CHAPTER 3

Finding Circles with Deep Learning

3.1 Introduction

Finding circles is a classification problem. Given a bunch of geometric shapes, we want the deep learning system to classify a shape as either a circle or something else. This is much simpler than classifying faces or digits. It is a good way to determine how well your classification system works. We will apply a convolutional network to the problem as it is the most appropriate for classifying image data.

In this chapter, we will first generate a set of image data. This will be a set of ellipses, a subset of which will be circles. Then we will build the neural net, using convolution, and train it to identify the circles. Finally, we will test the net and try some different options for training options and layer architecture.

3.2 Structure

The convolutional network consists of multiple layers. Each layer has a specific purpose. The layers may be repeated with different parameters as part of the convolutional network. The layer types we will use are

1. imageInputLayer
2. convolution2dLayer
3. batchNormalizationLayer
4. reluLayer
5. maxPooling2dLayer
6. fullyConnectedLayer
7. softmaxLayer
8. classificationLayer

M. Paluszek and S. Thomas, *Practical MATLAB Deep Learning*,
https://doi.org/10.1007/978-1-4842-5124-9_3

You can have multiple layers of each type of layer. Some convolutional nets have hundreds of layers. Krizhevsky [1] and Bai [3] give guidelines for organizing the layers. Studying the loss in the training and validation can guide you to improving your neural network.

3.2.1 imageInputLayer

This tells the network the size of the images. For example:

```
layer = imageInputLayer([28 28 3]);
```

says the image is RGB and 28 by 28 pixels.

3.2.2 convolution2dLayer

Convolution is the process of highlighting expected features in an image. This layer applies sliding convolutional filters to an image to extract features. You can specify the filters and the stride. Convolution is a matrix multiplication operation. You define the size of the matrices and their contents. For most images, like images of faces, you need multiple filters. Some types of filters are

1. Blurring filter `ones(3,3)/9`
2. Sharpening filter `[0 -1 0;-1 5 -1;0 -1 0]`
3. Horizontal Sobel filter for edge detection `[-1 -2 -1; 0 0 0; 1 2 1]`
4. Vertical Sobel filter for edge detection `[-1 0 1;-2 0 2;-1 0 1]`

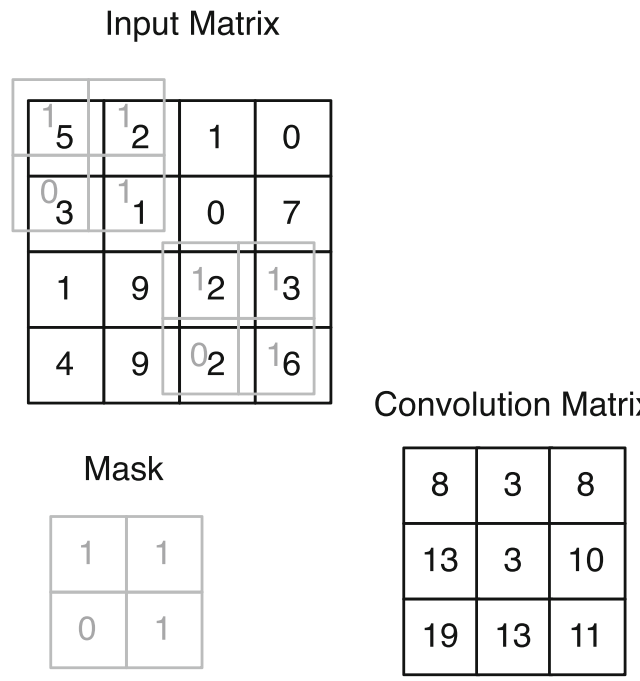
We create an n-by-n mask that we apply to an m-by-m matrix of data where m is greater than n. We start in the upper left corner of the matrix, as shown in Figure 3.1. We multiply the mask times the corresponding elements in the input matrix and do a double sum. That is the first element of the convolved output. We then move it column by column until the highest column of the mask is aligned with the highest column of the input matrix. We then return it to the first column and increment the row. We continue until we have traversed the entire input matrix and our mask is aligned with the maximum row and maximum column.

The mask represents a feature. In effect we are seeing if the feature appears in different areas of the image. Here is an example. We have a 2 by 2 mask with an L. `Convolution` demonstrates convolution.

Convolution.m

```
%% Demonstrate convolution

filter = [1 0;1 1]
image  = [0 0 0 0 0 0;...
          0 0 0 0 0 0;...
          0 0 1 0 0 0;...
          0 0 1 1 0 0;...
          0 0 0 0 0 0]
```

Figure 3.1: Convolution process showing the mask at the beginning and end of the process.




```
9
10  out = zeros(3,3);
11
12  for k = 1:4
13    for j = 1:4
14      g = k:k+1;
15      f = j:j+1;
16      out(k,j) = sum(sum(filter.*image(g,f)));
17    end
18  end
19
20  out
```

The 3 appears where the ‘‘L” is in the image.

```
>> Convolution
filter =
     1     0
     1     1
image =
     0     0     0     0     0     0
     0     0     0     0     0     0
     0     0     1     0     0     0
     0     0     1     1     0     0
```

```
     0     0     0     0     0     0
out =
     0     0     0     0
     0     1     1     0
     0     1     3     1
     0     0     1     1
```

We can have multiple masks. There is one bias and one weight for each element of the mask for each feature. In this case, the convolution works on the image itself. Convolutions can also be applied to the output of other convolutional layers or pooling layers. Pooling layers further condense the data. In deep learning, the masks are determined as part of the learning process. Each pixel in a mask has a weight and may have a bias; these are computed from the learning data. Convolution should be highlighting important features in the data. Subsequent convolution layers narrow down features. The MATLAB function has two inputs: the `filterSize`, specifying the height and width of the filters as either a scalar or an array of `[h w]`, and `numFilters`, the number of filters.

