R2010a

Version: 6.0.4
New Features: No
Bug Fixes: Yes

R2009b

Version: 6.0.3
New Features: No
Bug Fixes: Yes

R2009a

Version: 6.0.2
New Features: No
Bug Fixes: Yes

R2008b

Version: 6.0.1
New Features: No
Bug Fixes: Yes

R2008a

Version: 6.0
New Features: Yes
Bug Fixes: Yes

New Training GUI with Animated Plotting Functions

Compatibility Considerations: Yes

Training networks with the `train` function now automatically opens a window that shows the network diagram, training algorithm names, and training status information.

The window also includes buttons for plots associated with the network being trained. These buttons launch the plots during or after training. If the plots are open during training, they update every epoch, resulting in animations that make understanding network performance much easier.

The training window can be opened and closed at the command line as follows:

```
nntraintool
nntraintool('close')
```

Two plotting functions associated with the most networks are:

- `plotperform`—Plot performance.
- `plottrainstate`—Plot training state.

Compatibility Considerations

To turn off the new training window and display command-line output (which was the default display in previous versions), use these two training parameters:

```
net.trainParam.showWindow = false;
net.trainParam.showCommandLine = true;
```

New Pattern Recognition Network, Plotting, and Analysis GUI

The `nprtool` function opens a GUI wizard that guides you to a neural network solution for pattern recognition problems. Users can define their own problems or use one of the new data sets provided.

The `newpr` function creates a pattern recognition network at the command line. Pattern recognition networks are feed-forward networks that solve problems with Boolean or 1-of-*N* targets and have confusion (`plotconfusion`) and receiver operating characteristic (`plotroc`) plots associated with them.

The new `confusion` function calculates the true/false, positive/negative results from comparing network output classification with target classes.

New Clustering Training, Initialization, and Plotting GUI

Compatibility Considerations: Yes

The `nctool` function opens a GUI wizard that guides you to a self-organizing map solution for clustering problems. Users can define their own problem or use one of the new data sets provided.

The `initsompc` function initializes the weights of self-organizing map layers to accelerate training. The `learnsomb` function implements batch training of SOMs that is orders of magnitude faster than incremental training. The `newsom` function now creates a SOM network using these faster algorithms.

Several new plotting functions are associated with self-organizing maps:

- `plotsomhits`—Plot self-organizing map input hits.
- `plotsomnc`—Plot self-organizing map neighbor connections.
- `plotsomnd`—Plot self-organizing map neighbor distances.
- `plotsomplanes`—Plot self-organizing map input weight planes.
- `plotsompos`—Plot self-organizing map weight positions.
- `plotsomtop`—Plot self-organizing map topology.

Compatibility Considerations

You can call the `newsom` function using conventions from earlier versions of the toolbox, but using its new calling conventions gives you faster results.

New Network Diagram Viewer and Improved Diagram Look

The new neural network diagrams support arbitrarily connected network architectures and have an improved layout. Their visual clarity has been improved with color and shading.

Network diagrams appear in all the Neural Network Toolbox graphical interfaces. In addition, you can open a network diagram viewer of any network from the command line by typing

```
view(net)
```

New Fitting Network, Plots and Updated Fitting GUI

Compatibility Considerations: Yes

The `newfit` function creates a fitting network that consistes of a feed-forward backpropagation network with the fitting plot (`plotfit`) associated with it.

The `nftool` wizard has been updated to use `newfit`, for simpler operation, to include the new network diagrams, and to include sample data sets. It now allows a Simulink block version of the trained network to be generated from the final results panel.

Compatibility Considerations

The code generated by `nftool` is different the code generated in previous versions. However, the code generated by earlier versions still operates correctly.

