Deep Learning Toolbox™ Release Notes

MATLAB®

How to Contact MathWorks

	Latest news:	www.mathworks.com
	Sales and services:	www.mathworks.com/sales_and_services
	User community:	www.mathworks.com/matlabcentral
	Technical support:	www.mathworks.com/support/contact_us
	Phone:	508-647-7000

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

Contents

R2022a

R2021b

R2021a

R2020b

R2019b

R2019a

R2018b

R2018a

R2017b

R2017a

R2016b

R2016a

R2015b

R2015a

R2014b

Bug Fixes

R2014a

R2013b

Bug Fixes

Bug Fixes

Bug Fixes

R2011a

Bug Fixes

R2010b

R2010a

Bug Fixes

R2009b

Bug Fixes

R2009a

Bug Fixes

R2008b

Bug Fixes

R2008a

R2007b

R2007a

No New Features or Changes

R2006b

No New Features or Changes

R2006a

R14SP3

No New Features or Changes

R2022b

Version: 14.5

New Features

Bug Fixes

Compatibility Considerations

Apps and Visualization

Experiment Manager: Monitor model performance while you run experiments

During training, the results table now displays the intermediate values for standard training and validation metrics for built-in training experiments. These metrics include loss, accuracy of classification experiments, and root mean squared error of regression experiments.

Experiment Manager: Inspect execution environment for each trial

In built-in training experiments, the new **Execution Environment** column of the results table displays whether each trial runs on a single CPU, a single GPU, multiple CPUs, or multiple GPUs. To show this information, click the show or hide columns button located above the results table and select **Execution Environment**.

Experiment Manager: Restart multiple trials

Restart multiple trials of your experiment by opening the **Restart** list, selecting one or more restarting criteria, and clicking **Restart** . The restarting criteria include `All Canceled`, `All Stopped`, `All Error`, and `All Discarded`. For more information, see "Stop and Restart Training".

Experiment Manager: Reduce storage size by discarding unwanted results

To reduce the size of your experiments, you can now discard the results of trials that are no longer relevant. In the **Actions** column of the results table, click the Discard button for a trial. Experiment Manager deletes the training plot, confusion matrix, trained network, training information, and training output from your project.

Experiment Manager: New properties for experiments.Monitor objects

The `experiments.Monitor` object has these new properties:

- `InfoData` — Information column values for trial
- `MetricData` — Metric column values for trial

An `experiments.Monitor` object has the same properties and object functions as a `TrainingProgressMonitor` object. Therefore, you can easily adapt your custom training loop plotting code for use in an **Experiment Manager** setup script. For more information, see "Prepare Plotting Code for Custom Training Experiment".

Deep Network Designer: New layer groups and colors

Deep Network Designer has improved layer colors and layer groups. Smaller layer groups mean you can easily find the layer you need.

Visualization: Monitor and plot custom training loop progress

Track custom training loop progress and generate training plots using a `TrainingProgressMonitor` object.

Create a `TrainingProgressMonitor` object using the `trainingProgressMonitor` function. Use the `TrainingProgressMonitor` object to:

- Track training progress.
- Display animated plots of metric values during training.
- Display and monitor information values during training.
- Log metric and information values.

After you create a `TrainingProgressMonitor` object, you can use these object functions:

- `recordMetrics` — Record metric values during training
- `updateInfo` — Update information values during training
- `groupSubPlot` — Group metrics in training plot

For more information, see "Monitor Custom Training Loop Progress".

A `TrainingProgressMonitor` object has the same properties and object functions as an `experiments.Monitor` object. Therefore, you can easily adapt your plotting code for use in an **Experiment Manager** setup script. For more information, see "Prepare Plotting Code for Custom Training Experiment".

Visualization: Compute and plot ROC curve

Compute and plot classification performance metrics including receiver operating characteristic (ROC) curves.

Use `rocmetrics` to evaluate the performance of classification models with performance metrics. You can create a `rocmetrics` object by passing true labels, classification scores, and class names. By default, `rocmetrics` computes the true positive rates (TPR), false positive rates (FPR), and area under the ROC curve for each class. Additionally, you can specify more supported performance metrics by using the `AdditionalMetrics` name-value argument.

After you create a `rocmetrics` object, you can use these object functions:

- `plot` — Plot ROC or other classifier performance curves. The `plot` function returns a `ROCCurve` graphics object for each curve. You can control the appearance of each curve by modifying the properties of the objects. For details, see ROCCurve Properties.
- `average` — Compute performance metrics for an average ROC curve for multiclass problems.
- `addMetrics` — Compute additional classification performance metrics.

For an example that shows how to use ROC curves to compare the performance of deep learning models, see "Compare Deep Learning Models Using ROC Curves".

In R2022a, `rocmetrics` was introduced in Statistics and Machine Learning Toolbox™. You can now use `rocmetrics` without a Statistics and Machine Learning Toolbox license. Some options, for example, `rocmetrics` using confidence intervals, still require a Statistics and Machine Learning Toolbox license. For more information, see `rocmetrics`.

Interpretability: Create Grad-CAM maps with 1-D convolutional networks for sequence and time-series data

Compute Grad-CAM interpretability maps for deep neural networks that are trained on sequence and time-series data. To compute Grad-CAM interpretability maps, use the `gradCAM` function. For an example that shows how to use a Grad-CAM map to interpret the predictions of a network trained on time series data, see "Interpret Deep Learning Time-Series Classifications Using Grad-CAM".

Algorithms

Complex Numbers: Train networks using complex-valued data

To pass complex-valued data to a neural network, you can use the input layer to split the complex values into their real and imaginary parts before the network passes the data to the subsequent layers. When the layer splits complex-valued data, the layer outputs the data in twice as many channels as the input data.

The `ImageInputLayer`, `SequenceInputLayer`, and `FeatureInputLayer` objects support splitting input data into its real and imaginary components. To input complex-valued data into a network by splitting it into its real and imaginary parts, set the `SplitComplexInputs` option of the network input layer to `1` (`true`).

To input complex data into a network, the `SplitComplexInputs` option must be `1`.

For an example that shows how to train a network with complex-valued data, see "Train Network with Complex-Valued Data".

Gaussian Error Linear Unit (GELU) Activation: Create and train networks with GELU activation

The Gaussian error linear unit (GELU) operation weights the input by its probability under a Gaussian distribution. This operation is given by

$$\mathrm{GELU}(x) = \frac{x}{2}\left(1 + \mathrm{erf}\left(\frac{x}{\sqrt{2}}\right)\right),$$

where erf denotes the error function.

To apply the GELU operation in a layer array or layer graph, use a `geluLayer` object.

To apply the GELU operation in a custom layer or deep learning model function, use the `gelu` function.

Spatio-Temporal Data: 2-D average and max pooling layers support data with both spatial and time dimensions

`AveragePooling2DLayer` and `MaxPooling2DLayer` objects now support input data with these data formats:

- One spatial dimension and a time dimension
- Two spatial dimensions and a time dimension

For an example that shows how to create a 2-D CNN-LSTM network for speech classification tasks by combining a 2-D convolutional neural network (CNN) with a long short-term memory (LSTM) layer, see "Sequence Classification Using CNN-LSTM Network".

Sequence Networks: Make predictions in parallel

The `predict`, `classify`, and `activations` functions now support making predictions in parallel for networks with sequence input. To make predictions in parallel, set the `ExecutionEnvironment`

option to `"parallel"` or `"multi-gpu"`. Making predictions in parallel requires a Parallel Computing Toolbox™ license.

When you make predictions in parallel for networks with recurrent layers, the `SequenceLength` option must be `"longest"` or `"shortest"`.

Networks with custom layers that contain `State` parameters do not support making predictions in parallel.

LSTM Projected Layer: Perform LSTM operations with fewer learnable parameters

To compress a deep learning network by reducing the number of learnable parameters, you can use *projected layers*. A projected layer is a variant of a deep learning layer that enables compression by reducing the number of stored learnable parameters by replacing multiplications of the form Wx, where W is a learnable matrix, with the multiplication $WQQ^\top x$, where Q is a projector matrix. Instead of storing W, the layer instead stores Q and $W' = WQ$. Projecting into a lower-dimensional space with Q typically requires less memory and can have similarly strong prediction accuracy.

To create a long short-term memory (LSTM) projected layer, use `lstmProjectedLayer`. For an example that shows how to train a network with an LSTM projected layer and compare the prediction accuracy and number of learnable parameters to a network without projection, see "Train Network with LSTM Projected Layer".

Custom Layers: Define Custom Layer Learnable Parameter Initialization

When you create a custom layer, specify a custom learnable parameter initialization function by implementing a function with this syntax:

```
function layer = initialize(layer,layout1,...,layoutN)
    ...
end
```

The input `layer` is an instance of the custom layer and `layout1,...,layoutN` are `networkDataLayout` objects corresponding to each of the `N` inputs of the layer.

The software uses this `initialize` function when you include the layer in a `dlnetwork` object or call the `initialize` function on the `dlnetwork` object.

For an example that shows how to define a custom layer that uses automatic learnable parameter initialization, see "Define Custom Deep Learning Layer with Learnable Parameters".

For more information about defining custom layers, see "Define Custom Deep Learning Layers".

Customization: Add, remove, and replace layers of dlnetwork objects

Add, remove, and replace layers in `dlnetwork` objects using these functions:

- `addInputLayer` — Add input layer to `dlnetwork` object
- `addLayers` — Add layers to `dlnetwork` object

- removeLayers — Remove layers from `dlnetwork` object
- connectLayers — Connect layers in `dlnetwork` object
- disconnectLayers — Disconnect layers in `dlnetwork` object
- replaceLayer — Replace layer in `dlnetwork` object

Customization: Plot and view summary of dlnetwork objects

Plot `dlnetwork` architecture using the `plot` function.

Print a summary of objects using the `summary` function. The summary shows whether the network is initialized, the total number of learnable parameters, and information about the network inputs.

Customization: Initialize dlnetwork objects using only input size and format information

Initialize the learnable parameters of a `dlnetwork` object by passing a `networkDataLayout` object to the `dlnetwork` and `initialize` functions.

In previous versions, the `dlnetwork` and `initialize` functions require you to pass examples of input data. You can now use `networkDataLayout` objects, which only contain the data size and `dlarray` format.

Create an unformatted `networkDataLayout` object using the syntax `layout = networkDataLayout(sz)`, where `sz` specifies the size of the data. Create a formatted `networkDataLayout` object using the syntax `layout = networkDataLayout(sz,fmt)`, where `fmt` specifies the `dlarray` format of the data.

Attention: Apply attention operation to dlarray input

The attention operation focuses on parts of the input using weighted multiplication operations. To apply the attention operation to a set of queries, keys, and values, use the `attention` function. You can specify the scale, dropout probability, and padding and attention masks used in the operation.

You can use the attention function when you work with custom training loops and custom layers to implement:

- Dot product attention
- Scaled dot product attention
- Multi-head attention
- Luong attention

For an example that shows how to use attention for sequence-to-sequence translation, see "Sequence-to-Sequence Translation Using Attention".

Automatic Differentiation: Use more functions with dlarray input

Use these functions with a `dlarray` object as input when you work with custom training loops and custom layers.

- Attention — Apply the attention operation to a set of queries, keys, and values, using `attention`.
- GELU — Apply the GELU activation function to input elements using `gelu`.
- Error function — Compute the error function of input elements using `erf`.
- String – Convert `dlarray` objects to `string` objects using `string`.
- Categorical – Convert `dlarray` objects to `categorical` objects using `categorical`.

For `dlarray` input, the `string` and `categorical` functions convert to these respective data types only and do not support automatic differentiation.

For a full list of functions that support `dlarray` input, see "List of Functions with dlarray Support".

Function Layer: Accelerate layer functions

You can now specify that a layer function supports acceleration using `dlaccelerate` by setting the `Acceleratable` property to `1` (`true`) when you create a `functionLayer`. Setting `Acceleratable` to `1` (`true`) can improve the performance of training and inference (prediction) when you use a `dlnetwork` object. For example, calling `predict` on a `dlnetwork` object containing a number of `functionLayer` objects in this test is about 3x faster than in the previous release:

```
function timeFunctionLayer

% Prepare input data
X = dlarray(rand(500,500,3,"single","gpuArray"),"SSCB");

% Prepare a convolution and ReLU block
convBlock = [convolution2dLayer(4,20); reluLayer()];

if version("-release") == "2022b"
    % Create a network using functionLayer ReLU with Acceleratable
    convBlock(2) = functionLayer(@(x) relu(x),Acceleratable=true);
    layers = repmat(convBlock,80,1);
    net = dlnetwork(layers,X);
else
    % Create a network using functionLayer ReLU
    convBlock(2) = functionLayer(@(x) relu(x));
    layers = repmat(convBlock,80,1);
    net = dlnetwork(layers,X);
end

% Time the predict function
gputimeit(@()predict(net,X))

end
```

The approximate execution times are:

R2022a: 0.12 seconds

R2022b: 0.04 seconds

The code was timed on a Windows® 10, Intel® Xeon® W-2133 @ 3.60 GHz test system with an NVIDIA® RTX A5000 GPU by calling the `timeFunctionLayer` function.

Bayesian Neural Networks: Create and train Bayesian neural networks

A Bayesian neural network (BNN) is a type of deep learning network that uses Bayesian methods to quantify the uncertainty in the predictions of a deep learning network.

You can train a BNN to generate a distribution of weights and biases, rather than a single set. You can then use these distributions to measure the uncertainty of the network predictions.

To learn how to train a BNN using the Bayes by backpropagation method, see "Train Bayesian Neural Network".

Background Dispatch: Use DispatchInBackground on thread pools

You can now use background dispatch (asynchronous prefetch queuing) for reading training data from datastores on thread-based parallel pools.

To use background dispatch when training a network using `trainNetwork`, set the `DispatchInBackground` training option to `1` (`true`) using the `trainingOptions` function and open a thread-based parallel pool using `parpool("Threads")`.

To use background dispatch when training a network using a custom training loop, create a `minibatchqueue` object and set the `DispatchInBackground` property to `1` (`true`), and open a thread-based parallel pool using `parpool("Threads")`.

As an alternative to opening a thread-based parallel pool using `parpool("Threads")`, you can set the default parallel environment on your local machine from the MATLAB desktop **Home** tab, in the **Environment** area, by selecting **Parallel > Select Parallel Environment > Threads**, or by calling `parallel.defaultProfile("Threads")`.

Network Creation: Improved performance

Assembling a `DAGNetwork` object using `assembleNetwork` and assembling a `dlnetwork` object using `dlnetwork` show improved performance. Improvements are greater for networks that contain more layers. For example, assembling a `DAGNetwork` object in this test is about 1.9x faster than in the previous release:

```
function timeAssembleNetwork

% Create a layer graph
lgraph = layerGraph(resnet50);

% Time assembling a DAGNetwork object from the layer graph
tic
net = assembleNetwork(lgraph);
toc

end
```

The approximate execution times are:

R2022a: 3.80 seconds

R2022b: 2.01 seconds

Assembling a `dlnetwork` in this test is about 1.9x faster than in the previous release:

```
function timeDlnetwork

% Create a layer graph
lgraph = layerGraph(resnet50);

% Remove the output layer from the layer graph
lgraph = removeLayers(lgraph, lgraph.Layers(end).Name);

% Time assembling a dlnetwork object from the layer graph
tic
dlnet = dlnetwork(lgraph);
toc

end
```

The approximate execution times are:

R2022a: 4.09 seconds

R2022b: 2.13 seconds

The code was timed on a Windows 10, Intel Xeon W-2133 @ 3.60 GHz test system with an NVIDIA RTX A5000 GPU by calling the `timeAssembleNetwork` and `timeDlnetwork` functions.

dlarray Constructor: Improved performance

Creating `dlarray` data shows improved performance. For example, creating formatted `dlarray` objects in this test is about 3.2x faster than in the previous release:

```
function timeDlarray

% Prepare data
params = arrayfun(@(i)randn(5,5,20,20,'single'),1:10000,UniformOutput=0);

% Time the dlarray constructor
tic
for i = 1:numel(params)
    params{i} = dlarray(params{i},'SSCB');
end
toc

end
```

The approximate execution times are:

R2022a: 0.55 seconds

R2022b: 0.17 seconds

The code was timed on a Windows 10, Intel Xeon W-2133 @ 3.60 GHz test system with an NVIDIA RTX A5000 GPU by calling the `timeDlarray` function.

replaceLayer Function: Improved performance

The `replaceLayer` function shows improved performance. For example, replacing the final classification layer of a network containing a classification layer with different classes in this test is about 1.5x faster than in the previous release:

```
function timeReplaceLayer

% Load a pretrained network and get the name of the layer to be replaced
net = squeezenet;
cLayer = net.Layers(end);
layerName = cLayer.Name;

% Set the classes for the replacement layer
cLayer.Classes = string(0:1000);

% Convert the network to a layer graph
lgraph = layerGraph(net);

% Time the layer replacement
tic
lgraph = replaceLayer(lgraph,layerName,cLayer);
toc

end
```

The approximate execution times are:

R2022a: 0.41 seconds

R2022b: 0.28 seconds

The code was timed on a Windows 10, Intel Xeon W-2133 @ 3.60 GHz test system with an NVIDIA RTX A5000 GPU by calling the `timeReplaceLayer` function.

Parameter Updates: Improved performance of parameter updates using a GPU

Element-wise operations on large numbers of `gpuArray` objects show improved performance, significantly speeding up parameter updates using the `adamupdate`, `sgdmupdate`, and `rmspropupdate` functions. Improvements are greater for networks that contain more layers. For example, updating parameters using `adamupdate` in this test is about 2.7x faster than in the previous release:

```
function timeAdamUpdate

% Create a layer graph
net = resnet101;
lgraph = layerGraph(net);

% Remove the output layer from the layer graph and create a dlnetwork
lgraph = removeLayers(lgraph,lgraph.Layers(end).Name);
net = dlnetwork(lgraph);

% Convert the learnable parameters to gpuArray objects
net = dlupdate(@gpuArray,net);
```

```
% Initialise the update variables
fakeG = net.Learnables;
avgG = [];
avgsqG = [];
[net, avgG, avgsqG] = adamupdate(net,fakeG,avgG,avgsqG,1);

% Time adamupdate
gputimeit(@() adamupdate(net,fakeG,avgG,avgsqG,2))

end
```

The approximate execution times are:

R2022a: 0.81 seconds

R2022b: 0.30 seconds

The code was timed on a Windows 10, Intel Xeon W-2133 @ 3.60 GHz test system with an NVIDIA RTX A5000 GPU by calling the `timeAdamUpdate` function.

Custom Layers: Improved performance in dlnetwork

Custom layers in a `dlnetwork` object show improved performance for training and inference. For example, computing the network outputs for training using `forward` for a network containing a number of instances of `functionLayer` in this test is about 3.7x faster than in the previous release:

```
function timeCustomLayer

% Create a dlnetwork object containing custom layers
layer = functionLayer(@(x) plus(x,1));
layerArray = repelem(layer,100);
dlnet = dlnetwork(layerArray,dlarray(1,"SSCB"));

% Prepare input data
X = dlarray(1,"SSCB");

% Warm-up iterations
for i=1:10
    forward(dlnet,X);
end

% Timed iterations
tic
for i=1:100
    forward(dlnet,X);
end
toc

end
```

The approximate execution times are:

R2022a: 6.61 seconds

R2022b: 1.79 seconds

The code was timed on a Windows 10, Intel Xeon W-2133 @ 3.60 GHz test system with an NVIDIA RTX A5000 GPU by calling the `timeCustomLayer` function.

tan and tanh Functions: Improved performance within dlgradient call

The `tan` and `tanh` functions show improved performance when used within a `dlgradient` call. For example, evaluating gradients of a `tan` function in this test is about 1.5x faster than in the previous release:

```matlab
function timeTan

% Prepare input data and gradient function
x = dlarray(randn(5000));
Tan = @(x) dlgradient(sum(tan(x),"all"),x);

% Warm-up iterations
for i = 1:10
    dlfeval(Tan,x);
end

% Timed iterations
tic
for i = 1:10
    dlfeval(Tan,x);
end
toc

end
```

The approximate execution times are:

R2022a: 2.31 seconds

R2022b: 1.58 seconds

Evaluating gradients of a `tanh` function in this test is about 1.5x faster than in the previous release:

```matlab
function timeTanh

% Prepare input data and gradient function
x = dlarray(randn(5000));
Tanh = @(x) dlgradient(sum(tanh(x),"all"),x);

% Warm-up iterations
for i = 1:10
    dlfeval(Tanh,x);
end

% Timed iterations
tic
for i = 1:10
    dlfeval(Tanh,x);
end
toc

end
```

The approximate execution times are:

R2022a: 3.10 seconds

R2022b: 2.03 seconds

The code was timed on a Windows 10, Intel Xeon W-2133 @ 3.60 GHz test system with an NVIDIA RTX A5000 GPU by calling the `timeTan` and `timeTanh` functions.

dlode45 Function: Improved performance within dlgradient call

The `dlode45` function shows improved performance when computing gradients using the `adjoint` gradient mode within a `dlgradient` call. In particular, the first call to `dlode45` shows a significant performance improvement. For example, evaluating gradients of a function calling `dlode45` for the first time in this test is about 6.1x faster than in the previous release and subsequent calls are about 1.2x faster than in the previous release:

```
function timeDlode45

% Prepare input data
tspan = [0 1];
nChannels = 5;
pp = cell(1,2);
pp{1} = 0.01*dlarray(rand(nChannels));
pp{2} = 0.01*dlarray(rand(nChannels));
x   = dlarray(rand(nChannels,100));

% Prepare gradient function
ODE = @(~,y,p) p{2}*sin(p{1}*y);
sol = @(x,pp,tspan) dlode45(ODE,tspan,x,pp,DataFormat="CB",GradientMode="adjoint");
gradsAdj = @(x,pp,tspan) dlgradient(sum(sol(x,pp,tspan),"all"),x,pp);

% Time only the first call to dlode45
for i=1:5
    % Clear any previously cached traces of the accelerated function
    clear functions

    tic
    dlfeval(gradsAdj,x,pp,tspan);
    time(i) = toc;
end

% Calculate the mean time for the first call to dlode45
timeFirstCall = mean(time)


% Time many calls to dlode45
% Warm-up iterations
clear functions
for i = 1:10
    dlfeval(gradsAdj,x,pp,tspan);
end

% Timed iterations
tic
for i=1:100
```

```
    dlfeval(gradsAdj,x,pp,tspan);
end
toc

end
```

The approximate execution times for the first call are:

R2022a: 44.5 seconds

R2022b: 7.3 seconds

The approximate execution times for the 100 subsequent calls are:

R2022a: 6.78 seconds

R2022b: 5.80 seconds

The code was timed on a Windows 10, Intel Xeon W-2133 @ 3.60 GHz test system with an NVIDIA RTX A5000 GPU by calling the `timeDlode45` function.

DAGNetwork Inference: Improved performance of GPU inference on networks performing operations with a stride

GPU inference using a `DAGNetwork` object containing layers performing operations with a stride shows improved performance. Layers that perform operations with a stride include pooling and convolution layers, such as a `convolution2dLayer` and a `maxPooling2dLayer`, when any element of the `stride` property is greater than 1. The performance improvement is greater for networks containing more layers that perform operations with a stride. For example, making predictions using `resnet101` on the GPU in this test is about 1.1x faster than in the previous release:

```
function timeStrideInference

% Load a trained network and prepare input data
net = resnet101;
X = rand(224,224,3,500,"gpuArray");

% Time the predict function
gputimeit(@() predict(net,X))

end
```

The approximate execution times are:

R2022a: 0.52 seconds

R2022b: 0.49 seconds

The code was timed on a Windows 10, Intel Xeon W-2133 @ 3.60 GHz test system with an NVIDIA RTX A5000 GPU by calling the `timeStrideInference` function.

Pretrained Model on GitHub: Deep Speech speech-to-text model

To learn how to load a pretrained Deep Speech model into MATLAB®, see the Speech-to-Text Transcription Using Deep Speech repository. The Deep Speech model is suitable for transfer learning, code generation, and streaming.

To find the latest pretrained models for deep learning in MATLAB, see MATLAB Deep Learning Model Hub.

Functionality being removed or changed

trainNetwork pads mini-batches to length of longest sequence before splitting when you specify SequenceLength training option as an integer
Behavior change

Starting in R2022b, when you train a network with sequence data using the `trainNetwork` function and the `SequenceLength` option is an integer, the software pads sequences to the length of the longest sequence in each mini-batch and then splits the sequences into mini-batches with the specified sequence length. If `SequenceLength` does not evenly divide the sequence length of the mini-batch, then the last split mini-batch has a length shorter than `SequenceLength`. This behavior prevents the network training on time steps that contain only padding values.

In previous releases, the software pads mini-batches of sequences to have a length matching the nearest multiple of `SequenceLength` that is greater than or equal to the mini-batch length and then splits the data. To reproduce this behavior, use a custom training loop and implement this behavior when you preprocess mini-batches of data.

Prediction functions pad mini-batches to length of longest sequence before splitting when you specify SequenceLength option as an integer
Behavior change

Starting in R2022b, when you make predictions with sequence data using the `predict`, `classify`, `predictAndUpdateState`, `classifyAndUpdateState`, and `activations` functions and the `SequenceLength` option is an integer, the software pads sequences to the length of the longest sequence in each mini-batch and then splits the sequences into mini-batches with the specified sequence length. If `SequenceLength` does not evenly divide the sequence length of the mini-batch, then the last split mini-batch has a length shorter than `SequenceLength`. This behavior prevents time steps that contain only padding values from influencing predictions.

In previous releases, the software pads mini-batches of sequences to have a length matching the nearest multiple of `SequenceLength` that is greater than or equal to the mini-batch length and then splits the data. To reproduce this behavior, manually pad the input data such that the mini-batches have the length of the appropriate multiple of `SequenceLength`. For sequence-to-sequence workflows, you may also need to manually remove time steps of the output that correspond to padding values.

Import and Export

Deep Learning Toolbox Converter for PyTorch Models: Support for importing networks from PyTorch

You can now import a pretrained PyTorch® model for image classification as a MATLAB network by using the `importNetworkFromPyTorch` function. The `importNetworkFromPyTorch` function imports the PyTorch model as an uninitialized `dlnetwork` object. For an example that shows how to add an input image to the imported network, initialize the network, and use the network for image classification, see "Import Network from PyTorch and Classify Image".

The `importNetworkFromPyTorch` function requires the new support package Deep Learning Toolbox Converter for PyTorch Models. If this support package is not installed, then the function provides a download link.

Export to TensorFlow Model: Save MATLAB network or layer graph as TensorFlow model

You can now export a Deep Learning Toolbox network or layer graph to TensorFlow™ by using the `exportNetworkToTensorFlow` function. The `exportNetworkToTensorFlow` function saves the exported TensorFlow model in a regular Python® package. You can load the exported model and use it for prediction or training. You can also share the exported model by saving it to `SavedModel` or `HDF5` format.

TensorFlow Import Operator Support: Import models that include Assert, GreaterEqual, and Size operators

You can now import a TensorFlow model that includes `Assert`, `GreaterEqual`, and `Size` operators by using the `importTensorFlowNetwork` and `importTensorFlowLayers` functions. For a list of the TensorFlow operators that the functions support for conversion into MATLAB functions with `dlarray` support, see "Supported TensorFlow Operators".

Import and Export Workflows: New help and tips for interoperability between Deep Learning Toolbox, TensorFlow, PyTorch, and ONNX

These topics help you import networks from and export networks to external deep learning platforms:

- "Interoperability Between Deep Learning Toolbox, TensorFlow, PyTorch, and ONNX"
- "Tips on Importing Models from TensorFlow, PyTorch, and ONNX"

Deployment

Quantization: Independently select calibration, simulation, and validation environments

The prerequisites required for each step of the quantization workflow now depend on your selection at each stage of the quantization workflow. For details, see "Quantization Workflow Prerequisites".

In previous versions, the prerequisites required for validation of a quantized network on target hardware are also required for the calibration step of quantization. You can now choose the calibration environment to use independent of the selected execution environment and can choose to simulate the quantized network in MATLAB rather than quantizing and validating on hardware.

Quantization: Calibrate on host GPU or CPU

You can now choose whether to calibrate your network using the host GPU or host CPU. By default, the `calibrate` function and the **Deep Network Quantizer** app calibrate on the host GPU if one is available.

In previous versions, the execution environment must be the same as the instrumentation environment you use for the calibration step of quantization.

Quantization: Quantize dlnetwork objects

The `dlquantizer` object and **Deep Network Quantizer** now support `dlnetwork` objects for quantization with the `calibrate` and `validate` functions.

Quantization: Prepare for quantization with layer equalization

Use the `equalizeLayers` function to equalize the layer parameters of a deep neural network. Equalizing layer parameters before quantization can improve the accuracy of the quantized network and does not require data or retraining of the network.

Quantization: Specify mini-batch size for calibration

Use the `MiniBatchSize` argument of the `calibrate` function to specify the size of mini-batches for calibration. Larger mini-batch sizes require more memory, but can lead to faster calibration.

Quantization: Simulate quantized network for FPGA execution environment

You can now use the `quantize` function to create a quantized network for simulation when you set the `ExecutionEnvironment` property of `dlquantizer` to FPGA. The quantized network enables visibility of the quantized layers, weights, and biases of the network, as well as quantized inference behavior for simulation.

TensorFlow Lite: Generate C++ code for pretrained models and deploy on Windows platforms

Use the `loadTFLiteModel` function to load a pretrained TensorFlow Lite model into a `TFLiteModel` object. Use this object with the `predict` function in your MATLAB code to perform inference in MATLAB execution, code generation, or inside MATLAB Function blocks in Simulink® models.

To use this functionality, you must install the Deep Learning Toolbox Interface for TensorFlow Lite. For more information, see "Prerequisites for Deep Learning with TensorFlow Lite Models".

For examples, see:

- "Generate Code for TensorFlow Lite (TFLite) Model and Deploy on Raspberry Pi"
- "Deploy Super Resolution Application That Uses TensorFlow Lite (TFLite) Model on Host and Raspberry Pi"

Application Examples

Deep Learning Workflows: New and updated examples and topics

New examples and topics help you progress with deep learning:

- "Sequence Classification Using CNN-LSTM Network"
- "Train Bayesian Neural Network"
- "Interpret Deep Learning Time-Series Classifications Using Grad-CAM"
- "Monitor Custom Training Loop Progress"
- "Compare Deep Learning Models Using ROC Curves"
- "Train Latent ODE Network with Irregularly Sampled Time-Series Data"
- "Multivariate Time Series Anomaly Detection Using Graph Neural Network"
- "Export Image Classification Network from Deep Network Designer to Simulink"
- "Improve Performance of Deep Learning Simulations in Simulink"

Image Processing and Computer Vision: New examples

New examples for image processing and computer vision tasks, including the new Medical Imaging Toolbox™:

- "Segment Lungs from CT Scan Using Pretrained Neural Network"
- "Brain MRI Segmentation Using Pretrained 3-D U-Net Network"
- "Breast Tumor Segmentation from Ultrasound Using Deep Learning"

Signal Processing: New examples

New examples for signal processing tasks include:

- "Human Health Monitoring Using Continuous Wave Radar and Deep Learning"
- "Detect Air Compressor Sounds in Simulink Using Wavelet Scattering" (DSP System Toolbox)
- "Maritime Clutter Removal with Neural Networks" (Radar Toolbox)
- "Signal Recovery with Differentiable Scalograms and Spectrograms" (Signal Processing Toolbox)
- "Signal Source Separation Using W-Net Architecture" (Signal Processing Toolbox)

Audio Processing: New examples

New examples for audio processing tasks include:

- "Audio Transfer Learning Using Experiment Manager"
- "Audio-Based Anomaly Detection for Machine Health Monitoring"
- "Train 3-D Speech Enhancement Network Using Deep Learning"
- "3-D Speech Enhancement Using Trained Filter and Sum Network"

Wireless Communications: New examples

New examples for wireless applications include:

- "Train DQN Agent for Beam Selection"
- "CSI Feedback with Autoencoders"

Deployment: New examples

New deployment examples include:

- "Deploy Object Detection Model as Microservice" (MATLAB Compiler SDK)

R2022a

Version: 14.4

New Features

Bug Fixes

Compatibility Considerations

Apps and Visualization

Experiment Manager: Offload experiments as batch jobs in a cluster

Starting in R2022a, **Experiment Manager** supports offloading experiments as batch jobs in a cluster. You can configure the cluster to run multiple trials at the same time or to run a single trial at a time on multiple parallel workers. While the experiment is running in the cluster, you can run other experiments, close the app and continue using MATLAB, or close your MATLAB session. Batch execution of experiments requires Parallel Computing Toolbox. For more information, see Offload Experiments as Batch Jobs to Cluster.

Experiment Manager: Manage experiments using new context menu options

In R2022a, initiate actions directly in the **Experiment Browser** pane and in the results table:

- To add a new experiment to a project, in the **Experiment Browser** pane, right-click the name of the project and select **New Experiment**.
- To create a copy of an experiment, in the **Experiment Browser** pane, right-click the name of the experiment and select **Duplicate**.
- To stop a running trial, cancel a queued trial, or restart a stopped or canceled trial, in the results table, right-click the row for the trial and select **Stop**, **Cancel**, or **Restart**. Alternatively, click the **Stop**, **Cancel**, or **Restart** buttons in the **Actions** column of the results table.
- To export the training information or trained network for a stopped or completed trial, in the results table, right-click the row for the trial and select **Export Training Information** or **Export Trained Network**.

