ex_nn

December 3, 2021

1 Machine Learning: Artificial Neural Networks

Instructions ____

This file contains code that helps you get started. You will need to complete the following functions

- `predict.m`
- `sigmoidGradient.m`
- `randInitializeWeights.m`
- `nnCostFunction.m`

For this exercise, you will not need to change any code in this file, or any other files other than those mentioned above.

1.1 Import the required packages

[1]:
```
import scipy.io
import numpy as np

from predict import predict
from displayData import displayData
from sigmoidGradient import sigmoidGradient
from randInitializeWeights import randInitializeWeights
from nnCostFunction import nnCostFunction
from checkNNGradients import checkNNGradients
from fmincg import fmincg
```

1.2 Setup the parameters you will use for this exercise

[2]:
```
input_layer_size = 400;      # 20x20 Input Images of Digits
hidden_layer_size = 25;      # 25 hidden units
num_labels = 10;             # 10 labels, from 0 to 9
                             # (note that we have mapped "0" to label 9 to follow
                             # the same structure used in the MatLab version)
```

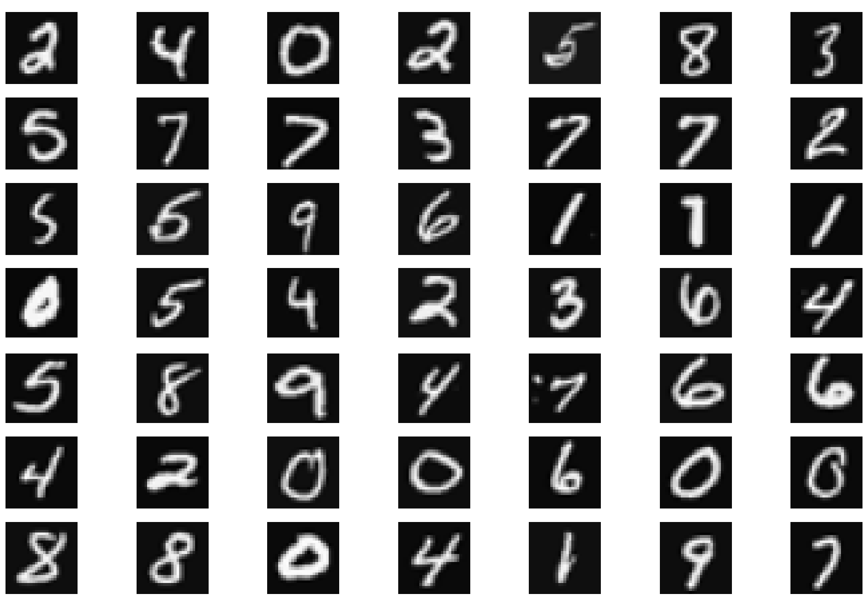
2 ============ Part 1: Loading and Visualizing Data ==============

We start the exercise by first loading and visualizing the dataset. You will be working with a dataset that contains handwritten digits.

2.1 Load Training Data

[83]:
```
print('Loading and Visualizing Data ...')

mat = scipy.io.loadmat('digitdata.mat')
X = mat['X']
y = mat['y']
y = np.squeeze(y)
m, _ = np.shape(X)

# Randomly select 100 data points to display
sel = np.random.choice(range(X.shape[0]), 49)
sel = X[sel,:]

displayData(sel)
```

```
Loading and Visualizing Data …
```



3 ================== Part 2: Loading Pameters ==================

In this part of the exercise, we load some pre-initialized neural network parameters.

[4]:
```
print('Loading Saved Neural Network Parameters ...')

# Load the weights into variables Theta1 and Theta2
mat = scipy.io.loadmat('debugweights.mat');

# Unroll parameters
Theta1 = mat['Theta1']
Theta1_1d = np.reshape(Theta1, Theta1.size, order='F')
Theta2 = mat['Theta2']
Theta2_1d = np.reshape(Theta2, Theta2.size, order='F')

nn_params = np.hstack((Theta1_1d, Theta2_1d))
```

```
Loading Saved Neural Network Parameters …
```

4 =================== Part 3: Implement Predict ===================

After training the neural network, we would like to use it to predict the labels. You will now implement the "predict" function to use the neural network to predict the labels of the training set. This lets you compute the training set accuracy.

[5]:
```
pred = predict(Theta1, Theta2, X);
print('Training Set Accuracy: ', (pred == y).mean()*100)
```

```
Training Set Accuracy:  97.52
```

4.1 Testing (you can skip this block)

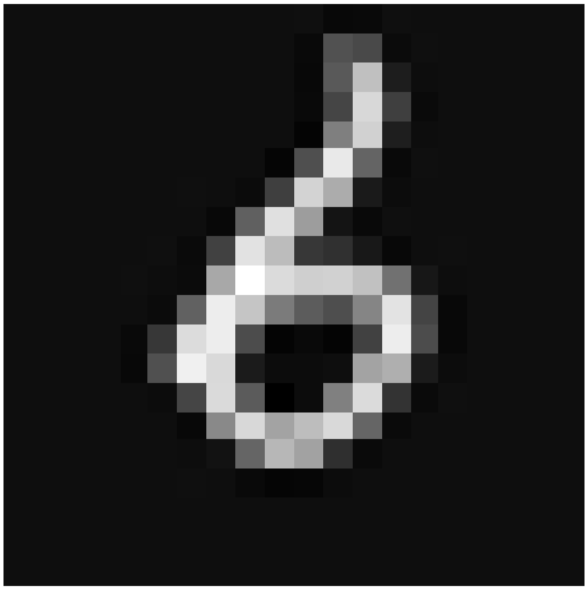
To give you an idea of the network's output, you can also run through the examples one at the a time to see what it is predicting. Run the code in the following block to view examples.

NOTE: to avoid the printing of all the sample instances, you can replace *range(m)* with a small number

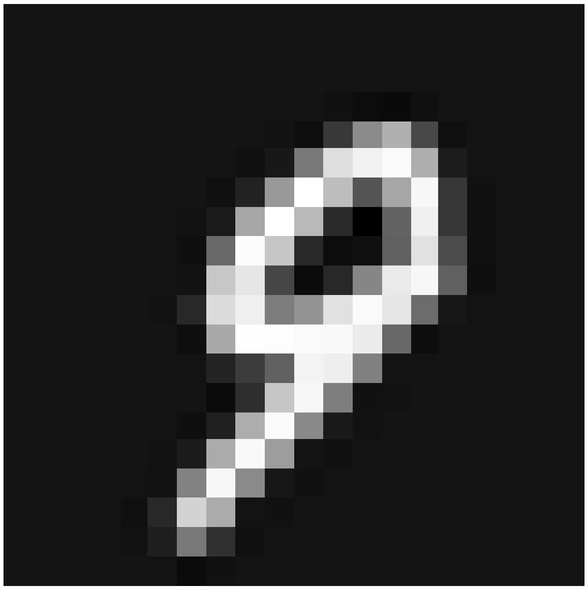
[6]:
```
#  Randomly permute examples
rp = np.random.permutation(m)

for i in range(m):
    print(i)
    # Display
    print('Displaying Example Image')
    tmp = np.transpose(np.expand_dims(X[rp[i], :], axis=1))
    displayData(tmp)
```

```python
    pred = predict(Theta1, Theta2, tmp)
    print('Neural Network Prediction: ', pred, '(digit ', pred%10, ')')
```

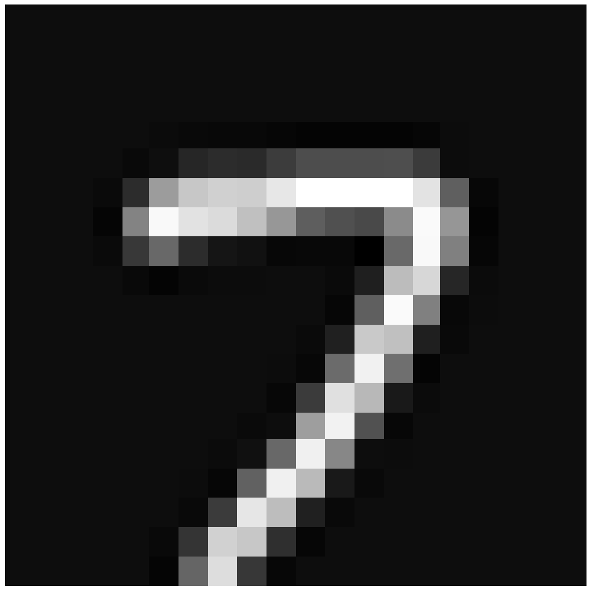
```
0
Displaying Example Image
```



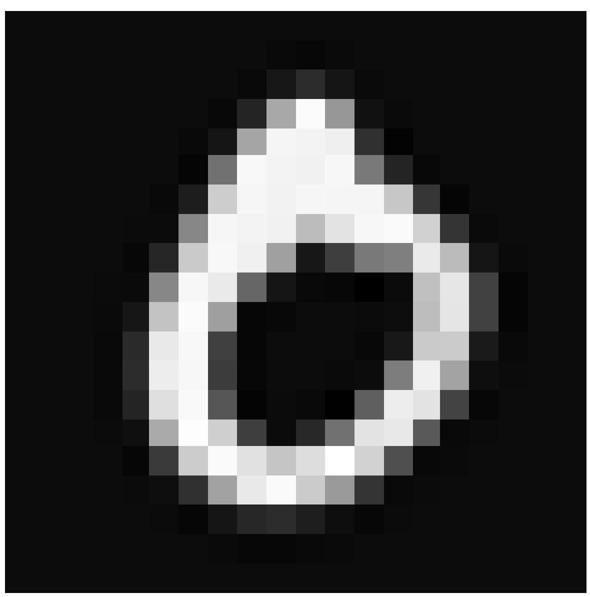
```
Neural Network Prediction:  [6.] (digit  [6.] )
1
Displaying Example Image
```



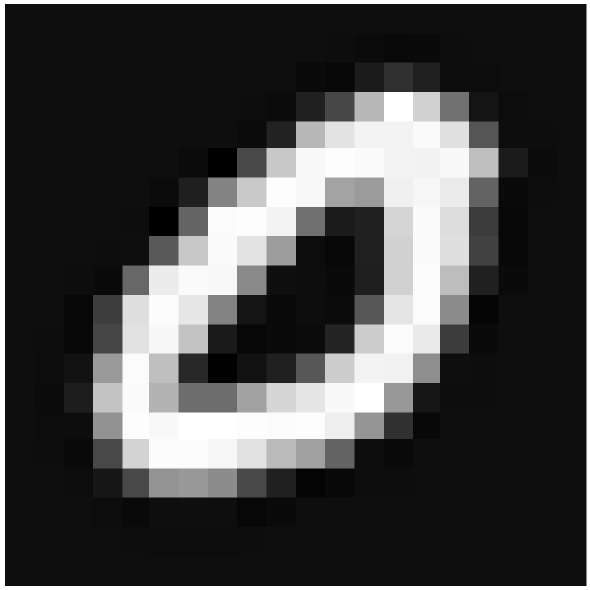
```
Neural Network Prediction:  [9.] (digit  [9.] )
2
Displaying Example Image
```



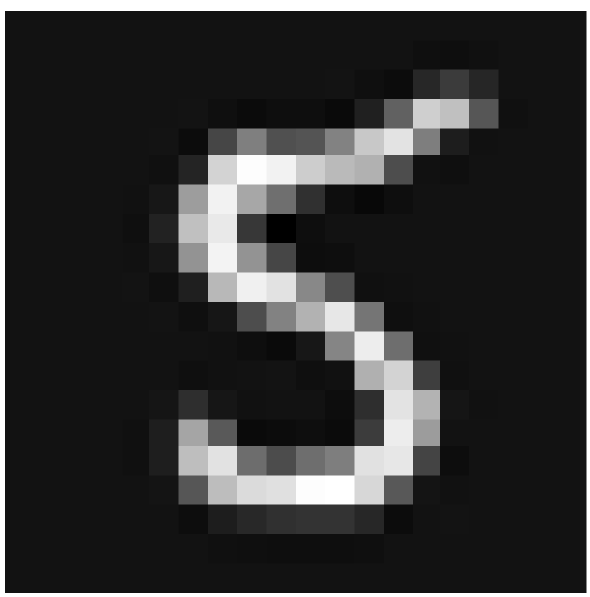
```
Neural Network Prediction:  [7.] (digit  [7.] )
3
Displaying Example Image
```



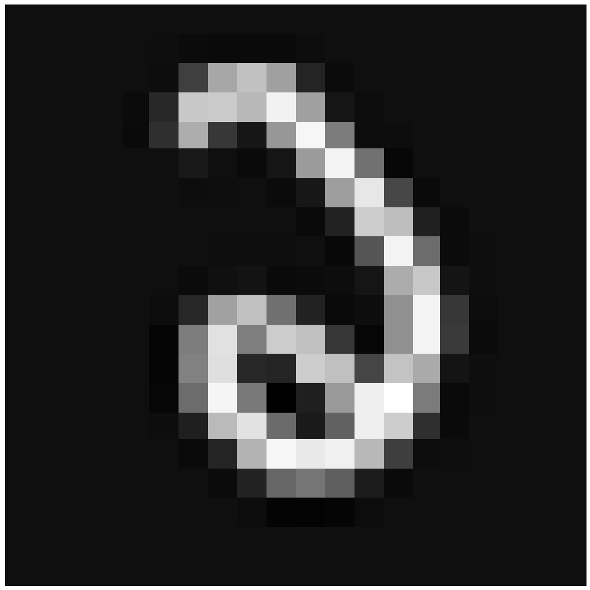
```
Neural Network Prediction:  [10.] (digit  [0.] )
4
Displaying Example Image
```



