分类和检测示例

创建简单的 CNN 网络以用于图像分类

openExample('nnet/TrainABasicConvolutionalNeuralNetworkForClassificationExample')

使用 GoogLeNet 对网络摄像头图像进行分类

openExample('nnet/ClassifyImagesFromWebcamUsingDeepLearningExample')

使用预训练 GoogLeNet 对新图像进行分类（迁移学习）

openExample('nnet/TransferLearningUsingGoogLeNetExample')

创建 FasterRCNN 网络并用于目标检测

openExample('vision/CreateFasterRCNNObjectDetectionNetworkExample')

openExample('deeplearning_shared/DeepLearningFasterRCNNObjectDetectionExample')

使用 Yolov3 用于目标检测

openExample('deeplearning_shared/ObjectDetectionUsingYOLOV3DeepLearningExample')

使用 SSD 用于目标检测

openExample('deeplearning_shared/ObjectDetectionUsingSSDDeepLearningExample')

Create Simple Deep Learning Network for Classification

This example shows how to create and train a simple convolutional neural network for deep learning classification. Convolutional neural networks are essential tools for deep learning, and are especially suited for image recognition.

The example demonstrates how to:

- Load and explore image data.
- Define the network architecture.
- Specify training options.
- Train the network.
- Predict the labels of new data and calculate the classification accuracy.

For an example showing how to interactively create and train a simple image classification network, see Create Simple Image Classification Network Using Deep Network Designer.

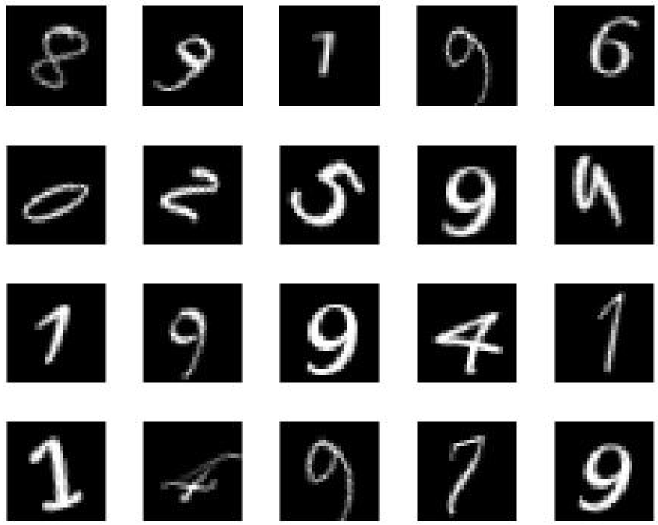
Load and Explore Image Data

Load the digit sample data as an image datastore. `imageDatastore` automatically labels the images based on folder names and stores the data as an `ImageDatastore` object. An image datastore enables you to store large image data, including data that does not fit in memory, and efficiently read batches of images during training of a convolutional neural network.

```
digitDatasetPath = fullfile(matlabroot,'toolbox','nnet','nndemos', ...
    'nndatasets','DigitDataset');
imds = imageDatastore(digitDatasetPath, ...
    'IncludeSubfolders',true,'LabelSource','foldernames');
```

Display some of the images in the datastore.

```
figure;
perm = randperm(10000,20);
for i = 1:20
    subplot(4,5,i);
    imshow(imds.Files{perm(i)});
end
```



Calculate the number of images in each category. `labelCount` is a table that contains the labels and the number of images having each label. The datastore contains 1000 images for each of the digits 0-9, for a total of 10000 images. You can specify the number of classes in the last fully connected layer of your network as the `OutputSize` argument.

```
labelCount = countEachLabel(imds)
```

labelCount = 10×2 table

	Label	Count
1	0	1000
2	1	1000
3	2	1000
4	3	1000
5	4	1000
6	5	1000
7	6	1000
8	7	1000
9	8	1000
10	9	1000

You must specify the size of the images in the input layer of the network. Check the size of the first image in `digitData`. Each image is 28-by-28-by-1 pixels.

```
img = readimage(imds,1);
size(img)
```

ans = 1×2

28 28

Specify Training and Validation Sets

Divide the data into training and validation data sets, so that each category in the training set contains 750 images, and the validation set contains the remaining images from each label. `splitEachLabel` splits the datastore `digitData` into two new datastores, `trainDigitData` and `valDigitData`.

```
numTrainFiles = 750;
[imdsTrain,imdsValidation] = splitEachLabel(imds,numTrainFiles,'randomize');
```

Define Network Architecture

Define the convolutional neural network architecture.

```
layers = [
    imageInputLayer([28 28 1])

    convolution2dLayer(3,8,'Padding','same')
    batchNormalizationLayer
    reluLayer

    maxPooling2dLayer(2,'Stride',2)

    convolution2dLayer(3,16,'Padding','same')
    batchNormalizationLayer
    reluLayer

    maxPooling2dLayer(2,'Stride',2)

    convolution2dLayer(3,32,'Padding','same')
    batchNormalizationLayer
    reluLayer

    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer];
```

Image Input Layer An imageInputLayer is where you specify the image size, which, in this case, is 28-by-28-by-1. These numbers correspond to the height, width, and the channel size. The digit data consists of grayscale images, so the channel size (color channel) is 1. For a color image, the channel size is 3, corresponding to the RGB values. You do not need to shuffle the data because `trainNetwork`, by default, shuffles the data at the beginning of training. `trainNetwork` can also automatically shuffle the data at the beginning of every epoch during training.

Convolutional Layer In the convolutional layer, the first argument is `filterSize`, which is the height and width of the filters the training function uses while scanning along the images. In this example, the number 3 indicates that the filter size is 3-by-3. You can specify different sizes for the height and width of the filter. The second argument is the number of filters, `numFilters`, which is the number of neurons that connect to the same region of the input. This parameter determines the number of feature maps. Use the `'Padding'` name-value pair to add padding to the input feature map. For a convolutional layer with a default stride of 1, `'same'` padding ensures that the spatial output size is the same as the input size. You can also define the stride and learning rates for this layer using name-value pair arguments of convolution2dLayer.

Batch Normalization Layer Batch normalization layers normalize the activations and gradients propagating through a network, making network training an easier optimization problem. 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. Use batchNormalizationLayer to create a batch normalization layer.

ReLU Layer The batch normalization layer is followed by a nonlinear activation function. The most common activation function is the rectified linear unit (ReLU). Use reluLayer to create a ReLU layer.

Max Pooling Layer Convolutional layers (with activation functions) are sometimes followed by a down-sampling operation that reduces the spatial size of the feature map and removes redundant spatial information. Down-sampling makes it possible to increase the number of filters in deeper convolutional layers without increasing the required amount of computation per layer. One way of down-sampling is using a max pooling, which you create using maxPooling2dLayer. The max pooling layer returns the maximum values of rectangular regions of inputs, specified by the first argument, `poolSize`. In this example, the size of the rectangular region is [2,2]. The `'Stride'` name-value pair argument specifies the step size that the training function takes as it scans along the input.

Fully Connected Layer The convolutional and down-sampling layers are followed by one or more fully connected layers. As its name suggests, a fully connected layer is a layer in which the neurons connect to all the neurons in the preceding layer. This layer combines all the features learned by the previous layers across the image to identify the larger patterns. The last fully connected layer combines the features to classify the images. Therefore, the `OutputSize` parameter in the last fully connected layer is equal to the number of classes in the target data. In this example, the output size is 10, corresponding to the 10 classes. Use fullyConnectedLayer to create a fully connected layer.

Softmax Layer The softmax activation function normalizes the output of the fully connected layer. The output of the softmax layer consists of positive numbers that sum to one, which can then be used as classification probabilities by the classification layer. Create a softmax layer using the softmaxLayer function after the last fully connected layer.

Classification Layer The final layer is the classification layer. This layer uses the probabilities returned by the softmax activation function for each input to assign the input to one of the mutually exclusive classes and compute the loss. To create a classification layer, use classificationLayer.

Specify Training Options

After defining the network structure, specify the training options. Train the network using stochastic gradient descent with momentum (SGDM) with an initial learning rate of 0.01. Set the maximum number of epochs to 4. An epoch is a full training cycle on the entire training data set. Monitor the network accuracy during training by specifying validation data and validation frequency. Shuffle the data every epoch. The software trains the network on the training data and calculates the accuracy on the validation data at regular intervals during training. The validation data is not used to update the network weights. Turn on the training progress plot, and turn off the command window output.

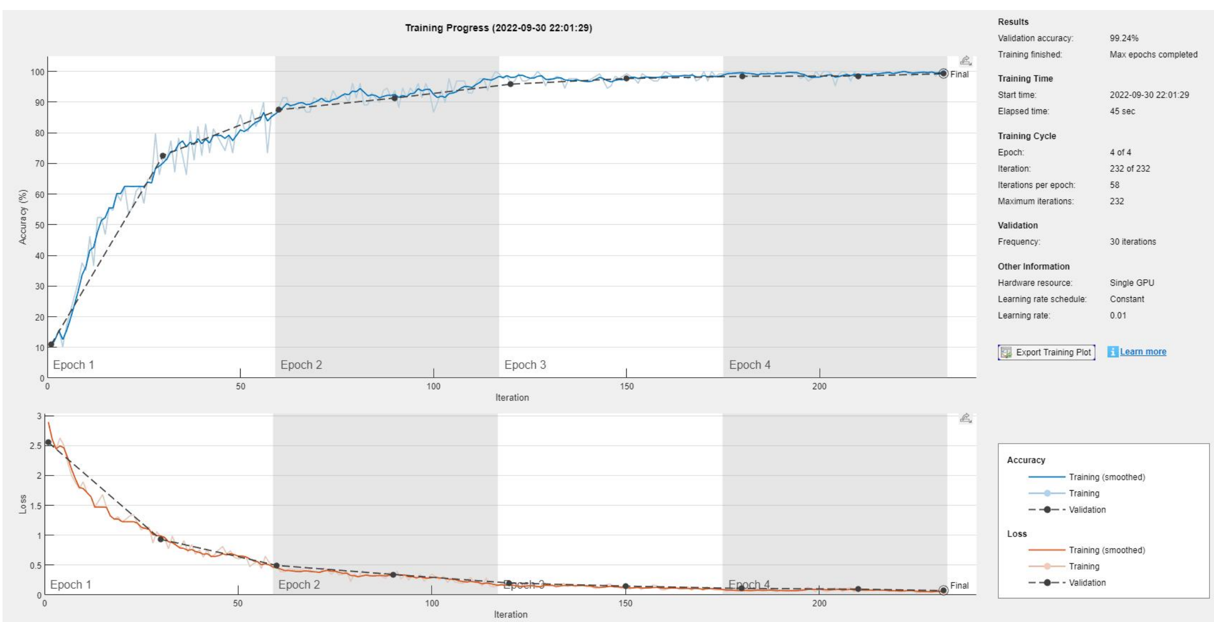
```
options = trainingOptions('sgdm', ...
    'InitialLearnRate',0.01, ...
    'MaxEpochs',4, ...
    'Shuffle','every-epoch', ...
    'ValidationData',imdsValidation, ...
    'ValidationFrequency',30, ...
    'Verbose',false, ...
    'Plots','training-progress');
```

Train Network Using Training Data

Train the network using the architecture defined by `layers`, the training data, and the training options. By default, `trainNetwork` uses a GPU if one is available, otherwise, it uses a CPU. Training on a GPU requires Parallel Computing Toolbox™ and a supported GPU device. For information on supported devices, see GPU Support by Release. You can also specify the execution environment by using the `'ExecutionEnvironment'` name-value pair argument of `trainingOptions`.

The training progress plot shows the mini-batch loss and accuracy and the validation loss and accuracy. For more information on the training progress plot, see Monitor Deep Learning Training Progress. The loss is the cross-entropy loss. The accuracy is the percentage of images that the network classifies correctly.

```
net = trainNetwork(imdsTrain,layers,options);
```



Classify Validation Images and Compute Accuracy

Predict the labels of the validation data using the trained network, and calculate the final validation accuracy. Accuracy is the fraction of labels that the network predicts correctly. In this case, more than 99% of the predicted labels match the true labels of the validation set.

```
YPred = classify(net,imdsValidation);
YValidation = imdsValidation.Labels;

accuracy = sum(YPred == YValidation)/numel(YValidation)
```

```
accuracy = 0.9924
```

Classify Webcam Images Using Deep Learning

This example shows how to classify images from a webcam in real time using the pretrained deep convolutional neural network GoogLeNet.

Use MATLAB®, a simple webcam, and a deep neural network to identify objects in your surroundings. This example uses GoogLeNet, a pretrained deep convolutional neural network (CNN or ConvNet) that has been trained on over a million images and can classify images into 1000 object categories (such as keyboard, coffee mug, pencil, and many animals). You can download GoogLeNet and use MATLAB to continuously process the camera images in real time.

GoogLeNet has learned rich feature representations for a wide range of images. It takes the image as input and provides a label for the object in the image and the probabilities for each of the object categories. You can experiment with objects in your surroundings to see how accurately GoogLeNet classifies images. To learn more about the network's object classification, you can show the scores for the top five classes in real time, instead of just the final class decision.

Load Camera and Pretrained Network

Connect to the camera and load a pretrained GoogLeNet network. You can use any pretrained network at this step. The example requires MATLAB Support Package for USB Webcams, and Deep Learning Toolbox™ Model *for GoogLeNet Network*. If you do not have the required support packages installed, then the software provides a download link.

```
camera = webcam;
net = googlenet;
```

If you want to run the example again, first run the command `clear camera` where `camera` is the connection to the webcam. Otherwise, you see an error because you cannot create another connection to the same webcam.

Classify Snapshot from Camera

To classify an image, you must resize it to the input size of the network. Get the first two elements of the `InputSize` property of the image input layer of the network. The image input layer is the first layer of the network.

```
inputSize = net.Layers(1).InputSize(1:2)
```

inputSize = 1×2

  224   224

Display the image from the camera with the predicted label and its probability. You must resize the image to the input size of the network before calling `classify`.

```
figure
im = snapshot(camera);
image(im)
im = imresize(im,inputSize);
```

```
[label,score] = classify(net,im);
title({char(label),num2str(max(score),2)});
```

Continuously Classify Images from Camera

To classify images from the camera continuously, include the previous steps inside a loop. Run the loop while the figure is open. To stop the live prediction, simply close the figure. Use `drawnow` at the end of each iteration to update the figure.

```
h = figure;

while ishandle(h)
    im = snapshot(camera);
    image(im)
    im = imresize(im,inputSize);
    [label,score] = classify(net,im);
    title({char(label), num2str(max(score),2)});
    drawnow
end
```

Display Top Predictions

The predicted classes can change rapidly. Therefore, it can be helpful to display the top predictions together. You can display the top five predictions and their probabilities by plotting the classes with the highest prediction scores.

Classify a snapshot from the camera. Display the image from the camera with the predicted label and its probability. Display a histogram of the probabilities of the top five predictions by using the `score` output of the `classify` function.

Create the figure window. First, resize the window to have twice the width, and create two subplots.

```
h = figure;
h.Position(3) = 2*h.Position(3);
ax1 = subplot(1,2,1);
ax2 = subplot(1,2,2);
```

In the left subplot, display the image and classification together.

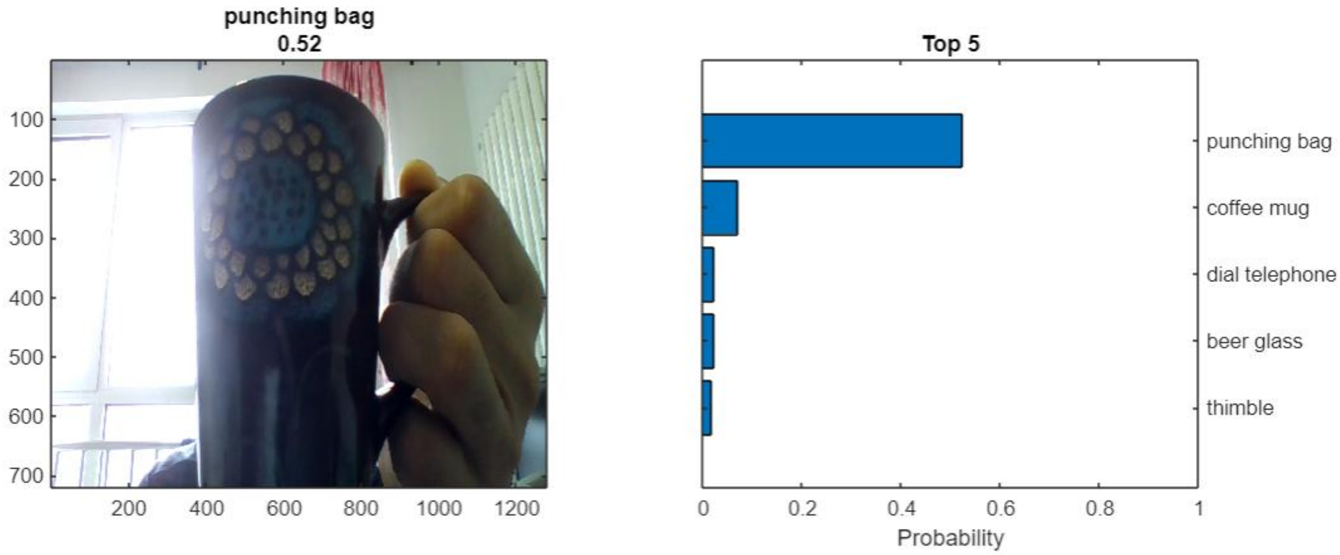
```
im = snapshot(camera);
image(ax1,im)
im = imresize(im,inputSize);
[label,score] = classify(net,im);
title(ax1,{char(label),num2str(max(score),2)});
```

Select the top five predictions by selecting the classes with the highest scores.

```
[~,idx] = sort(score,'descend');
idx = idx(5:-1:1);
classes = net.Layers(end).Classes;
classNamesTop = string(classes(idx));
scoreTop = score(idx);
```

Display the top five predictions as a histogram.

```
barh(ax2,scoreTop)
xlim(ax2,[0 1])
title(ax2,'Top 5')
xlabel(ax2,'Probability')
yticklabels(ax2,classNamesTop)
ax2.YAxisLocation = 'right';
```



Continuously Classify Images and Display Top Predictions

To classify images from the camera continuously and display the top predictions, include the previous steps inside a loop. Run the loop while the figure is open. To stop the live prediction, simply close the figure. Use `drawnow` at the end of each iteration to update the figure.

