Programming Exercise 6: Neural Networks Learning (Back Propagation)

Introduction

In this exercise, you will implement the backpropagation algorithm for neural networks and apply it to the task of hand-written digit recognition. To get started with the exercise, you will need to download the starter code and unzip its contents to the directory where you wish to complete the exercise. If needed, use the `cd` command in Octave/MATLAB to change to this directory before starting this exercise.

Files included in this exercise

`ex6.m` – Octave/MATLAB script that steps you through the exercise
`ex6data1.mat` – Training set of hand-written digits
`ex6weights.mat` – Pre-calculated weights for the neural network (used for debugging)
`displayData.m` – Function to help visualize the dataset
`fmincg.m` – Function minimization routine
`sigmoid.m` – Sigmoid function
`computeNumericalGradient.m` – Function to compute gradients numerically (used for debugging)
`checkNNGradients.m` – Function to help check your gradients (used for debugging)
`debugInitializeWeights.m` – Function for initializing weights (used for debugging)
`predict.m` – Neural network prediction function
`randInitializeWeights.m` – Randomly initialize weights
[*] `nnCostFunction.m` – Neural network cost function and gradients

For this exercise, you will use neural networks to recognize handwritten digits (from 0 to 9). Automated handwritten digit recognition is widely used today – from recognizing zip codes (postal codes) on mail envelopes to recognizing amounts written on bank checks. This exercise will show you how the methods you've learned can be used for this classification task.
Throughout the exercise, you will be using the script `ex6.m`. This script sets up the dataset for the problem and makes call to the function that you will write. You do not need to modify this script. You are only required to modify a function in some other file, by following the instructions in this assignment.

[*] indicates files you will need to complete

1 Neural Networks

In the previous exercise, you implemented forward propagation for neural networks and used it to predict handwritten digits with the pre-trained weights provided. In this exercise, you will implement the backpropagation algorithm to learn the parameters for the neural network. The provided script, `ex6.m`, will help you step through this exercise.

1.1 Visualizing the data

In the first part of `ex6.m`, the code will load the data. This is the same dataset that you used in the previous exercise[1]. There are 5000 training examples in `ex6data1.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.

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

The second part of the training set is a 5000-dimensional vector y that contains labels for the training set. To make things more compatible with Octave/MATLAB indexing, where there is no zero index, 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.
In Part 1 of `ex6.m`, the code randomly selects 100 rows from X, calls them `X_test`, and passes those rows to the `displayData` function. This function maps each row to a 20 pixel by 20 pixel grayscale image and displays the images together. The `displayData` function is provided, and you are encouraged to examine the code to see how it works. After you run this step, you should see an image like Figure 1.

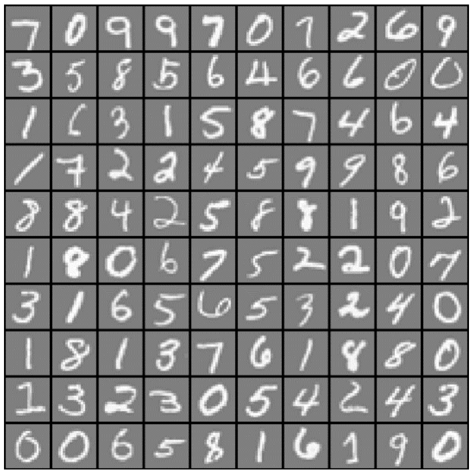

Figure 1: Examples from the dataset

[1] This is a subset of the MNIST handwritten digit dataset (http://yann.lecun.com/exdb/mnist/)

1.2 Model representation

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 digit images. Since the images are of size 20×20, this gives us 400 input layer units (excluding the extra bias unit which always outputs +1). The training data will be loaded into the variables X and y by the `ex6.m` script.
You have been provided with a set of network parameters $(\Theta^{(1)}, \Theta^{(2)})$. These are stored in `ex6weights.mat` and will be loaded by `ex6.m` into `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).

```
% Load the weights into variables Theta1 and Theta2
load('ex6weights.mat');
% The matrices Theta1 and Theta2 will now be in your Octave/MATLAB environment
% Theta1 has size 25 x 401
% Theta2 has size 10 x 26
```

