Laboratory # 3
Neural Networks

In groups of 2 develop the following exercises:

Files included in this laboratory:

1. exercise3.m - Octave script that will help you through the first part of exercise 3,
2. exercise3_nn.m - Octave script that will help you through the second part of exercise 3,
3. exercise3data1.mat - Training set of hand-written digits,
4. exercise3weights.mat - Initial weights for the neural network exercise,
5. [*] lrCostFunction.m - Logistic regression cost function,
6. [*] oneVsAll.m - Train a one-vs-all multi-class classifier,
7. [*] predictOneVsAll.m - Predict using a one-vs-all multi-class classifier,
8. [*] predict.m - Neural network prediction function.
9. exercise4.m - Octave script that will help you through exercise 4,
10. exercise4data1.mat - Training set of hand-written digits,
11. exercise4weights.mat - Neural network parameters for exercise 4,
12. [*] sigmoidGradient.m - Compute the gradient of the sigmoid function,
13. [*] randInitializeWeights.m - Randomly initialize weights,
14. [*] nnCostFunction.m - Neural network cost function.

* indicates the files you must complete

This laboratory is composed by 2 exercises; you will implement a neural network to recognize handwritten digits using forward and back propagation. The neural network will be able to represent complex models that form non-linear hypotheses. For these exercises, you will use parameters from a neural network that has been already trained. Your goal is to implement the feedforward propagation, as well as the backpropagation algorithm and gradient descent to use the found weights for prediction.

For the exercises, you will use neural networks to recognize handwritten digits (from 0 to 9). Automated handwritten digit recognition is widely used today. These exercises will show you how the methods you've learned in class can be used for this classification task.

1. The dataset

For the first exercise, you are given a dataset in `exercise3data1.mat` that contains 5000 training examples of handwritten digits. (This is a subset of the MNIST handwritten digit dataset available in (http://yann.lecun.com/exdb/mnist/).

The `.mat` format means that the data has been saved in a native Octave matrix format, instead of a text (ASCII) format like a csv-file. These matrices can be read directly into your program by using the load command. After loading the data, matrices of the correct dimensions and values will appear in your program's memory. The matrix will already be named, so you do not need to assign names to them.

```
% Load saved matrices from file
load('exercise3data1.mat');
% The matrices X and y will now be in your Octave environment
```

Each of the 5000 training examples in `exercise3data1.mat`, 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.

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

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 indexing, where there is no zero index, the digit zero has been mapped 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.

2. Visualizing the data

As before, you will begin by visualizing a subset of the training set. In Part 1 of `exercise3.m`, the code randomly selects 100 rows from *X* 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 has been provided, 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.

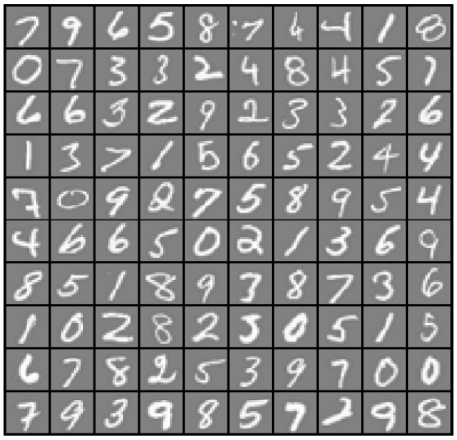

Figure 1: Examples from the dataset

3. 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 x 20, this gives us 400 input layer units (excluding the extra bias unit which always outputs +1). As before, the training data will be loaded into the variables *X* and *y*.

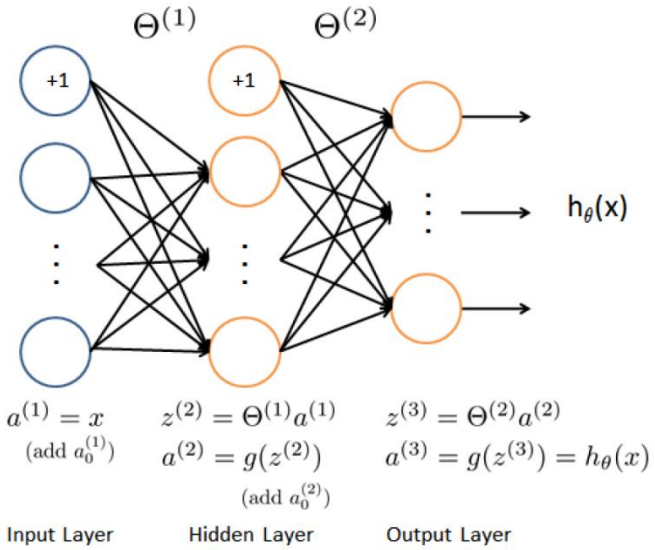

Figure 2: Neural network model.

