

1. Transfer Learning

Pretrained Convolutional Neural Networks

A convolutional neural network (CNN) is a type of deep learning network that has been applied to image recognition tasks.

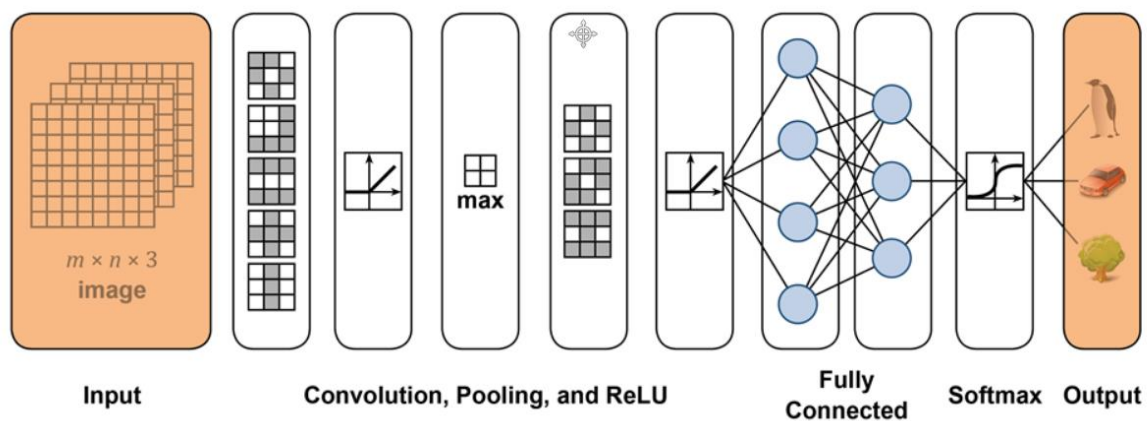
You will use the pretrained network AlexNet. This network was designed and trained by a team of deep learning researchers in 2012. AlexNet was trained for a week on a million images in 1,000 categories.

The `alexnet` function returns the AlexNet network.

```
net = alexnet
```

In MATLAB®, a deep neural network is represented as an array of layers. You can view the layers by accessing the `Layers` property of the network variable `net`.

```
layers = net.Layers
```



The AlexNet architecture, as any other deep network architecture, consists of groups of layers forming stages. These stages can be described as convolutional stages and fully connected stages. Pretrained networks usually name the individual layers to indicate the corresponding stage.

Convolutional stages extract features from images, while keeping the image in a 2-D structure. Fully connected stages use these features for classification.

Importing Pretrained Networks

There are multiple ways to import trained networks into MATLAB.

- Support packages
- Open-source formats
- MAT-files

Support Packages

Before you can use pretrained networks like AlexNet, you need to install the appropriate support package. These support packages are freely available in the Add-On Explorer.

To determine if you have a network installed, you can try to load the network.

```
net = googlenet
```

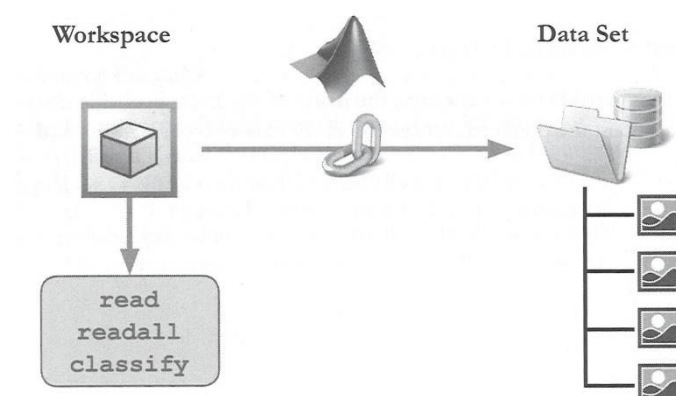
If the network is not already installed, you will receive an error with a link to install the network.

Open-Source Formats

You can also import networks created using the open-source formats such as Caffe and Keras. You can use the corresponding functions `importCaffeNetwork` and `importKerasNetwork`.

To view available pretrained networks, refer to the documentation:

```
doc('Pretrained Convolutional Networks')
```



Preprocessing an Image

You can import and view an image of any standard format by using `imread` and `imshow`.

```
im = imread(filename)

imshow(im)
```

To use a pretrained network for image classification, you have to preprocess the images to the required format.

CNNs begin with an image input layer. This layer specifies the size of an input image. For example, AlexNet requires images to be size 227-by-227-by-3.

You can reshape images to the expected size using the `imresize` function.

```
imRes = imresize(im, [227 227])
```

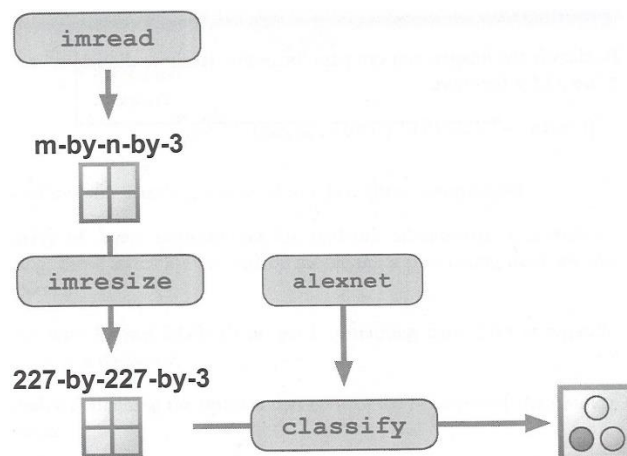
Preprocessing image can also involve modifications such as cropping, filtering, adjusting contrast, converting from an indexed image to an RGB image, etc.

Once the image has the required size, you can use pretrained network to classify the image.

```
pred = classify(net, imRes)
```

The output variable `pred` contains the name of the most probable class. You can obtain the predicted scores for all the classes by using a second output.

```
[pred, scores] = classify(net, imRes)
```



Using Image Datastores

So far, you have imported each image into memory individually, and then used AlexNet to classify it. This is useful for one or two images, but deep learning can involve thousands of images.

MATLAB can also read image files by creating a datastore, which references a data source such as a folder of image files. When you create a datastore, basic meta information like the file name and formats is stored.

The datastore does not import the data into memory until it is needed. This allows your data set to contain more images than can fit in memory at once. You can read and process the images incrementally.

Use the `imageDatastore` function to create a new image datastore, providing the source location as the input.

```
imds = imageDatastore('images')
```

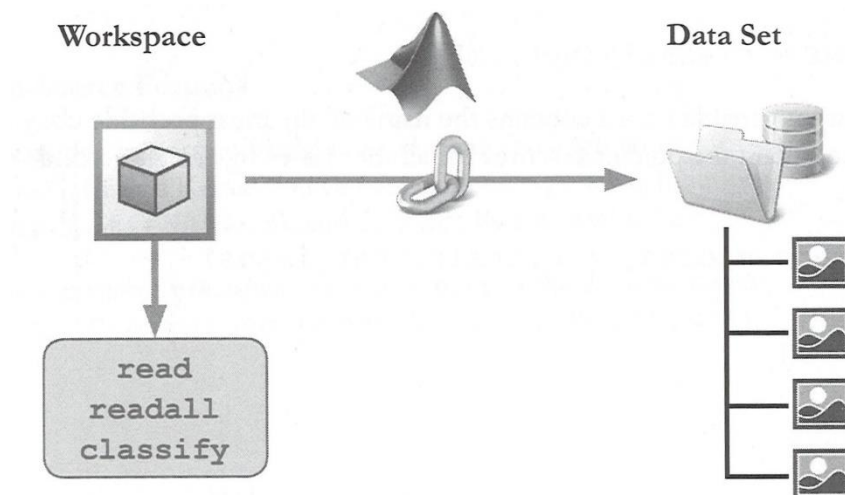
An image datastore imports images with no modifications. If you want to resize an entire collection of images, you can create an augmented image datastore.

```
auds = augmentedImageDatastore([227 227], imds)
```

An augmented image datastore can also convert images to grayscale or RGB.

To classify the images, you can pass the entire datastore directly to the `classify` function.

```
preds = classify (net, auds)
```



Transfer Learning

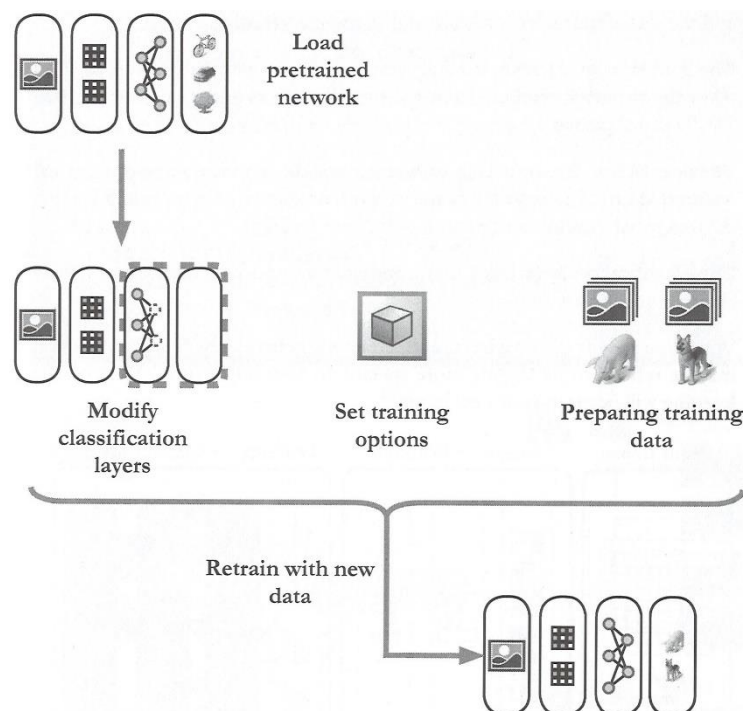
Pretrained networks classify images into predetermined categories. Your data set may contain categories not present in an existing network. In these cases, you cannot directly use AlexNet in your application.

It is possible to build and train a network yourself. A brand new network can be customized to your application, but it is initialized with random weights. Achieving reasonable results requires a new architecture, a large amount of training data, and the computational resources to train a new network.

Rather than starting from scratch, you can modify a pretrained network to fit your problem. Pretrained networks have learned rich feature representations for a wide range of images. This process of taking a pretrained network, modifying it, and retraining it on new data is called transfer learning.

To perform transfer learning, you need to create three components:

- An array of layers representing the network architecture. For transfer learning, these layers are created by modifying a preexisting network like AlexNet.
- Images with known labels to be used as training data. This is typically provided as a datastore.
- A variable containing the options that control the behavior of the training algorithm.



Modifying a Pretrained Network

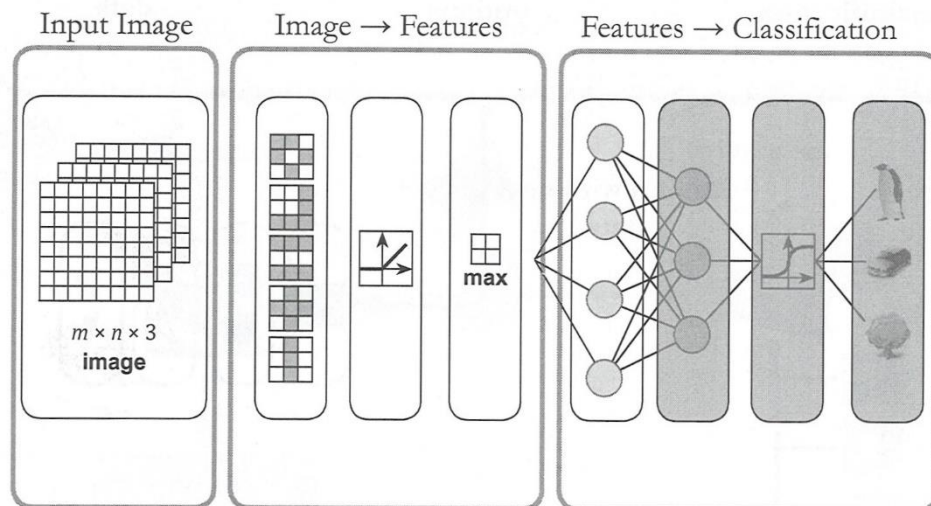
Most layers in a pretrained network extract features from an input image. The final layers map these features to your image classes. When performing transfer learning, you will typically just change the last fully connected layer and the classification layer to suit your specific application.

The 23rd layer of AlexNet is a fully connected layer with 1,000 neurons. This takes the extracted features from the previous layers and maps them to the 1,000 output classes.

The next layer is the softmax layer, which turns the raw values into normalized scores that can be interpreted as the network prediction of the probability that the image belongs to that class.

The classification layer takes these probabilities and returns the most likely class as the network output.

When you modify the final layers and retrain the network, the feature extraction may be refined to be slightly more specific to your application. Most of the learning will occur in your new layers.



You can replace the fully connected layer by creating a new layer. This layer accepts the number of output classes as input. In AlexNet, you want to replace is the 23rd layer because this is the last fully connected layer.

```
layers(23) = fullyConnectedLayer(10)
```

To replace the classification layer, you can create a new output layer. The number of output classes will be determined from the previous layer in the architecture, so the `classificationLayer` function does not need any inputs.

```
layers(end) = classificationLayer()
```

Preparing Training Data

When training a network, you need to provide known labels for the training images.

If your images are organized by folder, you can label your images in an image datastore using the folder name.

```
imds = imageDatastore('images', ...
    'IncludeSubfolders', true, ...
    'LabelSource', 'foldernames')
```

Before training, you should split your collection of images into two groups: one to train the network and one to test the network performance.

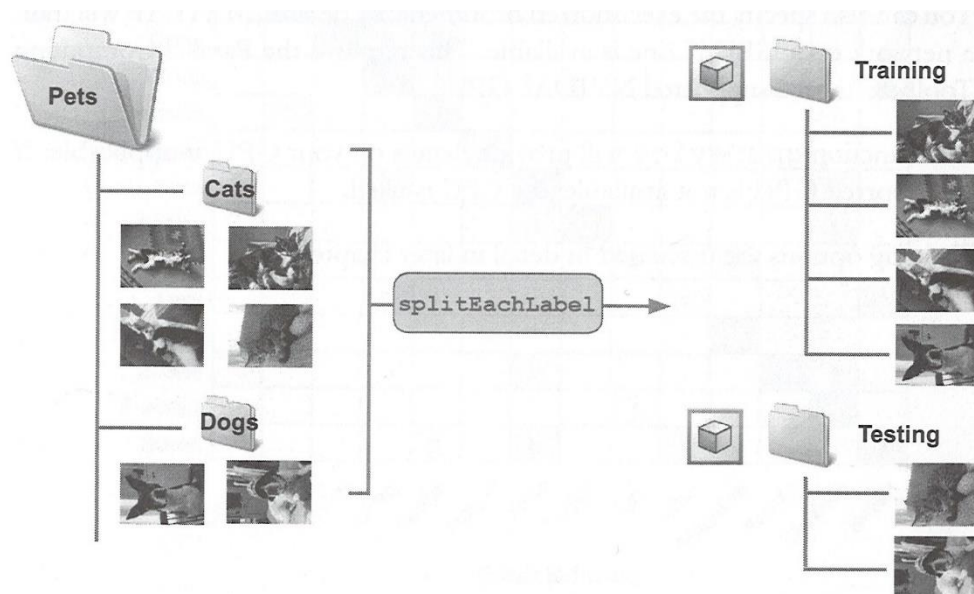
It is important for a network to have high training and testing accuracy. A test data set should represent the images a network will encounter after it is deployed. You should test a network with images the network did not see during training. This allows you to check that your network will generalize to new images.

You can use the `splitEachLabel` function to split a datastore using a proportion or number of images.

```
[trainImgs , testImgs] = splitEachLabel (imds, 0.8)
```

```
[trainImgs , testImgs] = splitEachLabel (imds, 50)
```

You can request some of your images from each label to be dedicated to training. The rest of the images can be used for testing. These datastores are respectively named `trainImgs` and `testImgs`.



Setting Training Options

Training options control the network behavior while it is trained. First, you should choose the algorithm. There are many algorithms available, but they are all used to minimize a loss function. For example, you can use stochastic gradient descent with momentum(`sgdm`).

```
opts = trainingOptions ('sgdm')
```

There are many more training options available, such as initial learning rate and maximum number of times to sweep through the data during training. These options can be set using Name-Value pairs.

```
opts = trainingOptions ('sgdm',  
    'InitialLearnRate', 0.005)
```

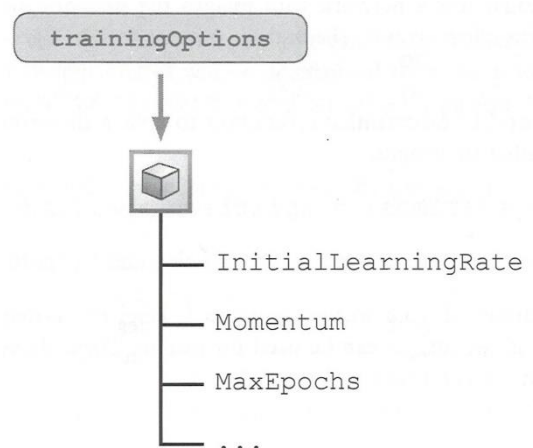
You can see more options in the documentation.

```
doc trainingOptions
```

You can also specify the execution environment. By default, MATLAB will train a network on a GPU if one is available. This requires the Parallel Computing Toolbox™ and a supported NVIDIA® GPU.

The function `gpuDevice` will provide details on your GPU, if applicable. If a supported GPU is not available, the CPU is used.

Training options are discussed in detail in later chapters.



Training and Evaluating the Network

You will pass all three components to the `trainNetwork` function to train a new network.

```
newnet = trainNetwork (data, layers, options)
```

You should test the performance of the newly trained network. If it is not adequate, typically you will try adjusting some of the training options and then retraining.

To evaluate the network performance, you look to the test of images. You have the true labels of each image, and you can classify them with your network to see if the network has learned. You may also do this with the training set to see how the fraction of misclassified images matches the test set.

```
labels = imTest.Labelsc
```

```
preds = classify(net, imTest)
```

The true labels are in `labels` and the predicted labels are in `preds`. You can count the number of correct predictions and the accuracy to see how well your network has learned.

```
numCorrect = nnz(labels == preds)
```

```
accuracy = numCorrect/numel(preds)
```

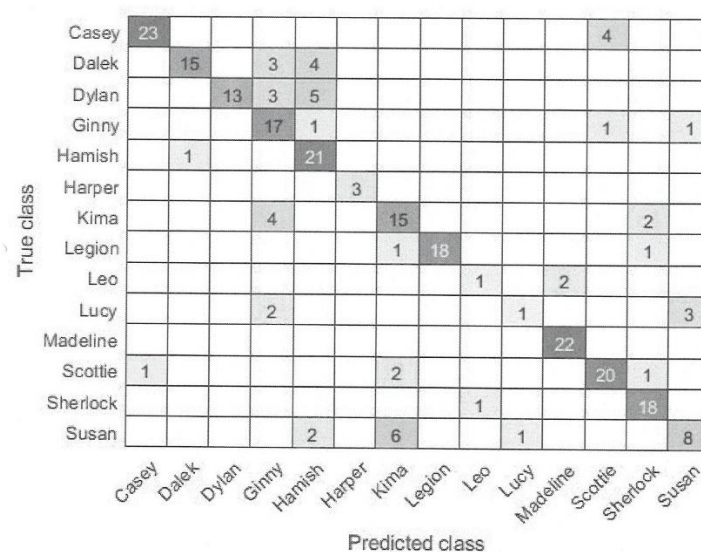
If your network makes some errors, it is a good idea to see what kind of errors they are. Dealing with multiple labels, there are different kinds of misclassifications that can be made. You can see these mistakes laid out in what is called a confusion matrix, by using the function `confusionchart`.

```
confusionchart(labels, preds)
```

Other kinds of performance statistics can be extracted by using a second output with `trainNetwork`, like training a accuracy and loss.

```
[newnet, info] = trainNetwork(data, layers, opts)
```

```
plot(info.TrainingAccuracy)
```



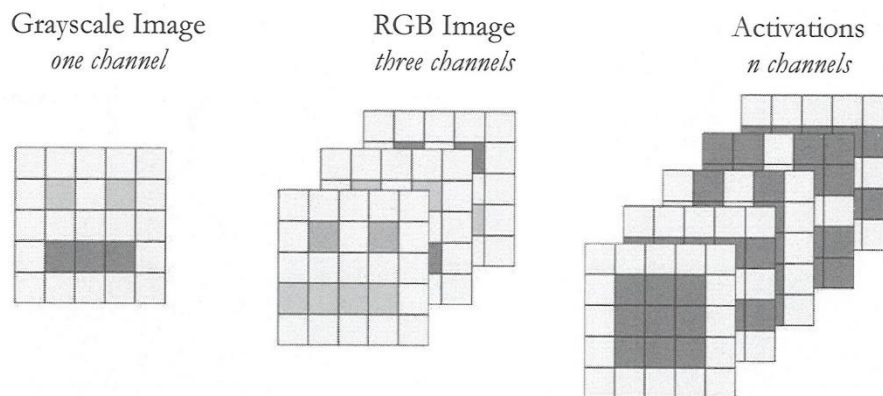
Activations

Neural networks are made of layers that take information from the previous layer, manipulate it, and pass the result to the next layer. In the case of CNNs, information is passed between layers as a collection of 2-D arrays. Each of these arrays can be visualized as a grayscale image, and together can be thought of as a set of features used to represent the original image. You can extract these features with the `activations` function.

```
features = activations(net, im, layer)
```

`net` is the pretrained network, `im` is an input image, and `layer` is the layer to extract the features from. The output is a 3-D numeric array, where the third dimension is often called a *channel*.

