DS4023 Machine Learning : Neural Network Exercise

Introduction

In this exercise, you will implement the backpropagation algorithm for neural networks and apply it to the task of hand-written digit recognition.

1. Data loading and visualization

In [1]:

```
# import necessary packages
import numpy as np
import pandas as pd
import matplotlib
import matplotlib.pyplot as plt
from scipy.io import loadmat
```

In [2]:

```
#load and check the data
data = loadmat('ex4data1.mat')
data
```

Out[2]:

```
{'X': array([[0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        ...,
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.]]),
 '__globals__': [],
 '__header__': b'MATLAB 5.0 MAT-file, Platform: GLNXA64, Created on: Sun Oct 16 13:0
9:09 2011',
 '__version__': '1.0',
 'y': array([[10],
        [10],
        [10],
        ...,
        [ 9],
        [ 9],
        [ 9]], dtype=uint8)}
```

There are 5000 training examples in ex3data1.mat, where each training example is a 20 pixel by 20 pixel grayscale image of the digit. Each pixel is represented by a floating point number indicating the grayscale intensity at that location. The 20 by 20 grid of pixels is “unrolled” into a 400-dimensional vector. Each of these training examples becomes a single row in our data matrix X. This gives us a 5000 by 400 matrix X where every row is a training example for a handwritten digit image.

In [3]:

```
X = data['X']
y = data['y']

X.shape, y.shape#check the shape of your data
```

Out[3]:

```
((5000, 400), (5000, 1))
```

In [4]:

```
def plot_100_image(X):
    """ sample 100 image and show them
    assume the image is square

    X : (5000, 400)
    """
    size = int(np.sqrt(X.shape[1]))

    # sample 100 image, reshape, reorg it
    sample_idx = np.random.choice(np.arange(X.shape[0]), 100)  # 100*400
    sample_images = X[sample_idx, :]

    fig, ax_array = plt.subplots(nrows=10, ncols=10, sharey=True, sharex=True, figsize=(8, 8))

    for r in range(10):
        for c in range(10):
            ax_array[r, c].matshow(sample_images[10 * r + c].reshape((size, size)),
                                   cmap=matplotlib.cm.binary)
            plt.xticks(np.array([]))
            plt.yticks(np.array([]))
```

In [5]:

```
plot_100_image(X)
plt.show()
```

ToDo:

Please explain here why the above figure looks weird and what did I do in the block below to make it look right? You explaination goes below:

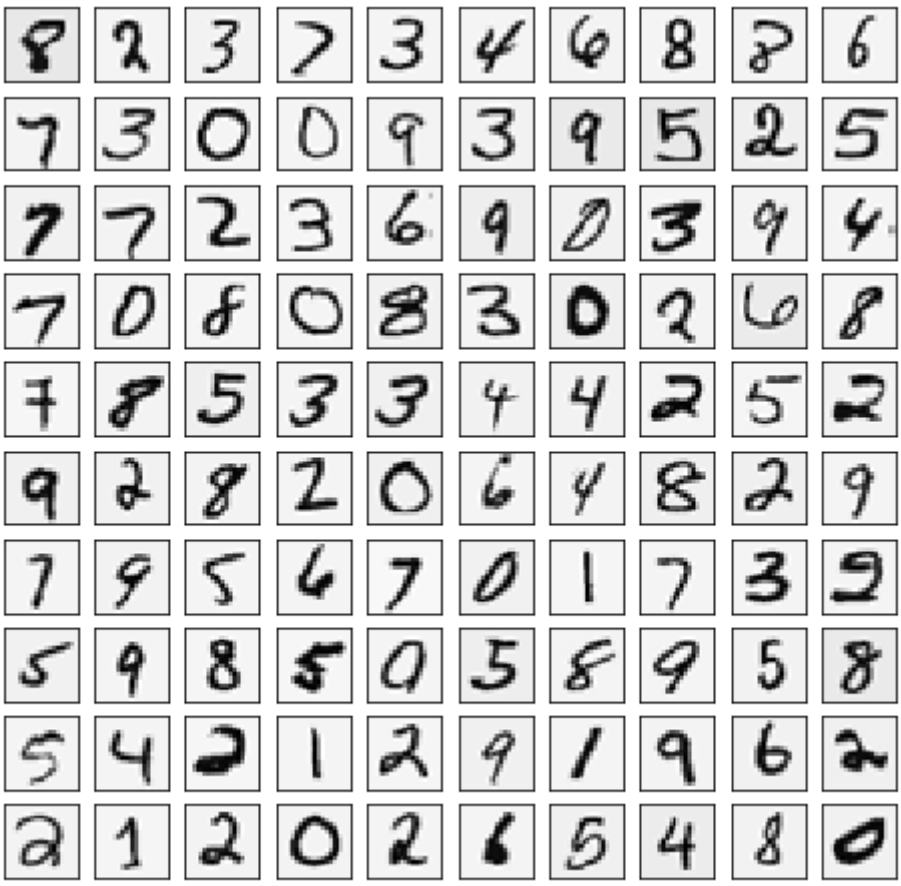
##

In [6]:

```
# for this dataset, you need a transpose to get the orientation right
X = np.array([im.reshape((20, 20)).T for im in X])

# and I flat the image again to preserve the vector presentation
X = np.array([im.reshape(400) for im in X])
```

In [7]:

```
plot_100_image(X)
plt.show()
```

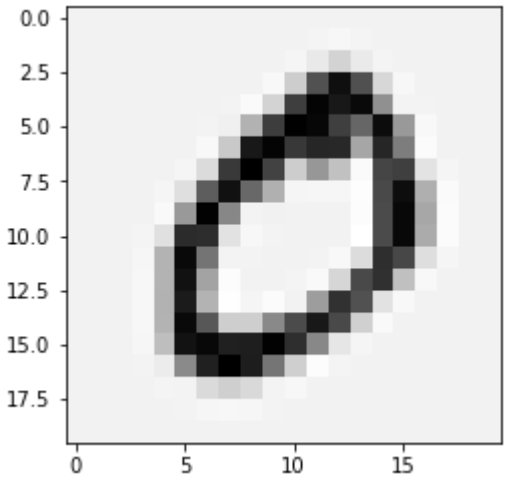


In [8]:

```
# following the plot_100_image() function, please implement a plot_image function that take a (400,)
# display it as a 20x20 image
def plot_image(x):
    """ assume the image is square
    display a 20x20 image

    x : (400,)

    """
    #complete the following
    size = ...

    plt.imshow(...)
```

In [9]:

```
plot_image(X[0])
plt.show()
```



whereas the original labels (in the variable y) were 1, 2, ..., 10, for the purpose of training a neural network, we need to recode the labels as vectors containing only values 0 or 1, so that For example, if $x^{(i)}$ is an image of the digit 5, then the corresponding $y^{(i)}$ (that you should use with the cost function) should be a 10-dimensional vector with y_5 = 1, and the other elements equal to 0. This is called "one-hot encoding"

$$y = \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \quad \begin{bmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \quad \dots \quad \text{or} \quad \begin{bmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 1 \end{bmatrix}.$$

In [10]:

```
#finish the code below to implement "one-hot" encoding of y
from sklearn.preprocessing import OneHotEncoder
encoder = OneHotEncoder(sparse=False)
y_onehot = ...
y_onehot.shape
```

```
C:\Users\xyz_p\Miniconda3\envs\trafficsign\lib\site-packages\sklearn\preprocessing\_
encoders.py:363: FutureWarning: The handling of integer data will change in version
0.22. Currently, the categories are determined based on the range [0, max(values)],
while in the future they will be determined based on the unique values.
If you want the future behaviour and silence this warning, you can specify "categori
es='auto'".
In case you used a LabelEncoder before this OneHotEncoder to convert the categories
to integers, then you can now use the OneHotEncoder directly.
  warnings.warn(msg, FutureWarning)
```

Out[10]:

```
(5000, 10)
```

In [11]:

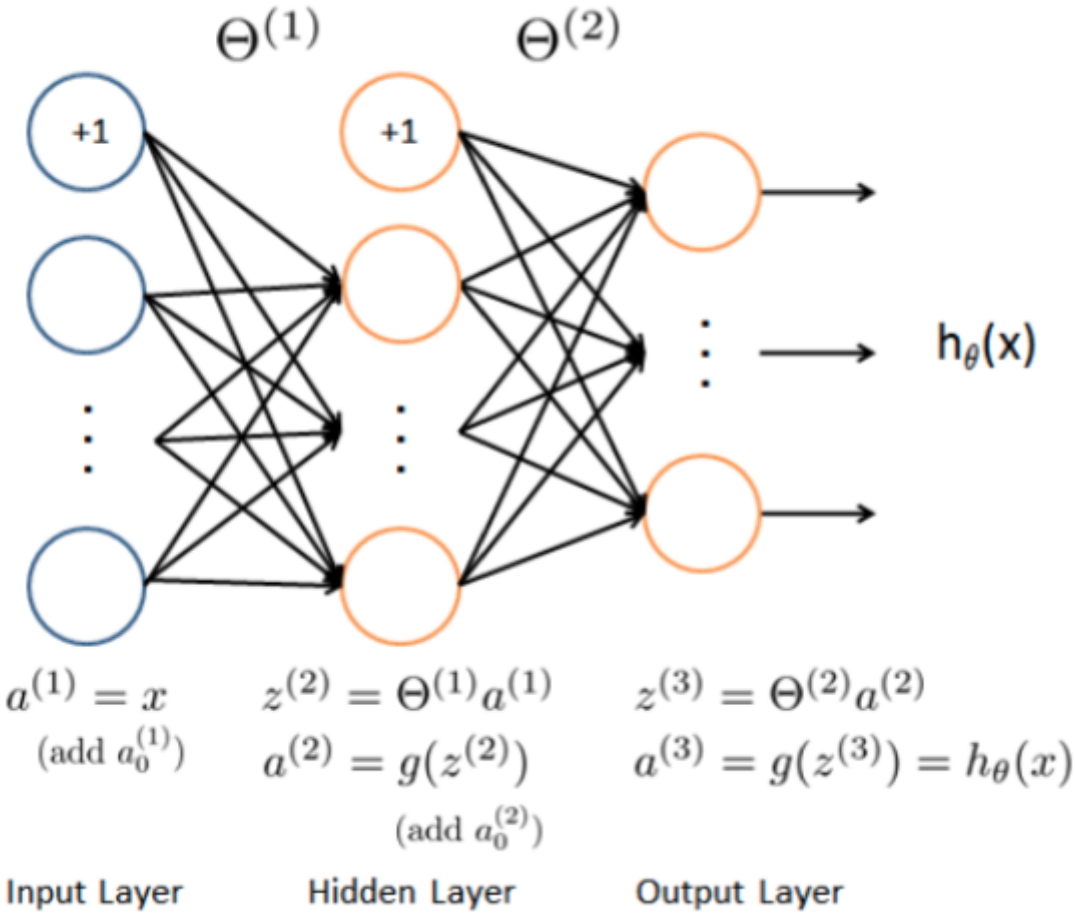
```
# check if you get the correct format of y
y[0], y_onehot[0,:]
```

Out[11]:

```
(array([10], dtype=uint8), array([0., 0., 0., 0., 0., 0., 0., 0., 0., 1.]))
```

2. Model Representation

Our neural network is shown in Figure below. It has 3 layers – an input layer, a hidden layer and an output layer. Recall that our inputs are pixel values of digit images. Since the images are of size 20 × 20, this gives us 400 input layer units (not counting the extra bias unit which always outputs 1). The hidden layer has 25 units. And the output layer has 10 units(corresponding to the 10 digit classes).



Initialized the network shapes

In [12]:

```
# Initialized the shape of the network
input_size =
hidden_size =
num_labels =
```

2.1 Sigmoid function implementation

The activation function we use in this model is the sigmoid function:

$$g(z) = \frac{1}{1 + e^{-z}}$$

In [13]:

```
# Please implement the sigmoid function below
def sigmoid(z):
```

2.2 Feedforward

We need to finish the feed forward calculation of output of th network, following the network model. The output of each unit is given by:

$$h_\theta(x) = \frac{1}{1 + e^{-\theta^T X}}$$

Pay attention that for input and hidden layer, we need to add a "1" to the original vector, as the bias term.