R2007b

Version: 5.1
New Features: Yes
Bug Fixes: Yes

Simplified Syntax for Network-Creation Functions

Compatibility Considerations: Yes

The following network-creation functions have new input arguments to simplify the network creation process:

- `newcf`
- `newff`
- `newdtdnn`
- `newelm`
- `newfftd`
- `newlin`
- `newlrn`
- `newnarx`
- `newnarxsp`

For detailed information about each function, see the corresponding reference pages.

Changes to the syntax of network-creation functions have the following benefits:

- You can now specify input and target data values directly. In the previous release, you specified input ranges and the size of the output layer instead.
- The new syntax automates preprocessing, data division, and postprocessing of data.

For example, to create a two-layer feed-forward network with 20 neurons in its hidden layer for a given a matrix of input vectors `p` and target vectors `t`, you can now use `newff` with the following arguments:

```
net = newff(p,t,20);
```

This command also sets properties of the network such that the functions `sim` and `train` automatically preprocess inputs and targets, and postprocess outputs.

In the previous release, you had to use the following three commands to create the same network:

```
pr = minmax(p);
s2 = size(t,1);
net = newff(pr,[20 s2]);
```

Compatibility Considerations

Your existing code still works but might produce a warning that you are using obsolete syntax.

Automated Data Preprocessing and Postprocessing During Network Creation

Automated data preprocessing and postprocessing occur during network creation in the Network/Data Manager GUI (`nntool`), Neural Network Fitting Tool GUI (`nftool`), and at the command line.

At the command line, the new syntax for using network-creation functions, automates preprocessing, postprocessing, and data-division operations.

For example, the following code returns a network that automatically preprocesses the inputs and targets and postprocesses the outputs:

```
net = newff(p,t,20);
net = train(net,p,t);
y = sim(net,p);
```

To create the same network in a previous release, you used the following longer code:

```
[p1,ps1] = removeconstantrows(p);
[p2,ps2] = mapminmax(p1);
[t1,ts1] = mapminmax(t);
pr = minmax(p2);
s2 = size(t1,1);
net = newff(pr,[20 s2]);
net = train(net,p2,t1);
y1 = sim(net,p2)
y = mapminmax('reverse',y1,ts1);
```

Default Processing Settings

The default input `processFcns` functions returned with a new network are, as follows:

```
net.inputs{1}.processFcns = ...
              {'fixunknowns','removeconstantrows', 'mapminmax'}
```

These three processing functions perform the following operations, respectively:

- `fixunknowns`—Encode unknown or missing values (represented by `NaN`) using numerical values that the network can accept.
- `removeconstantrows`—Remove rows that have constant values across all samples.
- `mapminmax`—Map the minimum and maximum values of each row to the interval `[-1 1]`.

The elements of `processParams` are set to the default values of the `fixunknowns`, `removeconstantrows`, and `mapminmax` functions.

The default output `processFcns` functions returned with a new network include the following:

```
net.outputs{2}.processFcns = {'removeconstantrows','mapminmax'}
```

These defaults process outputs by removing rows with constant values across all samples and mapping the values to the interval `[-1 1]`.

`sim` and `train` automatically process inputs and targets using the input and output processing functions, respectively. `sim` and `train` also reverse-process network outputs as specified by the output processing functions.

For more information about processing input, target, and output data, see "Multilayer Networks and Backpropagation Training" in the Neural Network Toolbox User's Guide.

Changing Default Input Processing Functions

You can change the default processing functions either by specifying optional processing function arguments with the network-creation function, or by changing the value of `processFcns` after creating your network.

You can also modify the default parameters for each processing function by changing the elements of the `processParams` properties.