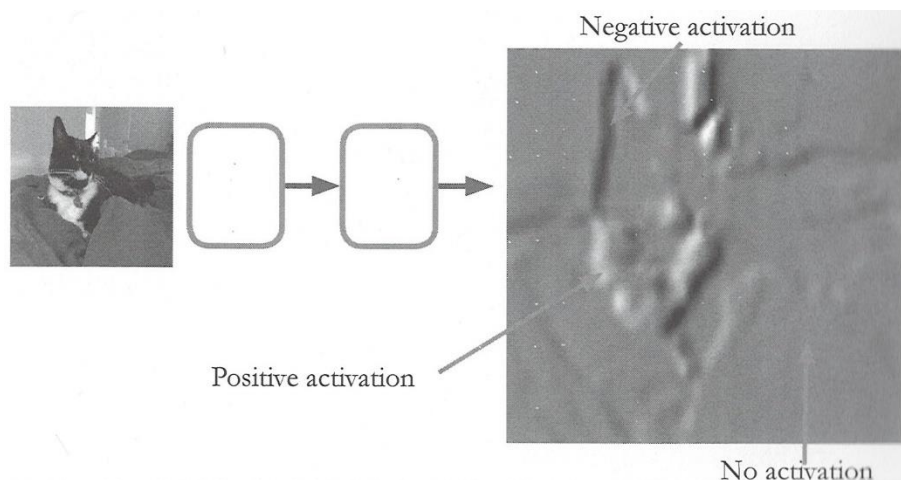
Many images have multiple channels. For example, RGB image have three (red, green, and blue).



There is one output channel for each filter in a convolution layer. A convolution layer can have hundreds of filters, so each layer can create hundreds of channels. You can visualize each channel as a grayscale image.

A white pixel indicates that the channel is positively activated at that position.

A black pixel is where the channel activated negatively.



Deeper Layer Activations

Most convolutional neural networks learn to detect features like color and edges in their first convolutional layer. In deeper convolutional layer, the network learns to detect more complicated features. Later layers build up their features by combining features of earlier layers.

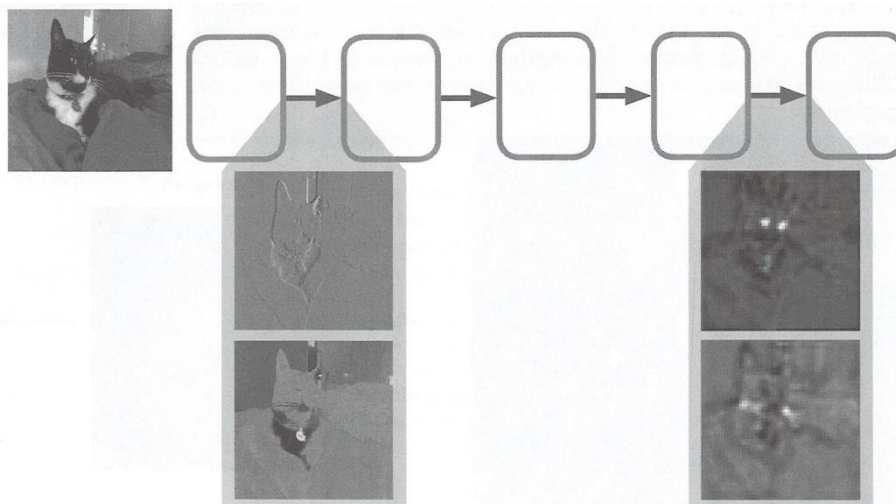
A deep network like AlexNet has never been told to learn specific features, but it often finds recognizable features. AlexNet often detects eyes in an image of a face. During training, the network learned that eyes are a useful feature to distinguish between classes of images.

For example, learning to identify eyes could help the network distinguish between a leopard and a leopard print rug.

It can also be useful to view all activations at once using the `imtile` function. Each activation can take any value, so you should normalize the activations before you display them with the `rescale` function. You can then display all the activations from one layer.

```
normalized = rescale(features);
```

```
imshow(imtile(normalized))
```



Following Activations in the Network Architecture

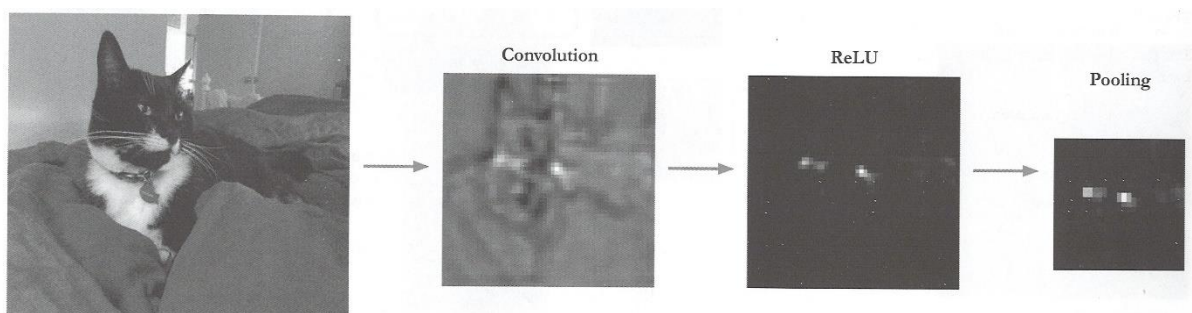
You have looked at specific activations at the first and last convolutional layers in AlexNet.

You can also track the evolution of an image through the network to see how the image is modified at each layer.

In the first convolution layer, the input image is still mostly recognizable. As the image progresses through the network, it will look less like the input. This is because the network learns a smaller set of features to represent the original image.

Rectified linear unit (ReLU) layers apply a threshold to an input such that every value less than zero is set to zero. You will see that any negative activations from a convolution layer will be set to zero after passing through the ReLU layer.

Max pooling layers perform downsampling by dividing the input into rectangular pooling regions and computing the maximum of each region. This cause large activations to be more pronounced after passing through the pooling layer.



Feature Extraction for Machine Learning

It is difficult to perform deep learning on a computer without a GPU because of the long training time. An alternative is to use the activations from pretrained networks as image features. You may then use traditional machine learning methods to classify these features.

Machine Learning

Traditional machine learning is a broad term that encompasses many kinds of algorithms. These algorithms learn from predictor variables. Instead of using an entire image as the training data for a model, you need a set of features. Features can be anything that accurately describes your data set. For example, features to describe dogs could be coat colors, pattern, and size.

Image Features in Pretrained Networks

CNNs learn to extract useful features while learning how to classify image data. As you have seen, the early read an input image and extract features. Then, fully connected layers classify these features.

Extracting Features

With deep learning, you can use the activations function to extract features. These features can be used as the predictor variables for machine learning.

The rest of the machine learning workflow is very similar to that of deep learning:

1. Extract training features.
2. Train model.
3. Extract test features.
4. Predict test features.
5. Evaluate model.

You can get activations from an entire datastore of images at once. The activations need to be stored as rows to train a machine learning model. This makes each row correspond to one image, and each column one feature. You can set the `OutputAs` option to accomplish this.

```
Features = activations (net, images, layer, ...
    'OutputAs', 'rows')
```

Extract features from the training and testing images stored in the datastore. You can get features from any layer in a network. For AlexNet, the fc7 layer is often used.

Applying Traditional Machine Learning

You will use the extracted features and the training image labels to create a k -nearest neighbors model. One way to categorize images is to classify a new image based on the pairwise distance between its features and the nearest image features. Here you have to think of the features as being a point in space. The k -nearest neighbors (k -NN) algorithm uses the majority vote from k neighboring points to determine the class of the unlabeled data. You can train a k -nearest neighbors classifier using the `fitcknn` function

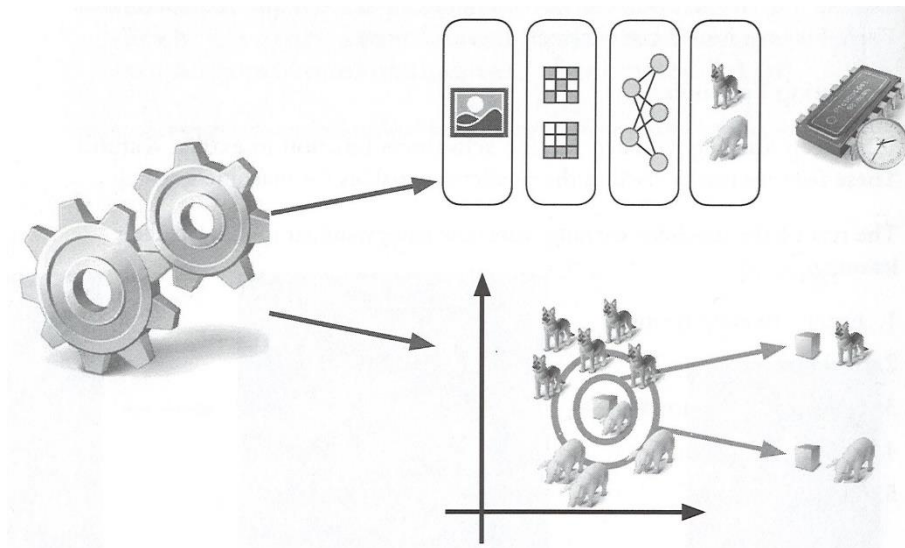
```
classifier = fitcknn (trainfeatures, labels)
```

where `trainfeatures` is the output matrix from the `activations` function on the training data, and `labels` is from the `labels` property of the image datastore. Your features are now predictor variables and your image labels are your response variable.

```
pedLabels = predict (classifier, testfeatures)
```

As with deep networks, you can evaluate the performance of a machine learning model using a confusion matrix.

```
confusionchart (known, predicted)
```



Creating and Training New Networks

Transfer Learning Limitations

Transfer learning allows you to leverage the work of deep learning researchers who have access to large amounts of data and computational resources. This gives you a significant head start on training because the network parameters have already been tuned to extract features for a wide variety of images.

However, transfer learning works only if you don't need to modify the early layers of the network. Because information flows forward from layer to layer, if you change the input layer of your network, the downstream architecture and parameters are no longer valid. In this situation, you can't use transfer learning.

Different Types of Images

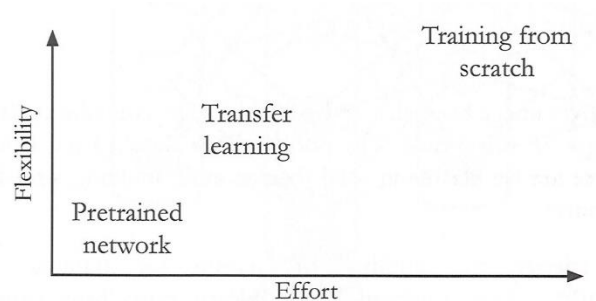
Most pretrained networks expect inputs to be color images of a specific size. If you have grayscale images, or color images of a slightly different size, you can always resize your images to match the expected input size.

But what if your images are a significantly different size? Or your images have more than three channels, like infrared, sonar, or seismic readings? You won't be able to force your input into a size these pretrained networks expect without losing potentially important information.

Training from Scratch

If you have enough training data, the best solution might be to train a network from scratch. The primary difference from transfer learning is that you need to create the array of layers, rather than importing a network as a starting point.

How do you know what layers to use, and in what order? Rather than trying to design your own deep network, you can take advantage of common architectures developed by experts over many iterations of trial and error. This approach starts with an existing architecture, but trains the weights and biases from scratch.



Creating a Network from Layers

Your new network is going to be much shallower than AlexNet, but the structure applies the same ideas. The layers in this architecture are common to most pretrained networks.

You can create a column vector of seven layers with the functions listed below.

```
layers = [  
    imageInputLayer(inputSize)  
    convolution2dLayer(filterSize, numFilters)  
    reluLayer()  
    maxPooling2dLayer(poolSize)  
    fullyConnectedLayer(numClasses)  
    softmaxLayer()  
    classificationLayer()  
]
```

Each land cover image has a size 28-by-28-by-4. The convolution layer should have 20 filters of size 3-by-3. The pooling layer should have a pool size of 3-by-3. There are six classes in total (barren land, building, grassland, road, trees, and water).

You can specify any number of setting for training using the `trainingOptions` function. You will learn more setting training options in a later chapter.

Once you have created a network architecture, you can confirm the network is valid using `analyzeNetwork`. Use your created layers as input to the function to visualize the architecture and detect errors.

```
analyzeNetwork(layers)
```


Neurons in a Network Layer

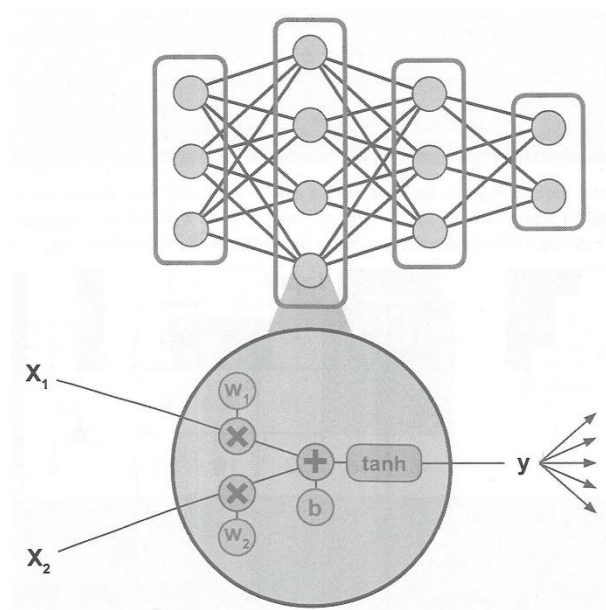
Many layers in deep networks are specific to deep learning applications, but these layers are derived from classical neural networks.

A classical neural network uses fully connected layers. Every neuron in a layer is connected to every neuron in the next layer. The information passed along each connection is a single number.

Inside a neuron, the input values are multiplied by parameters known as weights. The results are added, along with another parameter known as the bias. The sum is passed to a function called the transfer function to get the output value.

If you change the parameters – the weights and biases – then the output from the neuron will be different. These parameters govern the behavior of the neurons and, collectively, the whole network. The training process involves adjusting these parameters so that the network produces the desired output for known samples.

The **Weights** property of a layer contains the weights of the layers. Weights are learned during the training process. Pretrained network have already learned weights and biases for image classification.



Convolutional Layer

In contrast to fully connected layers, a convolution layer applies sliding filters to the input image. More details on the convolution operation are on the next page.

A convolution will usually apply several filters to the input image. The number of filters is specified with the `numFilters` input.

Each filter must have the same size, specified by `filterSize`. Ideally, each filter will learn to detect different features.

```
layer = convolution2dLayer(filterSize, numFilters)
```

Input						Filter			Output			
7	14	-9	-2	5	*	1	0	=	2	11	-5	4
13	-5	-3	4	6		0	1		9	-2	7	16
-6	-4	3	10	12					-4	5	14	3
0	2	9	11	-7					8	17	1	10
1	8	15	-8	-1								

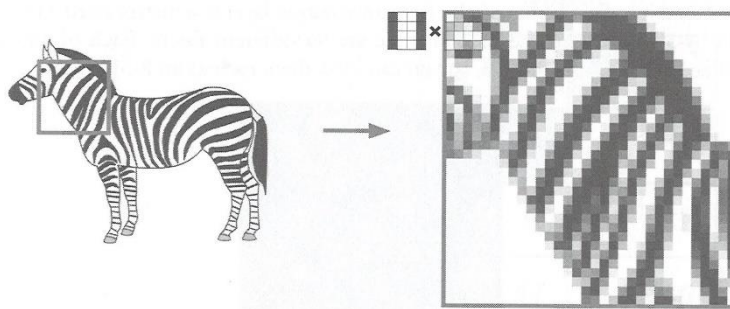
Performing Convolutions

Convolution highlights parts of images that match a filter. The filters are represented in a convolution layer's weights.

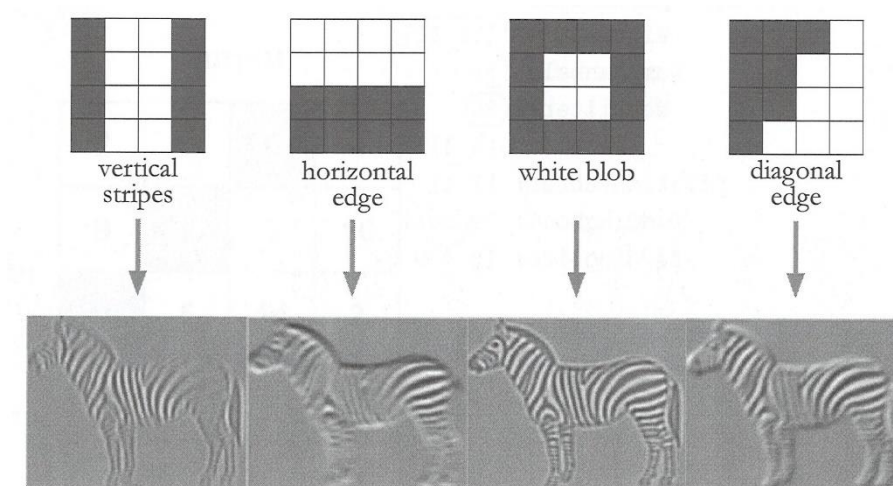
A small piece of the image is multiplied with the filter, resulting in a 2-D matrix. All the values of this 2-D matrix are added to get a scalar value. This value can be interpreted as how closely that small piece of the image matches the pattern of the filter.

$$\begin{bmatrix} -1 & 1 & 1 & -1 \\ -1 & 1 & 1 & -1 \\ -1 & 1 & 1 & -1 \\ -1 & 1 & 1 & -1 \end{bmatrix} \times \begin{bmatrix} 1 & 0.5 & -0.4 & -0.5 \\ 0.4 & 0.6 & 0.9 & 0.1 \\ 0.6 & 1 & 1 & 1 \\ 1 & 0.6 & 0.1 & 0.6 \end{bmatrix} = \begin{bmatrix} -1 & 0.5 & -0.4 & 0.5 \\ -0.4 & 0.6 & 0.9 & 0.1 \\ -0.6 & 1 & 1 & -1 \\ -1 & 0.6 & 0.1 & -0.6 \end{bmatrix} = 0.1$$

This filter is moved over the image until it is scanned completely. The mathematical operation is called a *convolution*. A value is calculated for each position, and the result is a new image – the activation – that highlights the presence of a pattern in the input image.



A convolutional layer has many filters, so the layer takes a given input image and returns a set of new images, one for each filter. You can think of each filter as performing the job of detecting a particular feature in the input image.



Viewing Learned Filters

Consider the first convolution layer in AlexNet and its properties shown below.

Hyperparameters are parameters that are set when a layer is created. Recall that the first two inputs to `convolution2dLayer` is the filter size and the number of filters. ‘NumChannels’ is the number of channels of the input to this convolution layer. In ‘conv1’, the input the image sized 227-by-227-by-3, so the number of channels is three.

The *learnable* parameters are updated during training, but the size of these arrays is calculated from the filter size, number of input channels, and the number of filters.

`convolution2dLayer` with properties:

Name: ‘conv1’

Hyperparameters

FilterSize: [11 11]

NumChannels: 3

NumFilters: 96

Stride: [4 4]

DilationFactor: [1 1]

PaddingMode: ‘Manual’

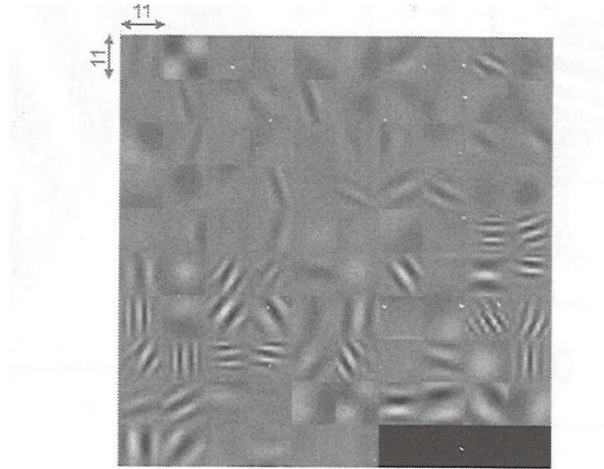
PaddingSize: [0 0 0 0]

Learnable Parameters

Weights: [11x11x3x96, single]

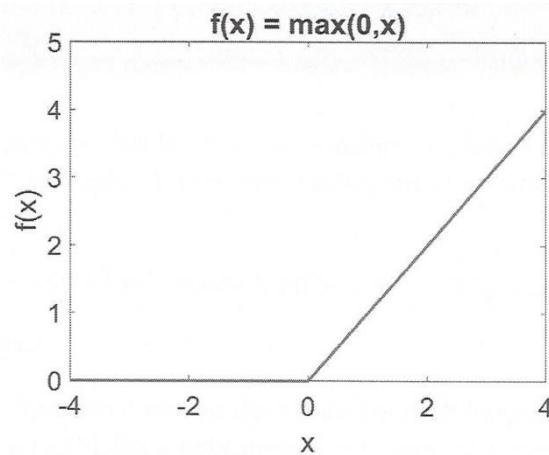
Bias: [1x1x96, single]

In AlexNet, the weights of the first convolution layer is a matrix sized 11-by-11-by-3-by-96. This means that there are 96 different filters. Each of those filters is size 11-by-11-by-3, so you can view them each as an RGB image.



