ex_nn

December 3, 2021

1 Lena Feiler - i6246119

2 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.

2.1 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
```

2.2 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)
```

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

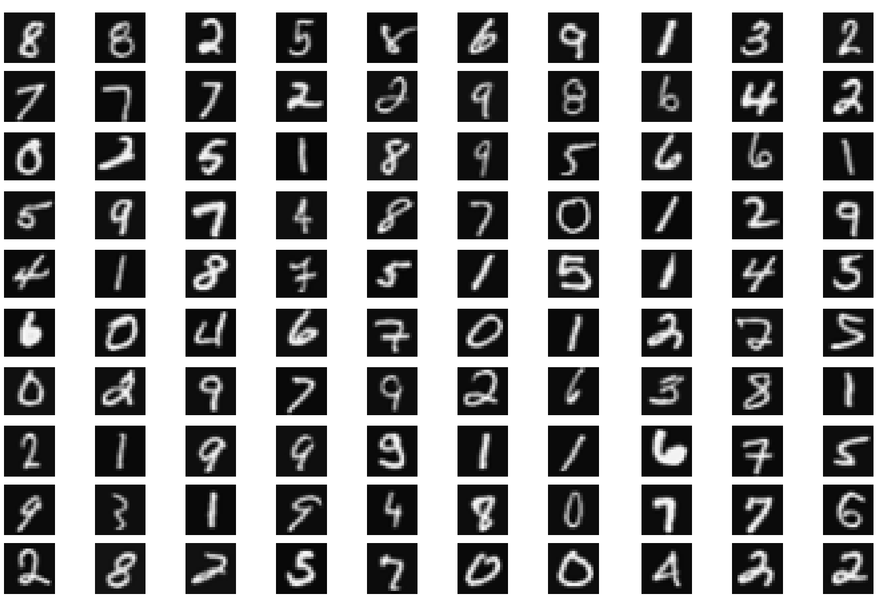
We start the exercise by first loading and visualizing the dataset. You will be working with a dataset that contains handwritten digits.

3.1 Load Training Data

[3]:
```
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]), 100)
sel = X[sel,:]

displayData(sel)
```

```
Loading and Visualizing Data …
```

```
/Users/lena/DKE/Year_2/period_2.2/machine_learning/Lab5/displayData.py:8:
MatplotlibDeprecationWarning: Passing non-integers as three-element position
specification is deprecated since 3.3 and will be removed two minor releases
later.
  ax = fig.add_subplot(np.sqrt(data.shape[0]), np.sqrt(data.shape[0]), i+1)
```



4 ================== 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 …
```

5 =================== 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
```

5.1 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

[ ]:
```
#  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)

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

6 ================== 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.

[6]:
```
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]
```

7 ================== 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)

[7]:
```
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 …
[-0.09458821 -0.08262094 -0.04324267 …  0.07132938 -0.03409781
 -0.10495504]
```

8 ================= 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.

[8]:

```
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:  2.2957544655995076e-11
```

9 ================ Part 7: Implement Regularization ================

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

[9]:
```
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:  2.2006043191433586e-11
Cost at (fixed) debugging parameters (w/ lambda = 10):  0.5760512469501331 (this
value should be about 0.576051)
```

10 ==================== 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.

[10]:
```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:  [3.28439445]
Iteration   2 | Cost:  [3.24307806]
```