Experiment Manager: Specify hyperparameters using character vectors

You can now specify hyperparameter values as cell arrays of character vectors. Before R2022a, Experiment Manager only supported hyperparameter specifications using scalars and vectors with numeric, logical, or string values. Now, these are valid hyperparameter specifications:

- `0.01`
- `0.01:0.01:0.05`
- `[0.01 0.02 0.04 0.08]`
- `["sgdm" "rmsprop" "adam"]`
- `{'squeezenet' 'googlenet' 'resnet18'}`

Experiment Manager: View stopping reasons in results table

When an experiment trial ends, the **Status** column of the results table now displays one of these reasons for stopping:

- `Max epochs completed`
- `Met validation criterion`

- `Stopped by OutputFcn`
- `Training loss is NaN`

For an example of an experiment that displays multiple stopping reasons, see Experiment with Weight Initializers for Transfer Learning.

Experiment Manager: Export results table to MATLAB workspace

Starting in R2022a, you can save the contents of the results table as a `table` array in the MATLAB workspace. On the **Experiment Manager** toolstrip, select **Export > Results Table**. For an example that shows how to export the results table as part of evaluating an experiment, see Evaluate Deep Learning Experiments by Using Metric Functions.

Experiment Manager: Sort your experiment annotations

You can now sort annotations. By default, annotations are sorted from oldest to newest creation time. To sort annotations, use the **Sort By** list. You can sort by creation time or trial number. For more information, see Sort, Filter, and Annotate Experiment Results.

Deep Network Designer: Create deep learning experiments suitable for Experiment Manager

Use **Deep Network Designer** to create deep learning experiments suitable for hyperparameter sweeping in **Experiment Manager**. After you train a network using **Deep Network Designer**, create an experiment by clicking **Export > Create Experiment**. **Deep Network Designer** generates an experiment setup function using your network and the imported data. You can use the generated setup function as a starting point for an **Experiment Manager** experiment. For example, sweep through a range of hyperparameter values or use Bayesian optimization to find optimal training options. For more information, see Generate Experiment Using Deep Network Designer.

Deep Network Designer: Access pretrained audio networks

You can now access the following pretrained audio networks from the Deep Network Designer Start Page:

- `crepe` (Audio Toolbox)
- `openl3` (Audio Toolbox)
- `vggish` (Audio Toolbox)
- `yamnet` (Audio Toolbox)

These networks require Deep Learning Toolbox and Audio Toolbox™.

You can use Deep Network Designer to visualize, edit, and train using the pretrained audio networks. For an example showing how to retrain a pretrained audio classification network for a new task, see Transfer Learning with Pretrained Audio Networks in Deep Network Designer.

Deep Network Designer: Export training plot as image

Export the deep learning training progress plot as an image using **Deep Network Designer**. After training, you can save the training progress plot by clicking **Export Training Plot** in the **Training**

tab. You can save the plot as a PNG, JPEG, or TIFF file. You can also save the individual plots of loss, accuracy, and root mean squared error using the axes toolbar.

Network Analyzer: View dimension labels and total number of learnables

View the activation dimension labels and the total number of learnable parameters of a network using **Network Analyzer**. To analyze your network, use the `analyzeNetwork` function or click **Analyze** in **Deep Network Designer**.

Each activation dimension has one of the following labels:

- S — Spatial
- C — Channel
- B — Batch observations
- T — Time or sequence
- U — Unspecified

View the dimension labels to understand how data propagates through the network and how the layers modify the size and layout of activations.

Training Progress Plot: Export training plot as image

Export the deep learning training progress plot as an image or PDF. You can view the training progress plot when using `trainNetwork` by setting the `Plots` training option to `'training-progress'`. To save the training progress plot, click **Export Training Plot** in the training window. You can save the plot as a PNG, JPEG, TIFF, or PDF file. You can also save the individual plots of loss, accuracy, and root mean squared error using the axes toolbar.

Shallow Neural Networks: Improved visual design of network diagram and training window

The `view` function and the network training window have been updated with new visual designs. The training window opens by default when using any of the shallow neural network training functions (for example, `trainlm`, `trainscg`, `trainbr`). For example, use the `train` function to train a simple feedforward network and view the updated training window.

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

Functionality being removed or changed

nntraintool has been removed
Errors

`nntraintool` has been removed. To train a shallow neural network and open the training window, use `train` instead.

nntool has been removed
Errors

`nntool` has been removed. Use `nnstart` instead.

Algorithms

1-D Convolutional Networks: Create and train networks with 1-D transposed convolution for sequence and time series data

Create and train deep learning networks with 1-D transposed convolution layers for sequence and time series data.

Create a 1-D transposed convolution layer using the `transposedConv1dLayer` function. The function returns a `TransposedConvolution1DLayer` object.

The dimension that the layer upsamples depends on the layer input:

- For time series and vector sequence input (data with three dimensions corresponding to the channels, observations, and time steps, respectively), the layer upsamples the time dimension.
- For 1-D image input (data with three dimensions corresponding to the spatial pixels, channels, and observations, respectively), the layer upsamples the spatial dimension.
- For 1-D image sequence input (data with four dimensions corresponding to the spatial pixels, channels, observations, and time steps, respectively), the layer upsamples the spatial dimension.

For custom training loop workflows, you can also apply the 1-D transposed convolution operation to `dlarray` data using the `dltranspconv` function by specifying formatted weights with a "T" (time) dimension or using the `WeightsFormat` option. For more information, use the command `help dlarray/dltranspconv`.

For an example showing how to do anomaly detection using transposed convolutions, see Time Series Anomaly Detection Using Deep Learning.

Batch Normalization: Normalize mini-batches of 1-D images and 1-D, 2-D, and 3-D image sequence input

Normalize mini-batches of 1-D images and 1-D, 2-D, and 3-D image sequence input using `BatchNormalizationLayer` objects. To create a batch normalization layer, use `batchNormalizationLayer`.

Spatio-Temporal Convolution and Pooling: Apply 2-D and 3-D convolution and pooling to sequences of images

Apply 2-D convolutions and pooling to sequences of 1-D or 2-D images using `Convolution2DLayer` and `GlobalAveragePooling2DLayer` objects.

The dimensions that the layer convolves over depend on the layer input:

- For 2-D image input (data with four dimensions corresponding to pixels in two spatial dimensions, the channels, and the observations), the layer convolves or pools over the spatial dimensions.
- For 2-D image sequence input (data with five dimensions corresponding to the pixels in two spatial dimensions, the channels, the observations, and the time steps), the layer convolves or pools over the two spatial dimensions.

- For 1-D image sequence input (data with four dimensions corresponding to the pixels in one spatial dimension, the channels, the observations, and the time steps), the layer convolves or pools over the spatial and time dimensions.

Apply 3-D convolutions and pooling to sequences of 2-D or 3-D images using `Convolution3DLayer`, `MaxPooling3DLayer`, `AveragePooling3DLayer`, and `GlobalAveragePooling3DLayer` objects.

The dimensions that the layer convolves or pools over depends on the layer input:

- For 3-D image input (data with five dimensions corresponding to pixels in three spatial dimensions, the channels, and the observations), the layer convolves or pools over the spatial dimensions.
- For 3-D image sequence input (data with six dimensions corresponding to the pixels in three spatial dimensions, the channels, the observations, and the time steps), the layer convolves or pools over the spatial dimensions.
- For 2-D image sequence input (data with five dimensions corresponding to the pixels in two spatial dimensions, the channels, the observations, and the time steps), the layer convolves or pools over the spatial and time dimensions.

Network Training: Specify checkpoint frequency

The `CheckpointPath` training option enables you to save networks as MAT files periodically during training. This periodic saving is especially useful when you have a large network or a large data set, and training takes a long time. If the training is interrupted for some reason, you can resume training from the last saved checkpoint network.

You can now specify how often the software saves checkpoint networks. To specify the number of iterations or epochs between saving checkpoints, use the `CheckpointFrequency` training option. To specify the checkpoint frequency unit, use the `CheckpointFrequencyUnit` training option.

Network Training: Train networks with sequence input in parallel

The `trainNetwork` function supports training networks with sequence input in parallel. To train a network in parallel, set the `ExecutionEnvironment` training option to `'parallel'` or `'multi-gpu'` using the `trainingOptions` function.

To train a networks with `lstmLayer`, `bilstmLayer`, or `gruLayer` objects in parallel, the `SequenceLength` training option must be `'longest'` or `'shortest'`.

Multi-Input Networks: Train networks with mixtures of image, sequence, or feature inputs

The `trainNetwork` function now supports layer graphs with mixtures of `ImageInputLayer`, `Image3DInputLayer`, `SequenceInputLayer`, and `FeatureInputLayer` layer objects. When using the `trainNetwork` function, the network must have at most one sequence input layer.

When training a multi-input network with a sequence input layer, the `SequenceLength` training option must be `'longest'` or `'shortest'`.

When making network predictions using the `predict`, `classify`, `predictAndUpdateState`, `classifyAndUpdateState`, or `activations` functions, the `SequenceLength` option must be

`'longest'` or `'shortest'` and the `ExecutionEnvironment` option must be `'auto'`, `'gpu'`, or `'cpu'`.

To specify training data with multiple inputs, use a transformed or combined datastore. To transform the outputs of a datastore, use the `transform` function. To create a datastore that outputs multiple values by combining datastores, use the `combine` function.

For an example showing how to train a network with both image and feature input, see Train Network on Image and Feature Data.

Deep Learning Model Hub: Discover pretrained models for deep learning in MATLAB

To find the latest pretrained models for deep learning in MATLAB, see MATLAB Deep Learning Model Hub.

You can find models suitable for a range of deep learning applications, such as lidar point cloud processing, audio speech to text, and pose estimation from images. For example:

- Find transformer models, such as GPT-2, BERT, and FinBERT, suitable for natural language processing tasks.
- Find models such as YOLO v4 and Mask R-CNN, suitable for object detection tasks.

Flatten Layer: Use flatten layers in networks with image or feature input

Networks with image or feature input now support `FlattenLayer` objects. Use flatten layers to collapse the spatial dimensions of the layer input into the channel dimension.

Function Layer: Generate code for function layer

Function layers with unformatted inputs support code generation and GPU code generation when the predict function is a named function on the path.

To create a function layer that applies a specified function to it's input, use `functionLayer`.

For an example showing how to generate code for an image classification network, see Code Generation for Deep Learning Networks.

Custom Layers: Accelerate custom layer functions

When you create a custom layer, if you do not specify a backward function, then the software automatically determines the gradients using automatic differentiation.

When training a network with a custom layer without a backward function, the software traces each input `dlarray` object of the custom layer forward function to determine the computation graph used for automatic differentiation. This tracing process can take some time and can spend time recomputing the same trace. By optimizing, caching, and reusing the traces, you can speed up gradient computation when training a network. The software can also reuse these traces to speed up network predictions after training.

To indicate that the custom layer supports acceleration, also inherit from the `nnet.layer.Acceleratable` class when defining the custom layer. When a custom layer inherits from `nnet.layer.Acceleratable`, the software automatically caches traces when passing data through a `dlnetwork` object.

For more information, see Custom Layer Function Acceleration. To learn more about defining custom layers, see Define Custom Deep Learning Layers.

Recurrent Layers: Recurrent layers in dlnetwork objects support inputs without time dimensions

When included in a `dlnework` object, LSTMLayer, BiLSTMLayer, and GRULayer objects automatically infer a singleton time dimension when the input data does not specify a time dimension. In these cases, the layer output does not contain time dimensions.

For more information, see the Layer Input and Output Formats sections of `lstmLayer`, `bilstmLayer`, and `gruLayer`.

Custom Training Loops: dlnetwork objects support TransposedConvolution1DLayer objects

Create `dlnetwork` objects containing `TransposedConvolution1DLayer` objects. To create a 1-D transposed convolution layer, use the `transposedConv1dLayer` function.

Custom Training Loops: Apply transposed convolution over time dimension of dlarray

For custom training loop workflows, you can apply the transposed convolution operation to `dlarray` data over the time dimension using the `dltranspconv` function by specifying formatted weights with a "T" (time) dimension or using the `WeightsFormat` option. For more information, use the command `help dlarray/dltranspconv`.

Custom Training Loops: Reset state parameters of dlnetwork objects

The `resetState` function now supports `dlnetwork` input.

The `resetState` function has an effect only if the input network has state parameters (for example, a network with at least one recurrent layer, such as an LSTM layer). If the input network does not have state parameters, then the function has no effect and returns the input network.

Custom Training Loops: Plot dlarray objects

The `plot` function and the `addpoints` function for `animatedLine` objects support `dlarray` data as input.

Plotting functions do not support tracing `dlarray` objects.

For more information about support for `dlarray` objects, see List of Functions with dlarray Support.

fullyconnect Function: Improved single-precision performance with GPUs using the TensorFloat-32 (TF32) compute mode

The `fullyconnect` function shows improved single-precision performance with GPUs using the TF32 compute mode, such as NVIDIA Ampere architecture GPUs. For example, computing the weighted sum of input data using the following test is about 3.4x faster than in the previous release:

```
function timeFullyConnect

% Set up random single-precision input data, weights, and biases
inputSize = 10000;
observations = 100000;
outputFeatures = 5000;

X  = dlarray(gpuArray(rand(inputSize,observations,'single')),'SB');
weights = gpuArray(dlarray(rand(outputFeatures,inputSize,'single')));
bias    = gpuArray(dlarray(rand(outputFeatures,1,'single')));


% Time fullyconnect
f = @() fullyconnect(X,weights,bias);
gputimeit(f)

end
```

The approximate execution times are:

R2021b: 0.88 seconds

R2022a: 0.26 seconds

The code was timed on a Windows 10, Intel Xeon W-2133 @ 3.60 GHz test system with NVIDIA RTX A5000 GPU by calling the function `timeFullyConnect`.

max Function: Improved performance within a dlgradient call

The `max` function shows improved performance when used within a `dlgradient` call. The `min` function shows a similar performance improvement when used with a `dlgradient` call. For example, evaluating gradients containing a `max` function in the following test is about 4.9x faster than in the previous release:

```
function timeMax

% Prepare input data and gradient function
x = dlarray(rand(220, 200, 3, 100)-0.5);
fcn = @(x) max(x,0);
gradFcn = @(x) dlgradient(sum(fcn(x), 'all'), x);

% Warm up iterations
for i=1:5
    dlfeval(gradFcn, x);
end


% Timed iterations
tic
```

```
for i=1:10
    dlfeval(gradFcn, x);
end
toc

end
```

The approximate execution times are:

R2021b: 2.26 seconds

R2022a: 0.46 seconds

The code was timed on a Windows 10, Intel Xeon W-2133 @ 3.60 GHz test system with NVIDIA RTX A5000 GPU by calling the function `timeMax`.

Accelerated Functions: Improved performance of accelerated functions that use parentheses indexing with non-repeated indices

Accelerated functions that use parentheses indexing with non-repeated indices show improved performance. For example, evaluating an accelerated convolution operation in the following test is about 7.0x faster than in the previous release:

```
function timeAccFcn

% Prepare input data
inputSize = [256,256,64,90];
X = dlarray(gpuArray(randn(inputSize)));
W = dlarray(gpuArray(randn(3,3,1,1,64)));
bias = dlarray(gpuArray(randn(1)));

% Prepare convolution function
fcn = @(x, W, bias) dlconv(double(squeeze(x(:, :, :, 2, :)-x(:, :, :, 1, :))), W, bias, ...
    'Stride', [1 1], 'Padding', 'same', 'DataFormat', 'SSCB');
accFcn = dlaccelerate(fcn);
clearCache(accFcn);

% Warm up iterations
for i = 1:6
    dlfeval(accFcn,X, W, bias);
end

% Timed iterations
gd = gpuDevice;
tic
for i = 1:200
    dlfeval(accFcn,X, W, bias);
    wait(gd);
end
toc

end
```

The approximate execution times are:

R2021b: 2.23 seconds

R2022a: 0.32 seconds

The code was timed on a Windows 10, Intel Xeon W-2133 @ 3.60 GHz test system with NVIDIA RTX A5000 GPU by calling the function `timeAccFcn`.

dlnetwork State Updates: Improved performance when updating the network state of a dlnetwork

Updating the `State` property of a `dlnetwork` shows improved performance. For example, updating the network state of the Resnet-50 network in the following test is about 5.5x faster than in the previous release:

```matlab
function timeStateUpdate

% Prepare network and input variables
lgraph=layerGraph(resnet50);
lgraph = removeLayers(lgraph, {lgraph.Layers(end-2:end).Name});
net = dlnetwork(lgraph);
x = gpuArray(dlarray(rand(10, 10, 3, 10)-0.5, 'SSCB'));

% Calculate a new state
[~, newState] = forward(net, x);

% Timed iterations
tic
for i=1:10
    net.State = newState;
end
toc

end
```

The approximate execution times are:

R2021b: 1.2 seconds

R2022a: 0.22 seconds

The code was timed on a Windows 10, Intel Xeon W-2133 @ 3.60 GHz test system with NVIDIA RTX A5000 GPU by calling the function `timeStateUpdate`.

forward Function: Reduced GPU memory usage when accelerated

Computing the output of a `dlnetwork` using the `forward` function and specifying the `acceleration` input as `'auto'` uses less GPU memory.

Import and Export

ONNX Version Support: Updated support for ONNX intermediate representation and operator sets

The `importONNXNetwork`, `importONNXLayers`, and `exportONNXNetwork` functions now support ONNX™ intermediate representation version 7 and ONNX operator sets 6 to 14.

TensorFlow-Keras and ONNX Code Generation: Additional Keras and ONNX built-in layers support code generation

You can use MATLAB Coder™ or GPU Coder™ together with Deep Learning Toolbox to generate MEX, standalone CPU, CUDA® MEX, or standalone CUDA code for an imported network. You can generate code for any imported network whose layers support code generation. For lists of the layers that support code generation with MATLAB Coder and GPU Coder, see Supported Layers (MATLAB Coder) and Supported Layers (GPU Coder), respectively.

You can now generate generic C or C++ code for the following Keras and ONNX built-in layers:

- `nnet.keras.layer.GlobalAveragePooling2dLayer`
- `nnet.keras.layer.FlattenCStyleLayer`
- `nnet.keras.layer.ZeroPadding2dLayer`
- `nnet.onnx.layer.ElementwiseAffineLayer`
- `nnet.onnx.layer.FlattenLayer`

You can now generate MEX, standalone CPU, CUDA MEX, or standalone CUDA code for the following Keras and ONNX built-in layers:

- `nnet.keras.layer.ClipLayer`
- `nnet.keras.layer.PreluLayer`
- `nnet.keras.layer.TimeDistributedFlattenCStyleLayer`
- `nnet.onnx.layer.ClipLayer`
- `nnet.onnx.layer.GlobalAveragePooling2dLayer`
- `nnet.onnx.layer.PreluLayer`
- `nnet.onnx.layer.SigmoidLayer`
- `nnet.onnx.layer.TanhLayer`

ONNX Export Support: Specify batch size of exported network

You can now specify a dynamic or fixed batch size for a trained MATLAB deep learning network that you export to the ONNX model format. To do so, use the `BatchSize` name-value argument of `exportONNXNetwork`.

ONNX Import Layer Support: Import networks that include 1-D convolution and pooling layers

You can now import an ONNX network that includes 1-D convolution and pooling layers (`convolution1dLayer`, `maxPooling1dLayer`, `averagePooling1dLayer`,

globalMaxPooling1dLayer, and globalAveragePooling1dLayer) by using importONNXNetwork and importONNXLayers. For a list of supported layers, see ONNX Operators Supported for Conversion into Built-In MATLAB Layers.

TensorFlow Operator Support: Import networks that include ExpandDims operators

You can now import a TensorFlow network that includes ExpandDims operators by using the importTensorFlowNetwork and importTensorFlowLayers functions. For a list of the TensorFlow operators that the functions support for conversion into MATLAB functions with dlarray support, see Supported TensorFlow Operators.

Deployment

Acceleration Modes: Use accelerator and rapid accelerator modes to speedup Simulink simulations

Deep learning Simulink models now support the accelerator and rapid accelerator modes to speed up the execution of your model.

The accelerator modes use the Intel MKL-DNN library to perform acceleration. In the **Model Configuration Parameters**, on the **Simulation Target** pane, set the **Language** to C++ and the **Target library** to MKL-DNN. Then on the **Simulation** tab, in the **Simulate** section, select Accelerator or Rapid Accelerator from the drop-down menu and start the simulation.

For information on improving simulation speed with accelerator and rapid accelerator modes, Acceleration for Simulink Deep Learning Models.

For an example showing how to speed up the execution of your model, see Lane and Vehicle Detection in Simulink Using Deep Learning.

Quantization: Quantize neural networks without a specified target

With MATLAB, you can quantize your neural networks without generating code or committing to a specific target for code deployment. This can be useful if you:

- Do not have access to your target hardware.
- Want to inspect your quantized network without generating code.

Your quantized network implements int8 data instead of single data. It keeps the same layers and connections as the original network, and it has the same inference behavior as it would when running on hardware.

Once you have quantized your network, you can use the quantizationDetails function to inspect your quantized network. Additionally, you also have the option to deploy the code to a GPU target.

For an example showing how to quantize your neural network with MATLAB, see Emulate Target Agnostic Quantized Network.

Quantization: Estimate neural network layer metrics

With the estimateNetworkMetrics function, you can estimate the metrics for each layer of your neural network. For more information, see estimateNetworkMetrics.

Quantization: Validate the performance of the optimized network for a CPU target

You can now use the dlquantizer object and the validate function to quantize a network and generate code for CPU targets. Additionally, the **Deep Network Quantizer** app now fully supports the workflow for a CPU target.

Taylor Pruning: Prune dlnetwork object to compress the model

You can prune a `dlnetwork` object by using the first-order Taylor approximation algorithm. This process identifies and removes filters from the prunable layers of the `dlnetwork` object. Pruning is an optimization that returns another `dlnetwork` object that is a compressed version of your original model and that consumes less computational resources.

Use the `taylorPrunableNetwork` function to convert a `dlnetwork` object to another representation that is suitable for pruning using the Taylor pruning algorithm.

To perform pruning, use these object functions:

- `forward` and `predict`: Perform training and inference respectively.
- `updateScore` and `updatePrunables`: Compute and update first-order Taylor scores. Remove filters from prunable layers.
- `dlnetwork`: Extract the compressed `dlnetwork` object back from the pruned `TaylorPrunableNetwork` object.

For examples showing how to prune the convolutional filters of a network using Taylor pruning, see:

- Prune Image Classification Network Using Taylor Scores
- Prune Filters in a Detection Network Using Taylor Scores

TensorFlow Lite: Generate C++ code for pretrained models and deploy on Linux platforms

Use the `loadTFLiteModel` function to load a pretrained TensorFlow Lite model into a `TFLiteModel` object. Use this object with the `predict` function in your MATLAB code to perform inference in MATLAB execution, code generation, or inside MATLAB Function blocks in Simulink models.

To use this functionality, you must install the Deep Learning Toolbox Interface for TensorFlow Lite. For more information, see Prerequisites for Deep Learning with TensorFlow Lite Models. For an example, see Generate Code for TensorFlow Lite Model and Deploy on Raspberry Pi.

Application Examples

Deep Learning Workflows: New and updated examples and topics

New and updated examples and topics help you progress with deep learning:

- Time Series Forecasting Using Deep Learning
- Deep Learning in MATLAB
- Detect Vanishing Gradients in Deep Neural Networks by Plotting Gradient Distributions
- Generate Experiment Using Deep Network Designer
- Custom Training with Multiple GPUs in Experiment Manager
- Sequence-to-One Regression Using Deep Learning
- Sequence Classification Using Inverse-Frequency Class Weights
- Train Network on Image and Feature Data
- Time Series Anomaly Detection Using Deep Learning
- Generate Adversarial Examples for Semantic Segmentation
- Multilabel Image Classification Using Deep Learning
- Predict Battery State of Charge Using Deep Learning
- Multilabel Graph Classification Using Graph Attention Networks
- Train a Network on Amazon Web Services Using MATLAB Deep Learning Container
- Use Amazon S3 Buckets with MATLAB Deep Learning Container
- Use Experiment Manager in the Cloud with MATLAB Deep Learning Container
- Prune Image Classification Network Using Taylor Scores

Import and Export: New Examples

New examples help you import networks from external deep learning platforms:

- Inference Comparison Between TensorFlow and Imported Networks for Image Classification
- Inference Comparison Between ONNX and Imported Networks for Image Classification

Image Processing and Computer Vision: New and updated examples

New and updated examples for image processing and computer vision tasks include:

- Detect Image Anomalies Using Explainable One-Class Classification Neural Network
- Detect Image Anomalies Using Pretrained ResNet-18 Feature Embeddings
- Classify Defects on Wafer Maps Using Deep Learning
- Object Detection Using YOLO v4 Deep Learning
- Prune Filters in a Detection Network Using Taylor Scores

Lidar Processing: New and updated examples

New and updated examples for lidar processing workflows include:

- Lidar Object Detection Using Complex-YOLO v4 Network
- Code Generation for Lidar Object Detection Using SqueezeSegV2 Network
- Aerial Lidar Semantic Segmentation Using PointNet++ Deep Learning

Audio Processing: New examples

New examples for audio processing tasks include:

- Transfer Learning with Pretrained Audio Networks in Deep Network Designer
- Investigate Audio Classifications Using Deep Learning Interpretability Techniques
- Train 3-D Sound Event Localization and Detection (SELD) Using Deep Learning
- 3-D Sound Event Localization and Detection Using Trained Recurrent Convolutional Neural Network
- Speech Command Recognition Code Generation with Intel MKL-DNN Using Simulink
- Speech Command Recognition on Raspberry Pi Using Simulink

Signal Processing: New examples

New examples for signal processing tasks include:

- Denoise Signals with Adversarial Learning Denoiser Model (Signal Processing Toolbox)

Wireless Communications: New examples

New examples for wireless applications include:

- Neural Network for Digital Predistortion Design - Offline Training
- Neural Network for Beam Selection

Computational Finance: New examples

New examples for computational finance applications include:

- Use Deep Learning to Approximate Barrier Option Prices with Heston Model
- Backtest Strategies Using Deep Learning

Simulink: New examples

New examples for Simulink tasks include:

- Physical System Modeling Using LSTM Network in Simulink
- Battery State of Charge Estimation in Simulink Using Deep Learning Network

R2021b

Version: 14.3

New Features

Bug Fixes

Compatibility Considerations

Experiment Manager: Use Bayesian optimization in custom training experiments

With Experiment Manager, you can now use Bayesian optimization to determine the best combination of hyperparameters for a custom training experiment. Previously, custom training experiments only supported sweeping hyperparameters. Bayesian optimization requires Statistics and Machine Learning Toolbox. For more information, see Use Bayesian Optimization in Custom Training Experiments.

Experiment Manager: Run deep learning experiments in your web browser using MATLAB Online

Starting in R2021b, you can use Experiment Manager in MATLAB Online. To run an experiment in parallel using MATLAB Online, you must have access to a Cloud Center cluster. For more information, see Use Parallel Computing Toolbox with Cloud Center Cluster in MATLAB Online (Parallel Computing Toolbox).

Experiment Manager: Improved accessibility with keyboard shortcuts

You can now use keyboard shortcuts to navigate Experiment Manager when using a mouse is not an option. For more information, see Keyboard Shortcuts for Experiment Manager.

Experiment Manager: Stop experiments faster by discarding the results of running trials

When stopping a built-in training experiment that uses exhaustive sweep, you now have the option of discarding the results of any running trials. Experiment Manager provides two options for stopping experiments:

- **Stop** marks any running trials as `Stopped` and saves their results. When the experiment stops, you can display the training plot and export the training output for these trials.

- **Cancel** marks any running trials as `Canceled` and discards their results. When the experiment stops, you cannot display the training plot or export the training output for these trials.

Both options save the results of any completed trials and cancel any queued trials. Typically, **Cancel** is faster than **Stop**. For more information, see Stop and Restart Training.

Simulink Blocks: Simulate and generate code for deep learning object detectors

Simulate and generate code for deep learning object detectors in Simulink. The `Analysis & Enhancement` block library from Computer Vision Toolbox™ now includes the Deep Learning Object Detector (Computer Vision Toolbox) block. This block predicts bounding boxes, class labels, and scores for the input image data by using a specified trained object detector. This block enables you to load a pretrained object detector into the Simulink model from a MAT file or a MATLAB function.

For more information about working with the Deep Learning Object Detector block, see Lane and Vehicle Detection in Simulink Using Deep Learning. To learn more about using deep learning with Simulink, see Deep Learning with Simulink.

Deep Network Designer: Export trained network to Simulink

Export networks trained in Deep Network Designer to Simulink. Export a trained network from the **Training** tab by clicking **Export > Export to Simulink**.

Deep Network Designer: Analyze for dlnetwork

Deep Network Designer now supports analyzing networks for usage with `dlnetwork` objects. For example, analyzing for `dlnetwork` objects checks that the network does not have any output layers. Use this feature when you train a network outside of Deep Network Designer, by using a custom training loop.

To analyze a network for `dlnetwork` usage, click **Analyze > Analyze for dlnetwork**. Doing so is equivalent to using the `analyzeNetwork` function with the name-value argument `TargetUsage` set to `"dlnetwork"` (available in R2021a).

1-D Convolutional Networks: Create and train networks with 1-D convolution and pooling layers for sequence and time-series data

Create and train deep learning networks with 1-D convolution and pooling layers for sequence and time series data.

Create networks using the following layers:

- `convolution1dLayer`
- `averagePooling1dLayer`
- `maxPooling1dLayer`
- `globalAveragePooling1dLayer`
- `globalMaxPooling1dLayer`

The dimension that the layers convolve or pool over depends on the layer input:

- For time series and vector sequence input (data with three dimensions corresponding to the channels, observations, and time steps, respectively), the layer convolves or pools over the time dimension.
- For 1-D image input (data with three dimensions corresponding to the spatial pixels, channels, and observations, respectively), the layer convolves or pools over the spatial dimension.
- For 1-D image sequence input (data with four dimensions corresponding to the spatial pixels, channels, observations, and time steps, respectively), the layer convolves or pools over the spatial dimension.

For an example showing how to train a sequence-to-label classification network using 1-D convolutions, see Sequence Classification Using 1-D Convolutions.

For an example showing how to train a sequence-to-sequence classification network using 1-D convolutions, see Sequence-to-Sequence Classification Using 1-D Convolutions.

1-D Convolutional Networks: Specify minimum sequence length

When you create a network that downsamples data in the time dimension, you must take care that the network supports your training data and any data for prediction. Some deep learning layers require that the input has a minimum sequence length. For example, a 1-D convolution layer requires that the input has at least as many time steps as the filter size.

As a time series of sequence data propagates through a network, the sequence length can change. For example, downsampling operations such as 1-D convolutions can output data with fewer time steps than the input. Consequently, downsampling operations can cause later layers in the network to throw an error because the data has a shorter sequence length than the minimum length required by the layer.

When you train or assemble a network, the software automatically checks that sequences of length 1 can propagate through the network. Some networks might not support sequences of length 1, but can successfully propagate sequences of longer lengths. To check that a network supports propagating your training or prediction data, set the `MinLength` property of `sequenceInputLayer` to a value less than or equal to the minimum length of your data and the expected minimum length of your prediction data.

Recurrent Neural Networks: Pass recurrent layer states between layers

Pass recurrent states to and from recurrent layers by connecting them to the state input and outputs of the layer.

When you use `lstmLayer`, `bilstmLayer`, or `gruLayer`, to specify that the layer has state inputs, set the `HasStateInputs` property of the layer to `1` (true). To specify that a recurrent layer has state outputs, set the `HasStateOutputs` property of the layer to `1` (true).

For an example showing how to train a neural network for language translation that passes the recurrent state of an LSTM layer between layers, see Language Translation Using Deep Learning.

Network Training: Create layer graphs without specifying layer names

Starting in R2021b, specifying layer names when you create a layer graph is optional.

Network Training: Return network with lowest validation loss

When training a neural network using the `trainNetwork` function, output the network with the lowest validation loss by setting the `OutputNetwork` name-value argument of the `trainingOptions` function to `"best-validation-loss"`.

Network Analyzer: Use example inputs when analyzing networks for custom training workflows

You can now provide example network inputs when you use the `analyzeNetwork` function to analyze networks for custom training workflows.

Use example inputs when you want to analyze a network that has inputs that are unconnected to an input layer. The software propagates the example inputs through the network to determine the number and sizes of layer activations, learnable parameters, and state parameters.

Provide example network inputs as formatted `dlarray` objects with the same size and format as typical inputs for your network. For more information, see `analyzeNetwork`.

MEX Acceleration: Use MEX acceleration with multi-input and multi-output networks

Use automatically generated MEX functions to accelerate prediction, classification, and feature extraction of multi-input and multi-output `DAGNetwork` objects. You can apply MEX acceleration using the name-value option `Acceleration="mex"` in the following functions:

- `activations`
- `classify`
- `predict`

Residual Networks: Easily create 2-D and 3-D residual networks

Create residual networks for 2-D and 3-D image classification using `resnetLayers` and `resnet3dLayers`.

Neural Network Apps: New toolstrip design for improved usability

The **Neural Net Fitting** (`nftool`), **Neural Net Pattern Recognition** (`nprtool`), **Neural Net Clustering** (`nctool`), and **Neural Net Time Series** (`ntstool`) apps have been updated with new user interfaces. The new toolstrip design makes the apps easier to use and speeds up your workflow.