Rectified Linear Unit Layer

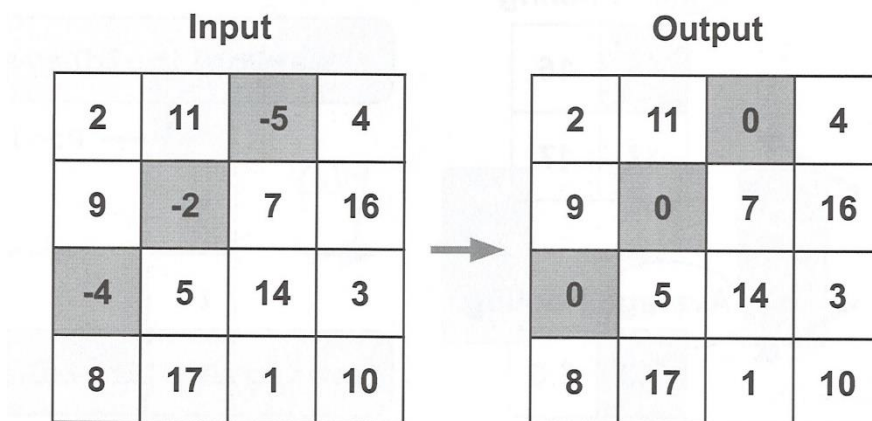
Convolutional layers are usually followed by a nonlinear activation layer such as a rectified linear unit (ReLU). A ReLU layer performs a threshold operation on each element of the input. Any value less than zero is set to zero.



The ReLU layer does not change the size of its input.

ReLU layers are commonly used in deep networks because they are fast to compute.

```
layer = reluayer ( )
```



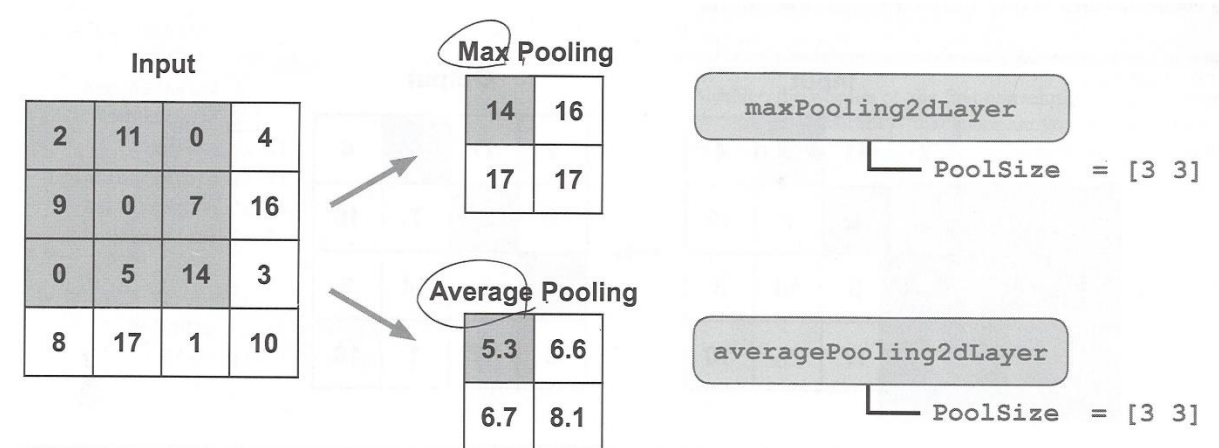
Pooling Layer

A pooling layer performs downsampling by dividing the input into rectangular pooling regions and computing the maximum (max pooling) or average (average pooling) of each region.

Pooling reduces the network complexity and create a more general network.

```
layer = maxPooling2dLayer (poolSize )
```

```
layer = averagePooling2dLayer (poolSize)
```



Output Layers

Fully Connected Layer

Features passing through the network are stored in a collection of matrices until they reach the fully connected layer. At this layer, the input is “flattened” so that it can be mapped to the output classes. This layer is a classical neural network.

The output size for this layer is the number of classes for your classification problem. For example, if you were classifying dogs and cats, the output size would be two.

```
layer = fullyConnectedLayer (outputSize)
```

Softmax Layer

The softmax function converts the values for each output class into normalized scores using a normalized exponential function. You can interpret each value as the probability that the input image belongs to each class.

```
layer = softmaxLayer ( )
```

Output Layer

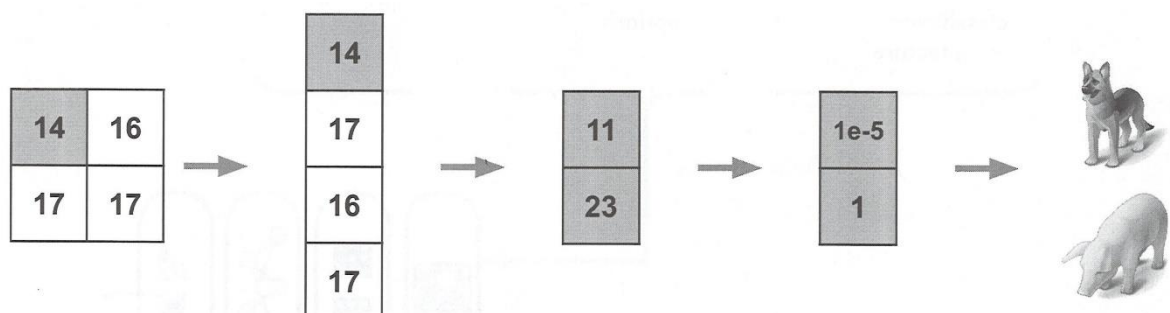
The classification output layer returns the name of the most likely class.

```
layer = classificationLayer ( )
```

Others Layers

More information on the layers can be found in the documentation:

`doc('Deep Learning Training From Scratch') > Layers`



Training the Network

Once you have prepared your training data, architecture, and training options, you can train your network. If your images and responses are not stored in a datastore, you can provide them as two separate inputs to `trainNetwork`.

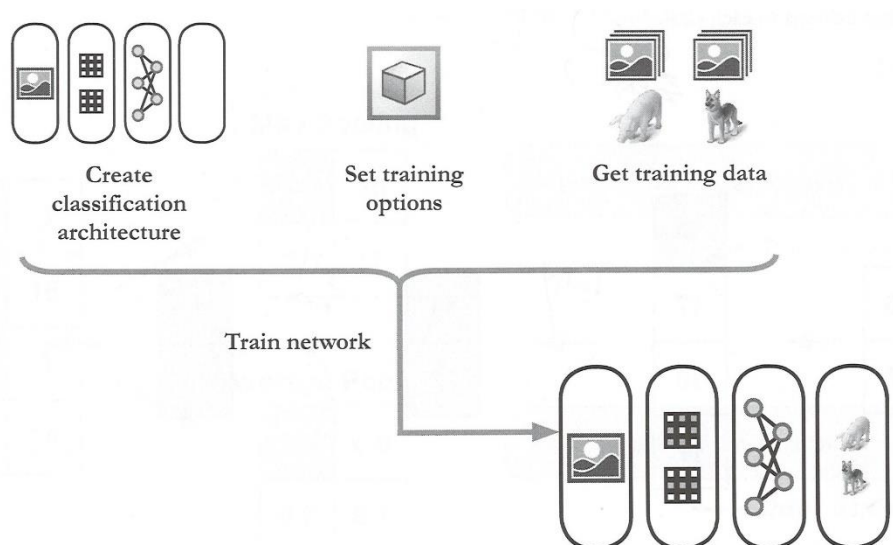
```
net = trainNetwork (X, Y, layers, options)
```

Once the network is trained, you can predict labels for new images using the `classify` function.

```
predictions = classify (net, testdata);
```

You can create a confusion matrix using the `confusionchart` function.

If you train the network multiple times, randomness in the algorithm will cause your network to perform slightly different. The accuracy should remain about the same.



Configuring Network Training

You can set training options with the `trainigOptions` function.

```
options = trainingOptions (solverName, ...)
```

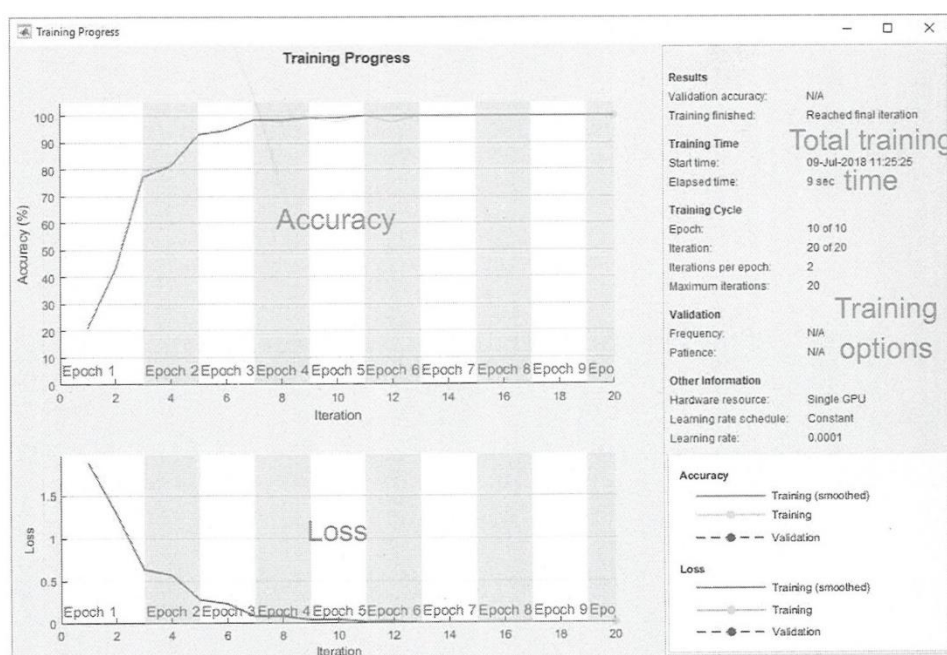
The `solverName` argument is the name of the training method used.

Solver Name	Algorithm
'sgdm' – Stochastic gradient descent with momentum	Minimize the loss function by taking small steps in the direction of the negative gradient of the loss, using momentum to reduce oscillation
'rmsprop' – Root mean square propagation	Use learning rates that are different for different parameters
'adam' – Adaptive moment estimation	Similar to root mean square propagation with momentum

You can monitor the training by setting the `s 'Plots'` property in the `trainingOptions` function to `'training-progress'`. While the network is training, a separate window will open up that provides insight into the training progress.

Accuracy shows the percentage of training images that the network classified correctly during an iteration. Accuracy does not measure how confident the network is about each prediction. It is better if the network predicts the correct class with 90% confidence 52% confidence.







Loss is a measure of how far from a perfect prediction the network was, totaled over the set of training images.



Understanding Network Training

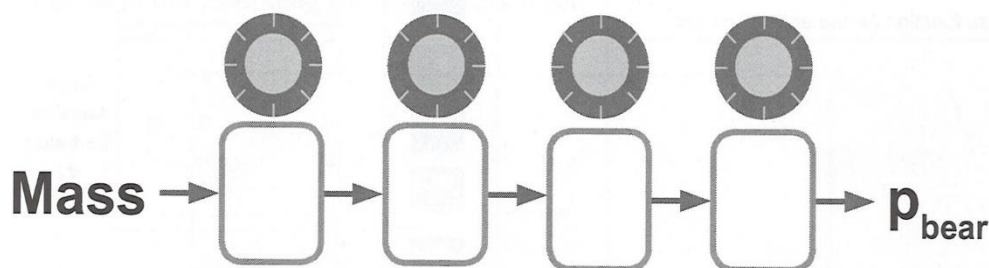
You've seen that you need to provide three things to train a deep network: data, network architecture, and training algorithm settings.

The data provides a set of example inputs with their corresponding outputs.

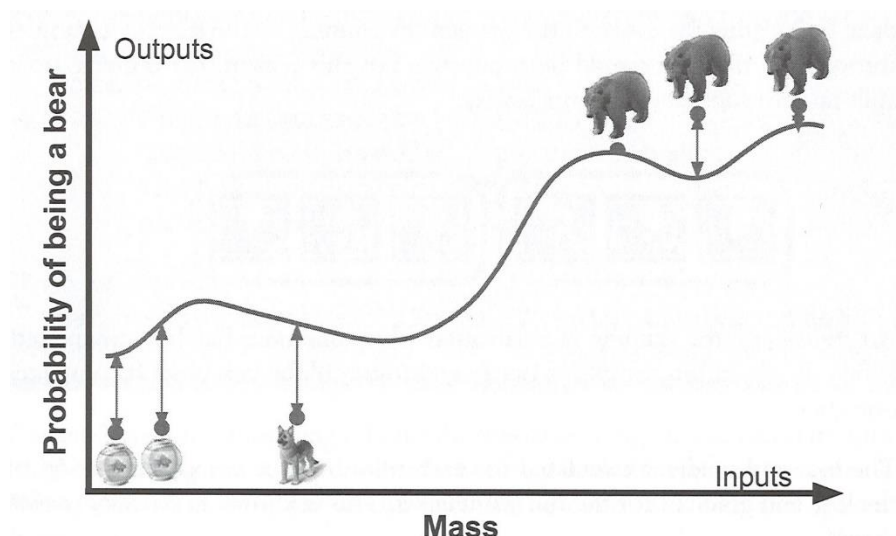
	Mass	Bear?
	0.9	0
	1.2	0
	25	0
	440	1
	600	1
	650	1

The network architecture provides a general framework for the math behind how the network maps those inputs to the outputs.

But the network has many tunable parameters – the weights and biases on the layers. Depending on the choice of parameters, the same architecture can map a set of inputs to most sets of outputs.



When you train a network, you use the example inputs and outputs to determine the values of the parameters. This is done by looking for the parameters that minimize the error between the network's predictions and the known outputs.



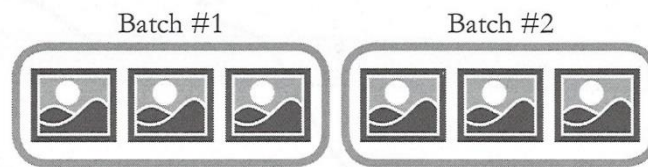
The measure of error used in training the network is called the loss function. For a given set of parameters, you can calculate the loss, but you can't calculate the loss for every possible set of parameters. How do you find where the loss is the smallest?

Training works by starting at random parameter values, and refining them. This refinement of parameters is referred to as back propagation. Instead of using the parameter values to determine the loss, you use the calculated loss to determine the parameters.

The descent gradient descent method performs back propagation by updating the parameters a small amount in the direction of the gradient, or the direction that the loss function decreases the fastest.

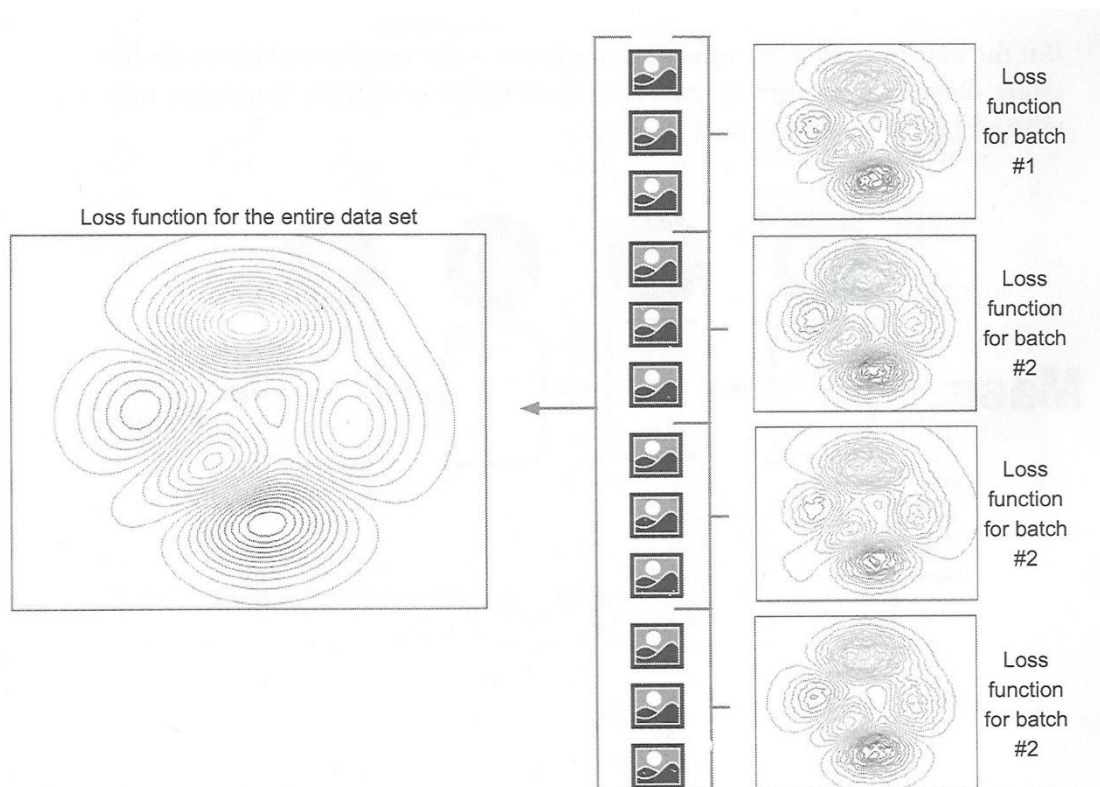
Mini-Batches

To get good results with deep networks, you generally need a lot of training data. Calculating the loss and the gradient by running all the training examples through the network would be expensive. For this reason, the training set is split up into subsets *mini-batches*.



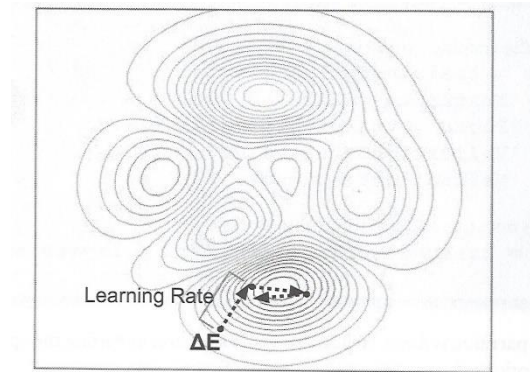
An *iteration* of the training is performed when one mini-batch is completed. When the algorithm completes one pass through all the iterations, it completes one *epoch*.

The loss and gradient calculated for each mini-batch is an approximation of the loss and gradient for the full training set. This is known as *stochastic gradient descent*.

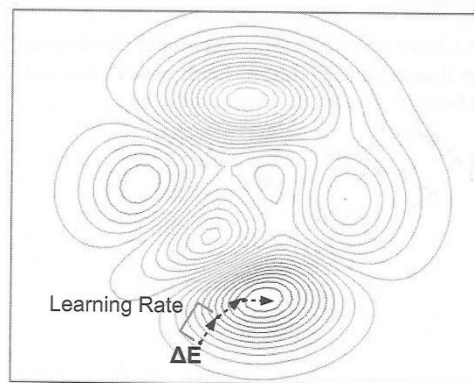


Learning Rate

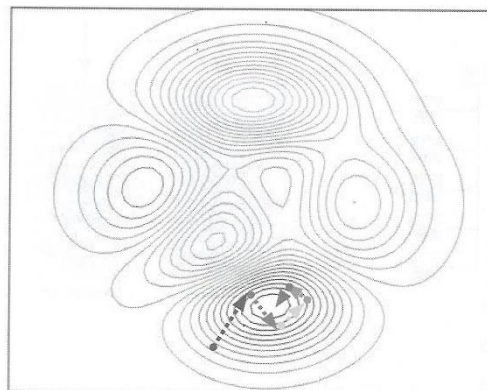
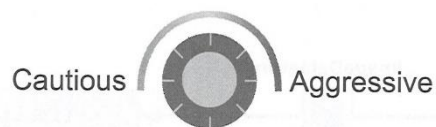
Gradient descent takes a step in the direction of the gradient. The size of that step is known as the *learning rate*.



You can think of the learning rate as how aggressively you want to modify your parameters at each step. A smaller learning rate makes training slower, but can be better at finding the optimal solution.



You can have a fixed learning rate for the whole training, or you can start with a higher learning rate and periodically decrease it as the training progresses. This is called a *learning rate schedule* and can be controlled with the options shown here.



