Programming Exercise 4:

Part 1: Unsupervised Learning
Part 2 Neural Networks

Machine Learning

Introduction

This exercise consists of three parts: part 1 – Unsupervised learning, part 2 Neural Network.

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.

You can also find instructions for installing Octave/MATLAB in the "Environment Setup Instructions" of the course website.

Where to get help

The exercises in this course use Octave[1] or MATLAB, a high-level program- ming language well-suited for numerical computations. If you do not have Octave or MATLAB installed, please refer to the installation instructions in the "Environment Setup Instructions" of the course website.

At the Octave/MATLAB command line, typing `help` followed by a function name displays documentation for a built-in function. For example, `help plot` will bring up help information for plotting. Further documentation for Octave functions can be found at the Octave documentation pages. MAT- LAB documentation can be found at the MATLAB documentation pages.

We also strongly encourage using the online **Discussions** to discuss exercises with other students. However, do not look at any source code written by others or share your source code with others.

Part 1: K-means Clustering

Introduction

In this exercise, you will implement the *K*-means clustering algorithm and apply it to compress an image.

Files included in this exercise

`ex7data2.mat` - Example Dataset for *K*-means
`ex7faces.mat` - Faces Dataset
`bird small.png` - Example Image
`displayData.m` - Displays 2D data stored in a matrix
`drawLine.m` - Draws a line over an existing figure
`plotDataPoints.m` - Initialization for *K*-means centroids
`plotProgresskMeans.m` - Plots each step of *K*-means as it proceeds
`runkMeans.m` - Runs the *K*-means algorithm
`[*] projectData.m` - Projects a data set into a lower dimensional space
`[*] recoverData.m` - Recovers the original data from the projection
`[*] findClosestCentroids.m` - Find closest centroids (used in *K*-means)
`[*] computeCentroids.m` - Compute centroid means (used in *K*-means)
`[*] kMeansInitCentroids.m` - Initialization for *K*-means centroids

* indicates files you will need to complete

Throughout this part of the exercise, you will be using the script `ex7.m`, This script sets up the dataset for the problems and makes calls to functions that you will write/update. You are only required to modify functions in other files, by following the instructions in this assignment.

1 *K*-means Clustering

In this this exercise, you will implement the *K*-means algorithm and use it for image compression. You will first start on an example 2D dataset that will help you gain an intuition of how the *K*-means algorithm works. After that, you will use the *K*-means algorithm for image compression by reducing the number of colors that occur in an image to only those that are most common in that image. You will be using `ex7.m` for this part of the exercise.

1.1 Implementing *K*-means

The *K*-means algorithm is a method to automatically cluster similar data examples together. Concretely, you are given a training set $\{x^{(1)},\ldots,x^{(m)}\}$ (where $x^{2(i)} = \mathbb{R}^n$), and want to group the data into a few cohesive "clusters". The intuition behind *K*-means is an iterative procedure that starts by guessing the initial centroids, and then refines this guess by repeatedly assigning examples to their closest centroids and then recomputing the centroids based on the assignments.

The *K*-means algorithm is as follows:

```
% Initialize centroids
centroids = kMeansInitCentroids(X,K);
for iter = 1:iterations
    % Cluster assignment step: Assign each data point to the
    % closest centroid. idx(i) corresponds
    % to c^(i), the index of the centroid
    % assigned to example i

    idx = findClosestCentroids(X, centroids);

    % Move centroid step: Compute means based on centroid
    % assignments
    centroids = computeMeans(X, idx, K);
end
```

The inner-loop of the algorithm repeatedly carries out two steps: (i) Assigning each training example $x^{(i)}$ to its closest centroid, and (ii) Recomputing the mean of each centroid using the points assigned to it. The *K*-means algorithm will always converge to some final set of means for the centroids. Note that the converged solution may not always be ideal and depends on the initial setting of the centroids. Therefore, in practice the *K*-means algorithm is usually run a few times with different random initializations. One way to choose between these different solutions from different random initializations is to choose the one with the lowest cost function value (distortion).

You will implement the two phases of the *K*-means algorithm separately in the next sections.

1.1.1 Finding closest centroids

In the "cluster assignment" phase of the *K*-means algorithm, the algorithm assigns every training example $x^{(i)}$ to its closest centroid, given the current positions of centroids. Specifically, for every example i we set

$$c^{(i)} := j \quad \text{that minimizes} \quad ||x^{(i)} - \mu_j||^2,$$

where $c^{(i)}$ is the index of the centroid that is closest to $x^{(i)}$, and μ_j is the position (value) of the j'th centroid. Note that $c^{(i)}$ corresponds to `idx(i)` in the starter code.