Function Layer: Create layers that apply a function to the input

A function layer applies a specified function to the layer input. If Deep Learning Toolbox does not provide the layer that you need for your task, then you can define new layers by creating function layers using `functionLayer`.

For example, to create a layer that applies the `exp` function to the input, use the following.

```
layer = functionLayer(@exp);
```

For an example showing how to import the layers of a pretrained TensorFlow-Keras network and replace the unsupported softsign layers with a function layer, see Replace Unsupported Keras Layer with Function Layer.

Function layers only support operations that do not require additional properties, learnable parameters, or states. For layers that require this functionality, define the layer as a custom layer. For more information, see Define Custom Deep Learning Layers.

Parallel Inference: Predict, classify, and extract features in parallel with DAGNetwork and SeriesNetwork objects

You can now use multiple GPUs or multiple CPUs to accelerate prediction, classification, and feature extraction with `DAGNetwork` and `SeriesNetwork` objects. Previously, prediction, classification, and feature extraction supported single GPU or single CPU execution only.

The following functions now support running in parallel:

- `activations`
- `classify`
- `predict`

To run in parallel, set the `ExecutionEnvironment` name-value option to `"multi-gpu"` or `"parallel"`. The `"multi-gpu"` option allows you to process mini-batches of data in parallel using multiple GPUs in your local machine. The `"parallel"` option allows you to process data in parallel using a local or remote cluster. If you are using a remote cluster that has access to multiple GPUs, use the `"parallel"` option.

The `"multi-gpu"` and `"parallel"` options do not support recurrent neural networks (RNNs) containing `lstmLayer`, `bilstmLayer`, or `gruLayer` objects.

Parallel Training: Improved instructions for deep learning in the cloud

MathWorks® provides several ways of accessing MATLAB in public clouds such as Amazon® Web Services (AWS®) and Azure® that are configurable depending on your needs.

These cloud offerings make it easy for you to run your MATLAB deep learning applications in the cloud. For more information, see Deep Learning in the Cloud.

Custom Layers: Define stateful custom layers

Define stateful custom layers by specifying the `State` attribute in a `properties` block and specifying the state parameters in the `predict` and the optional `forward` and `backward` functions.

For nested layer workflows (custom layers with `dlnetwork` objects as learnable parameters), if the nested network is stateful, then specify the network parameter in a properties block with both the attributes `Learnable` and `State`.

For an example showing how to define a custom peephole LSTM layer, see Define Custom Recurrent Deep Learning Layer.

For more information about custom layers, see Define Custom Deep Learning Layers.

Custom Training Loops: Apply neural ODE operations

The neural ordinary differential equation (ODE) operation returns the solution of a specified ODE. In particular, given an input, a neural ODE operation outputs the numerical solution of the ODE $y' = f(t, y, \theta)$ for the time horizon (t_0, t_1) and with initial condition $y(t_0) = y_0$, where t and y denote the ODE function inputs and θ is a set of learnable parameters. Typically, the initial condition y_0 is either the network input or the output of another deep learning operation.

To apply the neural ODE operation to `dlarray` objects, use the `dlode45` function.

For an example showing how to train an image classification model using an augmented neural ODE, see Train Neural ODE Network. For an example showing how to train a neural network with a neural ODE to learn the dynamics of a physical system, see Dynamical System Modeling Using Neural ODE.

Custom Training Loops: Calculate L1 and L2 loss

Calculate the L_1 and L_2 loss using the `l1loss` and `l2loss` functions, respectively.

The L_1 loss operation computes the L_1 loss given network predictions and target values. When the `Reduction` option is `"sum"` and the `NormalizationFactor` option is `"batch-size"`, the computation yields as the mean absolute error (MAE).

The L_2 loss operation computes the L_2 loss (based on the squared L_2 norm) given network predictions and target values. When the `Reduction` option is `"sum"` and the `NormalizationFactor` option is `"batch-size"`, the computation yields the mean squared error (MSE).

For an example showing how to train a neural network with a neural ODE to learn the dynamics of a physical system by minimizing the L_1 loss, see Dynamical System Modeling Using Neural ODE.

Custom Training Loops: Use MEX acceleration to optimize performance for dlnetwork prediction

Use automatically generated MEX functions to accelerate predictions of `dlnetwork` outputs using the `predict` function. The software generates a MEX function to optimize performance when making multiple calls to the `predict` function.

To enable MEX acceleration for the `predict` function, specify the name-value option `Acceleration="mex"`.

For example, making predictions using the following test is about 3.4x faster when `Acceleration="mex"`.

```
% Load resnet50 and convert to a dlnetwork
lgraph = layerGraph(resnet50);
lgraph = removeLayers(lgraph,"ClassificationLayer_fc1000");
dlnet = dlnetwork(lgraph);

% Create an example image and convert to a formatted dlarray
img = im2single(imread("peppers.png"));
img = imresize(img, [224 224]);
img = gpuArray(img);
X = dlarray(img,"SSCB");

% Time with auto acceleration
tAuto = gputimeit(@()predict(dlnet,X))

tAuto = 0.0273

% Time with mex acceleration
tMex = gputimeit(@()predict(dlnet,X,Acceleration="mex"))

tMex = 0.0082
```

```
speedup = tAuto/tMex
```

```
speedup = 3.3507
```

The code was timed on a Windows 10, Intel Xeon E5-1650 v4 @ 3.60 GHz test system with NVIDIA Titan RTX GPU.

Custom Training Loops: Compute gradients of loss functions involving complex numbers

Use `dlfeval` and `dlgradient` to compute gradients of loss functions involving complex numbers.

Previously, complex functions and gradients were not supported. Now, you can use custom training loops with loss functions that involve complex numbers and gradients. The resulting gradient can be complex. However, the value to differentiate—for example, the loss—must be real, even if the function is complex.

By default, computing complex gradients is enabled. If you want to restrict the inputs and gradients and variables in the function to real numbers, set the name-value option `AllowComplex` to `0` (false). For more information, see `dlgradient`.

Custom Training Loops: Specify network outputs

Specify the network outputs of a `dlnetwork` object using the `OutputNames` property. The `predict` and `forward` functions for `dlnetwork` objects, by default, use the outputs specified by `OutputNames`.

Custom Training Loops: Use flatten layer in dlnetwork objects

`dlnetwork` objects now support layer graphs containing `flattenLayer` objects.

Custom Training Loops: Improved instructions for running custom training loops on GPU and in parallel

Follow improved instructions to speed up custom training loops by running on a GPU, in parallel using multiple GPUs, or on a cluster. For more information, see Run Custom Training Loops on a GPU and in Parallel

TensorFlow Operator Support: Import networks that include Square operators

You can now import a TensorFlow network that includes `Square` operators by using the `importTensorFlowNetwork` and `importTensorFlowLayers` functions. For a list of the TensorFlow operators that the functions support for conversion into MATLAB functions with `dlarray` support, see Supported TensorFlow Operators.

TensorFlow-Keras Layer Support: Import 1-D convolution and pooling layers

`importTensorFlowNetwork`, `importTensorFlowLayers`, `importKerasNetwork`, and `importKerasLayers` can now import networks with TensorFlow-Keras 1-D convolution and pooling

layers. The functions support the conversion of these layers into the `convolution1dLayer`, `maxPooling1dLayer`, `averagePooling1dLayer`, `globalMaxPooling1dLayer`, and `globalAveragePooling1dLayer` built-in Deep Learning Toolbox layers. For a full list of supported layers, see TensorFlow -Keras Layers Supported for Conversion into Built-In MATLAB Layers.

ONNX Import Support: Automatic custom layer generation

The functions `importONNXNetwork` and `importONNXLayers` now try to generate a custom layer when the software cannot convert an ONNX operator into an equivalent built-in MATLAB layer. Specify `GenerateCustomLayers` as `false` to opt out of the automatic custom layer generation. By default, `GenerateCustomLayers` is set to `true`.

The functions save the generated custom layers in a package in the current folder. By default, the software saves the custom layers in a package named `+modelfile`, where `modelfile` is the file containing the ONNX model. Use `PackageName` to specify the name of the custom layers package.

ONNX Import Support: Constant folding optimization

The functions `importONNXNetwork` and `importONNXLayers` can now perform constant folding, which optimizes the imported network architecture by computing operations on ONNX initializers (initial constant values) during the conversion of ONNX operators to equivalent built-in MATLAB layers. If the ONNX network contains operators that the software cannot convert to equivalent built-in MATLAB layers, constant folding optimization can reduce the number of unsupported layers. Use the `FoldConstants` name-value argument to specify constant folding optimization options.

ONNX Import Support: Import ONNX network as a dlnetwork object

The `importONNXNetwork` function can now import an ONNX model as a `dlnetwork` object, when you specify the name-value argument `TargetNetwork` as `"dlnetwork"`. By default, `importONNXNetwork` returns the network `net` as a `DAGNetwork` object.

The `importONNXLayers` function can now import an ONNX model as a `LayerGraph` object compatible with a `dlnetwork` object, when you specify the name-value argument `TargetNetwork` as `"dlnetwork"`. By default, `importONNXLayers` returns a layer graph compatible with a `DAGNetwork` object. For more information on how to convert the imported layer graph to a `DAGNetwork` or `dlnetwork` object, see the output argument `lgraph`.

ONNX Import Support: New and modified options for network inputs and outputs

The functions `importONNXNetwork` and `importONNXLayers` can import an ONNX network with multiple inputs and multiple outputs (MIMO). Use the name-value arguments of the functions to specify options for the network inputs and outputs.

- `importONNXNetwork` can now import networks with multiple outputs, and `importONNXLayers` can now import these networks without inserting placeholder layers for the outputs. The name-value argument `OutputLayerType` specifies the layer type for the first network output.

- The functions now try to derive the type of the output layers from the ONNX file. Use the name-value argument `OutputLayerType` only when the functions cannot derive the output layer type.

- Use the new name-value arguments `InputDataFormats` and `OutputDataFormats` to specify the data format of the network inputs and outputs, respectively, when the functions cannot derive the data formats from the ONNX file.
- Use the new name-value argument `ImageInputSize` to specify the size of a 2-D or 3-D input image for the first network input.

ONNX Export Layer Support: Export networks that include 1-D convolution and pooling layers

You can now export a trained MATLAB deep learning network that includes 1-D convolution and pooling layers (`convolution1dLayer`, `maxPooling1dLayer`, `averagePooling1dLayer`, `globalMaxPooling1dLayer`, and `globalAveragePooling1dLayer`) to the ONNX model format by using `exportONNXNetwork`. For a full list of supported layers, see Layers Supported for ONNX Export.

Automatic Differentiation: Use complex numbers with dlarray

You can now use complex numbers with `dlarray`. Functions that support `dlarray` objects and complex numbers now also support complex `dlarray` objects.

The following complex number functions now support `dlarray` objects:

- `angle`
- `complex`
- `conj`
- `imag`
- `real`
- `reallog`
- `realsqrt`
- `realpow`

Additionally, the inverse trigonometric functions `acos`, `acosh`, `acsc`, `asec`, `asin`, and `atanh` and the `log`, `sqrt`, and `power` functions no longer produce an error for `dlarray` input if the result is complex.

Automatic Differentiation: Use more functions with dlarray input

Use the following functions with `dlarray` input. You can use these functions to define model functions and custom layers.

- Fast Fourier transforms — Compute fast Fourier transforms using `fft` and `ifft`.
- Attribute validation — Check the validity of arrays using `validateattributes`.
- Complex numbers — Use the functions `angle`, `complex`, `conj`, `imag`, `real`, `reallog`, `realsqrt`, and `realpow`.
- Ordinary differential equations – Compute the solution of a nonstiff ordinary differential equation (ODE) using `ode45`. For neural ODE workflows, use `dlode45`.

For a full list of functions that support `dlarray` input, see List of Functions with dlarray Support.

Network Training: Create layer graphs from series networks

Create a layer graph from a series network using the `layerGraph` function.

Network Training: Include softmax layers in regression networks

Including `softmaxLayer` objects in regression networks is now supported.

Network Training: Train classification networks without a softmax layer

Including `softmaxLayer` objects before a classification layer is no longer required.

Deep Learning Examples: Explore deep learning workflows

New and updated examples and topics help you progress with deep learning:

- Build Networks with Deep Network Designer
- Train Networks Using Deep Network Designer
- Train Network for Time Series Forecasting Using Deep Network Designer
- Choose Training Configurations for LSTM Using Bayesian Optimization
- Use Bayesian Optimization in Custom Training Experiments
- Explore Network Predictions Using Deep Learning Visualization Techniques
- Deep Learning Visualization Methods

New examples for image processing and computer vision tasks include:

- Recover Images from Extreme Low-Light Conditions Using Deep Learning
- Detect Image Anomalies Using Explainable One-Class Classification Neural Network
- Unsupervised Medical Image Denoising Using CycleGAN
- Unsupervised Medical Image Denoising Using UNIT
- Classify Tumors in Multiresolution Blocked Images
- Gesture Recognition using Videos and Deep Learning

New and updated examples for lidar processing tasks include:

- Code Generation For Aerial Lidar Semantic Segmentation Using PointNet++ Deep Learning
- Aerial Lidar Semantic Segmentation Using PointNet++ Deep Learning
- Lidar 3-D Object Detection Using PointPillars Deep Learning

New examples for audio processing tasks include:

- End-to-End Deep Speech Separation
- Acoustics-Based Machine Fault Recognition
- Acoustics-Based Machine Fault Recognition Code Generation with Intel MKL-DNN
- Acoustics-Based Machine Fault Recognition Code Generation on Raspberry Pi

- Accelerate Audio Deep Learning Using GPU-Based Feature Extraction

New examples for signal processing tasks include:

- Denoise EEG Signals Using Deep Learning Regression
- Hand Gesture Classification Using Radar Signals and Deep Learning
- Learn Pre-Emphasis Filter Using Deep Learning
- Human Activity Recognition Using Mobile Phone Data (Signal Processing Toolbox)

New examples for text analytics tasks include:

- Language Translation Using Deep Learning

New examples for deep learning quantization tasks include:

- Quantize Object Detectors and Generate CUDA® Code
- Parameter Pruning and Quantization of Image Classification Network

New examples for computational finance tasks include:

- Compare Deep Learning Networks for Credit Default Prediction
- Interpret and Stress-Test Deep Learning Networks for Probability of Default
- Hedging an Option Using Reinforcement Learning Toolbox

New examples of wavelet-based techniques include:

- Parasite Classification Using Wavelet Scattering and Deep Learning
- Fault Detection Using Wavelet Scattering and Recurrent Deep Networks
- Anomaly Detection Using Autoencoder and Wavelets

New and updated examples for using deep learning with Simulink include:

- Classify Images in Simulink Using GoogLeNet
- Time Series Prediction in Simulink Using Deep Learning Network
- Lane and Vehicle Detection in Simulink Using Deep Learning
- Classify Sequence of Images in Simulink with Imported TensorFlow Network
- Generate Generic C/C++ for Sequence-to-Sequence Deep Learning Simulink Models (Simulink Coder)

DenseNet-201: Improved CPU performance for inference

The `predict` and `classify` functions show improved performance for the DenseNet-201 pretrained network when the `ExecutionEnvironment` option is `"cpu"`. For example, making predictions using the following test is about 1.7x faster than in the previous release.

```
function timeDenseNet201
    miniBatchSize = 32;

    net = densenet201;
    X = randn(224,224,3,miniBatchSize,"single");
```

```
    % Warm up iterations
    for k = 1:5
        predict(net,X,ExecutionEnvironment="cpu");
    end

    % Timed iterations
    tic
    for k = 1:25
        predict(net,X,ExecutionEnvironment="cpu");
    end
    toc
end
```

The approximate execution times are:

R2021a: 60.0 seconds

R2021b: 35.1 seconds

The code was timed on a Windows 10, Intel Xeon E5-1650 v4 @ 3.60 GHz test system by calling the function `timeDenseNet201`.

Custom Training Loops: Improved performance for dlnetwork training and inference

Using the `predict` function for `dlnetwork` objects shows improved performance for models with blocks of convolution and ReLU layers, and models with blocks of convolution, batch normalization, and ReLU layers.

Using the `dlfeval` function to evaluate model gradients using `dlnetwork` objects shows improved performance for models with blocks of convolution and ReLU layers.

For example, making predictions using the following test is about 3.7x faster than in the previous release.

```
function timePrediction()
    % Create resnet50 dlnetwork
    rng(0);
    net = resnet50();
    lgraph = layerGraph(net);
    lgraph = removeLayers(lgraph,"ClassificationLayer_fc1000");
    dlnet = dlnetwork(lgraph);

    % Prepare input data
    imageSize = [224,224,3];
    miniBatchSize = 16;
    X = single(unifrnd(0,255,[imageSize,miniBatchSize]));
    X = dlarray(X,"SSCB");
    X = gpuArray(X);

    % Warm up iterations
    for i = 1:20
        Y = predict(dlnet,X);
    end

    % Timed iteration
```

```
    gputimeit(@() predict(dlnet,X))
end
```

The approximate execution times are:

R2021a: 0.07 seconds

R2021b: 0.02 seconds

The code was timed on a Windows 10, Intel Xeon E5-1650 v4 @ 3.60 GHz test system with NVIDIA Titan RTX GPU by calling the function `timePrediction`.

Functionality being removed or changed

trainNetwork automatically stops training when loss is NaN
Behavior change

When you train a network using the `trainNetwork` function, training automatically stops when the loss is `NaN`. Usually, a loss value of `NaN` introduces `NaN` values to the network learnable parameters, which in turn can cause the network to fail to train or to make valid predictions. This change helps you identify issues with the network before training completes.

In previous releases, the network continues to train when the loss is `NaN`.

nntraintool will be removed
Warns

`nntraintool` will be removed in a future release. To train a network and open the training progress plot, use `train` instead.

ClassNames option in importONNXNetwork has been removed
Errors

`ClassNames` has been removed from the `importONNXNetwork` function. Use `Classes` instead. To update your code, replace all instances of `ClassNames` with `Classes`.

ImportWeights option of importONNXLayers has been removed
Warns

`ImportWeights` has been removed from the `importONNXLayers` function. Starting in R2021b, the ONNX model weights are automatically imported. In most cases, you do not need to make any changes to your code.

- If `ImportWeights` is not set in your code, `importONNXLayers` now imports the weights.
- If `ImportWeights` is set to `true` in your code, the behavior of `importONNXLayers` remains the same.
- If `ImportWeights` is set to `false` in your code, `importONNXLayers` now ignores the name-value argument `ImportWeights` and imports the weights.

importONNXNetwork cannot create input and output layers from ONNX file information
Behavior change

If you import an ONNX model as a `DAGNetwork` object, the imported network must include input and output layers. `importONNXNetwork` tries to convert the input and output ONNX tensors into built-in

MATLAB layers. When importing some networks, which `importONNXNetwork` could previously import with input and output built-in MATLAB layers, `importONNXNetwork` might now return an error. In this case, do one of the following to update your code:

- Specify the name-value argument `TargetNetwork` as `"dlnetwork"` to import the network as a `dlnetwork` object.
- Use the name-value arguments `InputDataFormats`, `OutputDataFormats`, and `OutputLayerType` to specify the imported network's inputs and outputs.
- Use `importONNXLayers` to import the network as a layer graph with placeholder layers.
- Use `importONNXFunction` to import the network as a model function and an `ONNXParameters` object.

importONNXLayers cannot create input and output layers from ONNX file information
Behavior change

If you import an ONNX model as a `LayerGraph` object compatible with a `DAGNetwork` object, the imported layer graph must include input and output layers. `importONNXLayers` tries to convert the input and output ONNX tensors into built-in MATLAB layers. When importing some networks, which `importONNXLayers` could previously import with input and output built-in MATLAB layers, `importONNXLayers` might now insert placeholder layers. In this case, do one of the following to update your code:

- Specify the name-value argument `TargetNetwork` as `"dlnetwork"` to import the network as a `LayerGraph` object compatible with a `dlnetwork` object.
- Use the name-value arguments `InputDataFormats`, `OutputDataFormats`, and `OutputLayerType` to specify the imported network's inputs and outputs.
- Use `importONNXFunction` to import the network as a model function and an `ONNXParameters` object.

Layer names of network imported by importONNXNetwork might differ
Behavior change

The layer names of a network imported by `importONNXNetwork` might differ from previous releases. To update your code, replace the existing name of a layer with the new name or `net.Layers(n).Name`.

Names of layer imported by importONNXLayers might differ
Behavior change

The layer names of a layer graph imported by `importONNXLayers` might differ from previous releases. To update your code, replace the existing name of a layer with the new name or `lgraph.Layers(n).Name`.

R2021a

Version: 14.2

New Features

Bug Fixes

Compatibility Considerations

Experiment Manager: Train networks using custom training experiments

With **Experiment Manager**, you can now create built-in and custom training experiments. Built-in training experiments use the `trainNetwork` function. Custom training experiments support workflows that require another training function. These workflows include:

- Using a custom training loop on a `dlnetwork`, such as a Siamese network or a generative adversarial network (GAN).
- Training a network using a model function or a custom learning rate schedule.
- Updating the learnable parameters of a network by using a custom function.

When you create an experiment, you can select **Built-In Training (`trainNetwork`)** or **Custom Training** to add a blank experiment to your project. For more information, see Configure Custom Training Experiment. For examples that use custom training experiments, see Run a Custom Training Experiment for Image Comparison and Use Experiment Manager to Train Generative Adversarial Networks (GANs).

Experiment Manager: Quickly set up your experiment by loading preconfigured templates

Set up your experiment quickly by selecting a preconfigured template in **Experiment Manager**. When you create an experiment, you can use one of several templates as a guide for defining your experiment. In R2021a, the experiment templates support workflows that include image classification, image regression, sequence classification, semantic segmentation, and custom training loops. You can also try a getting started tutorial or access further examples by opening a new project.
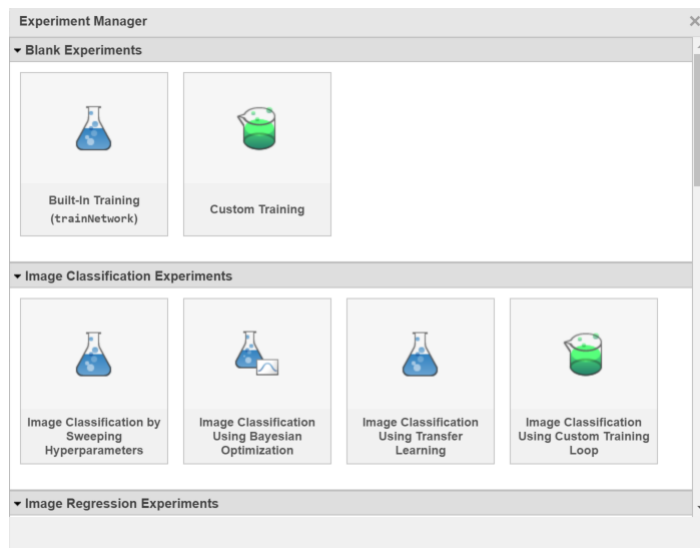
Experiment Manager: Annotate your experiment results

With **Experiment Manager**, you can now add annotations to record observations about the results of your experiment. To add an annotation, right-click a cell in the results table and select **Add Annotation**. Alternatively, select a cell in the results table and, on the **Experiment Manager**

toolstrip, select **Annotations > Add Annotation**. To open the **Annotations** pane and view all of your annotations, on the **Experiment Manager** toolstrip, select **Annotations > View Annotations**. For more information, see Sort, Filter, and Annotate Experiment Results.

Simulink Blocks: Simulate and generate code for deep learning recurrent neural networks

In R2021a, you can simulate and generate code for trained recurrent neural networks in Simulink.

The `Deep Neural Networks` block library includes the:

- Stateful Predict block — To predict responses for the data at the input by using the trained recurrent neural network specified through the block parameter.
- Stateful Classify block — To predict class labels for the data at the input by using the trained recurrent neural network specified through the block parameter.

The Stateful Predict and Stateful Classify blocks allow you to load a pretrained network into the Simulink model from a MAT-file or from a MATLAB function. The blocks update the state of the network with every prediction.

Adversarial Examples: Investigate network robustness using adversarial examples

Neural networks can be susceptible to a phenomenon known as *adversarial examples*, where very small changes to an input can cause the input to be misclassified. These changes are often imperceptible to humans.

You can investigate network robustness to adversarial examples by generating untargeted and targeted adversarial examples. For an example showing how to use the fast gradient sign method (FGSM) and the basic iterative method (BIM) to generate adversarial examples for an image classification network, see Generate Untargeted and Targeted Adversarial Examples for Image Classification.

You can train a network to be robust to adversarial examples using adversarial training. For an example showing how to use FGSM adversarial training to make an image classification network robust to adversarial images, see Train Image Classification Network Robust to Adversarial Examples. For an example showing how to use Jacobian regularization to make an image classification network robust to adversarial images, see Train Robust Deep Learning Network with Jacobian Regularization.

Visualization: Explain network predictions using Grad-CAM

Use the gradient-weighted class activation mapping (Grad-CAM) technique to help explain network prediction results. You can use the `gradCAM` function to map which parts of your input image data strongly affect network predictions.

Grad-CAM utilizes the gradient of a differentiable output with respect to convolutional features, to identify the parts of an input image that most impact the network predictions. Use `gradCAM` to understand network behavior for 2-D and 3-D image data tasks, such as classification, regression, and semantic segmentation.

For examples showing how to use Grad-CAM to explain network predictions, see Grad-CAM Reveals the Why Behind Deep Learning Decisions and Explore Semantic Segmentation Network Using Grad-CAM.

Visualization: Use custom segmentation map with LIME

The `'Segmentation'` name-value argument of `imageLIME` now accepts a two-dimensional segmentation matrix the same size as the input image. Custom segmentation maps are useful for using LIME on tasks involving non-natural images, such as spectrogram or floor plan data.

For an example showing how to explain spectrogram classifications using `imageLIME` with a custom segmentation map, see Investigate Spectrogram Classifications Using LIME.

Weighted Classification: Train network with imbalanced data using weighted classification

When training a neural network with imbalanced data, you can specify the class weights for weighted cross entropy loss using the `'ClassWeights'` option of the `classificationLayer` function.

Batch Normalization: Train network using moving batch normalization statistics

To make predictions with the network after training, batch normalization requires a fixed mean and variance to normalize the data. This fixed mean and variance can be calculated from the training data after training, or approximated during training using running statistic computations.

If the `'BatchNormalizationStatistics'` training option is `'moving'`, then the software approximates the batch normalization statistics during training using a running estimate and, after training, sets the `TrainedMean` and `TrainedVariance` properties to the latest values of the moving estimates of the mean and variance, respectively.

If the `'BatchNormalizationStatistics'` training option is `'population'`, then after network training finishes, the software passes through the data once more and sets the `TrainedMean` and `TrainedVariance` properties to mean and variance computed from the entire training data set, respectively.

The layer uses the `TrainedMean` and `TrainedVariance` to normalize the input during prediction.

Layer Normalization: Train network using layer normalization

Train a network using layer normalization layers. When using layer graph and `dlnetwork` objects, use the `layerNormalizationLayer` function. When defining a model as a function and using a custom training loop, use the `layernorm` function.

For a list of available layers, see List of Deep Learning Layers.

Instance Normalization: Train network using instance normalization

Train a network using instance normalization layers. When using layer graph and `dlnetwork` objects, use the `instanceNormalizationLayer` function. When defining a model as a function, use the `instancenorm` function.

For a list of available layers, see List of Deep Learning Layers.

Swish Layer: Train network with swish activation layers

You can now use swish layers in deep learning networks. Create a swish layer using the `swishLayer` function.

For an example showing how to compare activation layers when training a deep neural network, see Compare Activation Layers.

For a list of available layers, see List of Deep Learning Layers.

Convolution: Specify custom padding values for convolution operations

Specify custom padding options when using the `convolution2dLayer`, `convolution3dLayer`, `groupedConvolution2dLayer`, and `dlconv` functions using the `'PaddingValue'` option.

Specify the padding value as one of the following:

- Scalar – Pad with the specified scalar value.
- `'symmetric-include-edge'` – Pad using mirrored values of the input data, including the edge values.
- `'symmetric-exclude-edge'` – Pad using mirrored values of the input data, excluding the edge values.
- `'replicate'` – Pad using repeated border elements of the input data.

GPU Data: Use gpuArray data for training and inference with DAGNetworks and SeriesNetworks

You can now provide input data as `gpuArray` objects for training and inference using both convolutional and recurrent `DAGNetwork` and `SeriesNetwork` objects. Previously, training and inference required input data to be stored on the CPU, even when executing your network on the GPU.

This can be useful when your data is already stored on the GPU. You do not need to convert your data to `gpuArray` objects to perform training or inference on the GPU.

The following functions for training and inference now support `gpuArray` inputs:

- `activations`
- `classify`
- `classifyAndUpdateState`
- `predict`
- `predictAndUpdateState`
- `trainNetwork`

To use `gpuArray` inputs for training and inference, you must set the `"ExecutionEnvironment"` name-value option to `"auto"` or `"gpu"`. Using `gpuArray` inputs when `"ExecutionEnvironment"` is set to `"cpu"` results in an error.

Deep Learning Networks: Check layer graphs and networks are equal

To check that `LayerGraph`, `SeriesNetwork`, `DAGNetwork`, and `dlnetwork` objects have identical class, architecture, and properties, use the `isequal` function. To check for identical class, architecture, and properties ignoring NaN values, use the `isequaln` function.

Deep Learning Quantization: ARM Cortex-A calibration support

The Deep Learning Toolbox Model Quantization Library support package now supports calibration of a network for quantization and deployment on ARM® Cortex®-A microcontrollers. Use `dlquantizer` to specify a CPU execution environment.

```
quantObj = dlquantizer(net,'ExecutionEnvironment','CPU');
```

Use `calibrate` to exercise the network and collect the dynamic ranges of the learnable parameters in the network.

Custom Layers: Use MEX acceleration with custom layers

Accelerate prediction, classification, and feature extraction of `DAGNetwork` and `SeriesNetwork` containing custom layers using automatically generated MEX functions. You can apply MEX acceleration using the name-value option `"Acceleration","mex"` in the following functions:

- `activations`
- `classify`
- `predict`

For restrictions and usage notes on custom layers with MEX acceleration, see Custom Layers (GPU Coder).

Custom Layers: Define custom layers with formatted inputs and outputs

When defining a custom layer, defining a backward function is optional when the forward functions `dlarray` objects as input. Using `dlarray` objects makes working with high dimensional data easier by allowing you to label the dimensions. For example, you can label which dimensions correspond to spatial, time, channel, and batch dimensions using the `'S'`, `'T'`, `'C'`, and `'B'` labels, respectively. For unspecified and other dimensions, use the `'U'` label. For `dlarray` object functions that operate over particular dimensions, you can specify the dimension labels by formatting the `dlarray` object directly, or by using the `'DataFormat'` option.

Using formatted `dlarray` objects in custom layers also allows you to define layers where the inputs and outputs have different formats. For example, you can define a layer that takes as input a mini-batch of images with format `'SSCB'` (spatial, spatial, channel, batch) and output a mini-batch of sequences with format `'CBT'` (channel, batch, time).

To enable support for using formatted `dlarray` objects in custom layer forward functions, also inherit from the `nnet.layer.Formattable` class when defining the custom layer.

To learn how to create a layer that uses formatted `dlarray` objects, see Define Custom Deep Learning Layer with Formatted Inputs. For examples of custom layers that use formatted input data, open the example Train Conditional Generative Adversarial Network (CGAN) as a live script and inspect the custom layers `projectAndReshapeLayer` and `embedAndReshapeLayer`.

Network Composition: Simplify creation of custom layers containing nested layers

Previously, network composition using `dlnetwork` objects inside custom layers required each nested network to contain an input layer. The input layer provides the information needed to initialize the learnable and state parameters with initial values, ready for training. This requires keeping track of the size and shape of the inputs passed to each custom layer containing a nested `dlnetwork` with an input layer.

Now, network composition is simplified because you can defer initialization of `dlnetwork` objects to a later time. This means that you can create uninitialized `dlnetwork` objects that do not contain input layers.

When you create an uninitialized `dlnetwork`, the software does not initialize any learnable and state parameters with initial values at the time of construction.

Creating an uninitialized `dlnetwork` is useful for network composition so that you do not have to keep track of the size and shape of data as it propagates through the network. You can construct your network using custom layers with nested networks and automatically initialize all nested networks at once.

When the parent network is initialized, the learnable parameters of any nested `dlnetwork` objects are initialized at the same time. If the parent network is trained using the `trainNetwork` function, then any nested `dlnetwork` objects are initialized when you call `trainNetwork`. If the parent network is a `dlnetwork`, then any nested `dlnetwork` objects are initialized when the parent network is constructed or, if the parent network is uninitialized on construction, when you use the `initialize` function with the parent network.

To create an uninitialized `dlnetwork`, set the `'Initialize'` option to `false` when you construct the `dlnetwork`. You can check if a `dlnetwork` is initialized using the `Initialized` property of the `dlnetwork` object.

For more information on network composition, see Deep Learning Network Composition.

For a template showing how to create a custom layer containing an uninitialized nested `dlnetwork`, see Define Nested Deep Learning Layer.

Sequence Padding: Apply custom padding to sequence data

Apply custom padding to sequence data for training using the `padsequences` function. The `padsequences` function prepares your numeric or categorical sequence data for training by padding or truncating sequences to the same length.

You can pad or truncate data to the same length as the longest or shortest sequence, or to a fixed length. You can apply padding to the left, right, or both sides of the data. You can pad with zeros, a specified value, or apply symmetric padding using the reflected values of each sequence.