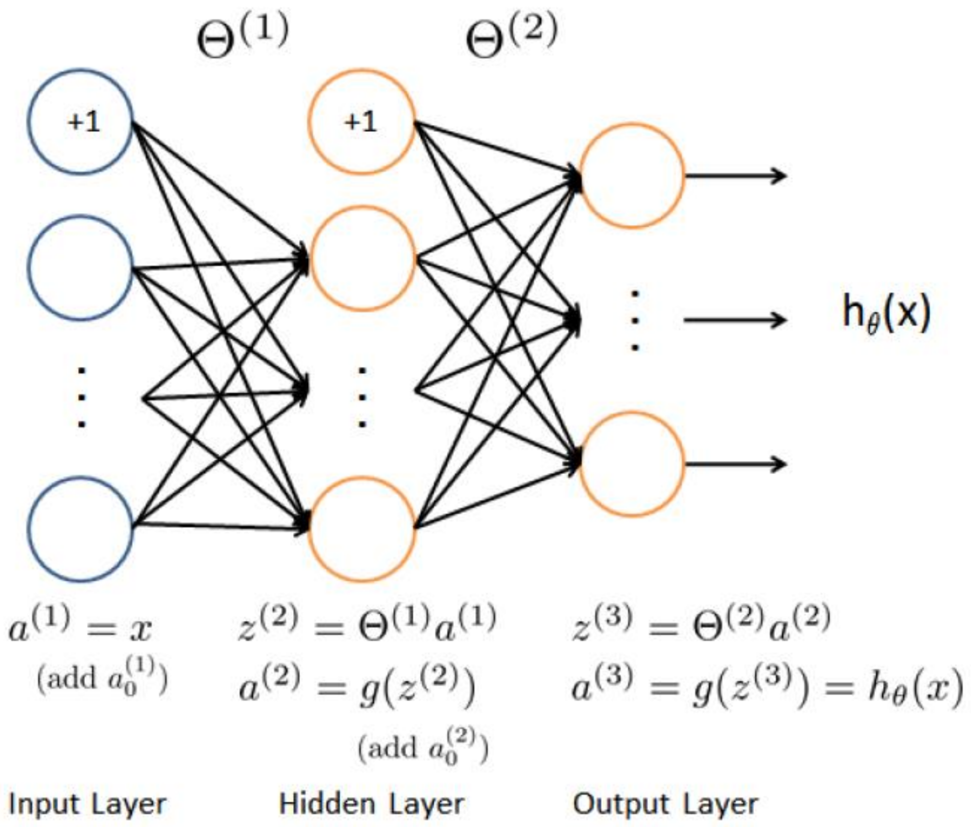

Figure 2: Neural network model

1.3 Forward propagation and cost function

Now you will implement the cost function and gradient for the neural network. First, complete the code in `nnCostFunction.m` to return the cost. Recall that the cost function for the neural network is

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

where $h(x^{(i)})$ is computed as shown in the Figure 2 and $K = 10$ is the total number of possible labels. Note that $h(x^{(i)})_k = a_k^{(3)}$ is the activation (output value) of the k^{th} output unit. Also, recall that 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

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

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.
You should implement the forward propagation computation that computes $h(x^{(i)})$ 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).

Implementation Note: The matrix X contains the examples in rows (i.e., $X(i,:)$ is the i^{th} training example $x^{(i)}$, expressed as a $n \times 1$ vector.) When you complete the code in `nnCostFunction.m`, you will need to add the column of 1's to the X matrix. The parameters for each unit in the neural network is represented in `Theta1` and `Theta2` as one row. Specifically, the first row of `Theta1` corresponds to the first hidden unit in the second layer. You can use a for-loop over the examples to compute the cost.

Once you are done, `ex6.m` will call your `nnCostFunction` using the loaded set of parameters (used here only for debugging purposes) in `Theta1` and `Theta2`. You should see that the cost is about 0.287135.