After you create a network object (`net`), you can use the following input properties to view and modify the automatic processing settings:

- `net.inputs{1}.exampleInput`—Matrix of example input vectors
- `net.inputs{1}.processFcns`—Cell array of processing function names
- `net.inputs{1}.processParams`—Cell array of processing parameters

The following input properties are automatically set and you cannot change them:

- `net.inputs{1}.processSettings`—Cell array of processing settings
- `net.inputs{1}.processedRange`—Ranges of example input vectors after processing
- `net.inputs{1}.processedSize`—Number of input elements after processing

Changing Default Output Processing Functions

After you create a network object (`net`), you can use the following output properties to view and modify the automatic processing settings:

- `net.outputs{2}.exampleOutput`—Matrix of example output vectors
- `net.outputs{2}.processFcns`—Cell array of processing function names
- `net.outputs{2}.processParams`—Cell array of processing parameters

Note These output properties require a network that has the output layer as the second layer.

The following new output properties are automatically set and you cannot change them:

- `net.outputs{2}.processSettings`—Cell array of processing settings
- `net.outputs{2}.processedRange`—Ranges of example output vectors after processing

- `net.outputs{2}.processedSize`—Number of input elements after processing

Automated Data Division During Network Creation

When training with supervised training functions, such as the Levenberg-Marquardt backpropagation (the default for feed-forward networks), you can supply three sets of input and target data. The first data set trains the network, the second data set stops training when generalization begins to suffer, and the third data set provides an independent measure of network performance.

Automated data division occurs during network creation in the Network/Data Manager GUI, Neural Network Fitting Tool GUI, and at the command line.

At the command line, to create and train a network with early stopping that uses 20% of samples for validation and 20% for testing, you can use the following code:

```
net = newff(p,t,20);
net = train(net,p,t);
```

Previously, you entered the following code to accomplish the same result:

```
pr = minmax(p);
s2 = size(t,1);
net = newff(pr,[20 s2]);
[trainV,validateV,testV] = dividevec(p,t,0.2,0.2);
[net,tr] = train(net,trainV.P,trainV.T,[],[],validateV,testV);
```

For more information about data division, see "Multilayer Networks and Backpropagation Training" in the Neural Network Toolbox User's Guide.

New Data Division Functions

The following are new data division functions:

- `dividerand`—Divide vectors using random indices.
- `divideblock`—Divide vectors in three blocks of indices.
- `divideint`—Divide vectors with interleaved indices.
- `divideind`—Divide vectors according to supplied indices.

Default Data Division Settings

Network creation functions return the following default data division properties:

- `net.divideFcn = 'dividerand'`
- `net.divedeParam.trainRatio = 0.6;`
- `net.divideParam.valRatio = 0.2;`
- `net.divideParam.testRatio = 0.2;`

Calling `train` on the network object net divided the set of input and target vectors into three sets, such that 60% of the vectors are used for training, 20% for validation, and 20% for independent testing.

Changing Default Data Division Settings

You can override default data division settings by either supplying the optional data division argument for a network-creation function, or by changing the corresponding property values after creating the network.

After creating a network, you can view and modify the data division behavior using the following new network properties:

- `net.divideFcn`—Name of the division function
- `net.divideParam`—Parameters for the division function

New Simulink Blocks for Data Preprocessing

New blocks for data processing and reverse processing are available. For more information, see “Processing Blocks” in the Neural Network Toolbox User’s Guide.

The function `gensim` now generates neural networks in Simulink that use the new processing blocks.

Properties for Targets Now Defined by Properties for Outputs

Compatibility Considerations: Yes

The properties for targets are now defined by the properties for outputs. Use the following properties to get and set the output and target properties of your network:

- `net.numOutputs`—The number of outputs and targets
- `net.outputConnect`—Indicates which layers have outputs and targets
- `net.outputs`—Cell array of output subobjects defining each output and its target

Compatibility Considerations

Several properties are now obsolete, as described in the following table. Use the new properties instead.

Recommended Property	Obsolete Property
`net.numOutputs`	`net.numTargets`
`net.outputConnect`	`net.targetConnect`
`net.outputs`	`net.targets`

R2007a

Version: 5.0.2
New Features: No
Bug Fixes: No

No New Features or Changes

R2006b

Version: 5.0.1
New Features: No
Bug Fixes: No

No New Features or Changes

R2006a

Version: 5.0
New Features: Yes
Bug Fixes: No

Dynamic Neural Networks

Version 5.0 now supports these types of dynamic neural networks:

Time-Delay Neural Network

Both focused and distributed time-delay neural networks are now supported. Continue to use the `newfftd` function to create focused time-delay neural networks. To create distributed time-delay neural networks, use the `newdtdnn` function.