Create the figure window. First resize the window, to have twice the width, and create two subplots. To prevent the axes from resizing, set the `PositionConstraint` property to `'innerposition'`.

```
h = figure;
h.Position(3) = 2*h.Position(3);
ax1 = subplot(1,2,1);
ax2 = subplot(1,2,2);
ax2.PositionConstraint = 'innerposition';
```

Continuously display and classify images together with a histogram of the top five predictions.

```
while ishandle(h)
    % Display and classify the image
    im = snapshot(camera);
    image(ax1,im)
    im = imresize(im,inputSize);
    [label,score] = classify(net,im);
    title(ax1,{char(label),num2str(max(score),2)});

    % Select the top five predictions
    [~,idx] = sort(score,'descend');
    idx = idx(5:-1:1);
    scoreTop = score(idx);
    classNamesTop = string(class(idx));
```

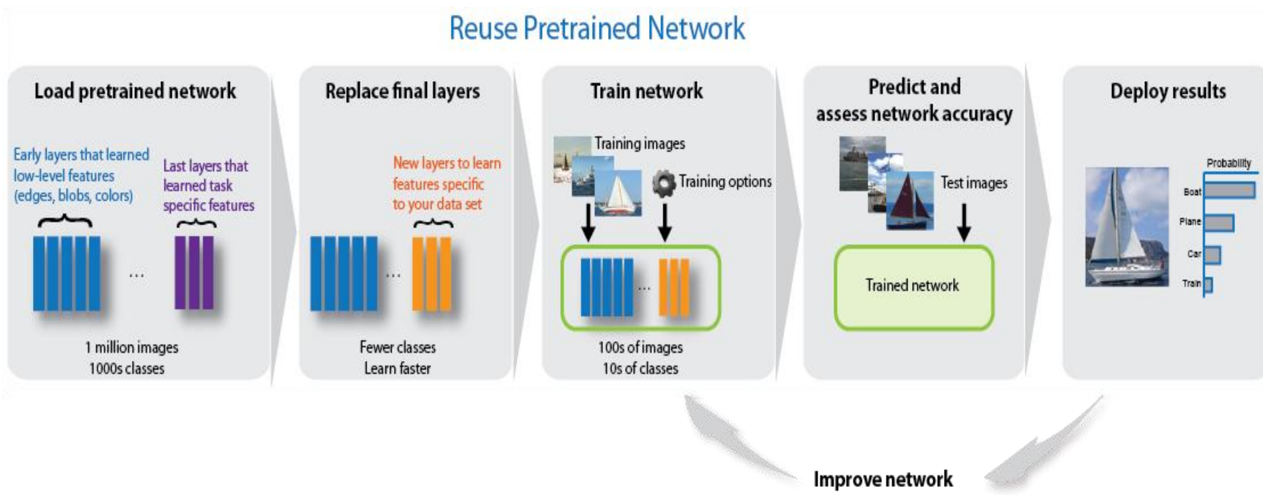
```
    % Plot the histogram
    barh(ax2,scoreTop)
    title(ax2,'Top 5')
    xlabel(ax2,'Probability')
    xlim(ax2,[0 1])
    yticklabels(ax2,classNamesTop)
    ax2.YAxisLocation = 'right';

    drawnow
end
```

Train Deep Learning Network to Classify New Images

This example shows how to use transfer learning to retrain a convolutional neural network to classify a new set of images.

Pretrained image classification networks have been trained on over a million images and can classify images into 1000 object categories, such as keyboard, coffee mug, pencil, and many animals. The networks have learned rich feature representations for a wide range of images. The network takes an image as input, and then outputs a label for the object in the image together with the probabilities for each of the object categories.

Transfer learning is commonly used in deep learning applications. You can take a pretrained network and use it as a starting point to learn a new task. Fine-tuning a network with transfer learning is usually much faster and easier than training a network from scratch with randomly initialized weights. You can quickly transfer learned features to a new task using a smaller number of training images.



Load Data

Unzip and load the new images as an image datastore. This very small data set contains only 75 images. Divide the data into training and validation data sets. Use 70% of the images for training and 30% for validation.

```
unzip('MerchData.zip');
imds = imageDatastore('MerchData', ...
    'IncludeSubfolders',true, ...
    'LabelSource','foldernames');
[imdsTrain,imdsValidation] = splitEachLabel(imds,0.7);
```

Load Pretrained Network

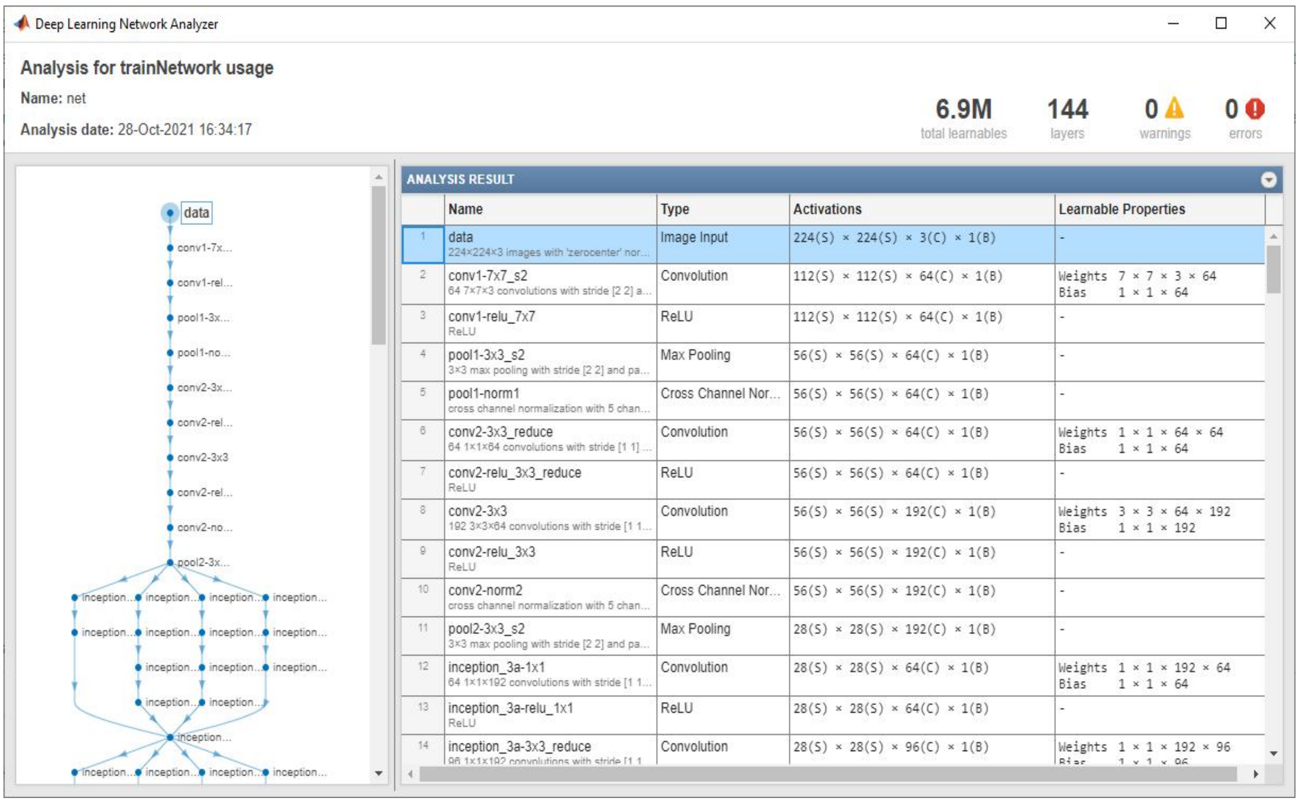
Load a pretrained GoogLeNet network. If the Deep Learning Toolbox™ Model *for GoogLeNet Network* support package is not installed, then the software provides a download link.

To try a different pretrained network, open this example in MATLAB® and select a different network. For example, you can try `squeezenet`, a network that is even faster than `googlenet`. You can run this example with other pretrained networks. For a list of all available networks, see [Load Pretrained Networks](#).

```
net = googlenet;
```

Use `analyzeNetwork` to display an interactive visualization of the network architecture and detailed information about the network layers.

```
analyzeNetwork(net)
```



The first element of the `Layers` property of the network is the image input layer. For a GoogLeNet network, this layer requires input images of size 224-by-224-by-3, where 3 is the number of color channels. Other networks can require input images with different sizes. For example, the Xception network requires images of size 299-by-299-by-3.

```
net.Layers(1)
inputSize = net.Layers(1).InputSize;
```

Replace Final Layers

The convolutional layers of the network extract image features that the last learnable layer and the final classification layer use to classify the input image. These two layers, `'loss3-classifier'` and `'output'` in GoogLeNet, contain information on how to combine the features that the network extracts into class probabilities, a loss value, and predicted labels. To retrain a pretrained network to classify new images, replace these two layers with new layers adapted to the new data set.

Convert the trained network to a layer graph.

```
lgraph = layerGraph(net);
```

Find the names of the two layers to replace. You can do this manually or you can use the supporting function findLayersToReplace to find these layers automatically.

```
[learnableLayer,classLayer] = findLayersToReplace(lgraph);
[learnableLayer,classLayer]
```

In most networks, the last layer with learnable weights is a fully connected layer. Replace this fully connected layer with a new fully connected layer with the number of outputs equal to the number of classes in the new data set (5, in this example). In some networks, such as SqueezeNet, the last learnable layer is a 1-by-1 convolutional layer instead. In this case, replace the convolutional layer with a new convolutional layer with the number of filters equal to the number of classes. To learn faster in the new layer than in the transferred layers, increase the learning rate factors of the layer.

```
numClasses = numel(categories(imdsTrain.Labels));

if isa(learnableLayer,'nnet.cnn.layer.FullyConnectedLayer')
    newLearnableLayer = fullyConnectedLayer(numClasses, ...
        'Name','new_fc', ...
        'WeightLearnRateFactor',10, ...
        'BiasLearnRateFactor',10);

elseif isa(learnableLayer,'nnet.cnn.layer.Convolution2DLayer')
    newLearnableLayer = convolution2dLayer(1,numClasses, ...
        'Name','new_conv', ...
        'WeightLearnRateFactor',10, ...
        'BiasLearnRateFactor',10);
end

lgraph = replaceLayer(lgraph,learnableLayer.Name,newLearnableLayer);
```

The classification layer specifies the output classes of the network. Replace the classification layer with a new one without class labels. `trainNetwork` automatically sets the output classes of the layer at training time.

```
newClassLayer = classificationLayer('Name','new_classoutput');
lgraph = replaceLayer(lgraph,classLayer.Name,newClassLayer);
```

To check that the new layers are connected correctly, plot the new layer graph and zoom in on the last layers of the network.

```
figure('Units','normalized','Position',[0.3 0.3 0.4 0.4]);
```

```
plot(lgraph)
ylim([0,10])
```

Freeze Initial Layers

The network is now ready to be retrained on the new set of images. Optionally, you can "freeze" the weights of earlier layers in the network by setting the learning rates in those layers to zero. During training, `trainNetwork` does not update the parameters of the frozen layers. Because the gradients of the frozen layers do not need to be computed, freezing the weights of many initial layers can significantly speed up network training. If the new data set is small, then freezing earlier network layers can also prevent those layers from overfitting to the new data set.

Extract the layers and connections of the layer graph and select which layers to freeze. In GoogLeNet, the first 10 layers make out the initial 'stem' of the network. Use the supporting function freezeWeights to set the learning rates to zero in the first 10 layers. Use the supporting function createLgraphUsingConnections to reconnect all the layers in the original order. The new layer graph contains the same layers, but with the learning rates of the earlier layers set to zero.

```
layers = lgraph.Layers;
connections = lgraph.Connections;

layers(1:10) = freezeWeights(layers(1:10));
lgraph = createLgraphUsingConnections(layers,connections);
```

Train Network

The network requires input images of size 224-by-224-by-3, but the images in the image datastore have different sizes. Use an augmented image datastore to automatically resize the training images. Specify additional augmentation operations to perform on the training images: randomly flip the training images along the vertical axis and randomly translate them up to 30 pixels and scale them up to 10% horizontally and vertically. Data augmentation helps prevent the network from overfitting and memorizing the exact details of the training images.

```
pixelRange = [-30 30];
scaleRange = [0.9 1.1];
imageAugmenter = imageDataAugmenter( ...
    'RandXReflection',true, ...
    'RandXTranslation',pixelRange, ...
    'RandYTranslation',pixelRange, ...
    'RandXScale',scaleRange, ...
    'RandYScale',scaleRange);
augimdsTrain = augmentedImageDatastore(inputSize(1:2),imdsTrain, ...
    'DataAugmentation',imageAugmenter);
```

To automatically resize the validation images without performing further data augmentation, use an augmented image datastore without specifying any additional preprocessing operations.

```
augimdsValidation = augmentedImageDatastore(inputSize(1:2),imdsValidation);
```

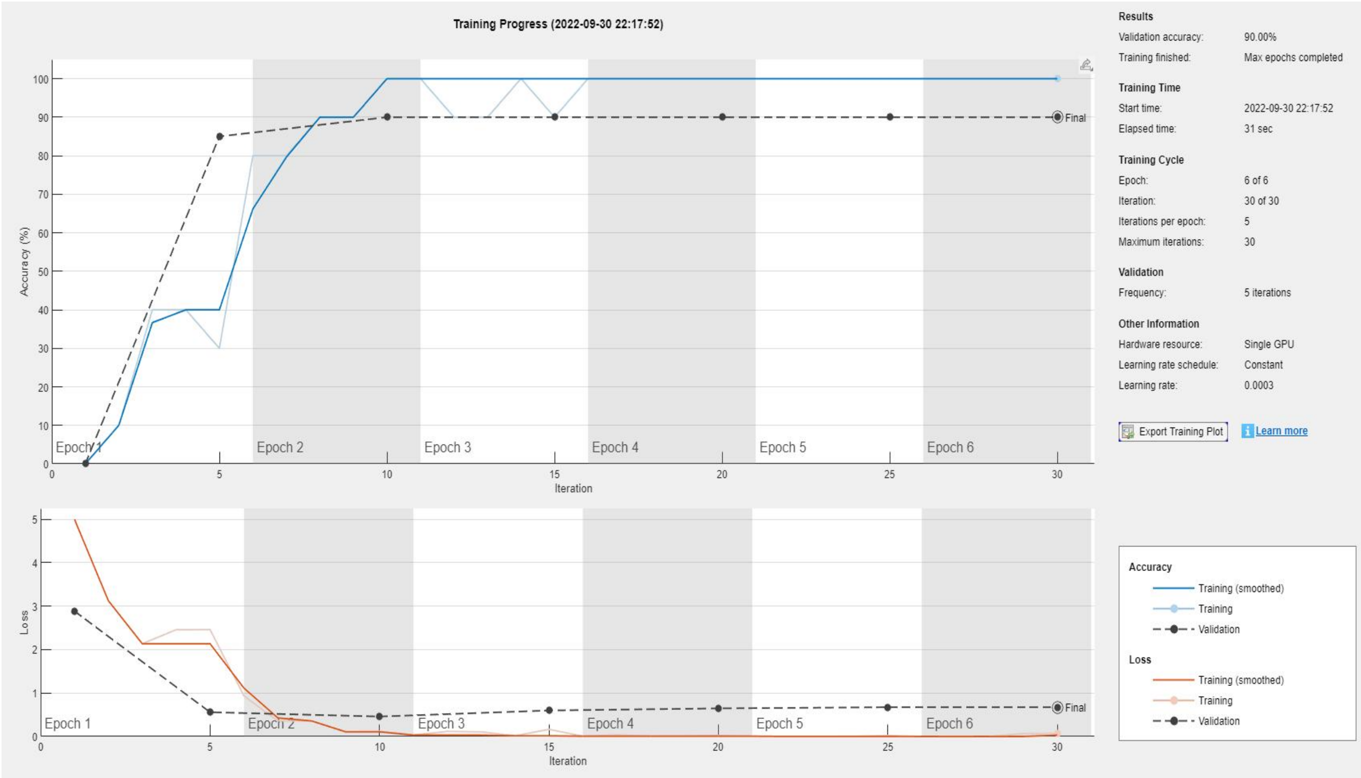
Specify the training options. Set `InitialLearnRate` to a small value to slow down learning in the transferred layers that are not already frozen. In the previous step, you increased the learning rate factors for the last learnable layer to speed up learning in the new final layers. This combination of learning rate settings results in fast learning in the new layers, slower learning in the middle layers, and no learning in the earlier, frozen layers.

Specify the number of epochs to train for. When performing transfer learning, you do not need to train for as many epochs. An epoch is a full training cycle on the entire training data set. Specify the mini-batch size and validation data. Compute the validation accuracy once per epoch.

```
miniBatchSize = 10;
valFrequency = floor(numel(augimdsTrain.Files)/miniBatchSize);
options = trainingOptions('sgdm', ...
    'MiniBatchSize',miniBatchSize, ...
    'MaxEpochs',6, ...
    'InitialLearnRate',3e-4, ...
    'Shuffle','every-epoch', ...
    'ValidationData',augimdsValidation, ...
    'ValidationFrequency',valFrequency, ...
    'Verbose',false, ...
    'Plots','training-progress');
```

Train the network using the training data. By default, `trainNetwork` uses a GPU if one is available. This requires Parallel Computing Toolbox™ and a supported GPU device. For information on supported devices, see GPU Support by Release. Otherwise, `trainNetwork` uses a CPU. You can also specify the execution environment by using the `'ExecutionEnvironment'` name-value pair argument of `trainingOptions`. Because the data set is so small, training is fast.

```
net = trainNetwork(augimdsTrain,lgraph,options);
```



Classify Validation Images

Classify the validation images using the fine-tuned network, and calculate the classification accuracy.

```
[YPred,probs] = classify(net,augimdsValidation);
accuracy = mean(YPred == imdsValidation.Labels)
```

```
accuracy = 0.9000
```

Display four sample validation images with predicted labels and the predicted probabilities of the images having those labels.

```
idx = randperm(numel(imdsValidation.Files),4);
figure
for i = 1:4
    subplot(2,2,i)
    I = readimage(imdsValidation,idx(i));
    imshow(I)
    label = YPred(idx(i));
    title(string(label) + ", " + num2str(100*max(probs(idx(i),:)),3) + "%");
end
```

MathWorks Screwdriver, 99.9%

MathWorks Screwdriver, 88.5%

MathWorks Cap, 100%

MathWorks Torch, 99.6%

Create Faster R-CNN Object Detection Network

This example builds upon the Create Fast R-CNN Object Detection Network example above. It transforms a pretrained ResNet-50 network into a Faster R-CNN object detection network by adding an ROI pooling layer, a bounding box regression layer, and a region proposal network (RPN). The Faster R-CNN network can then be trained using `trainFasterRCNNObjectDetector`.