2 Back Propagation

In this part of the exercise, you will implement the backpropagation algorithm to compute the gradient for the neural network cost function. You will need to complete the `nnCostFunction.m` so that it returns an appropriate value for `grad`. 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 `fmincg`. Later, you will have a chance to verify that your gradient computation is correct.

2.1 Random initialization

When training neural networks, it is important to randomly initialize the parameters for symmetry breaking (initializing the parameters to zeros all together will not work here). One effective strategy for random initialization is to randomly select values for $\Theta^{(l)}$ uniformly in the range $[-\epsilon_{init}, \epsilon_{init}]$. You should use $\epsilon_{init} = 0.12$.[2] This range of values ensures that the parameters are kept small and makes the learning more efficient.
Take a look at `randInitializeWeights.m` which initializes the weights for Θ through the following code (you don't need to change any code in this file):

```
% Randomly initialize the weights to small values
epsilon_init = 0.12;
W = rand(L_out, 1 + L_in) * 2 * epsilon_init – epsilon_init;
```

2.2 Back propagation

Now, you will implement the backpropagation algorithm. Recall that the intuition behind the backpropagation algorithm is as follows. Given a training example $(x^{(i)}, y^{(i)})$, we will first run a "forward pass" to compute all the activations throughout the network, including the output value of the hypothesis $h(x^{(i)})$. 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 (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 $i = 1{:}m$ and place steps 1-4 below inside the for-loop, with the i^{th} iteration performing the calculation on the i^{th} training example $(x^{(i)}, y^{(i)})$.

1. Set the input layer's values $a^{(1)}$ to the i^{th} training example $x^{(i)}$. Perform a forward propagation (Figure 2), 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. In Octave/MATLAB, if `a1` is a column vector, adding one corresponds to `a1=[1; a1]`.

[2] One effective strategy for choosing ϵ_{init} is to base it on the number of units in the network. A good choice of ϵ_{init} is $\epsilon_{init} = \frac{\sqrt{6}}{\sqrt{L_{in}+L_{out}}}$, where $L_{in} = s_l$ and $L_{out} = s_{l+1}$ are the number of unit in the layers adjacent to $\Theta^{(l)}$.

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

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

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 ($y_k = 0$). You may find logical arrays helpful for this task.

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

$$\delta^{(2)} = (\Theta^{(2)})^T \delta^{(3)} \ .* \ \left(a^{(2)}\right) \ .* \ (1 - a^{(2)})$$

4. Accumulate the gradient from this example using the following formula. Note that you should skip or remove $\delta_0^{(2)}$. In Octave/MATLAB, removing $\delta_0^{(2)}$ corresponds to `delta2 = delta2(2:end)`.

$$Theta2\_grad = Theta2\_grad + \left(\delta^{(3)} * \left(a^{(2)}\right)^T\right)$$

$$Theta1\_grad = Theta1\_grad + \left(\delta^{(2)} * \left(a^{(1)}\right)^T\right)$$

Implementation Note: You should implement the backpropagation algorithm only after you have successfully completed the forward propagation and cost function. While implementing the backpropagation algorithm, it is often useful to use the `size` function to print out the sizes of the variables you are working with if you run into dimension mismatch errors ("`nonconformant arguments`" errors in Octave/MATLAB).

After you have implemented the backpropagation algorithm, the script `ex6.m` will proceed to run gradient checking on your implementation. The gradient check will allow you to increase your confidence that your code is computing the gradients correctly.