In [14]:

```
# Implement the feed forward calculation function below
# theta1 has size ?
# theta2 has size ?
def forward_propagate(X, theta1, theta2):
    m =

    a1 =
    z2 =
    a2 =
    z3 =
    h =

    return a1, z2, a2, z3, h
```

2.3 Cost funciton

Recall that the cost function for the neural network (without regularization) is

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K}\left[-y_k^{(i)}\log((h_\theta(x^{(i)}))_k) - (1-y_k^{(i)})\log(1-(h_\theta(x^{(i)}))_k)\right],$$

You should implement the feedforward computation that computes $h_\theta(x^{(i)})$ for every example i and sum the cost over all examples. Your code should also work for a dataset of any size, with any number of labels (you can assume that there are always at least $K \geq 3$ labels)

In [15]:

```
# Implement the cost function below

def cost(theta1, theta2, X, y):
    m =
    X =
    y =

    # run the feed-forward pass
    a1, z2, a2, z3, h = forward_propagate(X, theta1, theta2)

    # compute the cost

    J =

    return ...
```

Now that you have implemented the cost function, before we move on, we should try if our implementation is correct so far. To do the test, please use the parameters theta prepared for you and calculate the cost value in this case.

If you get a cost value equal to approximately 5.83, congratulations you are ok so far and you may move on to the next steps.

In [16]:

```
import scipy.io as sio
def load_weight(path):
    data = sio.loadmat(path)
    return data['Theta1'], data['Theta2']
```

In [17]:

```
theta1, theta2 = load_weight('ex4weights.mat')
print(theta1.shape)
print(theta2.shape)
```

```
(25, 401)
(10, 26)
```

In [18]:

```
cost(theta1, theta2, X, y_onehot)
```

Out[18]:

```
5.830351257959722
```

2.4 Regularized Cost function

The cost function for neural networks with regularization is given by

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K}\left[-y_k^{(i)}\log((h_\theta(x^{(i)}))_k) - (1-y_k^{(i)})\log(1-(h_\theta(x^{(i)}))_k)\right] + \frac{\lambda}{2m}\left[\sum_{j=1}^{25}\sum_{k=1}^{400}(\Theta_{j,k}^{(1)})^2 + \sum_{j=1}^{10}\sum_{k=1}^{25}(\Theta_{j,k}^{(2)})^2\right].$$

Note that you should not be regularizing the terms that correspond to the bias. For the matrices theta1 and theta2, this corresponds to the first column of each matrix. You should now add regularization to your cost function. Notice that you can first compute the unregularized cost function J and then later add the cost for the regularization terms.

In [19]:

```
def cost_regularized(theta1, theta2, X, y, reg_lambda=1):
    m =


    #
    a1, z2, a2, z3, h = forward_propagate(X, theta1, theta2)

    # compute the cost
    J =

    # add the cost regularization term
    J +=

    return J
```

To test your implementation, calculate the regularized cost under given theta1 and theta2, if you get a cost roughly at 5.9264, then you can move on

In [20]:

