Homework 6

Deadline: 2018.06.30 (Tuesday) 23:59

You can EITHER create your own hw6.ipynb to write the code (meet the requirements) [BONUS] from the scratch OR use the hw6.ipynb we provide to complete the required parts.

Learning a 3-layer Neural Network [hw6.ipynb]

Happy to see you still here to complete the last homework. ☺ In this homework, you will implement the backpropagation algorithm for learning neural networks and apply it to the task of hand-written digit recognition (手寫數字辨識). We have provided you two Matlab data files. One is **hw6_data.mat**, and the other is **hw6_weights.mat**. In addition, we also provide you a sample ipynb file, in which we have input some codes and you are asked to complete the remaining and required parts. In other words, throughout the homework, you will be using hw6.ipynb, in which we have loaded the dataset for the following problems and make calls to functions that you will write.

1. The Neural Network

1.1 Visualize the Dataset [No code needs to be written in this part]

First, the code will load the data and display it on a 2-dimensional plot, as shown in Figure 1 by calling the function `displayData`. There are 5000 training examples in **hw6_data.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. The second part of the training set is a 5000-dimensional vector y that contains labels for the training set. We have mapped the digit zero to the value ten. Therefore, a "0" digit is labeled as "10", while the digits "1" to "9" are labeled as "1" to "9" in their natural order.

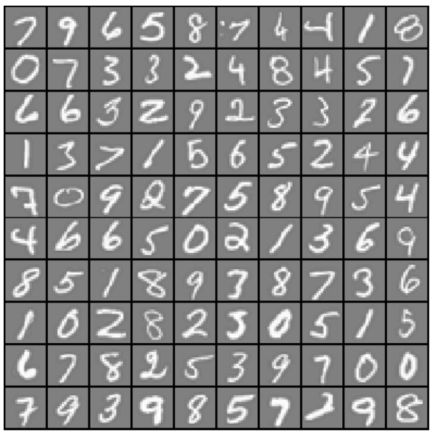

Figure 1: Examples from the dataset.

1.2 Network Representation

[No code needs to be written in this part]

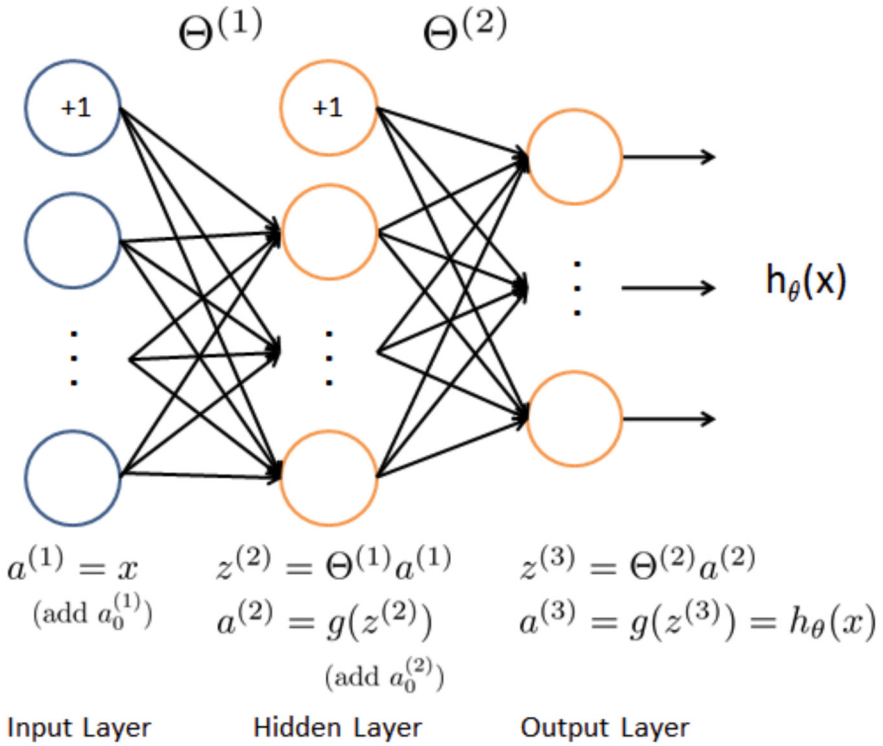

Figure 2: Neural network model.