The `padsequences` function also provides a mask of the location of the original data, so that you can exclude the padded values in loss computations using the `"Mask"` name-value option in the `crossentropy` function.

To automatically pad your sequence data for use in custom training loops, you can use `padsequences` in conjunction with a `minibatchqueue` object.

Federated Learning: Train network using decentralized data

Use federated learning to train a network without moving data to a central location, even if individual data sources do not match the overall distribution of the data. This is known as non-independent and identically distributed (non-IID) data. Federated learning can be especially useful when the training data is large, or when there are privacy concerns about transferring the training data.

Instead of distributing data, the federated learning technique trains multiple models in parallel, each in the same location as a data source. The global model learns from all the data sources via periodic averaging of the learnable parameters of the locally trained models.

For an example showing how to train a network using federated learning, see Train Network Using Federated Learning.

Custom Training Loops: Apply 1-D convolution and pooling operations to sequence and time-series data

Apply 1-D convolution and pooling operations to sequence and time-series data

The `dlconv` function now supports convolving over the `'T'` (time) dimension of the input data.

The function, by default, convolves over up to three dimensions of the input data labeled `'S'` (spatial). To convolve over dimensions labeled `'T'` (time), specify the `weights` argument with a `'T'` dimension using a formatted `dlarray` object or by using the `'WeightsFormat'` option.

The `maxpool` and `avgpool` functions now support pooling over the `'T'` dimension of the input data. Specify which dimensions to pool over by using the `'PoolFormat'` option.

The `maxunpool` function also supports unpooling over the `'T'` dimension.

For an example showing how to train a neural network using 1-D convolutions, see Sequence-to-Sequence Classification Using 1-D Convolutions.

Custom Training Loops: Use Huber and CTC loss in custom training loops

Use the following loss functions in custom training loops:

- Huber – Calculate the Huber loss using the `huber` function.

- Connectionist temporal classification (CTC) – Calculate the connectionist temporal classification loss using the `ctc` function.

Custom Training Loops: Specify weights, masks, and reduction options for custom training loop loss function

Specify weights, mask, and reduction options for the `crossentropy` and `huber` functions using the `weights`, `Mask`, and `Reduction` arguments, respectively.

Custom Training Loops: Train using higher-order derivatives

When training using a custom training loop, enable the calculation of higher-order derivatives using the `dlgradient` function using the `'EnableHigherDerivatives'` option. This allows you to develop models such as Wasserstein generative adversarial networks (WGANs) which require calculating second order derivatives.

When the `'EnableHigherDerivatives'` option of `dlgradient` is `true`, the function traces the backward pass so that the returned gradients can be used in subsequent calls to `dlgradient` function.

For examples showing how to train models that require calculating higher-order derivatives, see:

- Train Wasserstein GAN with Gradient Penalty (WGAN-GP)
- Solve Partial Differential Equations Using Deep Learning
- Solve Partial Differential Equation with LBFGS Method and Deep Learning (requires Optimization Toolbox™)

Custom Training Loops: Accelerate custom deep learning functions

When using the `dlfeval` function in a custom training loop, the software traces each input `dlarray` object of the model gradients function to determine the computation graph used for automatic differentiation. This tracing process can take some time and can spend time recomputing the same trace. By optimizing, caching, and reusing the traces, you can speed up gradient computation in deep learning functions. You can also optimize, cache, and reuse traces to accelerate other deep learning functions that do not require automatic differentiation, for example you can also accelerate model functions and functions used for prediction.

To accelerate a deep learning function, use the `dlaccelerate` function. The returned `AcceleratedFunction` object optimizes and caches the traces of calls to the underlying function and reuses the cached result when the same input pattern reoccurs.

Try using `dlaccelerate` for function calls that:

- are long-running
- have `dlarray` object, structures of `dlarray` objects, or `dlnetwork` objects as inputs
- do not have side effects like writing to files or displaying output

To clear the cache of an `AcceleratedFunction` object, use the `clearCache` function.

For an example showing how to accelerate training and prediction functions, see Accelerate Custom Training Loop Functions. For an example showing how to compare the performance of functions with

and without acceleration, see Evaluate Performance of Accelerated Deep Learning Function. For an example showing how to compare the outputs of accelerated functions and the underlying function, see Check Accelerated Deep Learning Function Outputs.

To learn more, see Deep Learning Function Acceleration for Custom Training Loops.

Custom Training Loops: Automatic performance optimization for training and inference

Accelerate training and inference when using `dlnetwork` objects with automatic performance optimization. The software automatically applies a number of performance optimizations suitable for the input network and hardware resources.

Use performance optimization when you plan to call the function multiple times using different input data with the same size and shape.

The following functions support automatic performance optimization:

- `forward`
- `predict`

Automatic performance optimization is enabled by default. You can disable performance optimization using the name-value option `'Acceleration','none'`. To programmatically enable optimization, use the name-value option `'Acceleration','auto'`.

Custom Training Loops: Create dlnetwork objects without input layers

You can now create `dlnetwork` objects that do not contain input layers. Previously, input layers were required. The software uses the information from the input layer to initialize the learnable and state parameters with initial values, ready for training. Now, you can specify that information using example inputs when you create the `dlnetwork`.

You can create an initialized `dlnetwork` that is ready for training by providing example inputs when you construct the `dlnetwork`.

Alternatively, you can create an uninitialized dlnetwork without an input layer, and initialize it later. An uninitialized network has unset, empty values for learnable and state parameters and is not ready for training. You must initialize an uninitialized `dlnetwork` before you can use it. Create an uninitialized network when you want to defer initialization to a later point. You can use uninitialized `dlnetwork` objects to create complex networks using intermediate building blocks that you then connect together, for example, using Deep Learning Network Composition workflows. You can initialize an uninitialized `dlnetwork` using the `initialize` function.

Custom Training Loops: Create dlnetwork objects using Layer arrays

You can now create `dlnetwork` objects directly from a `Layer` array, Previously, you had to convert the `Layer` array to an intermediate `LayerGraph` before you could create a `dlnetwork`.

When you create a `dlnetwork` from a `Layer` array, the software connects the layers in series.

TensorFlow Model Import: Import a TensorFlow network in the saved model format

You can now import a pretrained TensorFlow network in the saved model format by using the `importTensorFlowNetwork` or `importTensorFlowLayers` function. You can import networks created with the TensorFlow-Keras sequential or functional API. `importTensorFlowNetwork` and `importTensorFlowLayers` try to generate a custom layer when you import a custom TensorFlow layer or when the software cannot convert a TensorFlow layer into an equivalent built-in MATLAB layer.

TensorFlow-Keras Layer Support: Import CuDNNGRU layers, swish activation layers, and feature input layers from TensorFlow-Keras

`importKerasNetwork` and `importKerasLayers` now support CuDNNGRU layers, swish activation layers, and feature input layers. For a full list of supported layers, see `importKerasNetwork` and `importKerasLayers`.

ONNX Layer Support: Import and export networks that include depth to space layers, swish activation layers, and feature input layers

You can now import an ONNX (Open Neural Network Exchange) network that includes depth to space layers by using `importONNXNetwork` and `importONNXLayers`. Also, you can export a trained MATLAB deep learning network that includes depth to space layers, swish activation layers, and feature input layers to the ONNX model format by using `exportONNXNetwork`. For a full list of supported layers, see `importONNXNetwork`, `importONNXLayers`, and `exportONNXNetwork`.

ONNX Export Support: Export layerGraph and dlnetwork objects

You can now export `dlnetwork` and `layerGraph` objects to the ONNX model format by using the `exportONNXNetwork` function.

Pretrained Models on GitHub: BERT and FinBERT transformer models

To learn how to load pretrained BERT and FinBERT transformer models into MATLAB, see the Transformer Models for MATLAB repository. You can use BERT and FinBERT for text classification, sentiment analysis, and other text analytics workflows.

To find the latest pretrained models and examples for deep learning, see MATLAB Deep Learning (GitHub).

Deep Learning Examples: Explore deep learning workflows

New examples help you progress with deep learning:

- Run a Custom Training Experiment for Image Comparison
- Use Experiment Manager to Train Generative Adversarial Networks (GANs)
- Choose Training Configurations for Sequence-to-Sequence Regression
- Adapt Code Generated in Deep Network Designer for Use in Experiment Manager

- Investigate Spectrogram Classifications Using LIME
- Interpret Deep Network Predictions on Tabular Data Using LIME
- Explore Semantic Segmentation Network Using Grad-CAM
- Generate Untargeted and Targeted Adversarial Examples for Image Classification
- Train Image Classification Network Robust to Adversarial Examples
- Train Wasserstein GAN with Gradient Penalty (WGAN-GP)
- Solve Partial Differential Equations Using Deep Learning
- Solve Partial Differential Equation with LBFGS Method and Deep Learning
- Train Neural ODE Network
- Node Classification Using Graph Convolutional Network
- Classify Videos Using Deep Learning with Custom Training Loop
- Train Robust Deep Learning Network with Jacobian Regularization
- Compare Activation Layers
- Define Custom Deep Learning Layer with Formatted Inputs
- Accelerate Custom Training Loop Functions
- Evaluate Performance of Accelerated Deep Learning Function
- Check Accelerated Deep Learning Function Outputs
- Train Network Using Federated Learning

New examples to help you get started with deep learning in Simulink:

- Predict and Update Network State in Simulink
- Classify and Update Network State in Simulink

New examples for computer vision tasks include:

- Instance Segmentation Using Mask R-CNN Deep Learning

New examples for image processing tasks include:

- Unsupervised Day-To-Dusk Image Translation Using UNIT
- Quantify Image Quality Using Neural Image Assessment

New examples for lidar processing tasks include:

- Aerial Lidar Semantic Segmentation Using PointNet++ Deep Learning (Lidar Toolbox)
- Data Augmentations for Lidar Object Detection Using Deep Learning (Lidar Toolbox)
- Code Generation For Lidar Object Detection Using PointPillars Deep Learning (Lidar Toolbox)
- Automate Ground Truth Labeling For Vehicle Detection Using PointPillars (Lidar Toolbox)
- The PointPillars and SqueezeSegV2 networks pre-trained on the PandaSet multiclass dataset are now available. For more information, see Lidar Point Cloud Semantic Segmentation Using SqueezeSegV2 Deep Learning Network and Lidar 3-D Object Detection Using PointPillars Deep Learning.

New examples for audio processing tasks include:

- Speaker Recognition Using x-vectors (Audio Toolbox)
- Speaker Diarization Using x-vectors (Audio Toolbox)
- Train Spoken Digit Recognition Network Using Out-of-Memory Features (Audio Toolbox)
- Train Spoken Digit Recognition Network Using Out-of-Memory Audio Data (Audio Toolbox)
- Speaker Identification Using Custom SincNet Layer and Deep Learning (Audio Toolbox)
- Dereverberate Speech Using Deep Learning Networks (Audio Toolbox)
- Speech Command Recognition in Simulink (Audio Toolbox)
- **Keyword Spotting in Noise Code Generation with Intel MKL-DNN** (Audio Toolbox)
- **Keyword Spotting in Noise Code Generation on Raspberry Pi** (Audio Toolbox)

New examples and topics for reinforcement learning tasks include:

- Reinforcement Learning Using Deep Neural Networks
- Train PPO Agent for Automatic Parking Valet
- **Train Humanoid Walker**

New examples for deep learning quantization tasks include:

- Quantize Residual Network Trained for Image Classification and Generate CUDA Code

predict Function: Improved performance for dlnetwork inference

The `predict` function for `dlnetwork` input shows improved performance for models that do not have custom layers. For example, making predictions using the following test is about 2.7x faster than in the previous release:

```matlab
function timePrediction
    load("digitsCustom.mat","dlnet");
    X = dlarray(randn(28,28,1,64),'SSCB');

    % Warm up iterations
    for k = 1:5
        predict(dlnet,X);
    end

    % Timed iterations
    tic
    for k = 1:100
        predict(dlnet,X);
    end
    toc

end
```

The approximate execution times are:

R2020b: 1.97 seconds

R2021a: 0.72 seconds

The code was timed on a Windows 10, Intel Xeon E5-1650 v4 @ 3.60 GHz test system by calling the function `timePredictions`.

EfficientNet-B0 Pretrained Network: Improved CPU performance for inference

The `predict` and `classify` functions show improved performance for the EfficientNet-B0 pretrained network when the `'ExecutionEnvironment'` option is `'cpu'`. For example, making predictions using the following test is about 1.7x faster than in the previous release.

```
function timeEfficientNetB0
    miniBatchSize = 32;

    net = efficientnetb0;
    X = randn(224,224,3,miniBatchSize,'single');

    % Warm up iterations
    for k = 1:5
        predict(net,X,'ExecutionEnvironment','cpu');
    end

    % Timed iterations
    tic
    for k = 1:100
        predict(net,X,'ExecutionEnvironment','cpu');
    end
    toc
end
```

The approximate execution times are:

R2020b: 73.4 seconds

R2021a: 42.0 seconds

The code was timed on a Windows 10, Intel Xeon E5-1650 v4 @ 3.60 GHz test system by calling the function `timeEfficientNetB0`.

Functionality being removed or changed

dlnetwork state values are dlarray objects
Behavior change

The `State` of a `dlnetwork` object is a table containing the state parameter names and values for each layer in the network.

Starting in R2021a, the state values are `dlarray` objects. This change enables better support when using `AcceleratedFunction` objects. To accelerate deep learning functions that have frequently changing input values, for example, an input containing the network state, the frequently changing values must be specified as `dlarray` objects.

In previous versions, the state values are numeric arrays.

In most cases, you will not need to update your code. If you have code that requires the state values to be numeric arrays, then to reproduce the previous behavior, extract the data from the state values manually using the `extractdata` function with the `dlupdate` function.

```
state = dlupdate(@extractdata,dlnet.State);
```

forward and predict returns state values as dlarray objects
Behavior change

For `dlnetwork` objects, the `state` output argument returned by the `forward` and `predict` functions is a table containing the state parameter names and values for each layer in the network.

Starting in R2021a, the state values are `dlarray` objects. This change enables better support when using `AcceleratedFunction` objects. To accelerate deep learning functions that have frequently changing input values, for example, an input containing the network state, the frequently changing values must be specified as `dlarray` objects.

In previous versions, the state values are numeric arrays.

In most cases, you will not need to update your code. If you have code that requires the state values to be numeric arrays, then to reproduce the previous behavior, extract the data from the state values manually using the `extractdata` function with the `dlupdate` function.

```
state = dlupdate(@extractdata,dlnet.State);
```

trainNetwork support for tables of MAT file paths will be removed in a future release
Warns

When specifying sequence data for the `trainNetwork` function, support for specifying tables of MAT file paths will be removed in a future release.

To train networks with sequences that do not fit in memory, use a datastore. You can use any datastore to read your data and then use the `transform` function to transform the datastore output to the format the `trainNetwork` function requires. For example, you can read data using a `FileDatastore` or `TabularTextDatastore` object then transform the output using the `transform` function.

R2020b

Version: 14.1

New Features

Bug Fixes

Compatibility Considerations

Image Classification and Network Prediction Blocks: Simulate and generate code for deep learning models in Simulink

In R2020b, you can simulate and generate code for trained deep learning networks in Simulink.

The `Deep Neural Networks` block library includes the:

- Predict block — To predict responses using the trained network specified through the block parameter. This block allows loading of a pretrained network into the Simulink model from a MAT-file or from a MATLAB function.

  For more information about working with the Predict block, see Lane and Vehicle Detection in Simulink Using Deep Learning. To learn more about generating code for Simulink models containing the Predict block, see Code Generation for a Deep Learning Simulink Model that Performs Lane and Vehicle Detection (GPU Coder).

- Image Classifier block — To classify data using a trained deep learning neural network specified through the block parameter. This block allows loading of a pretrained network into the Simulink model from a MAT-file or from a MATLAB function.

  For more information about working with the Image Classifier block, see Classify ECG Signals in Simulink Using Deep Learning. To learn more about generating code for Simulink models containing the Image Classifier block, see Code Generation for a Deep Learning Simulink Model to Classify ECG Signals (GPU Coder).

Experiment Manager: Train networks in parallel and using Bayesian optimization

With **Experiment Manager**, you can now run multiple trials of an experiment in parallel. You can also use Bayesian optimization to determine the best combination of hyperparameters for an experiment.

Running the trials of an experiment in parallel allows you to try different training configurations at the same time. You can also use MATLAB while the training is in progress. To run your experiment in parallel, on the **Experiment Manager** toolstrip, click **Use Parallel** and then **Run**. Experiment Manager starts the parallel pool and executes multiple simultaneous trials, depending on the number of parallel workers available. Parallel execution requires Parallel Computing Toolbox. For more information, see Use Experiment Manager to Train Networks in Parallel.

Using Bayesian optimization provides an alternative to sweeping hyperparameters. To create an experiment that uses Bayesian optimization, specify a range of values for each hyperparameter and select a metric to maximize or minimize. When you click **Run**, Experiment Manager searches for a combination of hyperparameters that optimizes the metric you selected. The combination used in each trial is generated based on the results of the previous trials. You can train for a maximum length of time or a maximum number of trials. Experiment Manager indicates the trial with the optimal value for the selected metric. Bayesian optimization requires Statistics and Machine Learning Toolbox. Additionally, if you have Parallel Computing Toolbox, you can run multiple Bayesian optimization trials at the same time. For more information, see Tune Experiment Hyperparameters by Using Bayesian Optimization.

Deep Network Designer: Train networks for semantic segmentation, multi-input, out-of-memory, and image-to-image regression workflows

Deep Network Designer now supports import and training for built-in or custom datastores. Load a datastore object by selecting **Import Data > Import Datastore** from the **Data** tab. To learn more about importing data into Deep Network Designer, see Import Data into Deep Network Designer.

Use datastores for semantic segmentation, multi-input, out-of-memory, and image-to-image regression workflows. For an example showing how to train an image-to-image regression network in Deep Network Designer, see Image-to-Image Regression in Deep Network Designer. For an example showing how to train a simple semantic segmentation network in Deep Network Designer, see Create Simple Semantic Segmentation Network in Deep Network Designer. These examples require Computer Vision Toolbox.

Deep Network Designer: Explore prebuilt recurrent networks for sequence and time series data

Explore prebuilt sequence networks using the Deep Network Designer app.

To load a prebuilt sequence network, open Deep Network Designer and select the icon for the desired untrained sequence network. Deep Network Designer contains prebuilt networks suitable for sequence-to-sequence and sequence-to-label classification.

For an example showing how to build a sequence network in Deep Network Designer, see Create Simple Sequence Classification Network Using Deep Network Designer.

Deep Network Designer: Visualize data on import

Deep Network Designer now provides a preview of the data imported, allowing you to perform simple checks before training, including:

- For image datastores, visualize the images by class.
- For pixel label image datastores, visualize the images and the pixel labels.
- For combined datastores, visualize the observations in the underlying datastores.
- For other datastores, display the size of the observations.

Example Image Data Visualization	Example Datastore Visualization
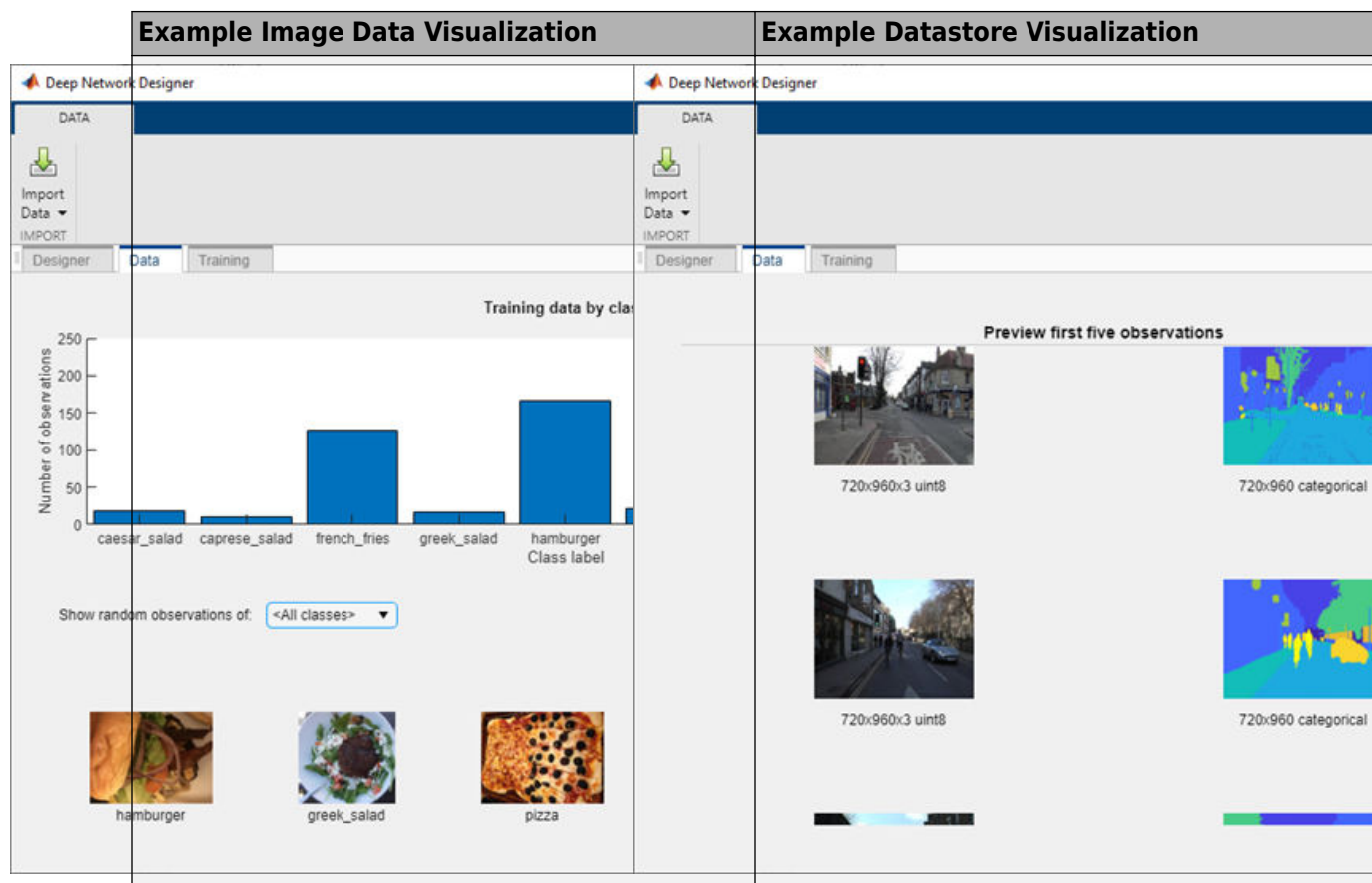
For more information about visualizing data in Deep Network Designer, see Import Data into Deep Network Designer.

Deep Network Designer: Load multiple networks and custom layers

Deep Network Designer now supports importing multiple networks, layer arrays, and layer graphs from the workspace into a session.

This allows you to add custom layers to a network and combine layers from multiple networks. For example, you can create a semantic segmentation network by combining a pretrained network with a decoder subnetwork.

For an example showing how to create a weighted classification network by importing a custom layer into Deep Network Designer, see Import Custom Layer into Deep Network Designer.

Deep Network Designer: Randomize splitting of training and validation data

Randomize the splitting of imported data into training and validation sets by selecting the **Randomize** check box in Deep Network Designer.

Use randomization to prevent overfitting when training a network on data stored in a nonrandom order. For more information, see Split Validation Data from Training Data.

For an example showing how to randomize the splitting of training and validation data in Deep Network Designer, see Create Simple Image Classification Network Using Deep Network Designer.

Deep Network Designer: Programmatically open a network in Deep Network Designer

You can now open Deep Network Designer with a network input argument. For example, to open SqueezeNet in Deep Network Designer, use the command `deepNetworkDesigner(squeezenet)`. For more information, see Deep Network Designer.

Deep Network Designer: Edit more properties of convolution layers

In Deep Network Designer, you can now edit more layer properties without losing the weights and biases information. You can now edit the `Stride`, `Padding`, and `DilationFactor` properties of convolution and grouped convolution layers, and the `Stride` and `Cropping` properties of transposed convolution layers without losing the weights and biases information.

Editing these properties can be useful for creating a semantic segmentation network using a pretrained network as a basis, and then editing the padding, stride, dilation, or cropping properties to reach the desired alignment.

Feature Input: Train networks with feature input

Train networks with data sets of numeric scalars representing features (for example, tabular data with rows corresponding to observations and columns containing numeric features) using a `featureInputLayer` object. A feature input layer inputs feature data into a network and applies data normalization.

For an example showing how to train a deep learning model that classifies thyroid conditions given data set of numeric features, see Train Network with Numeric Features.

To train a network with both image and feature data, you must create a `dlnetwork` and use a custom training loop. For an example showing how to classify digits given the images and corresponding angles of rotation, see Train Network on Image and Feature Data.

Multiplication Layer: Add multiplication layer to networks

You can now use multiplication layers in deep learning networks. Create a multiplication layer using `multiplicationLayer`.

For a list of available layers, see List of Deep Learning Layers.

Sigmoid Layer: Train networks using sigmoid activation

You can now use sigmoid layers in deep learning networks. Create a sigmoid layer using `sigmoidLayer`.

For a list of available layers, see List of Deep Learning Layers.

Group Normalization: Train networks using group normalization

A group normalization layer divides channels of data into groups and normalizes the data within each group. You can use group normalization as an alternative to batch normalization

To include a group normalization layer in a layer graph or layer array, use the `groupNormalizationLayer` function. To apply the group normalization operation in a model function used in a custom training loop, use the `groupnorm` function.

Batch Normalization: Use batch normalization layers for sequence data

Networks with sequence input now support batch normalization layers. To create a batch normalization layer, use the `batchNormalizationLayer` function.

Visualization: Explain image classification predictions using LIME

Use the local interpretable model-agnostic explanation (LIME) technique to help explain image classification results. You can use the `imageLIME` function to map which parts of your input data strongly affect classification decisions.

The LIME algorithm divides the input image into features and then generates a large number of sample images with features removed at random. These sample images are used to fit a simple model — such as a decision tree — that approximates the behavior of the network. The model is used to map the importance of each feature, highlighting the areas that are significant to the classification score.

For an example of how to use LIME to explain classification results, see Understand Network Predictions Using LIME.

Pretrained Networks: Perform transfer learning with EfficientNet-b0 pretrained convolutional neural network

You can now install an add-on for the EfficientNet-b0 pretrained convolutional neural network. To download and install the pretrained network, use the Add-On Explorer. You can also download the network from MathWorks Deep Learning Toolbox Team. After you install the add-on, use the `efficientnetb0` function to load the network.

To retrain a network on a new classification task, follow the steps in Train Deep Learning Network to Classify New Images and load EfficientNet-b0 instead of GoogLeNet.

For more information on pretrained neural networks in MATLAB, see Pretrained Deep Neural Networks.

Multi-input Networks: Make predictions using multiple numeric arrays

For networks with multiple inputs, you can pass numeric arrays directly to the `predict`, `classify`, and `activations` functions.

Multi-input Networks: Monitor network training progress using validation data

For networks with multiple inputs, monitor network training progress using validation data. Specify validation data using the `'ValidationData'` option of the `trainingOptions` function.

Multi-input Networks: Train networks with multiple inputs using a parallel pool

The `trainNetwork` function now supports training networks with multiple inputs using multiple workers. To train networks using a parallel pool, set the `'ExecutionEnvironment'` option of the `trainingOptions` function to `'parallel'`.

Gated Recurrent Units: Apply recurrent bias to GRU operation

`gruLayer` objects now support using an additional bias term in the state and gate calculations. To apply the reset gate after matrix multiplication and use a recurrent bias term, set the `ResetGateMode` property to `'recurrent-bias-after-multiplication'`.

For more information about the reset gate calculations, see Gated Recurrent Unit Layer section in the `gruLayer` reference page.

Network Composition: Define custom layers containing layer graphs

To create a custom layer that itself defines a layer graph, you can specify a `dlnetwork` object as a learnable parameter. This is known as *network composition*. You can use network composition to:

- Create a single custom layer that represents a block of learnable layers. For example, a residual block.
- Create networks with control flow. For example, where a section of the network can dynamically change depending on the input data.
- Create networks with loops. For example, where sections of the network feed its output back into itself.

For more information, see Deep Learning Network Composition.

For an example showing how to create a custom layer that represents a residual block consisting of multiple learnable layers, optional layers, and a skip connection, see Define Nested Deep Learning Layer.

For an example showing how to train a network with nested layers, see Train Deep Learning Network with Nested Layers.

To learn more about defining custom layers, see Define Custom Deep Learning Layers.

Custom Layers: Define custom layers for code generation

Define custom layers that support code generation workflows by specifying the `%#codegen` pragma. For an example showing how to define a custom layer with learnable parameters that supports code generation, see Define Custom Deep Learning Layer for Code Generation.

To check that a custom layer is valid and supports code generation, use the `checkLayer` function and set the `'CheckCodegenCompatibility'` option to `true`.

For an example showing how to generate CUDA MEX for a YOLO-v3 object detector with custom layers, see Code Generation For Object Detection Using YOLO v3 Deep Learning.

Average Pooling: Exclude padded values from average pooling

You can now exclude padded values when calculating the average pooling of padded inputs. Typically, padding adds zeros to the edges of the input data, which affects the average values of the edge regions. You can now exclude the padded values from the averages of these padded regions by padding with the mean value instead of zeros.

To exclude padded values when pooling using the average pooling layers `averagePooling2dLayer` and `averagePooling3dLayer`, set the `'PaddingValue'` property to `'mean'`.

To exclude padded values when pooling using the `avgpool` function in a model function, set the `'PaddingValue'` option to `'mean'`.

Custom Training Loops: Automatically create and preprocess mini-batches of data

Create mini-batches of data for custom training loops using a `minibatchqueue` object. Use mini-batch queues to automatically convert data to `dlarray` or `gpuArray`, or apply a custom mini-batch preprocessing function.

Work with mini-batches using the following object functions

- `next` – Return the next mini-batch
- `reset` – Reset the `minibatchqueue` object to the start of the data set
- `shuffle` – Shuffle the mini-batch queue
- `hasdata` – Check if the mini-batch queue can return a mini-batch
- `partition` – Partition the mini-batch queue for parallel computations

`minibatchqueue` objects support datastore input only. For data in numeric arrays, first convert the data to a datastore using an `ArrayDatastore` object.

One-Hot Encoding: Encode and decode categorical data into vectors

Transform categorical data into one-hot vectors suitable for training networks with categorical input using custom training loops. One-hot encoding expands categorical data into vectors with the same number of elements as the total number of categories. The vectors contain a `1` in the position corresponding to the appropriate category, and `0` otherwise. To create one-hot encoded vectors, use the `onehotencode` function. When training networks with feature input, use the `onehotencode` function to transform categorical data into numeric one-hot encoded vectors.

One-hot decoding takes one-hot encoded vectors and determines the encoded category. To decode one-hot encoded vectors, use the `onehotdecode` function. You can also use the `onehotdecode` function to determine the top class given a vector of scores, such as the output from a trained `dlnetwork`.

To automatically process your data and class labels for use in custom training loops, you can use `onehotencode` in conjunction with a `minibatchqueue` object.

Custom Training Loops: Analyze networks for custom training loop workflows

The `analyzeNetwork` function now supports analyzing layer graphs and `dlnetwork` objects for custom training loop workflows.

Use the `analyzeNetwork` function to visualize and understand the architecture or a network, check that you have defined the architecture correctly, and detect problems before training. Problems that `analyzeNetwork` detects include missing or unconnected layers, incorrectly sized layer inputs, an incorrect number of layer inputs, and invalid graph structures.

To analyze a layer graph for custom training loop workflows set the `'TargetUsage'` option to `'dlnetwork'`.

Custom Training Loops: Use bidirectional LSTM and word embedding layers in dlnetwork objects

`dlnetwork` objects now support layer graphs containing `bilstmLayer` and `wordEmbeddingLayer` objects.

Use BiLSTM layers to learn bidirectional long-term dependencies in sequence data.

Use word embedding layers to convert text data to sequences of numeric vectors. For an example showing how to use a word embedding layer in a custom training loop, see Classify Text Data Using Custom Training Loop.

For a list of layers that `dlnetwork` objects support, see the Supported Layers section of the `dlnetwork` reference page.

Embeddings: Convert categorical and discrete data to numeric vectors

Embeddings map elements in a discrete vocabulary to numeric vectors. These embeddings can capture semantic details of discrete data so that similar elements map to similar vectors.

To convert categorical and discrete data to numeric for custom training loop and custom layer workflows, use the `embed` function.

For an example showing how to embed text data to train a model for sequence-to-sequence translation, see Sequence-to-Sequence Translation Using Attention.

Automatic Differentiation: Use more functions with dlarray input

Use the following functions with `dlarray` input. You can use these functions to define model functions and custom layers.

- Group normalization – Apply cross channel normalization using the `groupnorm` function.
- Interpolation – Interpolate function values between sample points using the `interp1` function.

- Inverse trigonometric functions – Apply inverse trigonometric functions `acos`, `acosh`, `acot`, `acsc`, `asec`, `asin`, `asinh`, `atan`, `atan2` and `atanh` to data in model functions and custom layers.
- Query data values – Use the functions `isfinite`, `isinf`, and `isnan` to check the value of `dlarray` data.
- Batch matrix multiplication – Perform matrix multiplication in batches using the `pagemtimes` function.
- Average pooling – Exclude padded values when pooling using `avgpool` by setting the `'PaddingValue'` option to `'mean'`.