Validation Data Set

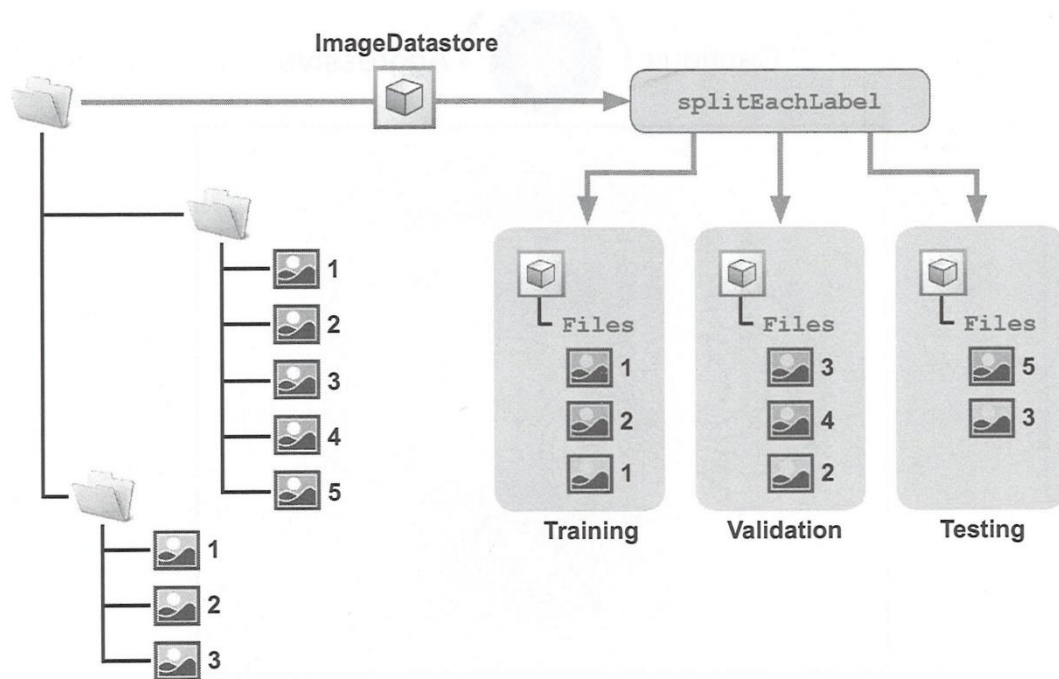
Overfitting is when a network has higher training accuracy than testing accuracy. The network learns the specific details of the training images instead of the inherent features present in the images.

To prevent overfitting, you can use a validation data set. This is a separate set of images used to evaluate the network during training. If the validation accuracy is consistently lower than the training accuracy, you can stop the training and prevent overfitting.

You can use the `splitEachLabel` function to split your images into three data sets.

```
[trads, vds, testds] = splitEachLabel (imds, 0.8, 0.1)
```

This is use 80% of your data for training, 10% for validation, and 10% for testing.



After this partition is done, you can control validation by setting the appropriate training options:

- 'ValidationData': Specify which data set to use for validation
- 'ValidationFrequency': Number of iterations between each validation (default:50)
- 'ValidationPatience' : Number of times the loss on the validation can be larger than or equal to the previously smaller loss before training is stopped (default: inf – the training never stops).

Improving Performance of Land Cover Classifier

In the last chapter, you trained the land cover network using previously determined training options. How were these options selected?

The two main considerations when evaluating network performance are accuracy and training time. There is typically a trade-off between accuracy and training time.

A network trained using the default training options does not always provide good results. There is no “right” way to set the training options, but there are a few recommended steps to improve the performance of your network. In this chapter, you will improve the performance of the land cover classifier trained by changing various aspects of the training process.

Try

Import the network and the data.

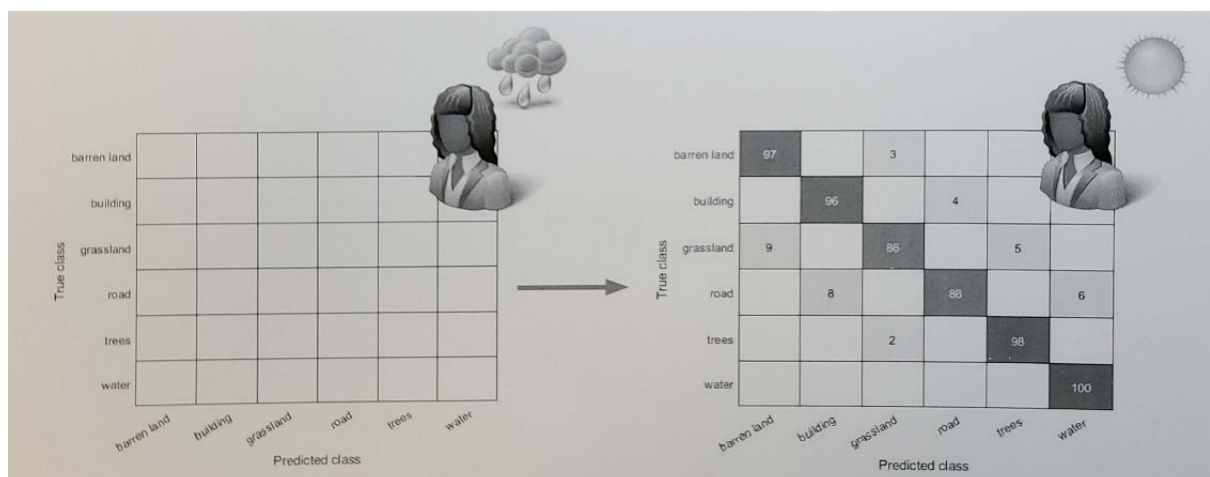
```
load satData.mat
load satLayers.mat
```

Use the default settings for the training options.

```
opts = trainingOptions( 'sgdm', ...
    'Plots', 'training-progress' ...
    'ValidationData', {XVal, YVal})
```

Train the network and view the performance.

```
net = trainNetwork(XTrain, YTrain, layers, opts)
```



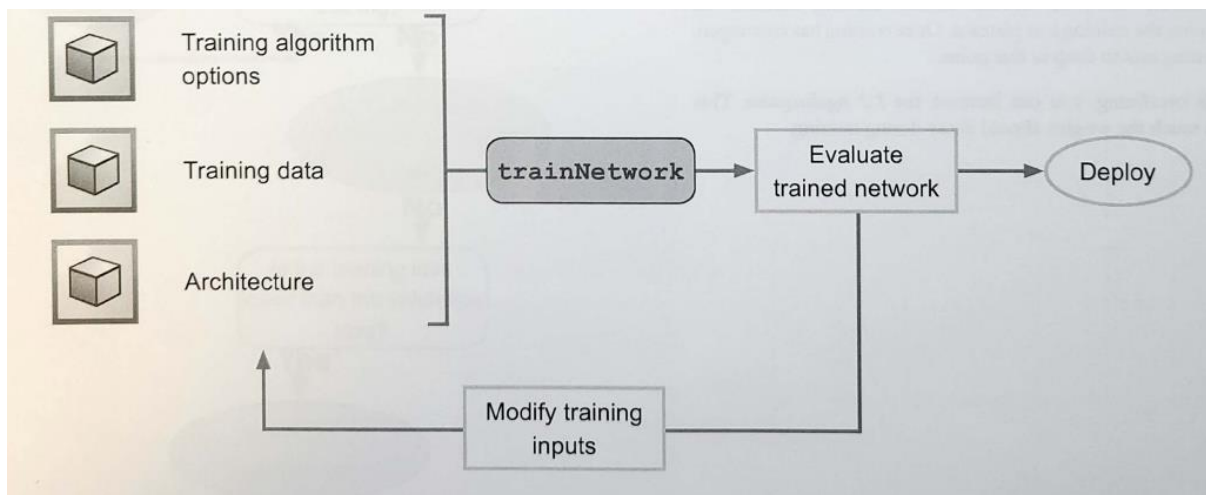
Improving Performance Overview

Training a deep neural network requires three components:

- Data with known labels
- An array of layers representing the network architecture
- Options that control the behavior of the training algorithm

After the network has been trained, accuracy you can evaluate the new network. If you are happy with the performance, you can begin using this network in production. However, the accuracy of the network is often not adequate.

To improve the performance, you can look at changing the three training inputs.



Train Algorithm Option

Modifying the training options is generally the first place to begin improving a network. Up until now, training options that were found to be effective were provided to you. In this chapter you will learn the process behind the selection of those options.

Training Data

If you do not have enough training data, your network may not generalize to unseen data. If you cannot get more training data, augmentation is a good alternative.

Architecture

If you are performing transfer learning, you often do not need to modify the network architecture. However, if you are training a network from scratch, is more difficult to know whether your architecture is effective or not. One option is to use an existing architecture from a problem similar to yours.

Guidelines for Training Options

The chart on the next page shows some general guidelines with setting training options to train a convolutional neural network.

When possible, you should use a validation data set and use the training progress plot to observe network training.

The ideal learning rate is the largest value such that the network trains effectively.

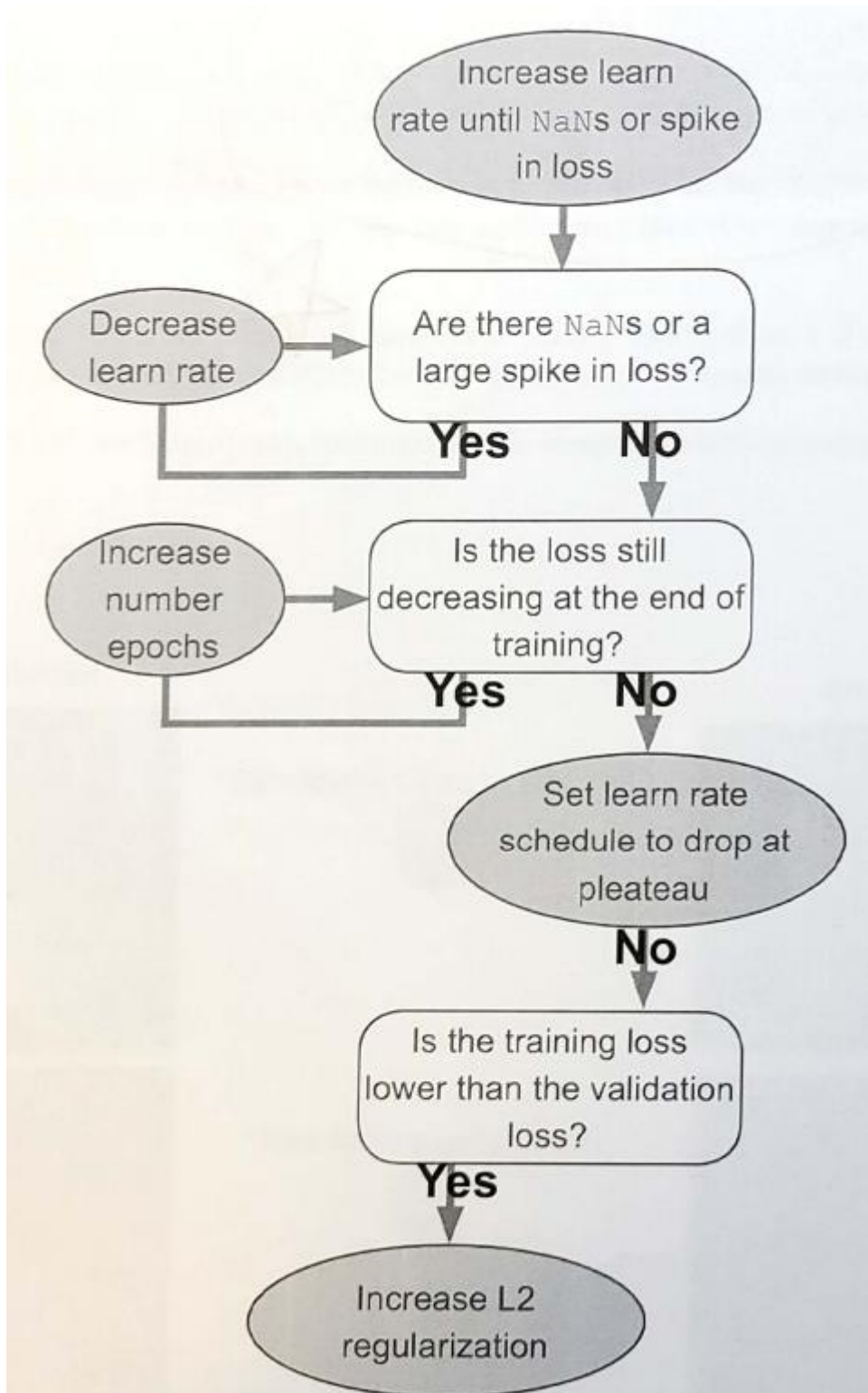
- If the learning rate is too small, it will take a very long time to train your network
- If the learning rate is too large, you may converge to a suboptimal set of parameters.

To find this learning rate, you should increase the learning rate until you see the loss “explode”. Then, decrease the loss until training proceeds appropriately.

Next, you should ensure that your network finishes learning. Increase the number of epochs until the training loss plateaus. Once training has converged, you can set the learning rate to drop at this point.

If your network is overfitting, you can increase the *L2 regularization*. This parameter controls much the weights should decay during training.

Training Option Flow Chart



Try

Use the chart to change training options to improve accuracy of the land cover network.

```
opts = trainingOptions( 'sgdm', ...  
    'Plots', 'training-progress' ...  
    'ValidationData', {XVal, YVal})
```

Train the network and view the performance.

```
net = trainNetwork(XTrain, YTrain, layers, opts)
```

Predict labels for the test data and calculate the accuracy.

```
testpreds = classify( net, XTest)  
nnz(testpreds == Ytest)/numel(testpreds)
```

Modifying Training Data

Balance or Weight Classes

If you have very unbalanced classes, try splitting the data so that the training images have an equal number of samples from each class.

Normalize

If you are performing regression, normalizing can help stabilize and speed up network training.

Image Augmentation

If you notice that the validation loss is consistently higher than the training loss, you can to augment your training data by transforming them with a combination of translation rotation, scaling, etc. Augmentation prevents overfitting because the network is unable to memorize the exact details of the training data.

Try

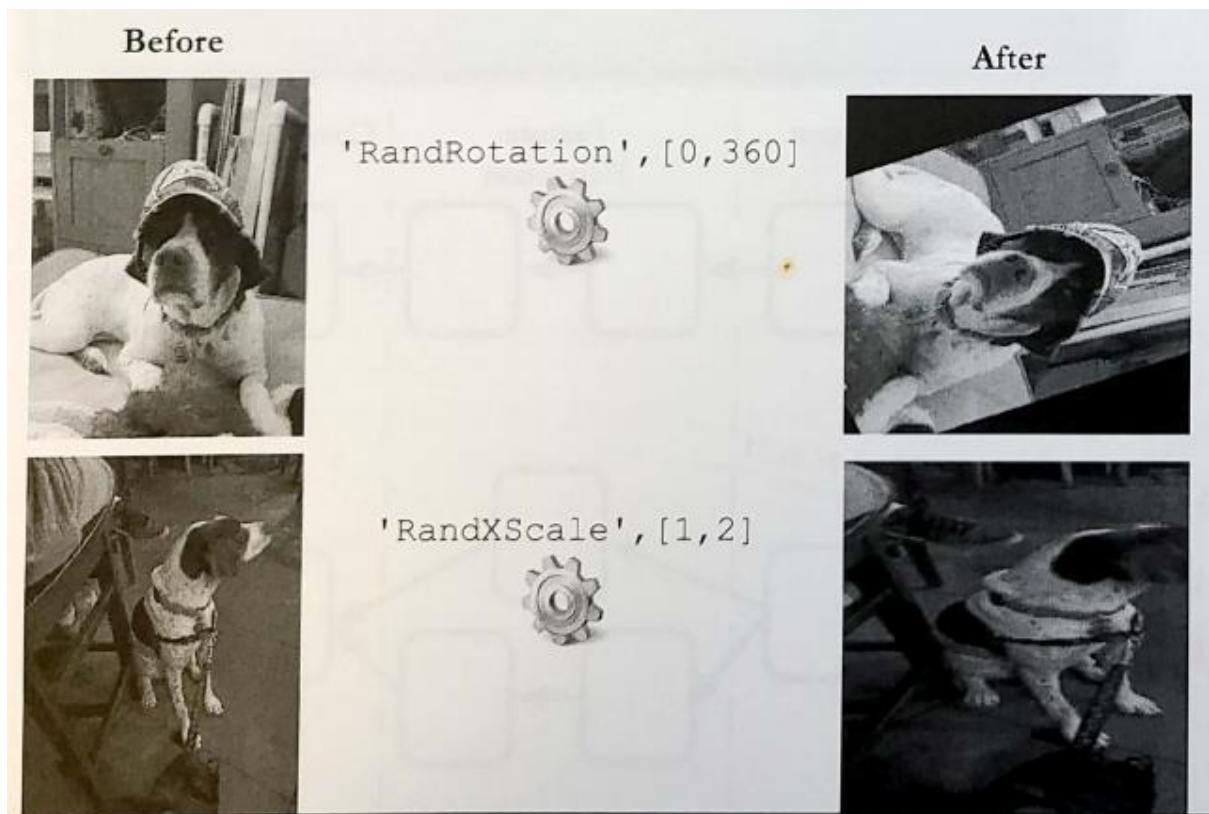
View more details on preprocessing image data in the documentation.
doc ('Deep Learning Tips and Tricks')

Augmenting Images

You can use an `augmentedImageDatastore` to perform augmentation. When you train the network, the datastore will randomly transform the images using settings you provide. The transformed images are not stored in memory, which is useful when training on large data sets.

To provide those settings, use the `imageDataAugmenter` function. For example:

- `RandXScale`: Range of horizontal scaling specified as a 2-element vector; similarly `RandYScale` affects the range of vertical scaling.
- `RandXReflection`: Horizontally flip image with 50% probability.



Try

Create a normal image datastore of pets and display the first image.

```
petds = imageDatastore('petImages', ...  
    'Includesubfolders', true)
```

Configure augmentation settings to do random scaling and rotation.

```
aug = imageDataAugmenter('RandXReflection', ...  
    true, 'RandXScale', [1 1.2])
```

Create the augmented datastore.

```
augds = augmentedImageDatastore([227 227], ...  
    petds, 'dataAugmentation', aug)
```

Read in a mini-batch and display the first augmented image.

```
mb = read(augds);  
imshow (mb.input{1})
```

Thus, the augmentation of your images is done in the following two steps:

```
aug = imageDataAugmenter( ...);
```

```
augds = augmentedImageDatastore(outputSize, ...  
    imds, 'DataAugmentation', aug);
```

Course Example: Color Correction

In this chapter, a set of images have been distorted by modifying their red, green, and blue channels. The response (or Y-data) for each modified image is three numeric values that correspond to the intensity increase or decrease of the corresponding channel.

You will train a deep network that will perform color correction by predicting the three intensity changes of an image. Since these intensity changes are continuous valued numbers, this is a regression problem.

To perform transfer learning for regression, you can use the `trainNetwork` function with your data, architecture, and training options as input.

You cannot use an image datastore to store your training data when performing regression because the labels are not categorical. One convenient alternative is to use the `table` data type.

The first variable of the table should contain the file directories and names of each image. The remaining columns should contain the numeric responses.

In the color correction data set, the response is a three-element vector of color intensities.

	1 File	2 Color		
1	'TrainingImages/img0001.jpg'	18	-15	-25
2	'TrainingImages/img0002.jpg'	5	16	-8
3	'TrainingImages/img0003.jpg'	32	15	-1
4	'TrainingImages/img0004.jpg'	-7	-13	14
5	'TrainingImages/img0005.jpg'	-3	-40	-8
6	'TrainingImages/img0006.jpg'	20	-22	-11
7	'TrainingImages/img0007.jpg'	3	2	8
8	'TrainingImages/img0008.jpg'	26	22	7

Try

View an original image of a dog.

```
imshow(imread('dog.jpg'))
```

Load the color correction training data set.

```
load trainingData.mat
```

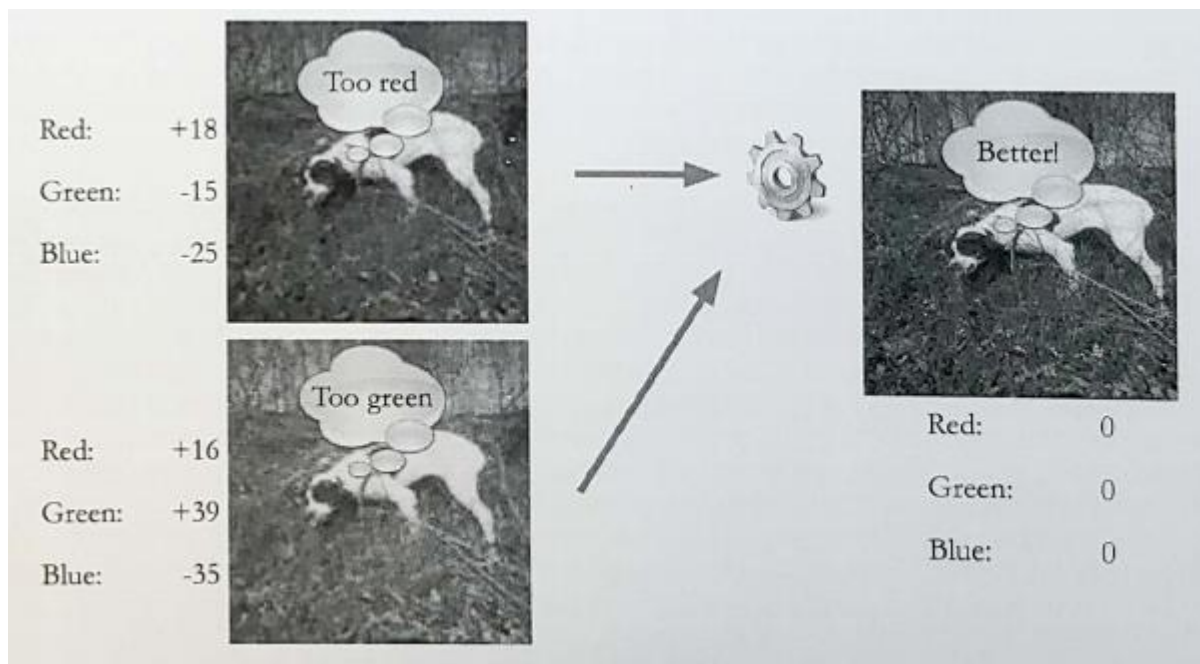
Navigate to the folder containing the color correction images (ColorCast).