3.2.3 batchNormalizationLayer

A batch normalization layer normalizes each input channel across a mini-batch. It automatically divides up the input channel into batches. This reduces the sensitivity to the initialization.

3.2.4 reluLayer

`reluLayer` is a layer that uses the rectified linear unit activation function.

$$f(x) = \begin{cases} x & x >= 0 \\ 0 & x < 0 \end{cases} \tag{3.1}$$

Its derivative is

$$\frac{df}{dx} = \begin{cases} 1 & x >= 0 \\ 0 & x < 0 \end{cases} \tag{3.2}$$

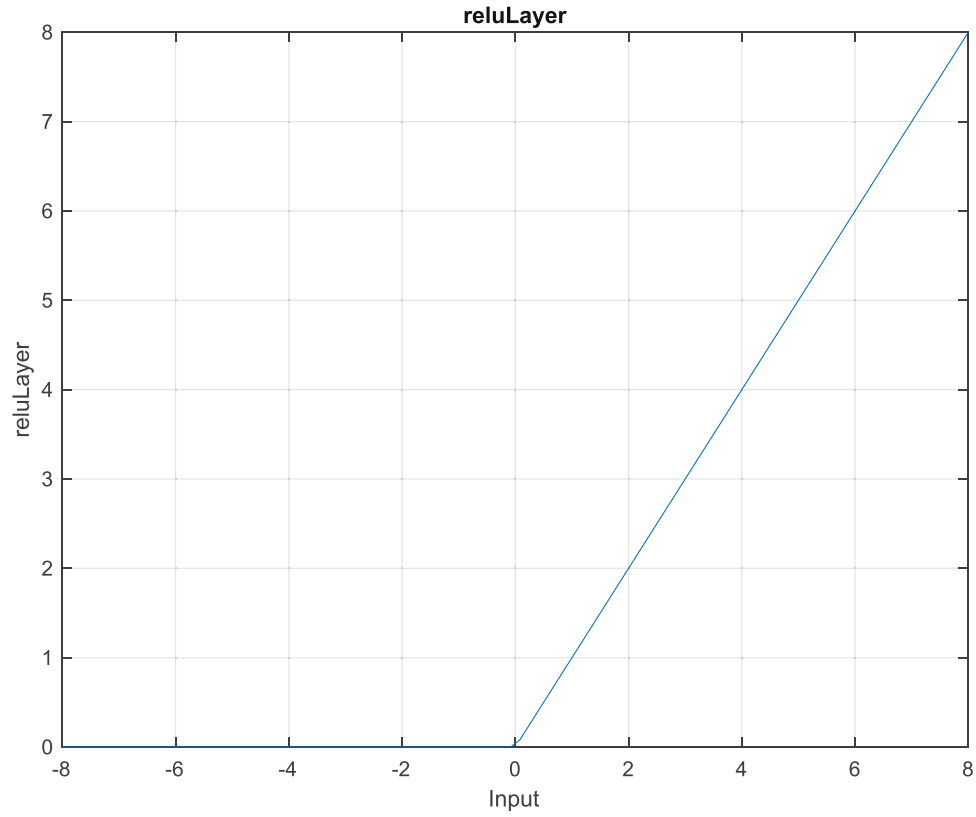
This is very fast to compute. It says that the neuron is only activated for positive values, and the activation is linear for any value greater than zero. You can adjust the activation point with a bias. This code snippet generates a plot of `reluLayer`:

```
x = linspace(-8,8);
y = x;
y(y<0) = 0;
PlotSet(x,y,'x label','Input','y label','reluLayer','plot title','
    reluLayer')
```

Figure 3.2 shows the activation function. An alternative is a leaky reluLayer where the value is not zero below zero. Now the difference in the `y` computation in the snippet:

```
x = linspace(-8,8);
y = x;
```

Figure 3.2: reluLayer.




```
y(y<0) = 0.01*x(y<0);
PlotSet(x,y,'x label','Input','y label','reluLayer','plot title','leaky
    reluLayer')
```

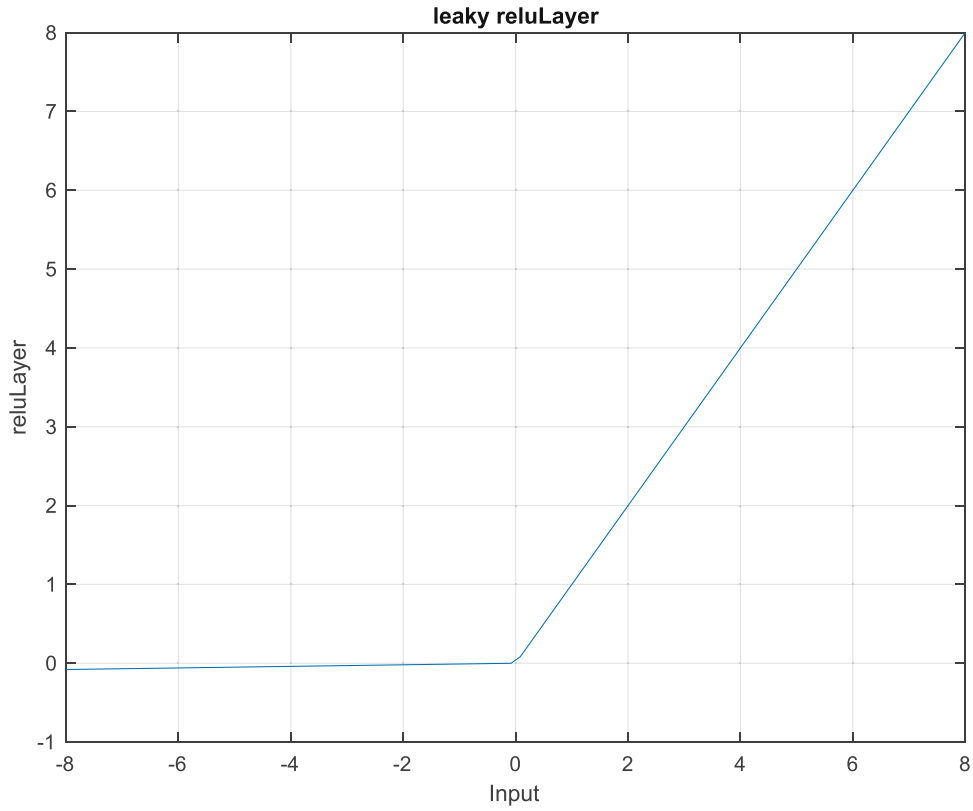
Figure 3.3 shows the leaky function. Below zero it has a slight slope.