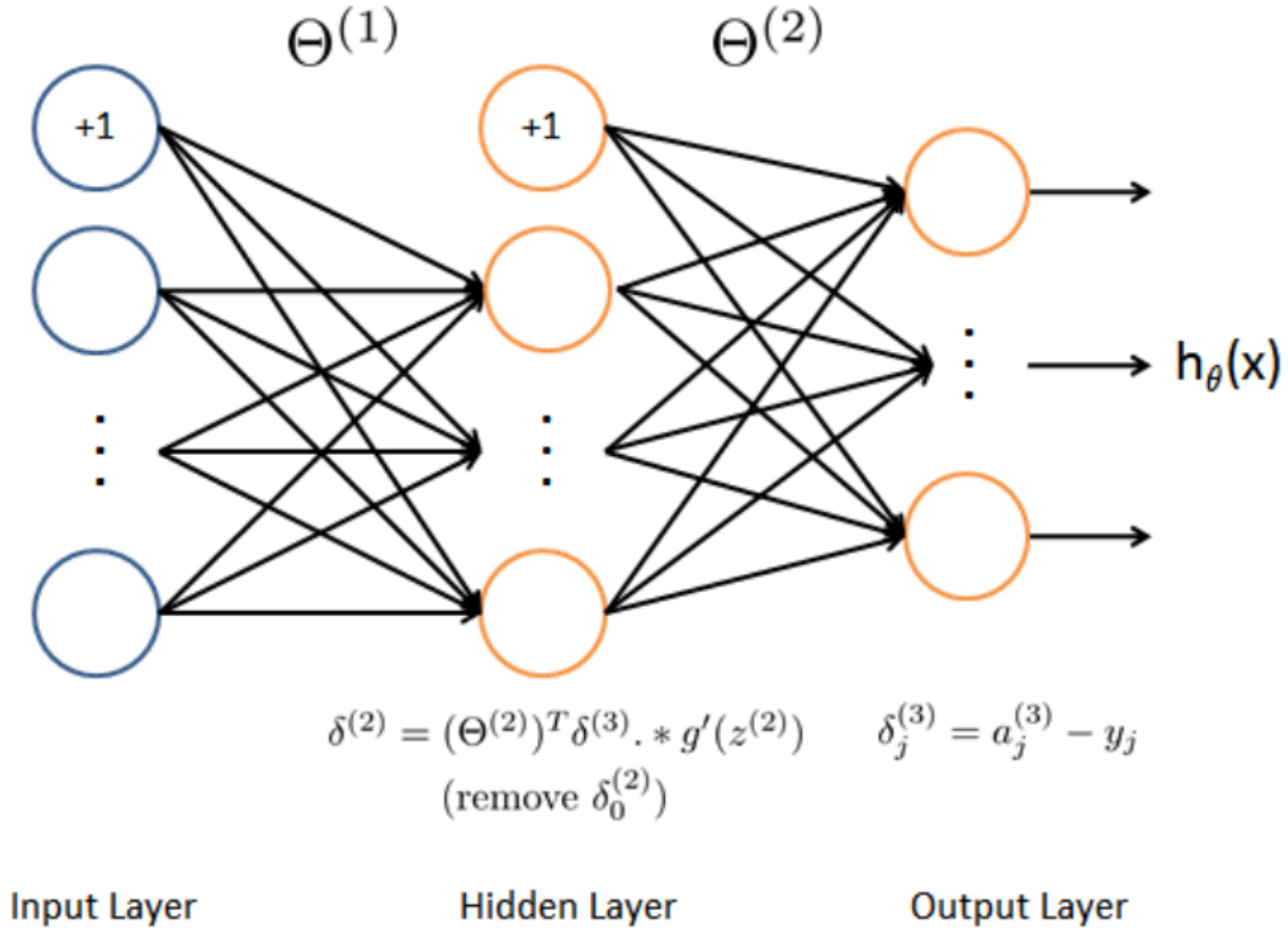
```
cost_regularized(theta1, theta2,  X, y_onehot, reg_lambda=1)
```

Out[20]:

```
5.926491951889327
```

3. Backpropagation

Now, you will implement the backpropagation algorithm. Recall that the intuition behind the backpropagation algorithm is as follows. Given a training example $(x^{(t)}, y^{(t)})$, we will first run a “forward pass” to compute all the activations throughout the network, including the output value of the hypothesis $h_\Theta(x)$. Then, for each node j in layer l, we would like to compute an “error term” $\delta_j^{(l)}$ that measures how much that node was “responsible” for any errors in our output. For an output node, we can directly measure the difference between the network’s activation and the true target value, and use that to define $\delta_j^{(3)}$ (since layer 3 is the output layer). For the hidden units, you will compute $\delta_j^{(l)}$ based on a weighted average of the error terms of the nodes in layer $(l+1)$. In detail, here is the backpropagation algorithm depicted in figure below. You should implement steps 1 to 4 in a loop that processes one example at a time. Concretely, you should implement a for-loop for $t = 1:m$ and place steps 1-4 below inside the for-loop, with the t_{th} iteration performing the calculation on the t_{th} training example $(x^{(t)}, y^{(t)})$. Step 5 will divide the accumulated gradients by m to obtain the gradients for the neural network cost function.



First, let's implment the derivitive function of our $h_\Theta(x)$, i.e. the derivitive of the sigmoid function

In [21]:

```
# implement the derivitive of sigmoid function below
def sigmoid_gradient(z):
    return ...
```

Prepare an initial randomized set of theta vector within range [-0.25, 0.25]

In [22]:

```
# the theta parameter vector
params = (np.random.random(size=hidden_size * (input_size + 1) + num_labels * (hidden_size + 1)) -

# the theta matrix derived from params
#theta1 = np.matrix(np.reshape(params[:hidden_size * (input_size + 1)], (hidden_size, (input_size +
#theta2 = np.matrix(np.reshape(params[hidden_size * (input_size + 1):], (num_labels, (hidden_size +
```

3.1 steps to implement backprop algorithm (unregularized)

For each data samle $(x^{(t)}, y^{(t)})$ in the dataset $((x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(m)}, y^{(m)}))$, put the following steps in a for-loop from $t = 1 : m$:

1. Set the input layer's values $(a^{(1)})$ to the t^{th} training example $x^{(t)}$. Perform a feedforward pass, computing the activations $(z^{(2)}, a^{(2)}, z^{(3)}, a^{(3)})$ for layers 2 and 3. Note that you need to add a +1 term to ensure that the vectors of activations for layers $a^{(1)}$ and $a^{(2)}$ also include the bias unit.
2. For each output unit k in layer 3 (the output layer), set $\delta_k^{(3)} = (a_k^{(3)} - y_k)$ (think about why by following the equations given in the lecture notes), where $y_k \in \{0, 1\}$ indicates whether the current training example belongs to class $k(y_k = 1)$, or if it belongs to a different class $(y_k = 0)$.
3. For the hidden layer $l = 2$, set $\delta^{(2)} = (\Theta^{(2)})^T \delta^{(3)} .* g'(z^{(2)})$, where $.*$ means element-wise multiplication between two vectors of the same size.
4. Calculate $\frac{\partial J^x(\Theta)}{\partial \Theta_{ij}^{(x,l)}} = a_j^{(x,l)} \delta_i^{(x,l+1)}$, accumulate the gradient of each θ by summation of the results from all m data samples.

Finally, out of the for loop, obtain the (unregularized) gradient for the neural network cost function by dividing the accumulated gradients by m, $\frac{\partial J(\Theta)}{\partial \Theta_{ij}^{(l)}} = \frac{1}{m} \Sigma_x a_j^{(x,l)} \delta_i^{(x,l+1)}$

In [23]:

```
def backprop(params, X, y):
    m =
    X =
    y =

    # reshape the parameter array into parameter matrices for each layer
    theta1 =
    theta2 =

    # run the feed-forward pass
    a1, z2, a2, z3, h = forward_propagate(X, theta1, theta2)

    # initializations
    J =
    delta1 =   # (25, 401)
    delta2 =   # (10, 26)

    # compute the cost
    for i in range(m):
        ...




    # perform backpropagation
    for t in range(m):
          ...


    grad =

    return J, grad
```

When implementing the back propagation funtion, you may need a lot of vector multiplication or element-wise vector multiplication, pay attetion to the difference and always check whether the result has the correct shape as you expected.

In [24]:

```
J, grad = backprop(params, X, y_onehot)
J, grad.shape
```

Out[24]:

```
(6.769300511302108, (10285,))
```

You may not get exactly the same cost as above, it is normal, why?

Explain below:

3.2 Gradient Checking (optional, 20 additional points)

In your neural network, you are minimizing the cost function $J(\Theta)$. To perform gradient checking on your parameters, you can imagine "unrolling" the parameters $\Theta^{(1)}$, $\Theta^{(2)}$ into a long vector θ. By doing so, you can think of the cost function being $J(\theta)$ instead and use the following gradient checking procedure.

Suppose you have a function $f_i(\theta)$ that compute the gradient, you'd like to check if f_i is outputting correct derivative values.

$$\text{Let} \quad \theta^{(i+)} = \theta + \begin{bmatrix} 0 \\ 0 \\ \vdots \\ \epsilon \\ \vdots \\ 0 \end{bmatrix} \qquad \text{and} \quad \theta^{(i-)} = \theta - \begin{bmatrix} 0 \\ 0 \\ \vdots \\ \epsilon \\ \vdots \\ 0 \end{bmatrix}$$

So, $\theta^{(i+)}$ is the same as θ, except its i-th element has been incremented by ϵ. Similarly, $\theta^{(i-)}$ is the corresponding vector with the i-th element decreased by ϵ. You can now numerically verify $f_i(\theta)$'s correctness by checking, for each i, that:

$$f_i(\theta) \approx \frac{J(\theta^{(i+)}) - J(\theta^{(i-)})}{2\epsilon}.$$

The degree to which these two values should approximate each other will depend on the details of J. But assuming $\epsilon = 10^{-4}$, you'll usually find that the left- and right- hand sides of the above will agree to at least 4 significant digits (and often many more).

When performing gradient checking, it is much more efficient to use a small neural network with a relatively small number of input units and hidden units, thus having a relatively small number of parameters.

