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.

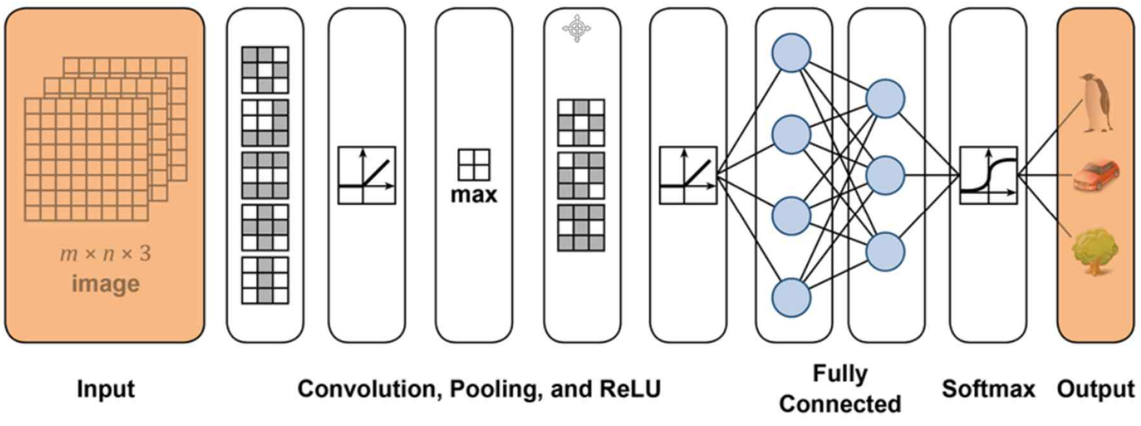
You will use the pretrained network AlexNet. This network was designed and trained by a team of deep learning researchers in 2012. AlexNet was trained for a week on a million images in 1,000 categories.

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, consist 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 state.

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
```

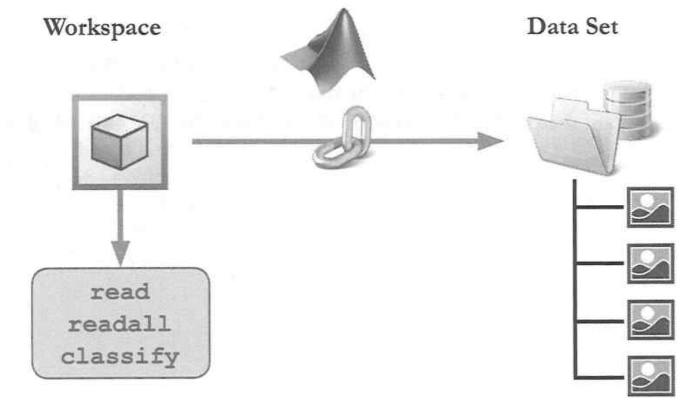
If the network is not already installed, you will receive an error with a link to install the network.

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])
```

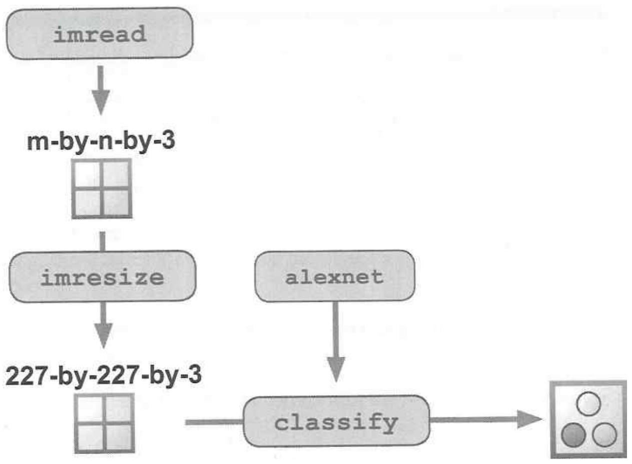
Preprocessing image can also involve modifications such as cropping, filtering, adjusting contrast, converting from an indexed image to an RGB image, etc.