Create Fast R-CNN Network

Start by creating Fast R-CNN, which forms the basis of Faster R-CNN. The Create Fast R-CNN Object Detection Network example explains this section of code in detail.

```
% Load a pretrained ResNet-50.
net = resnet50;
lgraph = layerGraph(net);

% Remove the last 3 layers.
layersToRemove = {
    'fc1000'
    'fc1000_softmax'
    'ClassificationLayer_fc1000'
    };
lgraph = removeLayers(lgraph, layersToRemove);

% Specify the number of classes the network should classify.
numClasses = 2;
numClassesPlusBackground = numClasses + 1;

% Define new classification layers.
newLayers = [
    fullyConnectedLayer(numClassesPlusBackground, 'Name', 'rcnnFC')
    softmaxLayer('Name', 'rcnnSoftmax')
    classificationLayer('Name', 'rcnnClassification')
    ];

% Add new object classification layers.
lgraph = addLayers(lgraph, newLayers);

% Connect the new layers to the network.
lgraph = connectLayers(lgraph, 'avg_pool', 'rcnnFC');

% Define the number of outputs of the fully connected layer.
numOutputs = 4 * numClasses;

% Create the box regression layers.
boxRegressionLayers = [
    fullyConnectedLayer(numOutputs,'Name','rcnnBoxFC')
```

```
    rcnnBoxRegressionLayer('Name','rcnnBoxDeltas')
    ];

% Add the layers to the network.
lgraph = addLayers(lgraph, boxRegressionLayers);

% Connect the regression layers to the layer named 'avg_pool'.
lgraph = connectLayers(lgraph,'avg_pool','rcnnBoxFC');

% Select a feature extraction layer.
featureExtractionLayer = 'activation_40_relu';

% Disconnect the layers attached to the selected feature extraction layer.
lgraph = disconnectLayers(lgraph, featureExtractionLayer,'res5a_branch2a');
lgraph = disconnectLayers(lgraph, featureExtractionLayer,'res5a_branch1');

% Add ROI max pooling layer.
outputSize = [14 14];
roiPool = roiMaxPooling2dLayer(outputSize,'Name','roiPool');
lgraph = addLayers(lgraph, roiPool);

% Connect feature extraction layer to ROI max pooling layer.
lgraph = connectLayers(lgraph, featureExtractionLayer,'roiPool/in');

% Connect the output of ROI max pool to the disconnected layers from above.
lgraph = connectLayers(lgraph, 'roiPool','res5a_branch2a');
lgraph = connectLayers(lgraph, 'roiPool','res5a_branch1');
```

Add Region Proposal Network (RPN)

Faster R-CNN uses a region proposal network (RPN) to generate region proposals. An RPN produces region proposals by predicting the class, “object” or “background”, and box offsets for a set of predefined bounding box templates known as "anchor boxes". Anchor boxes are specified by providing their size, which is typically determined based on a priori knowledge of the scale and aspect ratio of objects in the training dataset.

Learn more about [Anchor Box Basics](#).

Define the anchor boxes and create a `regionProposalLayer`.

```
% Define anchor boxes.
anchorBoxes = [
    16 16
    32 16
    16 32
    ];
```

```
% Create the region proposal layer.
proposalLayer = regionProposalLayer(anchorBoxes,'Name','regionProposal');

lgraph = addLayers(lgraph, proposalLayer);
```

Add the convolution layers for RPN and connect it to the feature extraction layer selected above.

```
% Number of anchor boxes.
numAnchors = size(anchorBoxes,1);

% Number of feature maps in coming out of the feature extraction layer.
numFilters = 1024;

rpnLayers = [
    convolution2dLayer(3, numFilters,'padding',[1 1],'Name','rpnConv3x3')
    reluLayer('Name','rpnRelu')
    ];

lgraph = addLayers(lgraph, rpnLayers);

% Connect to RPN to feature extraction layer.
lgraph = connectLayers(lgraph, featureExtractionLayer, 'rpnConv3x3');
```

Add the RPN classification output layers. The classification layer classifies each anchor as "object" or "background".

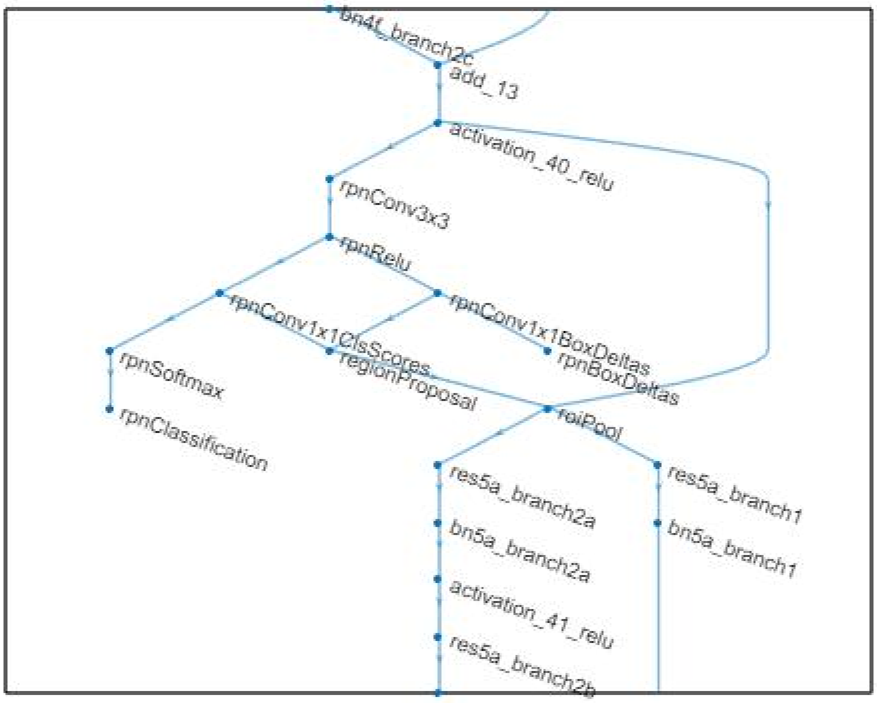
```
% Add RPN classification layers.
rpnClsLayers = [
    convolution2dLayer(1, numAnchors*2,'Name', 'rpnConv1x1ClsScores')
    rpnSoftmaxLayer('Name', 'rpnSoftmax')
    rpnClassificationLayer('Name','rpnClassification')
    ];
lgraph = addLayers(lgraph, rpnClsLayers);

% Connect the classification layers to the RPN network.
lgraph = connectLayers(lgraph, 'rpnRelu', 'rpnConv1x1ClsScores');
```

Add the RPN regression output layers. The regression layer predicts 4 box offsets for each anchor box.

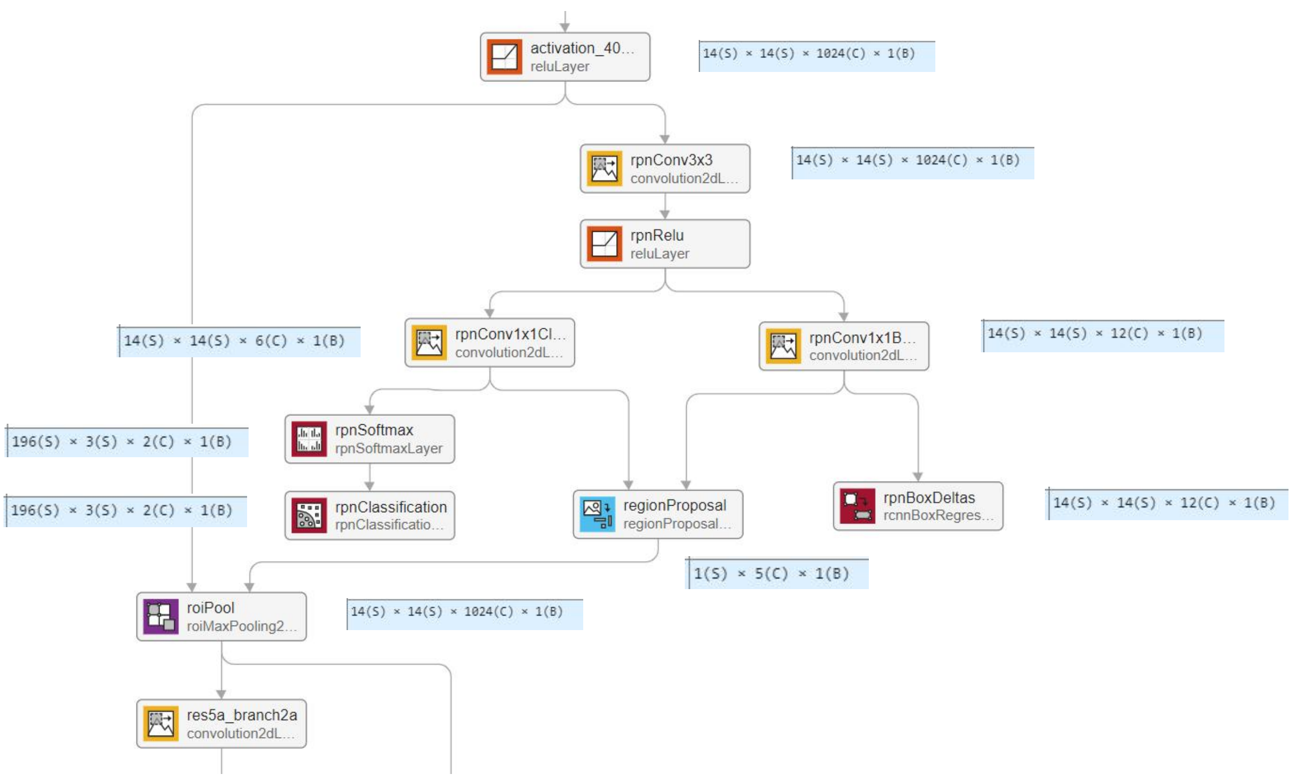
```
% Add RPN regression layers.
rpnRegLayers = [
    convolution2dLayer(1, numAnchors*4, 'Name', 'rpnConv1x1BoxDeltas')
    rcnnBoxRegressionLayer('Name', 'rpnBoxDeltas');
    ];

lgraph = addLayers(lgraph, rpnRegLayers);

% Connect the regression layers to the RPN network.
lgraph = connectLayers(lgraph, 'rpnRelu', 'rpnConv1x1BoxDeltas');
```

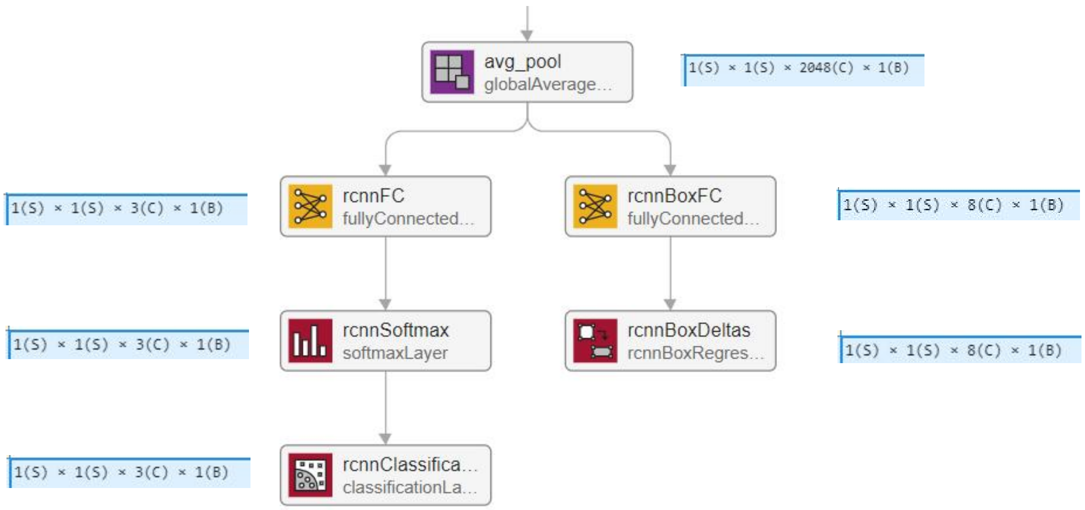
Finally, connect the classification and regression feature maps to the region proposal layer inputs, and the ROI pooling layer to the region proposal layer output.

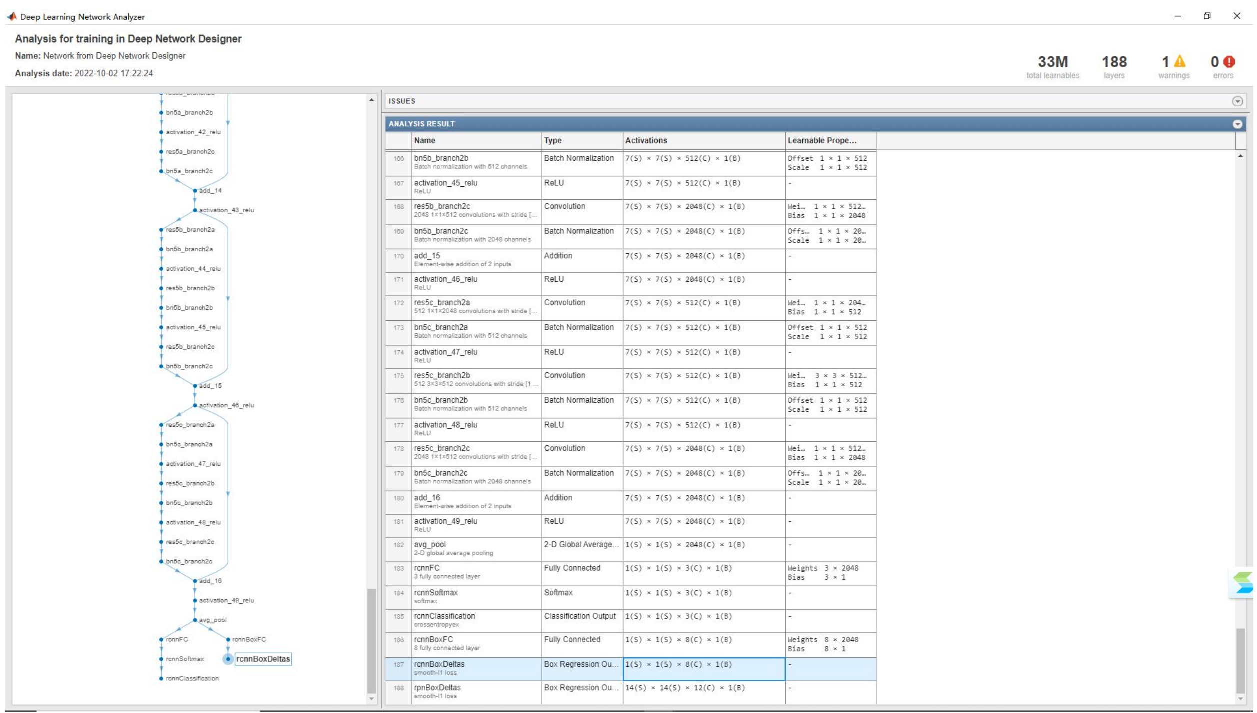
```
% Connect region proposal network.
lgraph = connectLayers(lgraph, 'rpnConv1x1ClsScores', 'regionProposal/scores');
lgraph = connectLayers(lgraph, 'rpnConv1x1BoxDeltas',
'regionProposal/boxDeltas');

% Connect region proposal layer to roi pooling.
lgraph = connectLayers(lgraph, 'regionProposal', 'roiPool/roi');

% Show the network after adding the RPN layers.
figure
plot(lgraph)
ylim([30 42])
```

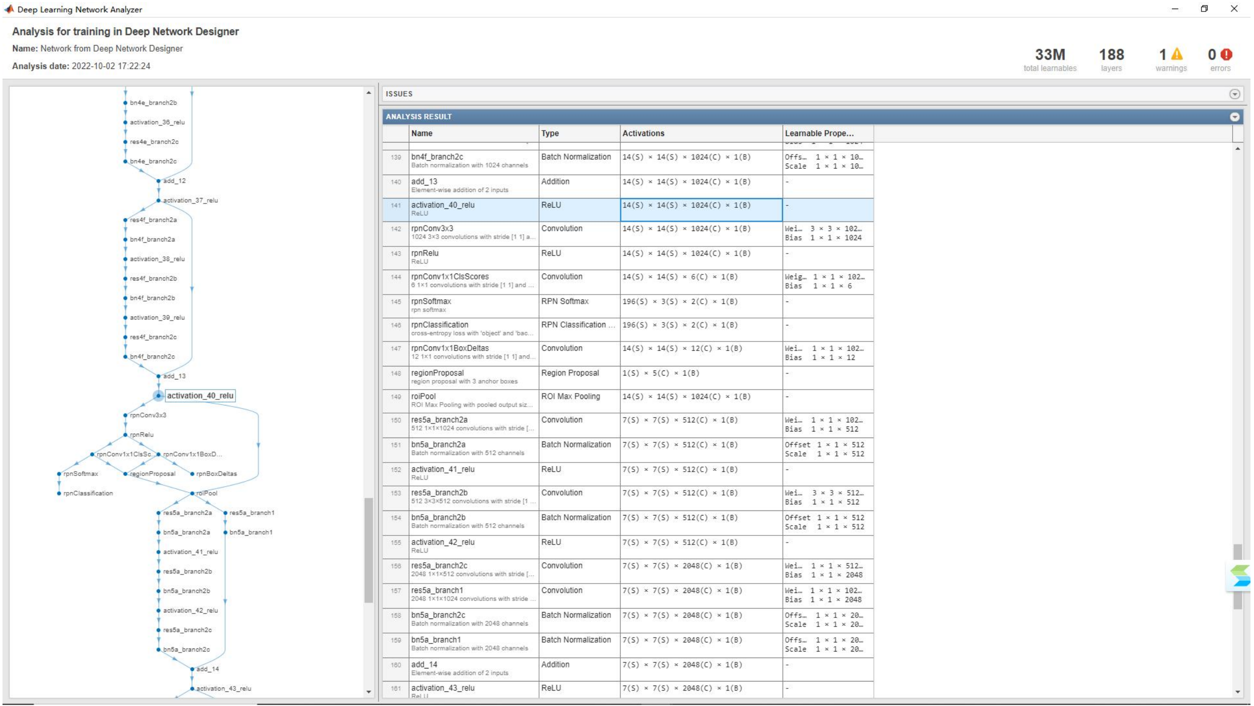


The network is ready to be trained using `trainFasterRCNNObjectDetector`.



