Assignment 7: Handwritten Digit Recognition using Neural Networks

In this assignment, you will complete the code of backpropagation algorithm for neural networks, and apply it to the problem of handwritten digit recognition.

The following code files are used in this assignment. The files marked with * are those which you should edit to complete:

- **main.py**: The main process of the code which uses other files to execute the steps
- **displayData.py**: Function to help visualize the dataset
- **computeNumericalGradient.py**: Numerically computes the gradients for comparison
- **debugInitializeWeights.py**: Function which initializes the weights for comparison
- **checkNNGradients.py**: Function to help check your computed gradients with the numerical ones
- **randInitializeWeights.py**: Randomly initializes the weights in a predefined range
- **sigmoid.py**: The sigmoid function
- *** sigmoidGradient.py**: Computes the gradient of the sigmoid function
- *** predict.py**: Function used to predict new digits based on the trained neural network
- *** nnCostFunction.py**: Computes the neural network cost function and its gradient

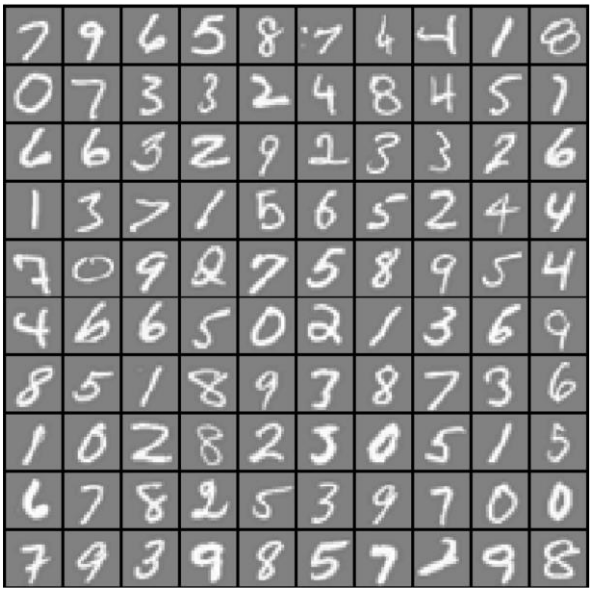
Also the following data files are used in the assignment:

- **dataset.csv**: Training dataset of handwritten digits' pixel values
- **sampleTheta1.csv**: Sample values for neural network first layer weights, used for test purposes
- **sampleTheta2.csv**: Sample values for neural network second layer weights, used for test purposes

Now you should follow the steps below to complete the assignment. It is recommended that you first go over all the explanations and also skim through the **main.py** contents to have a good overview of the entire assignment in mind, then start implementing the required parts in the incomplete code files.

1) Training Data:

The **main.py** file first loads the data and attempts to visualize some of them using the **displayData** function:



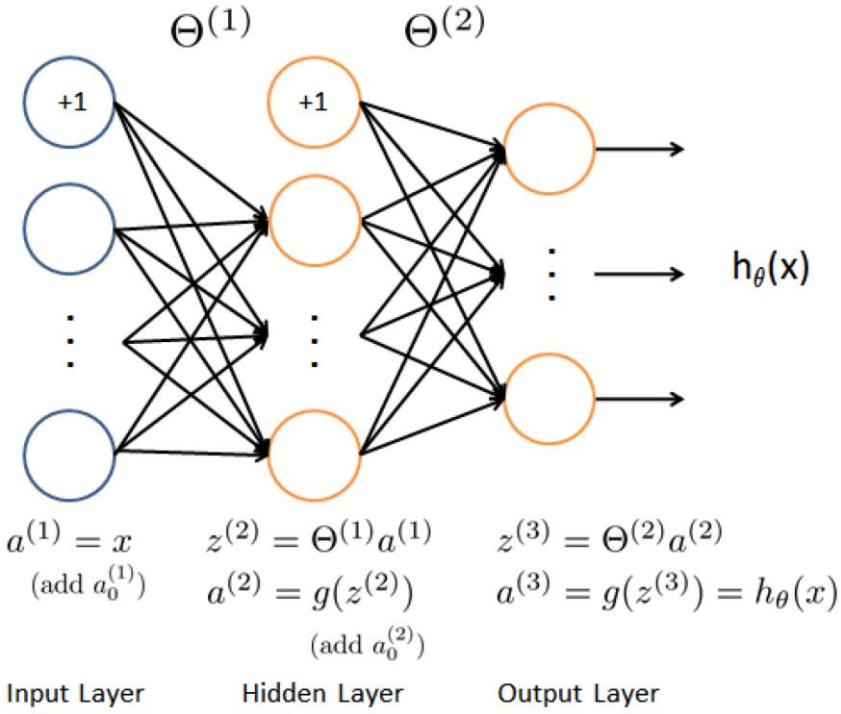
The dataset contains 5000 training examples, where each training example is the 20 × 20 pixels grayscale image of a digit, which is unrolled into a 400-dimensional vector. This gives us a 5000 × 400 matrix **X** for the training data.