Read and display the first distorted image.

```
trainfiles = trainingData.File
fn = trainfiles{1}
imshow(imread(fn))
```

Display the distortion on the first image.

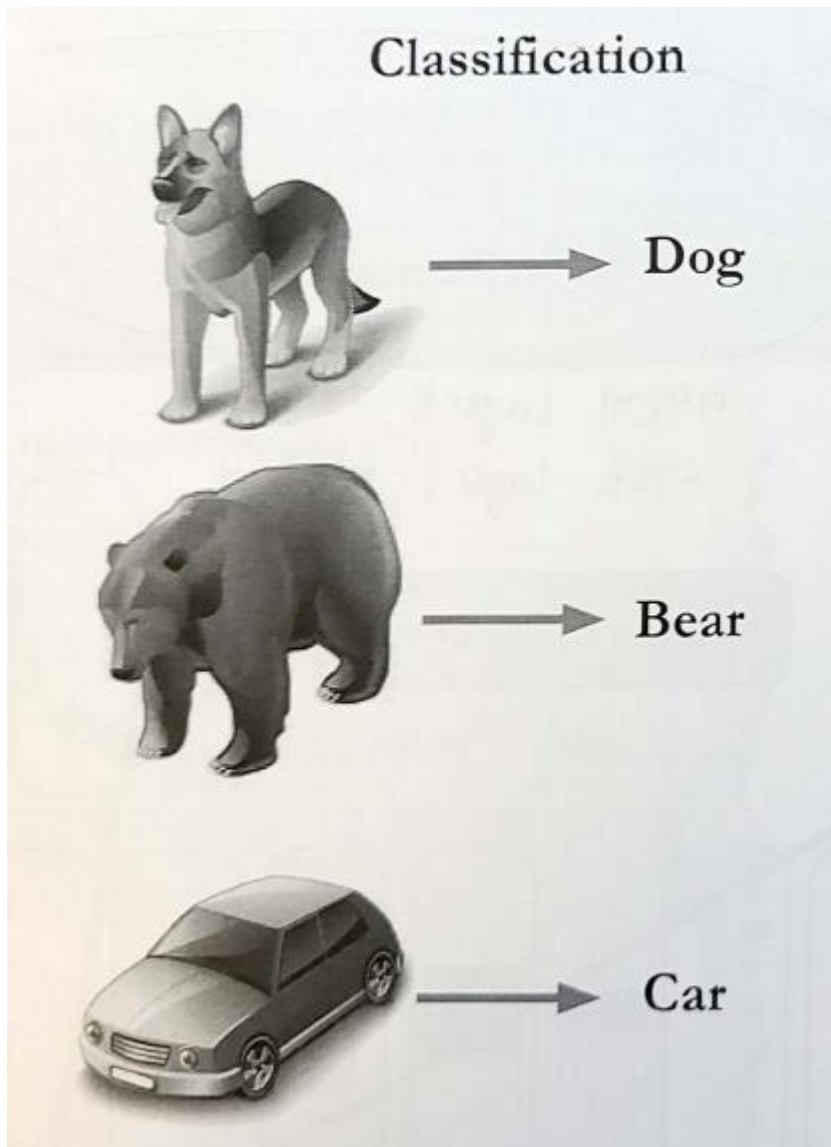
```
disp(trainingData.Color(1, :))
```



What Is Regression?

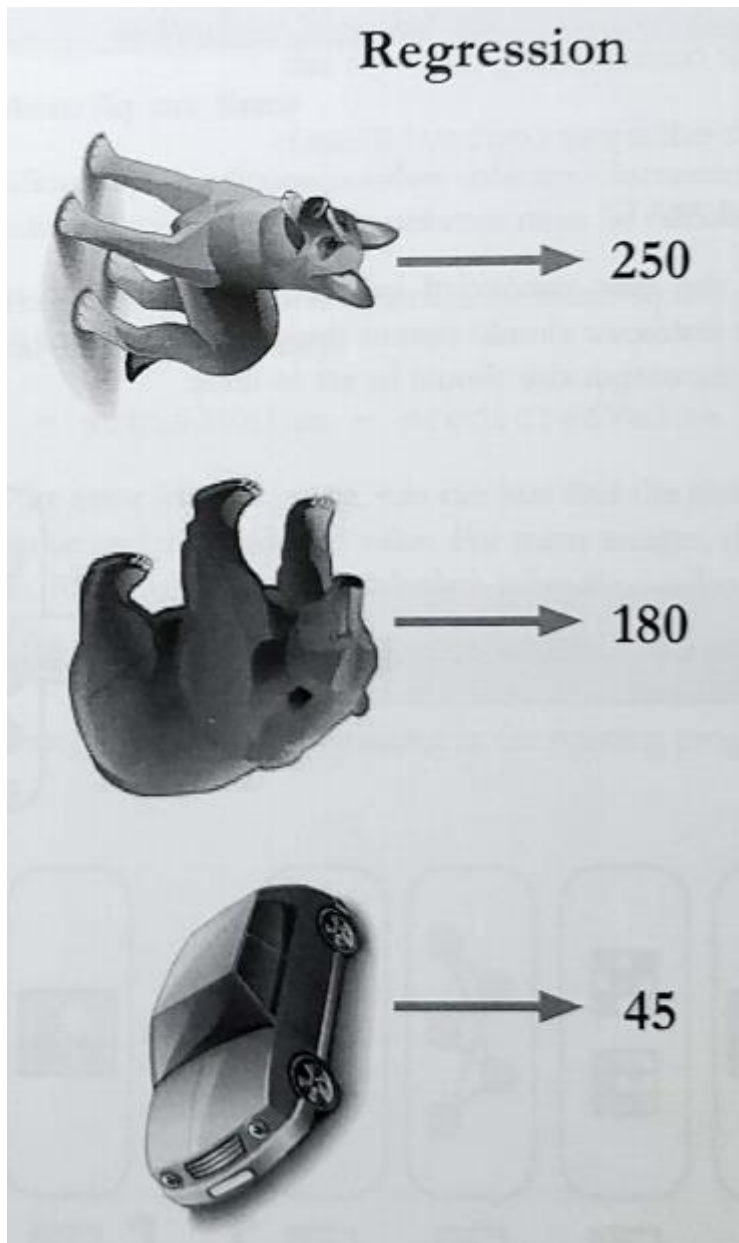
In previous chapters, you classified images using convolutional neural networks.

More generally, classification refers to the task of using data and known responses to build a model predicts a discrete response for new data. For image classification, the input data is the image and the known response is the label of the image subject.



Regression is another task that can be accomplished with deep learning. *Regression* refers to assigning continuous response values to data, instead of discrete classes.

One example of image regression is correcting rotated images. The input data is a rotated image, and the known response is the angle of rotation.



Modifying Network Layers for Regression

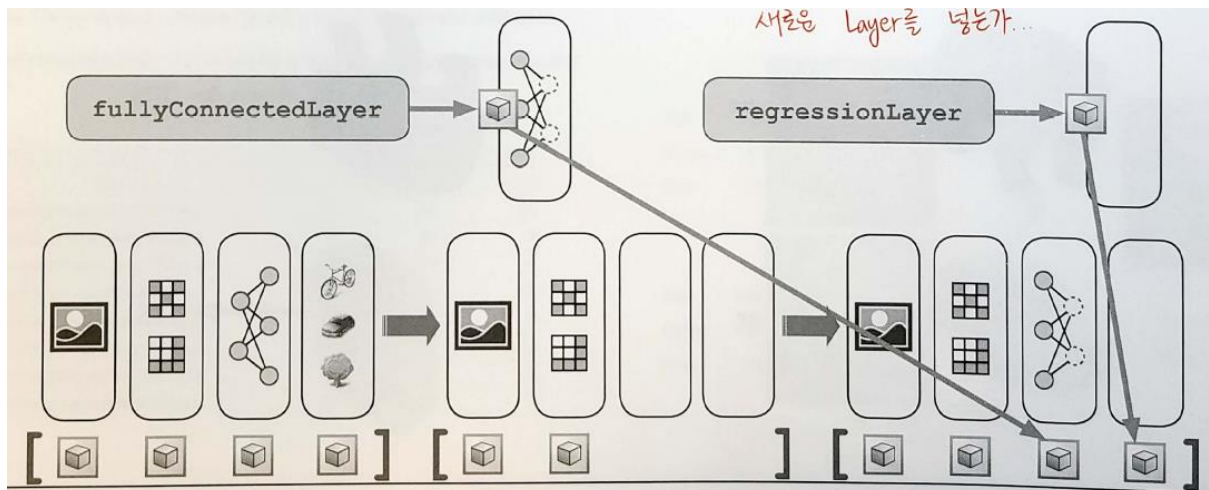
Most pretrained networks were designed to classify images. You can modify the output layers of pretrained networks to perform regression instead.

The last three of AlexNet are specific to image classification. You should replace them with the correct layers to perform regression.

For regression network, the last two layers must be a fully connected layer and a regression layer. The corresponding functions are:

- `fullyConnectedLayer (outputSize)`
- `regressionLayer()`

The output size of the fully connected layer is the number of numeric responses. The color corrector should output three numeric values (one for each RGB value), so the output size should be set to three.



Try

Import AlexNet and extract the layers.

```
net = alexnet  
layers = net.Layers
```

Remove the last three layers.

```
layers(end-2:end) = []
```

Create the layers needed for regression.

```
newLayers = [fullyConnectedLayer(3); ...  
             regressionLayer() ]
```

Append the new layers to the existing architecture.

```
layers = [layers; newLayers]
```

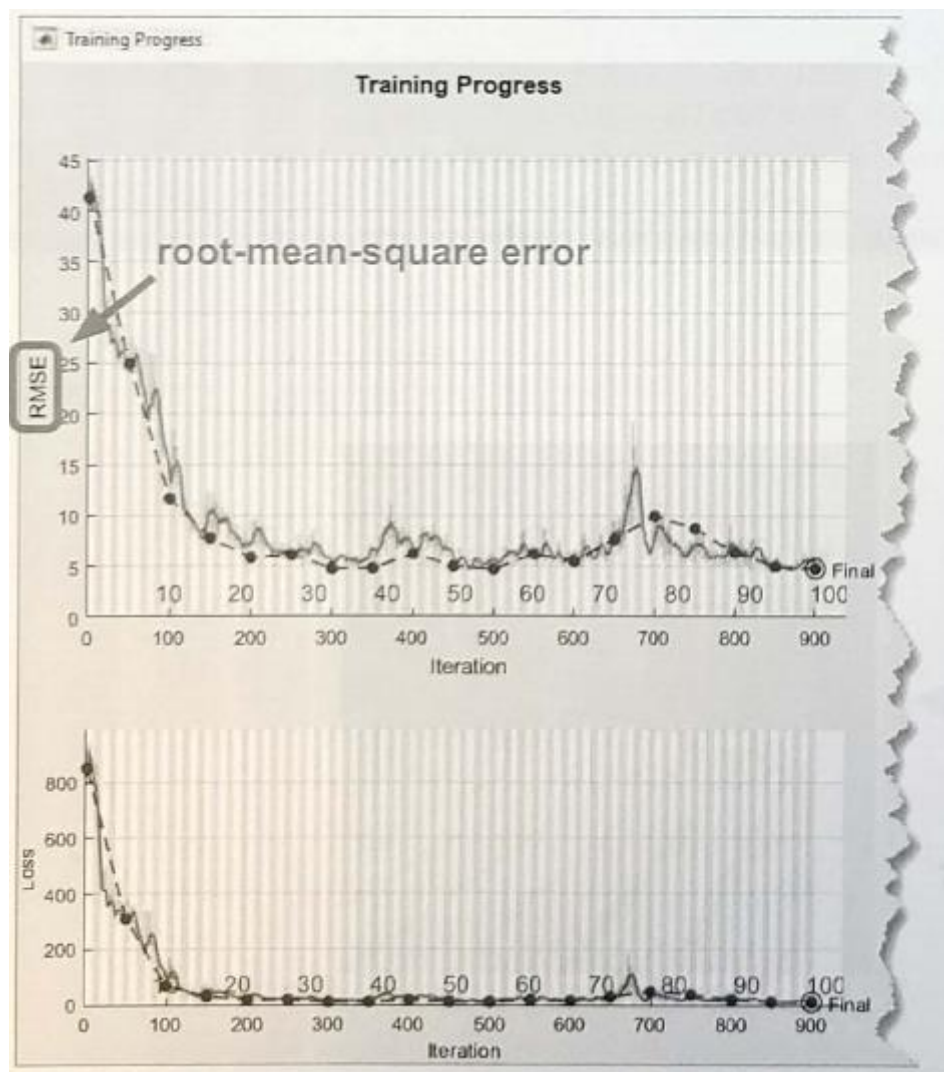

Training a Regression Network

Training the Network

You can perform transfer learning for image regression with the `trainNetwork` function. The command shown below can be used to train the color correcting network.

```
ccnet = trainNetwork (trainingData, ...  
    'Color', layers, opts);
```

The second input is the name of the variable containing the response value. The output network `ccnet` will perform regression.



Try

You can train a network using this script.

`edit TrainColorCorrector.mlx`

Or import a trained network regression.

`load ccnet.mat`

Root-Mean-Square Error

In classification, a prediction is either correct or incorrect. The effectiveness of a prediction with a regression network must be calculated differently.

When a regression network is trained, root-mean-square error (RMSE) is calculated instead of accuracy.

$$\text{err} = \text{actualValue} - \text{predictedValue}$$

To find the error for one image, you can find the distance between the known value and the predicted value. For many images, this error is a vector of values. RMSE allows you to calculate value for a collection of images.

$$\text{rmse} = \sqrt{\text{mean}(\text{err}^2)}$$

You can see this value during training in the training progress plot in place of accuracy.

Evaluating a Regression Network

You can predict one image to visually evaluate the effectiveness of your network. As with classification networks, it is more robust to evaluate with a test data set.

To make predictions on the testing data, use the `predict` function.

```
predictedValue = predict (net, X);
```

`X` can be a single image or a table where the first column is the path to your images. The predictions will be stored in a N -by- r matrix where N is the number of images and r is the number of responses.

You can calculate RMSE to evaluate the test error.

```
err = testData.Color - predictedValue
```

```
rmse = sqrt (mean (err. ^2))
```

Try

Classify a test data set. Make sure you are in the ColorCast folder.

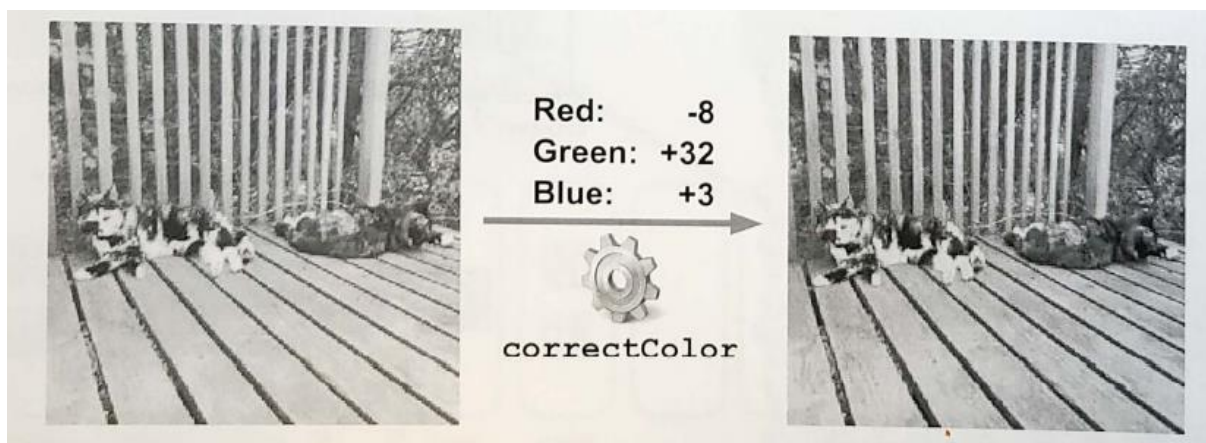
```
imshow (imread ('dog.jpg'))  
pred = predict(ccnet, testData)
```

Use the network to correct the first image. The function `correctColor` is part of the course files.

```
rgb = pred(1, : )  
testim = imread (testData.File{1})  
correctedim = correctColor (testim, rgb);  
imshow (correctedim)
```

Calculate the root-mean-square error (RMSE) for the test data set.

```
err = testData.Color - pred  
rmse = sqrt(mean (err. ^2))
```

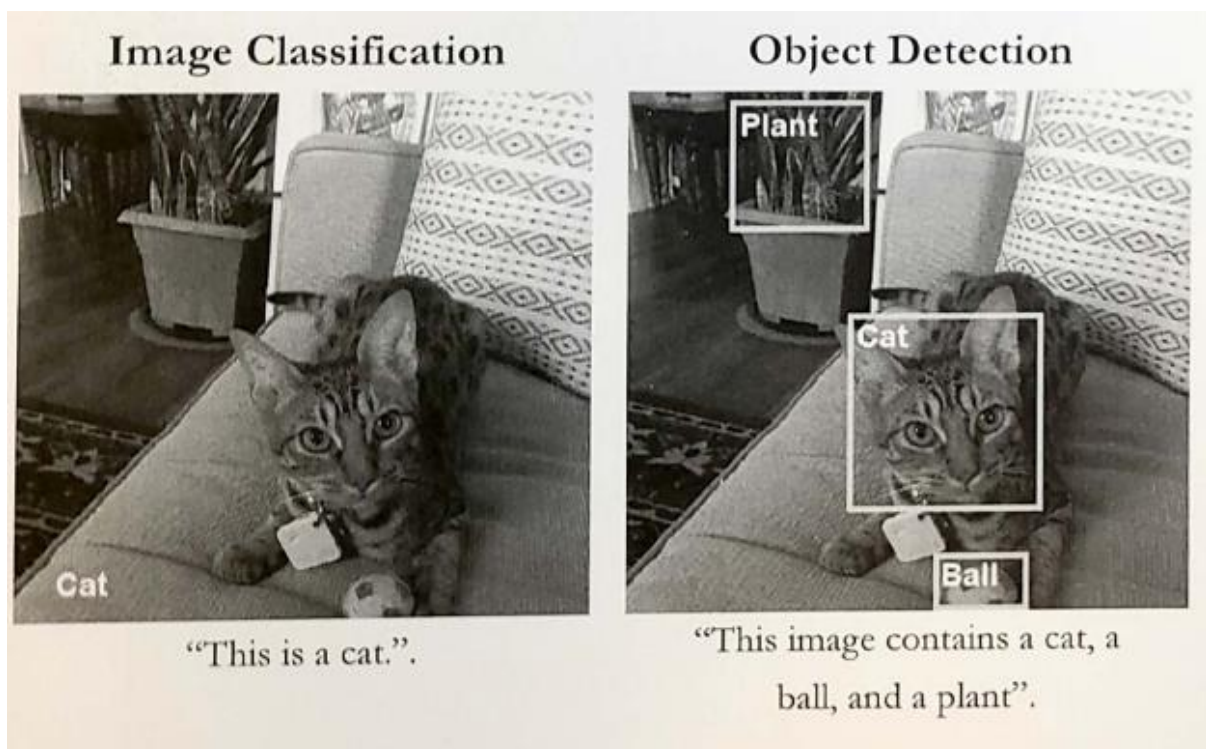


Course Example: Face Detector

In previous chapters, pretrained networks were used for image classification and regression. Object detection is a combination of classification and regression.

A region of interest, or ROI, is located within the image (regression). The ROI is labeled (classification). This process is known as object detection.

You will train an object detector that can distinguish between five of the pets in the data from Chapter 2. Unlike a classifier, the detector will be able to find bounding boxes around multiple faces in the same images.



Try

Read and display an image.

```
im = imread ('Madeline.jpg');  
imshow (im)
```

Create a bounding box and an appropriate label.

```
bbox = [778 367 953 959]  
label = 'Cat'
```

Insert the bounding box and label into the image. Increase the line width and font size to account for the large size of the image.

```
im = insertObjectAnnotation (im,'rectangle',...  
    bbox, label, 'LineWidth', 10, 'FontSize', '72');  
imshow (im)
```

You can display the data set using the `insertObjectAnnotation` function.

```
detectedim = insertObjectAnnotation (im, ...  
    'rectangle', bbox, label);  
imshow (detectedim)
```

Each bounding box is a four element vector `[x, y, width, height]`, where `x` and `y` refer to the upper-left corner location.

Ground Truth for Object Detection

Labeled data is required to train any deep network

The labeled images for object detection are often called *ground truth*. This data consists of a bounding box for each object in the image that you want to detect.

The Image Labeler app is a useful tool for labeling images. You can populate the app with your data set using an image datastore or a folder name as input.

```
imageLabeler('images')  
imageLabeler(imds)
```

The app can return the ground truth in a table where the first variable contains the image file name and the remaining variables contain the bounding boxes organized by label name.

If you have multiple labels in your data set, the column names will correspond to each label. You can extract specific elements from the table using curly brace (`{}`).

```
bbox = myGroundTruth.myLabel{n}
```

Try

Navigate to the folder 'petImages'. Make an image datastore of the five pets with bounding boxes.

```
load pathToPetFaces.mat
petds = imageDatastore (pathToPetFaces, ...
    'IncludeSubfolders', true)
```

Open the Image Labeler app. Create bounding boxes around a few images.

```
ImageLabeler (petds)
```

Load a completed table of ground truth. View the first image.