For a list of functions that support automatic differentiation, see List of Functions with dlarray Support. To learn more about defining and training deep learning models using automatic differentiation, see Define Custom Training Loops, Loss Functions, and Networks.

Query Data Types: Query class and underlying data type

Check if data is stored as a `dlarray` or a `gpuArray` using the following functions.

- `isdlarray`
- `isgpuarray` (Parallel Computing Toolbox)

Use these functions to query your data class and avoid executing code that expects `dlarray` or `gpuArray` inputs.

You can also use the following functions to query the underlying data type of data stored in `gpuArray` or `dlarray` objects, or distributed arrays.

- `underlyingType`
- `isUnderlyingType`
- `mustBeUnderlyingType`

The `class` function is useful to determine the class of a variable. However, some classes in MATLAB can contain underlying data that has a different type compared to what `class` returns. Example classes include `gpuArray`, `dlarray`, and `distributed` arrays. The `underlyingType`, `isUnderlyingType`, and `mustBeUnderlyingType` functions now provide a simple way to query the underlying data types of those classes.

For most classes, `class(X)` and `underlyingType(X)` return the same answer. However, for classes that can contain underlying data of a different type, `class(X)` returns the name of the class (such as `gpuArray`), and `underlyingType(X)` returns the type of the underlying data (such as `double`).

Custom Training Loops: Learn More About Custom Training Loop Workflows

To learn more about defining model gradients functions for custom training loops, see Define Model Gradients Function for Custom Training Loop.

To learn more about initializing learnable parameters for custom training loops, see Initialize Learnable Parameters for Model Functions.

To learn more about how to train deep learning networks, see Training Deep Learning Models in MATLAB.

TensorFlow-Keras Support: Import sigmoid layers, multiplication layers, 2-D upsampling layers, and 3-D upsampling layers from TensorFlow-Keras

`importKerasNetwork` and `importKerasLayers` now support sigmoid layers, multiplication layers, 2-D upsampling layers, and 3-D upsampling layers. For a full list of supported layers, see `importKerasNetwork` and `importKerasLayers`.

ONNX Support: Import and export networks that include sigmoid layers, multiplication layers, 2-D resize layers, 3-D resize layers, space to depth layers, and instance normalization layers

You can now import an ONNX (Open Neural Network Exchange) network that includes sigmoid layers, multiplication layers, 2-D resize layers, 3-D resize layers, space to depth layers, and instance normalization layers by using `importONNXNetwork` and `importONNXLayers`. Also, you can export a trained MATLAB deep learning network that includes these layers to the ONNX model format by using `exportONNXNetwork`. For a full list of supported layers, see `importONNXNetwork`, `importONNXLayers`, and `exportONNXNetwork`.

ONNX Support: Import an ONNX network by using importONNXFunction

Use `importONNXFunction` to import an ONNX (Open Neural Network Exchange) network as a function. For example, use `importONNXFunction` to import YOLOv3. `importONNXFunction` returns an `ONNXParameters` object that contains the network parameters, and creates a model function that contains the network architecture. Use `ONNXParameters` and the model function to perform common deep learning tasks, such as image and sequence data classification, transfer learning, object detection, and image segmentation. `importONNXFunction` is also useful if you want to define a custom training loop.

Deep Learning Examples: Explore deep learning workflows

New examples to help you get started with deep learning in Simulink:

- Lane and Vehicle Detection in Simulink Using Deep Learning
- Classify ECG Signals in Simulink Using Deep Learning

New examples and topics to help you get started with deep learning:

- Create Simple Image Classification Network Using Deep Network Designer
- Create and Explore Datastore for Image Classification
- Import Data into Deep Network Designer
- Image-to-Image Regression in Deep Network Designer
- Create Simple Semantic Segmentation Network in Deep Network Designer

- Import Custom Layer into Deep Network Designer
- Transfer Learning Using Pretrained Network

New examples and topics help you progress with deep learning:

- Use Experiment Manager to Train Networks in Parallel
- Tune Experiment Hyperparameters by Using Bayesian Optimization
- Train Network with Numeric Features
- Train Network on Image and Feature Data
- Understand Network Predictions Using LIME
- Define Model Gradients Function for Custom Training Loop
- Train Fast Style Transfer Network
- Define Nested Deep Learning Layer
- Train Deep Learning Network with Nested Layers
- Define Custom Deep Learning Layer for Code Generation

New examples for text analytics tasks include:

- Classify Text Data Using Custom Training Loop
- Define Text Decoder Model Function
- Define Text Encoder Model Function
- Generate Text Using Autoencoders

New examples for computer vision tasks include:

- Generate Image from Segmentation Map Using Deep Learning
- Estimate Body Pose Using Deep Learning
- Activity Recognition from Video and Optical Flow Data Using Deep Learning
- Code Generation For Object Detection Using YOLO v3 Deep Learning

New examples for image processing tasks include:

- Develop Raw Camera Processing Pipeline Using Deep Learning

New examples for automated driving tasks include:

- Train Deep Learning Semantic Segmentation Network Using 3-D Simulation Data

New examples for lidar processing tasks include:

- Lidar Point Cloud Semantic Segmentation Using PointSeg Deep Learning Network
- Lidar Point Cloud Semantic Segmentation Using SqueezeSegV2 Deep Learning Network
- Lidar 3-D Object Detection Using PointPillars Deep Learning
- Code Generation for Lidar Point Cloud Segmentation Network

New examples for audio processing tasks include:

- Speech Command Recognition Code Generation with Intel MKL-DNN

- Speech Command Recognition Code Generation on Raspberry Pi

New examples of wavelet-based techniques include:

- Crack Identification From Accelerometer Data
- Deploy Signal Classifier on NVIDIA Jetson Using Wavelet Analysis and Deep Learning
- Deploy Signal Classifier Using Wavelets and Deep Learning on Raspberry Pi

trainNetwork Function: Improved training performance using multiple GPUs on Windows

The `trainNetwork` function shows improved performance on Windows for network training when the `'ExecutionEnvironment'` option is `'multi-gpu'`. For example, training a network using two GPUs in the following test is about 1.3x faster than in the previous release.

```
% Set up network for transfer learning
net = resnet50;
inputSize = net.Layers(1).InputSize;

numObs = 2000;

data = rand([inputSize numObs]);
labels = categorical(randi(5,[numObs 1]));

lgraph = layerGraph(net);
lgraph = replaceLayer(lgraph,'ClassificationLayer_fc1000',classificationLayer('Name','newClass'));
lgraph = replaceLayer(lgraph,'fc1000',fullyConnectedLayer(5,'Name','newFC'));

% Measure timings for 1, 2, and 4 GPUs
numGPUs = [1 2 4];
timings = zeros(numel(numGPUs),1);

for ii = 1:numel(numGPUs)

    % Set execution evironment open parallel pool
    if numGPUs(ii) == 1
        execEnv = 'gpu';
    else
        parpool(numGPUs(ii));
        execEnv = 'multi-gpu';
    end

    % Set appropriate mini-batch size for number of GPUs
    miniBatchSize = 64*numGPUs(ii);
    % Set training options
    options = trainingOptions('sgdm', ...
        'MiniBatchSize',miniBatchSize, ...
        'MaxEpochs',10, ...
        'ExecutionEnvironment', execEnv);

    % Time network training
    tic;
    trainedNet = trainNetwork(data,labels,lgraph,options);
    timings(ii) = toc;

    % Delete parallel pool
    delete(gcp('nocreate'))
end
```

The approximate execution times are:

Release	1 GPU	2 GPUs	4 GPUs
R2020a:	204 s	189 s	157 s
R2020b:	198 s	150 s	99 s

The following chart shows the execution times using one, two, and four GPUs in R2020a and R2020b.
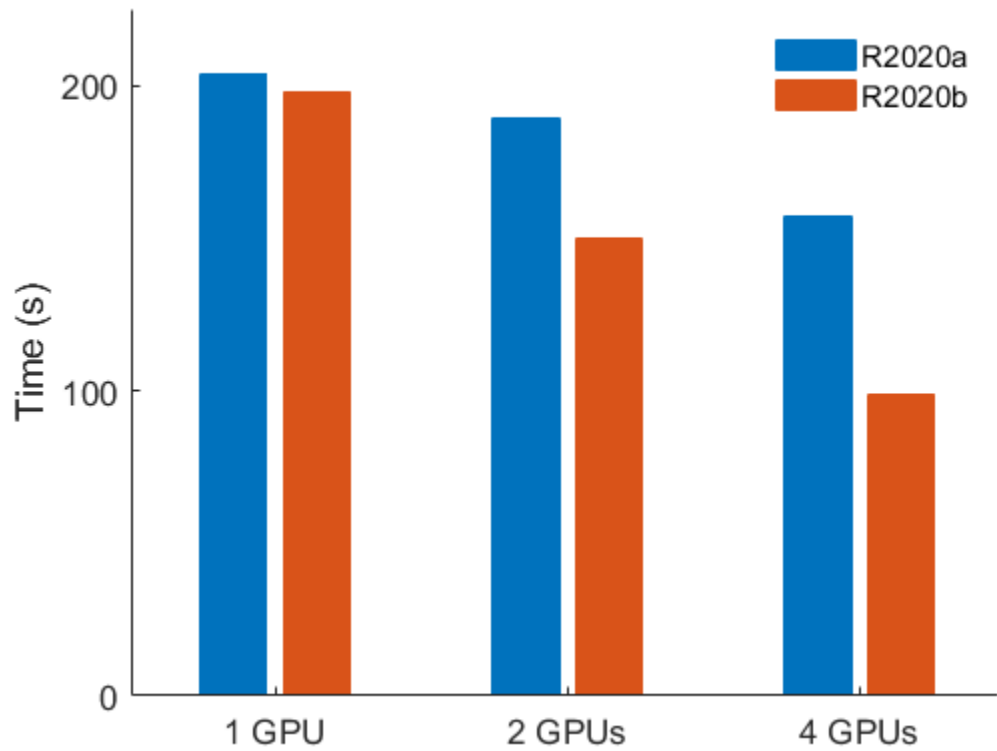
The code was timed on a Windows 10, Intel Xeon E5-2623 v4 @ 2.60 GHz test system with four NVIDIA Titan V 12GB GPUs by running the above script.

predict, classify, and activations Functions: Improved CPU performance for recurrent neural networks

The `predict`, `classify`, and `activations` functions show improved performance for recurrent neural networks when the `'ExecutionEnvironment'` option is `'cpu'`. For example, making predictions using the following test is about 3.4x faster than in the previous release.

```
function timePrediction

layers = [
    sequenceInputLayer(12)
    lstmLayer(100,'InputWeights',rand(400,12),'RecurrentWeights',rand(400,100),'Bias',rand(400,1),'OutputMode','last')
    fullyConnectedLayer(9,'Weights',rand(9,100),'Bias',rand(9,1))
    softmaxLayer
    classificationLayer('Classes',string(1:9))];
net = assembleNetwork(layers);

X = arrayfun(@(~) rand(12,100),1:128,'UniformOutput',false);

tic
for k = 1:200
    YPred = classify(net,X,'MiniBatchSize',32,'ExecutionEnvironment','cpu');
end
toc

end
```

The approximate execution times are:

R2020a: 23.1 seconds

R2020b: 6.75 seconds

The code was timed on a Windows 10, Intel Xeon E5-1650 v4 @ 3.60 GHz test system by running the above script.

dlfeval Function: Improved GPU performance for dlnetwork training

Using the `dlfeval` function to evaluate model gradients using `dlnetwork` objects shows improved performance for models that do not have custom layers. For example, using a GPU to process a mini-batch of 32 images for training using a ResNet-50 network is about 1.3x faster than in the previous release:

```
function timeGradients

net = resnet50;
lgraph = layerGraph(net);
lgraph = removeLayers(lgraph, lgraph.Layers(end).Name);
dlnet = dlnetwork(lgraph);
dlnet = dlupdate(@gpuArray,dlnet);

miniBatchSize = 32;
dlX = dlarray(randn(224,224,3,miniBatchSize,'single'),'SSCB');
dlX = gpuArray(dlX);
fun = @(dlnet,dlX) dlgradient(sum(forward(dlnet,dlX),'all'),dlnet.Learnables);

for i = 1:10
    grad = dlfeval(fun,dlnet,dlX);
end

wait(gpuDevice);
tic;
for i = 1:50
    grad = dlfeval(fun,dlnet,dlX);
end
wait(gpuDevice)
toc

end
```

The approximate execution times are:

R2020a: 16.2 seconds

R2020b: 12.9 seconds

The code was timed on a Linux®, Intel Xeon W-2133 @ 3.60 GHz test system with a NVIDIA Titan RTX GPU by running the above script.

When processing batches of images, expect performance improvements of up to 1.4x for small batches of fixed size.

Functionality being removed or changed

dlmtimes is not recommended
Still runs

`dlmtimes` is not recommended. Use `pagemtimes` instead. The two-input syntax of `pagemtimes` performs the same functionality as `dlmtimes`. For information on how to use `pagemtimes` with `dlarray` inputs, see the `pagemtimes` entry in List of Functions with dlarray Support

R2020a

Version: 14

New Features

Bug Fixes

Compatibility Considerations

Experiment Manager: Design and run experiments to train deep learning networks

Create a deep learning experiment to train networks under various initial conditions and compare the results by using the **Experiment Manager** app. For example, you can use deep learning experiments to:

- Sweep through a range of hyperparameter values to train a deep network.
- Compare the results of using different data sets to train a network.
- Test different deep network architectures by reusing the same set of training data on several networks.

The **Experiment Manager** app provides visualization tools such as training plots, confusion matrices, filters to refine your experiment results, and the ability to define custom metrics to evaluate your results. To improve reproducibility, every time that you run an experiment, the **Experiment Manager** app stores a snapshot of your experiment definition. For example, you can use snapshots to track the hyperparameter combinations that produce each of your results.

For more information, see these examples:

- Create a Deep Learning Experiment for Classification
- Create a Deep Learning Experiment for Regression
- Evaluate Deep Learning Experiments by Using Metric Functions
- Try Multiple Pretrained Networks for Transfer Learning
- Experiment with Weight Initializers for Transfer Learning

Deep Network Designer: Train networks and generate MATLAB code

Train image classification networks using the **Deep Network Designer** app.

The **Deep Network Designer** app now supports importing training and validation data, visualizing data distribution, augmenting images, specifying training options, training networks with progress plots, and exporting trained networks.

For an example showing how to train an image classification network using Deep Network Designer, see Get Started with Deep Network Designer.

You can also generate code MATLAB from the **Deep Network Designer** app. The generated code contains information on the network architecture, training and validation data import, image augmentation, and training options.

For more information, see Generate MATLAB Code from Deep Network Designer.

Deep Network Designer: Easily import pretrained networks for transfer learning

Import pretrained networks for transfer learning using the **Deep Network Designer** app. To load a pretrained network, open the **Deep Network Designer** app and select the icon for the desired pretrained network.

Each icon displays key information about the depth, size, number of parameters and input size of the pretrained network. For an example showing how to retrain a pretrained network, see Transfer Learning with Deep Network Designer.

Deep Network Designer: Import and export networks with multiple inputs or multiple outputs

Use the **Deep Network Designer** app to import, edit, export, and generate code for networks with multiple input or multiple output layers.

Pretrained Networks: Perform transfer learning with DarkNet-19 and DarkNet-53 pretrained convolutional neural networks

You can now install add-ons for the DarkNet-19 and DarkNet-53 pretrained convolutional neural networks. To download and install the pretrained networks, use the Add-On Explorer. You can also download the networks from MathWorks Deep Learning Toolbox Team. After you install the add-ons, use the `darknet19` and `darknet53` functions to load the networks, respectively.

To retrain a network on a new classification task, follow the steps in Train Deep Learning Network to Classify New Images and load the pretrained network you want to use instead of GoogLeNet.

For more information on pretrained neural networks in MATLAB, see Pretrained Deep Neural Networks.

Network Architectures: Load untrained versions of common network architectures

You can now load untrained versions of the pretrained networks in Deep Learning Toolbox. To load an untrained version of a pretrained network as a layer graph, use the corresponding pretrained network function and set the `'Weights'` option to `'none'`. Loading an untrained version of pretrained networks does not require installing a support package.

For a list of pretrained networks in deep Learning Toolbox, see Pretrained Deep Neural Networks.

Deep Learning Data Sets: Explore data sets used for deep learning

For a list of data sets used for different deep learning workflows and how to import them into MATLAB, see Data Sets for Deep Learning.

Conditional Generative Adversarial Networks: Train GANs using data labels and other attributes

A conditional generative adversarial network is a type of type of generative adversarial network that also takes advantage of labels, responses, and other attributes during the training process. You can then use the generator of conditional GANs to generate images with specified classes or attributes.

For an example showing how to train a conditional GAN using labeled data, see Train Conditional Generative Adversarial Network (CGAN).

Generative Adversarial Networks: Monitor GAN training progress and identify common failure modes

Training GANs can be a challenging task. To learn how identify common failure modes and for suggestions on how to fix them, see Monitor GAN Training Progress and Identify Common Failure Modes. For an example showing how to plot GAN training progress, see Train Generative Adversarial Network (GAN).

Image Captioning: Train networks that generate textual captions for images using attention

Train models with an encoder-decoder architecture for image captioning by using pretrained image classification networks as encoders and recurrent neural network (RNN) that takes the extracted features as input and generates captions as decoders. You can incorporate an *attention mechanism* in the decoder that allows the model to focus on parts of the encoded input while generating the caption.

For an example showing how to train an image captioning network, see Image Captioning Using Attention. This example requires Text Analytics Toolbox™.

Multi-label Classification: Define and train networks for multi-label classification

For multi-label classification tasks (where observations can be assigned to more than one independent category), you can train the network using a custom training loop by specifying cross-entropy loss for independent classes. This loss function is also known as binary cross-entropy loss.

To specify cross-entropy loss for independent classes, use the `crossentropy` function and set the `'TargetCategories'` option to `'independent'`.

For an example showing to how to do multi-label classification for text data, see Multilabel Text Classification Using Deep Learning.

Gated Recurrent Units: Train networks for sequence data using gated recurrent unit (GRU) layers

A GRU layer learns dependencies between time steps in time series and sequence data. To include a GRU layer in a layer graph or layer array, use the `gruLayer` function. To apply the GRU operation in a model function used in a custom training loop, use the `gru` function.

For an example showing how to train an image captioning network using gated recurrent units, see Image Captioning Using Attention. This example requires Text Analytics Toolbox.

Global Max Pooling: Reduce network size and help prevent overfitting using global max pooling layers

A global max pooling layer performs downsampling by computing the maximum of the spatial dimensions of the input. For 2-D data, create a global max pooling layer with the `globalMaxPooling2dLayer` function. For 3-D data, use the `globalMaxPooling3dLayer` function.

Custom Training Loops: Specify networks with 3-D layers and networks with multiple inputs or outputs

The `dlnetwork` function now supports layer graphs with multiple input or multiple outputs, and layer graphs with the following 3-D layers:

- `convolution3dLayer`
- `transposedConv3dLayer`
- `crop3dLayer`
- `averagePooling3dLayer`
- `globalAveragePooling3dLayer`
- `maxPooling3dLayer`
- `globalMaxPooling3dLayer`

To evaluate a `dlnetwork` object that has multiple inputs or multiple outputs during training or inference, use the `forward` and `predict` functions, respectively. For an example showing how to train a conditional GAN using multi-input networks, see Train Conditional Generative Adversarial Network (CGAN).

To learn more about custom training loops, see Define Custom Training Loops, Loss Functions, and Networks.

Custom Training Loops: Specify custom layers with custom backward functions

The `dlnetwork` function now supports layer graphs with custom layers with custom backward functions. Specifying a backward function is optional. Use custom backward functions when the forward function does not support automatic differentiation or when you want to use a specific algorithm for the backward pass.

For a list of layers in Deep Learning Toolbox, see List of Deep Learning Layers. For an example showing how to define a custom backward function, see Specify Custom Layer Backward Function. For an example showing how to define a custom backward loss function, see Specify Custom Output Layer Backward Loss Function.

For more information about defining custom deep learning layers, see Define Custom Deep Learning Layers.

Automatic Differentiation: New deep learning operations

Define custom model functions using the following operations:

- Gated recurrent units – Apply gated recurrent unit (GRU) operations using the `gru` function.
- Cross channel normalization – Apply cross channel normalization (also known as local response normalization) using the `crosschannelnorm` function.
- Global average and global max pooling – Apply global average and global max pooling operations using the `avgpool` and `maxpool` functions, respectively, by setting the pooling region size to `'global'`.

- Batch matrix multiplication – Perform matrix multiplication in batches using the `dlmtimes` function.
- Rescaling – Rescale data using the `rescale` function.
- Ceiling and floor operations – Apply the `ceil` and `floor` functions to data in model functions.
- Floating point accuracy – For single and double precision, determine the distance to the next largest floating point number using the `eps` function. Use this value to bound calculations away from zero.

For a list of functions that support automatic differentiation, see List of Functions with dlarray Support. To learn more about defining and training deep learning models using automatic differentiation, see Define Custom Training Loops, Loss Functions, and Networks.

Training Options: Edit training option properties

You can now edit training option properties of `TrainingOptionsSGDM`, `TrainingOptionsADAM`, and `TrainingOptionsRMSProp` objects directly.

For example, to change the mini-batch size after using the `trainingOptions` function, you can edit the `MiniBatchSize` property directly:

```
options = trainingOptions('sgdm');
options.MiniBatchSize = 64;
```

Deep Learning Validation: Access final validation accuracy, loss, and RMSE after training

Access the final validation metrics after training using the information struct output by `trainNetwork`.

- For classification problems, access the validation loss and accuracy from the `FinalValidationLoss` and `FinalValidationAccuracy` fields of the information struct, respectively.
- For regression problems, access the validation loss and RMSE from the `FinalValidationLoss` and `FinalValidationRMSE` fields of the information struct, respectively.

Network Plotting: Plot series networks

The `plot` function now supports `SeriesNetwork` objects.

TensorFlow-Keras Support: Import networks with multiple inputs and multiple outputs

You can import a Keras network with multiple inputs and multiple outputs. Use `importKerasNetwork` if the network includes input size information for the inputs and loss information for the outputs. Otherwise, use `importKerasLayers`. The `importKerasLayers` function inserts placeholder layers for the inputs and outputs. After importing, you can find and replace the placeholder layers by using `findPlaceholderLayers` and `replaceLayer`, respectively.

TensorFlow-Keras Support: Import GRU layers, 2-D global max pooling layers, and PReLU advanced activation layers from TensorFlow-Keras

`importKerasNetwork` and `importKerasLayers` now support GRU layers, 2-D global max pooling layers, and PReLU advanced activation layers. For a full list of supported layers, see `importKerasNetwork` and `importKerasLayers`.

ONNX Support: Import and export networks with multiple inputs and multiple outputs

You can import an ONNX network with multiple inputs and multiple outputs. If the network has multiple inputs and a single output, use `importONNXNetwork`. If the network has multiple outputs, use `importONNXLayers`. The `importONNXLayers` function inserts placeholder layers for the outputs. After importing, you can find and replace the placeholder layers by using `findPlaceholderLayers` and `replaceLayer`, respectively. For an example, see Import ONNX Network with Multiple Outputs.

You can also export a trained MATLAB deep learning network that includes multiple inputs and multiple outputs to the ONNX model format by using the `exportONNXNetwork` function.

ONNX Support: Import and export networks that include exponential linear unit (ELU) layers, GRU layers, and 2-D global max pooling layers

You can now import an ONNX network that includes ELU layers, GRU layers, and 2-D global max pooling layers by using `importONNXNetwork` and `importONNXLayers`. Also, you can export a trained MATLAB deep learning network that includes these layers to the ONNX model format by using `exportONNXNetwork`. For a full list of supported layers, see `importONNXNetwork`, `importONNXLayers`, and `exportONNXNetwork`.

Pretrained Networks: Use SqueezeNet without installing support package

The `squeezenet` function no longer requires installing the Deep Learning Toolbox Model *for SqueezeNet Network* support package. To load a pretrained SqueezeNet network, use the `squeezenet` function.

Deep Learning Examples: Explore deep learning workflows

New examples to help you get started with deep learning:

- Get Started with Deep Network Designer
- Get Started with Transfer Learning
- Interactive Transfer Learning Using SqueezeNet
- Transfer Learning with Deep Network Designer
- Create Simple Sequence Classification Network Using Deep Network Designer
- Visualize Image Classifications Using Maximal and Minimal Activating Images

New examples and topics help you progress with deep learning:

- Train Conditional Generative Adversarial Network (CGAN)
- Update Batch Normalization Statistics in Custom Training Loop
- Update Batch Normalization Statistics Using Model Function
- Specify Custom Layer Backward Function
- Specify Custom Output Layer Backward Loss Function

New examples for experiment management tasks include:

- Create a Deep Learning Experiment for Classification
- Create a Deep Learning Experiment for Regression
- Evaluate Deep Learning Experiments by Using Metric Functions
- Try Multiple Pretrained Networks for Transfer Learning
- Experiment with Weight Initializers for Transfer Learning

New examples for text analytics tasks include:

- Image Captioning Using Attention
- Multilabel Text Classification Using Deep Learning

New examples for computer vision tasks include:

- Point Cloud Classification Using PointNet Deep Learning
- Object Detection Using YOLO v3 Deep Learning
- Object Detection Using SSD Deep Learning
- Import Pretrained ONNX YOLO v2 Object Detector
- Export YOLO v2 Object Detector to ONNX

New examples for image processing tasks include:

- Neural Style Transfer Using Deep Learning

New and updated examples for audio processing tasks include:

- Train Generative Adversarial Network (GAN) for Sound Synthesis
- Speech Emotion Recognition
- Sequential Feature Selection for Audio Features

predict Function: Improved performance for dlnetwork inference

The `predict` function for `dlnetwork` input shows improved performance for models that do not have custom layers. For example, using a GPU to process a single image using a ResNet-50 pretrained network is about 4x faster than in the previous release:

```
% Create dlnetwork and warm up
net = resnet50;
lgraph = layerGraph(net);
lgraph = removeLayers(lgraph, lgraph.Layers(end).Name);
dlnet = dlnetwork(lgraph);
dlnet = dlupdate(@gpuArray,dlnet);
x = dlarray(randn(224,224,3,'single'),'SSCB');
```

```
for i = 1:10
    predict(dlnet,x);
end

% Timing loop
wait(gpuDevice);
tic
for i = 1:100
    y = predict(dlnet,x);
end
wait(gpuDevice)
toc
```

The approximate execution times are:

R2019b: 10.4 s

R2020a: 2.6 s

The code was timed on a Windows 10, Intel Xeon W-2133 @ 3.60 GHz test system with a NVIDIA GeForce RTX 2080 Ti GPU by running the above script.

When processing batches of images, expect performance improvements of up to 4x for small batches of fixed size.

Functionality being removed or changed

rmspropupdate squared gradient decay factor default is 0.9
Behavior change

Starting in R2020a, the default value of the squared gradient decay factor in `rmspropupdate` is `0.9`. In previous versions, the default value was `0.999`. To reproduce the previous default behavior, use one of the following syntaxes:

```
[dlnet,averageSqGrad] = rmspropupdate(dlnet,grad,averageSqGrad,0.001,0.999)
[params,averageSqGrad] = rmspropupdate(params,grad,averageSqGrad,0.001,0.999)
```

sequenceInputLayer ignores padding values when normalizing
Behavior change

Starting in R2020a, `sequenceInputLayer` objects ignore padding values in the input data when normalizing. This means that the `Normalization` option in the `sequenceInputLayer` now makes training invariant to data operations, for example, `'zerocenter'` normalization now implies that the training results are invariant to the mean of the data.

If you train on padded sequences, then the calculated normalization factors may be different in earlier versions and can produce different results.

maxpool indices output argument changes shape and data type
Behavior change

Starting in R2020a, the data type and shape of the indices output argument of the `maxpool` function are changed. The `maxpool` function outputs the indices of the maximum values as a `dlarray` with the same shape and format as the pooled data, instead of a numeric vector.

The indices output of `maxpool` remains compatible with the indices input of `maxunpool`. The `maxunpool` function accepts the indices of the maximum values as a `dlarray` with the same shape

and format as the input data. To prevent errors, use only the indices output of the `maxpool` function as the indices input to the `maxunpool` function.

To reproduce the previous behavior and obtain the indices output as a numeric vector, use the following code:

```
[dlY,indx,inputSize] = maxpool(dlY,poolsize);
indx = extractdata(indx);
indx = reshape(indx,[],1);
```

R2019b

Version: 13.0

New Features

Bug Fixes

Compatibility Considerations

Deep Learning Customization: Define and train complex networks (including GANs) using custom training loops, automatic differentiation, shared weights, and custom loss functions

In R2019b, deep learning functionality is extended to support advanced networks and workflows:

- Create and train generative adversarial networks (GANs) and other networks that require shared weights, such as Siamese networks.
- Define networks using `dlnetwork`, more than 14 deep learning functions, and more than 80 math functions that support the new deep learning data type, `dlarray`.
- Define custom loss functions and training loops.
- Write custom layers without needing to write the backward function.

To get started with customizing deep learning workflows, see:

- Define Custom Training Loops, Loss Functions, and Networks
- Specify Training Options in Custom Training Loop
- Train Network Using Custom Training Loop
- Train Network Using Model Function
- Train Network in Parallel with Custom Training Loop
- Make Predictions Using `dlnetwork` Object
- Make Predictions Using Model Function

New examples of custom deep learning workflows include:

- Train Generative Adversarial Network (GAN)
- Train a Siamese Network for Dimensionality Reduction
- Train a Siamese Network to Compare Images
- Sequence-to-Sequence Classification Using 1-D Convolutions
- Sequence-to-Sequence Translation Using Attention
- Train Variational Autoencoder (VAE) to Generate Images
- Train Network Using Cyclical Learn Rate for Snapshot Ensembling
- Grad-CAM Reveals the Why Behind Deep Learning Decisions

To learn more about automatic differentiation, see:

- Automatic Differentiation Background
- Use Automatic Differentiation In Deep Learning Toolbox

New functions are available for custom deep learning workflows.

- List of Functions with dlarray Support
- Use new functions to create custom training loops.

dlnetwork	Deep learning network for custom training loop

forward	Compute deep learning network output for training
predict	Compute deep learning network output for inference
adamupdate	Update parameters using adaptive moment estimation (Adam)
sgdmupdate	Update parameters using stochastic gradient descent with momentum (SGDM)
rmspropupdate	Update parameters using root mean squared propagation (RMSProp)
dlupdate	Update parameters using custom function

- Use new functions for automatic differentiation.

dlarray	Deep learning array
dlgradient	Compute gradient using automatic differentiation
dlfeval	Evaluate deep learning array function using automatic differentiation
dims	Dimension labels of dlarray
finddim	Find dimensions with specified label
stripdims	Remove dlarray labels
extractdata	Extract data from dlarray
functionToLayerGraph	Convert deep learning array function to a layer graph

- Use new functions for deep learning operations.

dlconv	Deep learning convolution
dltranspconv	Deep learning transposed convolution
lstm	Long short-term memory
fullyconnect	Sum all weighted input data and apply a bias
relu	Apply rectified linear unit activation
leakyrelu	Apply leaky rectified linear unit activation
batchnorm	Normalize each channel of input data
avgpool	Pool data to average value
maxpool	Pool data to maximum value
maxunpool	Unpool the output of a maximum pooling operation
softmax	Apply softmax activation to channel dimension
crossentropy	Categorical cross-entropy loss
sigmoid	Apply sigmoid activation
mse	Half mean squared error

Generative Adversarial Networks: Create and train generative adversarial networks (GANs) for image generation

A generative adversarial network (GAN) is a type of deep learning network that can generate data with similar characteristics as the input training data. A GAN consists of two networks that train together:

1 Generator — Given a vector or random values as input, this network generates data with the same structure as the training data.
2 Discriminator — Given batches of data containing observations from both the training data, and generated data from the generator, this network attempts to classify the observations as "real" or "generated."

The objective of the generator is to generate data that the discriminator classifies as "real." To maximize the performance of the generator, maximize the loss of the discriminator when given generated data. The objective of the discriminator is to not be "fooled" by the generator. To maximize the performance of the discriminator, minimize the loss of the discriminator when given batches of both real and generated data. These strategies result in a generator that generates convincingly realistic data and a discriminator that has learned strong feature representations that are characteristic of the training data.

To learn how to train a GAN using a custom training loop, see Train Generative Adversarial Network (GAN).

Siamese Networks: Create and train Siamese networks for image comparison and dimensionality reduction

Use a `dlnetwork` with custom loss functions and training loops to create and train Siamese networks. A Siamese network consists of two or more identical subnetworks that have the same architecture and share weights and parameters. Siamese networks are useful for making comparisons. To learn more, see the following examples.

- Train a Siamese Network to Compare Images
- Train a Siamese Network for Dimensionality Reduction

Data Preprocessing: Improve training performance using different data normalization options

Data normalization can impact training, classification, and prediction accuracy. Try setting the `Normalization` option in `imageInputLayer`, `image3dInputLayer`, and `sequenceInputLayer` to different values and see which is best for your data.

New normalizations include:

- `'zscore'` — Normalize using z-score normalization.
- `'rescale-symmetric'` — Rescale the input to be in the range [-1, 1].
- `'rescale-zero-one'` — Rescale the input to be in the range [0, 1].
- function handle — Normalize the data using a custom function.

You can normalize using scalar statistics, or specify channel-wise or element-wise normalization using the `NormalizationDimension` option.