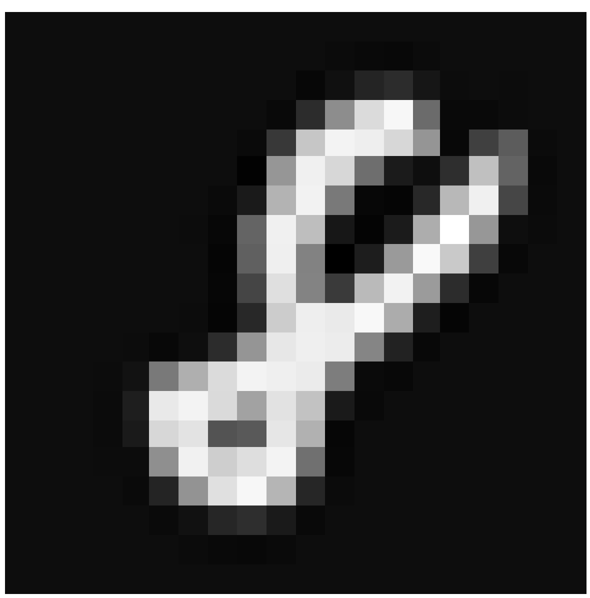
```
Neural Network Prediction:  [10.] (digit  [0.] )
5
Displaying Example Image
```



```
Neural Network Prediction:  [5.] (digit  [5.] )
6
Displaying Example Image
```

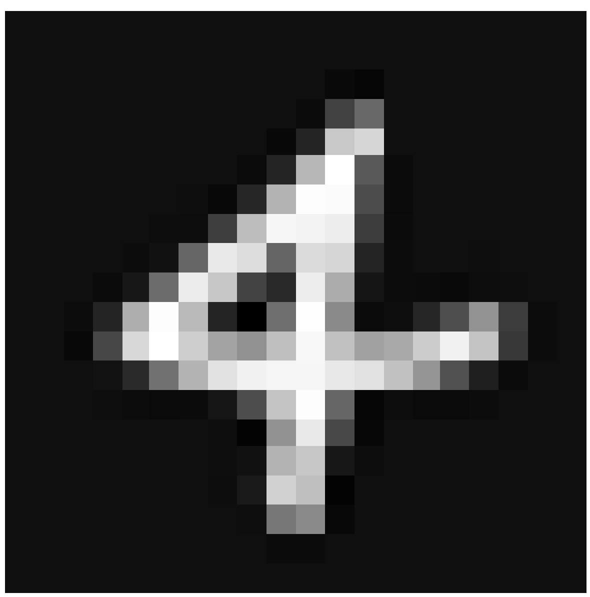


```
Neural Network Prediction:  [2.] (digit  [2.] )
7
Displaying Example Image
```

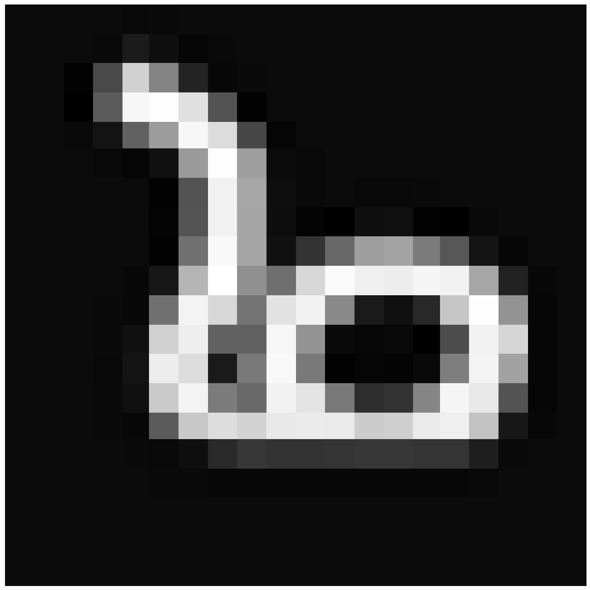


```
Neural Network Prediction:  [8.] (digit  [8.] )
8
Displaying Example Image
```

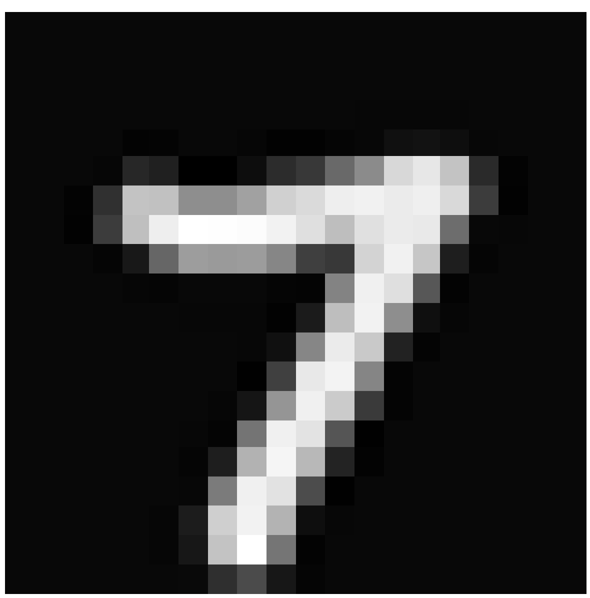
```
Neural Network Prediction:  [4.] (digit  [4.] )
9
Displaying Example Image
```



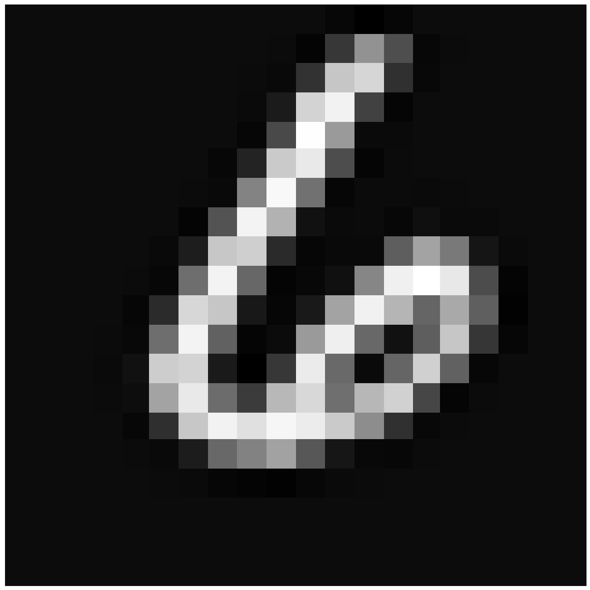
```
Neural Network Prediction:  [4.] (digit  [4.] )
10
Displaying Example Image
```



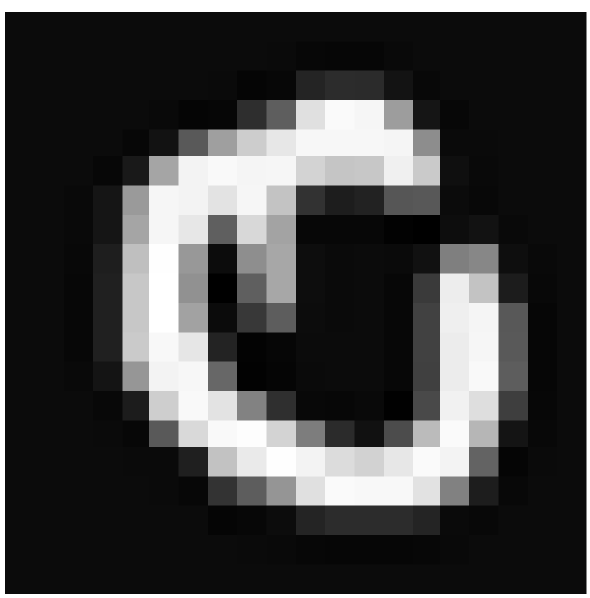
```
Neural Network Prediction:  [6.] (digit  [6.] )
11
Displaying Example Image
```



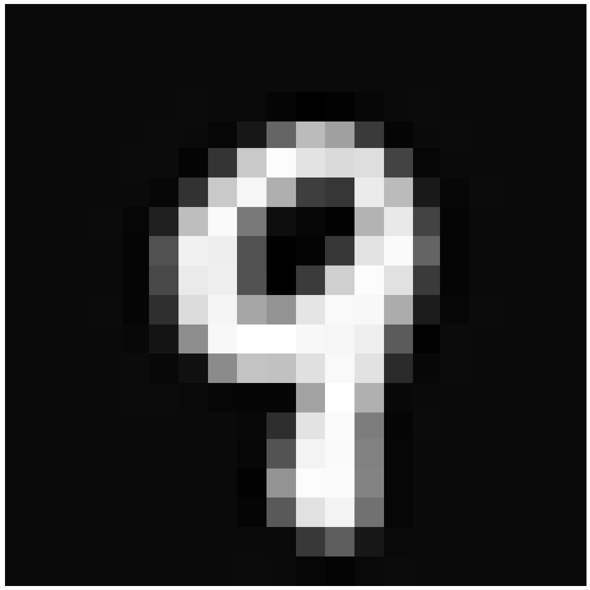
```
Neural Network Prediction:  [7.] (digit  [7.] )
12
Displaying Example Image
```



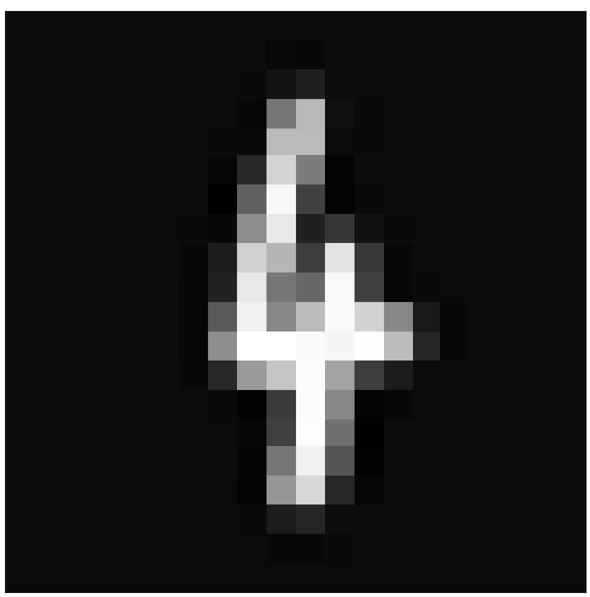
```
Neural Network Prediction:  [6.] (digit  [6.] )
13
Displaying Example Image
```



```
Neural Network Prediction:  [10.] (digit  [0.] )
14
Displaying Example Image
```

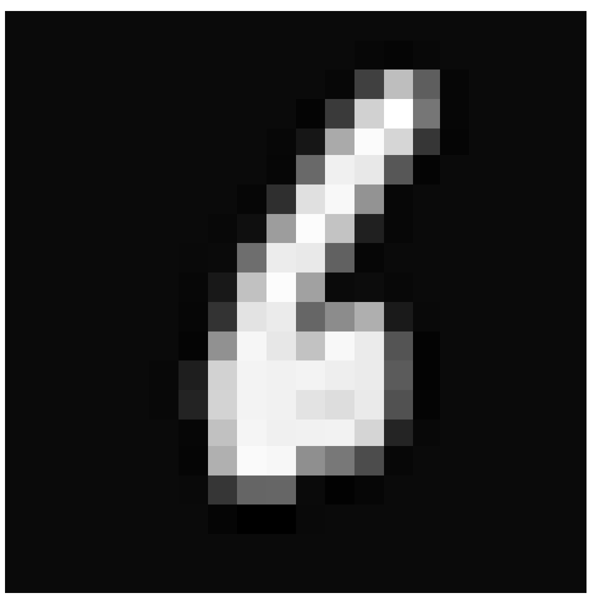


```
Neural Network Prediction:  [9.] (digit  [9.] )
15
Displaying Example Image
```

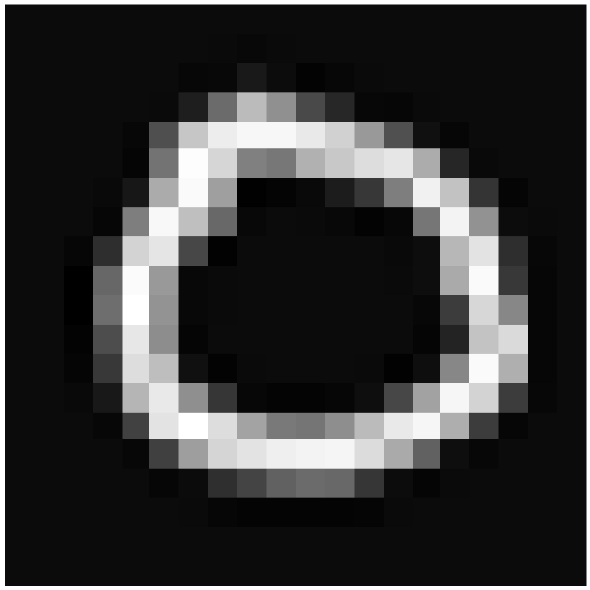


```
Neural Network Prediction:  [4.] (digit  [4.] )
16
Displaying Example Image
```