A leaky Relu layer solves the dead Relu problem where the network stops learning because the inputs to the activation problem are below zero, or whatever the threshold might be. It should let you worry a bit less on how you initialize the network.

3.2.5 maxPooling2dLayer

`maxPooling2dLayer` creates a layer that breaks the 2D input into rectangular pooling regions and outputs the maximum value of each region. The input `poolSize` specifies the width and height of a pooling region. `poolSize` can have one element (for square regions) or two for rectangular regions. This is a way to reduce the number of inputs that need to be evaluated. Typical images have to be more than a mega-pixel in size, and it is not practical to use all as inputs. Furthermore, most images, or two-dimensional entities of any sort, don't really have enough information to require finely divided regions. You can experiment with pooling and see how it works for your application. An alternative is `averagePooling2dLayer`.

Figure 3.3: Leaky reluLayer.



3.2.6 fullyConnectedLayer

The fully connected layer connects all of the inputs to the outputs with weights and biases. For example:

```
layer = fullyConnectedLayer(10);
```

creates ten outputs from any number of inputs. You don't have to specify the inputs. Effectively, this is the equation:

$$y = ax + b \tag{3.3}$$

If there are m inputs and n outputs, b is a column bias matrix of length n and a is n by m.

3.2.7 softmaxLayer

softmax finds a maximum of a set of values using the logistic function. The softmax is the maximum value of the set

$$p_k = e^{\frac{q_k}{\sum e^{q_k}}} \tag{3.4}$$

```
>> q = [1,2,3,4,1,2,3]

q =

     1     2     3     4     1     2     3

>> d = sum(exp(q));
>> p = exp(q)/d

p =

    0.0236    0.0643    0.1747    0.4748    0.0236    0.0643    0.1747
```

In this case, the maximum is element 4 in both cases. This is just a method of smoothing the inputs. Softmax is used for multiclass classification because it guarantees a well-behaved probability distribution. Well behaved means that the sum of the probabilities is 1.

3.2.8 classificationLayer

A classification layer computes the cross-entropy loss for multiclass classification problems with mutually exclusive classes. Let us define loss. Loss is the sum of the errors in training the neural net. It is not a percentage. For classification the loss is usually the negative log likelihood, which is

$$L(y) = -\log(y) \tag{3.5}$$

where y is the output of the softmax layer.

For regression it is the residual sum of squares. A high loss means a bad fit.

Cross-entropy loss means that an item being classified can only be in one class. The number of classes is inferred from the output of the previous layer. In this problem, we have only two classes, circle or ellipse, so the number of outputs of the previous layer must be 2. Cross-entropy is the distance between the original probability distribution and what the model believes it should be. It is defined as

$$H(y, p) = -\sum_i y_i \log p_i \tag{3.6}$$

where i is the index for the class. It is a widely used replacement for mean squared error. It is used in neural nets where softmax activations are in the output layer.

3.2.9 Structuring the Layers

For our first net to identify circles, we will use the following set of layers. The first layer is the input layer, for the 32x32 images. These are relatively low-resolution images. You can visually determine which are ellipses or circles so we would expect the neural network to be able to do the same. Nonetheless, the size of the input images is an important consideration. In our case, our images are tightly cropped around the shape. In a more general problem, the subject of interest, a cat, for example, might be in a general setting.

We use a `convolution2dLayer`, `batchNormalizationLayer`, and `reluLayer` in sequence, with a pool layer in between. There are three sets of convolution layers, each with an increasing number of filters. The output set of layers consists of a `fullyConnectedLayer`, `softmaxLayer`, and finally, the `classificationLayer`.

EllipsesNeuralNet.m

```
%% Define the layers for the net
% This gives the structure of the convolutional neural net
layers = [
    imageInputLayer(size(img))

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

    maxPooling2dLayer(2,'Stride',2)

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

    maxPooling2dLayer(2,'Stride',2)

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

    fullyConnectedLayer(2)
    softmaxLayer
    classificationLayer
        ];
```

3.3 Generating Data: Ellipses and Circles

3.3.1 Problem

We want to generate images of ellipses and circles of arbitrary sizes and with different thicknesses in MATLAB.

3.3.2 Solution

Write a MATLAB function to draw circles and ellipses and extract image data from the figure window. Our function will create a set of ellipses and a fixed number of perfect circles as specified by the user. The actual plot and the resulting downsized image will both be shown in a figure window so you can track progress and verify that the images look as expected.

3.3.3 How It Works

This is implemented in `GenerateEllipses.m`. The output of the function is a cell array with both the ellipse data and a set of image data obtained from a MATLAB figure using `getframe`. The function also outputs the type of image, that is, the ''truth'' data.