Nonlinear Autoregressive Network (NARX)

To create parallel NARX configurations, use the `newnarx` function. To create series-parallel NARX networks, use the `newnarxsp` function. The `sp2narx` function lets you convert NARX networks from series-parallel to parallel configuration, which is useful for training.

Layer Recurrent Network (LRN)

Use the `newlrn` function to create LRN networks. LRN networks are useful for solving some of the more difficult problems in filtering and modeling applications.

Custom Networks

The training functions in Neural Network Toolbox are enhanced to let you train arbitrary custom dynamic networks that model complex dynamic systems. For more information about working with these networks, see the Neural Network Toolbox documentation.

Wizard for Fitting Data

The new Neural Network Fitting Tool (`nftool`) is now available to fit your data using a neural network. The Neural Network Fitting Tool is designed as a wizard and walks you through the data-fitting process step by step.

To open the Neural Network Fitting Tool, type the following at the MATLAB prompt:

```
nftool
```

Data Preprocessing and Postprocessing

Compatibility Considerations: Yes

Version 5.0 provides the following new data preprocessing and postprocessing functionality:

dividevec Automatically Splits Data

The `dividevec` function facilitates dividing your data into three distinct sets to be used for training, cross validation, and testing, respectively. Previously, you had to split the data manually.

fixunknowns Encodes Missing Data

The `fixunknowns` function encodes missing values in your data so that they can be processed in a meaningful and consistent way during network training. To reverse this preprocessing operation and return the data to its original state, call `fixunknowns` again with `'reverse'` as the first argument.

removeconstantrows Handles Constant Values

`removeconstantrows` is a new helper function that processes matrices by removing rows with constant values.

mapminmax, mapstd, and processpca Are New

The `mapminmax`, `mapstd`, and `processpca` functions are new and perform data preprocessing and postprocessing operations.

Compatibility Considerations

Several functions are now obsolete, as described in the following table. Use the new functions instead.

New Function	Obsolete Functions

`mapminmax`	`premnmx`
	`postmnmx`
	`tramnmx`

`mapstd`	`prestd`
	`poststd`
	`trastd`

`processpca`	`prepca` `trapca`

Each new function is more efficient than its obsolete predecessors because it accomplishes both preprocessing and postprocessing of the data. For example, previously you used `premnmx` to process a matrix, and then `postmnmx` to return the data to its original state. In this release, you accomplish both operations using `mapminmax`; to return the data to its original state, you call `mapminmax` again with `'reverse'` as the first argument:

```
mapminmax('reverse',Y,PS)
```

Derivative Functions Are Obsolete

Compatibility Considerations: Yes

The following derivative functions are now obsolete:

```
ddotprod
dhardlim
dhardlms
dlogsig
dmae
dmse
dmsereg
dnetprod
dnetsum
dposlin
dpurelin
dradbas
dsatlin
dsatlins
dsse
dtansig
dtribas
```

Each derivative function is named by prefixing a `d` to the corresponding function name. For example, `sse` calculates the network performance function and `dsse` calculated the derivative of the network performance function.

Compatibility Considerations

To calculate a derivative in this version, you must pass a derivative argument to the function. For example, to calculate the derivative of a hyperbolic tangent sigmoid transfer function `A` with respect to `N`, use this syntax:

```
A = tansig(N,FP)
dA_dN = tansig('dn',N,A,FP)
```

Here, the argument `'dn'` requests the derivative to be calculated.

R14SP3

Version: 4.0.6
New Features: No
Bug Fixes: No

No New Features or Changes