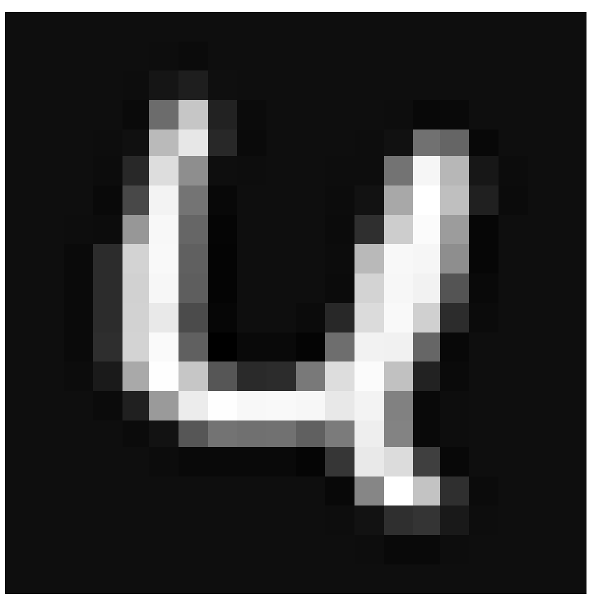
```
Neural Network Prediction:  [5.] (digit  [5.] )
17
Displaying Example Image
```



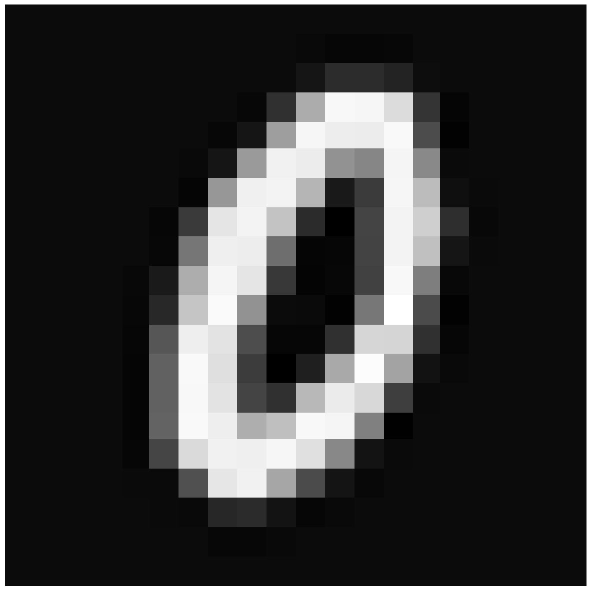
```
Neural Network Prediction:  [6.] (digit  [6.] )
18
Displaying Example Image
```



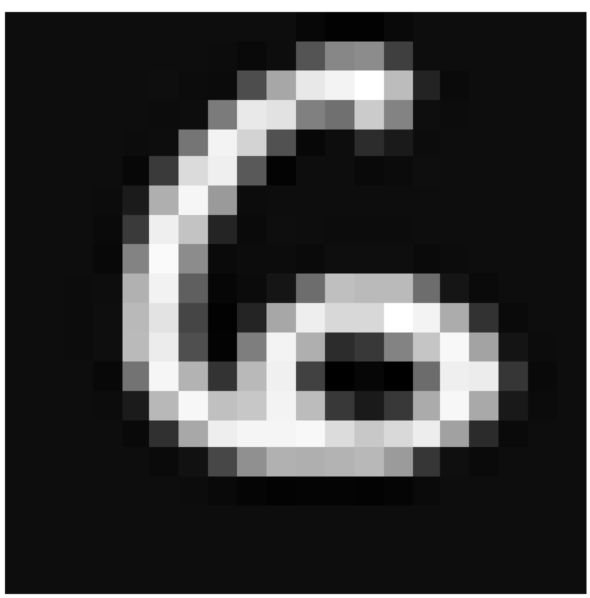
```
Neural Network Prediction:  [10.] (digit  [0.] )
19
Displaying Example Image
```



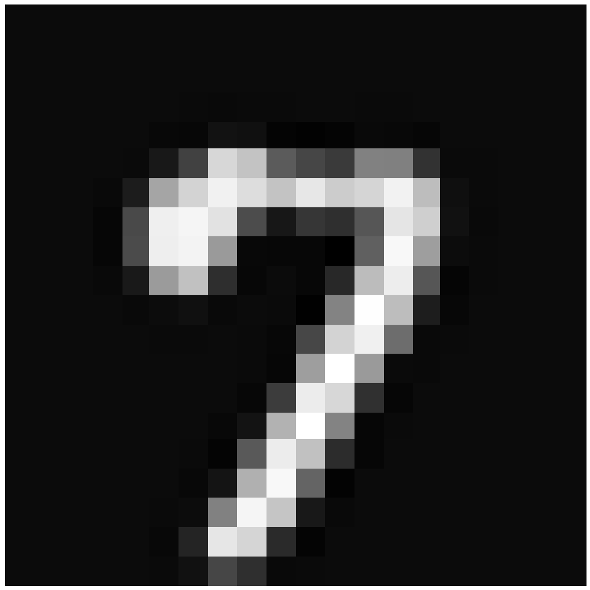
```
Neural Network Prediction:  [4.] (digit  [4.] )
20
Displaying Example Image
```



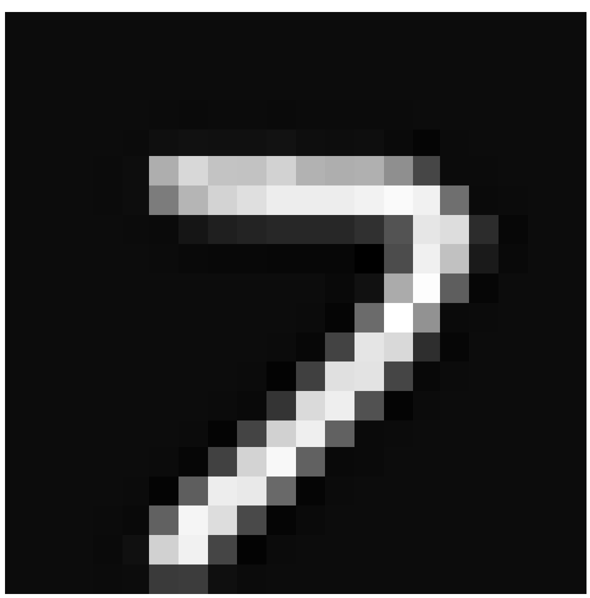
```
Neural Network Prediction:  [10.] (digit  [0.] )
21
Displaying Example Image
```



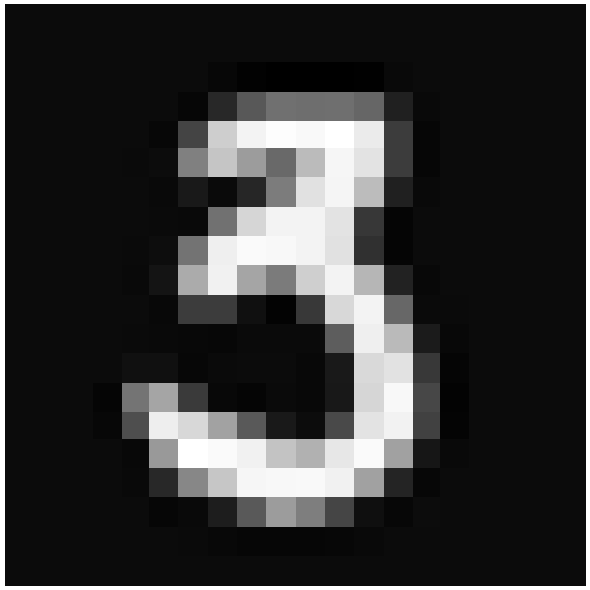
```
Neural Network Prediction:  [6.] (digit  [6.] )
22
Displaying Example Image
```



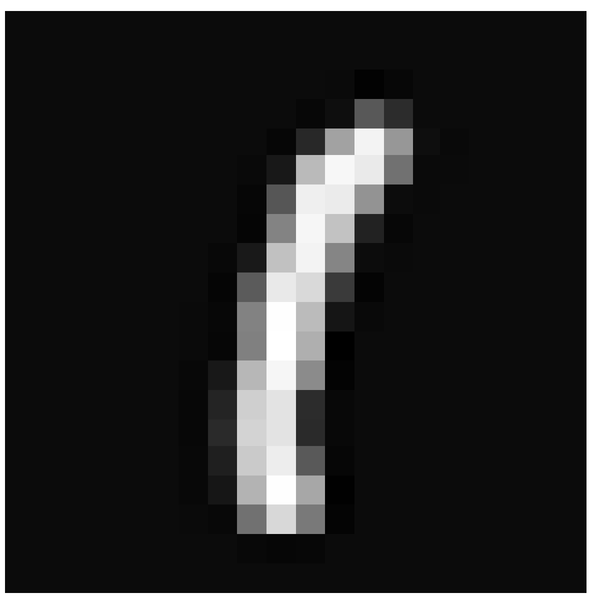
```
Neural Network Prediction:  [7.] (digit  [7.] )
23
Displaying Example Image
```



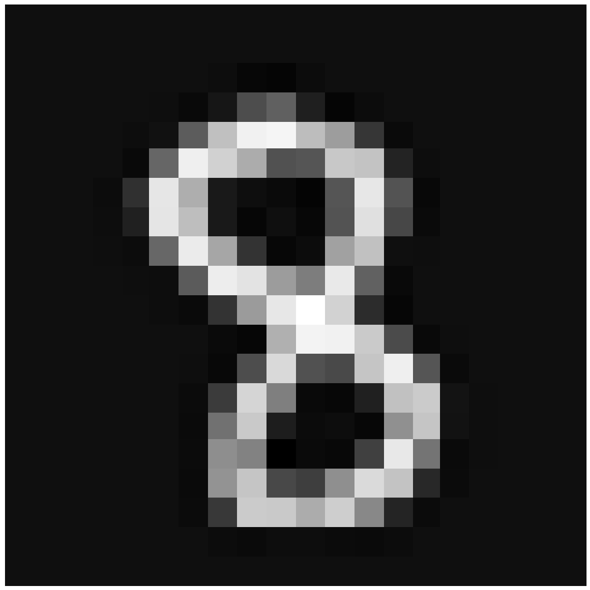
```
Neural Network Prediction:  [7.] (digit  [7.] )
24
Displaying Example Image
```



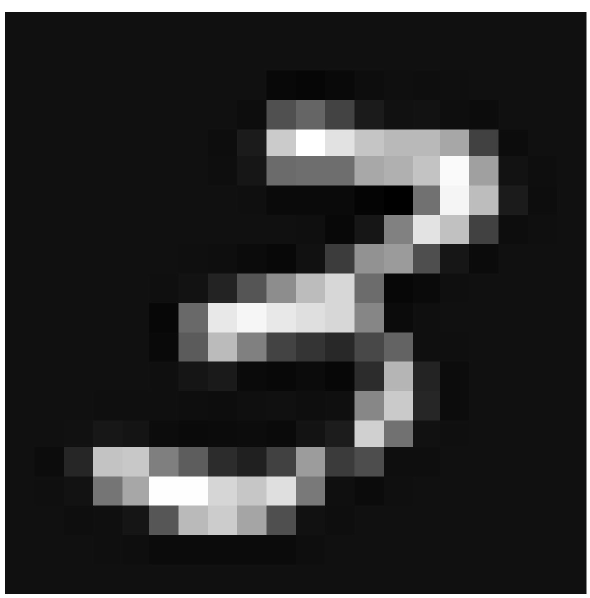
```
Neural Network Prediction:  [3.] (digit  [3.] )
25
Displaying Example Image
```



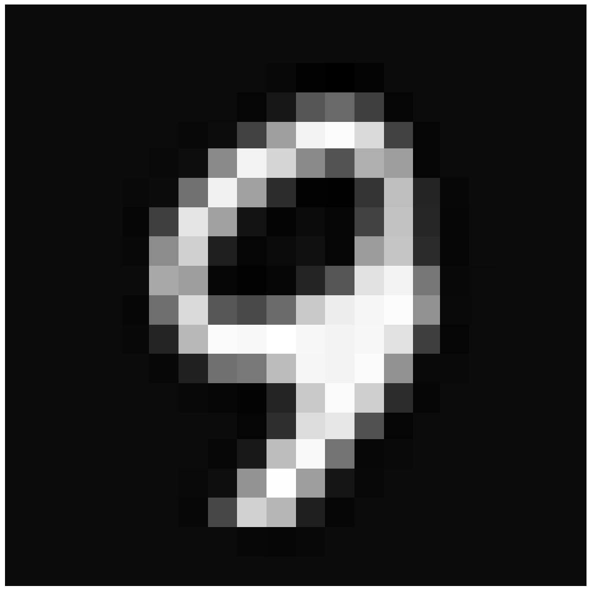
```
Neural Network Prediction:  [1.] (digit  [1.] )
26
Displaying Example Image
```