Your task is to complete the code in `findClosestCentroids.m`. This function takes the data matrix `X` and the locations of all centroids inside `centroids` and should output a one-dimensional array `idx` that holds the index (a value in $\{1,\ldots,K\}$ where K is total number of centroids) of the closest centroid to every training example.
You can implement this using a loop over every training example and every centroid.

Once you have completed the code in `findClosestCentroids.m`, the script `ex7.m` will run your code and you should see the output `[1 3 2]` corresponding to the centroid assignments for the first 3 examples.

You should now implement your solution.

1.1.2 Computing centroid means

Given assignments of every point to a centroid, the second phase of the algorithm recomputes, for each centroid, the mean of the points that were assigned to it. Specifically, for every centroid k we set

$$\mu_k := \frac{1}{|C_k|} \sum_{i \in C_k} x^{(i)}$$

where C_k is the set of examples that are assigned to centroid k. Concretely, if two examples say $x^{(3)}$ and $x^{(5)}$ are assigned to centroid $k = 2$, then you should update

$$\mu_2 = \frac{1}{2}\left(x^{(3)} + x^{(3)}\right)$$

You should now complete the code in `computeCentroids.m`. You can implement this function using a loop over the centroids. You can also use a loop over the examples; but if you can use a vectorized implementation that does not use such a loop, your code may run faster.

Once you have completed the code in `computeCentroids.m`, the script `ex7.m` will run your code and output the centroids after the first step of K- means.

You should now implement your solutions.

1.2 K-means on example dataset

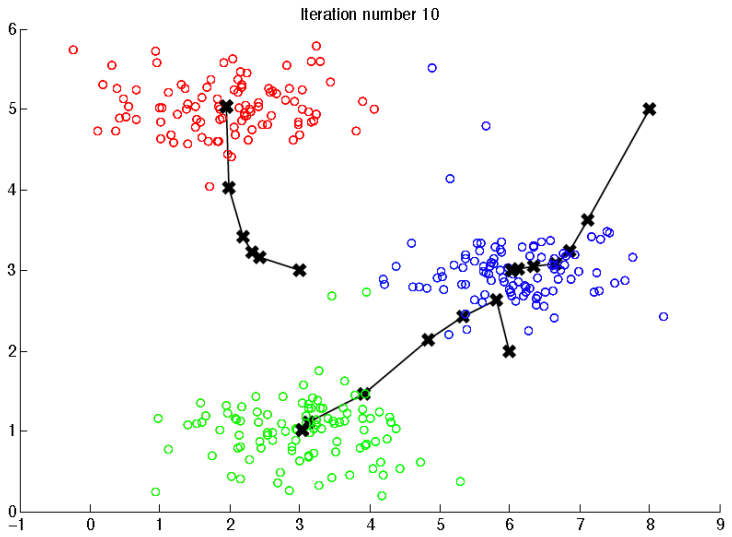

Figure 1: The expected output.

After you have completed the two functions (`findClosestCentroids` and

`computeCentroids`), the next step in `ex7.m` will run the *K*-means algorithm on a toy 2D dataset to help you understand how *K*-means works. Your functions are called from inside the `runKmeans.m` script. We encourage you to take a look at the function to understand how it works. Notice that the code calls the two functions you implemented in a loop.

When you run the next step, the *K*-means code will produce a visualization that steps you through the progress of the algorithm at each iteration. Press *enter* multiple times to see how each step of the *K*-means algorithm changes the centroids and cluster assignments. At the end, your figure should look as the one displayed in Figure 1

1.3 Random initialization

The initial assignments of centroids for the example dataset in `ex7.m` were designed so that you will see the same figure as in Figure 1. In practice, a good strategy for initializing the centroids is to select random examples from the training set. In this part of the exercise, you should complete the function `kMeansInitCentroids.m` with the following code:

```
% Initialize the centroids to be random examples

% Randomly reorder the indices of examples
randidx = randperm(size(X, 1));
% Take the first K examples as centroids
centroids = X(randidx(1:K), :);
```

The code above first randomly permutes the indices of the examples (using `randperm`). Then, it selects the first *K* examples based on the random permutation of the indices. This allows the examples to be selected at random without the risk of selecting the same example twice.

You should implement now your solution.

1.4 Image compression with K-means

Figure 2: The original 128x128 image.