```
Iteration  3 | Cost:  [3.20108698]
Iteration  4 | Cost:  [2.88219067]
Iteration  5 | Cost:  [2.5838405]
Iteration  6 | Cost:  [2.40483167]
Iteration  7 | Cost:  [2.11419278]
Iteration  8 | Cost:  [1.85711249]
Iteration  9 | Cost:  [1.71349158]
Iteration  10 | Cost:  [1.53586494]
Iteration  11 | Cost:  [1.46577259]
Iteration  12 | Cost:  [1.43507973]
Iteration  13 | Cost:  [1.31211977]
Iteration  14 | Cost:  [1.27110142]
Iteration  15 | Cost:  [1.23787513]
Iteration  16 | Cost:  [1.1909416]
Iteration  17 | Cost:  [1.13230706]
Iteration  18 | Cost:  [1.10533307]
Iteration  19 | Cost:  [1.07781607]
Iteration  20 | Cost:  [1.01960256]
Iteration  21 | Cost:  [0.94854419]
Iteration  22 | Cost:  [0.91385642]
Iteration  23 | Cost:  [0.89375593]
Iteration  24 | Cost:  [0.84258204]
Iteration  25 | Cost:  [0.82934597]
Iteration  26 | Cost:  [0.81361549]
Iteration  27 | Cost:  [0.77770115]
Iteration  28 | Cost:  [0.73481581]
Iteration  29 | Cost:  [0.69344115]
Iteration  30 | Cost:  [0.66212686]
Iteration  31 | Cost:  [0.64897138]
Iteration  32 | Cost:  [0.63451819]
Iteration  33 | Cost:  [0.62062928]
Iteration  34 | Cost:  [0.61105205]
Iteration  35 | Cost:  [0.60394273]
Iteration  36 | Cost:  [0.58742808]
Iteration  37 | Cost:  [0.57452468]
Iteration  38 | Cost:  [0.56952945]
Iteration  39 | Cost:  [0.56368648]
Iteration  40 | Cost:  [0.55752858]
Iteration  41 | Cost:  [0.55443359]
Iteration  42 | Cost:  [0.550767]
Iteration  43 | Cost:  [0.53937685]
Iteration  44 | Cost:  [0.52041994]
Iteration  45 | Cost:  [0.50907771]
Iteration  46 | Cost:  [0.49821091]
Iteration  47 | Cost:  [0.48588888]
Iteration  48 | Cost:  [0.47916008]
Iteration  49 | Cost:  [0.47746599]
Iteration  50 | Cost:  [0.47027926]
```

```
Iteration  51 | Cost:  [0.4673045]
Iteration  52 | Cost:  [0.46576225]
Iteration  53 | Cost:  [0.46520005]
Iteration  54 | Cost:  [0.46278354]
Iteration  55 | Cost:  [0.46105291]
Iteration  56 | Cost:  [0.46003879]
Iteration  57 | Cost:  [0.45903774]
Iteration  58 | Cost:  [0.45577482]
Iteration  59 | Cost:  [0.45194366]
Iteration  60 | Cost:  [0.45112513]
Iteration  61 | Cost:  [0.44878775]
Iteration  62 | Cost:  [0.43973194]
Iteration  63 | Cost:  [0.43607229]
Iteration  64 | Cost:  [0.43369701]
Iteration  65 | Cost:  [0.43247161]
Iteration  66 | Cost:  [0.43064263]
Iteration  67 | Cost:  [0.4290564]
Iteration  68 | Cost:  [0.42836698]
Iteration  69 | Cost:  [0.42747387]
Iteration  70 | Cost:  [0.4268125]
Iteration  71 | Cost:  [0.42459993]
Iteration  72 | Cost:  [0.42323659]
Iteration  73 | Cost:  [0.42008503]
Iteration  74 | Cost:  [0.41801322]
Iteration  75 | Cost:  [0.41560441]
Iteration  76 | Cost:  [0.41322673]
Iteration  77 | Cost:  [0.41148567]
Iteration  78 | Cost:  [0.41010694]
Iteration  79 | Cost:  [0.40746177]
Iteration  80 | Cost:  [0.40438447]
Iteration  81 | Cost:  [0.40350454]
Iteration  82 | Cost:  [0.40298928]
Iteration  83 | Cost:  [0.40288494]
Iteration  84 | Cost:  [0.40249783]
Iteration  85 | Cost:  [0.40204832]
Iteration  86 | Cost:  [0.40168254]
Iteration  87 | Cost:  [0.40041093]
Iteration  88 | Cost:  [0.39834516]
Iteration  89 | Cost:  [0.39302062]
Iteration  90 | Cost:  [0.38847072]
Iteration  91 | Cost:  [0.38694872]
Iteration  92 | Cost:  [0.38494207]
Iteration  93 | Cost:  [0.38315387]
Iteration  94 | Cost:  [0.38099824]
Iteration  95 | Cost:  [0.3790177]
Iteration  96 | Cost:  [0.3775966]
Iteration  97 | Cost:  [0.37645034]
Iteration  98 | Cost:  [0.37495215]
```