The software, by default, automatically calculates the required normalization statistics at training time. To save time when training, specify the required statistics for normalization and set the `'ResetInputNormalization'` option in `trainingOptions` to `false`.

Visualization: Map strongly activating features of input data using occlusion

An occlusion map highlights the behavior of a network by replacing different parts of the data with an occluding mask and measuring how the spatial location of the mask affects the activations of a particular channel. Determine the parts of your input data that strongly affect activations and classification scores using the `occlusionSensitivity` function. To learn more, see Understand Network Predictions Using Occlusion.

Visualization: Visualize the features learned by a DAG network using deep dream

The `deepDreamImage` function now supports DAG networks. Synthesize images that strongly activate DAG network layers using the `deepDreamImage` function. Visualizing these images highlights the features your trained network has learned, helping you understand network behavior.

Multi-Input, Multi-Output Networks: Create and train networks with multiple inputs and multiple outputs

Define and train network architectures with multiple inputs (for example, networks trained on multiple sources and types of data) or multiple outputs (for example, networks that predict both classification and regression responses). For examples showing how to train and make predictions with a network with multiple predictions, see:

- Train Network with Multiple Outputs
- Assemble Muliple-Output Network for Prediction
- Make Predictions Using Model Function

For more information, see Multiple-Input and Multiple-Output Networks.

Long Short-Term Memory Networks: Pad or truncate sequences on the left

The location of the padding and truncation can impact training, classification, and prediction accuracy. Try setting the `'SequencePaddingDirection'` option in `trainingOptions` to `'left'` or `'right'` and see which is best for your data.

Because LSTM layers process sequence data one time step at a time, when the layer `OutputMode` property is `'last'`, any padding in the final time steps can negatively influence the layer output. To pad or truncate sequence data on the left, set the `'SequencePaddingDirection'` option to `'left'`.

For sequence-to-sequence networks (when the `OutputMode` property is `'sequence'` for each LSTM layer), any padding in the first time steps can negatively influence the predictions for the earlier time steps. To pad or truncate sequence data on the right, set the `'SequencePaddingDirection'` option to `'right'`.

To learn more about the effect of padding, truncating, and splitting the input sequences, see Sequence Padding, Truncation, and Splitting.

Long Short-Term Memory Networks: Compute intermediate layer activations

The `activations` function now supports LSTM networks. Investigate and visualize the features learned by LSTM networks from sequence and time series data by extracting the activations using the `activations` function. To learn more, see Visualize Activations of LSTM Network.

ONNX Support: Export networks that combine CNN and LSTM layers and networks that include 3-D CNN layers to ONNX format

You can now export deep networks that combine convolutional neural networks (CNNs) and long short-term memory (LSTM) layers to ONNX format. `exportONNXNetwork` now also supports 3-D CNN layers. For a full list of supported layers, see `exportONNXNetwork`.

Global Average Pooling: Reduce network size and help prevent overfitting using global average pooling layers

A global average pooling layer performs downsampling by computing the mean of the spatial dimensions of the input. For 2-D data, create a global average pooling layer with the `globalAveragePooling2dLayer` function. For 3-D data, use the `globalAveragePooling3dLayer` function.

Cropping: Crop 2-D and 3-D input data to size of reference feature map

Crop 2-D and 3-D input data to the size of a reference feature map using `crop2dLayer` and `crop3dLayer` objects, respectively.

Starting in R2019b, you can use `crop2dLayer` objects in Deep Learning Toolbox without Computer Vision Toolbox. In previous versions, this object required Computer Vision Toolbox.

Deep Learning Examples: Explore deep learning workflows

New examples and topics help you progress with deep learning:

- Preprocess Data for Domain-Specific Deep Learning Applications
- Chemical Process Fault Detection Using Deep Learning
- View Network Behavior Using tsne

New examples for computer vision tasks include:

- Getting Started with Object Detection Using Deep Learning (Computer Vision Toolbox)
- Augment Bounding Boxes for Object Detection
- Augment Pixel Labels for Semantic Segmentation

New examples for image processing tasks include:

- Augment Images for Deep Learning Workflows Using Image Processing Toolbox
- Deep Learning Classification of Large Multiresolution Images

New examples for signal and audio processing tasks include:

- Label QRS Complexes and R Peaks of ECG Signals Using Deep Network
- Pedestrian and Bicyclist Classification Using Deep Learning
- Radar Waveform Classification Using Deep Learning
- Sequential Feature Selection for Speech Emotion Recognition
- Keyword Spotting in Noise Using MFCC and LSTM Networks
- Acoustic Scene Recognition Using Late Fusion

New examples for reinforcement learning tasks include:

- Create Simulink Environment and Train Agent
- Create Agent Using Deep Network Designer and Train Using Image Observations
- Train DDPG Agent to Swing Up and Balance Pendulum with Image Observation
- Train DQN Agent for Lane Keeping Assist Using Parallel Computing

New code generation examples include:

- Generate C++ Code for Object Detection Using YOLO v2 and Intel MKL-DNN
- Code Generation and Deployment of MobileNet-v2 Network to Raspberry Pi
- Deep Learning Prediction on ARM Mali GPU
- Code Generation for a Sequence-to-Sequence LSTM Network

Functionality being removed or changed

AverageImage property of imageInputLayer and image3dInputLayer will be removed
Still runs

The `AverageImage` property of `imageInputLayer` and `image3dInputLayer` will be removed. Use `Mean` instead. To update your code, replace all instances of `AverageImage` with `Mean`. There are no differences between the properties that require additional updates to your code.

imageInputLayer and image3dInputLayer, by default, use channel-wise normalization
Behavior change

Starting in R2019b, `imageInputLayer` and `image3dInputLayer`, use channel-wise normalization by default. In previous versions, these layers used element-wise normalization. To reproduce this behavior, set the `NormalizationDimension` option of these layers to `'element'`.

sequenceInputLayer, by default, uses channel-wise zero-center normalization
Behavior change

Starting in R2019b, `sequenceInputLayer` uses channel-wise zero-center normalization by default when `Normalization` is `'zerocenter'`. In previous versions, this layer used element-wise normalization. To reproduce this behavior, set the `NormalizationDimension` option of this layer to `'element'`.

R2019a

Version: 12.1

New Features

Bug Fixes

Compatibility Considerations

Deep Network Designer: Create networks for computer vision and text applications

Use **Deep Network Designer** to create networks for computer vision and text applications. Deep Network Designer now supports deep learning layers in Computer Vision Toolbox and Text Analytics Toolbox for applications such as semantic segmentation, object detection, and text classification. For a list of layers, see List of Deep Learning Layers.

Deep Network Designer: Generate MATLAB code that recreates your network

Generate MATLAB code that recreates a network constructed in Deep Network Designer and returns it as a `layerGraph` object or a `Layer` array in the MATLAB workspace. Use the generated code to modify the network using the command line and automate deep learning workflows. You can also save any pretrained weights and use the generated code to recreate the network including weights.

For more information, see Generate MATLAB Code from Deep Network Designer.

Convolutions for Image Sequences: Create LSTM networks for video classification and gesture recognition

Create deep learning networks for data containing sequences of images such as video data and medical images.

- To input sequences of images into a network, use `sequenceInputLayer`.
- To apply convolutional operations independently on each time step, first convert the sequences of images to an array of images using a `sequenceFoldingLayer`.
- To restore the sequence structure after applying these operations, use a `sequenceUnfoldingLayer`.
- To convert the output to an array of feature vectors, use a `flattenLayer`. After the flatten layer, you can use LSTM and BiLSTM layers.

For an example, see Classify Videos Using Deep Learning.

Layer Initialization: Initialize layer weights and biases using initializers or a custom function

Initialize layer weights and biases using initializers such as the Glorot initializer (also known as the Xavier initializer), the He initializer, and orthogonal initializers. To specify the initializer for the weights and biases of convolutional layers or fully connected layers, use the `'WeightsInitializer'` and `'BiasInitializer'` name-value pairs of the layers, respectively. To specify the initializer for the input weights, the recurrent weights, and the biases for LSTM and BiLSTM layers, use the `'InputWeightsInitializer'`, `'RecurrentWeightsInitializer'`, and `'BiasInitializer'` name-value pairs, respectively.

You can specify initializers for these layers:

- `batchNormalizationLayer`
- `bilstmLayer`

- `convolution2dLayer`
- `convolution3dLayer`
- `fullyConnectedLayer`
- `groupedConvolution2dLayer`
- `lstmLayer`
- `transposedConv2dLayer`
- `transposedConv3dLayer`
- `wordEmbeddingLayer` (Text Analytics Toolbox)

For an example showing how to compare the different initializers, see Compare Layer Weight Initializers. For an example showing how to create a custom initialization function, see Specify Custom Weight Initialization Function.

Grouped Convolutions: Create efficient deep learning networks with grouped and channel-wise convolutions

When training convolutional neural networks from scratch, for some networks, you can speed up training and prediction by replacing standard convolutions with grouped or channel-wise (also known as depth-wise) convolutions. To create a grouped convolutional layer, use `groupedConvolution2dLayer`. For channel-wise convolution, use `groupedConvolution2dLayer` and set `NumGroups` to `'channel-wise'`.

For an example showing how to create a block of layers for channel-wise separable convolution (also known as depth-wise separable convolution), see Create Layers for Channel-Wise Separable Convolution.

3-D Support: New layers enable deep learning with 3-D data

These new layers enable you to work with 3-D data:

- `image3dInputLayer`
- `convolution3dLayer`
- `transposedConv3dLayer`
- `averagePooling3dLayer`
- `maxPooling3dLayer`
- `concatenationLayer`

These existing layers are enhanced to support 3-D data in deep learning networks:

- `reluLayer`
- `leakyReluLayer`
- `clippedReluLayer`
- `fullyConnectedLayer`
- `softmaxLayer`
- `classificationLayer`
- `regressionLayer`

For a list of available layers, see List of Deep Learning Layers. For an example showing how to train a network using 3-D data, see 3-D Brain Tumor Segmentation Using Deep Learning.

Custom Layers: Create custom layers with multiple inputs or multiple outputs

You can now define custom layers with multiple inputs or multiple outputs. If Deep Learning Toolbox does not provide the deep learning layer you need for your task, then you can define your own layer by specifying the layer forward and backward functions.

For more information about defining custom layers, see Define Custom Deep Learning Layers. To learn how to check that the layer is valid automatically using the `checkLayer` function, see Check Custom Layer Validity.

Deep Learning Acceleration: Optimize deep learning applications using MEX functions

Accelerate prediction, classification, and feature extraction using automatically generated MEX functions. Use the `'Acceleration','mex'` name-value pair with the following functions.

- `activations`
- `classify`
- `predict`

Pretrained Networks: Perform transfer learning with NASNet-Large, NASNet-Mobile, MobileNet-v2, ShuffleNet, Xception, and Places365-GoogLeNet pretrained convolutional neural networks

You can now install add-ons for the NASNet-Large, NASNet-Mobile, MobileNet-v2, ShuffleNet, Xception, and Places365-GoogLeNet pretrained convolutional neural networks. To download and install the pretrained networks, use the Add-On Explorer. You can also download the networks from MathWorks Deep Learning Toolbox Team. After you install the add-ons, use the `nasnetlarge`, `nasnetmobile`, `mobilenetv2`, `shufflenet`, `xception`, and `googlenet` functions to load the networks, respectively. Places365-GoogLeNet is a version of GoogLeNet that is trained on the Places365 data set and classifies images into 365 different place categories, such as field, park, runway, and lobby. To load this network, use `net = googlenet('Weights','places365')`.

To retrain a network on a new classification task, follow the steps in Train Deep Learning Network to Classify New Images and load the pretrained network you want to use instead of GoogLeNet.

For more information on pretrained neural networks in MATLAB, see Pretrained Deep Neural Networks.

Deep Learning Layers: Hyperbolic tangent and exponential linear unit activation layers

You can now use hyperbolic tangent (tanh) and exponential linear unit (ELU) layers as activation layers in deep learning networks. To create a tanh or ELU layer, use `tanhLayer` and `eluLayer`, respectively.

For a list of available layers, see List of Deep Learning Layers.

Deep Learning Visualization: Investigate network predictions using class activation mapping

Follow the example Investigate Network Predictions Using Class Activation Mapping and use the class activation mapping (CAM) technique to investigate and explain the predictions of a deep convolutional neural network for image classification.

Deep Learning Examples: Explore deep learning workflows

New examples and topics help you progress with deep learning:

- Investigate Network Predictions Using Class Activation Mapping
- Classify Videos Using Deep Learning
- Run Multiple Deep Learning Experiments
- Train Network Using Out-of-Memory Sequence Data
- Compare Layer Weight Initializers
- Specify Custom Weight Initialization Function

New examples for computer vision problems include:

- Object Detection Using YOLO v2 Deep Learning
- 3-D Brain Tumor Segmentation Using Deep Learning

New examples for text problems include:

- Classify Text Data Using Convolutional Neural Network
- Classify Out-of-Memory Text Data Using Deep Learning

New examples for signal and audio processing include:

- Cocktail Party Source Separation Using Deep Learning Networks
- Voice Activity Detection in Noise Using Deep Learning
- Modulation Classification with Deep Learning
- Spoken Digit Recognition with Wavelet Scattering and Deep Learning
- Waveform Segmentation Using Deep Learning

New code generation examples include:

- Code Generation for Semantic Segmentation Network using U-net
- Train and Deploy Fully Convolutional Networks for Semantic Segmentation
- Code Generation for Object Detection Using YOLO v2
- Code Generation for Deep Learning on ARM Targets
- Code Generation for Deep Learning on Raspberry Pi
- Deep Learning Prediction with ARM Compute Using `cnncodegen`

Functionality being removed or changed

Glorot is default weights initialization for convolution, transposed convolution, and fully connected layers
Behavior change

Starting in R2019a, the software, by default, initializes the layer weights of `convolution2dLayer`, `transposedConv2dLayer`, and `fullyConnectedLayer` using the Glorot initializer. This behavior helps stabilize training and usually reduces the training time of deep networks.

In previous releases, the software, by default, initializes the layer weights by sampling from a normal distribution with a mean of zero and a variance of 0.01. To reproduce this behavior, set the `'WeightsInitializer'` option of these layers to `'narrow-normal'`.

Glorot is default input weights initialization for LSTM and BiLSTM layers
Behavior change

Starting in R2019a, the software, by default, initializes the layer input weights of `lstmLayer` and `bilstmLayer` using the Glorot initializer. This behavior helps stabilize training and usually reduces the training time of deep networks.

In previous releases, the software, by default, initializes the layer input weights by sampling from a normal distribution a mean of zero and a variance of 0.01. To reproduce this behavior, set the `'InputWeightsInitializer'` option of these layers to `'narrow-normal'`.

Orthogonal is default recurrent weights initialization for LSTM and BiLSTM layers
Behavior change

Starting in R2019a, the software, by default, initializes the layer recurrent weights of LSTM and BiLSTM layers with Q, the orthogonal matrix given by the QR decomposition of $Z = QR$ for a random matrix Z sampled from a unit normal distribution. This behavior helps stabilize training and usually reduces the training time of deep networks.

In previous releases, the software, by default, initializes the layer recurrent weights by sampling from a normal distribution with a mean of zero and a variance of 0.01. To reproduce this behavior, set the `'RecurrentWeightsInitializer'` option of the layer to `'narrow-normal'`.

Custom layers have new properties NumInputs, InputNames, NumOutputs, and OutputNames

Starting in R2019a, custom layers have the new properties `NumInputs`, `InputNames`, `NumOutputs`, and `OutputNames`. These properties enable support for custom layers with multiple inputs and multiple outputs.

If you use a custom layer created in R2018b or earlier, the layer cannot have any properties named `NumInputs`, `InputNames`, `NumOutputs`, or `OutputNames`. You must rename these properties to use the layer in R2019a and onwards.

Cropping property of TransposedConvolution2DLayer will be removed
Still runs

`Cropping` property of `TransposedConvolution2DLayer` will be removed. Use `CroppingSize` instead. To update your code, replace all instances of the `Cropping` property with `CroppingSize`.

matlab.io.datastore.MiniBatchable is not recommended for custom image preprocessing
Still runs

Before R2018a, to perform custom image preprocessing for training deep learning networks, you had to specify a custom read function using the `readFcn` property of `imageDatastore`. However, reading files using a custom read function was slow because `imageDatastore` did not prefetch files.

In R2018a, the four classes `matlab.io.datastore.MiniBatchable`, `matlab.io.datastore.BackgroundDispatchable`, `matlab.io.datastore.Shuffleable`, and `matlab.io.datastore.PartitionableByIndex` were introduced as a solution to perform custom image preprocessing with support for prefetching, shuffling, and parallel training. Implementing a custom mini-batch datastore using these classes has several challenges and limitations.

- In addition to specifying the preprocessing operations, you must also define properties and methods to support reading data in batches, reading data by index, and partitioning and shuffling data.
- You must specify a value for the `NumObservations` property, but this value may be ill-defined or difficult to define in real-world applications.
- Custom mini-batch datastores are not flexible enough to support common deep learning workflows, such as deployed workflows using GPU Coder.

Starting in R2019a, built-in datastores natively support prefetch, shuffling, and parallel training when reading batches of data. The `transform` function is the preferred way to perform custom image preprocessing using built-in datastores. The `combine` function is the preferred way to concatenate read data from multiple datastores, including transformed datastores. Concatenated data can serve as the network inputs and expected responses for training deep learning networks. The `transform` and `combine` functions have several advantages over custom mini-batch datastores.

- The functions enable data preprocessing and concatenation for all types of datastores, including `imageDatastore`.
- The `transform` function requires you to define only the data processing pipeline.
- When used on a deterministic datastore, the functions support `tall` data types and MapReduce.
- The functions support deployed workflows.

For more information about custom image preprocessing, see Preprocess Images for Deep Learning.

matlab.io.datastore.BackgroundDispatchable and matlab.io.datastore.PartitionableByIndex are not recommended
Still runs

`matlab.io.datastore.BackgroundDispatchable` and `matlab.io.datastore.PartitionableByIndex` add support for prefetching and parallel training to custom mini-batch datastores. You can use custom mini-batch datastores to preprocess sequence, time series, or text data, but recurrent networks such as LSTM networks do not support prefetching or parallel and multi-GPU training.

Starting in R2019a, built-in datastores natively support prefetching and parallel training, so custom mini-batch datastores are not recommended for custom image preprocessing.

There are no plans to remove `matlab.io.datastore.BackgroundDispatchable` or `matlab.io.datastore.PartitionableByIndex` at this time.

R2018b

Version: 12.0

New Features

Bug Fixes

Compatibility Considerations

Renamed Product: Neural Network Toolbox renamed to Deep Learning Toolbox

Neural Network Toolbox™ now has the name Deep Learning Toolbox.

Deep Network Designer: Edit and build deep learning networks

Build, visualize, and edit deep learning networks interactively in the **Deep Network Designer** app.

- Import pretrained networks and edit them for transfer learning.
- Edit networks and build new networks from scratch.
- Drag and drop to add new layers and create new connections.
- View and edit layer properties.
- Analyze the network to check for correct architecture and detect problems before training.
- Export the network to the workspace, where you can save or train the network.
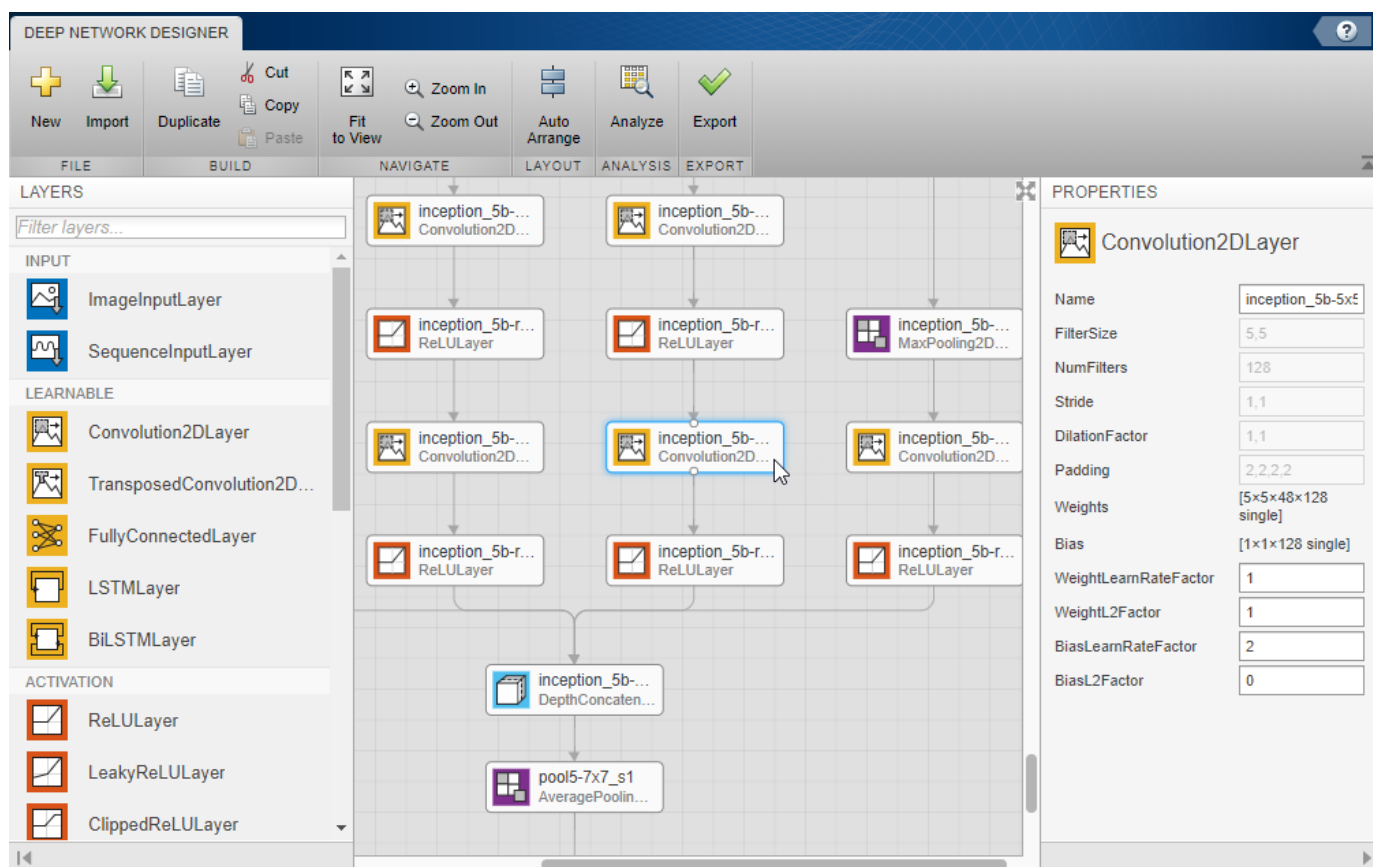
For examples, see:

- Transfer Learning with Deep Network Designer
- Build Networks with Deep Network Designer
- Interactive Transfer Learning Using AlexNet

ONNX Support: Import and export models using the ONNX model format for interoperability with other deep learning frameworks

Export a trained MATLAB deep learning network to the ONNX (Open Neural Network Exchange) model format using the `exportONNXNetwork` function. You can then import the ONNX model to other deep learning frameworks, such as TensorFlow, that support ONNX model import.

Import deep learning networks and network architectures from ONNX using `importONNXNetwork` and `importONNXLayers`.

Network Analyzer: Visualize, analyze, and find problems in network architectures before training

Analyze deep learning network architectures using the `analyzeNetwork` function. Use the network analyzer to visualize and understand the network architecture, check that you have defined the architecture correctly, and detect problems before training. Problems that `analyzeNetwork` detects include missing or disconnected layers, incorrectly sized layer inputs, an incorrect number of layer inputs, and invalid graph structures. For more information, see `analyzeNetwork`.

LSTM Network Validation: Validate networks for time series data automatically during training

Validate LSTM networks at regular intervals during network training, and automatically stop training when validation metrics stop improving.

To perform network validation during training, specify validation data using the `'ValidationData'` name-value pair argument of `trainingOptions`. You can change the validation frequency using the `'ValidationFrequency'` name-value pair argument. For more information, see Specify Validation Data.

For an example showing how to specify validation data for an LSTM network, see Classify Text Data Using Deep Learning.

Network Assembly: Assemble networks from imported layers and weights without training

Assemble networks from imported layers and weights without training using the `assembleNetwork` function. You can use this function for the following tasks:

- Convert a layer array or layer graph to a network ready for prediction.
- Assemble networks from imported layers.
- Modify the weights of a trained network.

For an example showing how to assemble a network from pretrained layers, see Assemble Network from Pretrained Keras Layers.

Output Layer Validation: Verify custom output layers for validity, GPU compatibility, and correctly defined gradients

Check custom output layers for validity using `checkLayer`. You can check GPU compatibility and correctly defined gradients. For more information, see Check Custom Layer Validity.

Visualization: Investigate network predictions using confusion matrix charts

Use `confusionchart` to calculate and plot a confusion matrix for a classification problem using true and predicted labels. A confusion matrix helps you evaluate how well the classifier performs on a data set and identify where it is accurate or inaccurate. Additionally, you can:

- Create a confusion matrix chart from a nonnegative integer-valued confusion matrix.
- Control the appearance and behavior of the confusion matrix chart by modifying `ConfusionMatrixChart Properties`.
- View summary statistics about your data, such as the number of correctly and incorrectly classified observations for each predicted and true class.
- Sort the classes of the confusion matrix by the total number of correctly classified observations, the class-wise precision (positive predictive value), or the class-wise recall (true positive rate) by using `sortClasses`.

Dilated Convolution: Change the dilation factor of convolutional layers to enhance prediction accuracy for tasks such as semantic segmentation

Specify dilated convolutions (also known as atrous convolutions) using the `DilationFactor` property of `convolution2dLayer`. Use dilated convolutions to increase the receptive field (the area of the input that the layer can see) of the layer without increasing the number of parameters or computation.

For an example showing how to use dilated convolutions for semantic segmentation, see Semantic Segmentation Using Dilated Convolutions

Sequence Mini-Batch Datastores: Develop datastores for sequence, time series, and signal data

Use custom mini-batch datastores for sequence, time series, and signal data when data is too large to fit in memory, or to perform specific operations when reading batches of data. You can optionally add support for functionalities such as shuffling during training, parallel and multi-GPU training, and background dispatch. For more information, see Develop Custom Mini-Batch Datastore.

For an example showing how to use a custom mini-batch datastore for sequence data, see Train Network Using Out-of-Memory Sequence Data.

Pretrained Networks: Perform transfer learning with ResNet-18 and DenseNet-201 pretrained convolutional neural networks

You can now install add-ons for the ResNet-18 and DenseNet-201 pretrained convolutional neural networks. To download and install the pretrained networks, use the Add-On Explorer. You can also

download the networks from MathWorks Deep Learning Toolbox Team. After you install the add-ons, use the `resnet18` and `densenet201` functions to load the networks, respectively.

To retrain a network on a new classification task, follow the steps of Train Deep Learning Network to Classify New Images and load ResNet-18 or DenseNet-201 instead of GoogLeNet.

For more information on pretrained neural networks in MATLAB, see Pretrained Convolutional Neural Networks.

TensorFlow-Keras: Import LSTM and BiLSTM layers from TensorFlow-Keras

Import pretrained LSTM and BiLSTM networks and layers from TensorFlow-Keras by using the `importKerasNetwork` and `importKerasLayers` functions.

To use `importKerasNetwork` and `importKerasLayers`, you must install the Deep Learning Toolbox Importer *for TensorFlow-Keras Models* support package. If this support package is not installed, the functions provide a download link.

Caffe Importer: Import directed acyclic graph networks from Caffe

Import directed acyclic graph (DAG) networks and network architectures from Caffe. In previous releases, you could only import networks with layers arranged in a sequence. To import a Caffe network with weights, use `importCaffeNetwork`. To import a network architecture without weights, use `importCaffeLayers`.

LSTM Layer Activation Functions: Specify state and gate activation functions

For LSTM layers, specify state and gate activation functions using the `StateActivationFunction` and `GateActivationFunction` properties of `lstmLayer` respectively. For BiLSTM layers, specify the state and gate activation functions using the `StateActivationFunction` and `GateActivationFunction` properties of `bilstmLayer`, respectively.

Deep Learning: New network layers

You can now use the following layers in deep learning networks:

- `wordEmbeddingLayer`
- `roiInputLayer`
- `roiMaxPooling2dLayer`
- `regionProposalLayer`
- `rpnSoftmaxLayer`
- `rpnClassificationLayer`
- `rcnnBoxRegressionLayer`
- `weightedClassificationLayer` (custom layer example)
- `dicePixelClassificationLayer` (custom layer example)

For a list of available layers, see List of Deep Learning Layers.

Image Data Augmenter: Additional options for augmenting and visualizing images

The `imageDataAugmenter` object now offers more flexibility for transforming images and visualizing the effect of the transformation.

- The `augment` function can apply identical random transformations to multiple images. Use the `augment` function to apply identical transformations to input and response image pairs in a custom mini-batch datastore.

  You can also use the `augment` function to easily visualize the transformations applied to sample images.
- The new `'RandScale'` property of `imageDataAugmenter` scales an image uniformly in the vertical and horizontal directions to maintain the image aspect ratio.
- Several properties of `imageDataAugmenter` now support sampling over disjoint intervals or using nonuniform probability distributions. Specify a custom sampling function using a function handle.

Deep Learning Examples: Explore deep learning workflows

New examples and topics help you progress with deep learning.

- Use the example Train Deep Learning Network to Classify New Images to fine-tune any pretrained network for a new image classification task.
- Compare Pretrained Networks
- Transfer Learning with Deep Network Designer
- Interactive Transfer Learning Using AlexNet
- Build Networks with Deep Network Designer
- Deep Learning Tips and Tricks
- Assemble Network from Pretrained Keras Layers
- List of Deep Learning Layers
- Convert Classification Network into Regression Network
- Resume Training from Checkpoint Network
- Semantic Segmentation Using Dilated Convolutions
- Image Processing Operator Approximation Using Deep Learning
- Assemble Network from Pretrained Keras Layers
- Train Network Using Out-of-Memory Sequence Data
- Denoise Speech Using Deep Learning Networks
- Classify Gender Using Long Short-Term Memory Networks

New examples for text problems include:

- Classify Text Data Using Deep Learning
- Generate Text Using Deep Learning

- Pride and Prejudice and MATLAB
- Word-By-Word Text Generation Using Deep Learning
- Classify Out-of-Memory Text Data Using Custom Mini-Batch Datastore

New examples for deep learning code generation include:

- Deep Learning Prediction with Intel MKL-DNN
- Code Generation for Denoising Deep Neural Network
- Code Generation for Semantic Segmentation Network

Functionality being removed or changed

'ValidationPatience' training option default is Inf
Behavior change

Starting in R2018b, the default value of the `'ValidationPatience'` option in `trainingOptions` is `Inf`, which means that automatic stopping via validation is turned off. This behavior prevents the training from stopping before sufficiently learning from the data.

In previous releases, the default value is 5. To reproduce this behavior, set the `'ValidationPatience'` option in `trainingOptions` to 5.

ClassNames property of ClassificationOutputLayer will be removed
Still runs

`ClassNames` property of `ClassificationOutputLayer` will be removed. Use `Classes` instead. To update your code, replace all instances of the `ClassNames` property with `Classes`. There are some differences between the functions that require additional updates to your code.

The `ClassNames` property contains a cell array of character vectors. The `Classes` property contains a categorical array. To use the `Classes` property with functions that require cell array input, convert the classes using the `cellstr` function.

'ClassNames' option of importKerasNetwork, importCaffeNetwork, and importONNXNetwork will be removed
Still runs

The `'ClassNames'` option of `importKerasNetwork`, `importCaffeNetwork`, and `importONNXNetwork` will be removed. Use `'Classes'` instead. To update your code, replace all instances of `'ClassNames'` with `'Classes'`. There are some differences between the corresponding properties in classification output layers that require additional updates to your code.

The `ClassNames` property of a classification output layer is a cell array of character vectors. The `Classes` property is a categorical array. To use the value of `Classes` with functions that require cell array input, convert the classes using the `cellstr` function.

Different file name for checkpoint networks
Behavior change

Starting in R2018b, when saving checkpoint networks, the software assigns file names beginning with `net_checkpoint_`. In previous releases, the software assigns file names beginning with `convnet_checkpoint_`. For more information, see the `'CheckpointPath'` option in `trainingOptions`.

If you have code that saves and loads checkpoint networks, then update your code to load files with the new name.

R2018a

Version: 11.1

New Features

Bug Fixes

Compatibility Considerations

Long Short-Term Memory (LSTM) Networks: Solve regression problems with LSTM networks and learn from full sequence context using bidirectional LSTM layers

Use recurrent LSTM networks to solve regression problems and use bidirectional LSTM networks to learn from full sequence context.

For an example showing how to create an LSTM network for sequence-to-sequence regression, see Sequence-to-Sequence Regression Using Deep Learning. For an example showing how to forecast future values in a time series, see Time Series Forecasting Using Deep Learning.

To create an LSTM network that learns from complete sequences at each time step, include a bidirectional LSTM layer in your network by using `bilstmLayer`.

Deep Learning Optimization: Improve network training using Adam, RMSProp, and gradient clipping

Use the Adam (adaptive moment estimation) and RMSProp (root-mean-square propagation) optimizers and gradient clipping to train deep learning neural networks.