In this exercise, you will apply *K*-means to image compression. In a straightforward 24-bit

color representation of an image[1], each pixel is represented as three 8-bit unsigned integers (ranging from 0 to 255) that specify the red, green and blue intensity values. This encoding is often referred to as the RGB encoding. Our image contains thousands of colors, and in this part of the exercise, you will reduce the number of colors to 16 colors.

By making this reduction, it is possible to represent (compress) the photo in an efficient way. Specifically, you only need to store the RGB values of the 16 selected colors, and for each pixel in the image you now need to only store the index of the color at that location (where only 4 bits are necessary to represent 16 possibilities).

In this exercise, you will use the *K*-means algorithm to select the 16 colors that will be used to represent the compressed image. Concretely, you will treat every pixel in the original image as a data example and use the *K*-means algorithm to find the 16 colors that best group (cluster) the pixels in the 3- dimensional RGB space. Once you have computed the cluster centroids on the image, you will then use the 16 colors to replace the pixels in the original image.

1.4.1 *K*-means on pixels

In Octave/MATLAB, images can be read in as follows:

```
% Load 128x128 color image (bird_small.png)
A = imread('bird_small.png');

% You will need to have installed the image package to used
% imread. If you do not have the image package installed,
% you should instead change the following line to

% load('bird_small.mat'); %Loads the image into the variable
```

This creates a three-dimensional matrix A whose first two indices identify a pixel position and whose last index represents red, green, or blue. For example, `A(50, 33, 3)` gives the blue intensity of the pixel at row 50 and column 33.

The code inside `ex7.m` first loads the image, and then reshapes it to create an $m \times 3$ matrix of pixel colors (where $m = 16384 = 128 \times 128$), and calls your *K*-means function on it.

After finding the top $K = 16$ colors to represent the image, you can now assign each pixel position to its closest centroid using the `findClosestCentroids` function. This allows you to represent the original image using the centroid assignments of each pixel. Notice that you have significantly reduced the number of bits that are required to describe the image. The original image required 24 bits for each one of the 128×128 pixel locations, resulting in total size of $128 \times 128 \times 24 = 393,216$ bits. The new representation requires some overhead storage in form of a dictionary of 16 colors, each of which require 24 bits, but the image itself then only requires 4 bits per pixel location. The final number of bits used is therefore $16 \times 24 + 128 \times 128 \times 4 = 65,920$ bits, which corresponds to compressing the original image by about a factor of 6.

[1] [2]The provided photo used in this exercise belongs to Frank Wouters and is used with his permission.

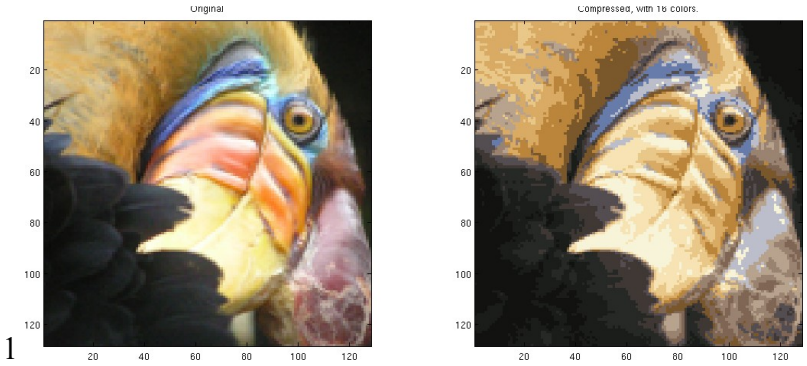

Figure 3: Original and reconstructed image (when using *K*-means to com- press the image).

Finally, you can view the effects of the compression by reconstructing the image based only on the centroid assignments. Specifically, you can replace each pixel location with the mean of the centroid assigned to it. Figure 3 shows the reconstruction we obtained. Even though the resulting image retains most of the characteristics of the original, we also see some compression artifacts.

You can implement now this part..

1.5 Optional (ungraded) exercise: Use your own image

In this exercise, modify the code we have supplied to run on one of your own images. Note that if your image is very large, then *K*-means can take a long time to run. Therefore, we recommend that you resize your images to manageable sizes before running the code. You can also try to vary *K* to see the effects on the compression.