GenerateEllipses.m

```
%% GENERATEELLIPSES Generate random ellipses
%% Form
%  [d, v] = GenerateEllipses(a,b,phi,t,n,nC,nP)
%% Description
% Generates random ellipses given a range of sizes and max rotation.
    The number
% of ellipses and circles must be specified; the total number generated
    is their
% sum. Opens a figure which displays the ellipse images in an animation
    after
% they are generated.
%% Inputs
%  a    (1,2) Range of a sizes of ellipse
%  b    (1,2) Range of b sizes of ellipse
%  phi  (1,1) Max rotation angle of ellipse
%  t    (1,1) Max line thickness in the plot of the circle
%  n    (1,1) Number of ellipses
%  nC   (1,1) Number of circles
%  nP   (1,1) Number of  pixels, image is nP by nP
%
%% Outputs
%  d      {:,2} Ellipse data and image frames
%  v      (1,:) Boolean for circles, 1 (circle) or 0 (ellipse)
```

The first section of the code generates random ellipses and circles. They are all centered in the image.

GenerateEllipses.m

```
nE    = n+nC;
d     = cell(nE,2);
r     = 0.5*(mean(a) + mean(b))*rand(1,nC)+a(1);
a     = (a(2)-a(1))*rand(1,n) + a(1);
b     = (b(2)-b(1))*rand(1,n) + b(1);
phi   = phi*rand(1,n);
cP    = cos(phi);
sP    = sin(phi);
theta = linspace(0,2*pi);
c     = cos(theta);
s     = sin(theta);
m     = length(c);
t     = 0.5+(t-0.5)*rand(1,nE);
aMax  = max([a(:);b(:);r(:)]);

% Generate circles
for k = 1:nC
  d{k,1} = r(k)*[c;s];
end

% Generate ellipses
for k = 1:n
  d{k+nC,1} = [cP(k) sP(k);-sP(k) cP(k)]*[a(k)*c;b(k)*s];
end

% True if the object is a circle
v       = zeros(1,nE);
v(1:nC) = 1;
```

The next section produces a 3D plot showing all the ellipses and circles. This is just to show you what you have produced. The code puts all the ellipses between z $\pm$ 1. You might want to adjust this when generating larger numbers of ellipses.

```
% 3D Plot
NewFigure('Ellipses');
z  = -1;
dZ = 2*abs(z)/nE;
o  = ones(1,m);
for k = 1:length(d)
  z   = z + dZ;
  zA  = z*o;
  plot3(d{k}(1,:),d{k}(2,:),zA,'linewidth',t(k));
  hold on
end
grid on
rotate3d on
```

The next section converts the frames to `nP` by `nP` sized images in grayscale. We set the figure and the axis to be square, and set the axis to `'equal'`, so that the circles will have the

correct aspect ratio and in fact be circular in the images. Otherwise, they would also appear as ellipses, and our neural net would not be able to categorize them. This code block also draws the resulting resized image on the right-hand side of the window, with a title showing the current step. There is a brief pause between each step. In effect, it is an animation that serves to inform you of the script's progress.

```
% Create images - this might take a while for a lot of images
f = figure('Name','Images','visible','on','color',[1 1 1]);
ax1 = subplot(1,2,1,'Parent', f, 'box', 'off','color',[1 1 1] );
ax2 = subplot(1,2,2,'Parent',f); grid on;
for k = 1:length(d)
  % Plot the ellipse and get the image from the frame
  plot(ax1,d{k}(1,:),d{k}(2,:),'linewidth',t(k),'color','k');
  axis(ax1,'off'); axis(ax1,'equal');
  axis(ax1,aMax*[-1 1 -1 1])
  frame  = getframe(ax1); % this call is what takes time
  imSmall = rgb2gray(imresize(frame2im(frame),[nP nP]));
  d{k,2} = imSmall;
  % plot the resulting scaled image in the second axes
        imagesc(ax2,d{k,2});
  axis(ax2,'equal')
        colormap(ax2,'gray');
  title(ax2,['Image ' num2str(k)])
        set(ax2,'xtick',1:nP)
        set(ax2,'ytick',1:nP)
        colorbar(ax2)
        pause(0.2)
end
close(f)
```

The conversion is `rgb2gray(imresize(frame2im(frame),[nP nP]))`, which performs these steps:

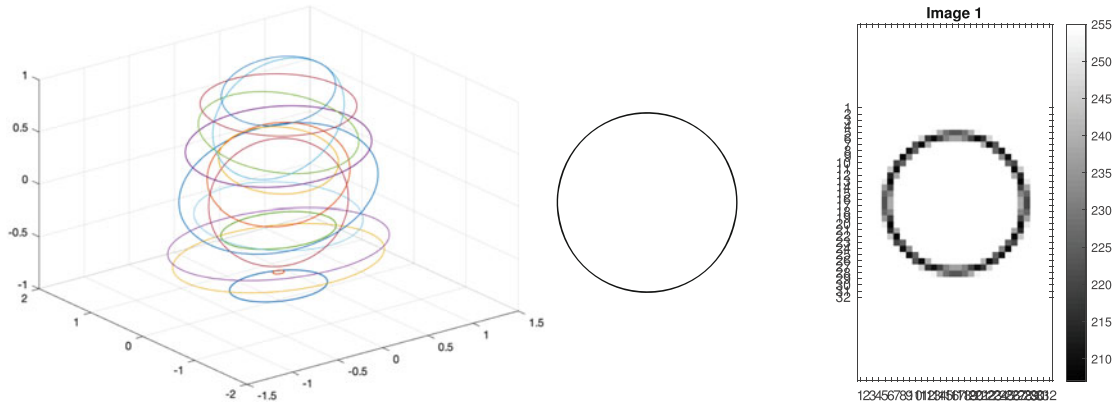
1. Get the frame with `frame2im`
2. Resize to `nP` by `nP` using `imresize`
3. Convert to grayscale using `rgb2gray`

Note that the image data originally ranges from 0 (black) to 255 (white), but is averaged to ligher gray pixels during the resize operation. The colorbar in the progress window shows you the span of the output image. The image looks black as before since it is plotted with `imagesc`, which automatically scales the image to use the entire colormap—in this case, the gray colormap.