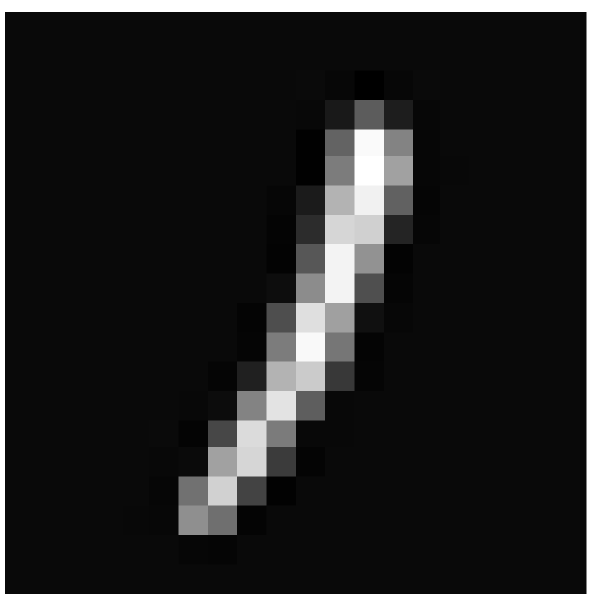
```
Neural Network Prediction:  [8.] (digit  [8.] )
27
Displaying Example Image
```



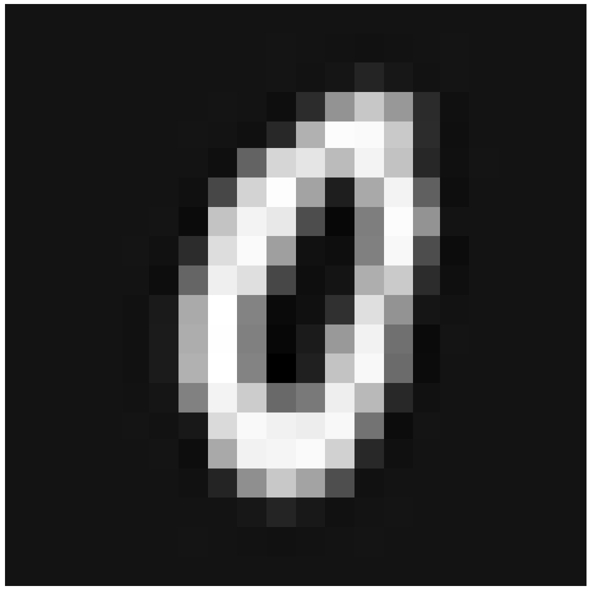
```
Neural Network Prediction:  [3.] (digit  [3.] )
28
Displaying Example Image
```



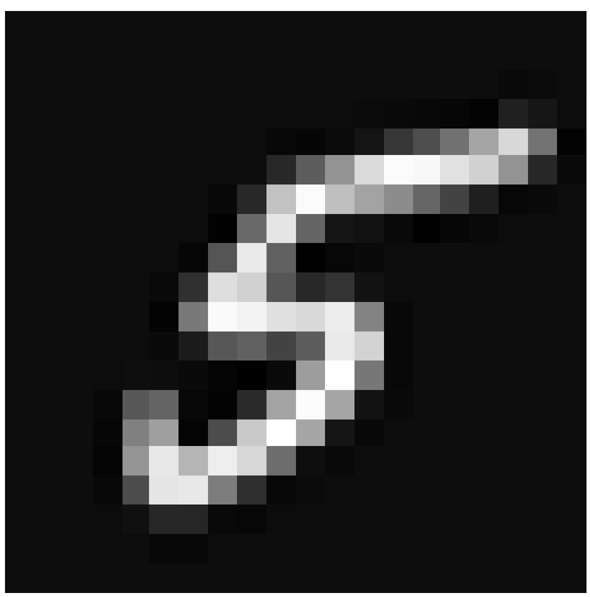
```
Neural Network Prediction:  [9.] (digit  [9.] )
29
Displaying Example Image
```



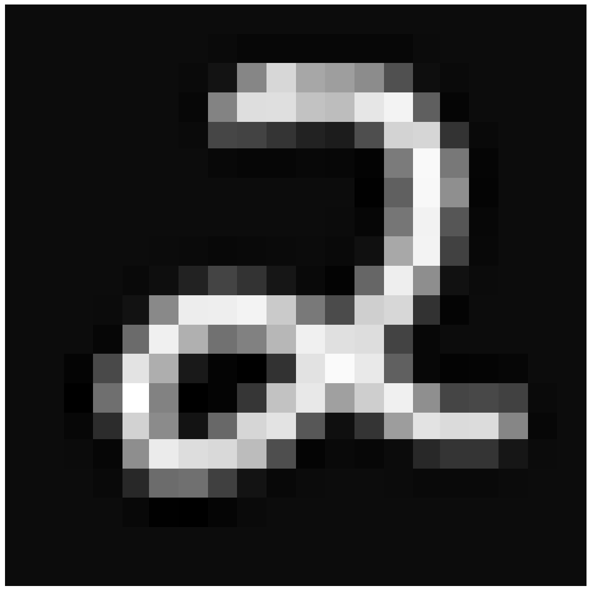
```
Neural Network Prediction:  [1.] (digit  [1.] )
30
Displaying Example Image
```



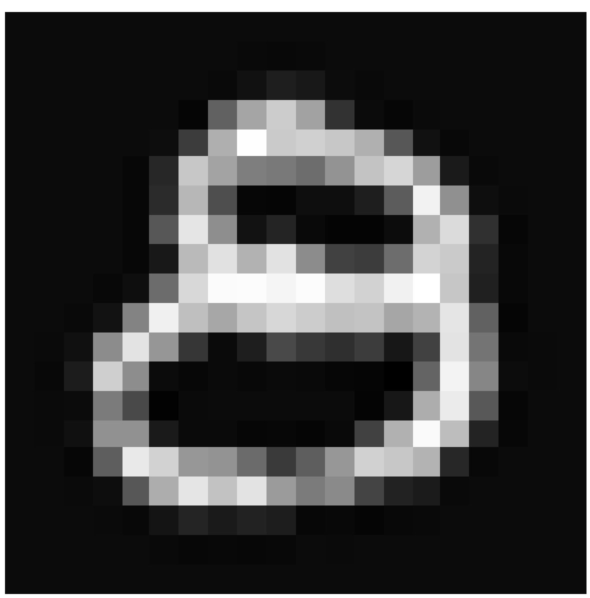
```
Neural Network Prediction:  [10.] (digit  [0.] )
31
Displaying Example Image
```



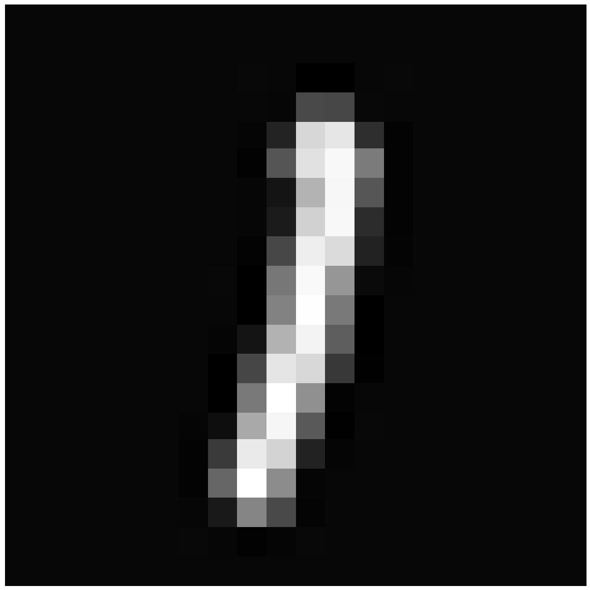
```
Neural Network Prediction:  [5.] (digit  [5.] )
32
Displaying Example Image
```



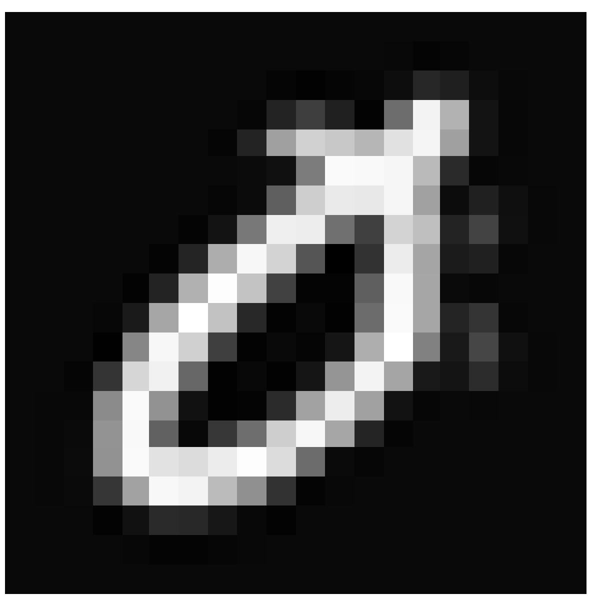
```
Neural Network Prediction:  [2.] (digit  [2.] )
33
Displaying Example Image
```



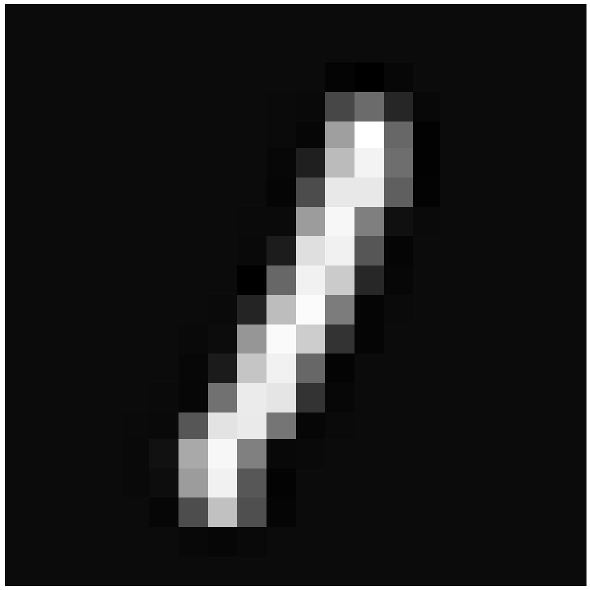
```
Neural Network Prediction:  [8.] (digit  [8.] )
34
Displaying Example Image
```



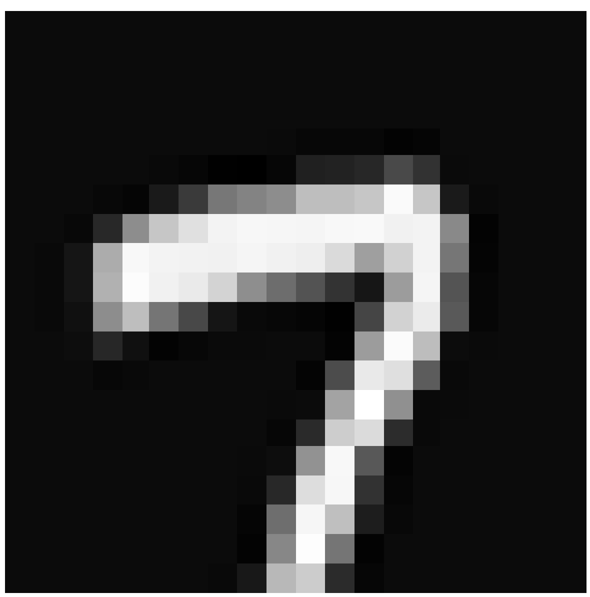
```
Neural Network Prediction:  [1.] (digit  [1.] )
35
Displaying Example Image
```



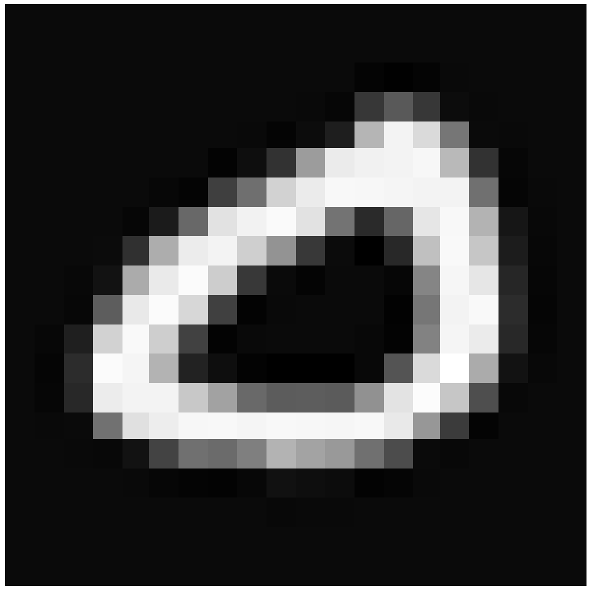
```
Neural Network Prediction:  [10.] (digit  [0.] )
36
Displaying Example Image
```