You have been provided with a set of network parameters ($\theta^{(1)}; \theta^{(2)}$) already trained. These are stored in `exercise3weights.mat` and will be loaded by `exercise3_nn.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 saved matrices from file
load('exercise3weights.mat');
% The matrices Theta1 and Theta2 will now be in your Octave
% environment
% Theta1 has size 25 x 401
% Theta2 has size 10 x 26
```

4. Feedforward Propagation and Prediction

Now you will implement feedforward propagation for the neural network. You will need to complete the code in `predict.m` to return the neural network's prediction.

You should implement the feedforward computation that computes $h_\theta(x^{(i)})$ for every example i and returns the associated predictions. The prediction from the neural network will be the label that has the largest output $(h_\theta(x))_k$.

Tip: The matrix X contains the examples in rows. When you complete the code in `predict.m`, you will need to add the column of 1's to the matrix. The matrices `Theta1` and `Theta2` contain the parameters for each unit in rows. That is, the first row of `Theta1` corresponds to the first hidden unit in the second layer. In Octave, when you compute $z^{(2)} = \theta^{(1)}a^{(1)}$, be sure that your index (and if necessary, transpose) X correctly so that you get $a^{(1)}$ as a column vector.

Once you are done, `exercise3_nn.m` will call your predict function using the loaded set of parameters for `Theta1` and `Theta2`. You should see that the playing images from the training set one at a time, while the console prints out the predicted label for the displayed image. To stop the image sequence, press Ctrl-C.

5. The cost function in Feedforward

For this part of laboratory # 3, you have been provided with a set of network parameters ($\Theta^{(1)}$; $\Theta^{(2)}$), which have been already trained for you. These are stored in `exercise4weights.mat` file and will be loaded by `exercise4.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 as before).