```
Iteration  99 | Cost:  [0.3742671]
Iteration  100 | Cost:  [0.37389859]
Iteration  101 | Cost:  [0.37360405]
Iteration  102 | Cost:  [0.37312382]
Iteration  103 | Cost:  [0.37237434]
Iteration  104 | Cost:  [0.37194321]
Iteration  105 | Cost:  [0.37173018]
Iteration  106 | Cost:  [0.37118032]
Iteration  107 | Cost:  [0.37082975]
Iteration  108 | Cost:  [0.3707602]
Iteration  109 | Cost:  [0.37049161]
Iteration  110 | Cost:  [0.37038189]
Iteration  111 | Cost:  [0.37035186]
Iteration  112 | Cost:  [0.37014281]
Iteration  113 | Cost:  [0.36988736]
Iteration  114 | Cost:  [0.36965214]
Iteration  115 | Cost:  [0.36889956]
Iteration  116 | Cost:  [0.36699026]
Iteration  117 | Cost:  [0.36205054]
Iteration  118 | Cost:  [0.35635538]
Iteration  119 | Cost:  [0.3548609]
Iteration  120 | Cost:  [0.3544416]
Iteration  121 | Cost:  [0.3533465]
Iteration  122 | Cost:  [0.35234534]
Iteration  123 | Cost:  [0.35182596]
Iteration  124 | Cost:  [0.35149403]
Iteration  125 | Cost:  [0.35120181]
Iteration  126 | Cost:  [0.35087614]
Iteration  127 | Cost:  [0.35041806]
Iteration  128 | Cost:  [0.35023378]
Iteration  129 | Cost:  [0.35017133]
Iteration  130 | Cost:  [0.34998164]
Iteration  131 | Cost:  [0.34989407]
Iteration  132 | Cost:  [0.34979186]
Iteration  133 | Cost:  [0.34954061]
Iteration  134 | Cost:  [0.34920649]
Iteration  135 | Cost:  [0.34883394]
Iteration  136 | Cost:  [0.34871074]
Iteration  137 | Cost:  [0.34861334]
Iteration  138 | Cost:  [0.34849444]
Iteration  139 | Cost:  [0.34823424]
Iteration  140 | Cost:  [0.3476669]
Iteration  141 | Cost:  [0.34708856]
Iteration  142 | Cost:  [0.34641996]
Iteration  143 | Cost:  [0.34518428]
Iteration  144 | Cost:  [0.34467471]
Iteration  145 | Cost:  [0.34395803]
Iteration  146 | Cost:  [0.34329639]
```

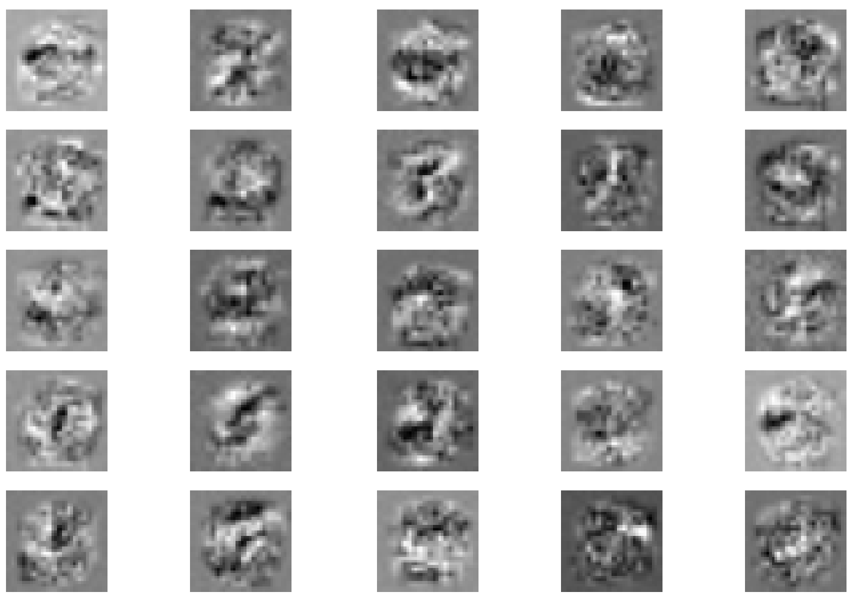
```
Iteration  147 | Cost:  [0.34304632]
Iteration  148 | Cost:  [0.3429067]
Iteration  149 | Cost:  [0.34275092]
Iteration  150 | Cost:  [0.34256184]
```

11 ================== 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.

[11]:
```python
print('\nVisualizing Neural Network... \n')