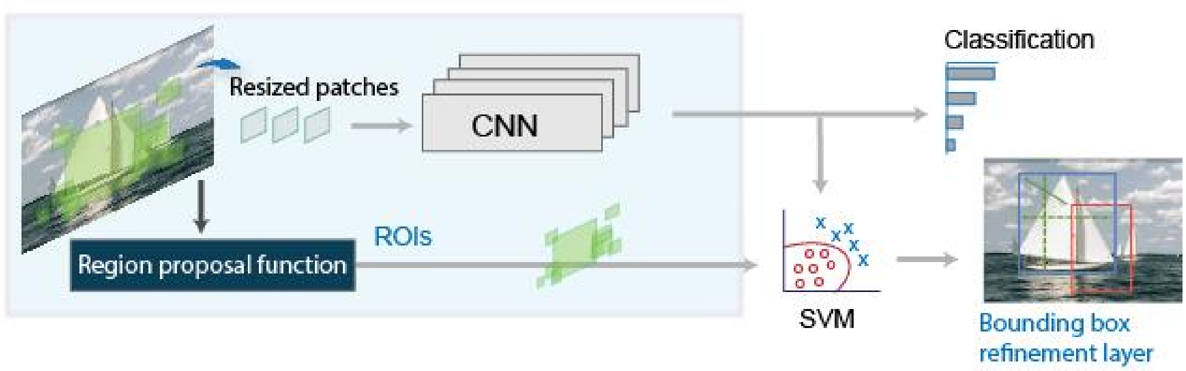

avg_pool
globalAverage...
1(S) × 1(S) × 2048(C) × 1(B)
rcnnFC
fullyConnected...
1(S) × 1(S) × 3(C) × 1(B)
rcnnBoxFC
fullyConnected...
1(S) × 1(S) × 8(C) × 1(B)
rcnnSoftmax
softmaxLayer
1(S) × 1(S) × 3(C) × 1(B)
rcnnBoxDeltas
rcnnBoxRegres...
1(S) × 1(S) × 8(C) × 1(B)
rcnnClassifica...
classificationLa...
1(S) × 1(S) × 3(C) × 1(B)


Deep Learning Network Analyzer
Analysis for training in Deep Network Designer
Name: Network from Deep Network Designer
Analysis date: 2022-10-02 17:22:24
33M total learnables
188 layers
1 warnings
0 errors
ISSUES
ANALYSIS RESULT
Name
Type
Activations
Learnable Prope...

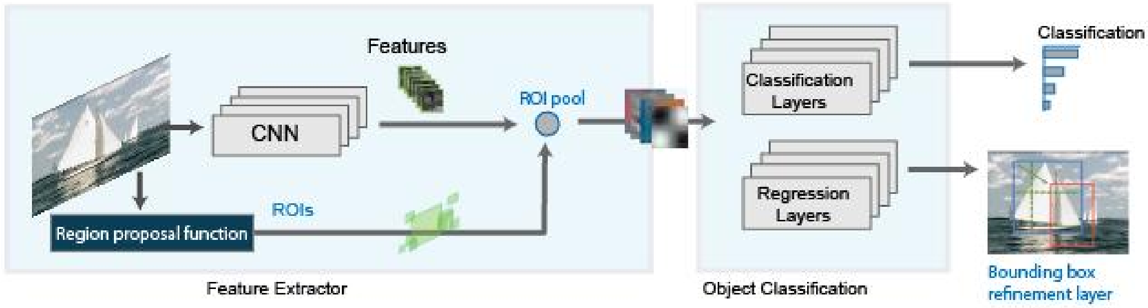


RCNNs Evolution

RCNN

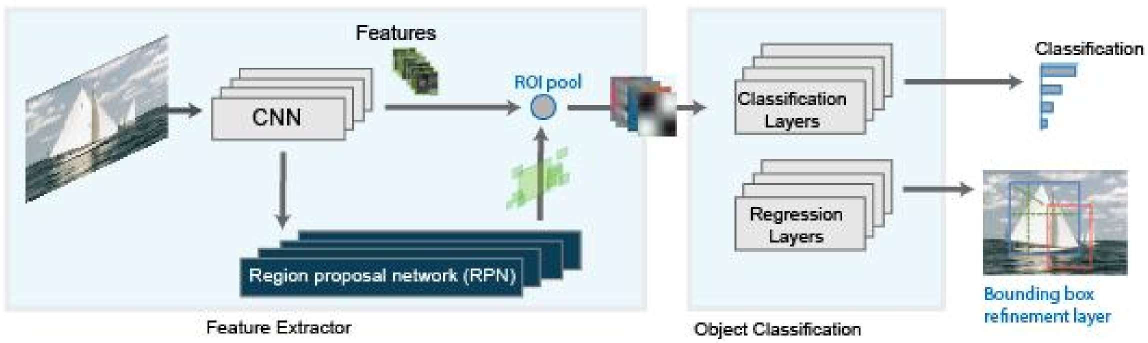


Fast-RCNN



Features
CNN
ROI pool
ROIs
Region proposal function
Feature Extractor
Classification Layers
Regression Layers
Object Classification
Classification
Bounding box refinement layer

Faster-RCNN



Features
CNN
ROI pool
Region proposal network (RPN)
Feature Extractor
Classification Layers
Regression Layers
Object Classification
Classification
Bounding box refinement layer

Object Detection Using Faster R-CNN Deep Learning

This example shows how to train a Faster R-CNN (regions with convolutional neural networks) object detector.

Deep learning is a powerful machine learning technique that you can use to train robust object detectors. Several deep learning techniques for object detection exist, including Faster R-CNN and you only look once (YOLO) v2. This example trains a Faster R-CNN vehicle detector using the `trainFasterRCNNObjectDetector` function. For more information, see Object Detection.

Download Pretrained Detector

Download a pretrained detector to avoid having to wait for training to complete. If you want to train the detector, set the `doTraining` variable to true.

```
doTraining = false;
if ~doTraining && ~exist('fasterRCNNResNet50EndToEndVehicleExample.mat','file')
    disp('Downloading pretrained detector (118 MB)...');
    pretrainedURL = 
'https://www.mathworks.com/supportfiles/vision/data/fasterRCNNResNet50EndToEndVehicleExample.mat';
    websave('fasterRCNNResNet50EndToEndVehicleExample.mat',pretrainedURL);
end
```

Load Data Set

This example uses a small labeled dataset that contains 295 images. Many of these images come from the Caltech Cars 1999 and 2001 data sets, available at the Caltech Computational Vision website, created by Pietro Perona and used with permission. Each image contains one or two labeled instances of a vehicle. A small dataset is useful for exploring the Faster R-CNN training procedure, but in practice, more labeled images are needed to train a robust detector. Unzip the vehicle images and load the vehicle ground truth data.

```
unzip vehicleDatasetImages.zip
data = load('vehicleDatasetGroundTruth.mat');
vehicleDataset = data.vehicleDataset;
```

The vehicle data is stored in a two-column table, where the first column contains the image file paths and the second column contains the vehicle bounding boxes.

Split the dataset into training, validation, and test sets. Select 60% of the data for training, 10% for validation, and the rest for testing the trained detector.

```
rng(0)
shuffledIndices = randperm(height(vehicleDataset));
idx = floor(0.6 * height(vehicleDataset));

trainingIdx = 1:idx;
trainingDataTbl = vehicleDataset(shuffledIndices(trainingIdx),:);
```

```
validationIdx = idx+1 : idx + 1 + floor(0.1 * length(shuffledIndices) );
validationDataTbl = vehicleDataset(shuffledIndices(validationIdx),:);

testIdx = validationIdx(end)+1 : length(shuffledIndices);
testDataTbl = vehicleDataset(shuffledIndices(testIdx),:);
```

Use `imageDatastore` and `boxLabelDatastore` to create datastores for loading the image and label data during training and evaluation.

```
imdsTrain = imageDatastore(trainingDataTbl{:,'imageFilename'});
bldsTrain = boxLabelDatastore(trainingDataTbl(:,'vehicle'));

imdsValidation = imageDatastore(validationDataTbl{:,'imageFilename'});
bldsValidation = boxLabelDatastore(validationDataTbl(:,'vehicle'));

imdsTest = imageDatastore(testDataTbl{:,'imageFilename'});
bldsTest = boxLabelDatastore(testDataTbl(:,'vehicle'));
```

Combine image and box label datastores.

```
trainingData = combine(imdsTrain,bldsTrain);
validationData = combine(imdsValidation,bldsValidation);
testData = combine(imdsTest,bldsTest);
```

Display one of the training images and box labels.

```
data = read(trainingData);
I = data{1};
bbox = data{2};
annotatedImage = insertShape(I,'Rectangle',bbox);
annotatedImage = imresize(annotatedImage,2);
figure
imshow(annotatedImage)
```

Create Faster R-CNN Detection Network

A Faster R-CNN object detection network is composed of a feature extraction network followed by two subnetworks. The feature extraction network is typically a pretrained CNN, such as ResNet-50 or Inception v3. The first subnetwork following the feature extraction network is a region proposal network (RPN) trained to generate object proposals - areas in the image where objects are likely to exist. The second subnetwork is trained to predict the actual class of each object proposal.

The feature extraction network is typically a pretrained CNN (for details, see Pretrained Deep Neural Networks). This example uses ResNet-50 for feature extraction. You can also use other pretrained networks such as MobileNet v2 or ResNet-18, depending on your application requirements.

Use `fasterRCNNLayers` to create a Faster R-CNN network automatically given a pretrained feature extraction network. `fasterRCNNLayers` requires you to specify several inputs that parameterize a Faster R-CNN network:

- Network input size
- Anchor boxes
- Feature extraction network

First, specify the network input size. When choosing the network input size, consider the minimum size required to run the network itself, the size of the training images, and the computational cost incurred by processing data at the selected size. When feasible, choose a network input size that is close to the size of the training image and larger than the input size required for the network. To reduce the computational cost of running the example, specify a network input size of [224 224 3], which is the minimum size required to run the network.

```
inputSize = [224 224 3];
```

Note that the training images used in this example are bigger than 224-by-224 and vary in size, so you must resize the images in a preprocessing step prior to training.

Next, use `estimateAnchorBoxes` to estimate anchor boxes based on the size of objects in the training data. To account for the resizing of the images prior to training, resize the training data for estimating anchor boxes. Use `transform` to preprocess the training data, then define the number of anchor boxes and estimate the anchor boxes.

```
preprocessedTrainingData = transform(trainingData,
@(data)preprocessData(data,inputSize));
numAnchors = 3;
anchorBoxes = estimateAnchorBoxes(preprocessedTrainingData,numAnchors)
```

For more information on choosing anchor boxes, see Estimate Anchor Boxes from Training Data (Computer Vision Toolbox™) and Anchor Box Basics.

Now, use `resnet50` to load a pretrained ResNet-50 model.

```
featureExtractionNetwork = resnet50;
```

Select `'activation_40_relu'` as the feature extraction layer. This feature extraction layer outputs feature maps that are downsampled by a factor of 16. This amount of downsampling is a good trade-off between spatial resolution and the strength of the extracted features, as features extracted further down the network encode stronger image features at the cost of spatial resolution. Choosing the optimal feature extraction layer requires empirical analysis. You can use `analyzeNetwork` to find the names of other potential feature extraction layers within a network.

```
featureLayer = 'activation_40_relu';
```

Define the number of classes to detect.

```
numClasses = width(vehicleDataset)-1;
```

Create the Faster R-CNN object detection network.

```
lgraph =
fasterRCNNLayers(inputSize,numClasses,anchorBoxes,featureExtractionNetwork,featu
reLayer);
```

You can visualize the network using `analyzeNetwork` or Deep Network Designer from Deep Learning Toolbox™.

If more control is required over the Faster R-CNN network architecture, use Deep Network Designer to design the Faster R-CNN detection network manually. For more information, see R-CNN, Fast R-CNN, and Faster R-CNN Basics.

Data Augmentation

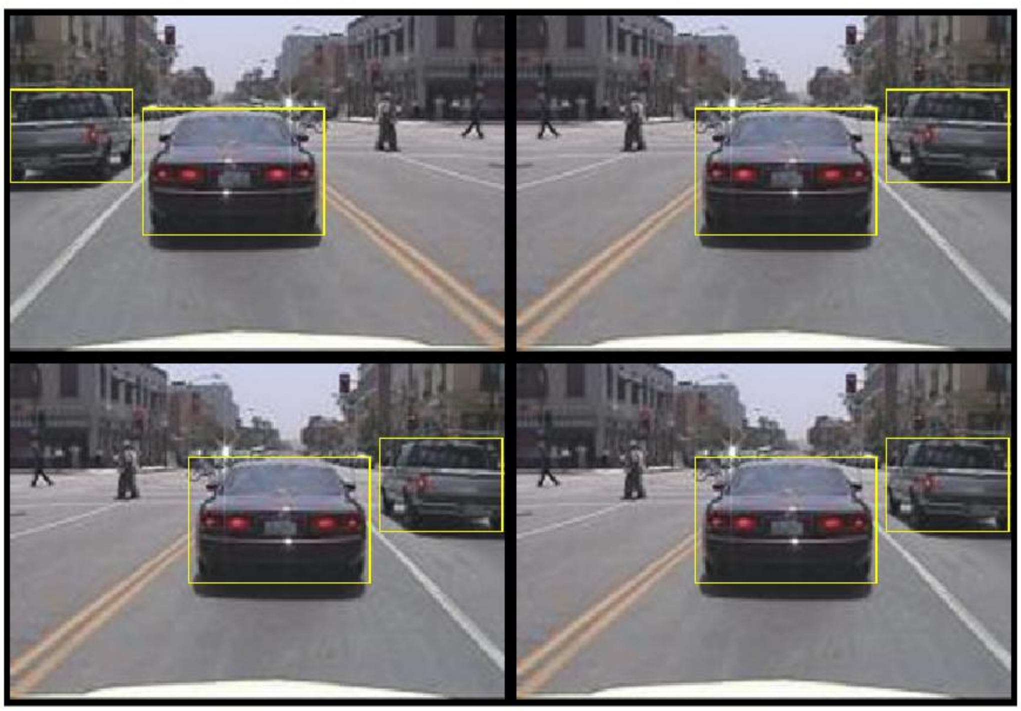
Data augmentation is used to improve network accuracy by randomly transforming the original data during training. By using data augmentation, you can add more variety to the training data without actually having to increase the number of labeled training samples.

Use `transform` to augment the training data by randomly flipping the image and associated box labels horizontally. Note that data augmentation is not applied to test and validation data. Ideally, test and validation data are representative of the original data and are left unmodified for unbiased evaluation.

```
augmentedTrainingData = transform(trainingData,@augmentData);
```

Read the same image multiple times and display the augmented training data.

```
augmentedData = cell(4,1);
for k = 1:4
    data = read(augmentedTrainingData);
    augmentedData{k} = insertShape(data{1},'Rectangle',data{2});
    reset(augmentedTrainingData);
end
figure
montage(augmentedData,'BorderSize',10)
```



Preprocess Training Data

Preprocess the augmented training data, and the validation data to prepare for training.

```
trainingData =
transform(augmentedTrainingData,@(data)preprocessData(data,inputSize));
validationData =
transform(validationData,@(data)preprocessData(data,inputSize));
```

Read the preprocessed data.

```
data = read(trainingData);
```

Display the image and box bounding boxes.

```
I = data{1};
bbox = data{2};
annotatedImage = insertShape(I,'Rectangle',bbox);
annotatedImage = imresize(annotatedImage,2);
figure
imshow(annotatedImage)
```

Train Faster R-CNN

Use `trainingOptions` to specify network training options. Set `'ValidationData'` to the preprocessed validation data. Set `'CheckpointPath'` to a temporary location. This enables the saving of partially trained detectors during the training process. If training is interrupted, such as by a power outage or system failure, you can resume training from the saved checkpoint.

```
options = trainingOptions('sgdm',...
    'MaxEpochs',10,...
    'MiniBatchSize',2,...
    'InitialLearnRate',1e-3,...
    'CheckpointPath',tempdir,...
    'ValidationData',validationData);
```

Use `trainFasterRCNNObjectDetector` to train Faster R-CNN object detector if `doTraining` is true. Otherwise, load the pretrained network.

```
if doTraining
    % Train the Faster R-CNN detector.
    % * Adjust NegativeOverlapRange and PositiveOverlapRange to ensure
    %   that training samples tightly overlap with ground truth.
    [detector, info] = 
trainFasterRCNNObjectDetector(trainingData,lgraph,options, ...
        'NegativeOverlapRange',[0 0.3], ...
        'PositiveOverlapRange',[0.6 1]);
else
    % Load pretrained detector for the example.
    pretrained = load('fasterRCNNResNet50EndToEndVehicleExample.mat');
    detector = pretrained.detector;
end
```

This example was verified on an Nvidia(TM) Titan X GPU with 12 GB of memory. Training the network took approximately 20 minutes. The training time varies depending on the hardware you use.

As a quick check, run the detector on one test image. Make sure you resize the image to the same size as the training images.

```
I = imread(testDataTbl.imageFilename{3});
I = imresize(I,inputSize(1:2));
[bboxes,scores] = detect(detector,I);
```

Display the results.

```
I = insertObjectAnnotation(I,'rectangle',bboxes,scores);
figure
imshow(I)
```

Evaluate Detector Using Test Set

Evaluate the trained object detector on a large set of images to measure the performance. Computer Vision Toolbox™ provides object detector evaluation functions to measure common metrics such as average precision (`evaluateDetectionPrecision`) and log-average miss rates (`evaluateDetectionMissRate`). For this example, use the average precision metric to evaluate performance. The average precision provides a single number that incorporates the ability of the detector to make correct classifications (precision) and the ability of the detector to find all relevant objects (recall).