To create training options for the Adam or RMSProp solvers, use the `trainingOptions` function. `trainingOptions('adam')` and `trainingOptions('rmsprop')` create training options for the Adam and RMSProp solvers, respectively. To specify solver options, use the `'GradientDecayFactor'`, `'SquaredGradientDecayFactor'`, and `'Epsilon'` name-value pair arguments.

To use gradient clipping when training neural networks, use the `'GradientThreshold'` and `'GradientThresholdMethod'` name-value pair arguments of `trainingOptions`.

Deep Learning Data Preprocessing: Read data and define preprocessing operations efficiently for training and prediction

Read and preprocess data efficiently for neural network training, prediction, and validation. You can use a built-in type of mini-batch datastore, such as an `augmentedImageDatastore`, to perform data augmentation with limited preprocessing operations, including resizing, rotation, reflection, and cropping.

Define custom data preprocessing operations by creating your own mini-batch datastore. You can optionally add support for functionality such as shuffling during training, parallel and multi-GPU training, and background dispatch. For more information, see Develop Custom Mini-Batch Datastore.

Compatibility Considerations

In previous releases, you could preprocess images with resizing, rotation, reflection, and other geometric transformations by using an `augmentedImageSource`. The `augmentedImageSource` function now creates an `augmentedImageDatastore` object. An `augmentedImageDatastore` behaves similarly to an `augmentedImageSource`, with additional properties and methods to assist with data augmentation.

You can now use `augmentedImageDatastore` for both training and prediction. In the previous release, you could use `augmentedImageSource` for training but not prediction.

Deep Learning Layer Validation: Check layers for validity, GPU compatibility, and correctly defined gradients

If you create a custom deep learning layer, then you can check that your layer is valid and GPU compatible, and that it calculates gradients correctly, by using `checkLayer`.

Directed Acyclic Graph (DAG) Networks: Accelerate DAG network training using multiple GPUs and compute intermediate layer activations

Speed up training of deep learning DAG networks using multiple GPUs. To train networks using multiple GPUs, specify the `'ExecutionEnvironment'` name-value pair argument of `trainingOptions`.

To use DAG networks for feature extraction or visualization of layer activations, use the `activations` function.

Confusion Matrix: Plot confusion matrices for categorical labels

Plot confusion matrices for categorical labels by using the `plotconfusion` function. `plotconfusion(targets,outputs)` plots a confusion matrix for the true labels `targets` and the predicted labels `outputs`. For an example, see Plot Confusion Matrix Using Categorical Labels.

Multispectral Deep Learning: Train convolutional neural networks on multispectral images

Train convolutional neural networks on images with an arbitrary number of channels. To specify the number of input channels to a network, set the `InputSize` property of the image input layer. For an example, see Semantic Segmentation of Multispectral Images Using Deep Learning.

Directed Acyclic Graph (DAG) Network Editing: Replace a layer in a layer graph more easily

Easily replace a layer in a `LayerGraph` object with a new layer or array of layers by using the `replaceLayer` function. In previous releases, you could replace layers by editing the layer graph, but you had to update the layer connections manually. The new function updates the layer connections automatically.

The `replaceLayer` function requires the Neural Network Toolbox Importer *for TensorFlow-Keras Models* support package. If this support package is not installed, type `importKerasLayer` or `importKerasNetwork` in the command line for a download link.

Pretrained Networks: Accelerate transfer learning by freezing layer weights

Speed up training of pretrained convolutional neural networks by freezing the weights of initial network layers. Freeze the layer weights by setting the learning rate factors of the layers to zero. If

you freeze the weights of the initial layers of a network, then `trainNetwork` does not compute the gradients of the frozen layer weights. For an example showing how to freeze layer weights, see Transfer Learning Using GoogLeNet.

Pretrained Networks: Transfer learning with pretrained SqueezeNet and Inception-ResNet-v2 convolutional neural networks

You can now install add-ons for the SqueezeNet and Inception-ResNet-v2 pretrained convolutional neural networks. To download and install the pretrained networks, use the Add-On Explorer. You can also download the networks from MathWorks Neural Network Toolbox Team. After you install the add-ons, use the `squeezenet` and `inceptionresnetv2` functions to load the networks, respectively.

To retrain a network on a new classification task, follow the steps of Transfer Learning Using GoogLeNet. Load a SqueezeNet or Inception-ResNet-v2 network instead of GoogLeNet, and change the names of the layers that you remove and connect to match the names of your pretrained network. For more information, see `squeezenet` and `inceptionresnetv2`.

For more information on pretrained neural networks in MATLAB, see Pretrained Convolutional Neural Networks.

Deep Learning Network Analyzer: Visualize, analyze, and find issues in network architectures

Analyze deep learning network architectures using the `analyzeNetwork` function. Use the network analyzer to visualize and understand the network architecture, check that you have defined the architecture correctly, and detect problems before training. Problems that `analyzeNetwork` detects include missing or disconnected layers, mismatching or incorrect sizes of layer inputs, incorrect number of layer inputs, and invalid graph structures.

The `analyzeNetwork` function requires the Deep Learning Network Analyzer for Neural Network Toolbox support package. To download and install support package, use the Add-On Explorer. You can also download the support package from MathWorks Neural Network Toolbox Team. For more information, see `analyzeNetwork`.

ONNX Support: Import and export models using the ONNX model format for interoperability with other deep learning frameworks

Export a trained MATLAB deep learning network to the ONNX (Open Neural Network Exchange) model format using the `exportONNXNetwork` function. You can then import the ONNX model to other deep learning frameworks, such as TensorFlow, that support ONNX model import.

Import deep learning networks and network architectures from ONNX using `importONNXNetwork` and `importONNXLayers`.

Deep Learning Speech Recognition: Train a simple deep learning model to detect speech commands

Use the Deep Learning Speech Recognition example to learn how to train a simple neural network to recognize a given set of speech commands (Requires Audio Toolbox).

Parallel Deep Learning Workflows: Explore deep learning with multiple GPUs locally or in the cloud

Use new examples to explore options for scaling up deep learning training. You can use multiple GPUs locally or in the cloud without changing your code. Use parallel computing to train multiple networks locally or on cloud clusters, and use datastores to access cloud data. New examples include:

- Train Network in the Cloud Using Built-in Parallel Support
- Use parfor to Train Multiple Deep Learning Networks
- Use `parfeval` to Train Multiple Deep Learning Networks
- Upload Deep Learning Data to the Cloud
- Send Deep Learning Batch Job To Cluster

To learn about options, see Scale Up Deep Learning in Parallel and in the Cloud.

Deep Learning Examples: Explore deep learning applications

Use examples to learn about different applications of deep learning. New examples for sequence, time series, text, and image problems include:

- Deep Learning Speech Recognition
- Train Residual Network on CIFAR-10
- Time Series Forecasting Using Deep Learning
- Sequence-to-Sequence Classification Using Deep Learning
- Sequence-to-Sequence Regression Using Deep Learning
- Classify Text Data Using Deep Learning
- Semantic Segmentation of Multispectral Images Using Deep Learning
- Single Image Super-Resolution Using Deep Learning
- JPEG Image Deblocking Using Deep Learning
- Remove Noise from Color Image Using Pretrained Neural Network

For more examples of deep learning applications, see Deep Learning Applications and Deep Learning GPU Code Generation.

Functionality Being Removed or Changed

Functionality	Result	Use Instead	Compatibility Considerations
Default output format of `activations`	Still runs	Not applicable	In R2018a, the new default output format of `activations` is a 4-D array. To reproduce the old default behavior, set the `'OutputAs'` value to `'rows'`.

Functionality	Result	Use Instead	Compatibility Considerations
`augmentedImageSource`	Still runs	`augmentedImageDatastore`	In R2018a, you cannot create an `augmentedImageSource` object. The `augmentedImageSource` function now creates an `augmentedImageDatastore` object.
`OutputSize` property of `lstmLayer`	Still runs	`NumHiddenUnits` property	Replace all instances of the `OutputSize` property of `lstmLayer` objects with `NumHiddenUnits`.

R2017b

Version: 11.0

New Features

Bug Fixes

Compatibility Considerations

Directed Acyclic Graph (DAG) Networks: Create deep learning networks with more complex architecture to improve accuracy and use many popular pretrained models

You can create and train DAG networks for deep learning. A DAG network is a neural network whose layers can be arranged as a directed acyclic graph. DAG networks can have a more complex architecture with layers that have inputs from, or outputs to, multiple layers.

To create and train a DAG network:

- Create a `LayerGraph` object using `layerGraph`. The layer graph specifies the network architecture. You can create an empty layer graph and then add layers to it. You can also create a layer graph directly from an array of network layers. The layers in the graph are automatically connected sequentially.
- Add layers to the layer graph using `addLayers` and remove layers from the graph using `removeLayers`.
- Connect layers of the layer graph using `connectLayers` and disconnect layers using `disconnectLayers`.
- Plot the network architecture using `plot`.
- Train the network using the layer graph as the layers input argument to `trainNetwork`. The trained network is a `DAGNetwork` object.
- Perform classification and prediction on new data using `classify` and `predict`.

For an example showing how to create and train a DAG network, see Create and Train DAG Network for Deep Learning.

You can also load a pretrained DAG network by installing the Neural Network Toolbox Model *for GoogLeNet Network* add-on. For a transfer learning example, see Transfer Learning Using GoogLeNet. For more information, see `googlenet`.

Long Short-Term Memory (LSTM) Networks: Create deep learning networks with the LSTM recurrent neural network topology for time-series classification and prediction

You can create and train LSTM networks for deep learning. LSTM networks are a type of recurrent neural network (RNN) that learn long-term dependencies between time steps of sequence data.

LSTM networks can be used for the following types of problems:

- Predict labels for a time series (sequence-to-label classification).
- Predict a sequence of labels for a time series (sequence-to-sequence classification).

To create an LSTM network:

- Include a sequence input layer using `sequenceInputLayer`, which inputs time-series data into the network.
- Include an LSTM layer using `lstmLayer`, which defines the LSTM architecture of the network.

For an example showing sequence-to-label classification, see Classify Sequence Data Using LSTM Networks.

You might want to make multiple predictions on parts of a long sequence, or might not have the complete time series in advance. For these tasks, you can make the LSTM network remember and forget the network state between predictions. To configure the state of LSTM networks, use the following functions:

- Make predictions and update the network state using `classifyAndUpdateState` and `predictAndUpdateState`.
- Reset the network state using `resetState`.

To learn more, see Long Short-Term Memory Networks.

Deep Learning Validation: Automatically validate network and stop training when validation metrics stop improving

You can validate deep neural networks at regular intervals during network training, and automatically stop training when validation metrics stop improving.

To perform network validation during training, specify validation data using the 'ValidationData' name-value pair argument of `trainingOptions`. By default, the software validates the network every 50 training iterations by predicting the response of the validation data and calculating the validation loss and accuracy (root mean square error for regression networks). You can change the validation frequency using the 'ValidationFrequency' name-value pair argument.

Network training stops when the validation loss stops improving. By default, if the validation loss is larger than or equal to the previously smallest loss five times in a row, then network training stops. To change the number of times that the validation loss is allowed to not decrease before training stops, use the 'ValidationPatience' name-value pair argument.

For more information, see Specify Validation Data.

Deep Learning Layer Definition: Define new layers with learnable parameters, and specify loss functions for classification and regression output layers

You can define new deep learning layers and specify your own forward propagation, backward propagation, and loss functions. To learn more, see Define New Deep Learning Layers.

- For an example showing how to define a PReLU layer, a layer with learnable parameters, see Define a Layer with Learnable Parameters.
- For an example showing how to define a classification output layer and specify a loss function, see Define a Classification Output Layer.
- For an example showing how to define a regression output layer and specify a loss function, see Define a Regression Output Layer.

Deep Learning Training Plots: Monitor training progress with plots of accuracy, loss, validation metrics, and more

You can monitor deep learning training progress by plotting various metrics during training. Plot accuracy, loss, and validation metrics to determine if and how quickly the network accuracy is improving, and whether the network is starting to overfit the training data. During training, you can

stop training and return the current state of the network by clicking the stop button in the top-right corner. For example, you might want to stop training when the accuracy of the network reaches a plateau and it is clear that the accuracy is no longer improving.

To turn on the training progress plot, use the 'Plots' name-value pair argument of `trainingOptions`. For more information, see Monitor Deep Learning Training Progress.

Deep Learning Image Preprocessing: Efficiently resize and augment image data for training

You can now preprocess images for network training with more options, including resizing, rotation, reflection, and other geometric transformations. To train a network using augmented images, create an `augmentedImageSource` and use it as an input argument to `trainNetwork`. You can configure augmentation options using the `imageDataAugmenter` function. For more information, see Preprocess Images for Deep Learning.

Augmentation helps to prevent the network from overfitting and memorizing the exact details of the training images. It also increases the effective size of the training data set by generating new images based on the training images. For example, use augmentation to generate new images that randomly flip the training images along the vertical axis, and randomly translate the training images horizontally and vertically.

To resize images in other contexts, such as for prediction, classification, and network validation during training, use `imresize`.

Compatibility Considerations

In previous releases, you could perform limited image cropping and reflection using the `DataAugmentation` property of `imageInputLayer`. The `DataAugmentation` property is not recommended. Use `augmentedImageSource` instead.

Bayesian Optimization of Deep Learning: Find optimal settings for training deep networks (Requires Statistics and Machine Learning Toolbox)

Find optimal network parameters and training options for deep learning using Bayesian optimization and the bayesopt function. For an example, see Deep Learning Using Bayesian Optimization.

GoogLeNet Pretrained Network: Transfer learning with pretrained GoogLeNet convolutional neural network

You can now install the Neural Network Toolbox Model *for GoogLeNet Network* add-on.

You can access the model using the `googlenet` function. If the Neural Network Toolbox Model *for GoogLeNet Network* support package is not installed, then the function provides a link to the required support package in the Add-On Explorer. GoogLeNet won the ImageNet Large-Scale Visual Recognition Challenge in 2014. The network is smaller and typically faster than VGG networks, and smaller and more accurate than AlexNet on the ImageNet challenge data set. The network is a directed acyclic graph (DAG) network, and `googlenet` returns the network as a `DAGNetwork` object. You can use this pretrained model for classification and transfer learning. For an example, see

Transfer Learning Using GoogLeNet. For more information on pretrained neural networks in MATLAB, see Pretrained Convolutional Neural Networks.

ResNet-50 and ResNet-101 Pretrained Networks: Transfer learning with pretrained ResNet-50 and ResNet-101 convolutional neural networks

You can now install add-ons for the ResNet-50 and ResNet-101 pretrained convolutional neural networks. To download and install the pretrained networks, use the Add-On Explorer. To learn more about finding and installing add-ons, see Get Add-Ons. You can also download the networks from MathWorks Neural Network Toolbox Team. After you install the add-ons, use the `resnet50` and `resnet101` functions to load the networks, respectively.

To retrain a network on a new classification task, follow the steps of Transfer Learning Using GoogLeNet. Load a ResNet network instead of GoogLeNet, and change the names of the layers that you remove and connect to match the names of the ResNet layers. To extract the layers and architecture of the network for further processing, use `layerGraph`. For more information, see `resnet50` and `resnet101`.

For more information on pretrained neural networks in MATLAB, see Pretrained Convolutional Neural Networks.

Inception-v3 Pretrained Network: Transfer learning with pretrained Inception-v3 convolutional neural network

You can now install the add-on for the Inception-v3 pretrained convolutional neural network. To download and install the pretrained network, use the Add-On Explorer. To learn more about finding and installing add-ons, see Get Add-Ons. You can also download the network from MathWorks Neural Network Toolbox Team. After you install the add-on, use the `inceptionv3` function to load the network.

To retrain the network on a new classification task, follow the steps of Transfer Learning Using GoogLeNet. Load the Inception-v3 network instead of GoogLeNet, and change the names of the layers that you remove and connect to match the names of the Inception-v3 layers. To extract the layers and architecture of the network for further processing, use `layerGraph`. For more information, see `inceptionv3`.

For more information on pretrained neural networks in MATLAB, see Pretrained Convolutional Neural Networks.

Batch Normalization Layer: Speed up network training and reduce sensitivity to network initialization

Use batch normalization layers between convolutional layers and nonlinearities, such as ReLU layers, to speed up network training and reduce the sensitivity to network initialization. Batch normalization layers normalize the activations and gradients propagating through a neural network, making network training an easier optimization problem. To take full advantage of this fact, you can try increasing the learning rate. Because the optimization problem is easier, the parameter updates can be larger and the network can learn faster.

To create a batch normalization layer, use `batchNormalizationLayer`.

Deep Learning: New network layers

You can now use the following layers in deep learning networks:

- Batch normalization layer — Create a layer using `batchNormalizationLayer`.
- Transposed convolution layer — Create a layer using `transposedConv2dLayer`.
- Max unpooling layer — Create a layer using `maxUnpooling2dLayer`.
- Leaky Rectified Linear Unit (ReLU) layer — Create a layer using `leakyReluLayer`.
- Clipped Rectified Linear Unit (ReLU) layer — Create a layer using `clippedReluLayer`.
- Addition layer — Create a layer using `additionLayer`.
- Depth concatenation layer — Create a layer using `depthConcatenationLayer`.
- Sequence input layer for long short-term memory (LSTM) networks — Create a layer using `sequenceInputLayer`.
- LSTM layer — Create a layer using `LSTMLayer`.

Pretrained Models: Import pretrained CNN models and layers from TensorFlow-Keras

You can import pretrained CNN models and weights from TensorFlow-Keras by using the `importKerasNetwork` function. This function imports a Keras model as a `SeriesNetwork` or `DAGNetwork` object, depending on the type of the Keras network. You can then use the imported classification or regression model for prediction or transfer learning on new data.

Alternatively, you can import CNN layers from TensorFlow-Keras by using the `importKerasLayers` function. This function imports the network architecture as a `Layer`array or `LayerGraph` object. You can then specify the training options using the `trainingOptions` function and train this network using the `trainNetwork` function.

For both `importKerasNetwork` and `importKerasLayers`, you must install the Neural Network Toolbox Importer *for TensorFlow-Keras Models* add-on from the MATLAB® Add-Ons menu.

Functionality Being Removed or Changed

Functionality	Result	Use Instead	Compatibility Considerations
`DataAugmentation` property of the `imageInputLayer`	Still runs	`augmentedImageSource`	The `DataAugmentation` property of `imageInputLayer` is not recommended. Use `augmentedImageSource` instead. For more information, see Preprocess Images for Deep Learning.

Functionality	Result	Use Instead	Compatibility Considerations
`Padding` property of `Convolution2dLayer`, `MaxPooling2dLayer`, and `AveragePooling2dLayer` objects	Warns	`PaddingSize` property of `Convolution2dLayer`, `MaxPooling2dLayer`, and `AveragePooling2dLayer` objects	Replace all instances of `Padding` property with `PaddingSize`. When you create network layers, use the `'Padding'` name-value pair argument to specify the padding. For more information, see `Convolution2dLayer`, `MaxPooling2dLayer`, and `AveragePooling2dLayer`.

R2017a

Version: 10.0

New Features

Bug Fixes

Deep Learning for Regression: Train convolutional neural networks (also known as ConvNets, CNNs) for regression tasks

You can now perform regression for numeric targets (responses) using convolutional neural networks. While defining your network, specify `regressionLayer` as the last layer. Specify the training parameters using the `trainingOptions` function. Train your network using the `trainNetwork` function. To try a regression example showing how to predict angles of rotation of handwritten digits, see Train a Convolutional Neural Network for Regression.

Pretrained Models: Transfer learning with pretrained CNN models AlexNet, VGG-16, and VGG-19, and import models from Caffe (including Caffe Model Zoo)

For pretrained convolutional neural network (CNN) models, AlexNet, VGG-16, and VGG-19, from the MATLAB Add-Ons menu, you can now install the following add-ons:

- Neural Network Toolbox Model *for AlexNet Network*
- Neural Network Toolbox Model *for VGG-16 Network*
- Neural Network Toolbox Model *for VGG-19 Network*

You can access the models using the functions `alexnet`, `vgg16`, and `vgg19`. These models are `SeriesNetwork` objects. You can use these pretrained models for classification and transfer learning.

You can also import other pretrained CNN models from Caffe by using the `importCaffeNetwork` function. This function imports models as a `SeriesNetwork` object. You can then use these models for classifying new data.

Alternatively, you can import CNN layers from Caffe by using the `importCaffeLayers` function. This function imports the layer architecture as a `Layer` array. You can then specify the training options using the `trainingOptions` function and train this network using the `trainNetwork` function.

For both `importCaffeNetwork` and `importCaffeLayers`, you can install the Neural Network Toolbox Importer *for Caffe Models* add-on from the MATLAB® Add-Ons menu.

Deep Learning with Cloud Instances: Train convolutional neural networks using multiple GPUs in MATLAB and MATLAB Distributed Computing Server for Amazon EC2

You can use MATLAB to perform deep learning in the cloud using Amazon Elastic Compute Cloud (Amazon EC2®) with new P2 instances and data stored in the cloud. If you do not have a suitable GPU available for faster training of a convolutional neural network, you can use Amazon Elastic Compute Cloud instead. Try different numbers of GPUs per machine to accelerate training. You can compare and explore the performance of multiple deep neural network configurations to find the best tradeoff of accuracy and memory use. Deep learning in the cloud also requires Parallel Computing Toolbox. For details, see Deep Learning in the Cloud.

Deep Learning with Multiple GPUs: Train convolutional neural networks on multiple GPUs on PCs (using Parallel Computing Toolbox) and clusters (using MATLAB Distributed Computing Server)

You can now train convolutional neural networks (ConvNets) on multiple GPUs and on clusters. Specify the required hardware using the `ExecutionEnvironment` name-value pair argument in the call to the `trainingOptions` function.

Deep Learning with CPUs: Train convolutional neural networks on CPUs as well as GPUs

You can now train a convolutional neural network (ConvNet) on a CPU using the `trainNetwork` function. If there is no available GPU, by default, then `trainNetwork` uses a CPU to train the network. You can also train a ConvNet on multiple CPU cores on your desktop or a cluster using `'ExecutionEnvironment','parallel'`.

For specifying the hardware on which to train the network, and for system requirements, see the `ExecutionEnvironment` name-value pair argument on `trainingOptions`.

Deep Learning Visualization: Visualize the features ConvNet has learned using deep dream and activations

`deepDreamImage` synthesizes images that strongly activate convolutional neural network (ConvNet) layers using a version of the deep dream algorithm. Visualizing these images highlights the features your trained ConvNet has learned, helping you understand and diagnose network behavior. For examples, see Deep Dream Images Using AlexNet and Visualize Features of a Convolutional Neural Network.

You can also display network activations on an image to investigate features the network has learned to identify. To try an example, see Visualize Activations of a Convolutional Neural Network.

table Support: Use data in tables for training of and inference with ConvNets

The `trainNetwork` function and `predict`, `activations`, and `classify` methods now accept data stored in a `table` for classification and regression problems. For details on how to specify your data, see the input argument descriptions on the function and method pages.

Progress Tracking During Training: Specify custom functions for plotting accuracy or stopping at a threshold

When training convolutional neural networks, you can specify one or more custom functions to call at each iteration during training. You can access and act on information during training, for example, to plot accuracy, or stop training early based on a threshold. Specify the functions using the `OutputFcn` name-value pair argument in `trainingOptions`. For examples, see Plot Training Accuracy During Network Training and Plot Progress and Stop Training at Specified Accuracy.

Deep Learning Examples: Get started quickly with deep learning

New examples and topics help you get started quickly with deep learning in MATLAB.

To find out what tasks you can do, see Deep Learning in MATLAB. To learn about convolutional neural networks and how they work in MATLAB, see:

- Introduction to Convolutional Neural Networks
- Specify Layers of Convolutional Neural Network
- Set Up Parameters and Train Convolutional Neural Network

New examples include:

- Try Deep Learning in 10 Lines of MATLAB Code
- Create Simple Deep Learning Network for Classification
- Transfer Learning and Fine-Tuning of Convolutional Neural Networks
- Transfer Learning Using AlexNet
- Feature Extraction Using AlexNet
- Deep Dream Images Using AlexNet
- Visualize Activations of a Convolutional Neural Network
- Visualize Features of a Convolutional Neural Network
- Create Typical Convolutional Neural Networks
- Plot Training Accuracy During Network Training
- Plot Progress and Stop Training at Specified Accuracy
- Resume Training from a Checkpoint Network
- Train a Convolutional Neural Network for Regression

R2016b

Version: 9.1

New Features

Bug Fixes

Compatibility Considerations

Deep Learning with CPUs: Run trained CNNs to extract features, make predictions, and classify data on CPUs as well as GPUs

You can choose a CPU to run a pretrained network for extracting features using `activations`, predicting image class scores using `predict`, and estimating image classes using `classify`. To specify the hardware on which to run the network, use the `'ExecutionEnvironment'` name-value pair argument in the call to the specific method.

Training a convolutional neural network (ConvNet) requires a GPU. To train a ConvNet, or to run a pretrained network on a GPU, you must have Parallel Computing Toolbox and a CUDA-enabled NVIDIA GPU with compute capability 3.0 or higher.

Deep Learning with Arbitrary Sized Images: Run trained CNNs on images that are different sizes than those used for training

You can run a trained convolutional neural network on arbitrary image sizes to extract features using the `activations` method with `channels` output option. For other output options, the sizes of the images you use in `activations` must be the same as the sizes of the ones used for training. To specify the `channels` output option, use the `OutputAs` name-value pair argument in the call to `activations`.

Performance: Train CNNs faster when using ImageDatastore object

`ImageDatastore` allows batch-reading of JPG or PNG image files using prefetching. This feature enables faster training of convolutional neural networks (ConvNets). If you use a custom function for reading the images, prefetching does not occur.

Deploy Training of Models: Deploy training of a neural network model via MATLAB Compiler or MATLAB Compiler SDK

Use MATLAB Runtime to deploy functions that can train a model. You can deploy MATLAB code that trains neural networks as described in Create Standalone Application from Command Line and Package Standalone Application with Application Compiler App.

The following methods and functions are NOT supported in deployed mode:

- Training progress dialog, `nntraintool`.
- `genFunction` and `gensim` to generate MATLAB code or Simulink blocks
- `view` method
- `nctool`, `nftool`, `nnstart`, `nprtool`, `ntstool`
- Plot functions (such as `plotperform`, `plottrainstate`, `ploterrhist`, `plotregression`, `plotfit`, and so on)

generateFunction Method: generateFunction generates code for matrices by default

`'MatrixOnly'` name-value pair argument of `generateFunction` method has no effect. `generateFunction` by default generates code for only matrices.

Compatibility Considerations

You do not need to specify for `generateFunction` to generate code for matrices. Previously, you needed to specify `'MatrixOnly',true`.

alexnet Support Package: Download and use pre-trained convolutional neural network (ConvNet)

You can use pretrained Caffe version of AlexNet convolutional neural network. Download the network from the Add-Ons menu.

For more information about the network, see Pretrained Convolutional Neural Network.

R2016a

Version: 9.0

New Features

Bug Fixes

Deep Learning: Train deep convolutional neural networks with built-in GPU acceleration for image classification tasks (using Parallel Computing Toolbox)

The new functionality enables you to

- Construct convolutional neural network (CNN) architecture (see `Layer`).
- Specify training options using `trainingOptions`.
- Train CNNs using `trainNetwork` for data in 4D arrays or `ImageDatastore`.
- Make predictions of class labels using a trained network using `predict` or `classify`.
- Extract features from a trained network using `activations`.
- Perform transfer learning. That is, retrain the last fully connected layer of an existing CNN on new data.

NOTE: This feature requires the Parallel Computing Toolbox and a CUDA-enabled NVIDIA GPU with compute capability 3.0 or higher.

R2015b

Version: 8.4

New Features

Bug Fixes

Autoencoder neural networks for unsupervised learning of features using the trainAutoencoder function

You can train autoencoder neural networks to learn features using the `trainAutoencoder` function. The trained network is an `Autoencoder` object. You can use the trained autoencoder to predict the inputs for new data, using the `predict` method. For all the properties and methods of the object, see the `Autoencoder` class page.

Deep learning using the stack function for creating deep networks from autoencoders

You can create deep networks using the `stack` method. To create a deep network, after training the autoencoders, you can

1   Extract features from autoencoders using the `encode` method.
2   Train a softmax layer for classification using the `trainSoftmaxLayer` function.
3   Stack the encoders and the softmax layer to form a deep network, and train the deep network.

The deep network is a `network` object.

Improved speed and memory efficiency for training with Levenberg-Marquardt (trainlm) and Bayesian Regularization (trainbr) algorithms

An optimized MEX version of the Jacobian backpropagation algorithm allows faster training and reduces memory requirements for training static and open-loop networks using the `trainlm` and `trainbr` functions.

Cross entropy for a single target variable

The `crossentropy` function supports binary encoding, that is, when there are only two classes and N = 1 (N is the number of rows in the `targets` input argument).

R2015a

Version: 8.3

New Features

Bug Fixes

Progress update display for parallel training

The Neural Network Training tool (`nntraintool`) now displays progress updates when conducting parallel training of a network.

R2014b

Version: 8.2.1

Bug Fixes

R2014a

Version: 8.2

New Features

Bug Fixes

Training panels for Neural Fitting Tool and Neural Time Series Tool Provide Choice of Training Algorithms

The training panels in the Neural Fitting and Neural Time Series tools now let you select a training algorithm before clicking **Train**. The available algorithms are:

- Levenberg-Marquardt (`trainlm`)
- Bayesian Regularization (`trainbr`)
- Scaled Conjugate Gradient (`trainscg`)

For more information on using Neural Fitting, see Fit Data with a Neural Network.

For more information on using Neural Time Series, see Neural Network Time Series Prediction and Modeling.

Bayesian Regularization Supports Optional Validation Stops

Because Bayesian-Regularization with `trainbr` can take a long time to stop, validation used with Bayesian-Regularization allows it to stop earlier, while still getting some of the benefits of weight regularization. Set the training parameter `trainParam.max_fail` to specify when to make a validation stop. Validation is disabled for `trainbr` by default when `trainParam.max_fail` is set to 0.

For example, to train as before without validation:

```
[x,t] = house_dataset;
net = feedforwardnet(10,'trainbr');
[net,tr] = train(net,x,t);
```

To train with validation:

```
[x,t] = house_dataset;
net = feedforwardnet(10,'trainbr');
net.trainParam.max_fail = 6;
[net,tr] = train(net,x,t);
```

Neural Network Training Tool Shows Calculations Mode

Neural Network Training Tool now shows its calculations mode (i.e., MATLAB, GPU) in its **Algorithms** section.

R2013b

Version: 8.1

New Features

Bug Fixes

Compatibility Considerations

Function code generation for application deployment of neural network simulation (using MATLAB Coder, MATLAB Compiler, and MATLAB Builder products)

- "New Function: genFunction" on page 19-2
- "Enhanced Tools" on page 19-3

New Function: genFunction

The function `genFunction` generates a stand-alone MATLAB function for simulating any trained neural network and preparing it for deployment in many scenarios:

- Document the input-output transforms of a neural network used as a calculation template for manual reimplementations of the network
- Create a Simulink block using the MATLAB Function block
- Generate C/C++ code with MATLAB Coder `codegen`
- Generate efficient MEX-functions with MATLAB Coder `codegen`
- Generate stand-alone C executables with MATLAB Compiler™ `mcc`
- Generate C/C++ libraries with MATLAB Compiler `mcc`
- Generate Excel® and .COM components with MATLAB Builder™ EX `mcc` options
- Generate Java components with MATLAB Builder JA `mcc` options
- Generate .NET components with MATLAB Builder NE `mcc` options

`genFunction(net,'path/name')` takes a neural network and file path and produces a standalone MATLAB function file `'name.m'`.

`genFunction(_____,'MatrixOnly','yes')` overrides the default cell/matrix notation and instead generates a function that uses only matrix arguments compatible with MATLAB Coder tools. For static networks the matrix columns are interpreted as independent samples. For dynamic networks the matrix columns are interpreted as a series of time steps. The default value is `'no'`.

`genFunction(_____,'ShowLinks','no')` disables the default behavior of displaying links to generated help and source code. The default is `'yes'`.

Here a static network is trained and its outputs calculated.