Part 2: Neural Networks – the complete example

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

Files included in this exercise

The files are placed in folder Lab4-2

`ex4.m` - Octave/MATLAB script that steps you through the exercise
`ex4data1.mat` - Training set of hand-written digits
`ex4weights.mat` - Neural network parameters for exercise 4
`displayData.m` - Function to help visualize the dataset
`fmincg.m` - Function minimization routine (similar to fminunc)
`sigmoid.m` - Sigmoid function
`computeNumericalGradient.m` - Numerically compute gradients
`checkNNGradients.m` - Function to help check your gradients
`debugInitializeWeights.m` - Function for initializing weights
`predict.m` - Neural network prediction function
`[*] sigmoidGradient.m` - Compute the gradient of the sigmoid function
`[*] randInitializeWeights.m` - Randomly initialize weights
`[*] nnCostFunction.m` - Neural network cost function

* indicates files you will need to complete

Throughout the exercise, you will be using the script `ex4.m`. These scripts set up the dataset for the problems and make calls to functions that you will write. You do not need to modify the script. You are only required to modify functions in other files, by following the instructions in this assignment.

1. Neural Networks

In the previous exercise, you implemented feedforward propagation for neural networks and used it to predict handwritten digits with the weights we provided. In this exercise, you will implement the backpropagation algorithm to *learn* the parameters for the neural network.
The provided script, `ex4.m`, will help you step through this exercise.

1.1. Visualizing the data

In the first part of `ex4.m`, the code will load the data and display it on a 2-dimensional plot (Figure 1) by calling the function `displayData`.

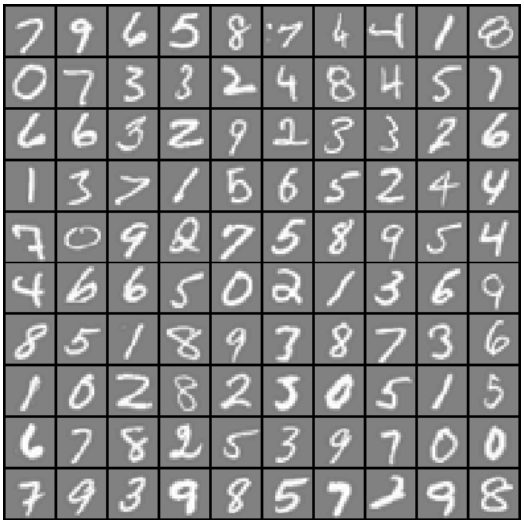

Figure 1: Examples from the dataset

This is the same dataset that you used in the previous exercise. 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.

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

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/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.

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 valuesof 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 training data will be loaded into the variables Xand yby the ex4.mscript.

You have been provided with a set of network parameters ($\Theta^{(1)}$, $\Theta^{(2)}$) already trained by us. These are stored in `ex4weights.mat` and will be loaded by `ex4.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( ' ex4weights.mat ' ) ;

% The matrices Theta1 and Theta2 will now be in your
workspace
% Theta1 has size 25 x 401
% Theta2 has size 10 x 26
```

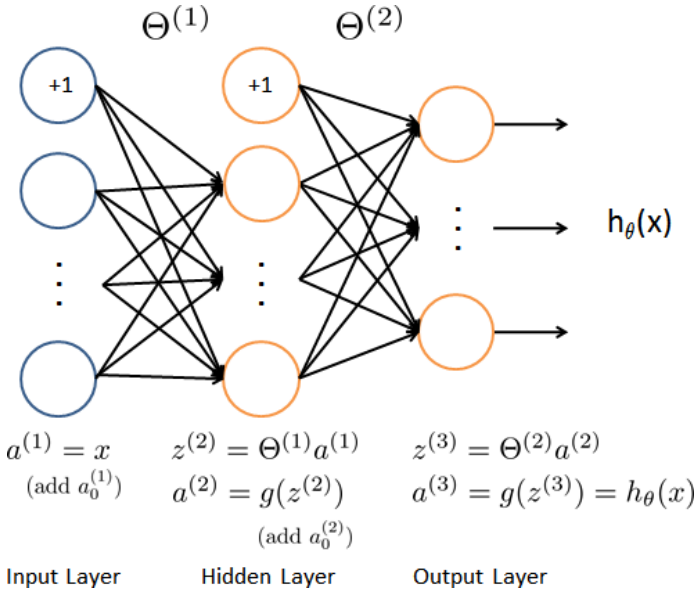

Figure 2: Neural network model.

1.3. Feedforward 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 (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 $h_\theta(x^{(i)})_k = a^{(3)}$ 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).

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, `ex4.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.

You should now implement your solutions.

1.4. Regularized cost function

The cost function for neural networks with regularization is given by

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

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**. 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 using your existing `nnCostFunction.m` and then later add the cost for the regularization terms.

Once you are done, `ex4.m` will call your `nnCostFunction` using the loaded set of parameters for `Theta1` and `Theta2`, and $\lambda = 1$. You should see that the cost is about 0.383770.

You should now implement your solutions.

2. 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 appropri- ate 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`.

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

2.1. 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

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

When you are done, try testing a few values by calling `sigmoidGradient(z)` at the Octave/MATLAB command line. For large values (both positive and negative) of `z`, the gradient should be close to 0. When `z = 0`, the gradi- ent 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.

You should now implement your solutions.

2.2. 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 $\varepsilon_{init} = 012$. 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;
```