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’)
```

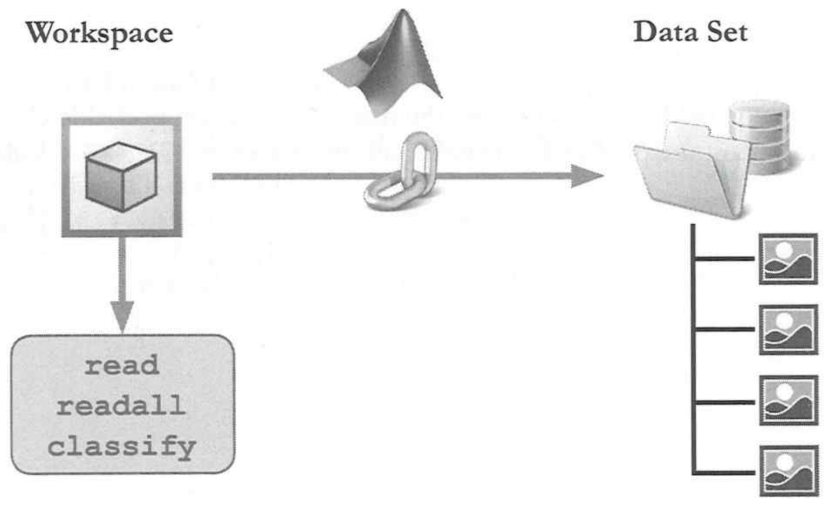
An image datastore imports images with no modifications. If you want to resize an entire collection of images, you can create an augmented image datastore.

```
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.

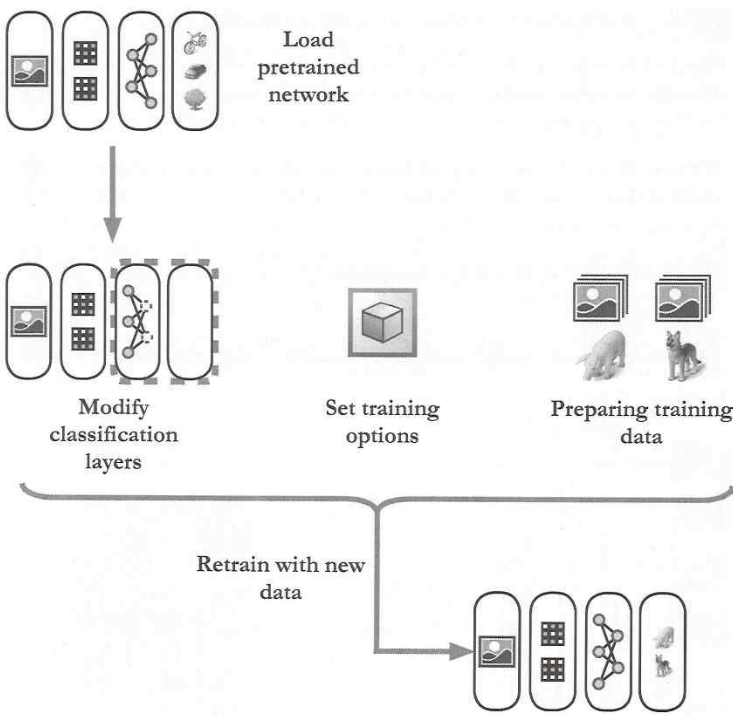
Rather than starting from scratch, you can modify a pretrained network to fit your problem. Pretrained networks have learned rich feature representations for a wide range of images. This process of taking a pretrained network, modifying it, and retraining it on new data is called transfer learning.

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

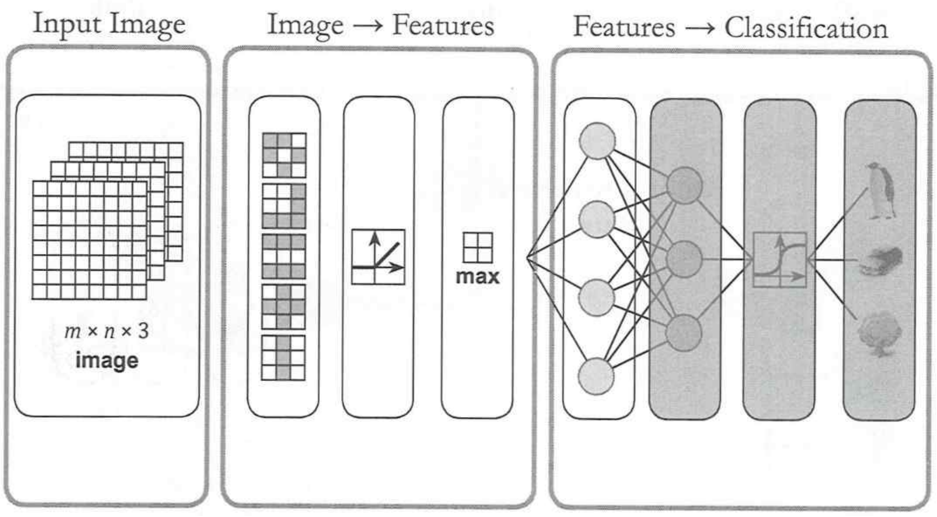
Most layers in a pretrained network extract features from an input image. The final layers map these features to your image classes. When performing transfer learning, you will typically just change the last fully connected layer and the classification layer to suit your specific application.

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.