Apply the same preprocessing transform to the test data as for the training data.

```
testData = transform(testData,@(data)preprocessData(data,inputSize));
```

Run the detector on all the test images.

```
detectionResults = detect(detector,testData,'MinibatchSize',4);
```

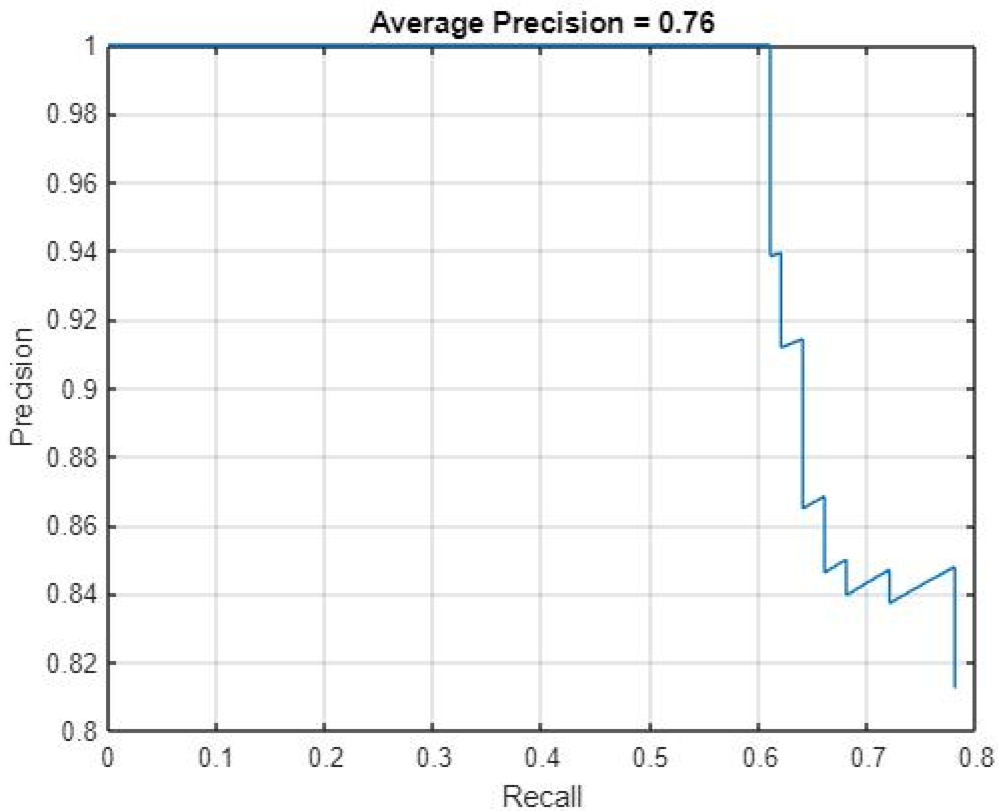
Evaluate the object detector using the average precision metric.

```
[ap, recall, precision] = evaluateDetectionPrecision(detectionResults,testData);
```

The precision/recall (PR) curve highlights how precise a detector is at varying levels of recall. The ideal precision is 1 at all recall levels. The use of more data can help improve the average precision but might require more training time. Plot the PR curve.

```
figure
plot(recall,precision)
xlabel('Recall')
ylabel('Precision')
grid on
```

```
title(sprintf('Average Precision = %.2f', ap))
```



Supporting Functions

```
function data = augmentData(data)
% Randomly flip images and bounding boxes horizontally.
tform = randomAffine2d('XReflection',true);
sz = size(data{1});
rout = affineOutputView(sz,tform);
data{1} = imwarp(data{1},tform,'OutputView',rout);

% Sanitize box data, if needed.
data{2} = helperSanitizeBoxes(data{2}, sz);

% Warp boxes.
data{2} = bboxwarp(data{2},tform,rout);
end

function data = preprocessData(data,targetSize)
% Resize image and bounding boxes to targetSize.
sz = size(data{1},[1 2]);
scale = targetSize(1:2)./sz;
```

```
data{1} = imresize(data{1},targetSize(1:2));

% Sanitize box data, if needed.
data{2} = helperSanitizeBoxes(data{2}, sz);

% Resize boxes.
data{2} = bboxresize(data{2},scale);
end
```

References

[1] Ren, S., K. He, R. Gershick, and J. Sun. "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks." *IEEE Transactions of Pattern Analysis and Machine Intelligence*. Vol. 39, Issue 6, June 2017, pp. 1137-1149.

[2] Girshick, R., J. Donahue, T. Darrell, and J. Malik. "Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation." *Proceedings of the 2014 IEEE Conference on Computer Vision and Pattern Recognition*. Columbus, OH, June 2014, pp. 580-587.

[3] Girshick, R. "Fast R-CNN." *Proceedings of the 2015 IEEE International Conference on Computer Vision*. Santiago, Chile, Dec. 2015, pp. 1440-1448.

[4] Zitnick, C. L., and P. Dollar. "Edge Boxes: Locating Object Proposals from Edges." *European Conference on Computer Vision*. Zurich, Switzerland, Sept. 2014, pp. 391-405.

[5] Uijlings, J. R. R., K. E. A. van de Sande, T. Gevers, and A. W. M. Smeulders. "Selective Search for Object Recognition." *International Journal of Computer Vision*. Vol. 104, Number 2, Sept. 2013, pp. 154-171.

Object Detection Using YOLO v3 Deep Learning

This example shows how to train a YOLO v3 object detector.

Deep learning is a powerful machine learning technique that you can use to train robust object detectors. Several techniques for object detection exist, including Faster R-CNN, you only look once (YOLO) v2, and single shot detector (SSD). This example shows how to train a YOLO v3 object detector. YOLO v3 improves upon YOLO v2 by adding detection at multiple scales to help detect smaller objects. The loss function used for training is separated into mean squared error for bounding box regression and binary cross-entropy for object classification to help improve detection accuracy.

Note: This example requires the Computer Vision Toolbox™ Model for YOLO v3 Object Detection. You can install the Computer Vision Toolbox Model for YOLO v3 Object Detection from Add-On Explorer. For more information about installing add-ons, see Get and Manage Add-Ons.

Download Pretrained Network

Download a pretrained network using the helper function `downloadPretrainedYOLOv3Detector` to avoid having to wait for training to complete. If you want to train the network, set the `doTraining` variable to `true`.

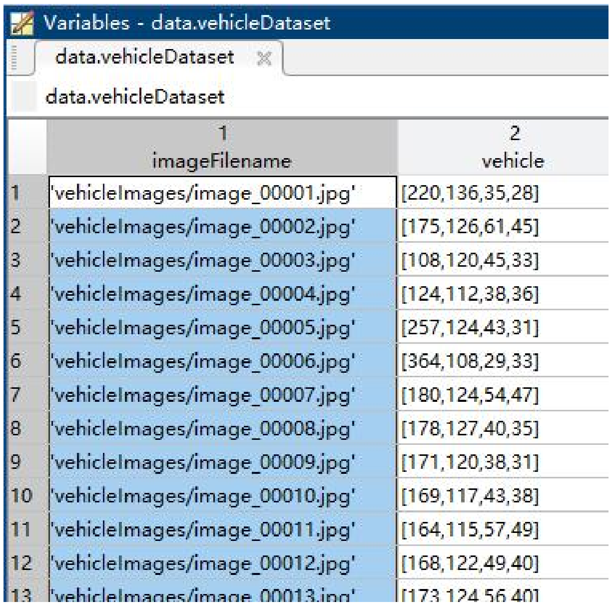
```
doTraining = false;

if ~doTraining
    preTrainedDetector = downloadPretrainedYOLOv3Detector();
end
```

Load Data

This example uses a small labeled data set that contains 295 images. Many of these images come from the Caltech Cars 1999 and 2001 data sets, available at the Caltech Computational Vision website, created by Pietro Perona and used with permission. Each image contains one or two labeled instances of a vehicle. A small data set is useful for exploring the YOLO v3 training procedure, but in practice, more labeled images are needed to train a robust network.

Unzip the vehicle images and load the vehicle ground truth data.

```
unzip vehicleDatasetImages.zip
data = load('vehicleDatasetGroundTruth.mat');
vehicleDataset = data.vehicleDataset;

% Add the full path to the local vehicle data folder.
vehicleDataset.imageFilename = fullfile(pwd, vehicleDataset.imageFilename);
```



	1 imageFilename	2 vehicle
1	'vehicleImages/image_00001.jpg'	[220,136,35,28]
2	'vehicleImages/image_00002.jpg'	[175,126,61,45]
3	'vehicleImages/image_00003.jpg'	[108,120,45,33]
4	'vehicleImages/image_00004.jpg'	[124,112,38,36]
5	'vehicleImages/image_00005.jpg'	[257,124,43,31]
6	'vehicleImages/image_00006.jpg'	[364,108,29,33]
7	'vehicleImages/image_00007.jpg'	[180,124,54,47]
8	'vehicleImages/image_00008.jpg'	[178,127,40,35]
9	'vehicleImages/image_00009.jpg'	[171,120,38,31]
10	'vehicleImages/image_00010.jpg'	[169,117,43,38]
11	'vehicleImages/image_00011.jpg'	[164,115,57,49]
12	'vehicleImages/image_00012.jpg'	[168,122,49,40]
13	'vehicleImages/image_00013.jpg'	[173,124,56,40]

Note: In case of multiple classes, the data can also organized as three columns where the first column contains the image file names with paths, the second column contains the bounding boxes and the third column must be a cell vector that contains the label names corresponding to each bounding box. For more information on how to arrange the bounding boxes and labels, see `boxLabelDatastore`.

All the bounding boxes must be in the form `[x y width height]`. This vector specifies the upper left corner and the size of the bounding box in pixels.

Split the data set into a training set for training the network, and a test set for evaluating the network. Use 60% of the data for training set and the rest for the test set.

```
rng(0);
shuffledIndices = randperm(height(vehicleDataset));
idx = floor(0.6 * length(shuffledIndices));
trainingDataTbl = vehicleDataset(shuffledIndices(1:idx), :);
testDataTbl = vehicleDataset(shuffledIndices(idx+1:end), :);
```

Create an image datastore for loading the images.

```
imdsTrain = imageDatastore(trainingDataTbl.imageFilename);
imdsTest = imageDatastore(testDataTbl.imageFilename);
```

Create a datastore for the ground truth bounding boxes.

```
bldsTrain = boxLabelDatastore(trainingDataTbl(:, 2:end));
bldsTest = boxLabelDatastore(testDataTbl(:, 2:end));
```

Combine the image and box label datastores.

```
trainingData = combine(imdsTrain, bldsTrain);
```

```
testData = combine(imdsTest, bldsTest);
```

Use `validateInputData` to detect invalid images, bounding boxes or labels i.e.,

- Samples with invalid image format or containing NaNs
- Bounding boxes containing zeros/NaNs/Infs/empty
- Missing/non-categorical labels.

The values of the bounding boxes should be finite, positive, non-fractional, non-NaN and should be within the image boundary with a positive height and width. Any invalid samples must either be discarded or fixed for proper training.

```
validateInputData(trainingData);
validateInputData(testData);
```

Data Augmentation

Data augmentation is used to improve network accuracy by randomly transforming the original data during training. By using data augmentation, you can add more variety to the training data without actually having to increase the number of labeled training samples.

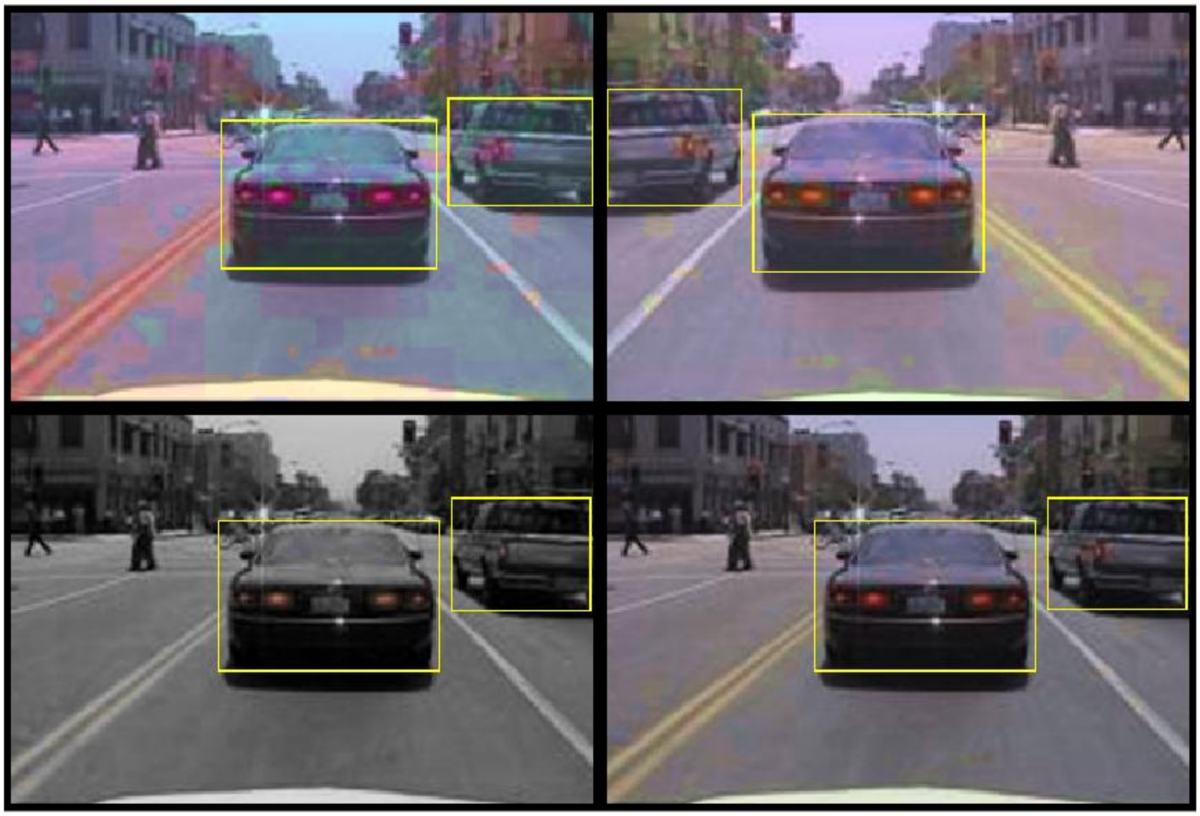
Use `transform` function to apply custom data augmentations to the training data. The `augmentData` helper function, listed at the end of the example, applies the following augmentations to the input data.

- Color jitter augmentation in HSV space
- Random horizontal flip
- Random scaling by 10 percent

```
augmentedTrainingData = transform(trainingData, @augmentData);
```

Read the same image four times and display the augmented training data.

```
% Visualize the augmented images.
augmentedData = cell(4,1);
for k = 1:4
    data = read(augmentedTrainingData);
    augmentedData{k} = insertShape(data{1,1}, 'Rectangle', data{1,2});
    reset(augmentedTrainingData);
end
figure
montage(augmentedData, 'BorderSize', 10)
```

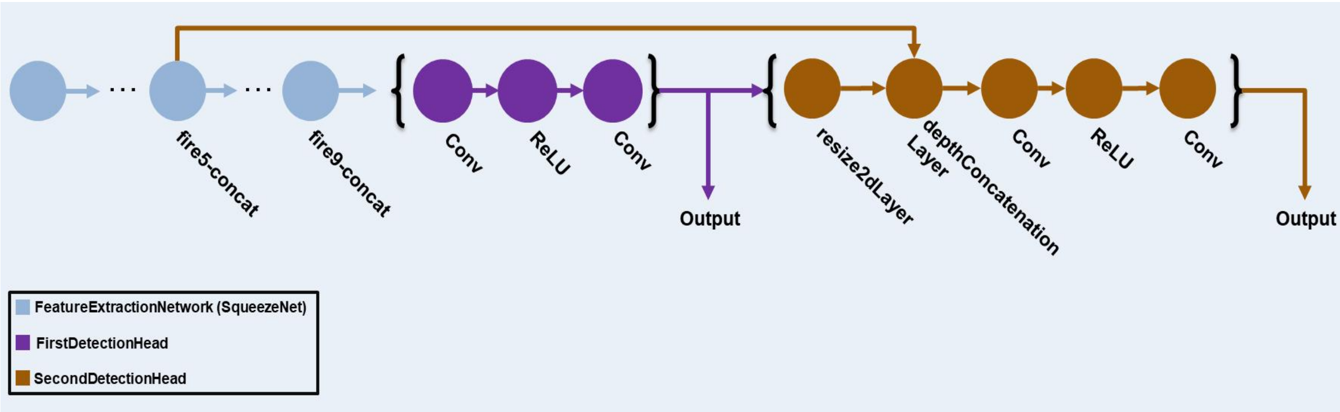


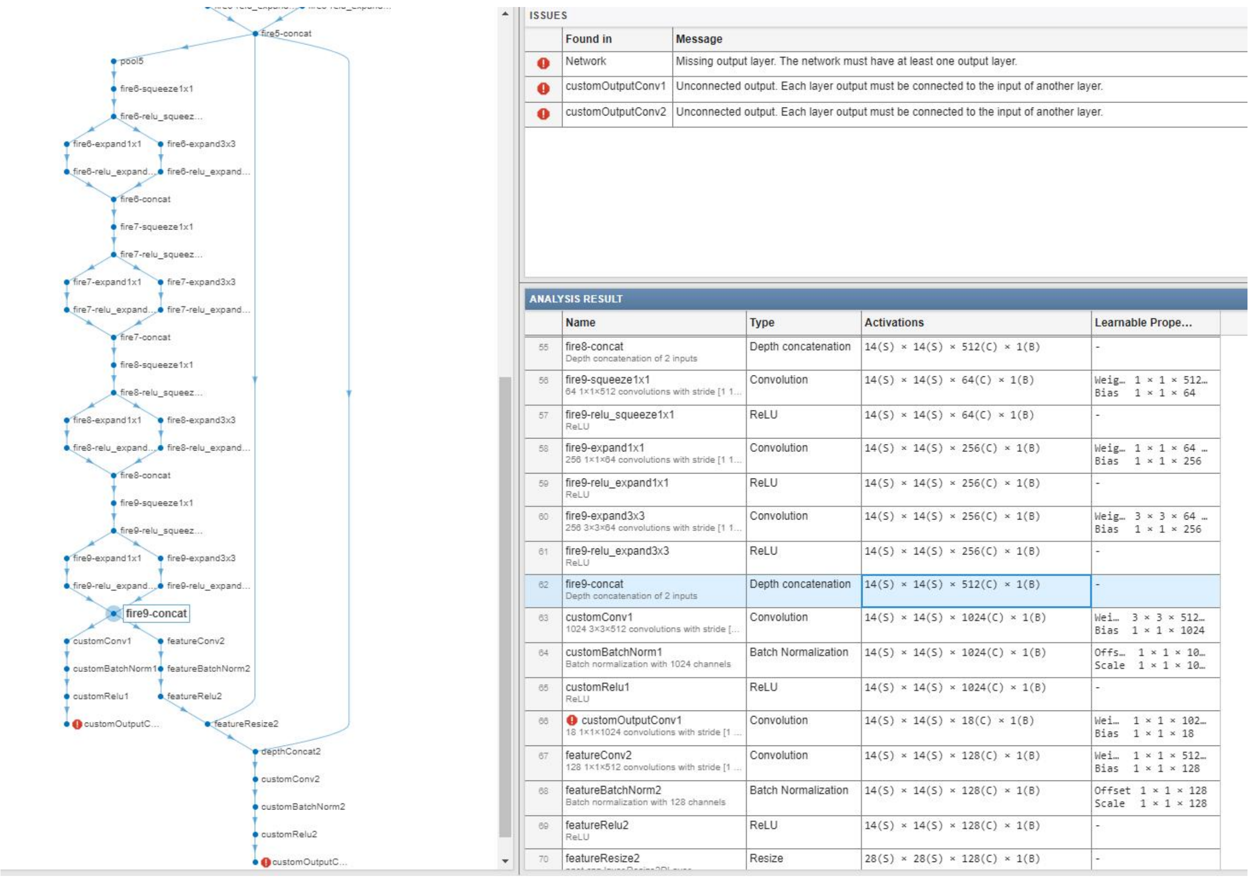
Define YOLO v3 Object Detector

The YOLO v3 detector in this example is based on SqueezeNet, and uses the feature extraction network in SqueezeNet with the addition of two detection heads at the end. The second detection head is twice the size of the first detection head, so it is better able to detect small objects. Note that you can specify any number of detection heads of different sizes based on the size of the objects that you want to detect. The YOLO v3 detector uses anchor boxes estimated using training data to have better initial priors corresponding to the type of data set and to help the detector learn to predict the boxes accurately. For information about anchor boxes, see Anchor Boxes for Object Detection.