$$X = \begin{bmatrix} — \ (x^{(1)})^T \ — \\ — \ (x^{(2)})^T \ — \\ \vdots \\ — \ (x^{(m)})^T \ — \end{bmatrix}$$

The last column in the training data contains the labels for training examples which shows their actual digit values. So **y** is a 5000-dimensional vector of the digit labels. Note that the value 10 is used in the data for the digit 0.

2) Neural Network Structure:

The neural network used in this assignment has three layers as shown in the figure below: the input layer, the hidden layer, and the output layer.



Because each input is the 400-dimensional pixel values of the digit image, the first layer has 400 main units. Also 25 units are used for the hidden layer.

Note that in the input and hidden layers, an extra **bias** unit with value 1 is introduced, which makes the actual input layer of size 401 and the hidden layer of size 26.

The size of the output layer depends on the number of classes to choose from, which is 10 here for the 10 different digits to choose from. For example if the digit is 3, the third unit in the output layer will be 1 and all other units will be 0, like below:

$$y = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

The matrix $\theta^{(1)}$ includes the weights for transitioning from the input layer to the hidden layer, so it is of shape 25×401. Similarly, $\theta^{(2)}$ is used as weights of moving from the hidden layer to the output layer and has a 10×26 size.

3) Cost Function:

Now you should implement the cost function for the neural network, which can be computed using this formula:

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

where $h_\theta(x^{(i)})$ is the final result of the output layer and can be computed using feed-forward formula based on the outputs of the previous layers:

$$\begin{array}{lll} a^{(1)} = x & z^{(2)} = \Theta^{(1)}a^{(1)} & z^{(3)} = \Theta^{(2)}a^{(2)} \\ (\text{add } a_0^{(1)}) & a^{(2)} = g(z^{(2)}) & a^{(3)} = g(z^{(3)}) = h_\theta(x) \\ & (\text{add } a_0^{(2)}) & \\ \text{Input Layer} & \text{Hidden Layer} & \text{Output Layer} \end{array}$$

Also $K = 10$ is the number of possible labels, and $h_\theta(x^{(i)})_k$ is the output value of the k-th output unit. For example, if the recognized digit is 3, $h_\theta(x^{(i)})_3$ is 1, while for all other values of k it is 0. The same interpretation holds for y_k for the actual digits.

For computing the cost, you should use the formula above to compute $h_\theta(x^{(i)})$ and the cost for every training example i, and then sum over all the training examples.

You should write the code for computing the cost function in nnCostFunction.py. Note that your code should work with any dataset and network layer sizes, but the number of layers can be considered fixed.

Then you can test it by running parts 2 and 3 in **main.py**, which tests the cost function using sample weights Theta1 and Theta2 loaded from **sampleTheta1** and **sampleTheta2** data files. The result cost should be about **0.287629**.

4) Regularized Cost Function:

The cost function for the regularized neural network 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 exclude the bias terms used in the input and hidden layers from the regularization term.

You should now add regularization to your cost function in nnCostFunction.py. Although the actual values of the indices of $\theta^{(1)}$ and $\theta^{(2)}$ are used in the formula above, your code should work with weights of any size.

Part 4 of **main.py** tests the regularized cost function with sample weights, which should give a result of about **0.383770**.