Our neural network is shown in Figure 2. It has 3 layers: an input layer, a hidden layer and an output layer. Recall that our inputs are pixel values of digital 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 training data will be loaded into the variables X and y by **hw6.ipynb**. You have been provided with a set of network parameters ($\Theta^{(1)}$ and $\Theta^{(2)}$) already trained by some method. These are stored in **hw6_weights.mat** and will be loaded by **hw6.ipynb** into matrices of variables `theta1` and `theta2`. The parameters have dimensions that are sized for a neural network with 25 units in the second layer and 10 output units (corresponding to the 10 digit classes). In other words, `theta1` has size 25 x 401 while `theta2` has size 10 x 26.

1.3 Forward Propagation and Cost Function

Now you will implement the cost function and gradient for the neural network. You are asked to complete two functions, `forward_propagate` and `cost`. The function `forward_propagate` accepts X, `theta1` and `theta2` as inputs, then computes and returns all generated elements of forward propagation, including `a1`, `z2`, `a2`, `z3`, and `h`, where `a1` is X with the bias unit, and `h` is an output vector (5000x10). The matrix dimensionality of `a1`, `z2`, `a2`, `z3`, and `h` are given in the code. The function `cost` accepts as `y`, `h`, `theta1`, `theta2`, `learning_rate` as inputs, then returns the computed regularized cost J of neural network. It is important to notice that you should add the bias term to `a1` and `a2`. For example, you may want to use this code statement for `a1`:

```
a1 = np.insert(X, 0, values=np.ones(X.shape[0]), axis=1)
```

Recall that the cost function for the neural network with regularization is given by:

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

where $h_\theta(x^{(i)})$ is computed as shown in the Figure 2 and $K = 10$ is the total number of possible labels. Note that $h_\theta(x^{(i)})_k$ is the activation (output value) of the k-th output unit. Also, recall that whereas the original labels (in the variable y) where k=1, 2, ..., 10.

You can assume that the neural network will only have 3 layers: an input layer, a hidden layer and an output layer. However, your code should work for any number of input units, hidden units and outputs units. While we have explicitly listed the indices above for $\Theta^{(1)}$ and $\Theta^{(2)}$ for clarity, do note that your code should in general work with $\Theta^{(1)}$ and $\Theta^{(2)}$ of any size.

For the purpose of training a neural network, we need to recode the labels as vectors containing only values 0 or 1, so that

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

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. We have already provided you some codes to help you do such transformation using `OneHotEncoder` in `sklearn`.

You need to implement the forward propagation that computes $h_\theta(x^{(i)})$ for every example i and sum the cost over all examples. Your code should 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). Note that you should not be regularizing the terms that correspond to the bias. For the matrices `theta1` and `theta2`, the bias corresponds to the first column of each matrix. You should add regularization to your functions.

Once you have completed functions `forward_propagate` and `cost`, the ipynb file will call the function `forward_and_cost` that further calls `forward_propagate` and `cost`. By using the default `theta1` and `theta2` loaded from **hw6_weights.mat**, you should see the cost is about

`0.287629` when `learning_rate=0` (without regularization) and is about `0.383770` when `learning_rate=0` (with regularization).

2. BackPropagation

In the second part, you will implement the backpropagation algorithm to compute the gradient for the neural network cost function. You will need to complete the `gradient` function so that it returns an appropriate value for `grad` (gradients). Once you have computed the gradient, you will be able to train the neural network by minimizing the cost function $J(\Theta)$ using an advanced optimizer such as the TNC method of `minimize` in `scipy.optimize`. That is, you will implement the backpropagation algorithm to compute the gradients for the parameters for the regularized neural network.

2.1 Sigmoid Gradient

[No code needs to be done in this part]

To help you get started with this part of the part, we have provided you the sigmoid gradient function. The gradient for the sigmoid function can be computed as:

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

where $sigmoid(z) = g(z) = \frac{1}{1+e^{-z}}$. In the function `sigmoid_gradient`, it is a vectorization implementation of the sigmoid gradient. Note that you will need to use this `sigmoid_gradient` function in the implementation of the BackPropagation algorithm.