The YOLO v3 network present in the YOLO v3 detector is illustrated in the following diagram.

You can use Deep Network Designer to create the network shown in the diagram.





Specify the network input size. When choosing the network input size, consider the minimum size required to run the network itself, the size of the training images, and the computational cost incurred by processing data at the selected size. When feasible, choose a network input size that is close to the size of the training image and larger than the input size required for the network. To reduce the computational cost of running the example, specify a network input size of [227 227 3].

```
networkInputSize = [227 227 3];
```

First, use `transform` to preprocess the training data for computing the anchor boxes, as the training images used in this example are bigger than 227-by-227 and vary in size. Specify the number of anchors as 6 to achieve a good tradeoff between number of anchors and mean IoU. Use the `estimateAnchorBoxes` function to estimate the anchor boxes. For details on estimating anchor boxes, see Estimate Anchor Boxes From Training Data. In case of using a pretrained YOLOv3 object detector, the anchor boxes calculated on that particular training dataset need to be specified. Note that the estimation process is not deterministic. To prevent the estimated anchor boxes from changing while tuning other hyperparameters set the random seed prior to estimation using rng.

```
rng(0)
trainingDataForEstimation = transform(trainingData, @(data)preprocessData(data,
networkInputSize));
numAnchors = 6;
[anchors, meanIoU] = estimateAnchorBoxes(trainingDataForEstimation, numAnchors)
```

```
anchors = 6×2
    41    34
   159   131
    98    93
   143   121
    33    23
    69    66
```

Specify `anchorBoxes` to use in both the detection heads. `anchorBoxes` is a cell array of [Mx1], where M denotes the number of detection heads. Each detection head consists of a [Nx2] matrix of `anchors`, where N is the number of anchors to use. Select `anchorBoxes` for each detection head based on the feature map size. Use larger `anchors` at lower scale and smaller `anchors` at higher scale. To do so, sort the `anchors` with the larger anchor boxes first and assign the first three to the first detection head and the next three to the second detection head.

```
area = anchors(:, 1).*anchors(:, 2);
[~, idx] = sort(area, 'descend');
anchors = anchors(idx, :);
anchorBoxes = {anchors(1:3,:)
    anchors(4:6,:)
    };
```

anchorBoxes = 2×1 cell

	1
1	[159,131;143,121;98,93]
2	[69,66;41,34;33,23]

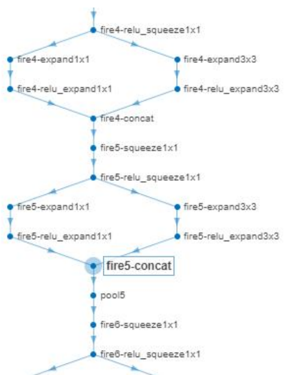
Load the SqueezeNet network pretrained on Imagenet data set and then specify the class names. You can also choose to load a different pretrained network trained on COCO data set such as `tiny-`

yolov3-coco or darknet53-coco or Imagenet data set such as MobileNet-v2 or ResNet-18. YOLO v3 performs better and trains faster when you use a pretrained network.

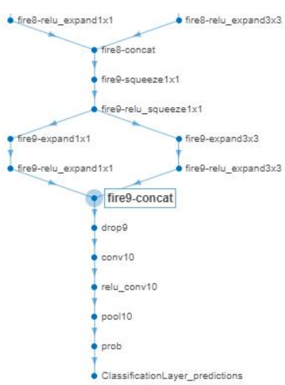
```
baseNetwork = squeezenet;
classNames = trainingDataTbl.Properties.VariableNames(2:end);
```

Next, create the yolov3ObjectDetector object by adding the detection network source. Choosing the optimal detection network source requires trial and error, and you can use analyzeNetwork to find the names of potential detection network source within a network. For this example, use the fire9-concat and fire5-concat layers as DetectionNetworkSource.

```
yolov3Detector = yolov3ObjectDetector(baseNetwork, classNames, anchorBoxes,
'DetectionNetworkSource', {'fire9-concat', 'fire5-concat'});
```



28	fire5-relu_squeeze1x1 ReLU	ReLU	28(S) × 28(S) × 32(C) × 1(B)	-
29	fire5-expand1x1 128 1×1×32 convolutions with stride [1 1...	Convolution	28(S) × 28(S) × 128(C) × 1(B)	Weig… 1 × 1 × 32 … Bias 1 × 1 × 128
30	fire5-relu_expand1x1 ReLU	ReLU	28(S) × 28(S) × 128(C) × 1(B)	-
31	fire5-expand3x3 128 3×3×32 convolutions with stride [1 1...	Convolution	28(S) × 28(S) × 128(C) × 1(B)	Weig… 3 × 3 × 32 … Bias 1 × 1 × 128
32	fire5-relu_expand3x3 ReLU	ReLU	28(S) × 28(S) × 128(C) × 1(B)	-
33	fire5-concat Depth concatenation of 2 inputs	Depth concatenation	28(S) × 28(S) × 256(C) × 1(B)	-
34	pool5 3×3 max pooling with stride [2 2] and pa...	Max Pooling	14(S) × 14(S) × 256(C) × 1(B)	-
35	fire6-squeeze1x1 48 1×1×256 convolutions with stride [1 1...	Convolution	14(S) × 14(S) × 48(C) × 1(B)	Weig… 1 × 1 × 256… Bias 1 × 1 × 48
36	fire6-relu_squeeze1x1 ReLU	ReLU	14(S) × 14(S) × 48(C) × 1(B)	-
37	fire6-expand1x1 192 1×1×48 convolutions with stride [1 1...	Convolution	14(S) × 14(S) × 192(C) × 1(B)	Weig… 1 × 1 × 48 … Bias 1 × 1 × 192



58	fire9-expand1x1 256 1×1×64 convolutions with stride [1 1...	Convolution	14(S) × 14(S) × 256(C) × 1(B)	Weig… 1 × 1 × 64 … Bias 1 × 1 × 256
59	fire9-relu_expand1x1 ReLU	ReLU	14(S) × 14(S) × 256(C) × 1(B)	-
60	fire9-expand3x3 256 3×3×64 convolutions with stride [1 1...	Convolution	14(S) × 14(S) × 256(C) × 1(B)	Weig… 3 × 3 × 64 … Bias 1 × 1 × 256
61	fire9-relu_expand3x3 ReLU	ReLU	14(S) × 14(S) × 256(C) × 1(B)	-
62	fire9-concat Depth concatenation of 2 inputs	Depth concatenation	14(S) × 14(S) × 512(C) × 1(B)	-
63	drop9 50% dropout	Dropout	14(S) × 14(S) × 512(C) × 1(B)	-
64	conv10 1000 1×1×512 convolutions with stride [...	Convolution	14(S) × 14(S) × 1000(C) × 1(B)	Wei… 1 × 1 × 512… Bias 1 × 1 × 1000
65	relu_conv10 ReLU	ReLU	14(S) × 14(S) × 1000(C) × 1(B)	-
66	pool10 2-D global average pooling	2-D Global Average...	1(S) × 1(S) × 1000(C) × 1(B)	-
67	prob softmax	Softmax	1(S) × 1(S) × 1000(C) × 1(B)	-
68	ClassificationLayer_predictions crossentropyex with 'tench' and 999 oth...	Classification Output	1(S) × 1(S) × 1000(C) × 1(B)	-

Alternatively, instead of the network created above using SqueezeNet, other pretrained YOLOv3 architectures trained using larger datasets like MS-COCO can be used to transfer learn the detector on custom object detection task. Transfer learning can be realized by changing the classNames and anchorBoxes.

Preprocess Training Data

Preprocess the augmented training data to prepare for training. The preprocess method in yolov3ObjectDetector, applies the following preprocessing operations to the input data.

- Resize the images to the network input size by maintaining the aspect ratio.
- Scale the image pixels in the range `[0 1]`.

```
preprocessedTrainingData = transform(augmentedTrainingData,
@(data)preprocess(yolov3Detector, data));
```

Read the preprocessed training data.

```
data = read(preprocessedTrainingData);
```

Display the image with the bounding boxes.

```
I = data{1,1};
bbox = data{1,2};
annotatedImage = insertShape(I, 'Rectangle', bbox);
annotatedImage = imresize(annotatedImage,2);
figure
imshow(annotatedImage)
```

Reset the datastore.

```
reset(preprocessedTrainingData);
```

Specify Training Options

Specify these training options.

- Set the number of epochs to be 80.
- Set the mini batch size as `8`. Stable training can be possible with higher learning rates when higher mini batch size is used. Although, this should be set depending on the available memory.
- Set the learning rate to 0.001.
- Set the warmup period as `1000` iterations. This parameter denotes the number of iterations to increase the learning rate exponentially based on the formula $\text{learningRate} \times \left(\frac{\text{iteration}}{\text{warmupPeriod}}\right)^4$. It helps in stabilizing the gradients at higher learning rates.
- Set the L2 regularization factor to 0.0005.
- Specify the penalty threshold as 0.5. Detections that overlap less than 0.5 with the ground truth are penalized.
- Initialize the velocity of gradient as `[]`. This is used by SGDM to store the velocity of gradients.

```
numEpochs = 80;
miniBatchSize = 8;
learningRate = 0.001;
warmupPeriod = 1000;
l2Regularization = 0.0005;
penaltyThreshold = 0.5;
velocity = [];
```

Train Model

Train on a GPU, if one is available. Using a GPU requires Parallel Computing Toolbox™ and a CUDA® enabled NVIDIA® GPU. For information about the supported compute capabilities, see GPU Support by Release.

Use the `minibatchqueue` function to split the preprocessed training data into batches with the supporting function `createBatchData` which returns the batched images and bounding boxes combined with the respective class IDs. For faster extraction of the batch data for training, `dispatchInBackground` should be set to "true" which ensures the usage of parallel pool.

`minibatchqueue` automatically detects the availability of a GPU. If you do not have a GPU, or do not want to use one for training, set the `OutputEnvironment` parameter to `"cpu"`.

```
if canUseParallelPool
   dispatchInBackground = true;
else
   dispatchInBackground = false;
end
```

```
mbqTrain = minibatchqueue(preprocessedTrainingData, 2,...
        "MiniBatchSize", miniBatchSize,...
        "MiniBatchFcn", @(images, boxes, labels) createBatchData(images, boxes, 
labels, classNames), ...
        "MiniBatchFormat", ["SSCB", ""],...
        "DispatchInBackground", dispatchInBackground,...
        "OutputCast", ["", "double"]);
```

Create the training progress plotter using supporting function `configureTrainingProgressPlotter` to see the plot while training the detector object with a custom training loop.

Finally, specify the custom training loop. For each iteration:

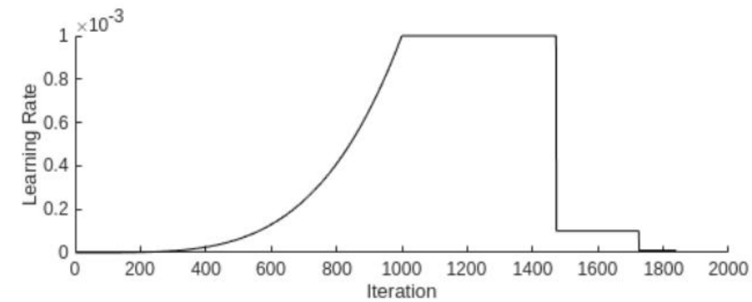
- Read data from the `minibatchqueue`. If it doesn't have any more data, reset the `minibatchqueue` and shuffle.
- Evaluate the model gradients using `dlfeval` and the `modelGradients` function. The function `modelGradients`, listed as a supporting function, returns the gradients of the loss with respect to the learnable parameters in `net`, the corresponding mini-batch loss, and the state of the current batch.
- Apply a weight decay factor to the gradients to regularization for more robust training.
- Determine the learning rate based on the iterations using the `piecewiseLearningRateWithWarmup` supporting function.
- Update the detector parameters using the `sgdmupdate` function.
- Update the `state` parameters of detector with the moving average.
- Display the learning rate, total loss, and the individual losses (box loss, object loss and class loss) for every iteration. These can be used to interpret how the respective losses are changing in each iteration. For example, a sudden spike in the box loss after few iterations implies that there are Inf or NaNs in the predictions.
- Update the training progress plot.

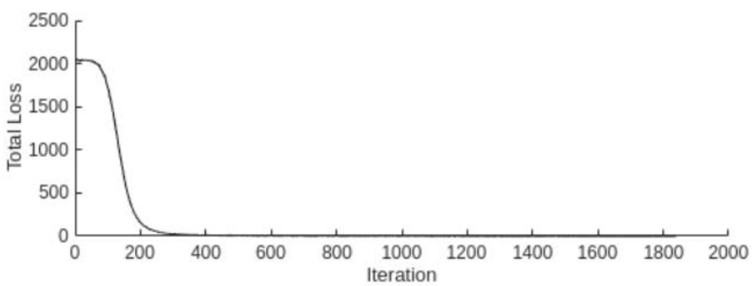
The training can also be terminated if the loss has saturated for few epochs.

```
if doTraining

    % Create subplots for the learning rate and mini-batch loss.
    fig = figure;
    [lossPlotter, learningRatePlotter] = configureTrainingProgressPlotter(fig);

    iteration = 0;
    % Custom training loop.
    for epoch = 1:numEpochs

        reset(mbqTrain);
        shuffle(mbqTrain);

        while(hasdata(mbqTrain))
```

```
            iteration = iteration + 1;

            [XTrain, YTrain] = next(mbqTrain);

            % Evaluate the model gradients and loss using dlfeval and the
            % modelGradients function.
            [gradients, state, lossInfo] = dlfeval(@modelGradients,
yolov3Detector, XTrain, YTrain, penaltyThreshold);

            % Apply L2 regularization.
            gradients = dlupdate(@(g,w) g + l2Regularization*w, gradients,
yolov3Detector.Learnables);

            % Determine the current learning rate value.
            currentLR = piecewiseLearningRateWithWarmup(iteration, epoch,
learningRate, warmupPeriod, numEpochs);

            % Update the detector learnable parameters using the SGDM optimizer.
            [yolov3Detector.Learnables, velocity] =
sgdmupdate(yolov3Detector.Learnables, gradients, velocity, currentLR);

            % Update the state parameters of dlnetwork.
            yolov3Detector.State = state;

            % Display progress.
            displayLossInfo(epoch, iteration, currentLR, lossInfo);

            % Update training plot with new points.
            updatePlots(lossPlotter, learningRatePlotter, iteration, currentLR,
lossInfo.totalLoss);
        end
    end
else
    yolov3Detector = preTrainedDetector;
end
```

```
Starting parallel pool (parpool) using the 'Processes' profile ...
Connected to the parallel pool (number of workers: 4).
Epoch : 1 | Iteration : 1 | Learning Rate : 1e-15 | Total Loss : 2039.7584 | Box Loss : 3.6046 | Object Loss : 2035.6167 | Class Loss : 0.53712
Epoch : 1 | Iteration : 2 | Learning Rate : 1.6e-14 | Total Loss : 2039.111 | Box Loss : 1.8679 | Object Loss : 2036.5901 | Class Loss : 0.653
Epoch : 1 | Iteration : 3 | Learning Rate : 8.1e-14 | Total Loss : 2040.2465 | Box Loss : 4.0019 | Object Loss : 2035.5315 | Class Loss : 0.71301
Epoch : 1 | Iteration : 4 | Learning Rate : 2.56e-13 | Total Loss : 2038.1058 | Box Loss : 6.0349 | Object Loss : 2031.5471 | Class Loss : 0.52376
Epoch : 1 | Iteration : 5 | Learning Rate : 6.25e-13 | Total Loss : 2036.6228 | Box Loss : 3.2029 | Object Loss : 2032.6777 | Class Loss : 0.74224
Epoch : 1 | Iteration : 6 | Learning Rate : 1.296e-12 | Total Loss : 2053.8889 | Box Loss : 2.8815 | Object Loss : 2050.2905 | Class Loss : 0.71679
Epoch : 1 | Iteration : 7 | Learning Rate : 2.401e-12 | Total Loss : 2049.2183 | Box Loss : 2.0645 | Object Loss : 2046.4749 | Class Loss : 0.67886
Epoch : 1 | Iteration : 8 | Learning Rate : 4.096e-12 | Total Loss : 2043.5309 | Box Loss : 4.4251 | Object Loss : 2038.4241 | Class Loss : 0.68174
Epoch : 1 | Iteration : 9 | Learning Rate : 6.561e-12 | Total Loss : 2040.01 | Box Loss : 1.9779 | Object Loss : 2037.3914 | Class Loss : 0.64071
Epoch : 1 | Iteration : 10 | Learning Rate : 1e-11 | Total Loss : 2039.1467 | Box Loss : 3.0199 | Object Loss : 2035.7871 | Class Loss : 0.33968
Epoch : 1 | Iteration : 11 | Learning Rate : 1.4641e-11 | Total Loss : 2033.5656 | Box Loss : 3.3712 | Object Loss : 2029.441 | Class Loss : 0.75325
Epoch : 1 | Iteration : 12 | Learning Rate : 2.0736e-11 | Total Loss : 2025.2057 | Box Loss : 2.256 | Object Loss : 2022.5033 | Class Loss : 0.44628
Epoch : 1 | Iteration : 13 | Learning Rate : 2.8561e-11 | Total Loss : 2024.9935 | Box Loss : 1.7195 | Object Loss : 2022.5596 | Class Loss : 0.71451
Epoch : 1 | Iteration : 14 | Learning Rate : 3.8416e-11 | Total Loss : 2033.99 | Box Loss : 4.2615 | Object Loss : 2028.7534 | Class Loss : 0.97508
Epoch : 1 | Iteration : 15 | Learning Rate : 5.0625e-11 | Total Loss : 2045.5712 | Box Loss : 5.5158 | Object Loss : 2039.7009 | Class Loss : 0.35435
Epoch : 1 | Iteration : 16 | Learning Rate : 6.5536e-11 | Total Loss : 2040.9429 | Box Loss : 2.0156 | Object Loss : 2038.3127 | Class Loss : 0.61456
Epoch : 1 | Iteration : 17 | Learning Rate : 8.3521e-11 | Total Loss : 2043.8091 | Box Loss : 3.3074 | Object Loss : 2039.9788 | Class Loss : 0.52297
Epoch : 1 | Iteration : 18 | Learning Rate : 1.0498e-10 | Total Loss : 2026.7371 | Box Loss : 1.1161 | Object Loss : 2024.9329 | Class Loss : 0.68814
```