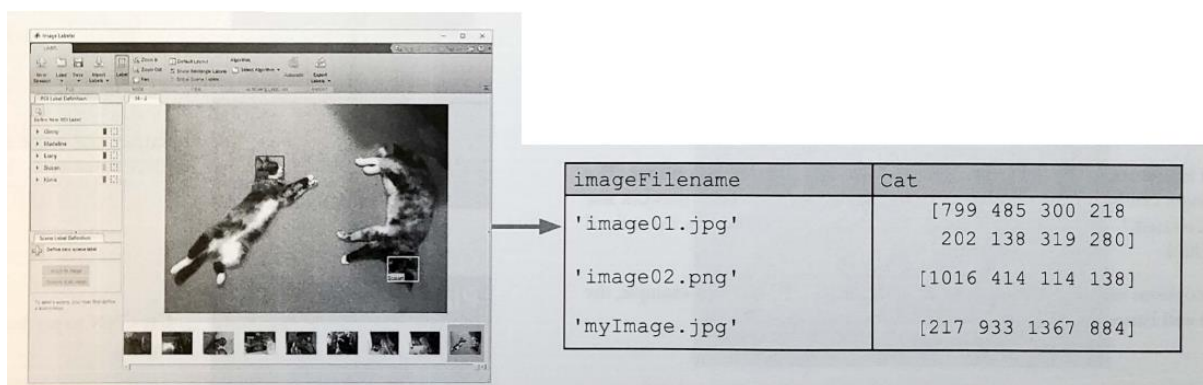
```
load petGT.mat
fn = petGT.imageFilename{1}
im = imread(fn)
imshow (im)
```

The first image contains Madeline. Get the bounding box and create the label.

```
bbox = petGT.Madeline{1}
label = 'Madeline'
```

Insert the bounding box and label into the image.

```
im = insertObjectAnnotation (im, 'rectangle',...
    bbox, label);
imshow (im)
```



Regions with Convolutional Neural Networks

Regions with convolutional neural networks (R-CNNs) can be used to perform object detection.

To train an R-CNN, you need an image data set and ground truth. You also need a network and training options.

You can train three different types of R-CNNs in MATLAB: R-CNN, Fast R-CNN, and Faster-RCNN. The corresponding functions are:

- `trainRCNNObjectDetector`
- `trainFastRCNNObjectDetector`
- `trainFasterRCNNObjectDetector`

These networks differ between training time and detection time. For example, a R-CNN can be trained quickly, but the time to detect a new image is slower than a Faster R-CNN network.

Choice of which network to use depends on your application. The Fast and Faster R-CNNs are designed to improve detection performance with a large number of regions at the cost of training time.

All of these functions have the same inputs and outputs:

```
detector = trainRCNNObjectDetector (data, net, opts)
```

The inputs are a table of ground truth, a network, and training options. The network can be a network objects, or the name of a pretrained network like 'alexnet' or 'squeezenet'. The output `detector` is the trained R-CNN.

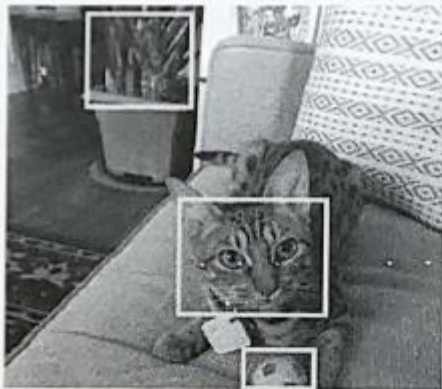
The options used with each type of R-CNN will be different. For example, the Fast and Faster R-CNNs require a `MiniBatchSize` of 1/

Try

Set options and train the network.

```
opts = trainingOptions('sgdm', ...  
    'InitialLearnRate', 0.0001, ...  
    'MaxEpochs', 5, ...
```

```
detector = trainFastRCNNObjectDetector(...  
    petGT, 'alexnet', opts)
```



Find regions likely to contain objects.



Extract and resize each region to CNN input size.



Use CNN to predict class of each region.

Using an Object Detector

Once the object detector is trained, you can detect objects in a new image using the `detect` function:

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

The inputs are the trained R-CNN `detector` and an image `I`. The outputs are as follows:

- bboxes**: The locations of the regions. Each row is one bounding box.
- scores**: The confidence in the classification ranging from 0 to 1.
- labels**: The predicted label for each detected region.

When you display the detection with `insertObjectAnnotation`, you can format the labels with the label and score.

```
newlabel = [char (labels) ': ' num2str (scores)]
```

Try

load a trained pet detector.

```
load petDetector.mat
```

Detect Ginny the dog in a sample image.

```
dogim = imread ('Ginny.jpg');
```

```
imshow (dogim)
```

```
[bboxes, scores, labels] = detect (detector, dogim)
```

Format the labels.

```
labelstr = cell (size (labels))
```

```
for k = 1: length (labels)
```

```
    labelstr{k} = [char (lables(k)),': ', ...
```

```
                    num2str (scores (k))]
```

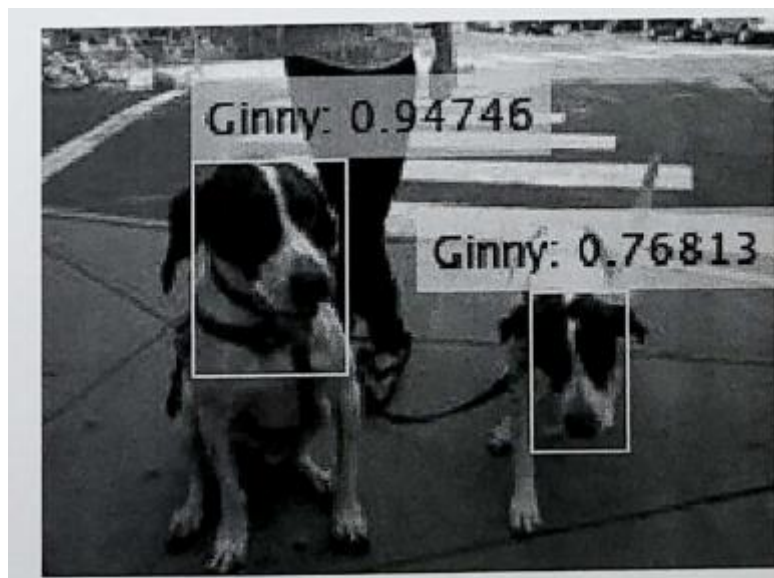
```
end
```

View the results of the detection.

```
detectedim = insertObjectAnnotation (dogim, ...
```

```
    'rectangle', bboxes, labelstr);
```

```
imshow (detectedim)
```



Thresholding Detections

If you are not happy with the results of your detection, you may want to apply a threshold the results to only include confident predictions.

```
idx = scores > 0.9  
bboxes = bboxes (idx, : )  
labels = labels (idx)
```

When you consider the effectiveness of your R-CNN, you should account for two things:

- The precision of the bounding box around the detection
- The correctness of the label of the detection

Try

Extract detections above 90% confidence.

```
idx = scores > 0.9
```

```
bboxes = bboxes (idx, :)
```

```
labels = labels (idx)
```

View the results of the detection.

```
detectedim = insertObjectAnnotation (dogim, ...
```

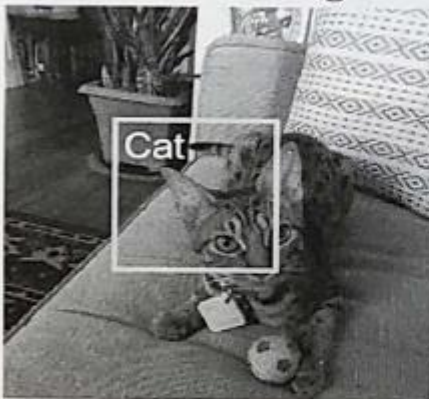
```
'rectangle', bbox, cellstr (label));
```

```
imshow (detectedim)
```

Run a script that detects a few example images.

```
edit DetectFaces.mlx
```

Incorrect Bounding Box



The label is correct, but the box is not precisely around the cat.

Incorrect Label



The bounding box is around the cat, but it was labeled as a dog.

Chapter Example: Author Identification

In this chapter, you will predict the author of text written by either Jane Austen or Charles Dickens.

{ it is a truth universally acknowledged }

{ it was the best of times it was the worst of times }

{ there is no charm equal to the tenderness of the heart }

{ have a heart that never hardens and a temper that never tires }

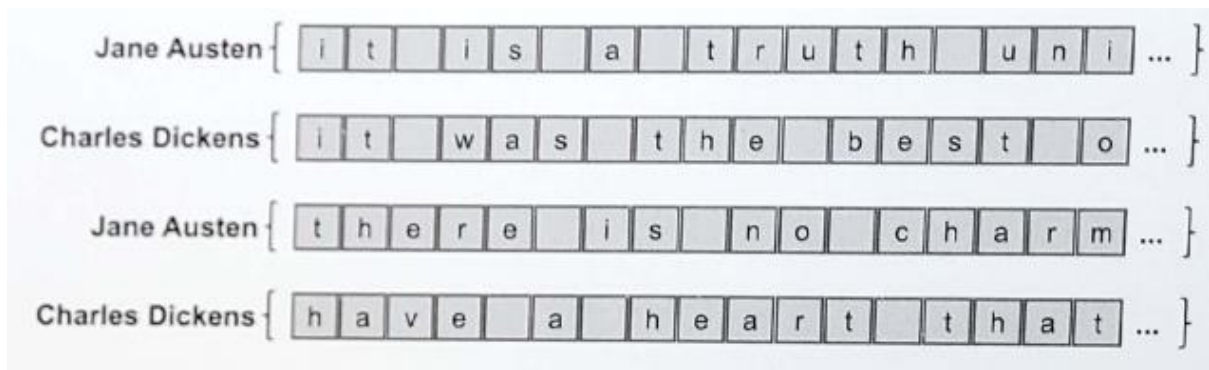
You will use a different architecture called a long short-term memory (LSTM) network. Convolutional neural networks and long short-term memory networks consist of different layers, but they both perform deep learning.

Try

Load the author samples.

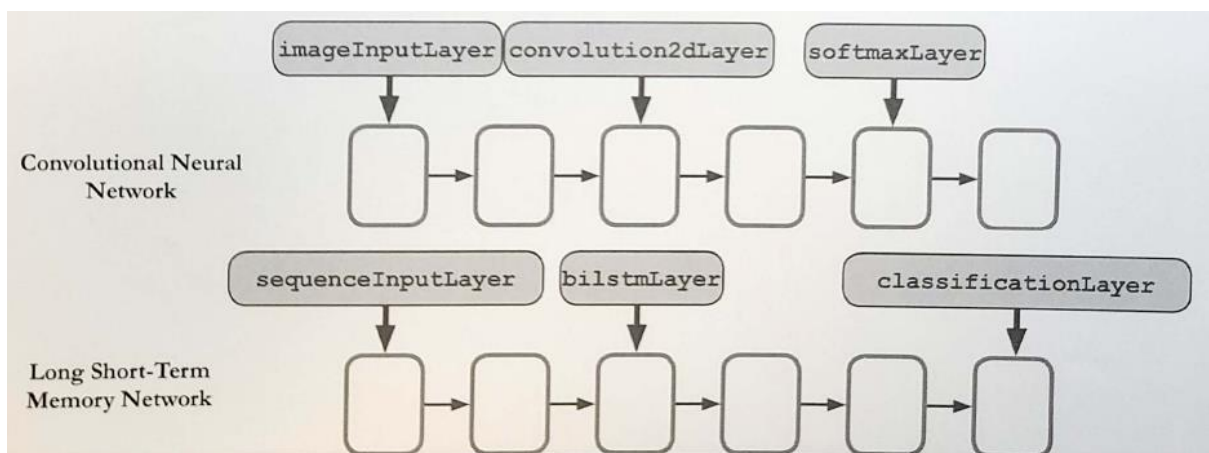
```
load authorSamples.mat
```

You can think of text data as a sequence of characters. If a sentence is one sequence, every letter is one time step. If you assign a label to each sequence of text, you can use long short-term memory networks to perform classification.



The text used to create this data set is available from Project Gutenberg:

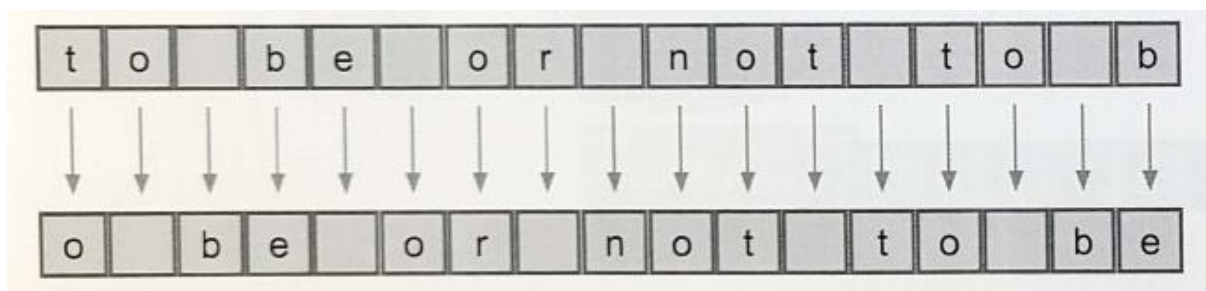
<http://www.gutenberg.org/>



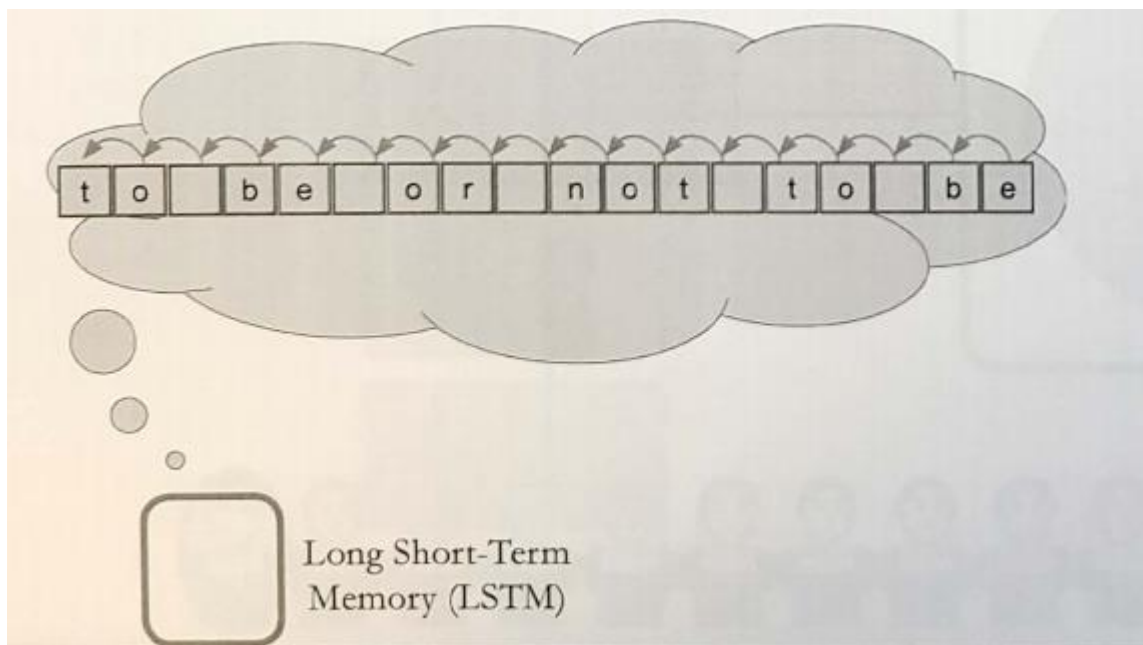
Long Short-Term Memory Networks

Long short-term memory networks, or LSTMs, are designed for applications where the input is an ordered sequence where information from earlier in the sequence may be important. These applications often involve signal, text, or time-series data.

Consider a sequence of letters “to be or not to be”. What’s likely to come next? This is a prediction problem where given a letter, the goal is to predict the next letter in the sequence. This prediction is impossible without the context of the sequence of letters.



For example, in this sequence, “o” is followed by three different letters-space, r, and t. In context, the next letter becomes easier to predict as the sequence progresses. LSTMs would be a good choice for this example because of the ordered nature of this data.



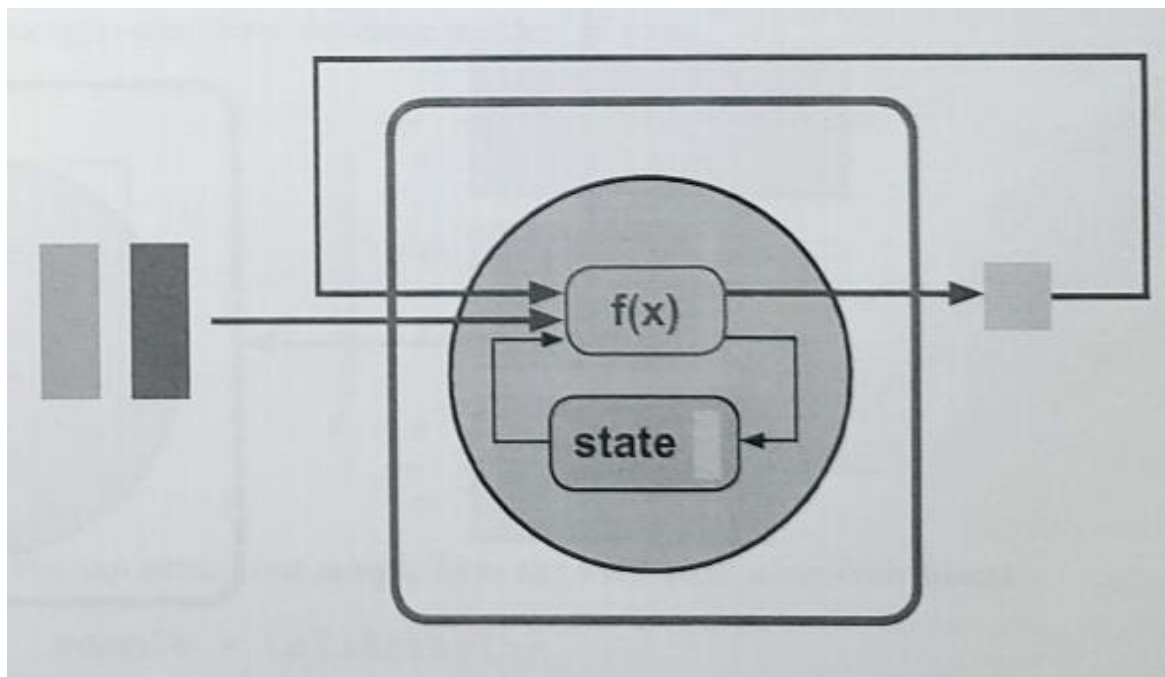
LSTMs are a type of recurrent network, which are networks that reuse the output from a previous step as an input for the next step.

Like all neural network, the node performs a calculation using the inputs and returns output value. In a recurrent network, this output is then used along with the next element as the inputs for the next step, and so on.

In an LSTM, the nodes are recurrent, but they also have an internal state. The node uses an internal state as a working memory space, which means information can be stored and retrieved over many time steps.

The input value, previous output, and the internal state, are all used in the nodes calculations.

The results of the calculation are used not only to provide an output value, but also to update the state.

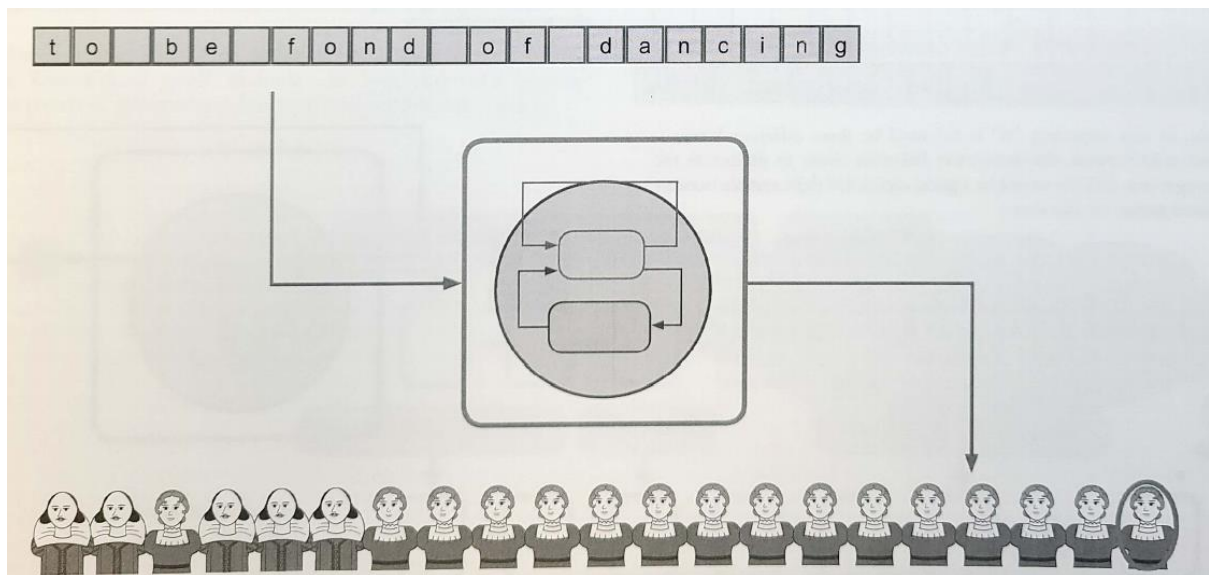


Sequence Classification

You can use an LSTM to predict which author wrote a given sequence of text.