The built-in demo generates ten ellipses and five circles.

```
function Demo

a   = [0.5 1];
```

Figure 3.4: Ellipses and a resulting image.




```
4  b    = [1 2];
5  phi  = pi/4;
6  t    = 3;
7  n    = 10;
8  nC   = 5;
9  nP   = 32;
10
11 GenerateEllipses(a,b,phi,t,n,nC,nP);
```

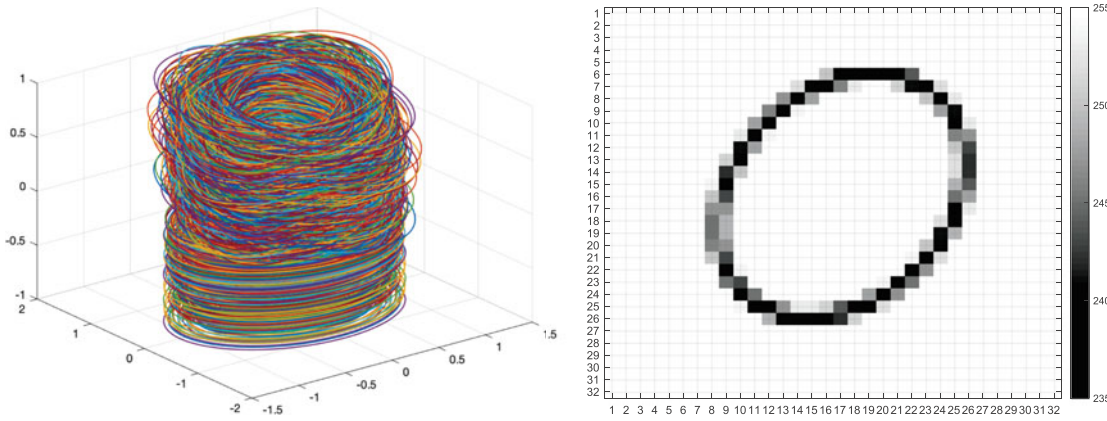
Figure 3.4 shows the generated ellipses and the first image displayed.

The script `CreateEllipses.m` generates 100 ellipses and 100 circles and stores them in the `Ellipses` folder along with the type of each image. Note that we have to do a small trick with the filename. If we simply append the image number to the filename, 1, 2, 3, ... 200, the images will not be in this order in the datastore; in alphabetical order, the images would be sorted as 1, 10, 100, 101, and so on. In order to have the filenames in alphabetical order match the order we are storing with the type, we generate a number a factor of 10 higher than the number of images and add it to the image index before appending it to the file. Now we have image 1001, 1001, and so on.

CreateEllipses.m

```
1  %% Create ellipses to train and test the deep learning algorithm
2  % The ellipse images are saved  as jpegs in the folder Ellipses.
3
4  % Parameters
5  nEllipses = 1000;
6  nCircles  = 1000;
7  nBits     = 32;
8  maxAngle  = pi/4;
9  rangeA    = [0.5 1];
10 rangeB    = [1 2];
11 maxThick  = 3.0;
```

Figure 3.5: Ellipses and a resulting image. 100 circles and 100 ellipse images are stored.



```
tic
[s, t] = GenerateEllipses(rangeA,rangeB,maxAngle,maxThick,nEllipses,
    nCircles,nBits);
toc
cd Ellipses
kAdd = 10^ceil(log10(nEllipses+nCircles)); % to make a serial number
for k = 1:length(s)
  imwrite(s{k,2},sprintf('Ellipse%d.jpg',k+kAdd));
end

% Save the labels
save('Type','t');
cd ..
```

The graphical output is shown in Figure 3.5. It first displays the 100 circles and then the 100 ellipses. It takes some time for the script to generate all the images.

If you open the resulting jpegs, you will see that they are in fact 32x32 images with gray circles and ellipses.

This recipe provides the data that will be used for the deep learning examples in the following sections. You must run `CreateEllipses.m` before you can run the neural net examples.

3.4 Training and Testing

3.4.1 Problem

We want to train and test our deep learning algorithm on a wide range of ellipses and circles.

3.4.2 Solution

The script that creates, trains, and tests the net is `EllipsesNeuralNet.m`.

3.4.3 How It Works

First we need to load in the generated images. The script in Recipe 3.3 generates 200 files. Half are circles and half ellipses. We will load them into an image datastore. We display a few images from the set to make sure we have the correct data and it is tagged correctly—that is, that the files have been correctly matched to their type, circle (1) or ellipse (0).

EllipsesNeuralNet.m

```
%% Get the images
cd Ellipses
type = load('Type');
cd ..
t    = categorical(type.t);
imds = imageDatastore('Ellipses','labels',t);

labelCount = countEachLabel(imds);

% Display a few ellipses
NewFigure('Ellipses')
n = 4;
m = 5;
ks = sort(randi(length(type.t),1,n*m)); % random selection
for i = 1:n*m
        subplot(n,m,i);
        imshow(imds.Files{ks(i)});
  title(sprintf('Image %d: %d',ks(i),type.t(ks(i))))
end

% We need the size of the images for the input layer
img = readimage(imds,1);
```

Once we have the data, we need to create the training and testing sets. We have 100 files with each label (0 or 1, for an ellipse or circle). We create a training set of 80% of the files and reserve the remaining as a test set using `splitEachLabel`. Labels could be names, like ‘‘circle’’ and ‘‘ellipse.’’ You are generally better off with descriptive ‘‘labels.’’ After all, a 0 or 1 could mean anything. The MATLAB software handles many types of labels.

EllipsesNeuralNet.m

```
% Split the data into training and testing sets
fracTrain             = 0.8;
[imdsTrain,imdsTest]  = splitEachLabel(imds,fracTrain,'randomize');
```

The layers of the net are defined as in the previous recipe. The next step is training. The `trainNetwork` function takes the data, set of layers, and options, runs the specified training

algorithm, and returns the trained network. This network is then invoked with the `classify` function, as shown later in this recipe. This network is a series network. The network has other methods which you can read about in the MATLAB documentation.