Evaluate Model

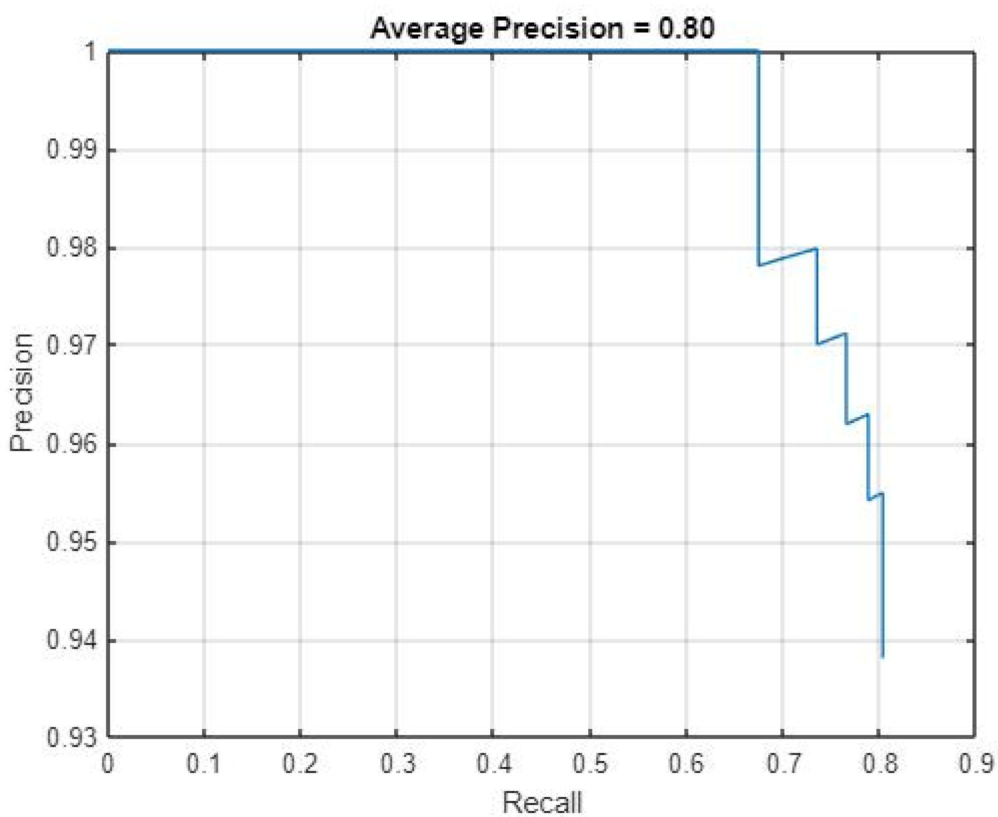
Computer Vision Toolbox™ provides object detector evaluation functions to measure common metrics such as average precision (`evaluateDetectionPrecision`) and log-average miss rates (`evaluateDetectionMissRate`). In this example, the average precision metric is used. The average precision provides a single number that incorporates the ability of the detector to make correct classifications (precision) and the ability of the detector to find all relevant objects (recall).

```
results = detect(yolov3Detector,testData,'MiniBatchSize',8);

% Evaluate the object detector using Average Precision metric.
[ap,recall,precision] = evaluateDetectionPrecision(results,testData);
```

The precision-recall (PR) curve shows how precise a detector is at varying levels of recall. Ideally, the precision is 1 at all recall levels.

```
% Plot precision-recall curve.
figure
```

```
plot(recall,precision)
xlabel('Recall')
ylabel('Precision')
grid on
title(sprintf('Average Precision = %.2f', ap))
```



Detect Objects Using YOLO v3

Use the detector for object detection.

```
% Read the datastore.
data = read(testData);

% Get the image.
I = data{1};

[bboxes,scores,labels] = detect(yolov3Detector,I);

% Display the detections on image.
I = insertObjectAnnotation(I,'rectangle',bboxes,scores);

figure
```

```
imshow(I)
```

Supporting Functions

Model Gradients Function

The function `modelGradients` takes the `yolov3ObjectDetector` object, a mini-batch of input data `XTrain` with corresponding ground truth boxes `YTrain`, the specified penalty threshold as input arguments and returns the gradients of the loss with respect to the learnable parameters in `yolov3ObjectDetector`, the corresponding mini-batch loss information, and the state of the current batch.

The model gradients function computes the total loss and gradients by performing these operations.

- Generate predictions from the input batch of images using the `forward` method.
- Collect predictions on the CPU for postprocessing.
- Convert the predictions from the YOLO v3 grid cell coordinates to bounding box coordinates to allow easy comparison with the ground truth data by using the `anchorBoxGenerator` method of `yolov3ObjectDetector`.
- Generate targets for loss computation by using the converted predictions and ground truth data. These targets are generated for bounding box positions (x, y, width, height), object confidence, and class probabilities. See the supporting function `generateTargets`.
- Calculates the mean squared error of the predicted bounding box coordinates with target boxes. See the supporting function `bboxOffsetLoss`.
- Determines the binary cross-entropy of the predicted object confidence score with target object confidence score. See the supporting function `objectnessLoss`.
- Determines the binary cross-entropy of the predicted class of object with the target. See the supporting function `classConfidenceLoss`.
- Computes the total loss as the sum of all losses.
- Computes the gradients of learnables with respect to the total loss.

```
function [gradients, state, info] = modelGradients(detector, XTrain, YTrain,
penaltyThreshold)
inputImageSize = size(XTrain,1:2);

% Gather the ground truths in the CPU for post processing
YTrain = gather(extractdata(YTrain));

% Extract the predictions from the detector.
[gatheredPredictions, YPredCell, state] = forward(detector, XTrain);

% Generate target for predictions from the ground truth data.
[boxTarget, objectnessTarget, classTarget, objectMaskTarget, boxErrorScale] =
generateTargets(gatheredPredictions,...
    YTrain, inputImageSize, detector.AnchorBoxes, penaltyThreshold);

% Compute the loss.
boxLoss = bboxOffsetLoss(YPredCell(:,[2 3 7
8]),boxTarget,objectMaskTarget,boxErrorScale);
objLoss = objectnessLoss(YPredCell(:,1),objectnessTarget,objectMaskTarget);
clsLoss = classConfidenceLoss(YPredCell(:,6),classTarget,objectMaskTarget);
totalLoss = boxLoss + objLoss + clsLoss;

info.boxLoss = boxLoss;
info.objLoss = objLoss;
info.clsLoss = clsLoss;
info.totalLoss = totalLoss;

% Compute gradients of learnables with regard to loss.
gradients = dlgradient(totalLoss, detector.Learnables);
end

function boxLoss = bboxOffsetLoss(boxPredCell, boxDeltaTarget, boxMaskTarget,
boxErrorScaleTarget)
% Mean squared error for bounding box position.
lossX = sum(cellfun(@(a,b,c,d)
mse(a.*c.*d,b.*c.*d),boxPredCell(:,1),boxDeltaTarget(:,1),boxMaskTarget(:,1),box
ErrorScaleTarget));
lossY = sum(cellfun(@(a,b,c,d)
mse(a.*c.*d,b.*c.*d),boxPredCell(:,2),boxDeltaTarget(:,2),boxMaskTarget(:,1),box
ErrorScaleTarget));
lossW = sum(cellfun(@(a,b,c,d)
mse(a.*c.*d,b.*c.*d),boxPredCell(:,3),boxDeltaTarget(:,3),boxMaskTarget(:,1),box
ErrorScaleTarget));
```

```
lossH = sum(cellfun(@(a,b,c,d)
mse(a.*c.*d,b.*c.*d),boxPredCell(:,4),boxDeltaTarget(:,4),boxMaskTarget(:,1),box
ErrorScaleTarget));
boxLoss = lossX+lossY+lossW+lossH;
end

function objLoss = objectnessLoss(objectnessPredCell, objectnessDeltaTarget,
boxMaskTarget)
% Binary cross-entropy loss for objectness score.
objLoss = sum(cellfun(@(a,b,c)
crossentropy(a.*c,b.*c,'TargetCategories','independent'),objectnessPredCell,obje
ctnessDeltaTarget,boxMaskTarget(:,2)));
end

function clsLoss = classConfidenceLoss(classPredCell, classTarget,
boxMaskTarget)
% Binary cross-entropy loss for class confidence score.
clsLoss = sum(cellfun(@(a,b,c)
crossentropy(a.*c,b.*c,'TargetCategories','independent'),classPredCell,classTarg
et,boxMaskTarget(:,3)));
end
```

Augmentation and Data Processing Functions

```
function data = augmentData(A)
% Apply random horizontal flipping, and random X/Y scaling. Boxes that get
% scaled outside the bounds are clipped if the overlap is above 0.25. Also,
% jitter image color.

data = cell(size(A));
for ii = 1:size(A,1)
    I = A{ii,1};
    bboxes = A{ii,2};
    labels = A{ii,3};
    sz = size(I);

    if numel(sz) == 3 && sz(3) == 3
        I = jitterColorHSV(I,...
            'Contrast',0.0,...
            'Hue',0.1,...
            'Saturation',0.2,...
            'Brightness',0.2);
    end

    % Randomly flip image.
    tform = randomAffine2d('XReflection',true,'Scale',[1 1.1]);
```

```
    rout = affineOutputView(sz,tform,'BoundsStyle','centerOutput');
    I = imwarp(I,tform,'OutputView',rout);

    % Apply same transform to boxes.
    [bboxes,indices] = bboxwarp(bboxes,tform,rout,'OverlapThreshold',0.25);
    bboxes = round(bboxes);
    labels = labels(indices);

    % Return original data only when all boxes are removed by warping.
    if isempty(indices)
        data(ii,:) = A(ii,:);
    else
        data(ii,:) = {I, bboxes, labels};
    end
end
end


function data = preprocessData(data, targetSize)
% Resize the images and scale the pixels to between 0 and 1. Also scale the
% corresponding bounding boxes.

for ii = 1:size(data,1)
    I = data{ii,1};
    imgSize = size(I);

    % Convert an input image with single channel to 3 channels.
    if numel(imgSize) < 3
        I = repmat(I,1,1,3);
    end
    bboxes = data{ii,2};

    I = im2single(imresize(I,targetSize(1:2)));
    scale = targetSize(1:2)./imgSize(1:2);
    bboxes = bboxresize(bboxes,scale);

    data(ii, 1:2) = {I, bboxes};
end
end

function [XTrain, YTrain] = createBatchData(data, groundTruthBoxes,
groundTruthClasses, classNames)
% Returns images combined along the batch dimension in XTrain and
% normalized bounding boxes concatenated with classIDs in YTrain
```

```
% Concatenate images along the batch dimension.
XTrain = cat(4, data{:,1});

% Get class IDs from the class names.
classNames = repmat({categorical(classNames')}, size(groundTruthClasses));
[~, classIndices] = cellfun(@(a,b)ismember(a,b), groundTruthClasses, classNames,
'UniformOutput', false);

% Append the label indexes and training image size to scaled bounding boxes
% and create a single cell array of responses.
combinedResponses = cellfun(@(bbox, classid)[bbox, classid], groundTruthBoxes,
classIndices, 'UniformOutput', false);
len = max( cellfun(@(x)size(x,1), combinedResponses ) );
paddedBBoxes = cellfun( @(v) padarray(v,[len-size(v,1),0],0,'post'),
combinedResponses, 'UniformOutput',false);
YTrain = cat(4, paddedBBoxes{:,1});
end
```

Learning Rate Schedule Function

```
function currentLR = piecewiseLearningRateWithWarmup(iteration, epoch,
learningRate, warmupPeriod, numEpochs)
% The piecewiseLearningRateWithWarmup function computes the current
% learning rate based on the iteration number.
persistent warmUpEpoch;

if iteration <= warmupPeriod
    % Increase the learning rate for number of iterations in warmup period.
    currentLR = learningRate * ((iteration/warmupPeriod)^4);
    warmUpEpoch = epoch;
elseif iteration >= warmupPeriod && epoch < warmUpEpoch+floor(0.6*(numEpochs-
warmUpEpoch))
    % After warm up period, keep the learning rate constant if the remaining
number of epochs is less than 60 percent.
    currentLR = learningRate;

elseif epoch >= warmUpEpoch + floor(0.6*(numEpochs-warmUpEpoch)) && epoch <
warmUpEpoch+floor(0.9*(numEpochs-warmUpEpoch))
    % If the remaining number of epochs is more than 60 percent but less
    % than 90 percent multiply the learning rate by 0.1.
    currentLR = learningRate*0.1;

else
    % If remaining epochs are more than 90 percent multiply the learning
    % rate by 0.01.
    currentLR = learningRate*0.01;
```

```
end

end
```

Utility Functions

```
function [lossPlotter, learningRatePlotter] =
configureTrainingProgressPlotter(f)
% Create the subplots to display the loss and learning rate.
figure(f);
clf
subplot(2,1,1);
ylabel('Learning Rate');
xlabel('Iteration');
learningRatePlotter = animatedline;
subplot(2,1,2);
ylabel('Total Loss');
xlabel('Iteration');
lossPlotter = animatedline;
end

function displayLossInfo(epoch, iteration, currentLR, lossInfo)
% Display loss information for each iteration.
disp("Epoch : " + epoch + " | Iteration : " + iteration + " | Learning Rate : "
+ currentLR + ...
    " | Total Loss : " + double(gather(extractdata(lossInfo.totalLoss))) + ...
    " | Box Loss : " + double(gather(extractdata(lossInfo.boxLoss))) + ...
    " | Object Loss : " + double(gather(extractdata(lossInfo.objLoss))) + ...
    " | Class Loss : " + double(gather(extractdata(lossInfo.clsLoss))));
end

function updatePlots(lossPlotter, learningRatePlotter, iteration, currentLR,
totalLoss)
% Update loss and learning rate plots.
addpoints(lossPlotter, iteration, double(extractdata(gather(totalLoss))));
addpoints(learningRatePlotter, iteration, currentLR);
drawnow
end

function detector = downloadPretrainedYOLOv3Detector()
% Download a pretrained yolov3 detector.
if ~exist('yolov3SqueezeNetVehicleExample_21aSPKG.mat', 'file')
    if ~exist('yolov3SqueezeNetVehicleExample_21aSPKG.zip', 'file')
        disp('Downloading pretrained detector...');
```

```
        pretrainedURL =
'https://ssd.mathworks.com/supportfiles/vision/data/yolov3SqueezeNetVehicleExamp
le_21aSPKG.zip';
        websave('yolov3SqueezeNetVehicleExample_21aSPKG.zip', pretrainedURL);
    end
    unzip('yolov3SqueezeNetVehicleExample_21aSPKG.zip');
end
pretrained = load("yolov3SqueezeNetVehicleExample_21aSPKG.mat");
detector = pretrained.detector;
end
```

References

[1] Redmon, Joseph, and Ali Farhadi. "YOLOv3: An Incremental Improvement." Preprint, submitted April 8, 2018. https://arxiv.org/abs/1804.02767.

Object Detection Using SSD Deep Learning

This example shows how to train a Single Shot Detector (SSD).

Overview

Deep learning is a powerful machine learning technique that automatically learns image features required for detection tasks. There are several techniques for object detection using deep learning such as Faster R-CNN, You Only Look Once (YOLO v2), and SSD. This example trains an SSD vehicle detector using the `trainSSDObjectDetector` function. For more information, see Object Detection.

Download Pretrained Detector

Download a pretrained detector to avoid having to wait for training to complete. If you want to train the detector, set the `doTraining` variable to true.

```
doTraining = false;
if ~doTraining && ~exist('ssdResNet50VehicleExample_20a.mat','file')
    disp('Downloading pretrained detector (44 MB)...');
    pretrainedURL = 
'https://www.mathworks.com/supportfiles/vision/data/ssdResNet50VehicleExample_20
a.mat';
    websave('ssdResNet50VehicleExample_20a.mat',pretrainedURL);
end
```

Load Dataset

This example uses a small vehicle data set that contains 295 images. Many of these images come from the Caltech Cars 1999 and 2001 data sets, available at the Caltech Computational Vision website, created by Pietro Perona and used with permission. Each image contains one or two labeled instances of a vehicle. A small data set is useful for exploring the SSD training procedure, but in practice, more labeled images are needed to train a robust detector.

```
unzip vehicleDatasetImages.zip
data = load('vehicleDatasetGroundTruth.mat');
vehicleDataset = data.vehicleDataset;
```

The training data is stored in a table. The first column contains the path to the image files. The remaining columns contain the ROI labels for vehicles. Display the first few rows of the data.

```
vehicleDataset(1:4,:)
```

```
ans = 4×2 table
```

	imageFilename	vehicle
1	'vehicleImages/image_00001.jpg'	[220,136,35,28]

2	'vehicleImages/image_00002.jpg'	[175,126,61,45]
3	'vehicleImages/image_00003.jpg'	[108,120,45,33]
4	'vehicleImages/image_00004.jpg'	[124,112,38,36]

Split the data set into a training set for training the detector and a test set for evaluating the detector. Select 60% of the data for training. Use the rest for evaluation.

```
rng(0);
shuffledIndices = randperm(height(vehicleDataset));
idx = floor(0.6 * length(shuffledIndices) );
trainingData = vehicleDataset(shuffledIndices(1:idx),:);
testData = vehicleDataset(shuffledIndices(idx+1:end),:);
```

Use `imageDatastore` and `boxLabelDatastore` to load the image and label data during training and evaluation.