2.2 Random Initialization

[No code needs to be done in this part]

When training neural networks, it is important to randomly initialize the parameters. One effective strategy for random initialization is to randomly select values for $\Theta^{(l)}$ uniformly in the range $[-\epsilon, \epsilon]$. Here we choose to set $\epsilon = 0.12$. This range of values ensures that the parameters are kept small and makes the learning more efficient. We have completed the random initialization for you.

2.3 BackPropagation Algorithm

[No code needs to be done in this part]

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 details, here is the backpropagation algorithm (also depicted in Figure 3). 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 in range(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. Note that m is the number of training examples.

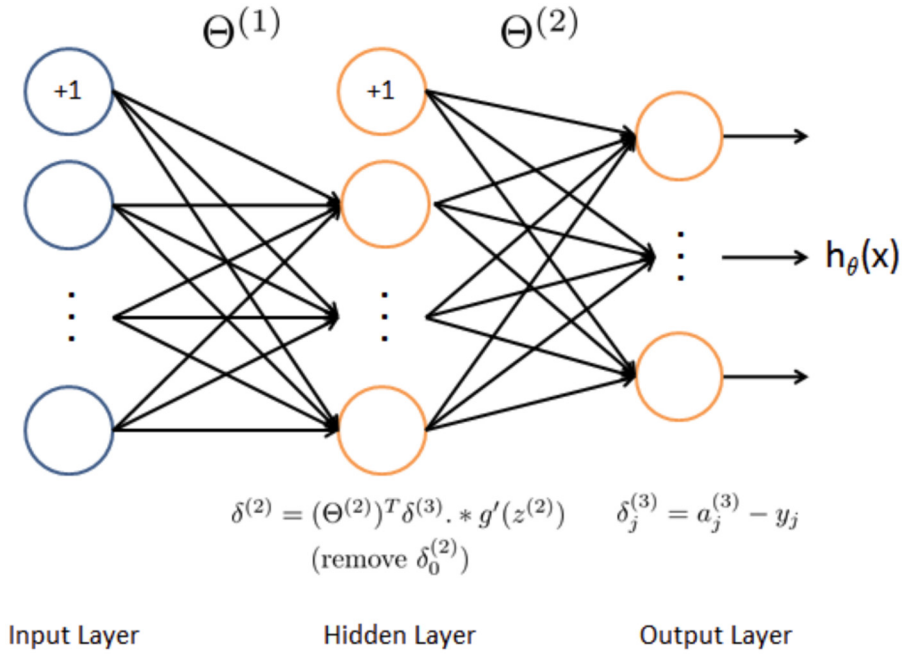

Figure 3: Backpropagation Updates.

Step 1 Set the input layer's values $a^{(1)}$ to the t-th training example $x^{(t)}$. Then you should perform a feedforward pass (as shown in Figure 2) to 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. It is important to notice that although you may already adding the bias terms in the implementation of `forward_propagate` and `cost` functions in Problem 1.3, you may still need to add the bias term somewhere in the implementation of the `gradient` function

Step 2 For each output unit k in layer 3 (the output layer), set

$$\delta_k^{(3)} = \left(a_k^{(3)} - y_k\right)$$

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 di erent class $(y_k = 0)$.

Step 3 For the hidden layer $l = 2$, set

$$\delta^{(2)} = \left(\Theta^{(2)}\right)^T \delta^{(3)} \otimes g'\left(z^{(2)}\right)$$

Step 4 Accumulate the gradient from this example using the following formula. Note that sometimes you should skip or remove $\delta_0^{(2)}$ using **slicing** (e.g. d=d[:,1:] to remove the first column) when computing the gradients.

$$\Delta^{(l)} = \Delta^{(l)} + \delta^{(l+1)}\left(a^{(l)}\right)^T$$

Step 5 Obtain the regularized gradient for the neural network cost function by dividing the accumulated gradients by $\frac{1}{m}$:

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

Note that you should not be regularizing the first column of $\Theta^{(l)}$ which is used for the bias term. Furthermore, in the parameters $\Theta_{ij}^{(l)}$, i is indexed starting from 1, and j is indexed starting from 0. Thus, you can image $\Theta^{(l)}$ is in the following form:

$$\Theta^{(l)} = \begin{bmatrix} \Theta_{1,0}^{(l)} & \Theta_{1,1}^{(l)} & \cdots \\ \Theta_{2,0}^{(l)} & \Theta_{2,1}^{(l)} & \cdots \\ \vdots & \vdots & \ddots \end{bmatrix}.$$

Here is a tip that would be very helpful for you. You should implement the backpropagation algorithm only after you have successfully completed the `forward_propagate` and `cost` functions. While implementing the backpropagation algorithm, it is often useful to use the `shape` function to print out the sizes of the variables you are working with if you run into dimension mismatch errors.