In this case, you want a single output class for the whole input sequence. The LSTM still takes a sequence as input, and calculates an output for each input element. But only the last output is used to make the final prediction. This is called *sequence to label classification*.

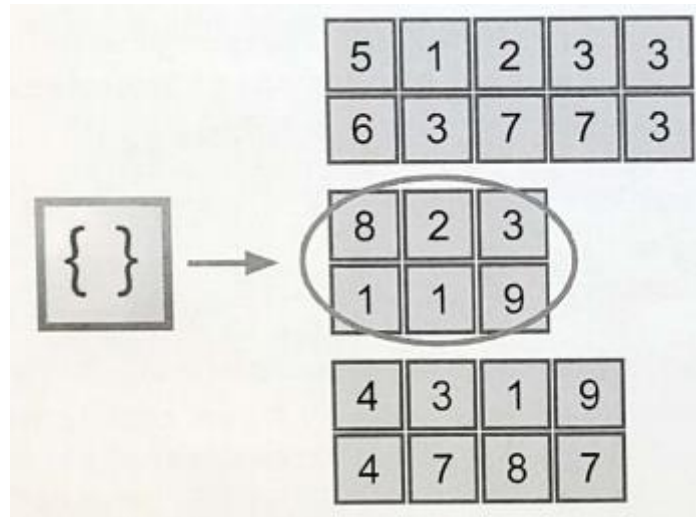
When you create LSTM layers in MATLAB, you can control this behavior by setting the output mode option to 'sequence' or 'last'. Either way, the output is informed by all the previous elements in the sequence.



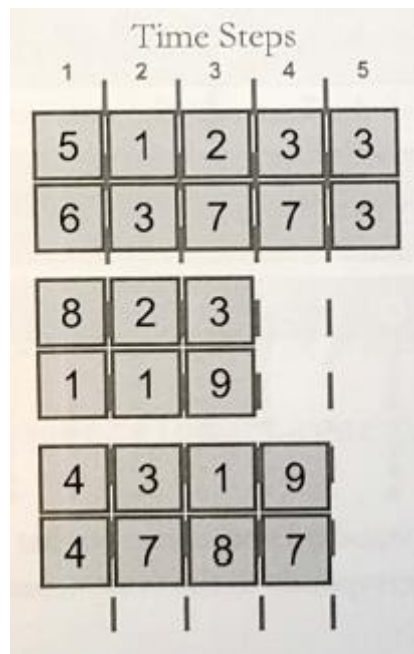
Structuring Sequence Data

Training an LSTM requires the data to be stored in a particular format.

The input data is a cell array with one column. Each element in the cell array is one sample, or sequence. This sample is a numeric matrix.



The column in each sample are the time steps. Every sample can have a different number of time steps.



Try

Load test sequences and view dimensions and class of `XTest` and `YTest`.

```
load testdata_s2label.mat
whos XTest YTest
```

Extract first sequence and label.

```
sequence = XTest{1}
label = YTest(1)
```

The first wequence is a numeric matrix. View size of the first sequence.
There are 51 features and 2000 time steps.

```
whos sequence
```

The rows correspond to the feature dimension of the sample. This could be signal data from different sensor, or different letters in a vocabulary. All samples must have the dame number of rows.

Feature Dimension	1	5	1	2	3	3		
	2	6	3	7	7	3		
	1	8	2	3				
	2	1	1	9				
	1	4	3	1	9			
	2	4	7	8	7			

You can extract one sample from the a cell array using curly braces.

```
sample = cellArray {n}
```

The corresponding labels for each training sample should be a categorical column vector. You can index into a categorical array with array indexing.

```
label = categoricalArray(n)
```

Classifying Categorical Sequences

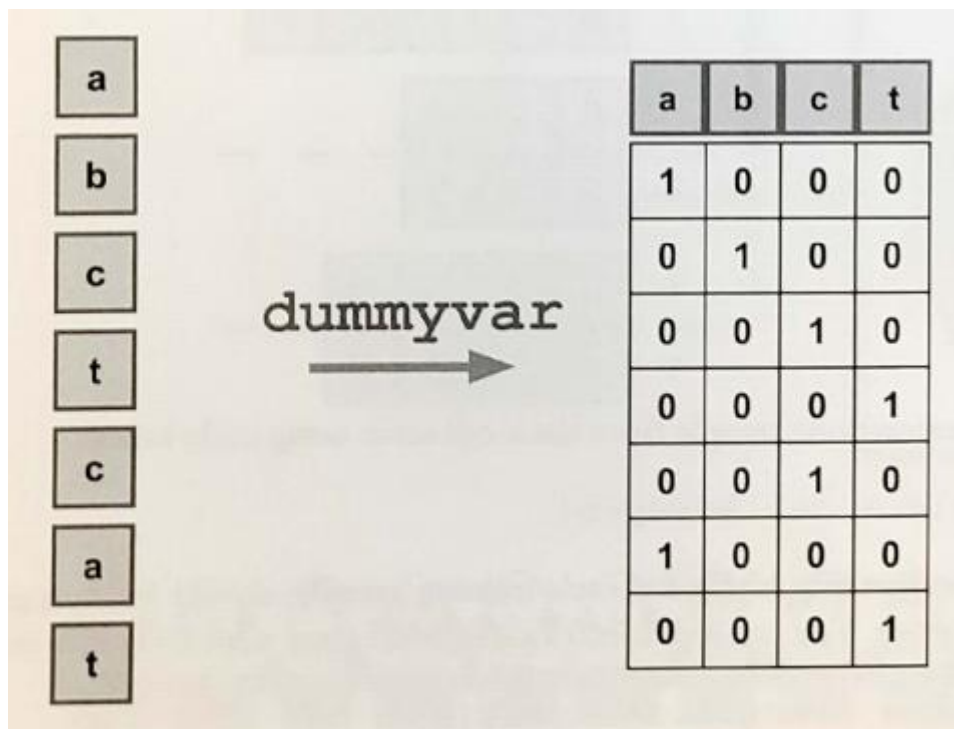
Training an LSTM requires the training data to be a numeric sequence. However, this chapter example is text data. How can you turn the text data to a numeric input?

Every character in the text can be assigned a number ('a'=1', 'b'=2', etc). However, this results in imposing a false numerical structure on the observations. For example, if you assign the numbers 1 through 4 to the letters 'a' through 'd', it implies that the distance between 'a' and 'd' is longer than the distance between 'a' and 'b'.

Instead, you will consider each character to be one category, and create *dummy predictors* for each category. Each dummy predictor can have only two values – 0 or 1. For any given character, only one of the dummy predictors can have the value equal to 1.

You can create a matrix of dummy variables using the function `dummyvar`.

```
d = dummyvar(c)
```



Try

Convert a sequence of characters into a numeric array.

```
t = 'to be or not to be'
tnum = unit8(t)
```

Use a standardized vocabulary to create a categorical array.

```
alphabet = unit8('abcdefghijklmnopqrstuvwxyz')
tcat = categorical(tnum, alphabet)
```

Convert the phrase into a dummy variable. Transpose so that the output rows correspond to each letter.

```
tdum = dymmyvar(tcat')
```

A function `dummifyText` has been created to make the dummy predictor matrix. Use it to create a dummy predictor for the Dickens sample. Notice that the capital I becomes NaN since this is not part of the vocabulary.

```
mDickens = dummifyText(tDickens, vocab)
```

You will create the dummy predictors such that the rows of the matrix correspond to each letter in the vocabulary.

The vocabulary consists of the letters in the data set. For example, consider the phrase “to be or not to be”. The vocabulary is seven characters (space, b, e, n, o, r, t). The dummy predictor matrix will have seven rows for this vocabulary of seven characters.

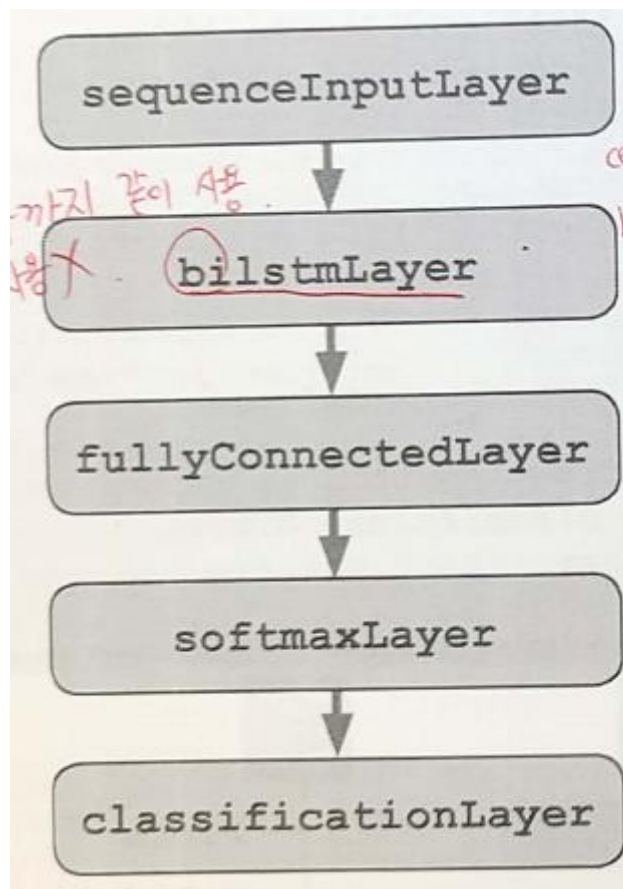
	t	o		b	e		o	r		n	o	t		t	o		b	e
t	0	0	1	0	0	1	0	0	1	0	0	0	1	0	0	1	0	0
o	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0
	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1
b	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
e	0	1	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0
n	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
r	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

You may want to declare a standard vocabulary so that each row of the dummy predictor matrix always corresponds to the same character. The Try code uses this method.

LSTM Architecture

The architecture for classifying images is stored in MATLAB as a column vector of layers. The same is true when classifying sequences with LSTMs.

All LSTMs begins with an input layer, follows with some LSTM or BiASLTM layers, and end with the same output layers as a CNN.



The input layer needs to know the number of features in a sample or the number of rows.

```
inputLayer = sequenceInputLayer(inputSize)
```


Try

Create an input layer. The vocabulary consists of 51 different characters, so this will be the number of inputs to the `sequenceInputLayer`.

```
InLayer = sequenceInputLayer (51)
```

Create two BiLSTM layers with 256 nodes each. Set the output mode to 'last' to perform sequence-to-label classification.

```
lstm = [biLstmLayer (256 'OutputMode', 'last')
        biLstmLayer (256, 'OutputMode', 'last')]
```

Create the three output layers. The fully connected layer should have an output size of two because you are classifying between two authors.

```
outLayers = [fullyConnectedLayer (2);
              softmaxLayer ( );
              classificationLayer ( ) ]
```

Concatenate the layers into one column vector.

```
layers = [InLayer; lstm; outLayers]
```

You should see the number of nodes and the output mode when create a LSTM or BiLSTM layer. A BiLSTM *layer* is a bidirectional long short-term memory layer. The difference between these layers is explained in the next chapter. Every instrument recording only contains one instrument, or one label. This means you can use the output mode 'last' because you only need the last prediction in the sequence.

```
BiLstmLayer (numNodes, 'OutputMode', 'last')
```

The last three layers in your LSTM are the same layers you used when training a CNN from scratch, `fullyConnectedLayer`, `softmaxLayer`, and `classificationLayer`. The fully connected layer requires the number of classes as input. The last two layers do not require any inputs.

Training and Using an LSTM

You can train a long short-term memory network with the `trainNetwork` function. The inputs are your data, architecture, and options.

Convolutional neural networks and long short-term memory networks consist of different layers, but they are both trained with the `trainNetwork` function and use options created with the `trainingOptions` function.

```
net = trainNetwork (sequences, labels, ...
    layers, options)
```

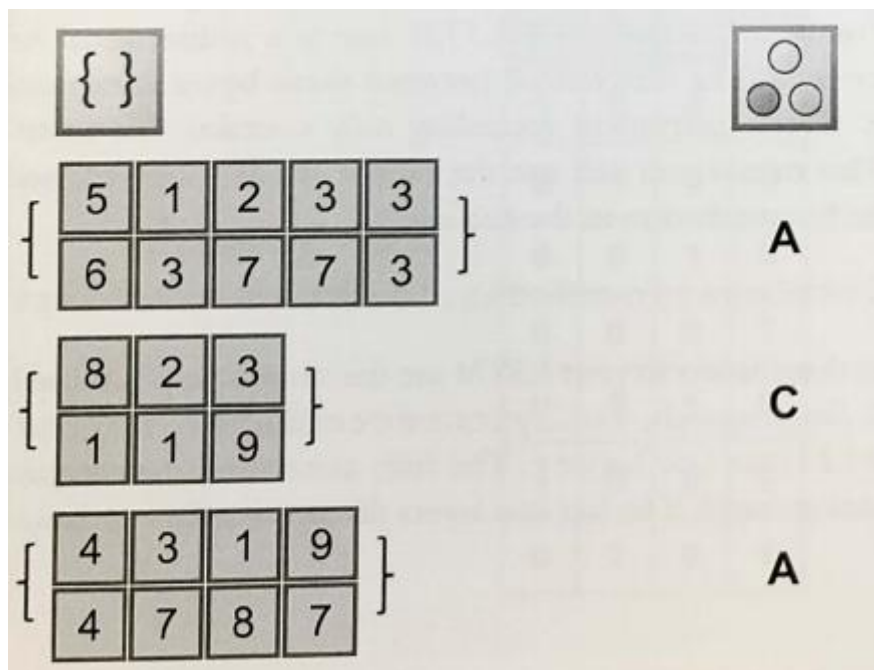
The `classify` function can also be used with an LSTM. The test sequences can be classified and compared to the corresponding output from the network.

```
predictedLabels = classify (net, testData)
```

You can make a confusion matrix using the `confusionchart` function.

```
confusionchart (testLabels, predictedLabels)
```

You should use the known labels as the first input to this function.



Try

Load the training data.

```
load traindata_s2label.mat
```

Sets the default training options used to train the LSTM. Decrease the sequence length so you do not run out of memory.

```
options = trainingOptions ('adam', ...  
    'Plots' 'training-progress', ...  
    'SequenceLength', 1000)
```

It takes about fifteen minutes to train this network.

```
net = trainNetwork (XTrain, YTrain, ...  
    layers, options)
```

Or load a trained network.

```
load net_s2label_initial.mat
```

Classify test data and view the misclassification with a confusion matrix.

```
testPred = classify (net, XTest)  
confusionchart(YTest, testPred)
```



Improving LSTM Performance

You learned about improving performance of CNNs in a previous chapter.

Most of these guidelines are still applicable to training LSTMs.

Modifying Training Data

Normalizing your sequences can make networks easier to train. Sequences can be normalized using a variety of methods. A simple method is to rescale the data to an interval like $[-1,1]$ using the `rescale` function.

You can also calculate the per-feature mean and standard deviation of all the sequences. Then, for each training observation, subtract the mean value and divide by the standard deviation.

Sequence length and padding is another consideration specific to sequence data. More information is on the next page.

Guidelines for Training Options

The flow chart in chapter six provides guidelines that are useful for training a sequence classification network.

Try

View a script that trains the improved network. It takes about fifteen minutes to run.

```
edit traingoodLSTM.mlx
```

Load a trained network and the testing data.

```
load net_s2label.mat
```

```
load testdata_s2label.mat
```

Evaluate the network using a confusion matrix.

```
testPred = classify (net, XTest)
```

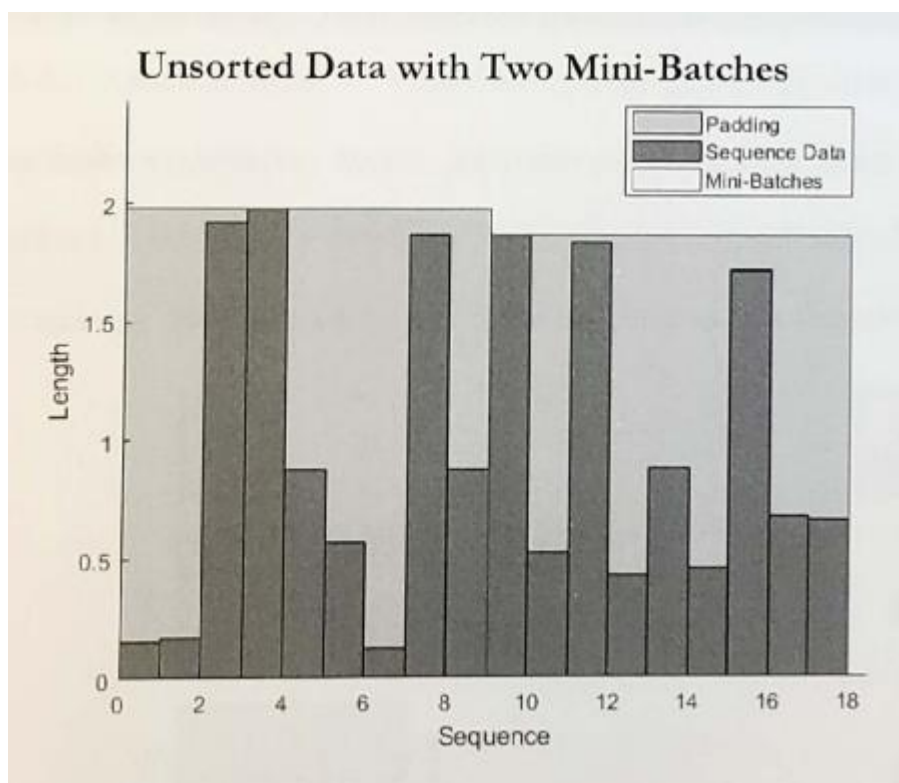
```
confusionchart (YTest, testPred)
```

Sequence Length and Padding

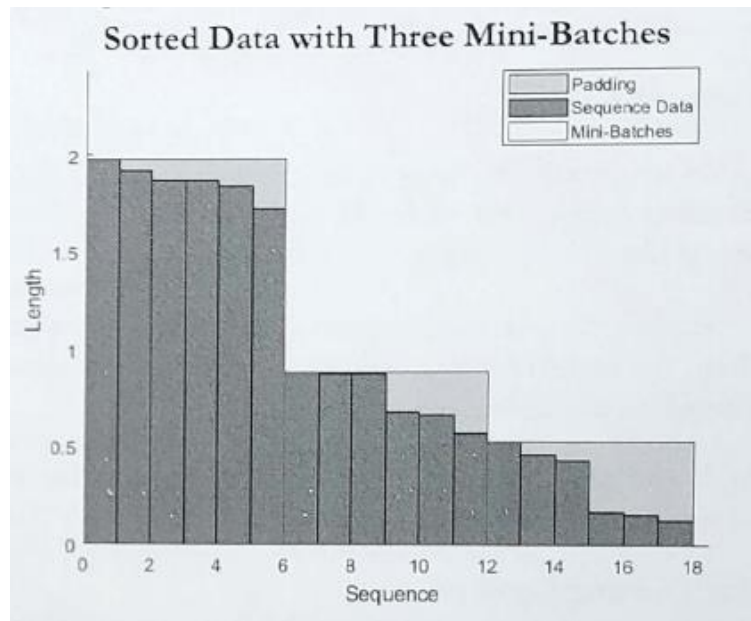
Sequences can contain any number of time steps. This is convenient, but you should be cautious if your sequences have different lengths.

During training, the sequences in each mini-batch are padded with a number, usually zero, to equalize the lengths. A network cannot distinguish between values created for padding and values that are part of the sequence. You should minimize the amount of padding by sorting your data by sequence length and carefully choosing the mini-batch size.

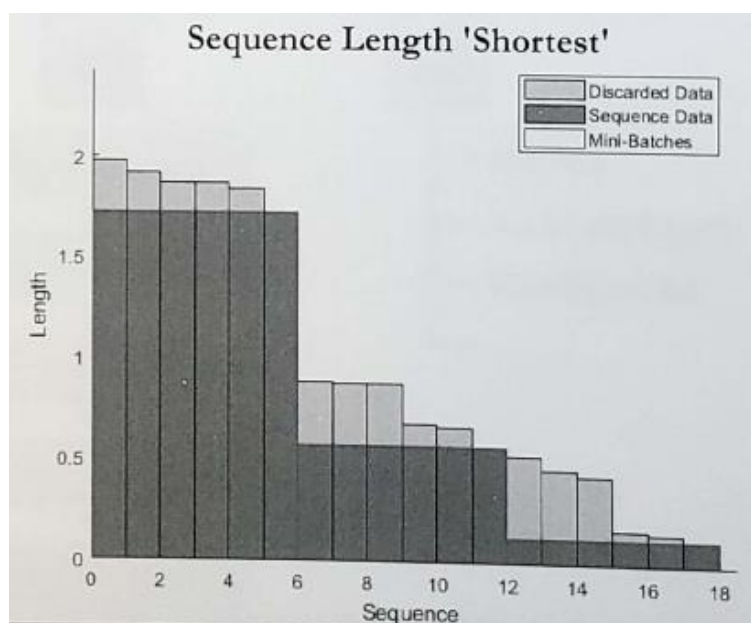
Consider these 18 sequences with varying lengths. If you trained a network with this data and 9 sequences per mini-batch, a substantial amount of padding is added to most of the sequences.



Sorting the data by sequence length reduces padding significantly. If you sort your training data, you should also set the 'Shuffle' training option to 'never'. Setting the mini-batch to 6 will also reduce padding.



You can also use the 'SequenceLength' option to modify the padding behavior. By default, sequences are padded to have the same length as the longest sequence. This behavior is shown above. Another option is to set the sequence length to the shortest sequence. This will discard some of your data.