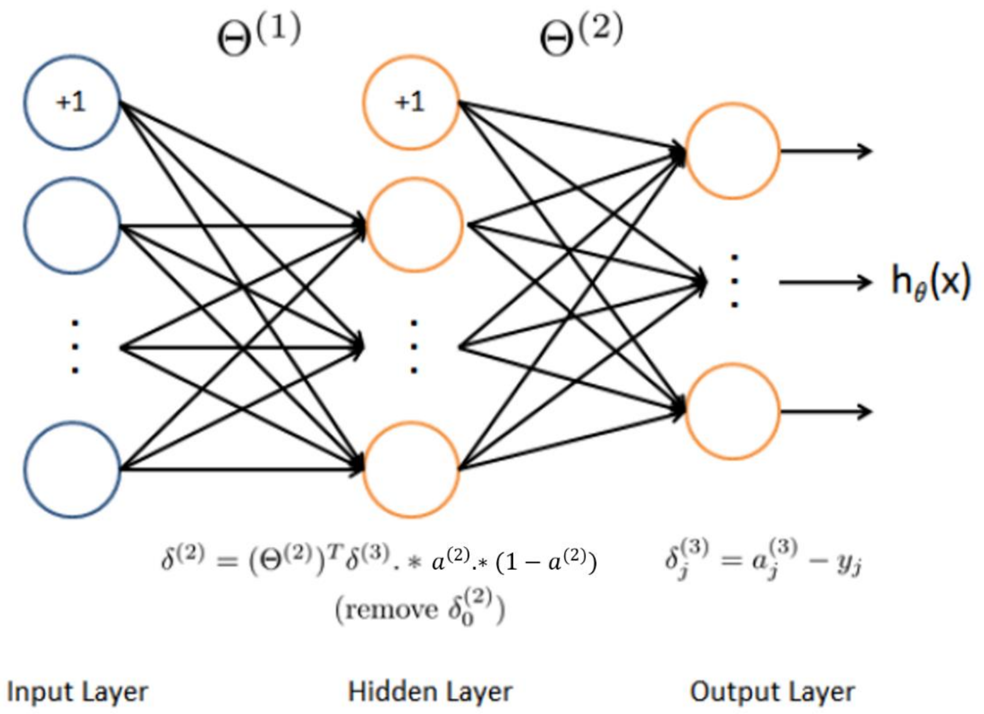
$\Theta^{(1)}$
$\Theta^{(2)}$
+1
+1
$h_\theta(x)$
$\delta^{(2)} = (\Theta^{(2)})^T\delta^{(3)} .* a^{(2)} .* (1 - a^{(2)})$
(remove $\delta_0^{(2)}$)
$\delta_j^{(3)} = a_j^{(3)} - y_j$
Input Layer
Hidden Layer
Output Layer

Figure 3: Back Propagation Updates

2.3 Gradient checking

`computeNumericalGradient.m` calculates a numerical estimate of gradients, $\frac{\partial}{\partial\theta_i}J(\theta)$ for each i, via a different technique from what you used in your `nnConstFunction.m`. While you are not required to modify this file, we highly encourage you to take a look at the code to understand how it works.
In the next step, `ex6.m` will run the provided function `checkNNGradients.m` which will create a small neural network and dataset that will be used for checking your gradients. If your back propagation implementation is correct, you should see a small (less than $1e-9$) relative difference between gradients you calculate in `nnConstFunction.m` and the numerical estimate of gradients `computeNumericalGradient.m` calculates.
(This step is only for debugging purposes and you do not need to do anything for this step other than checking the relative difference value. If the relative difference is small, this will give you confidence that your gradient calculation in `nnConstFunction.m` is correct. If the relative difference is not small, that means your implementation of gradients in `nnConstFunction.m` must have some mistakes.)

2.4 Learning parameters using **`fmincg`**

After you have successfully implemented the neural network cost function and gradient computation, the next step of the `ex6.m` script will use `fmincg` to learn a good set of parameters.
After the training completes, the `ex6.m` script will proceed to report the testing accuracy of your classifier by making predictions for the examples in the testing set and computing the percentage of examples it got correct. If your implementation is correct, you should see a reported testing accuracy of about 94% (this may vary by about 1% due to the random initialization). We encourage you to try training the neural network for more iterations (e.g., set MaxIter to 500) and investigate its effect on the testing accuracy.
At the very end, an interactive sequence will launch displaying images from the testing dataset one at a time, while the console prints out the actual and predicted labels for the displayed image. To stop the image sequence, enter `q` or press `Ctrl-C`.