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

```
Visualizing Neural Network…
```



12 ============== 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.

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

```
Training Set Accuracy:  98.9
```

```
[ ]:
```

13 Evaluation/ Report

13.1 Description of the network (e.g. number of layers, number of neurons per layer, etc.)

The network, as described in the assignment, has 3 layers, the input and output layer and one hidden layer. The input layer has 401 units, where 1 is the bias node. The hidden layer has 26 neurons (again, 1 bias node) and the output layer has 10. The output units correspond to the 10 digit classes.

13.2 Impact of specific parameters such as , number of iterations, weight initialization, etc.

- Impact of lambda:

  Lambda is our regularization rate. It is used in the nnCostFunction in the regularization process. It helps us limit how much regularisation we allow. Increasing its value effect of the regularization. If its value is increased too much, the model will become more general and can then happen to underfit the given data. On the other hand, if it is too low, we risk overfitting the data, since our model becomes more complex, since we allow the model to learn more about the training data.

- Impact of weight initialization:

  If both bias and weights are initialized to 0, all neurons will learn the same, since the output of the neurons and the backpropagation of th error, respectively, are the same. Otherwise, if all values are initialized randomly, the vanishing gradient can occur, since the values can be very high and therefore the value of the activation function is close to 0. Therefore, we know that the initialization of the weights can have a very high impact.

- Impact of max iterations:

  Increasing the number of allowed iterations (maxIterations), can help decrease the error and increase the accuracy of the model. Therefore, by increasing the number of iterations, the error will get closer to 0, but does not necessarily have to converge.

- Impact of weight epsilon:

  Epsilon is the range of the weight initialization. If epsilon is large, the random weights may take values that are very different from one another, which can also create outliers. On the other hand, if epsilon is small, the weights will be more similar to one another, which can make learning more efficient.

13.3 How does the regularization affect the training of your ANN?

Regularization is used to reduce the degree of freedom used to update and hence only make small modifications to the applied learning algorithm, in order to improve the model's generalization. Therefore, it may improve the performance on unseen data, since it limits overfitting of the model.

Running the code in part 7 with regularization and without regularization, gives us the following values:

> with Regularization: Relative Difference: 2.2006043191433586e-11
>
> without Regularization: Relative Difference: 0.22177454344825828

This shows that regurlarization decreased the relative difference between the given numerical Gradient, and the acquired Analytical Gradient, prodcued by the ANN. Therefore, the produced output has become more accurate.

13.4 Did you manage to improve the initial results (using values in debugweights.mat)? Which was your best result? How did you configure the system? How could you improve them even more?