EllipsesNeuralNet.m

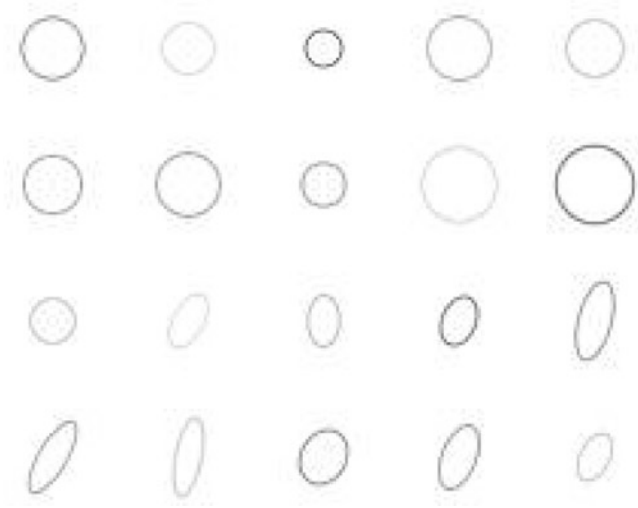
```
%% Training
% The mini-batch size should be less than the data set size; the mini-
    batch is
% used at each training iteration to evaluate gradients and update the
    weights.
options = trainingOptions('sgdm', ...
    'InitialLearnRate',0.01, ...
    'MiniBatchSize',16, ...
    'MaxEpochs',5, ...
    'Shuffle','every-epoch', ...
    'ValidationData',imdsTest, ...
    'ValidationFrequency',2, ...
    'Verbose',false, ...
    'Plots','training-progress');

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

Figure 3.6 shows some of the ellipses used in the testing and training. They were obtained randomly from the set using `randi`.

The training options need explanation. This is a subset of the parameter pairs available for `trainingOptions`. The first input to the function, `'sgdm'`, specifies the training method. There are three to choose from:

1. `'sgdm'` —Stochastic gradient descent with momentum

Figure 3.6: A subset of the ellipses used in the training and testing.



2. `'adam'` —Adaptive moment estimation (ADAM)

3. `'rmsprop'` —Root mean square propagation (RMSProp)

The `'InitialLearnRate'` is the initial speed of learning. Higher learn rates mean faster learning, but the training may get stuck in a suboptimal point. The default rate for `sgdm` is 0.01. `'MaxEpochs'` is the maximum number of epochs to be used in the training. In each epoch, the training sees the entire training set, in batches of `MiniBatchSize`. The number of iterations in each epoch is therefore determined by the amount of data in the set and the `MiniBatchSize`. We are using a smaller data set so we reduce the `MiniBatchSize` from the default of 128 to 16, which will give us 10 iterations per epoch. `'Shuffle'` tells the training how often to shuffle the training data. If you don't shuffle, the data will always be used in the same order. Shuffling should improve the accuracy of the trained neural network. `'ValidationFrequency'` is how often, in number of iterations, `'ValidationData'` is used to test the training. This validation data will be the data we reserved for testing when using `splitEachLabel`. The default frequency is every 30 iterations. We can use a validation frequency for our small problem of one, two, or five iterations. `'Verbose'` means print out status information to the command window. `'Plots'` only has the option `'training-progress'` (besides `'none'`). This is the plot you see in this chapter.

"Padding" in the `convolution2dLayer` means that the output size is `ceil(inputSize/stride)`, where `inputSize` is the height and width of the input.

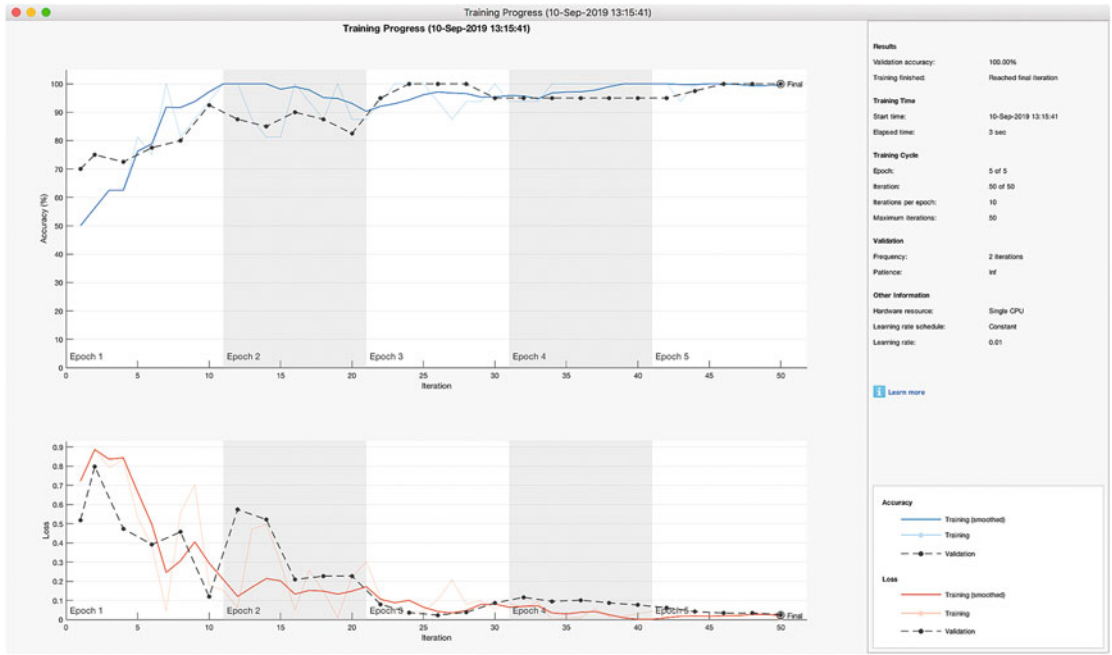
The training window runs in real time with the training process. The window is shown in Figure 3.7. Our network starts with a 50% accuracy since we only have two classes, circles and ellipses. Our accuracy approaches 100% in just five epochs, indicating that our classes of images are readily distinguishable. The loss plot shows how well we are doing. The lower the loss, the better the neural net. The loss plot approaches zero as the accuracy approaches 100%. In this case the validation data loss and the training data loss are about the same. This indicates good fitting of the neural net with the data. If the validation data loss is greater than the training data loss, the neural net is overfitting the data. Overfitting happens when you have an overly complex neural network. You can fit the training data, but it may not perform very well with new data, such as the validation data. For example, if you have a system which really is linear, and you fit it to a cubic equation, it might fit the data well but doesn't really model the real system. If the loss is greater than the validation data loss, your neural net is underfitting. Underfitting happens when your neural net is too simple. The goal is to make both zero.

Finally, we test the net. Remember that this is a classification problem. An image is either an ellipse or a circle. We therefore use `classify` to implement the network. `predLabels` is the output of the net, that is, the predicted labels for the test data. This is compared to the truth labels from the datastore to compute an accuracy.

EllipsesNeuralNet.m