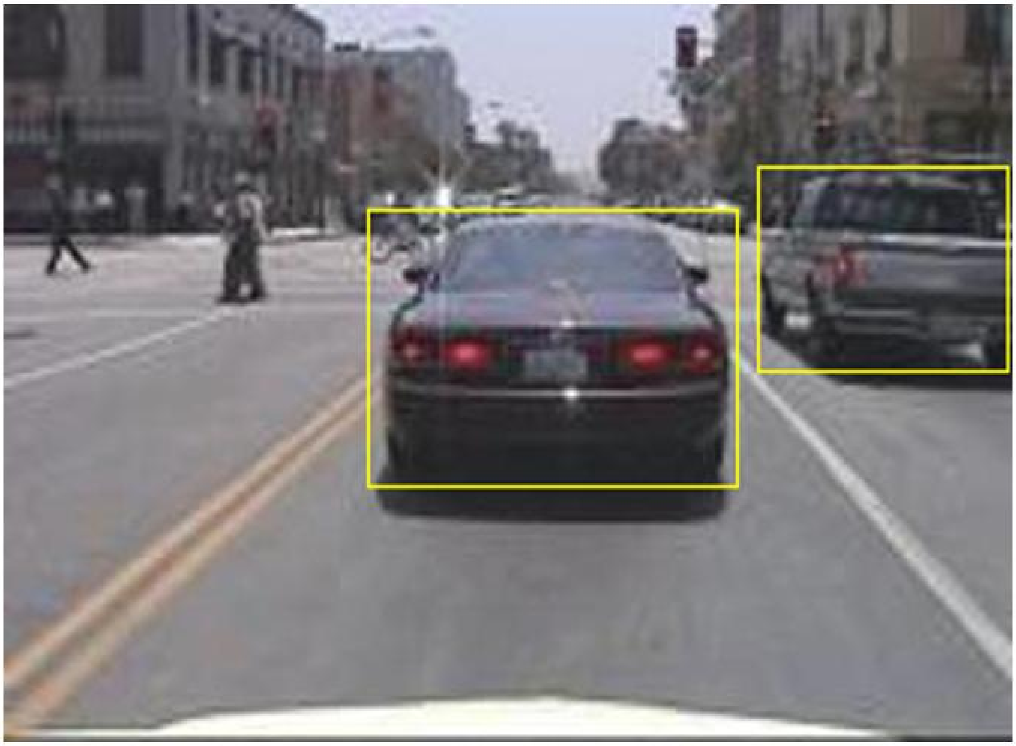
```
imdsTrain = imageDatastore(trainingData{:,'imageFilename'});
bldsTrain = boxLabelDatastore(trainingData(:,'vehicle'));

imdsTest = imageDatastore(testData{:,'imageFilename'});
bldsTest = boxLabelDatastore(testData(:,'vehicle'));
```

Combine image and box label datastores.

```
trainingData = combine(imdsTrain,bldsTrain);
testData = combine(imdsTest, bldsTest);
```

Display one of the training images and box labels.

```
data = read(trainingData);
I = data{1};
bbox = data{2};
annotatedImage = insertShape(I,'Rectangle',bbox);
annotatedImage = imresize(annotatedImage,2);
figure
imshow(annotatedImage)
```



Create a SSD Object Detection Network

The SSD object detection network can be thought of as having two sub-networks. A feature extraction network, followed by a detection network.

The feature extraction network is typically a pretrained CNN (see pretrained CNN for more details). This example uses ResNet-50 for feature extraction. Other pretrained networks such as MobileNet v2 or ResNet-18 can also be used depending on application requirements. The detection sub-network is a small CNN compared to the feature extraction network and is composed of a few convolutional layers and layers specific to SSD.

Use the `ssdLayers` function to automatically modify a pretrained ResNet-50 network into a SSD object detection network. `ssdLayers` requires you to specify several inputs that parameterize the SSD network, including the network input size and the number of classes. When choosing the network input size, consider the size of the training images, and the computational cost incurred by processing data at the selected size. When feasible, choose a network input size that is close to the size of the training image. However, to reduce the computational cost of running this example, the network input size is chosen to be [300 300 3]. During training, `trainSSDObjectDetector` automatically resizes the training images to the network input size.

```
inputSize = [300 300 3];
```

Define number of object classes to detect.

```
numClasses = width(vehicleDataset)-1;
```

Create the SSD object detection network.

```
lgraph = ssdLayers(inputSize, numClasses, 'resnet50');
```

You can visualize the network using `analyzeNetwork` or `DeepNetworkDesigner` from Deep Learning Toolbox™. Note that you can also create a custom SSD network layer-by-layer. For more information, see Create SSD Object Detection Network.

Data Augmentation

Data augmentation is used to improve network accuracy by randomly transforming the original data during training. By using data augmentation, you can add more variety to the training data without actually having to increase the number of labeled training samples. Use `transform` to augment the training data by

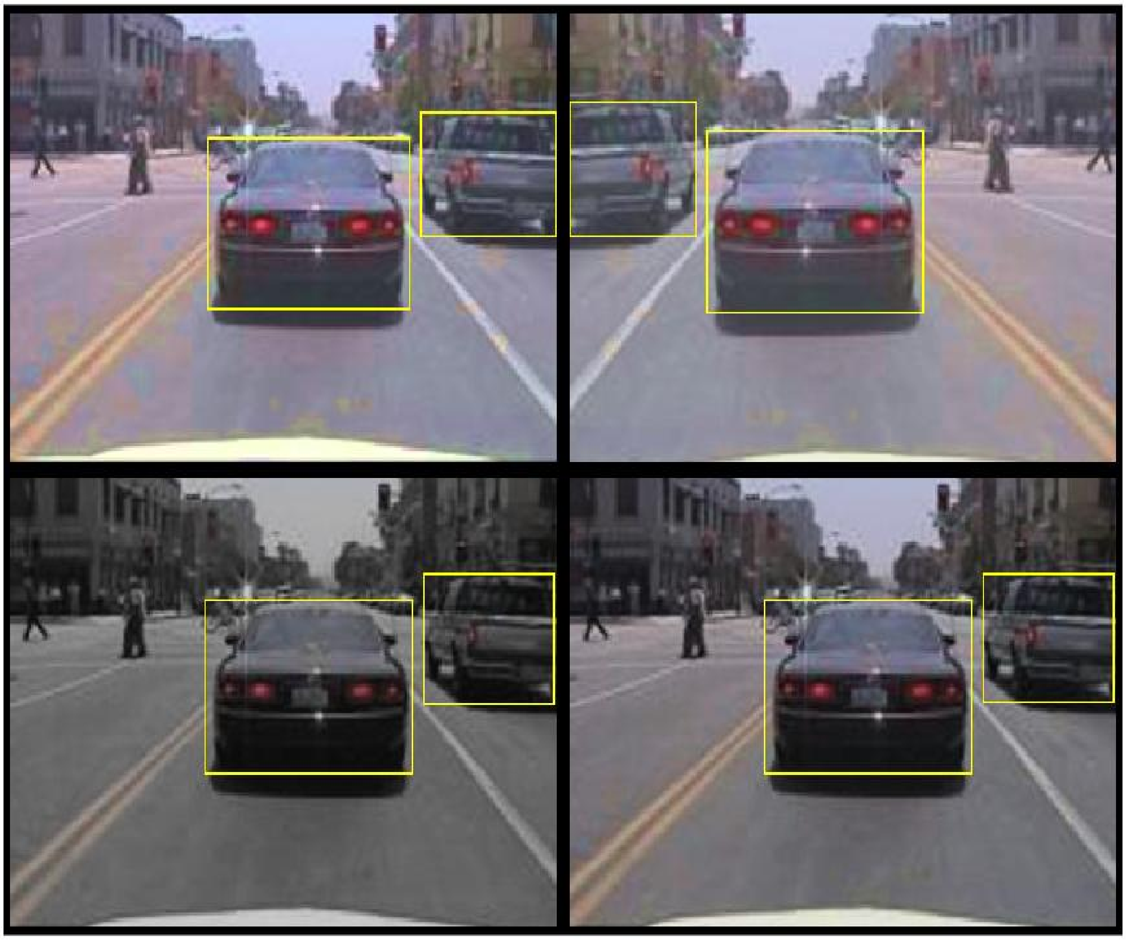
- Randomly flipping the image and associated box labels horizontally.
- Randomly scale the image, associated box labels.
- Jitter image color.

Note that data augmentation is not applied to the test data. Ideally, test data should be representative of the original data and is left unmodified for unbiased evaluation.

```
augmentedTrainingData = transform(trainingData,@augmentData);
```

Visualize augmented training data by reading the same image multiple times.

```
augmentedData = cell(4,1);
for k = 1:4
    data = read(augmentedTrainingData);
    augmentedData{k} = insertShape(data{1},'Rectangle',data{2});
    reset(augmentedTrainingData);
end

figure
montage(augmentedData,'BorderSize',10)
```



Preprocess Training Data

Preprocess the augmented training data to prepare for training.

```
preprocessedTrainingData =
transform(augmentedTrainingData,@(data)preprocessData(data,inputSize));
```

Read the preprocessed training data.

```
data = read(preprocessedTrainingData);
```

Display the image and bounding boxes.

```
I = data{1};
bbox = data{2};
annotatedImage = insertShape(I,'Rectangle',bbox);
annotatedImage = imresize(annotatedImage,2);
```

```
figure
imshow(annotatedImage)
```

Train SSD Object Detector

Use `trainingOptions` to specify network training options. Set `'CheckpointPath'` to a temporary location. This enables the saving of partially trained detectors during the training process. If training is interrupted, such as by a power outage or system failure, you can resume training from the saved checkpoint.

```
options = trainingOptions('sgdm', ...
        'MiniBatchSize', 16, ....
        'InitialLearnRate',1e-1, ...
        'LearnRateSchedule', 'piecewise', ...
        'LearnRateDropPeriod', 30, ...
        'LearnRateDropFactor', 0.8, ...
        'MaxEpochs', 300, ...
        'VerboseFrequency', 50, ...
```

```
        'CheckpointPath', tempdir, ...
        'Shuffle','every-epoch');
```

Use `trainSSDObjectDetector` function to train SSD object detector if `doTraining` to true. Otherwise, load a pretrained network.

```
if doTraining
    % Train the SSD detector.
    [detector, info] =
trainSSDObjectDetector(preprocessedTrainingData,lgraph,options);
else
    % Load pretrained detector for the example.
    pretrained = load('ssdResNet50VehicleExample_20a.mat');
    detector = pretrained.detector;
end
```

```
*************************************************************************
Training an SSD Object Detector for the following object classes:

* vehicle

在单 GPU 上训练。
正在初始化输入数据归一化。
|==========================================================================================|
|  轮  |  迭代  |     经过的时间     |   小批量损失   |  小批量准确度  |   小批量 RMSE   |  基础学习率  |
|      |        |  （hh：mm：ss）     |                |                |                 |              |
|==========================================================================================|
|    1 |      1 |       00：00：02 |     52.4445 |      47.77% |            1.96 |     0.0010 |
|    5 |     50 |       00：00：59 |      3.7708 |      99.80% |            1.14 |     0.0010 |
|   10 |    100 |       00：01：56 |      2.6134 |      99.84% |            0.88 |     0.0010 |
|   14 |    150 |       00：02：50 |      1.7091 |      99.87% |            0.67 |     0.0010 |
|   19 |    200 |       00：03：47 |      1.2035 |      99.92% |            0.48 |     0.0010 |
|   20 |    220 |       00：04：06 |      1.0571 |      99.92% |            0.49 |     0.0010 |
|==========================================================================================|
训练结束：已完成最大轮数。
Detector training complete.
*************************************************************************
```

This example is verified on an NVIDIA™ Titan X GPU with 12 GB of memory. If your GPU has less memory, you may run out of memory. If this happens, lower the '`MiniBatchSize`' using the `trainingOptions` function. Training this network took approximately 2 hours using this setup. Training time varies depending on the hardware you use.

As a quick test, run the detector on one test image.

```
data = read(testData);
I = data{1,1};
I = imresize(I,inputSize(1:2));
[bboxes,scores] = detect(detector,I, 'Threshold', 0.4);
```

Display the results.

```
I = insertObjectAnnotation(I,'rectangle',bboxes,scores);
figure
imshow(I)
```

Evaluate Detector Using Test Set

Evaluate the trained object detector on a large set of images to measure the performance. Computer Vision Toolbox™ provides object detector evaluation functions to measure common metrics such as average precision (`evaluateDetectionPrecision`) and log-average miss rates (`evaluateDetectionMissRate`). For this example, use the average precision metric to evaluate performance. The average precision provides a single number that incorporates the ability of the detector to make correct classifications (`precision`) and the ability of the detector to find all relevant objects (`recall`).

Apply the same preprocessing transform to the test data as for the training data. Note that data augmentation is not applied to the test data. Test data should be representative of the original data and be left unmodified for unbiased evaluation.

```
preprocessedTestData = 
transform(testData,@(data)preprocessData(data,inputSize));
```

Run the detector on all the test images.

```
detectionResults = detect(detector, preprocessedTestData, 'Threshold', 0.4);
```

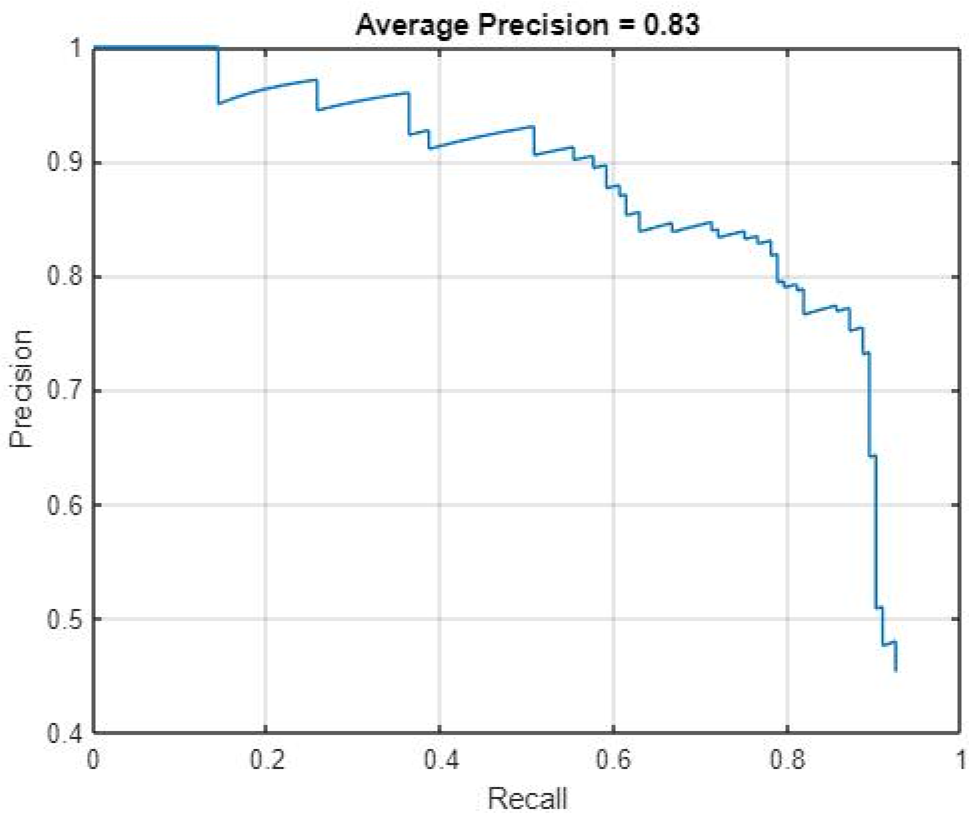
Evaluate the object detector using average precision metric.

```
[ap,recall,precision] = evaluateDetectionPrecision(detectionResults, 
preprocessedTestData);
```

The precision/recall (PR) curve highlights how precise a detector is at varying levels of recall. Ideally, the precision would be 1 at all recall levels. The use of more data can help improve the average precision, but might require more training time Plot the PR curve.

```
figure
plot(recall,precision)
```

```
xlabel('Recall')
ylabel('Precision')
grid on
title(sprintf('Average Precision = %.2f',ap))
```



Code Generation

Once the detector is trained and evaluated, you can generate code for the `ssdObjectDetector` using GPU Coder™. For more details, see Code Generation For Object Detection Using SSD example.

Supporting Functions

```
function B = augmentData(A)
% Apply random horizontal flipping, and random X/Y scaling. Boxes that get
% scaled outside the bounds are clipped if the overlap is above 0.25. Also,
% jitter image color.
B = cell(size(A));

I = A{1};
sz = size(I);
if numel(sz)==3 && sz(3) == 3
    I = jitterColorHSV(I,...
```

```
        'Contrast',0.2,...
        'Hue',0,...
        'Saturation',0.1,...
        'Brightness',0.2);
end

% Randomly flip and scale image.
tform = randomAffine2d('XReflection',true,'Scale',[1 1.1]);
rout = affineOutputView(sz,tform,'BoundsStyle','CenterOutput');
B{1} = imwarp(I,tform,'OutputView',rout);

% Sanitize boxes, if needed.
A{2} = helperSanitizeBoxes(A{2}, sz);

% Apply same transform to boxes.
[B{2},indices] = bboxwarp(A{2},tform,rout,'OverlapThreshold',0.25);
B{3} = A{3}(indices);

% Return original data only when all boxes are removed by warping.
if isempty(indices)
    B = A;
end
end

function data = preprocessData(data,targetSize)
% Resize image and bounding boxes to the targetSize.
sz = size(data{1},[1 2]);
scale = targetSize(1:2)./sz;
data{1} = imresize(data{1},targetSize(1:2));

% Sanitize boxes, if needed.
data{2} = helperSanitizeBoxes(data{2}, sz);

% Resize boxes.
data{2} = bboxresize(data{2},scale);
end
```

References

[1] Liu, Wei, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng Yang Fu, and Alexander C. Berg. "SSD: Single shot multibox detector." In 14th European Conference on Computer Vision, ECCV 2016. Springer Verlag, 2016.