At the beginning of 'Part 3: Implement Predict', we check the accuracy of the model using the values in debugweights.mat, using only the feedforward implementation. This gives us the following accuracy:

> Training Set Accuracy: 97.52

After implementing the backpropagation, the accuracy is again checked in 'Part 10: Predicting with learned weights'. This gives us:

> Training Set Accuracy: 98.66

This means, that our trained weights perform better than the initial given weights. Therefore, implementing the backpropagation leads to an increase in accuracy.

To Further improve, we could add more neurons in the hidden layer, or more layers. However, this can also lead to the model overfitting and loosing generalization.

13.5 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?

To apply the current code to the new problem, we would only need to redefine the input values in the section **Setup the parameters you will use for this exercise**.

In our current network we have a 20x20 grayscale image. So, to change to a 50x50 pixel grayscale image, we would have to change the **input layer** to have **2501 units** (2500 + 1 bias node).

Furthermore, since we now have 26 letters we wanr to classify, instead of 10 numbers, the **output layer** should be adjusted to have **27 neurons** (26 for the letters and 1 bais unit).

For the **hidden layer**, no explicit number is specified. However, it could be considered to add more layers. This has a chance of improving the accuracy, although this is not guaranteed, and might even cause the accuracy to decrease, if too many layers are added, since the model will end up overfitting. However, if we would add more hidden layers, we would need to change more parts of the code, therefore, we will continue working with 1 layer. For the number of neurons in the hidden layer(s), the number is also not defined. Since the problems' complexity ahs increased, we should also increase the number of hidden neurons, and for example use 501 neurons. This value can be adjusted.

So, our changes would be: > input_layer_size = 2500;

> hidden_layer_size = 500;
>
> num_labels = 26;

13.6 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?

Show_examples is a boolean variable. Therefore, we change its value from False to True. When executing the corresponding part in jupyter notebook, we are shown, for example:

> 21
>
> Displaying Example Image
>
> Neural Network Prediction: [6] (digit [6])

So, they show the number of the examples that is being displayed, the sentence 'Displaying Example Image', the digit that was detected '[6]', as well as the actual value of the current example '(digit [6])'. Furthermore, the image that is being analyzed is also shown.

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

The sigmoidGradient nethod computes the gradient of the sigmoid function evaluated at every given value of z. The method returns the sigmoid function of z, mulitplied with the sigmoid function of z substracted from 1, so: > g = sigmoid(z) * (1 - sigmoid(z))

The method is defined to be able to take z as a matrix, vector or scalar. Each example can also be seen in the code below. The type of the input value is the same as the type of the output value.

1. If z is a scalar, the output value wil be also be a scalar.
2. If z is a vector, the output value wil be also be a vector.
3. If z is a matrix, the output value wil be also a matrix.

[13]:
```
# scalar
sclar = 1
s = sigmoidGradient(1)
print('output of a scalar input: ', s)
print('shape: ', np.shape(s))
```

```python
# vector
vector = np.array([-15, -1, -0.5, 0, 0.5, 1, 15])
v = sigmoidGradient(vector)
print('output of a vector input: ', v)
print('shape: ', np.shape(v))

# matrix
matrix = np.array([[4,7,6],[1,2,5],[9,3,8]])
m = sigmoidGradient(matrix)
print('output of a matrix inout: ', m)
print('shape: ', np.shape(m))
```

```
output of a scalar input:  0.19661193324148185
shape:  ()
output of a vector input:  [3.05902133e-07 1.96611933e-01 2.35003712e-01
2.50000000e-01
 2.35003712e-01 1.96611933e-01 3.05902133e-07]
shape:  (7,)
output of a matrix inout:  [[1.76627062e-02 9.10221180e-04 2.46650929e-03]
 [1.96611933e-01 1.04993585e-01 6.64805667e-03]
 [1.23379350e-04 4.51766597e-02 3.35237671e-04]]
shape:  (3, 3)
```

[ ]:

```python

```