In [25]:

```
def expand_array(arr):
    """replicate array into matrix
    [1, 2, 3]

    [[1, 2, 3],
     [1, 2, 3],
     [1, 2, 3]]
    """
    # turn matrix back to ndarray
    return np.array(np.matrix(np.ones(arr.shape[0])).T @ np.matrix(arr))
```

In [26]:

```
def gradient_checking(params, X, y, epsilon, regularized=False):


    for i in range(len(params)):
        numeric_grad[i] = ...



    # analytical grad will depend on if you want it to be regularized or not
    _, analytic_grad = backprop_reg(params, X, y) if regularized else backprop(params, X, y)

    # If you have a correct implementation, and assuming you used EPSILON = 0.0001
    # the diff below should be less than 1e-9
    # this is how original matlab code do gradient checking
    diff = np.linalg.norm(numeric_grad - analytic_grad) / np.linalg.norm(numeric_grad + analytic_gr

    print('If your backpropagation implementation is correct,\nthe relative difference will be small
```

Running the gradient checking function may take several minutes, why? Please write your answer below:

Because: ...

In [ ]:

```
#gradient_checking(params, X, y_onehot, epsilon= 0.0001)#very slow, around 10 minutes
```

3.3 Regularized version of the back propagation function

Now we add the regularization terms to the gradients as stated in the lecture notes.

In [27]:

```
def backprop_reg(params, X, y, reg_lambda=1):
    m =
    X =
    y =

    # reshape the parameter array into parameter matrices for each layer
    theta1 =
    theta2 =

    # run the feed-forward pass
    a1, z2, a2, z3, h = forward_propagate(X, theta1, theta2)

    # initializations


    # compute the cost
    for i in range(m):


    # add the cost regularization term
    J +=

    # perform backpropagation
    for t in range(m):




    # add the gradient regularization term




    return J, grad
```

In [28]:

```
J, grad = backprop_reg(params, X, y_onehot, 1)
J, grad.shape
```

Out[28]:

```
(6.774565460565762, (10285,))
```

In [ ]:

```
# do the gradient checking for regularized cost functions
#gradient_checking(params, X, y_onehot, epsilon= 0.0001,regularized=True)#very slow, around 10 minu
```

4. Training the network

If everything goes well, it means you have correctly implemented the cost function and backpropagation algorithm to calculate gradients in each iteration, now we are ready to train the network. We can use an optimization function directly from scipy.optimize package, like what we do in logistic regress exercise.

You can try to train the network with both backpropagation with/without regularization and see the differences.

In [30]:

```
from scipy.optimize import minimize
#params = np.concatenate((np.ravel(theta1), np.ravel(theta2)))
# minimize the objective function
fmin = minimize(fun=backprop, x0=params, args=(X, y_onehot),
                method='TNC', jac=True, options={'maxiter': 250})
fmin
```

Out[30]:

```
     fun: 0.009389561321040565
     jac: array([3.17539791e-04, 0.00000000e+00, 2.20000358e-12, ...,
       7.12793225e-06, 3.61363783e-06, 3.45457304e-06])
 message: 'Max. number of function evaluations reached'
    nfev: 250
     nit: 24
  status: 3
 success: False
       x: array([-1.85590046,  0.11936094, -0.11755924, ..., -2.41352382,
       -1.55942958, -0.79665084])
```

Sometimes, the "success" tag may have value "False" instead of "True", which indicates the optimization doesn't converge within the limited iteration numbers. This does not mean the result cannot be used. For this application and dataset, if your optimization returns error value in "fun" less than 0.5, you can already get pretty good results in the digit classification task.

Now let's use the optimized parameters to predict hand writing digits in our dataset.

In [31]:

```
X =
theta1 =
theta2 =

a1, z2, a2, z3, h = forward_propagate(X, theta1, theta2)
y_pred = np.array(np.argmax(h, axis=1) + 1)
y_pred
```

Out[31]:

```
array([[10],
       [10],
       [10],
       ...,
       [ 9],
       [ 9],
       [ 9]], dtype=int64)
```

Check the accuracy of your network in prediction

In [32]:

```
correct = [1 if a == b else 0 for (a, b) in zip(y_pred, y)]
accuracy = (sum(map(int, correct)) / float(len(correct)))
print ('accuracy = {0}%'.format(accuracy * 100))
```

```
accuracy = 100.0%
```

You don't necessarily get 100% because the optimization result may be different depending on the initial value and optimization method used. But the whole pipeline, if implemented correctly, should achieve an accuracy above 95%.

In [ ]:

```

```