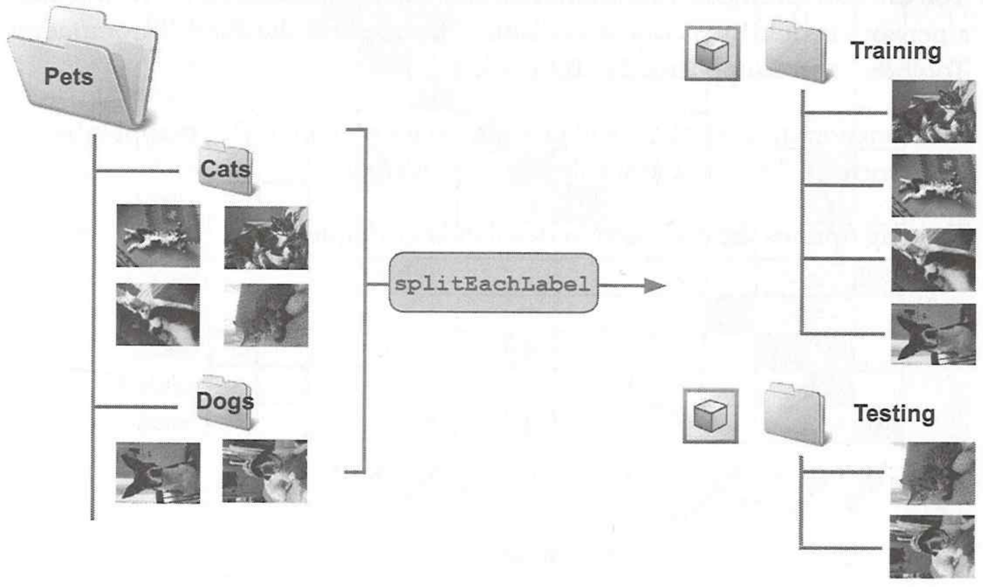
It is important for a network to have high training and testing accuracy. A test data set should represent the images a network will encounter after it is deployed. You should test a network with images the network did not see during training. This allows you to check that your network will generalize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

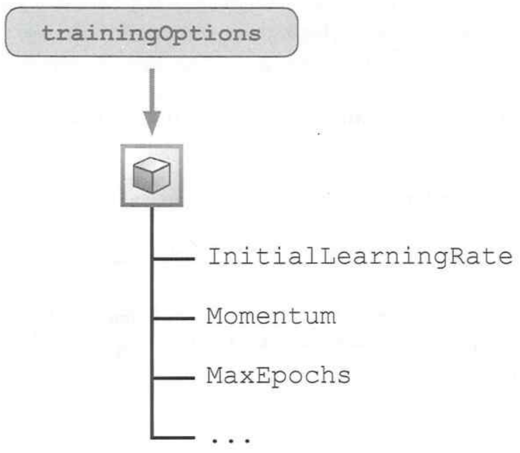
You can see more options in the documentation.

```
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^{TM} and a supported NVIDA $^{®}$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)
```

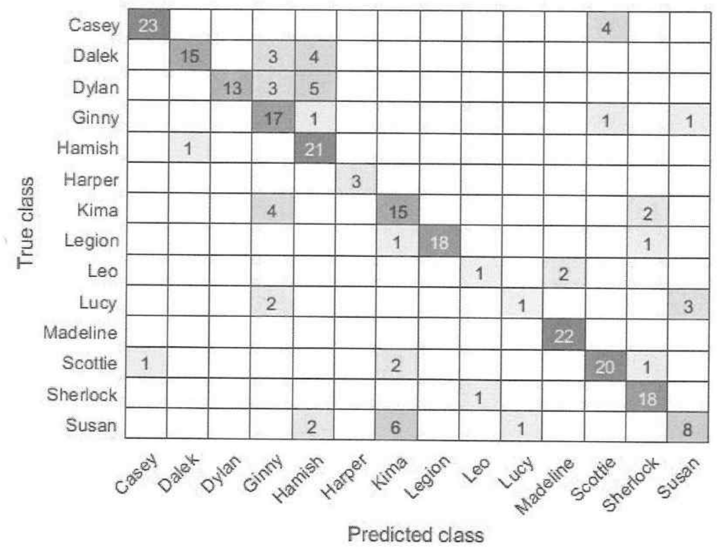
If your network makes some errors, it is a good idea to see what kind of errors they are. Dealing with multiple labels, there are different kinds of misclassifications that can be made. You can see these mistakes laid out in what is called a confusion matrix, by using the function confusionchart.

```
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)
```


True class
Predicted class
Casey
Dalek
Dylan
Ginny
Hamish
Harper
Kima
Legion
Leo
Lucy
Madeline
Scottie
Sherlock
Susan