```
1
2   %% Test the neural net
3  predLabels  = classify(net,imdsTest);
```

Figure 3.7: The training window with a learn rate of 0.01. The top plot is the accuracy expressed as a percentage.

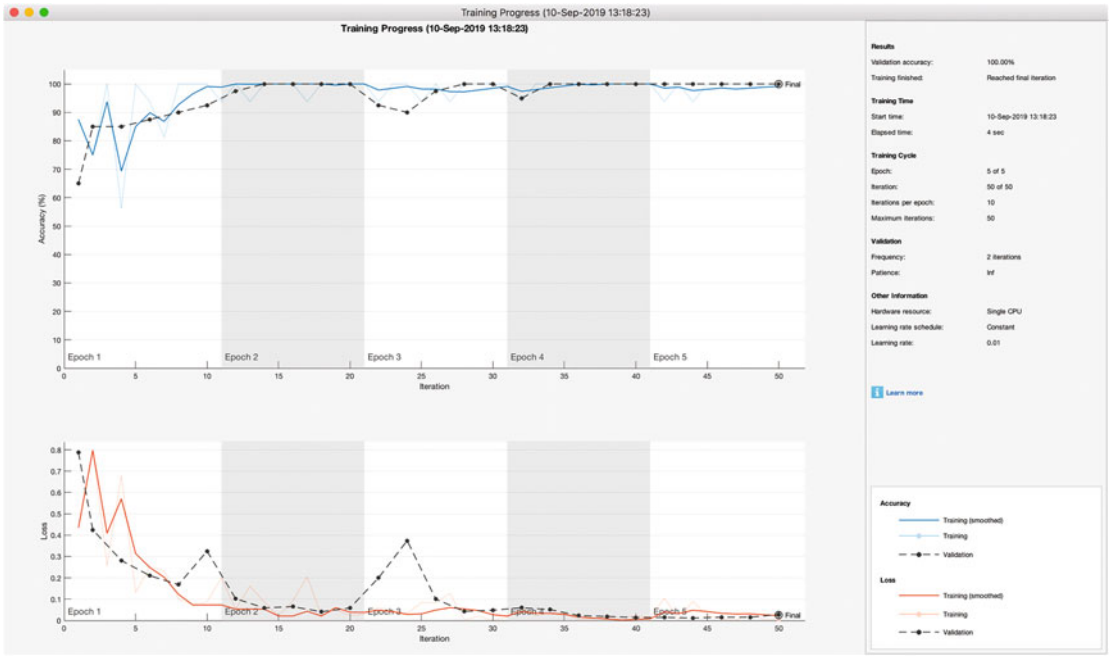



```
4  testLabels  = imdsTest.Labels;
5
6  accuracy = sum(predLabels == testLabels)/numel(testLabels);
```

The output of the testing is shown in the following. The accuracy of this run was 97.50%. On some runs, the net reaches 100%.

```
>> EllipsesNeuralNet

ans =

  Figure (1: Ellipses) with properties:

      Number: 1
        Name: 'Ellipses'
       Color: [0.9400 0.9400 0.9400]
    Position: [560 528 560 420]
       Units: 'pixels'

  Show all properties

Accuracy is    97.50%
```

Figure 3.8: The training window with a learn rate of 0.01 and a leaky reluLayer.



We can try different activation functions. `EllipsesNeuralNetLeaky` shows a leaky reluLayer. We replaced `reluLayer` with `leakyReluLayer`. The output is similar, but in this case, learning was achieved even faster than before. See Figure 3.8 for a training run.

EllipsesNeuralNetLeaky.m

```
% This gives the structure of the convolutional neural net
layers = [
    imageInputLayer(size(img))

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

    maxPooling2dLayer(2,'Stride',2)

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

    maxPooling2dLayer(2,'Stride',2)

    convolution2dLayer(3,32,'Padding','same')
    batchNormalizationLayer
    leakyReluLayer
```

```
20
21      fullyConnectedLayer(2)
22      softmaxLayer
23      classificationLayer
24          ];
```

The output with the leaky layer is shown as follows.