```
% Load saved matrices from file
load('exercise4weights.mat');
% The matrices Theta1 and Theta2 will now be in your workspace
% Theta1 has size 25 x 401
% Theta2 has size 10 x 26
```

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 (without regularization) is:

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

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 $h_\theta(x^{(i)})_k = a^{(3)}_k$ 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:

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

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 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).

Tip: The matrix *X* contains the examples in rows (i.e., `X(i,:)'` is the *i-th* training example $x^{(i)}$, expressed as a *n x 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. Remember, 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, `exercise4.m` will call your `nnCostFunction` using the loaded set of parameters for `Theta1` and `Theta2`. You should see that the cost is about 0.287629.

5.1 Regularized cost function

The cost function for neural networks with regularization is given by:

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

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 it has been explicitly listed the indices above for Θ(1) and Θ(2) for clarity, do note that your code should in general work with Θ(1) and Θ(2) of any size.

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 un-regularized cost function *J* using your existing `nnCostFunction.m` and then later add the cost for the regularization terms.

Once you are done, the next exercise, `exercise4.m` will call your `nnCostFunction` using the loaded set of parameters for `Theta1` and `Theta2`, and λ = 1. You should see that the cost is about 0.383770.

6. Backpropagation

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(Θ)* using an advanced optimizer such as *`fmincg`*.

You will first implement the backpropagation algorithm to compute the gradients for the parameters for the (un-regularized) neural network. After you have verified that your gradient computation for the un-regularized case is correct, you will implement the gradient for the regularized neural network.

6.1 Sigmoid gradient

To help you get started with this part of the exercise, you will first implement the sigmoid gradient function. The gradient for the sigmoid function can be computed as:

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

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

When you are done, try testing a few values by calling `sigmoidGradient(z)` at the Octave command line. For large values (both positive and negative) of *z*, the gradient should be close to 0. When *z* = 0, the gradient should be exactly 0.25. Your code should also work with vectors and matrices. For a matrix, your function should perform the sigmoid gradient function on every element.

6.2 Random initialization

When training neural networks, it is important to randomly initialize the parameters for symmetry breaking. One effective strategy for random initialization is to randomly select values for $\Theta^{(l)}$ uniformly in the range $[-\varepsilon_{init}; \varepsilon_{init}]$.

You should use $\varepsilon_{init} = 0:12$. This range of values ensures that the parameters are kept small and makes the learning more efficient. Your job is to complete `randInitializeWeights.m` to initialize the weights for Θ; modify the file and fill in the following code:

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

6.3 The backpropagation algorithm

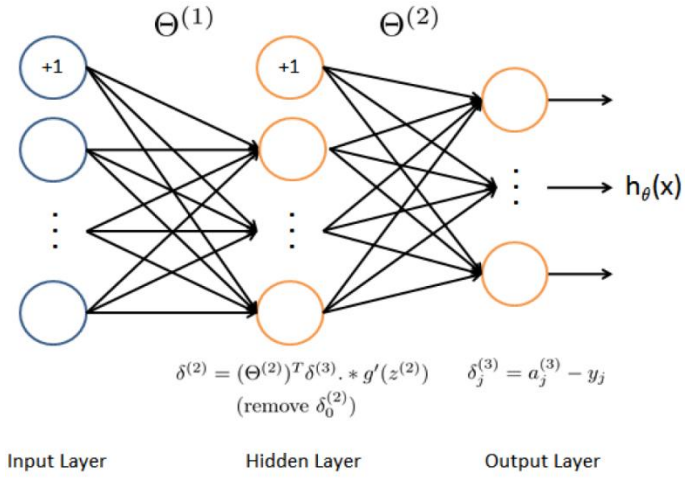

Figure 3: Backpropagation Updates.

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" $\delta^{(l)}_j$ 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 $\delta^{(3)}_j$ (since layer 3 is the output layer). For the hidden units, you will compute $\delta^{(l)}_j$ based on a weighted average of the error terms of the nodes in layer $(l + 1)$.

In detail, here is the backpropagation algorithm (also shown in Figure 3). You should implement steps 1 to 4 in a loop that processes one example at a time. You should implement a for-loop for `t = 1:m` and place the 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 below will divide the accumulated gradients by m to obtain the gradients for the neural network cost function.

1. Set the input layer's values $(a^{(1)})$ to the *t-th* training example $x^{(t)}$. Perform a feedforward pass (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, if a 1 is a column vector, adding one corresponds to `a1 = [1 ; a1]`.

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 ($y_k = 0$).

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 example using the following formula. Note that you should skip or remove $\delta^{(2)}_0$. In Octave, removing $\delta^{(2)}_0$ corresponds to `delta_2 = delta_2(2:end)`.

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

5. Obtain the (un-regularized) gradient for the neural network cost function by dividing the accumulated gradients by *1/m*:

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

Tip: You should implement the backpropagation algorithm only after you have successfully completed the feedforward and cost functions. 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).

After you have implemented the backpropagation algorithm, the script `exercise4.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.

6.4 Gradient checking

In your neural network, you are minimizing the cost function *J(Θ)*. 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(θ)* instead and use the following gradient checking procedure.

Suppose you have a function $f_i(\theta)$ that computes $\frac{\partial}{\partial \theta_i} J(\theta)$ 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 $\varepsilon = 10^{-4}$, you'll usually find that the left- and right-hand sides of the expression above will agree to at least 4 significant digits (and often many more).

The function to compute the numerical gradient has been implemented for you in `computeNumericalGradient.m`. While you are not required to modify the file, I highly encourage you to take a look at the code to understand how it works.

The next step of `exercise4.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 backpropagation implementation is correct, you should see a relative difference that is less than $1e^{-9}$.

Tip: When performing gradient checking, it is a lot 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. Each dimension of q requires two evaluations of the cost function and this can be expensive. In the function `checkNNGradients`, our code creates a small random model and dataset which is used with `computeNumericalGradient` for gradient checking. Furthermore, after you are confident that your gradient computations are correct, you should turn off gradient checking before running your learning algorithm.

Also, gradient checking works for any function where you are computing the cost and the gradient. Hence, you can use the same `computeNumericalGradient.m` function to check if your gradient implementations for the other exercises are correct too (e.g., logistic regression's cost function).

6.5 Regularized Neural Networks

After you have successfully implemented the backpropagation algorithm, you will add regularization to the gradient. To account for regularization, it turns out that you can add this as an additional term after computing the gradients using backpropagation.

After you have computed $\Delta_{ij}^{(l)}$ using backpropagation, you should add regularization using the following expressions:

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = D_{ij}^{(l)} = \frac{1}{m} \Delta_{ij}^{(l)} \qquad \text{for } j = 0$$

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

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

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

Recall that indexing in Octave starts from 1 (for both i and j), so `Theta1(2, 1)` actually corresponds to $\Theta^{(l)}_{2,0}$ (i.e., the entry in the second row, first column of the matrix $\Theta^{(l)}$, shown above) Now modify your code that computes grad in `nnCostFunction` to account for

regularization. After you are done, the `exercise4.m` script will proceed to run gradient checking on your implementation. If your code is correct, you should expect to see a relative difference that is less than $1e^{-9}$.

6.6 Learning parameters using *fmincg*

After you have successfully implemented the neural network cost function and gradient computation, the next step of the `exercise4.m` script will use *fmincg* to learn a good set of parameters.

After the training completes, the `exercise4.m` script 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 95.3% (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. I encourage you to try training the neural network for more iterations (e.g., set `MaxIter` to 400) and also vary the regularization parameter l. With the right learning settings, it is possible to get the neural network to perfectly fit the training set.

7. Submission and Grading

After completing the code in the necessary files, "up load" the modified code files in SIDWeb – section "Trabajos" – Laboratory 3. The following is a breakdown of how each part of this exercise is graded:

Task	Submitted File	Points
For exercise 3:		
Regularized Logistic Regression	lrCostFunction.m	15
One-vs-all classifier training	oneVsAll.m	10
One-vs-all classifier prediction	predictOneVsAll.m	10
Neural Network Prediction Function	predict.m	15
For exercise 4:		
Feedforward and Cost Function	nnCostFunction.m	15
Regularized Cost Function		8
Sigmoid Gradient	sigmoidGradient.m	2
Neural Net Gradient Function (Backpropagation)	nnCostFunction.m	20
Regularized Gradient		5
Total points		**100**