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

A MATLAB function with the same interface as the neural network object is generated and tested, and viewed.

```
genFunction(houseNet,'houseFcn');
y2 = houseFcn(x);
accuracy2 = max(abs(y-y2))
edit houseFcn
```

The new function can be compiled with the MATLAB Compiler tools (license required) to a shared/dynamically linked library with `mcc`.

```
mcc -W lib:libHouse -T link:lib houseFcn
```

Next, another version of the MATLAB function is generated which supports only matrix arguments (no cell arrays). This function is tested. Then it is used to generate a MEX-function with the MATLAB Coder tool codegen (license required) which is also tested.

```
genFunction(houseNet,'houseFcn','MatrixOnly','yes');
y3 = houseFcn(x);
accuracy3 = max(abs(y-y3))

x1Type = coder.typeof(double(0),[13 Inf]); % Coder type of input 1
codegen houseFcn.m -config:mex -o houseCodeGen -args {x1Type}
y4 = houseCodeGen(x);
accuracy4 = max(abs(y-y4))
```

Here, a dynamic network is trained and its outputs calculated.

```
[x,t] = maglev_dataset;
maglevNet = narxnet(1:2,1:2,10);
[X,Xi,Ai,T] = preparets(maglevNet,x,{},t);
maglevNet = train(maglevNet,X,T,Xi,Ai);
[y,xf,af] = maglevNet(X,Xi,Ai);
```

Next, a MATLAB function is generated and tested. The function is then used to create a shared/dynamically linked library with mcc.

```
genFunction(maglevNet,'maglevFcn');
[y2,xf,af] = maglevFcn(X,Xi,Ai);
accuracy2 = max(abs(cell2mat(y)-cell2mat(y2)))
mcc -W lib:libMaglev -T link:lib maglevFcn
```

Next, another version of the MATLAB function is generated which supports only matrix arguments (no cell arrays). This function is tested. Then it is used to generate a MEX-function with the MATLAB Coder tool codegen, and the result is also tested.

```
genFunction(maglevNet,'maglevFcn','MatrixOnly','yes');
x1 = cell2mat(X(1,:)); % Convert each input to matrix
x2 = cell2mat(X(2,:));
xi1 = cell2mat(Xi(1,:)); % Convert each input state to matrix
xi2 = cell2mat(Xi(2,:));
[y3,xf1,xf2] = maglevFcn(x1,x2,xi1,xi2);
accuracy3 = max(abs(cell2mat(y)-y3))

x1Type = coder.typeof(double(0),[1 Inf]); % Coder type of input 1
x2Type = coder.typeof(double(0),[1 Inf]); % Coder type of input 2
xi1Type = coder.typeof(double(0),[1 2]); % Coder type of input 1 states
xi2Type = coder.typeof(double(0),[1 2]); % Coder type of input 2 states
codegen maglevFcn.m -config:mex -o maglevNetCodeGen -args {x1Type x2Type xi1Type xi2Type}
[y4,xf1,xf2] = maglevNetCodeGen(x1,x2,xi1,xi2);
dynamic_codegen_accuracy = max(abs(cell2mat(y)-y4))
```

Enhanced Tools

The function genFunction is introduced with a new panel in the tools nftool, nctool, nprtool and ntstool.
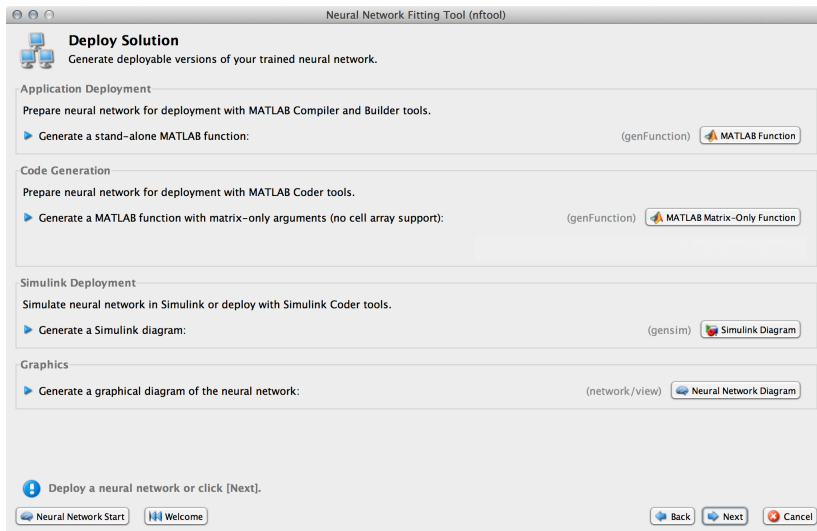
The advanced scripts generated on the Save Results panel of each of these tools includes an example of deploying networks with `genFunction`.
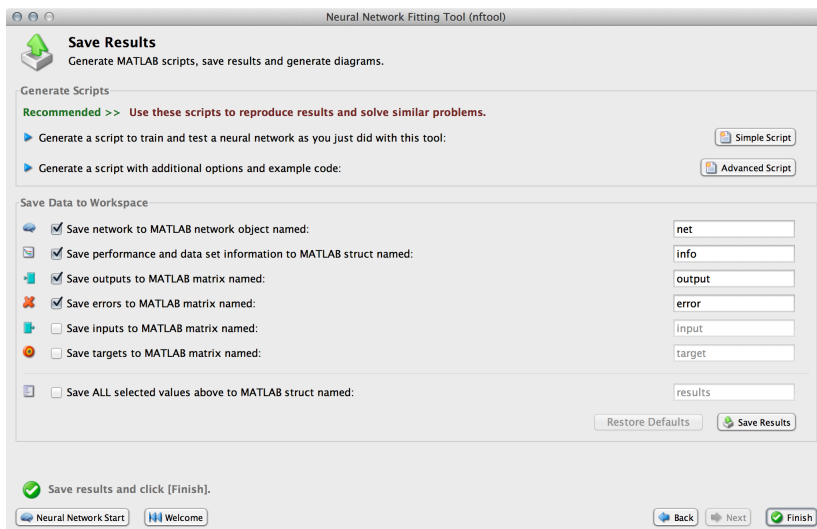
For more information, see Deploy Neural Network Functions.

Enhanced multi-timestep prediction for switching between open-loop and closed-loop modes with NARX and NAR neural networks

Dynamic networks with feedback, such as `narxnet` and `narnet` neural networks, can be transformed between open-loop and closed-loop modes with the functions `openloop` and `closeloop`. Closed-loop networks make multistep predictions. In other words, they continue to predict when external feedback is missing, by using internal feedback.

It can be useful to simulate a trained neural network up the present with all the known values of a time-series in open-loop mode, then switch to closed-loop mode to continue the simulation for as many predictions into the future as are desired. It is now much easier to do this.

Previously, `openloop` and `closeloop` transformed the neural network between those two modes.

```
net = openloop(net)
net = closeloop(net)
```

This is still the case. However, these functions now also support the transformation of input and layer delay state values between open- and closed-loop modes, making switching between closed-loop to open-loop multistep prediction easier.

```
[net,xi,ai] = openloop(net,xi,ai);
[net,xi,ai] = closeloop(net,xi,ai);
```

Here, a neural network is trained to model the magnetic levitation system in default open-loop mode.

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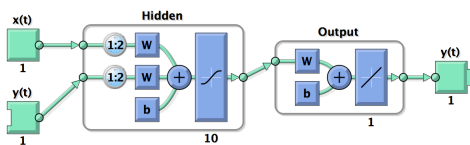

Then `closeloop` is used to convert the network to closed-loop form for simulation.

```
netc = closeloop(net);
[x,xi,ai,t] = preparets(netc,X,{},T);
y = netc(x,xi,ai);
view(netc)
```
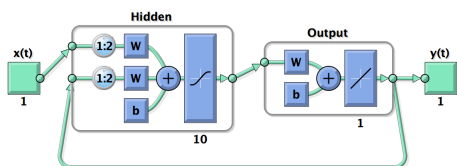

Now consider the case where you might have a record of the Maglev's behavior for 20 time steps, but then want to predict ahead for 20 more time steps beyond that.

Define the first 20 steps of inputs and targets, representing the 20 time steps where the output is known, as defined by the targets `t`. Then the next 20 time steps of the input are defined, but you use the network to predict the 20 outputs using each of its predictions feedback to help the network perform the next prediction.

```
x1 = x(1:20);
t1 = t(1:20);
x2 = x(21:40);
```

Then simulate the open-loop neural network on this data:

```
[x,xi,ai,t] = preparets(net,x1,{},t1);
[y1,xf,af] = net(x,xi,ai);
```

Now the final input and layer states returned by the network are converted to closed-loop form along with the network. The final input states `xf`, and layer states `af`, of the open-loop network become the initial input states `xi`, and layer states `ai`, of the closed-loop network.

```
[netc,xi,ai] = closeloop(net,xf,af);
```

Typically, `preparets` is used to define initial input and layer states. Since these have already been obtained from the end of the open-loop simulation, you do not need `preparets` to continue with the 20 step predictions of the closed-loop network.

```
[y2,xf,af] = netc(x2,xi,ai);
```

Note that `x2` can be set to different sequences of inputs to test different scenarios for however many time steps you would like to make predictions. For example, to predict the magnetic levitation system's behavior if 10 random inputs were used:

```
x2 = num2cell(rand(1,10));
[y2,xf,af] = netc(x2,xi,ai);
```

For more information, see Multistep Neural Network Prediction.

Cross-entropy performance measure for enhanced pattern recognition and classification accuracy

Networks created with `patternnet` now use the cross-entropy performance measure (`crossentropy`), which frequently produces classifiers with fewer percentage misclassifications than obtained using mean squared error.

See "Softmax transfer function in output layer gives consistent class probabilities for pattern recognition and classification" on page 19-6.

Softmax transfer function in output layer gives consistent class probabilities for pattern recognition and classification

`patternnet`, which you use to create a neural network suitable for learning classification problems, has been improved in two ways.

First, networks created with `patternnet` now use the cross-entropy performance measure (`crossentropy`), which frequently produces classifiers with fewer percentage misclassifications than obtained using mean squared error.

Second, `patternnet` returns networks that use the Soft Max transfer function (`softmax`) for the output layer instead of the `tansig` sigmoid transfer function. `softmax` results in output vectors normalized so they sum to 1.0, that can be interpreted as class probabilities. (`tansig` also produces outputs in the 0 to 1 range, but they do not sum to 1.0 and have to be manually normalized before being treated as consistent class probabilities.)

Here a `patternnet` with 10 neurons is created, its performance function and diagram are displayed.

```
net = patternnet(10);
net.performFcn

ans =
crossentropy
```

```
view(net)
```
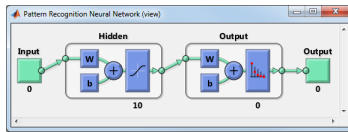

The output layer's transfer function is shown with the symbol for `softmax`.

Training the network takes advantage of the new `crossentropy` performance function. Here the network is trained to classify iris flowers. The cross-entropy performance algorithm is shown in the `nntraintool` algorithm section. Clicking the "Performance" plot button shows how the network's cross-entropy was minimized throughout the training session.

```
[x,t] = iris_dataset;
net = train(net,x,t);
```
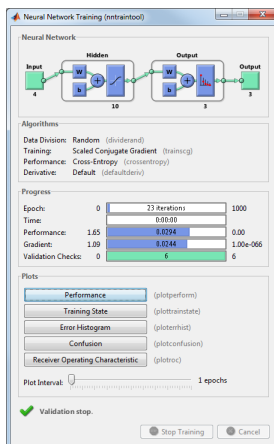

Simulating the network results in normalized output. Sample 150 is used to illustrate the normalization of class membership likelihoods:

```
y = net(x(:,150))

y   =

    0.0001
    0.0528
    0.9471

sum(y)

    1
```

The network output shows three membership probabilities with class three as by far the most likely. Each probability value is between 0 and 1, and together they sum to 1 indicating the 100% probability that the input `x(:,150)` falls into one of the three classes.

Compatibility Considerations

If a `patternnet` network is used to train on target data with only one row, the network's output transfer function will be changed to `tansig` and its outputs will continue to operate as they did

before the `softmax` enhancement. However, the `1-of-N` notation for targets is recommended even when there are only two classes. In that case the targets should have two rows, where each column has a `1` in the first or second row to indicate class membership.

If you prefer the older `patternnet` of mean squared error performance and a sigmoid output transfer function, you can specify this by setting those neural network object properties. Here is how that is done for a `patternnet` with 10 neurons.

```
net = patternnet(10);
net.layers{2}.transferFcn = 'tansig';
net.performFcn = 'mse';
```

Automated and periodic saving of intermediate results during neural network training

Intermediate results can be periodically saved during neural network training to a `.mat` file for recovery if the computer fails or the training process is killed. This helps protect the values of long training runs, which if interrupted, would otherwise need to be completely restarted.

This feature can be especially useful for long parallel training sessions that are more likely to be interrupted by computing resource failures and which you can stop only with a Ctrl+C break, because the `nntraintool` tool (with its **Stop** button) is not available during parallel training.

Checkpoint saves are enabled with an optional `'CheckpointFile'` training argument followed by the checkpoint file's name or path. If only a file name is specified, it is placed in the current folder by default. The file must have the `.mat` file extension, but if it is not specified it is automatically added. In this example, checkpoint saves are made to a file called `MyCheckpoint.mat` in the current folder.

```
[x,t] = house_dataset;
net = feedforwardnet(10);
net2 = train(net,x,t,'CheckpointFile','MyCheckpoint.mat');

22-Mar-2013 04:49:05 First Checkpoint #1: /WorkingDir/MyCheckpoint.mat
22-Mar-2013 04:49:06 Final Checkpoint #2: /WorkingDir/MyCheckpoint.mat
```

By default, checkpoint saves occur at most once every 60 seconds. For the short training example above this results in only two checkpoints, one at the beginning and one at the end of training.

The optional training argument `'CheckpointDelay'` changes the frequency of saves. For example, here the minimum checkpoint delay is set to 10 seconds, for a time-series problem where a neural network is trained to model a levitated magnet.

```
[x,t] = maglev_dataset;
net = narxnet(1:2,1:2,10);
[X,Xi,Ai,T] = preparets(net,x,{},t);
net2 = train(net,X,T,Xi,Ai,'CheckpointFile','MyCheckpoint.mat','CheckpointDelay',10);

22-Mar-2013 04:59:28 First Checkpoint #1: /WorkingDir/MyCheckpoint.mat
22-Mar-2013 04:59:38 Write Checkpoint #2: /WorkingDir/MyCheckpoint.mat
22-Mar-2013 04:59:48 Write Checkpoint #3: /WorkingDir/MyCheckpoint.mat
22-Mar-2013 04:59:58 Write Checkpoint #4: /WorkingDir/MyCheckpoint.mat
22-Mar-2013 05:00:08 Write Checkpoint #5: /WorkingDir/MyCheckpoint.mat
22-Mar-2013 05:00:09 Final Checkpoint #6: /WorkingDir/MyCheckpoint.mat
```

After a computer failure or training interruption, the checkpoint structure containing the best neural network obtained before the interruption and the training record can be reloaded. In this case the `stage` field value is `'Final'`, indicating the last save was at the final epoch, because training

completed successfully. The first epoch checkpoint is indicated by `'First'`, and intermediate checkpoints by `'Write'`.

```
load('MyCheckpoint.mat')

checkpoint =

     file: '/WorkingDir/MyCheckpoint.mat'
     time: [2013 3 22 5 0 9.0712]
   number: 6
    stage: 'Final'
      net: [1x1 network]
       tr: [1x1 struct]
```

Training can be resumed from the last checkpoint by reloading the dataset (if necessary), then calling `train` with the recovered network.

```
net = checkpoint.net;
[x,t] = maglev_dataset;
load('MyCheckpoint.mat');
[X,Xi,Ai,T] = preparets(net,x,{},t);
net2 = train(net,X,T,Xi,Ai,'CheckpointFile','MyCheckpoint.mat','CheckpointDelay',10);
```

For more information, see Automatically Save Checkpoints During Neural Network Training.

Simpler Notation for Networks with Single Inputs and Outputs

The majority of neural networks have a single input and single output. You can now refer to the input and output of such networks with the properties `net.input` and `net.output`, without the need for cell array indices.

Here a feed-forward neural network is created and its input and output properties examined.

```
net = feedforwardnet(10);
net.input
net.output
```

The `net.inputs{1}` notation for the input and `net.outputs{2}` notation for the second layer output continue to work. The cell array notation continues to be required for networks with multiple inputs and outputs.

For more information, see Neural Network Object Properties.

Neural Network Efficiency Properties Are Now Obsolete

The neural network property `net.efficiency` is no longer shown when a network object properties are displayed. The following line of code displays the properties of a feed-forward network.

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

Compatibility Considerations

The efficiency properties are still supported and do not yet generate warnings, so backward compatibility is maintained. However the recommended way to use memory reduction is no longer to set `net.efficiency.memoryReduction`. The recommended notation since R2012b is to use optional training arguments:

```
[x,t] = vinyl_dataset;
net = feedforwardnet(10);
net = train(net,x,t,'Reduction',10);
```

Memory reduction is a way to trade off training time for lower memory requirements when using Jacobian training such as `trainlm` and `trainbr`. The `MemoryReduction` value indicates how many passes must be made to simulate the network and calculate its gradients each epoch. The storage requirements go down as the memory reduction goes up, although not necessarily proportionally. The default `MemoryReduction` is 1, which indicates no memory reduction.

R2013a

Version: 8.0.1

Bug Fixes

R2012b

Version: 8.0

New Features

Bug Fixes

Compatibility Considerations

Speed and memory efficiency enhancements for neural network training and simulation

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);
```
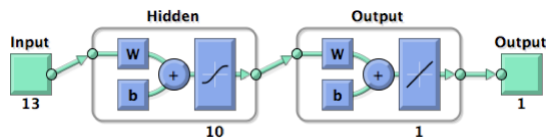

In Version 8.0, the above code does not need to be changed, but calculations now happen in compiled native MEX code.

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)
```
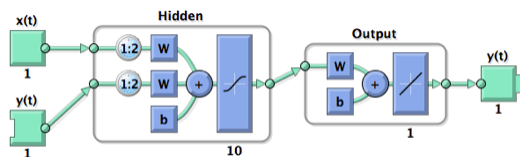

The following code measures training speed over 10 training sessions, with the training window disabled to avoid GUI timing interference.

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
```
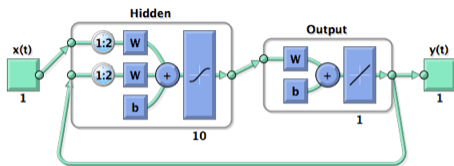

The following code trains the network in closed-loop mode:

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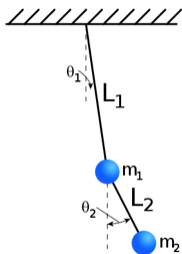
Because of the Version 8.0 MEX speedup, this only took a few hours, as opposed to the months it would have taken in Version 7.0.

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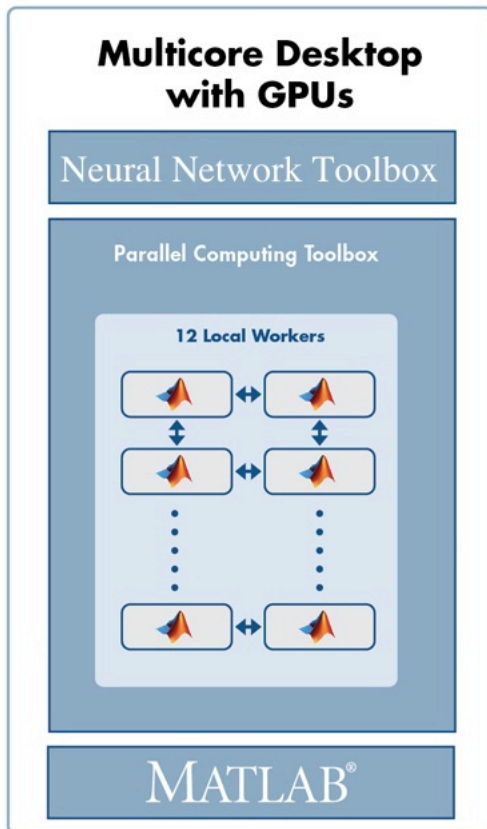
Note that, during training, the calculation of network outputs, performance, gradient, and Jacobian calculations are parallelized, while the main training code remains on one worker.

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')
```
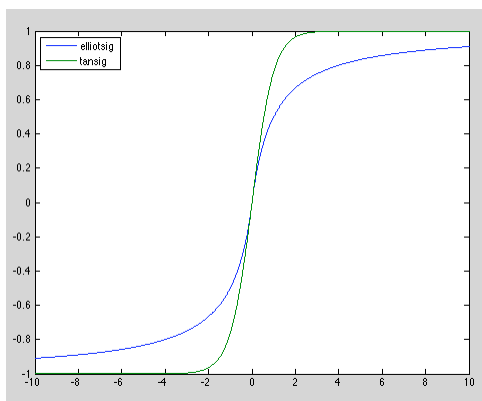

To set up a neural network to use the `elliotsig` transfer function, change each `tansig` layer's transfer function with its `transferFcn` property. For instance, here a network using `elliotsig` is created, viewed, trained, and simulated:

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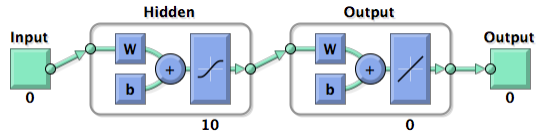


```
net.layers{1}.transferFcn = 'elliotsig';
view(net) % View ELLIOTSIG network
```
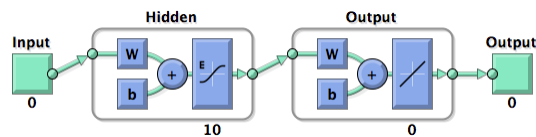


```
net = train(net,x,t);
y = net(x)
```

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.
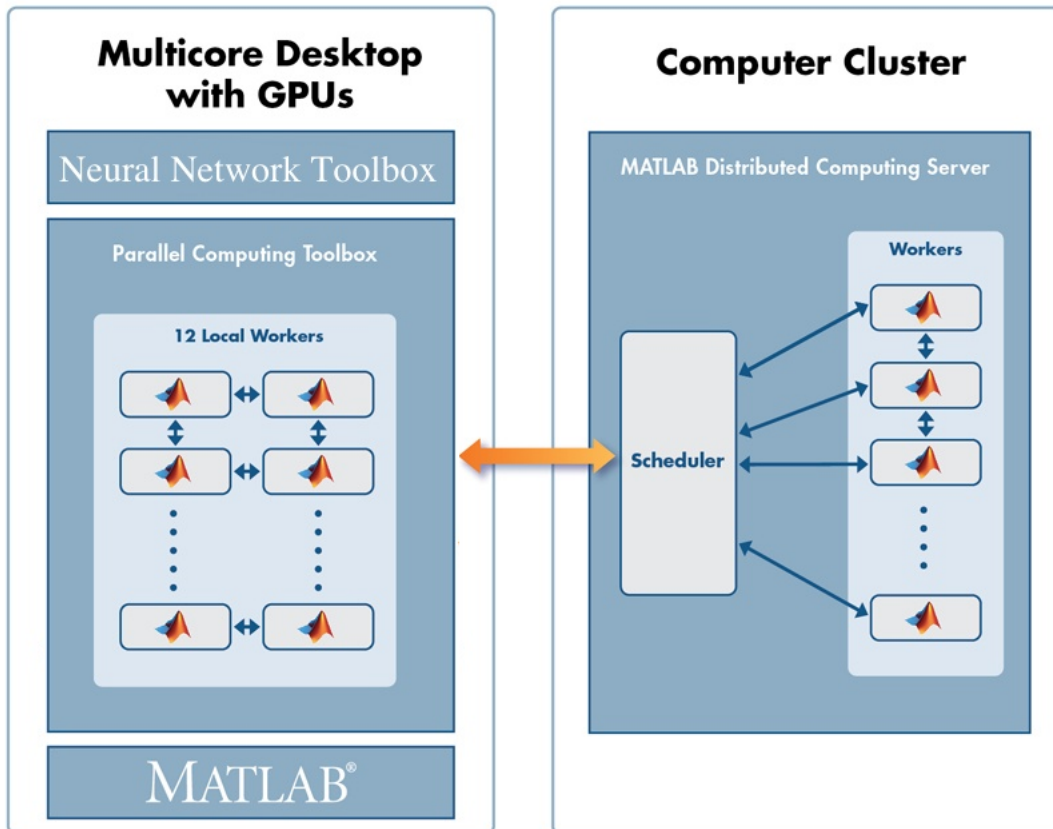
For more information, see Parallel and GPU Computing.

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 parallel pool of MATLAB workers.

For instance, if a parallel 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 you call `train`, the `'useParallel'` option is not needed, because `train` automatically trains in parallel when using Composite data.

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


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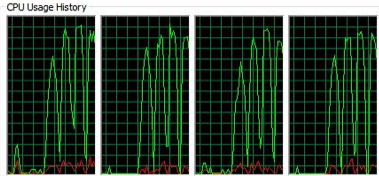
If you want workers 1 and 2 to use GPU devices 1 and 2, while workers 3 and 4 use CPUs, set up data for workers 1 and 2 using `nndata2gpu` inside an `spmd` clause.

```
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 supported 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.

Example 21.1. Example: View computing resources

```
[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

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

Bug Fixes

R2011b

Version: 7.0.2

Bug Fixes

R2011a

Version: 7.0.1

Bug Fixes

R2010b

Version: 7.0

New Features

Bug Fixes

Compatibility Considerations

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.
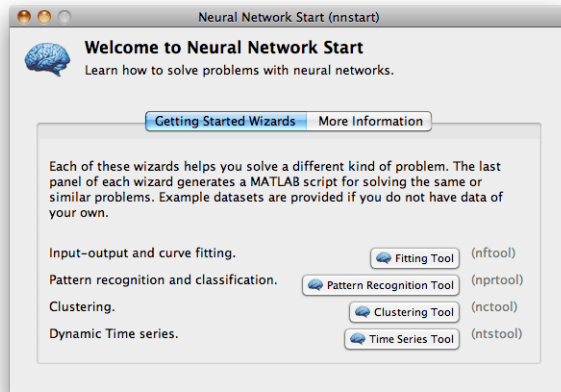
The second panel provides links to other toolbox starting points.
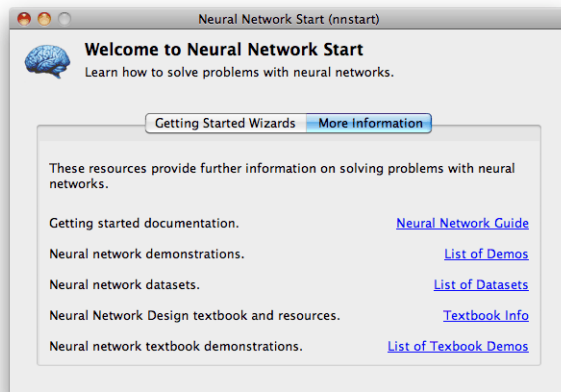
New Time Series GUI and Tools

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.
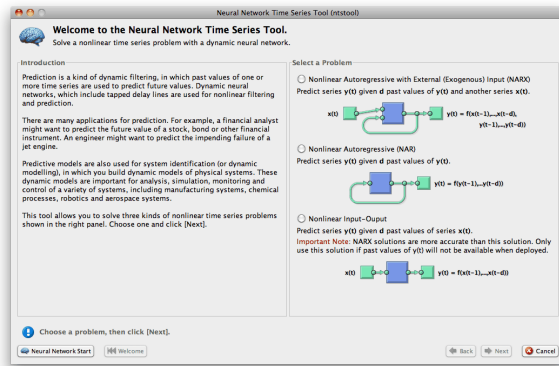
Network diagrams shown in the Neural Time Series Tool, Neural Training Tool, and with the `view(net)` command, have been improved to show tap delay lines in front of weights, the sizes of inputs, layers and outputs, and the time relationship of inputs and outputs. Open loop feedback outputs and inputs are indicated with matching tab and indents in their respective blocks.
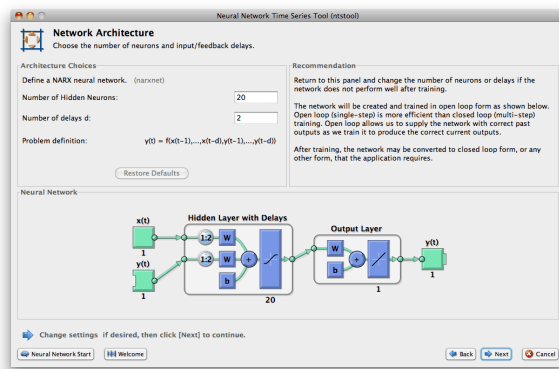
The Save Results panel of the Neural Network Time Series Tool allows you to generate both a Simple Script, which demonstrates how to get the same results as were obtained with the wizard, and an Advanced Script, which provides an introduction to more advanced techniques.

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 25-7.)

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

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

```
ploterrcorr(errors)
```
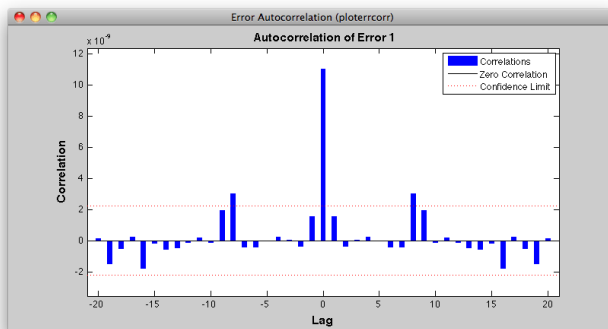

The input-to-error correlation can also be plotted for varying lags.

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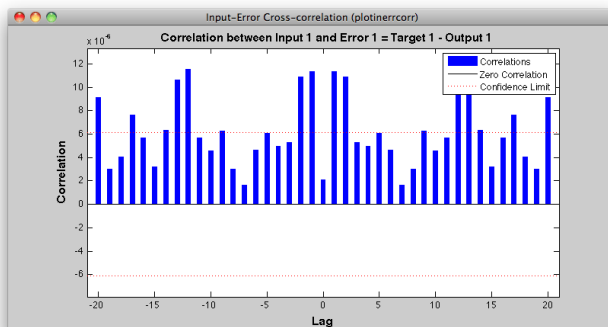

Simpler time series neural network creation is provided for NARX and time-delay networks, and a new function creates NAR networks. All the network diagrams shown here are generated with the command `view(net)`.

```
net = narxnet(inputDelays, feedbackDelays, hiddenSizes,
feedbackMode, trainingFcn
```

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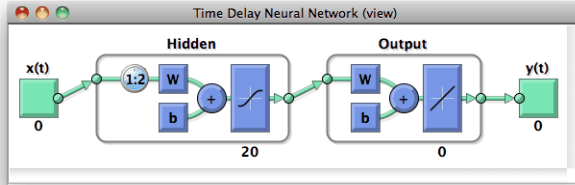

Several new data sets provide sample problems that can be solved with these networks. These data sets are also available within the `ntstool` GUI and the command line.

```
[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)
```
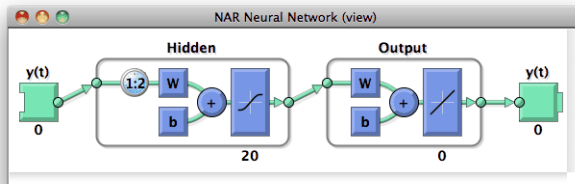

The total delay through a network can be adjusted with the two new functions `removedelay` and `adddelay`. Removing a delay from a NARX network which has a minimum input and feedback delay

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)
```
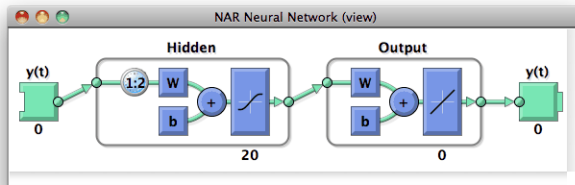

The new function `catsamples` allows you to combine multiple time series into a single neural network data variable. This is useful for creating input and target data from multiple input and target time series.

```
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

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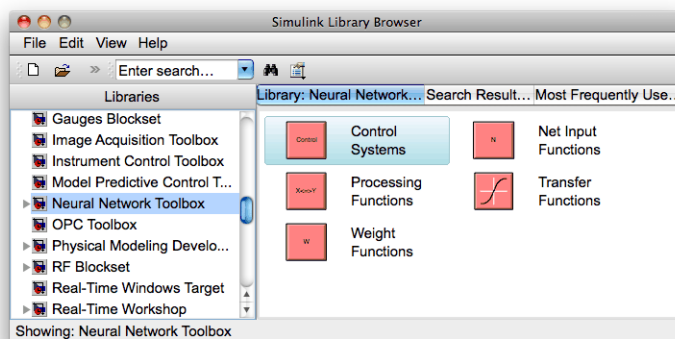
`gensim` has new options for generating neural network systems in Simulink.

```
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');
```
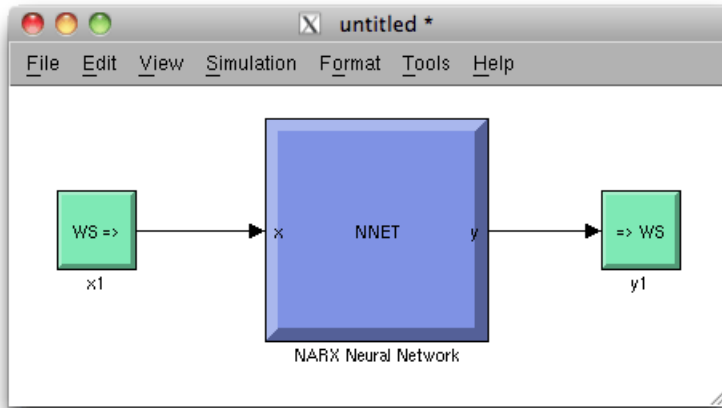

Simulink neural network blocks now allow initial conditions for input and layer delays to be set directly by double-clicking the neural network block. `setsiminit` and `getsiminit` provide command-line control for setting and getting input and layer delays for a neural network Simulink block.

```
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

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

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

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

Bug Fixes

R2009b

Version: 6.0.3

Bug Fixes

R2009a

Version: 6.0.2

Bug Fixes

R2008b

Version: 6.0.1

Bug Fixes

30

R2008a

Version: 6.0

New Features

Bug Fixes

Compatibility Considerations

New Training GUI with Animated Plotting Functions

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

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

The `newfit` function creates a fitting network that consists 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

Bug Fixes

Compatibility Considerations

Simplified Syntax for Network-Creation Functions

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

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

No New Features or Changes

R2006b

Version: 5.0.1

No New Features or Changes

R2006a

Version: 5.0

New Features

Compatibility Considerations

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

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

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

No New Features or Changes