5) Sigmoid Gradient:

Before computing the gradients of the cost function, you need to implement a utility function for computing the gradient of a sigmoid function using the formula below:

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

$$g'(z) = \frac{d}{dz} g(z) = g(z)(1 - g(z))$$

You should implement the sigmoid gradient function in sigmoidGradient.py, in a way that it can work with vectors or matrices as input.

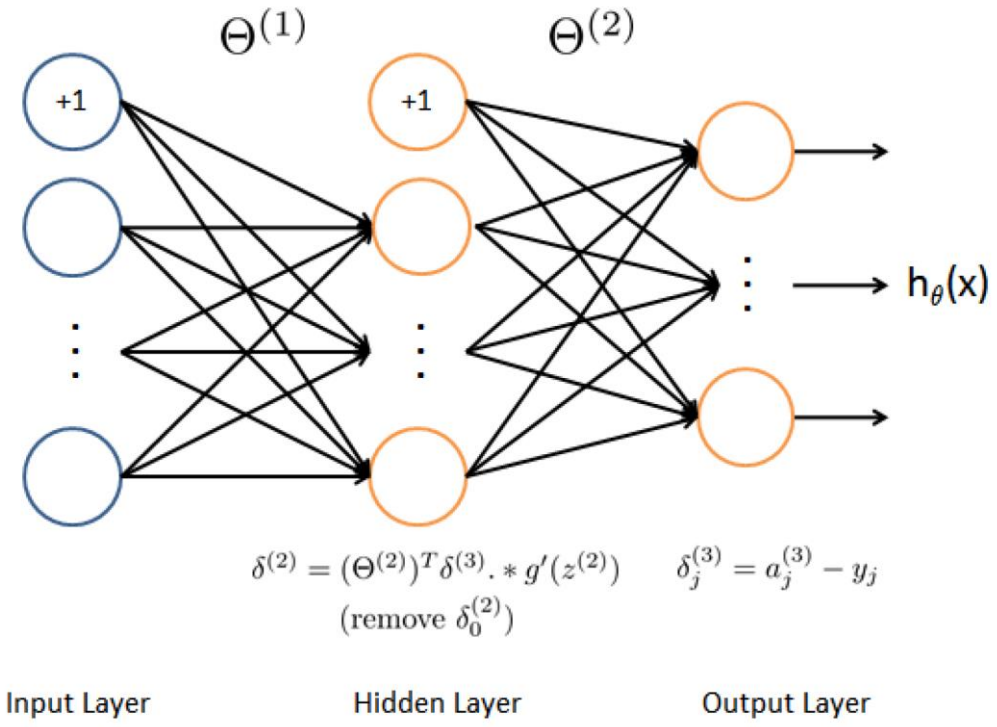
For large positive or large negative values, the sigmoid gradient should be close to zero, while for zero input the result should be 0.25, as checked in part 5 of **main.py**.

6) Backpropagation and Cost Function Gradients:

Now you should implement the backpropagation algorithm for computing the gradients of the cost function, which is the most important part of your code.

The intuition behind backpropagation is as follows: Given any training example $(x^{(t)}.y^{(t)})$, we first run a feed-forward pass to compute all the activations throughout the network, including the final output values of $h_\theta(x)$. Then for each node j in layer l, we should compute an error term $\delta_j^{(l)}$ which measures how much that node was responsible for any errors in the output.

These error terms should be computed recursively starting from the output layer to the input layer using the formulas in the figure below, because the error in layer l is computed based on a weighted average of the error terms of the nodes in layer $l + 1$. Finally we can use these error terms to compute the gradients in each layer.



Here are the detailed steps of backpropagation, which should be executed separately for all the training examples. So you should implement a loop over the list of training examples and implement the steps 1 to 4 inside the loop, before the final step 5:

1. Set the input layer values $a^{(1)}$ to the t-th training example $x^{(t)}$. Perform a feed-forward pass like section 3 for computing the activations ($z^{(2)}$, $a^{(2)}$, $z^{(3)}$, $a^{(3)}$) for layers 2 and 3. Make sure the bias terms are added to the vectors for $a^{(1)}$ and $a^{(2)}$.