Sequence-to-Sequence Classification

Sequence-to-sequence classification involves predicting one label for every time step in an input sequence.

Predicting one label for an entire sequence is useful when the label does not change within a sequence. But what if the label changes over time in a single sequence?

You can also use sequence-to-sequence networks to see how the network prediction changes throughout a sequence.

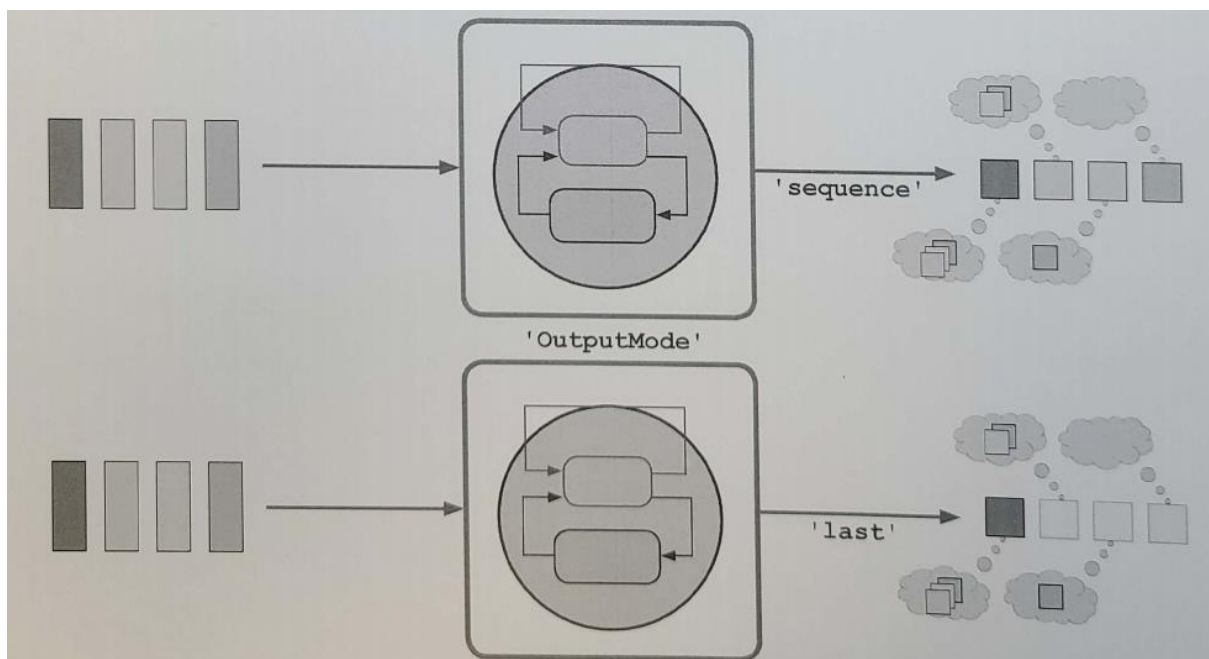
Try

View response of sequence-to-sequence data.

```
load testdata_s2s.mat
disp (YTest)
```

The responses for training a sequence-to-sequence network are stored in a N -by-1 cell array, where N is the number of input sequences.

Each cell contains a categorical sequence of labels. This sequence should contain the same number of time steps as the corresponding input sequence.



Training a Sequence-to-Sequence Network

The architecture for the sequence-to-sequence author identification network is the same as the last chapter, except the output mode should be set to **Sequence**.

```
bilstmLayer (256, 'OutputMode', 'Sequence')
```

A network has been trained that classifies text from Charles Dickens and Jane Austen. This network will output a prediction for each letter in the input text. The texts this network was trained on had only one author per sequence, such that the class does not change during the sequence.

You will can the prediction scores to see how the network confidence changes throughout the sequence. You can get the predictions scores using a second output from the **classify** function.

```
[labels, scores] = classify (net, sequence)
```

To plot the scores as lines, you can to transpose the output **scores**.

```
plot (scores')
```

You can add a legend to the plot by getting the order of the labels from the last layer in the network. The label order is stored in the **Classes** property.

```
order = net.Layers(end).Classes  
legend(order)
```

Try

Load the trained network.

```
load s2sBiLSTM.mat
```

Load the author samples and convert them into dummy variables.

```
load authorSample.mat
```

```
mDickens = dummifyText (lower (tDickens), vocab);
```

```
mAusten = dummifyText (lower (tAusten), vocab);
```

Classify the Jane Austen sequence.

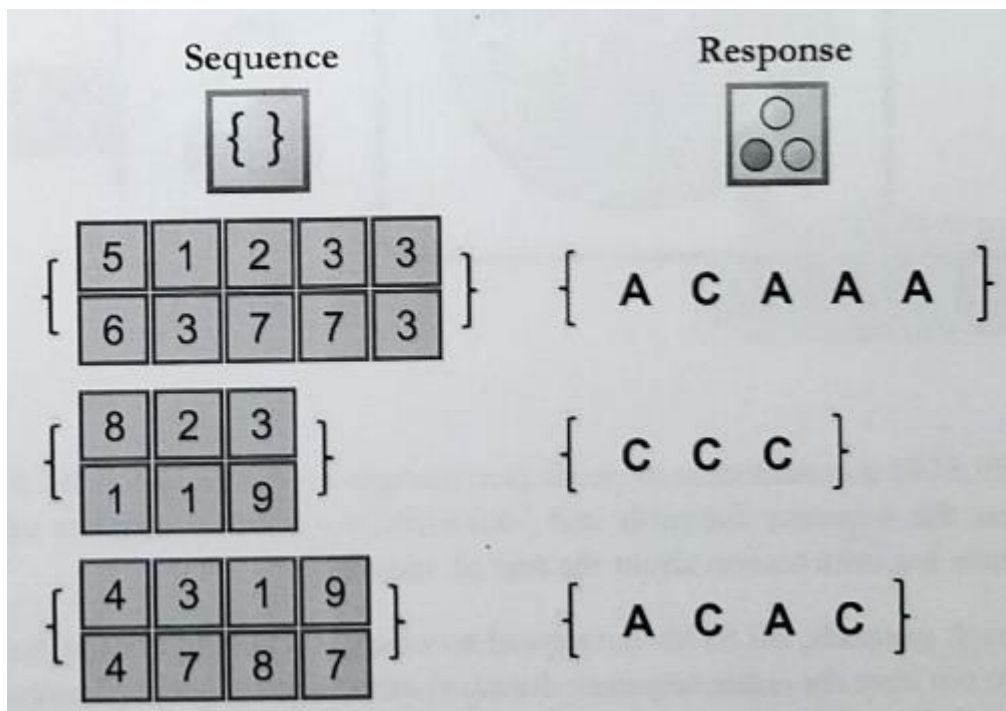
```
[authorA, scoresA] = classify (net, mAusten)
```

Plot the scores to see how they change with the sequence.

```
Plot (scoresA')
```

```
order = net.Layers (end).Classes
```

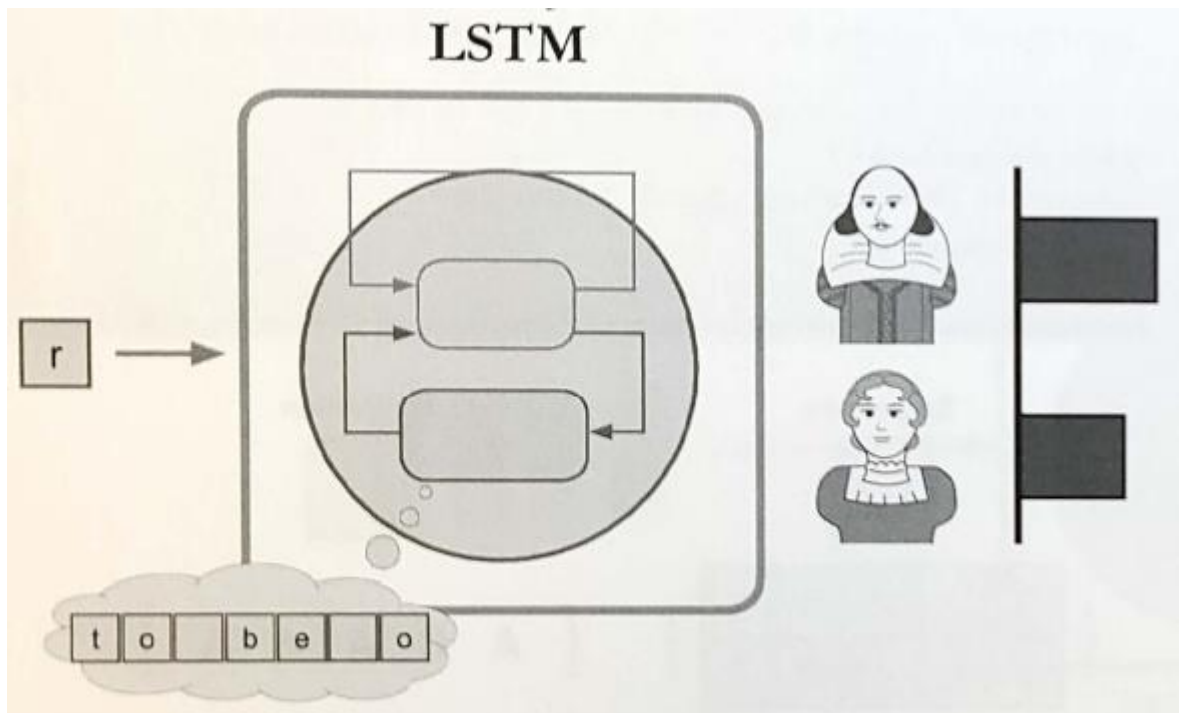
```
legend (order)
```



LSTMs and BiLSTMs

So far, all the long short-term memory network that you have used included the bidirectional long short-term memory (BiLSTM) layer. In many situations, BiLSTMs can achieve better accuracy than LSTMs because the beginning of the sequence has the context of the end of the sequence.

An LSTM layer only has information about previous letters in the sequence. At the first letter, the network has no prior information, so the prediction score only is often 0.5. Later in the sequence, the network gains enough prior information to make a confident prediction.



The BiLSTM is confident in its prediction throughout the sequence. BiLSTMs process the sequence forwards and backwards, so the first element of the sequence has information about the rest of the sequence.

Although generally, BiLSTMs have good accuracy, you should not use them if you do not have the entire sequence during classification. Sequence forecasting is one of these situations.

Try

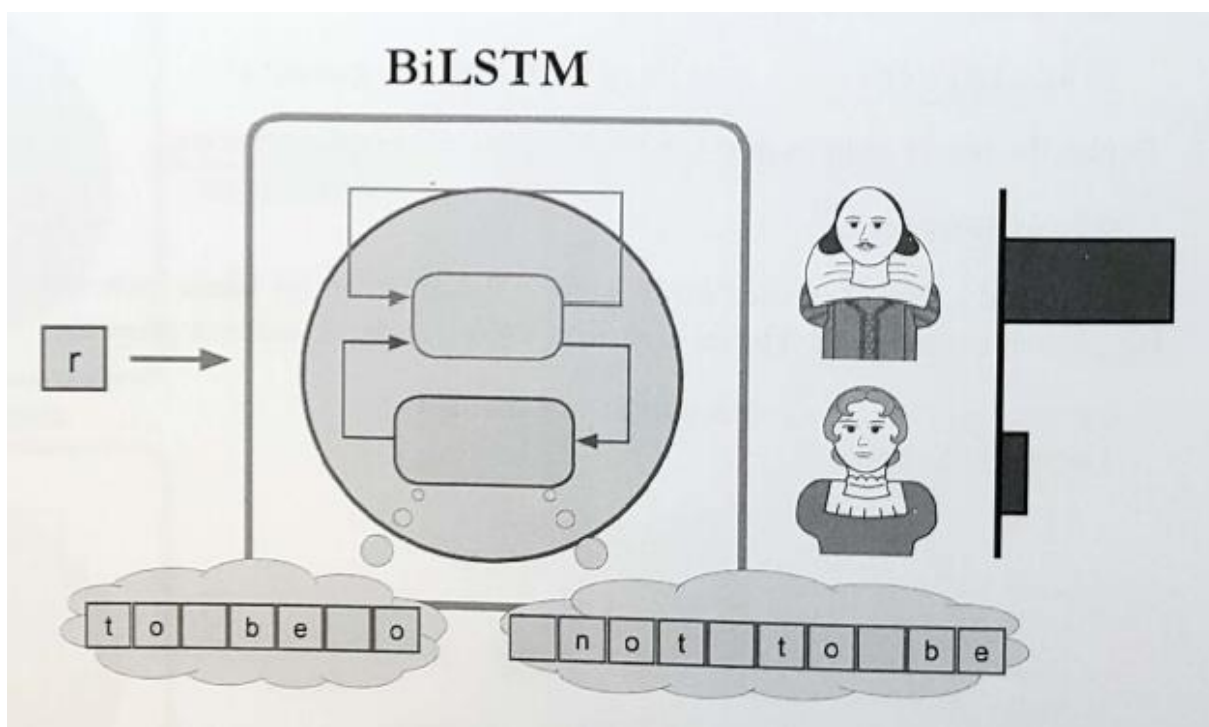
Load the LSTM network.

```
load s2sLSTM.mat
```

Classify Jane Austen to note the performance. Compare this plot to the accuracy of the BiLSTM from the previous page.

```
[author, scores] = classify (net, mAusten)
```

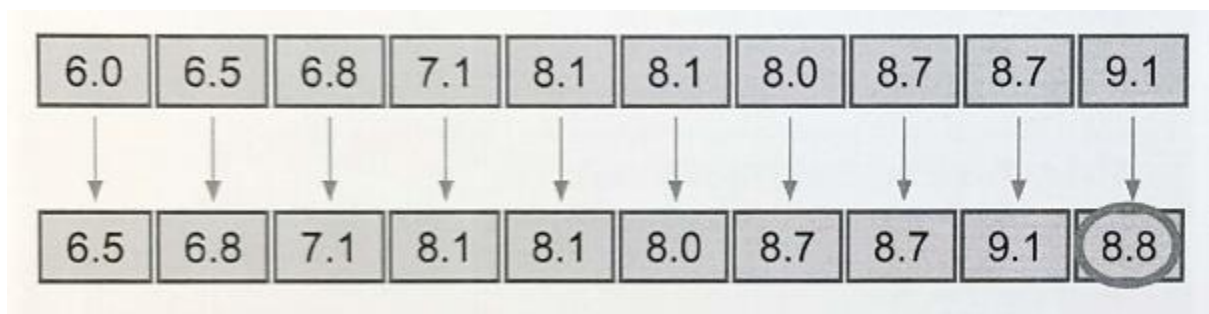
```
plot (scores')
```



Sequence Forecasting

Long short-term memory networks can be used to forecast future times steps of a sequence. Forecasting is often performed with time-series data.

The output for every element in the sequence is a prediction for the next value in the sequence.



To create the architecture and training options for the text-generating network, you will use LSTM layers in this network architecture. The inputs are the same as BiLSTM layers.

```
ly = lstmLayer(numNodes)
```

The number of inputs to the sequence input layer is equal to the number of outputs to the fully connected layer.

Use the `trainingOptions` function to make training options for generating text.

Try

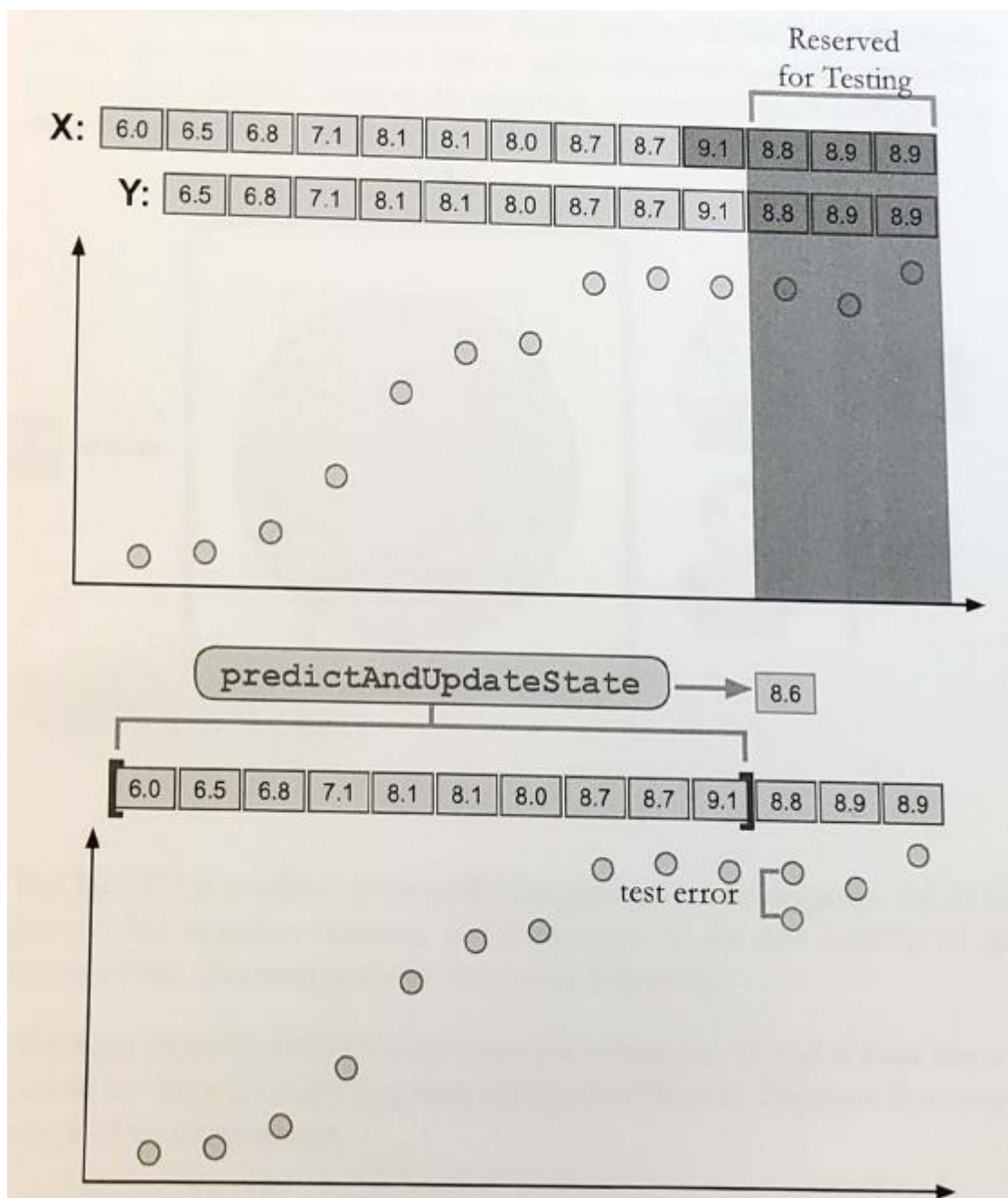
View a script that creates layers and options for the sequence forecasting networks.

[edit forecastingSetup.mlx](#)

Preparing Forecasting Data

The data used to train a forecasting network is the sequence you want to forecast. You will use a subset of the sequence to train, and the rest to test.

The input is the training sequence, except for the last value. The response is the sequence shifted by one time step.



Try

Using the Austen sample in `tAusten`, exclude the last character to make the predictor sequence.

```
Xtxt = tAusten (1: end-1)
```

Exclude the first character to make the response sequence.

```
Ytxt = tAusten (2: end)
```

Load a new vocabulary. Notice that there are 77 different characters. This LSTM will be able to process punctuation and uppercase characters.

```
load forecastingVocab.mat
```

```
Xtrain = dummifyText (Xtxt, vocab)
```

Convert the response into a categorical

```
YTrain = categorical (double (Ytxt), vocab)
```

Generating Jane Austen Text

A text-generating network was trained on about four million characters for two days.

To train the LSTM network, you can create a dummy predictor matrix from the input sequences with the helper function `dummifyText`. This network will be performing classification, so the response data needs to be of categorical type. Instead of using `dummyvar`, you can process the response data with just the `categorical` and `double` function.

The `predictAndUpdateState` function will predict the next letter for a sequence and output updated network. The new network will remember the input when it makes its next prediction.

```
[net, pred] = predictAndUpdateState (net, data)
```

The network needs an input to start its prediction. This is a sequence-to-sequence network, and so `pred` is a matrix with one column for each character from the start sequence. The last column contains the scores for the newly predicted letter.

The `predictAndUpdateState` function outputs the score for the corresponding letter in the vocabulary, rather than the next letter. A helper function `getLetterFromOutput` processes the output to find the most likely letter.

You can predict an entire sequence of letters using a `for`-loop.

If you'd like an entirely new prediction, you can reset the state of the network.

```
net = resetState (net)
```

Try

Load the text generating network.

```
load generateAustenNet.mat
```

Create a seed,

```
generatedtxt = 'It is a truth universall'
```

Use the `predictAndUpdateState` function to generate a new character.

```
[net, pred] = predictAndUpdateState (net, m)
```

Get the scores for the last (new) character, and get the character itself.

```
newm = pred (:, end)
```

```
newtxt = getLetterFromOutput (newm)
```

The generated character is a 'y', which is a good prediction, as it completes the work 'universally'. Concatenate new character with seed.

```
generatedtxt = [generatedtxt newtxt]
```

Check out the helper function used here.

```
edit getLetterFromOutput.m
```

Generate your own Jane Austen story using `for` loop with the function `AustenLoop`.

```
edit AusenLoop.mlx
```