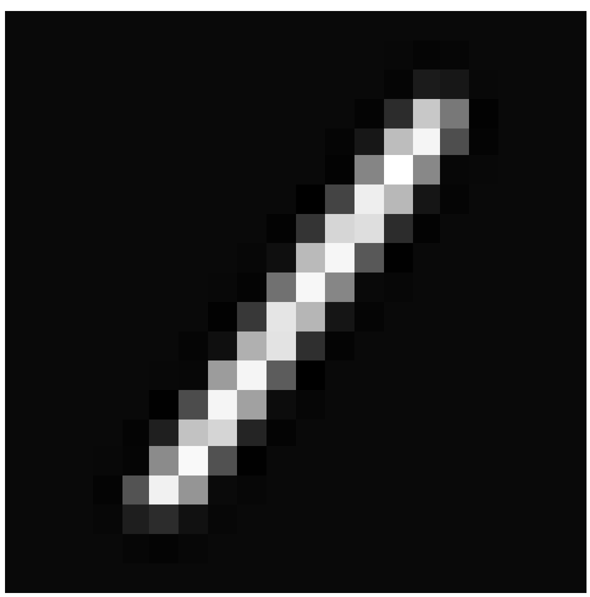
```
Neural Network Prediction:  [1.] (digit  [1.] )
37
Displaying Example Image
```



```
Neural Network Prediction:  [7.] (digit  [7.] )
38
Displaying Example Image
```



```
Neural Network Prediction:  [10.] (digit  [0.] )
39
Displaying Example Image
```



```
Neural Network Prediction:  [1.] (digit  [1.] )
40
Displaying Example Image
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
/var/folders/kj/r4bf7thj1gg5bp153_b6nf740000gn/T/ipykernel_29638/3251442423.py␣
↪in <module>
      7     print('Displaying Example Image')
      8     tmp = np.transpose(np.expand_dims(X[rp[i], :], axis=1))
----> 9     displayData(tmp)
     10
     11     pred = predict(Theta1, Theta2, tmp)

~/Desktop/Studium/Period8/MachineLearning/Lab5/Jupyter/Initial code/displayData
↪py in displayData(data)
      9         ax.imshow(np.transpose(np.reshape(data[i,:], (-1, 20))))
     10         ax.set_axis_off()
---> 11     plt.show()

~/miniforge3/lib/python3.9/site-packages/matplotlib/pyplot.py in show(*args,␣
↪**kwargs)
    376     """
    377     _warn_if_gui_out_of_main_thread()
--> 378     return _backend_mod.show(*args, **kwargs)
    379
    380

~/miniforge3/lib/python3.9/site-packages/matplotlib_inline/backend_inline.py in␣
↪show(close, block)
     41             display(
     42                 figure_manager.canvas.figure,
---> 43                 metadata=_fetch_figure_metadata(figure_manager.canvas.
↪figure)
     44             )
     45     finally:

~/miniforge3/lib/python3.9/site-packages/matplotlib_inline/backend_inline.py in␣
↪_fetch_figure_metadata(fig)
    229     if _is_transparent(fig.get_facecolor()):
    230         # the background is transparent
--> 231         ticksLight = _is_light([label.get_color()

    232                                 for axes in fig.axes
    233                                 for axis in (axes.xaxis, axes.yaxis)
```

```
~/miniforge3/lib/python3.9/site-packages/matplotlib_inline/backend_inline.py in␣
↪<listcomp>(.0)
    232                                  for axes in fig.axes
    233                                  for axis in (axes.xaxis, axes.yaxis)
--> 234                                  for label in axis.get_ticklabels()])
    235         if ticksLight.size and (ticksLight == ticksLight[0]).all():
    236             # there are one or more tick labels, all with the same␣
↪lightness

~/miniforge3/lib/python3.9/site-packages/matplotlib/axis.py in␣
↪get_ticklabels(self, minor, which)
   1230         if minor:
   1231             return self.get_minorticklabels()
-> 1232         return self.get_majorticklabels()
   1233 
   1234     def get_majorticklines(self):

~/miniforge3/lib/python3.9/site-packages/matplotlib/axis.py in␣
↪get_majorticklabels(self)
   1182     def get_majorticklabels(self):
   1183         """Return this Axis' major tick labels, as a list of `~.text.
↪Text`."""
-> 1184         ticks = self.get_major_ticks()
   1185         labels1 = [tick.label1 for tick in ticks if tick.label1.
↪get_visible()]
   1186         labels2 = [tick.label2 for tick in ticks if tick.label2.
↪get_visible()]

~/miniforge3/lib/python3.9/site-packages/matplotlib/axis.py in␣
↪get_major_ticks(self, numticks)
   1357         while len(self.majorTicks) < numticks:
   1358             # Update the new tick label properties from the old.
-> 1359             tick = self._get_tick(major=True)
   1360             self.majorTicks.append(tick)
   1361             self._copy_tick_props(self.majorTicks[0], tick)

~/miniforge3/lib/python3.9/site-packages/matplotlib/axis.py in _get_tick(self,␣
↪major)
   2057         else:
   2058             tick_kw = self._minor_tick_kw
-> 2059         return XTick(self.axes, 0, major=major, **tick_kw)
   2060 
   2061     def set_label_position(self, position):

~/miniforge3/lib/python3.9/site-packages/matplotlib/axis.py in __init__(self,␣
↪*args, **kwargs)
```

```
    418
    419     def __init__(self, *args, **kwargs):
--> 420         super().__init__(*args, **kwargs)
    421         # x in data coords, y in axes coords
    422         self.tick1line.set(

~/miniforge3/lib/python3.9/site-packages/matplotlib/_api/deprecation.py in␣
 ↪wrapper(*inner_args, **inner_kwargs)
    429                      else deprecation_addendum,
    430                 **kwargs)
--> 431         return func(*inner_args, **inner_kwargs)
    432
    433     return wrapper

~/miniforge3/lib/python3.9/site-packages/matplotlib/axis.py in __init__(self,␣
 ↪axes, loc, label, size, width, color, tickdir, pad, labelsize, labelcolor,␣
 ↪zorder, gridOn, tick1On, tick2On, label1On, label2On, major, labelrotation,␣
 ↪grid_color, grid_linestyle, grid_linewidth, grid_alpha, **kw)
    151         self.apply_tickdir(tickdir)
    152
--> 153         self.tick1line = mlines.Line2D(

    154             [], [],
    155             color=color, linestyle="none", zorder=zorder,␣
 ↪visible=tick1On,

~/miniforge3/lib/python3.9/site-packages/matplotlib/lines.py in __init__(self,␣
 ↪xdata, ydata, linewidth, linestyle, color, marker, markersize,␣
 ↪markeredgewidth, markeredgecolor, markerfacecolor, markerfacecoloralt,␣
 ↪fillstyle, antialiased, dash_capstyle, solid_capstyle, dash_joinstyle,␣
 ↪solid_joinstyle, pickradius, drawstyle, markevery, **kwargs)
    352         self.set_dash_capstyle(dash_capstyle)
    353         self.set_dash_joinstyle(dash_joinstyle)
--> 354         self.set_solid_capstyle(solid_capstyle)
    355         self.set_solid_joinstyle(solid_joinstyle)
    356

KeyboardInterrupt:
```

5 ================== Part 4: Sigmoid Gradient ==================

Before you start implementing backpropagation, you will first implement the gradient for the sigmoid function. You should complete the code in the sigmoidGradient.m file.

[7]:
```
print('Evaluating sigmoid gradient...')
example = np.array([-15, -1, -0.5, 0, 0.5, 1, 15])
g = sigmoidGradient(example)
```

```
print('Sigmoid gradient evaluated at', example, ':')
print(g)
```

```
Evaluating sigmoid gradient…
Sigmoid gradient evaluated at [-15.   -1.   -0.5   0.    0.5   1.   15. ] :
[3.05902133e-07 1.96611933e-01 2.35003712e-01 2.50000000e-01
 2.35003712e-01 1.96611933e-01 3.05902133e-07]
```

6 ================== Part 5: Initializing Pameters ==================

To learn a two layer neural network that classifies digits. You will start by implementing a function to initialize the weights of the neural network (randInitializeWeights.py)

[8]:
```
print('Initializing Neural Network Parameters ...')

initial_Theta1 = randInitializeWeights(input_layer_size, hidden_layer_size)
initial_Theta2 = randInitializeWeights(hidden_layer_size, num_labels)

# Unroll parameters
initial_Theta1 = np.reshape(initial_Theta1, initial_Theta1.size, order='F')
initial_Theta2 = np.reshape(initial_Theta2, initial_Theta2.size, order='F')
initial_nn_params = np.hstack((initial_Theta1, initial_Theta2))
print(initial_nn_params)
```

```
Initializing Neural Network Parameters …
[-0.05694027 -0.07043208  0.03860064 … -0.28053134 -0.21011184
  0.40030573]
```

7 ================= Part 6: Implement Backpropagation =================

Now you will implement the backpropagation algorithm for the neural network. You should add code to nnCostFunction.m to return the partial derivatives of the parameters.

[9]:
```
print('Checking Backpropagation...')

#  Check gradients by running checkNNGradients
checkNNGradients()
```

```
Checking Backpropagation…
[[-9.27825235e-03]
 [ 8.89911959e-03]
 [-8.36010761e-03]
 [ 7.62813551e-03]
 [-6.74798369e-03]
 [-3.04978931e-06]
```

```
[ 1.42869450e-05]
[-2.59383093e-05]
[ 3.69883213e-05]
[-4.68759787e-05]
[-1.75060084e-04]
[ 2.33146356e-04]
[-2.87468729e-04]
[ 3.35320347e-04]
[-3.76215588e-04]
[-9.62660640e-05]
[ 1.17982668e-04]
[-1.37149705e-04]
[ 1.53247079e-04]
[-1.66560297e-04]
[ 3.14544970e-01]
[ 1.11056588e-01]
[ 9.74006970e-02]
[ 1.64090819e-01]
[ 5.75736494e-02]
[ 5.04575855e-02]
[ 1.64567932e-01]
[ 5.77867378e-02]
[ 5.07530173e-02]
[ 1.58339334e-01]
[ 5.59235296e-02]
[ 4.91620841e-02]
[ 1.51127527e-01]
[ 5.36967009e-02]
[ 4.71456249e-02]
[ 1.49568335e-01]
[ 5.31542052e-02]
[ 4.65597186e-02]] [[-9.27825236e-03]
[ 8.89911960e-03]
[-8.36010762e-03]
[ 7.62813551e-03]
[-6.74798370e-03]
[-3.04978914e-06]
[ 1.42869443e-05]
[-2.59383100e-05]
[ 3.69883234e-05]
[-4.68759769e-05]
[-1.75060082e-04]
[ 2.33146357e-04]
[-2.87468729e-04]
[ 3.35320347e-04]
[-3.76215587e-04]
[-9.62660620e-05]
[ 1.17982666e-04]
```

```
 [-1.37149706e-04]
 [ 1.53247082e-04]
 [-1.66560294e-04]
 [ 3.14544970e-01]
 [ 1.11056588e-01]
 [ 9.74006970e-02]
 [ 1.64090819e-01]
 [ 5.75736493e-02]
 [ 5.04575855e-02]
 [ 1.64567932e-01]
 [ 5.77867378e-02]
 [ 5.07530173e-02]
 [ 1.58339334e-01]
 [ 5.59235296e-02]
 [ 4.91620841e-02]
 [ 1.51127527e-01]
 [ 5.36967009e-02]
 [ 4.71456249e-02]
 [ 1.49568335e-01]
 [ 5.31542052e-02]
 [ 4.65597186e-02]]
The above two columns you get should be very similar.
 (Left-Numerical Gradient, Right-(Your) Analytical Gradient)


If your backpropagation implementation is correct, then
 the relative difference will be small (less than 1e-9).

Relative Difference:  2.2957543313343497e-11
```

8 ================ Part 7: Implement Regularization ================

Once your backpropagation implementation is correct, you should now continue to implement the regularization gradient.

[10]:
```python
print('Checking Backpropagation (w/ Regularization) ... ')

##  Check gradients by running checkNNGradients
lambda_value = 3
checkNNGradients(lambda_value)

# Also output the costFunction debugging values
debug_J  = nnCostFunction(nn_params, input_layer_size, hidden_layer_size,
                          num_labels, X, y, lambda_value)

print('Cost at (fixed) debugging parameters (w/ lambda = 10): ',  debug_J[0][0],
```

```
      '(this value should be about 0.576051)')
```

```
Checking Backpropagation (w/ Regularization) …
[[-9.27825235e-03]
 [ 8.89911959e-03]
 [-8.36010761e-03]
 [ 7.62813551e-03]
 [-6.74798369e-03]
 [-1.67679797e-02]
 [ 3.94334829e-02]
 [ 5.93355565e-02]
 [ 2.47640974e-02]
 [-3.26881426e-02]
 [-6.01744725e-02]
 [-3.19612287e-02]
 [ 2.49225535e-02]
 [ 5.97717617e-02]
 [ 3.86410548e-02]
 [-1.73704651e-02]
 [-5.75658668e-02]
 [-4.51963845e-02]
 [ 9.14587966e-03]
 [ 5.46101547e-02]
 [ 3.14544970e-01]
 [ 1.11056588e-01]
 [ 9.74006970e-02]
 [ 1.18682669e-01]
 [ 3.81928689e-05]
 [ 3.36926556e-02]
 [ 2.03987128e-01]
 [ 1.17148233e-01]
 [ 7.54801264e-02]
 [ 1.25698067e-01]
 [-4.07588279e-03]
 [ 1.69677090e-02]
 [ 1.76337550e-01]
 [ 1.13133142e-01]
 [ 8.61628953e-02]
 [ 1.32294136e-01]
 [-4.52964427e-03]
 [ 1.50048382e-03]] [[-9.27825236e-03]
 [ 8.89911960e-03]
 [-8.36010762e-03]
 [ 7.62813551e-03]
 [-6.74798370e-03]
 [-1.67679797e-02]
 [ 3.94334829e-02]
 [ 5.93355565e-02]
```

```
 [ 2.47640974e-02]
 [-3.26881426e-02]
 [-6.01744725e-02]
 [-3.19612287e-02]
 [ 2.49225535e-02]
 [ 5.97717617e-02]
 [ 3.86410548e-02]
 [-1.73704651e-02]
 [-5.75658668e-02]
 [-4.51963845e-02]
 [ 9.14587966e-03]
 [ 5.46101547e-02]
 [ 3.14544970e-01]
 [ 1.11056588e-01]
 [ 9.74006970e-02]
 [ 1.18682669e-01]
 [ 3.81928696e-05]
 [ 3.36926556e-02]
 [ 2.03987128e-01]
 [ 1.17148233e-01]
 [ 7.54801264e-02]
 [ 1.25698067e-01]
 [-4.07588279e-03]
 [ 1.69677090e-02]
 [ 1.76337550e-01]
 [ 1.13133142e-01]
 [ 8.61628953e-02]
 [ 1.32294136e-01]
 [-4.52964427e-03]
 [ 1.50048382e-03]]
The above two columns you get should be very similar.
 (Left-Numerical Gradient, Right-(Your) Analytical Gradient)


If your backpropagation implementation is correct, then
 the relative difference will be small (less than 1e-9).

Relative Difference:  2.2006042941330916e-11
Cost at (fixed) debugging parameters (w/ lambda = 10):  0.5760512469501331 (this
value should be about 0.576051)
```

9 ==================== Part 8: Training NN ====================

You have now implemented all the code necessary to train a neural network. To train your neural network, we will now use "fmincg", which is a function which works similarly to "fminunc". Recall that these advanced optimizers are able to train our cost functions efficiently as long as we provide them with the gradient computations.