2.4 Learning Parameters Using `scipy.optimize` [No code needs to be done in this part]

After you successfully implement the neural network cost function and gradient computation, the next step of the ex4.m script will use **`minimize`** in **`scipy.optimize`** to learn a good set parameters.

After the training completes, hw6.ipynb will proceed to report the training accuracy of your classifier by computing the percentage of examples it got correct. If your implementation is correct, you should see a reported training accuracy of about 98% (this may vary by about 1% due to the random initialization). It is possible to get higher training accuracies by training the neural network for more iterations. We encourage you to try training the neural network for more iterations (e.g., set the `minimize` function's `maxiter` to 500) and also vary the regularization parameter λ. With the right learning settings, it is possible to get the neural network to perfectly fit the training set.

3. Visualize the Hidden Layer

[No code needs to be done in this part]

One way to understand what your neural network is learning is to visualize what kinds of the representations captured by the hidden units. Informally, given a particular hidden unit, one way to visualize what it computes is to find an input x that will cause it to activate (that is, to have an activation value $a_i^{(l)}$ close to 1). For the neural network you trained, notice that the i-th row of $\Theta^{(1)}$ is a 401-dimensional vector that represents the parameter for the i-th hidden unit. If we discard the bias term, we get a 400 dimensional vector that represents the weights from each input pixel to the hidden unit.

Thus, one way to visualize the "representation" captured by the hidden unit is to reshape this 400 dimensional vector into a 20 x 20 image and display it. The next step of hw6.ipynb does this by using the `displayData` function and it will show you an image (similar to Figure 4) with 25 units, each corresponding to one hidden unit in the network. In your trained network, you should find that the hidden units corresponds roughly to detectors that look for strokes and other patterns in the input.

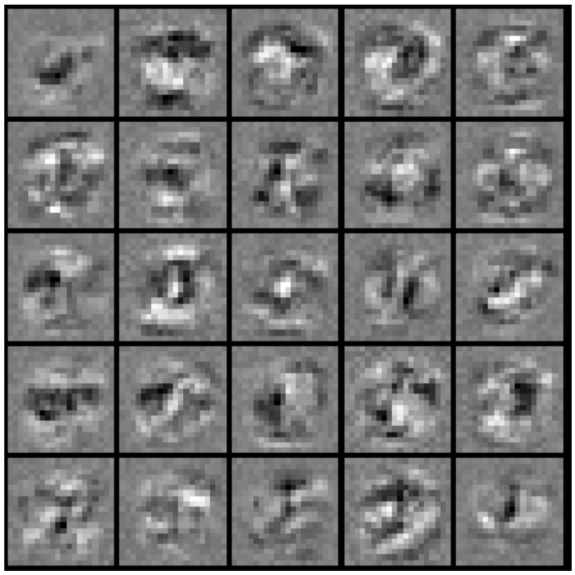

Figure 4: Visualization of Hidden Units.

Note that in your codes for these problems,
you need to write some comments to describe the meaning of each part.

How to Submit Your Homework?

Submission in NCKU Moodle. Before submitting your homework, please zip the files (**hw6.ipynb**, **hw6_weights.mat**, and **hw6_data.mat**) in a zip file, and name the file as "學號_hw6.zip". For example, if your 學號 of your team are H12345678, then your file name is:

"H12345678_hw**6**.zip" or "H12345678_hw**6**.rar"

When you zip your files, please follow the instructions provided by TA's slides to submit your file using NCKU Moodle platform http://moodle.ncku.edu.tw .

Have Questions about This Homework?

Please feel free to visit TAs, and ask/discuss any questions in their office hours. We will be more than happy to help you.