```
>> EllipsesNeuralNetLeaky

ans =

  Figure (1: Ellipses) with properties:

      Number: 1
        Name: 'Ellipses'
       Color: [0.9400 0.9400 0.9400]
    Position: [560 528 560 420]
       Units: 'pixels'

  Show all properties

Accuracy is    84.25%
```

We can try fewer layers. `EllipsesNeuralNetOneLayer` has only one set of layers.

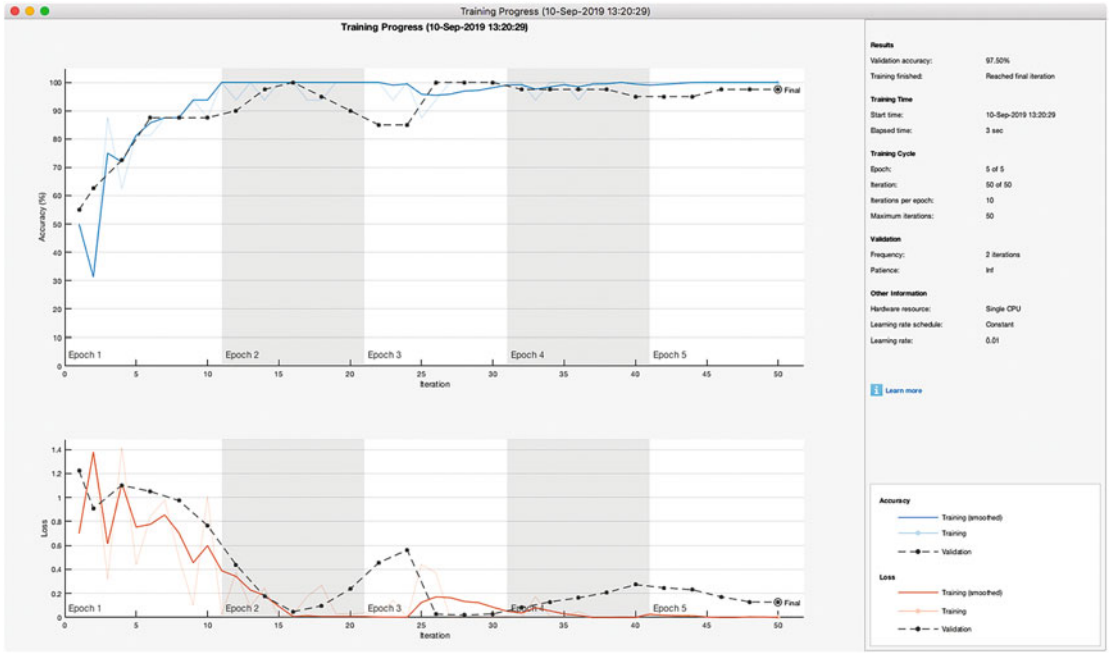
EllipsesNeuralNetOneLayer.m

```
1  %% Define the layers for the net
2  % This gives the structure of the convolutional neural net
3  layers = [
4      imageInputLayer(size(img))
5
6      convolution2dLayer(3,8,'Padding','same')
7      batchNormalizationLayer
8      reluLayer
9
10     fullyConnectedLayer(2)
11     softmaxLayer
12     classificationLayer
13         ];
14
15 analyzeNetwork(layers)
```

The results shown in Figure 3.9 with only one set of layers is still pretty good. This shows that you need to try different options with your net architecture as well. With this size of a problem, multiple layers are not buying very much.

```
>> EllipsesNeuralNetOneLayer
ans =
  Figure (2: Ellipses) with properties:
```

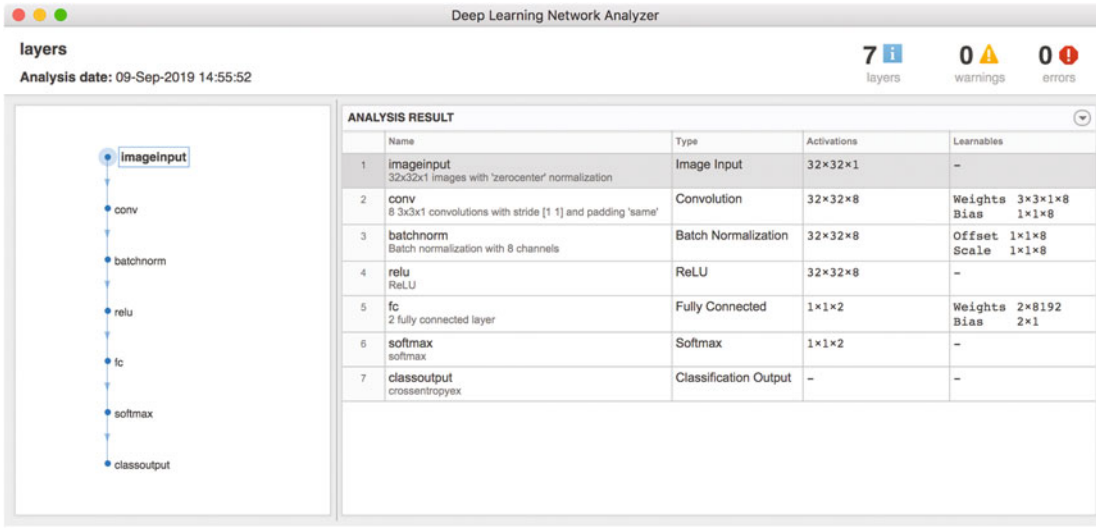
Figure 3.9: The training window for a net with one set of layers.




```
      Number: 2
        Name: 'Ellipses'
       Color: [0.9400 0.9400 0.9400]
    Position: [560 528 560 420]
       Units: 'pixels'

  Show all properties
Accuracy is    87.25%
```

The one-set network is short enough that the whole thing can be visualized inside the window of `analyzeNetwork`, as in Figure 3.10. This function will check your layer architecture before you start training and alert you to any errors. The size of the activations and ‘‘Learnables’’ is displayed explicitly.

Figure 3.10: The analyze window for the one-set convolutional network.



That concludes this chapter. We both generated our own image data and trained a neural net to classify features in our images! In this example, we were able to achieve 100% accuracy, but not after some debugging was required with creating and naming the images. It is critical to carefully examine your training and test data to ensure it contains the features you wish to identify. You should be prepared to experiment with your layers and training parameters as you develop nets for different problems.