[11]:
```python
print('Training Neural Network...')

#  After you have completed the assignment, change the MaxIter to a larger
#  value to see how more training helps.
MaxIter = 150

#  You should also try different values of lambda
lambda_value = 1

# Create "short hand" for the cost function to be minimized
y = np.expand_dims(y, axis=1)

costFunction = lambda p : nnCostFunction(p, input_layer_size, hidden_layer_size,
                                         num_labels, X, y, lambda_value)

# Now, costFunction is a function that takes in only one argument (the
# neural network parameters)
[nn_params, cost] = fmincg(costFunction, initial_nn_params, MaxIter)

# Obtain Theta1 and Theta2 back from nn_params
Theta1 = np.reshape(nn_params[0:hidden_layer_size * (input_layer_size + 1)],
                              (hidden_layer_size, (input_layer_size + 1)),␣
 ↪order='F')
Theta2 = np.reshape(nn_params[((hidden_layer_size * (input_layer_size + 1))):],
                              (num_labels, (hidden_layer_size + 1)), order='F')
```

```
Training Neural Network…
Iteration  1 | Cost:  [3.49823969]
Iteration  2 | Cost:  [3.24347381]
Iteration  3 | Cost:  [3.12158557]
Iteration  4 | Cost:  [2.97383784]
Iteration  5 | Cost:  [2.78142747]
Iteration  6 | Cost:  [2.56996023]
Iteration  7 | Cost:  [2.49661158]
Iteration  8 | Cost:  [1.91191044]
Iteration  9 | Cost:  [1.62883088]
Iteration  10 | Cost:  [1.41085677]
Iteration  11 | Cost:  [1.25879926]
Iteration  12 | Cost:  [1.21614708]
Iteration  13 | Cost:  [1.09110953]
Iteration  14 | Cost:  [1.04490237]
Iteration  15 | Cost:  [0.98906894]
Iteration  16 | Cost:  [0.92819063]
Iteration  17 | Cost:  [0.89113199]
Iteration  18 | Cost:  [0.87241911]
Iteration  19 | Cost:  [0.8266248]
```

```
Iteration  20 | Cost:  [0.79457046]
Iteration  21 | Cost:  [0.77800163]
Iteration  22 | Cost:  [0.72515449]
Iteration  23 | Cost:  [0.71050929]
Iteration  24 | Cost:  [0.69371792]
Iteration  25 | Cost:  [0.67281091]
Iteration  26 | Cost:  [0.65998649]
Iteration  27 | Cost:  [0.65369317]
Iteration  28 | Cost:  [0.63977478]
Iteration  29 | Cost:  [0.62722291]
Iteration  30 | Cost:  [0.61495667]
Iteration  31 | Cost:  [0.59925118]
Iteration  32 | Cost:  [0.57943578]
Iteration  33 | Cost:  [0.57681014]
Iteration  34 | Cost:  [0.57402233]
Iteration  35 | Cost:  [0.56448431]
Iteration  36 | Cost:  [0.55418161]
Iteration  37 | Cost:  [0.54394287]
Iteration  38 | Cost:  [0.53291131]
Iteration  39 | Cost:  [0.5126614]
Iteration  40 | Cost:  [0.49888647]
Iteration  41 | Cost:  [0.48317515]
Iteration  42 | Cost:  [0.47608402]
Iteration  43 | Cost:  [0.47234736]
Iteration  44 | Cost:  [0.46786548]
Iteration  45 | Cost:  [0.46407074]
Iteration  46 | Cost:  [0.46093942]
Iteration  47 | Cost:  [0.45626854]
Iteration  48 | Cost:  [0.45153547]
Iteration  49 | Cost:  [0.44902641]
Iteration  50 | Cost:  [0.4476314]
Iteration  51 | Cost:  [0.44678635]
Iteration  52 | Cost:  [0.44415176]
Iteration  53 | Cost:  [0.44240665]
Iteration  54 | Cost:  [0.44192092]
Iteration  55 | Cost:  [0.44055679]
Iteration  56 | Cost:  [0.43923347]
Iteration  57 | Cost:  [0.43750361]
Iteration  58 | Cost:  [0.43572285]
Iteration  59 | Cost:  [0.43027741]
Iteration  60 | Cost:  [0.42486924]
Iteration  61 | Cost:  [0.41320797]
Iteration  62 | Cost:  [0.40430599]
Iteration  63 | Cost:  [0.39447605]
Iteration  64 | Cost:  [0.39145467]
Iteration  65 | Cost:  [0.38963542]
Iteration  66 | Cost:  [0.38766581]
Iteration  67 | Cost:  [0.38531631]
```

```
Iteration  68 | Cost:  [0.38291395]
Iteration  69 | Cost:  [0.38147827]
Iteration  70 | Cost:  [0.38084556]
Iteration  71 | Cost:  [0.37911034]
Iteration  72 | Cost:  [0.37774182]
Iteration  73 | Cost:  [0.37705451]
Iteration  74 | Cost:  [0.37618309]
Iteration  75 | Cost:  [0.37581191]
Iteration  76 | Cost:  [0.37570479]
Iteration  77 | Cost:  [0.37526859]
Iteration  78 | Cost:  [0.37488594]
Iteration  79 | Cost:  [0.3747085]
Iteration  80 | Cost:  [0.3745231]
Iteration  81 | Cost:  [0.37308652]
Iteration  82 | Cost:  [0.37176486]
Iteration  83 | Cost:  [0.36994506]
Iteration  84 | Cost:  [0.36839749]
Iteration  85 | Cost:  [0.36761091]
Iteration  86 | Cost:  [0.36593911]
Iteration  87 | Cost:  [0.36428952]
Iteration  88 | Cost:  [0.36209676]
Iteration  89 | Cost:  [0.36131615]
Iteration  90 | Cost:  [0.36098567]
Iteration  91 | Cost:  [0.36058148]
Iteration  92 | Cost:  [0.36014886]
Iteration  93 | Cost:  [0.35967547]
Iteration  94 | Cost:  [0.35931384]
Iteration  95 | Cost:  [0.35882235]
Iteration  96 | Cost:  [0.35813908]
Iteration  97 | Cost:  [0.35765274]
Iteration  98 | Cost:  [0.35687305]
Iteration  99 | Cost:  [0.35589518]
Iteration  100 | Cost:  [0.35570832]
Iteration  101 | Cost:  [0.35560143]
Iteration  102 | Cost:  [0.35548232]
Iteration  103 | Cost:  [0.35517304]
Iteration  104 | Cost:  [0.3543256]
Iteration  105 | Cost:  [0.35321992]
Iteration  106 | Cost:  [0.3528937]
Iteration  107 | Cost:  [0.35257076]
Iteration  108 | Cost:  [0.35223261]
Iteration  109 | Cost:  [0.35187511]
Iteration  110 | Cost:  [0.35168459]
Iteration  111 | Cost:  [0.35164941]
Iteration  112 | Cost:  [0.35150996]
Iteration  113 | Cost:  [0.35136733]
Iteration  114 | Cost:  [0.35129319]
Iteration  115 | Cost:  [0.35123084]
```

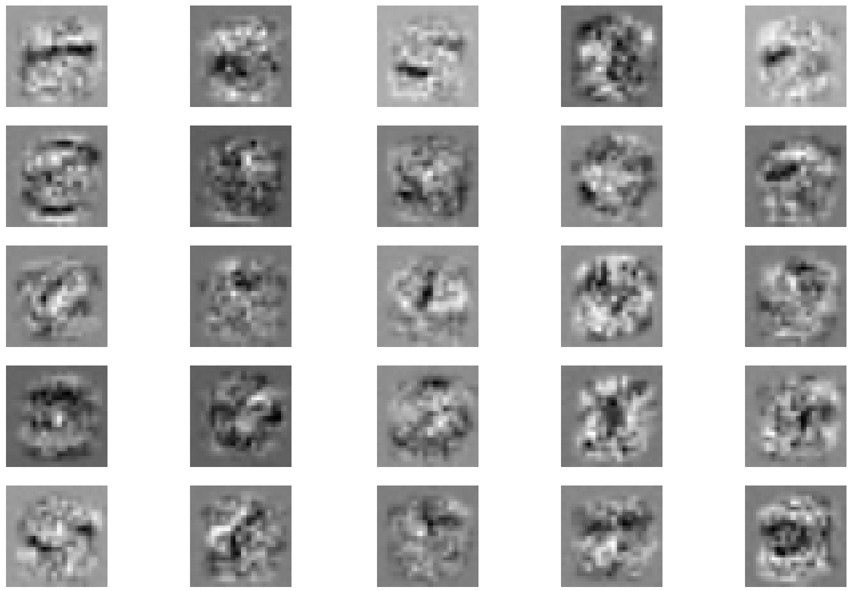
```
Iteration  116 | Cost:  [0.35086671]
Iteration  117 | Cost:  [0.35058053]
Iteration  118 | Cost:  [0.35048007]
Iteration  119 | Cost:  [0.35033374]
Iteration  120 | Cost:  [0.35020157]
Iteration  121 | Cost:  [0.35013218]
Iteration  122 | Cost:  [0.35008993]
Iteration  123 | Cost:  [0.34979006]
Iteration  124 | Cost:  [0.34905649]
Iteration  125 | Cost:  [0.34827103]
Iteration  126 | Cost:  [0.3474595]
Iteration  127 | Cost:  [0.34576245]
Iteration  128 | Cost:  [0.34467584]
Iteration  129 | Cost:  [0.34325712]
Iteration  130 | Cost:  [0.34214393]
Iteration  131 | Cost:  [0.34155858]
Iteration  132 | Cost:  [0.34135353]
Iteration  133 | Cost:  [0.34085519]
Iteration  134 | Cost:  [0.34074223]
Iteration  135 | Cost:  [0.34048641]
Iteration  136 | Cost:  [0.34021976]
Iteration  137 | Cost:  [0.34004822]
Iteration  138 | Cost:  [0.3398439]
Iteration  139 | Cost:  [0.33921049]
Iteration  140 | Cost:  [0.33887219]
Iteration  141 | Cost:  [0.33879885]
Iteration  142 | Cost:  [0.33847175]
Iteration  143 | Cost:  [0.33832326]
Iteration  144 | Cost:  [0.33828464]
Iteration  145 | Cost:  [0.33823076]
Iteration  146 | Cost:  [0.33819064]
Iteration  147 | Cost:  [0.33804039]
Iteration  148 | Cost:  [0.33794879]
Iteration  149 | Cost:  [0.33790889]
Iteration  150 | Cost:  [0.33782227]
```

10 ================== Part 9: Visualize Weights ==================

You can now “visualize” what the neural network is learning by displaying the hidden units to see what features they are capturing in the data.

[12]:
```
print('\nVisualizing Neural Network... \n')

displayData(Theta1[:, 1:])
```

```
Visualizing Neural Network…
```



11 ============== Part 10: Predicting with learned weights ========

After training the neural network, we would like to use it to predict the labels. The already implemented "predict" function is used by neural network to predict the labels of the training set. This letsyou compute the training set accuracy.

[13]:
```
pred = predict(Theta1, Theta2, X)
pred = np.expand_dims(pred,axis=1)
print('Training Set Accuracy: ', (pred == y).mean()*100)
```

```
Training Set Accuracy:  99.11999999999999
```

12 Report

12.1 Note: This report will not go into too much detail about the implementation, but I did my best to provide comments in the source code to explain what I am doing

12.1.1 The Neural Network:

The proposed and initialized neural network consists of 400 (+ 1 bias) input nodes, 25 (+1 bias) hidden nodes in one hidden layer and 10 output nodes (corresponding to the labels from 0 to 9 / 1 to 10). The network is fully connected on all layers and therefore has 10426 weights between the first and second and 260 weights between the second and third layer. This makes a total of 10686 parameters to adjust. The optimization algorithm in use is vanilla backpropagation, which is optimized using the fmincg optimizer. Furthermore, the weights are initialized using Xavier initialization, where we calculate an epsilon value as epsilon = sqrt(6) / sqrt(L_in + L_out) and then for each weight randomly sample from a uniform distribution over [-epsilon, +epsilon] and set the corresponding weight to that sampled value. This helps / tries to prevent exploding/vanishing gradients when training.