2. For each output unit k in layer 3 (the output layer), set: $\delta_k^{(3)} = (a_k^{(3)} - y_k)$, where y_k indicates whether the current training example belongs to class k (i.e. $y_k = 1$) or belongs to another class (i.e. $y_k = 0$).

3. For the hidden layer $l = 2$, set: $\delta^{(2)} = \left(\theta^{(2)}\right)^T \delta^{(3)} .* g'(z^{(2)})$

4. Accumulate the gradient from this training example using this formula for $l = 1$ and $l = 2$: $\Delta^{(l)} = \Delta^{(l)} + \delta^{(l+1)}\left(a^{(l)}\right)^T$. Note that you should skip or remove $\delta_0^{(2)}$.

5. Finally, after the loop over training examples finishes, compute the gradients for the neural network cost function by dividing the accumulated gradients by $\frac{1}{m}$ which means:

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = D_{ij}^{(l)} = \frac{1}{m} \Delta_{ij}^{(l)}$$

You should add the backpropagation code to compute the gradients in nnCostFunction.py.

A good practice is to initialize the weight parameters θ to small random values in a specific range. This is what the **randInitializeWeights** function does in part 6 of **main.py**.

In part 7 of **main.py**, the function **checkNNGradients** is used to test the backpropagation code, and check the parameters and gradients against a numerical estimation of these parameters. The difference should be less than $1e^{-9}$.

7) Regularized Backpropagation:

In this step, you can add the extra gradient regularization term to the final gradient values computed in the previous step. After you have computed $\Delta_{ij}^{(l)}$ using backpropagation, you should add regularization in this way:

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = D_{ij}^{(l)} = \frac{1}{m}\Delta_{ij}^{(l)} + \frac{\lambda}{m}\Theta_{ij}^{(l)} \qquad \text{for } j \geq 1$$

Note that you should not be regularizing the first column of $\theta^{(l)}$ which is used for the bias term.

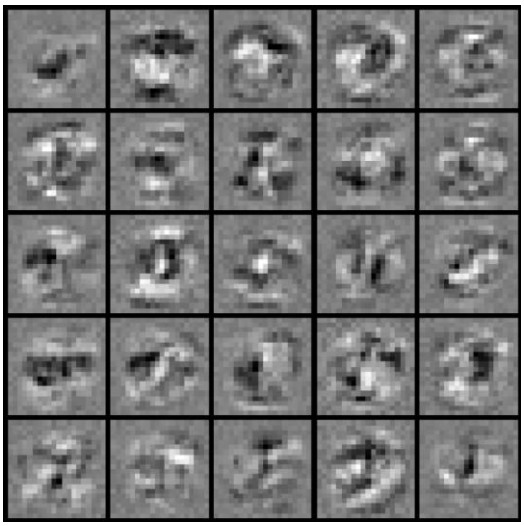
Now modify your code in nnCostFunction.py to handle regularization. Part 8 of **main.py** runs **checkNNGradients** to test the regularized backpropagation. Also the cost using fixed debugging parameters should be about **0.576051**.

8) Training the Neural Network:

In this stage, the optimization method **fmincg**, which is somehow similar to gradient descent but better for large number of parameters, can be used to learn a good set of parameters using the cost and gradient values provided by **nnCostFunction**.

Part 9 of **main.py** actually trains the model using fmincg and computes the best values for weight parameters Theta1 and Theta2.

Part 10 displays a visualization of some Theta1 values, which shows how neural networks learn important features of digit images in each layer. Because each row in Theta1 is a 400-dimensional vector (excluding the bias terms), each row can be interpreted as a transformed 20×20 image. Some of these images can be seen in the figure below, illustrating how the neural network is learning patterns and strokes in the training images:



9) Prediction using the Neural Network:

Now it is time to test the trained neural network on the training examples to find the training accuracy of the model.

The output values for $h_\theta(x)$ should be computed using feed-forward, having a new input image x and the trained weight parameters. The final predicted digit should be decoded by checking which output value is equal to 1.

You should now complete the code in predict.py for this purpose. Part 11 of **main.py** code predicts the digits for training images and computes the accuracy. If your code is correct, the accuracy should be about **96%**, which can vary by about 1% to 2% as a result of random initialization.