12.1.2 Impact of specific variables:

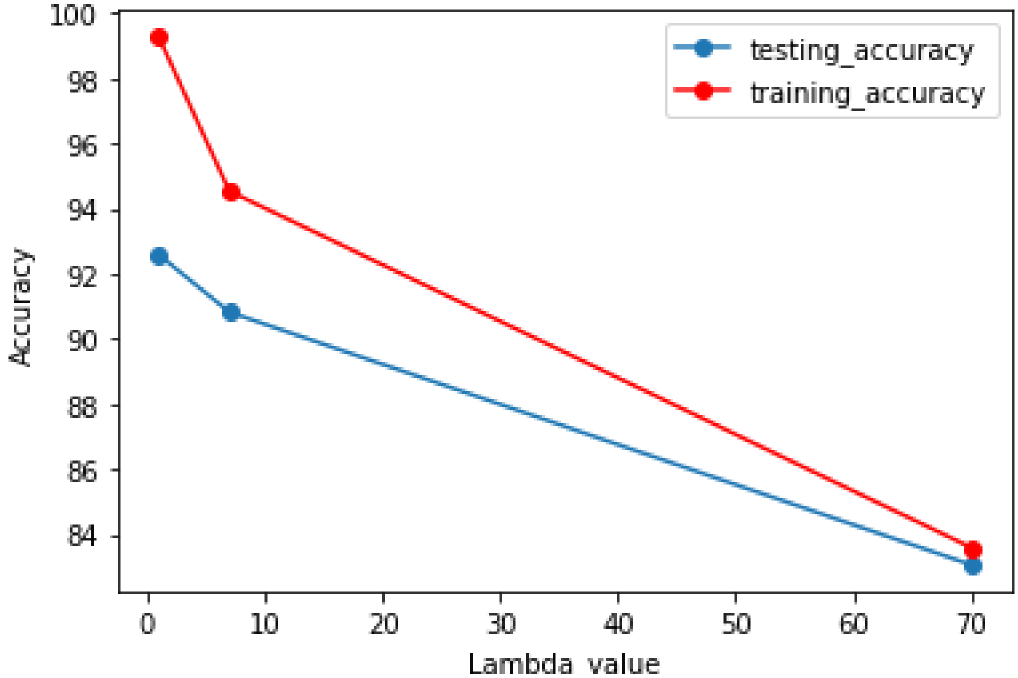
- Lambda

[59]:
```
%%capture
from sklearn.model_selection import train_test_split
print('Loading and data ...')
mat = scipy.io.loadmat('digitdata.mat')
X = mat['X']
y = mat['y']

X_train, X_test, Y_train, Y_test = train_test_split(X, y, test_size=0.34,␣
↪random_state=10)
print('Training Neural Network...')
#Lambda values:
lambda_values = [1, 7, 70]
#Accuracies
accuracies = {}
training_accuracy = {}
#Cost list
cost_list = []
#  After you have completed the assignment, change the MaxIter to a larger
#  value to see how more training helps.
MaxIter = 150
#Training
for lambda_value in lambda_values:
    print('Initializing Neural Network Parameters ...')

    initial_Theta1 = randInitializeWeights(input_layer_size, hidden_layer_size)
    initial_Theta2 = randInitializeWeights(hidden_layer_size, num_labels)

    # Unroll parameters
    initial_Theta1 = np.reshape(initial_Theta1, initial_Theta1.size, order='F')
    initial_Theta2 = np.reshape(initial_Theta2, initial_Theta2.size, order='F')
    initial_nn_params = np.hstack((initial_Theta1, initial_Theta2))

    # Create "short hand" for the cost function to be minimized
    costFunction = lambda p : nnCostFunction(p, input_layer_size,␣
↪hidden_layer_size,
                                             num_labels, X_train, Y_train,␣
↪lambda_value)
```

```
    # Now, costFunction is a function that takes in only one argument (the
    # neural network parameters)
    [nn_params, cost] = fmincg(costFunction, initial_nn_params, MaxIter)

    # Obtain Theta1 and Theta2 back from nn_params
    Theta1 = np.reshape(nn_params[0:hidden_layer_size * (input_layer_size + 1)],
                              (hidden_layer_size, (input_layer_size + 1)),␣
↪order='F')
    Theta2 = np.reshape(nn_params[((hidden_layer_size * (input_layer_size +␣
↪1))):],
                              (num_labels, (hidden_layer_size + 1)),␣
↪order='F')
    #Predicting
    pred = predict(Theta1, Theta2, X_test)
    pred = np.expand_dims(pred,axis=1)
    accuracy = (pred == Y_test).mean()*100
    accuracies[lambda_value] = accuracy
    pred = predict(Theta1, Theta2, X_train)
    pred = np.expand_dims(pred,axis=1)
    training_accuracy[lambda_value] = (pred == Y_train).mean()*100
    print('Testing Set Accuracy: ', accuracy)
    print('Training Set Accuracy: ', training_accuracy[lambda_value])
    print("Final Cost: ", cost[-1])
    cost_list.append(cost)
```

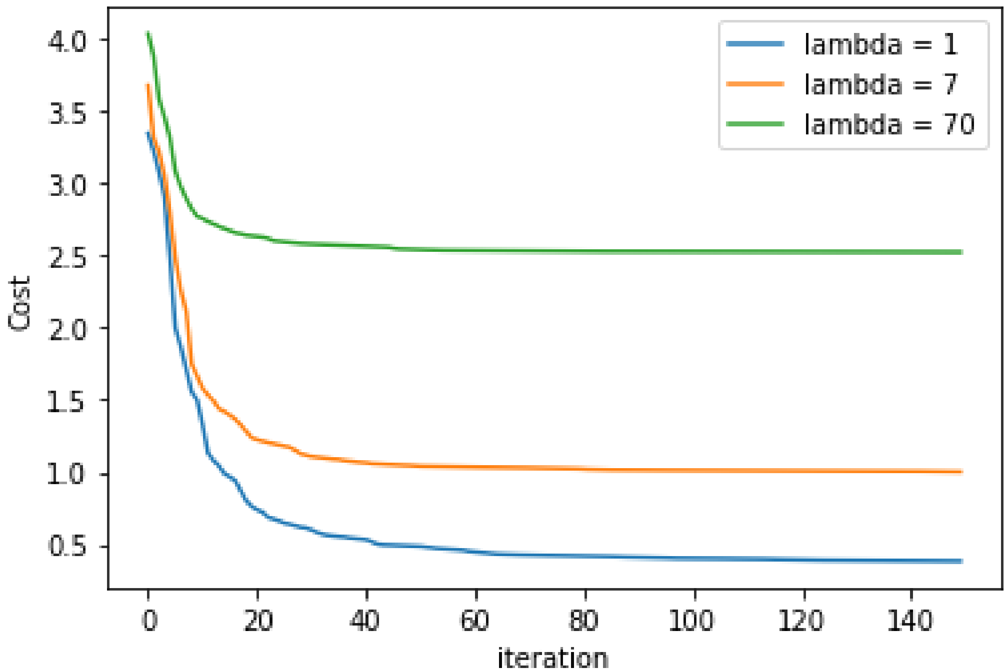
[61]:
```
import matplotlib.pyplot as plt
xs, ys = zip(*accuracies.items())
plt.plot(xs, ys, "o-")
plt.xlabel("Lambda_value")
plt.ylabel("Accuracy")
xs, ys = zip(*training_accuracy.items())
plt.plot(xs, ys, "ro-")
plt.legend(["testing_accuracy", "training_accuracy"])
```

[61]: <matplotlib.legend.Legend at 0x154165be0>



[62]:
```
for cost_arr in cost_list:
    plt.plot(range(len(cost_arr)), cost_arr, "-")

plt.xlabel("iteration")
plt.ylabel("Cost")
plt.legend(["lambda = 1", "lambda = 7", "lambda = 70"])
```

[62]: <matplotlib.legend.Legend at 0x1524ee1f0>



As one can see, the lambda value influences both the accuracy and the minimum cost. For large values of lambda, the test accuracy drops, but the training accuracy does as well. Also, the minimum cost is proportional to the value of lambda, meaning that it is bigger when lambda is bigger. However, this is to be expected, since the use of regularization is to try and prevent overfitting. So what we are trying to achieve is to find a good tradeoff between trainig and testing accuracy, for example at lambda = 1, the training accuracy is very close to 100, while the test accuracy is a little below that. One might suspect overfitting, even though it is not quite there yet, but with more iterations that effect could be amplified, as seen in the next experiment.

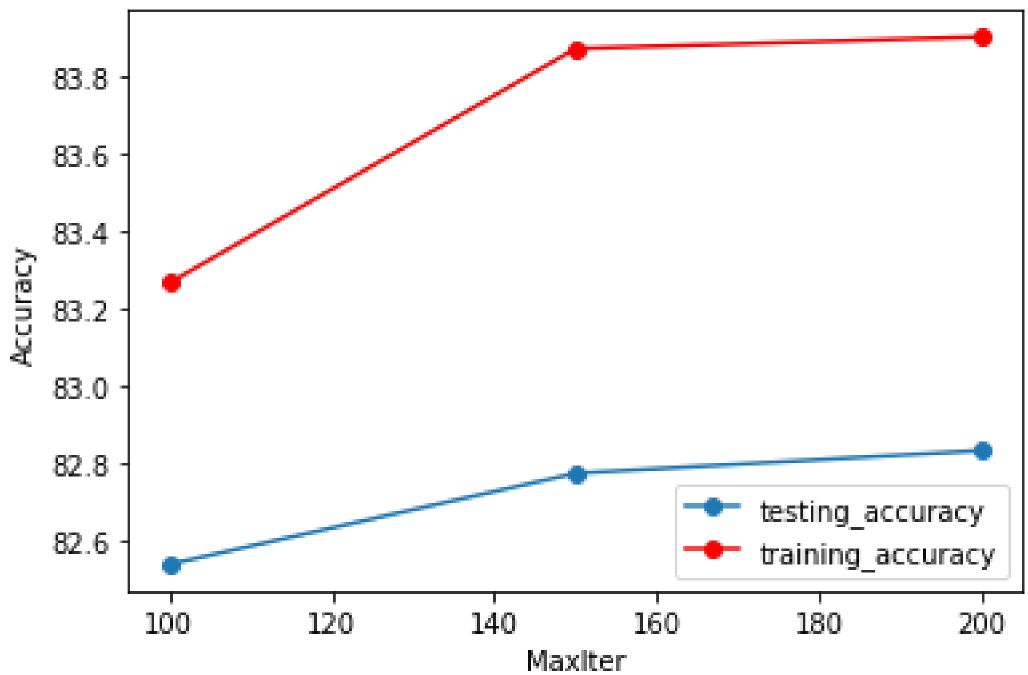
- Iteration number

[64]:
```
%%capture
from sklearn.model_selection import train_test_split
print('Loading and data ...')
mat = scipy.io.loadmat('digitdata.mat')
X = mat['X']
y = mat['y']

X_train, X_test, Y_train, Y_test = train_test_split(X, y, test_size=0.34,␣
 ↪random_state=10)
print('Training Neural Network...')
#Lambda values:
lambda_values = 7
#Accuracies
accuracies = {}
training_accuracy = {}
```

```
#Cost list
cost_list = []
#  After you have completed the assignment, change the MaxIter to a larger
#  value to see how more training helps.
iterations = [100, 150, 200]
#Training
for MaxIter in iterations:
    print('Initializing Neural Network Parameters ...')

    initial_Theta1 = randInitializeWeights(input_layer_size, hidden_layer_size)
    initial_Theta2 = randInitializeWeights(hidden_layer_size, num_labels)

    # Unroll parameters
    initial_Theta1 = np.reshape(initial_Theta1, initial_Theta1.size, order='F')
    initial_Theta2 = np.reshape(initial_Theta2, initial_Theta2.size, order='F')
    initial_nn_params = np.hstack((initial_Theta1, initial_Theta2))

    # Create "short hand" for the cost function to be minimized
    costFunction = lambda p : nnCostFunction(p, input_layer_size,␣
↪hidden_layer_size,
                                             num_labels, X_train, Y_train,␣
↪lambda_value)

    # Now, costFunction is a function that takes in only one argument (the
    # neural network parameters)
    [nn_params, cost] = fmincg(costFunction, initial_nn_params, MaxIter)

    # Obtain Theta1 and Theta2 back from nn_params
    Theta1 = np.reshape(nn_params[0:hidden_layer_size * (input_layer_size + 1)],
                              (hidden_layer_size, (input_layer_size + 1)),␣
↪order='F')
    Theta2 = np.reshape(nn_params[((hidden_layer_size * (input_layer_size +␣
↪1))):],
                              (num_labels, (hidden_layer_size + 1)),␣
↪order='F')
    #Predicting
    pred = predict(Theta1, Theta2, X_test)
    pred = np.expand_dims(pred,axis=1)
    accuracy = (pred == Y_test).mean()*100
    accuracies[MaxIter] = accuracy
    pred = predict(Theta1, Theta2, X_train)
    pred = np.expand_dims(pred,axis=1)
    training_accuracy[MaxIter] = (pred == Y_train).mean()*100
    print('Testing Set Accuracy: ', accuracy)
    print('Training Set Accuracy: ', training_accuracy[MaxIter])
    print("Final Cost: ", cost[-1])
    cost_list.append(cost)
```

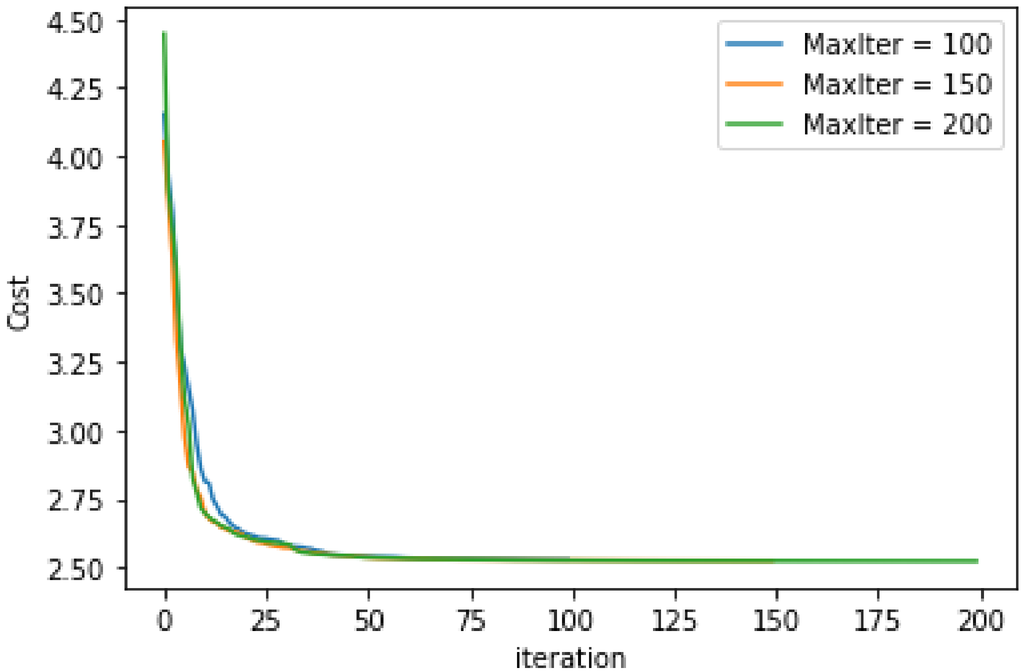
[67]:
```
import matplotlib.pyplot as plt
xs, ys = zip(*accuracies.items())
plt.plot(xs, ys, "o-")
plt.xlabel("MaxIter")
plt.ylabel("Accuracy")
xs, ys = zip(*training_accuracy.items())
plt.plot(xs, ys, "ro-")
plt.legend(["testing_accuracy", "training_accuracy"])
```

[67]:
```
<matplotlib.legend.Legend at 0x126b10910>
```



[68]:
```
for cost_arr in cost_list:
    plt.plot(range(len(cost_arr)), cost_arr, "-")

plt.xlabel("iteration")
plt.ylabel("Cost")
plt.legend(["MaxIter = 100", "MaxIter = 150", "MaxIter = 200"])
```

[68]:
```
<matplotlib.legend.Legend at 0x126b10100>
```



As one can clearly see in the above graphs, the NN achieves a higher accuracy for bigger MaxIter values, because it has more time to train. However, both accuracies increase less and less as MaxIter gets larger. So the training and testing accuracy's slope is anti-proportional to MaxIter, but the accuracies themselves are incrasing together with MaxIter until they reach a point where the NN cannot fit the data better without overfitting (note that the lambda I chose for this experiment is lambda = 7, so we do not overfit that easily). While the accuracy increases with the maximum number of iterations, the cost convergence is not as affected by it. You can clearly see the cost reaching a plateau early on, but the time when it reaches it is roughly the same for every value of MaxIter.

So, concludingly, one can say that lambda tries to prevent overfitting whilst sacrificing accuracy and cost-convergence and MaxIter increases the accuracies whilst also increasing the risk of overfitting the data. This means that a good model with nicely tuned hyperparameters should have reasonable values for lambda and MaxIter and, more importantly, a good tradeoff between their influences.

12.1.3 Improving on initial result from debugweights.mat

To improve, I simply loaded the debugweights into the cost function as initial parameters and just started optimizing from there. This model is probably highly overfit, since it has now an accuracy of 99.18% and the lambda value that was used is lambda = 1. It was trained for 100 additional iterations.

[79]:
```
print('Loading and Visualizing Data ...')

mat = scipy.io.loadmat('digitdata.mat')
X = mat['X']
```

```python
y = mat['y']

print('Loading Saved Neural Network Parameters ...')

# Load the weights into variables Theta1 and Theta2
mat = scipy.io.loadmat('debugweights.mat');

# Unroll parameters
Theta1 = mat['Theta1']
Theta1_1d = np.reshape(Theta1, Theta1.size, order='F')
Theta2 = mat['Theta2']
Theta2_1d = np.reshape(Theta2, Theta2.size, order='F')

initial_nn_params = np.hstack((Theta1_1d, Theta2_1d))
MaxIter = 100
lambda_value = 1
# Create "short hand" for the cost function to be minimized
costFunction = lambda p : nnCostFunction(p, input_layer_size, hidden_layer_size,
                                         num_labels, X, y, lambda_value)

# Now, costFunction is a function that takes in only one argument (the
# neural network parameters)
[nn_params, cost] = fmincg(costFunction, initial_nn_params, MaxIter)
# Obtain Theta1 and Theta2 back from nn_params

Theta1 = np.reshape(nn_params[0:hidden_layer_size * (input_layer_size + 1)],
                              (hidden_layer_size, (input_layer_size + 1)),␣
 ↪order='F')
Theta2 = np.reshape(nn_params[((hidden_layer_size * (input_layer_size + 1))):],
                              (num_labels, (hidden_layer_size + 1)), order='F')
```

```
Loading and Visualizing Data …
Loading Saved Neural Network Parameters …
Iteration  1 | Cost:  [0.3810212]
Iteration  2 | Cost:  [0.37127893]
Iteration  3 | Cost:  [0.36463237]
Iteration  4 | Cost:  [0.36100255]
Iteration  5 | Cost:  [0.35021463]
Iteration  6 | Cost:  [0.34503721]
Iteration  7 | Cost:  [0.34341822]
Iteration  8 | Cost:  [0.34133862]
Iteration  9 | Cost:  [0.33953897]
Iteration  10 | Cost:  [0.33740099]
Iteration  11 | Cost:  [0.33596374]
Iteration  12 | Cost:  [0.33504536]
Iteration  13 | Cost:  [0.33320868]
Iteration  14 | Cost:  [0.33260408]
Iteration  15 | Cost:  [0.33188161]
```

```
Iteration  16 | Cost:  [0.33108026]
Iteration  17 | Cost:  [0.33055596]
Iteration  18 | Cost:  [0.3300339]
Iteration  19 | Cost:  [0.3286211]
Iteration  20 | Cost:  [0.32799557]
Iteration  21 | Cost:  [0.32746564]
Iteration  22 | Cost:  [0.32698966]
Iteration  23 | Cost:  [0.32615938]
Iteration  24 | Cost:  [0.32560775]
Iteration  25 | Cost:  [0.32530679]
Iteration  26 | Cost:  [0.32514242]
Iteration  27 | Cost:  [0.3251085]
Iteration  28 | Cost:  [0.3248847]
Iteration  29 | Cost:  [0.32480782]
Iteration  30 | Cost:  [0.32462838]
Iteration  31 | Cost:  [0.32421904]
Iteration  32 | Cost:  [0.32389556]
Iteration  33 | Cost:  [0.32348843]
Iteration  34 | Cost:  [0.32327297]
Iteration  35 | Cost:  [0.32297332]
Iteration  36 | Cost:  [0.32268525]
Iteration  37 | Cost:  [0.32245862]
Iteration  38 | Cost:  [0.32218229]
Iteration  39 | Cost:  [0.32207838]
Iteration  40 | Cost:  [0.32200989]
Iteration  41 | Cost:  [0.32192025]
Iteration  42 | Cost:  [0.32170581]
Iteration  43 | Cost:  [0.32161984]
Iteration  44 | Cost:  [0.32152271]
Iteration  45 | Cost:  [0.32139461]
Iteration  46 | Cost:  [0.32134078]
Iteration  47 | Cost:  [0.32132509]
Iteration  48 | Cost:  [0.32125513]
Iteration  49 | Cost:  [0.3212128]
Iteration  50 | Cost:  [0.32115362]
Iteration  51 | Cost:  [0.32104472]
Iteration  52 | Cost:  [0.32058975]
Iteration  53 | Cost:  [0.32011243]
Iteration  54 | Cost:  [0.32004432]
Iteration  55 | Cost:  [0.32000438]
Iteration  56 | Cost:  [0.31995882]
Iteration  57 | Cost:  [0.31990999]
Iteration  58 | Cost:  [0.31985374]
Iteration  59 | Cost:  [0.31980389]
Iteration  60 | Cost:  [0.31968306]
Iteration  61 | Cost:  [0.31950328]
Iteration  62 | Cost:  [0.31935435]
Iteration  63 | Cost:  [0.31928132]
```

```
Iteration  64 | Cost:  [0.31923547]
Iteration  65 | Cost:  [0.31913285]
Iteration  66 | Cost:  [0.31905641]
Iteration  67 | Cost:  [0.31897811]
Iteration  68 | Cost:  [0.31884922]
Iteration  69 | Cost:  [0.31874822]
Iteration  70 | Cost:  [0.31867737]
Iteration  71 | Cost:  [0.31861595]
Iteration  72 | Cost:  [0.31853479]
Iteration  73 | Cost:  [0.3184923]
Iteration  74 | Cost:  [0.31846892]
Iteration  75 | Cost:  [0.31843724]
Iteration  76 | Cost:  [0.31840971]
Iteration  77 | Cost:  [0.31839738]
Iteration  78 | Cost:  [0.31827248]
Iteration  79 | Cost:  [0.31814848]
Iteration  80 | Cost:  [0.31801761]
Iteration  81 | Cost:  [0.3179069]
Iteration  82 | Cost:  [0.31777663]
Iteration  83 | Cost:  [0.31762719]
Iteration  84 | Cost:  [0.31746506]
Iteration  85 | Cost:  [0.31738721]
Iteration  86 | Cost:  [0.31735524]
Iteration  87 | Cost:  [0.31722898]
Iteration  88 | Cost:  [0.31719164]
Iteration  89 | Cost:  [0.31716785]
Iteration  90 | Cost:  [0.31716145]
Iteration  91 | Cost:  [0.31714139]
Iteration  92 | Cost:  [0.31711694]
Iteration  93 | Cost:  [0.31710172]
Iteration  94 | Cost:  [0.31707391]
Iteration  95 | Cost:  [0.31702016]
Iteration  96 | Cost:  [0.31699543]
Iteration  97 | Cost:  [0.31694357]
Iteration  98 | Cost:  [0.3168766]
Iteration  99 | Cost:  [0.31681225]
Iteration  100 | Cost:  [0.31676604]
```

[80]:

```
pred = predict(Theta1, Theta2, X)
pred = np.expand_dims(pred,axis=1)
print((pred == y).mean()*100)
```

```
99.18
```

12.1.4 Imagine that you want to use a similar solution to classify 50x50 pixel grayscale images containing letters (consider an alphabet with 26 letters). Which changes would you need in the current code in order to implement this classification task?

First of all, the input layer must be of size 50 * 50 = 2500 neurons. Then, the output layer should also resemble 26 classes instead of 10. This means that we would have way more parameters and a more complex problem, since our solution space became a lot bigger. So instead of 401 * 26 + 26 * 10 = 10686 parameters, we would now have to train 2500 * 26 + 26 * 26 = 65676 parameters, and that is only if we do not make any internal changes. Since the problem is more complex (we need to discriminate between more classes), we probably would also need additional internal neurons, maybe in different layers, which makes the optimization even more harder to solve. However, with convolutions, this could be achieved, since then we can efficiently decreased the amount of parameters to be trained because we would not have fully connected layers everywhere. Concretely, you would have to change input_layer_size, hidden_layer_size and num_labels, because based on these, the NN is built. Adding more layers or even convolutions with the current code would pose more of a challenge, because this version of the code is rather "hardcoded" to only have one hidden layer (we do not instantiate a list of weight matrices, but always use Theta1 and Theta2, which makes for a big effort to change in every place we use it).

12.1.5 How does your sigmoidGradient function work? Which is the return value for different values of z? How does it work with the input is a vector and with it is a matrix?

Yes, it does, because numpy is doing the heavy load for us in this. See comments in the code at sigmoidGradient.py and sigmoid.py. I tested it with a couple of values and they all seem to fine.

12.1.6 Change the value of the variable show_examples (in the python version, run the relevant block in the Jupyter one) in ex_nn, which information is provided? Did you get the expected information? Is anything unexpected there?

Cannot find this variable in the code. I suspect the block where 100 examples are displayed is meant. When I change this value, less examples are shown, however the value must be a square, so 2, 4, 8, 16,... for the code to run properly. This is probably because only then it can be shown in a suare format.