2.3. Backpropagation

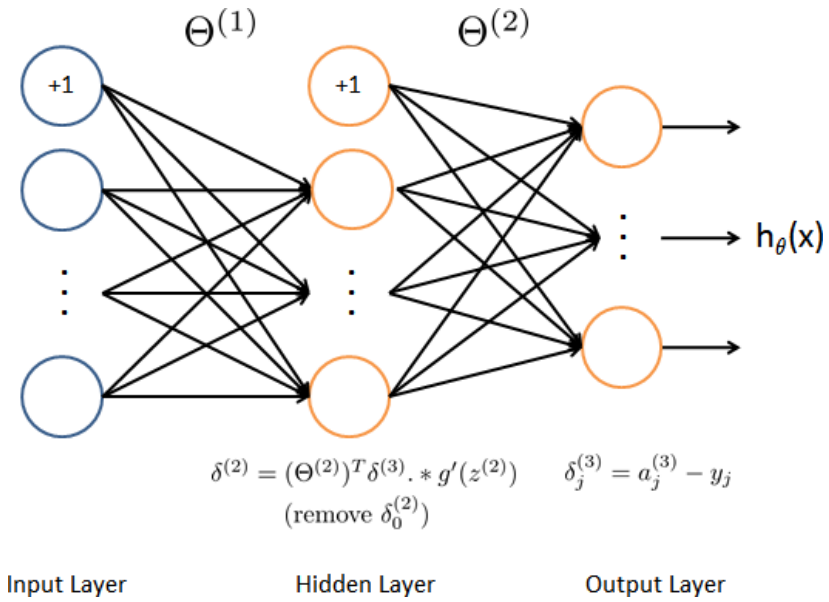

Figure 3: Backpropagation Updates.

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 hypthesis $h_\Theta(x)$. 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 (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 = 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.

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 MATLAB, if `a 1` is a column vector, adding one corresponds to `a 1 = [1 ; a 1]`.

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 be- longs 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 (explained in the previous programming exercise).

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 $\delta_0^{(2)}$. In Octave/MATLAB, removing $\delta_0^{(2)}$ corresponds to `delta 2 = delta 2(2:end)`.

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

5. Obtain the (unregularized) gradient for the neural network cost function by dividing the accumulated gradients by $1/m$

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

Octave/MATLAB 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/MATLAB).

After you have implemented the backpropagation algorithm, the script `ex4.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.

2.4. Gradient checking

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 purportedly computes 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 E. Similarly, $\theta^{(i-)}$ is the corresponding vector with the i-th element decreased by E. 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 $E = 10^{-4}$, 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).

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

In the next step of `ex4.m`, it 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.

Practical Tip: 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. Each dimension of θ 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.

Practical Tip: Gradient checking works for any function where you are computing the cost and the gradient. Concretely, 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).

Once your cost function passes the gradient check for the (unregularized) neural network cost function, you should submit the neural network gradient function (backpropagation).

2.5. 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.

Specifically, after you have computed $\Delta^{(l)}$ using backpropagation, you should add regularization using

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

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

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

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

Somewhat confusingly, indexing in Octave/MATLAB starts from 1 (for both i and j), thus `Theta1(2, 1)` actually corresponds to $\Theta^{(l}$ (i.e., the entry) in the second row, first column of the matrix $\Theta^{(1)}$ shown above).

Now modify your code that computes `grad` in `nnCostFunction` to account for regularization. After you are done, the `ex4.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.

You should now implement your solutions.

2.6. Learning parameters using `fmincg`

After you have successfully implemented the neural network cost function and gradient computation, the next step of the `ex4.m` script will use `fmincg` to learn a good set parameters.

After the training completes, the `ex4.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. We encourage you to try training the neural network for more iterations (e.g., set `MaxIter` to 400) 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. Visualizing the hidden layer

One way to understand what your neural network is learning is to visualize what 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 20 image and display it.[2] The next step of `ex4.m` 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 correspond roughly to detectors that look for strokes and other patterns in the input.

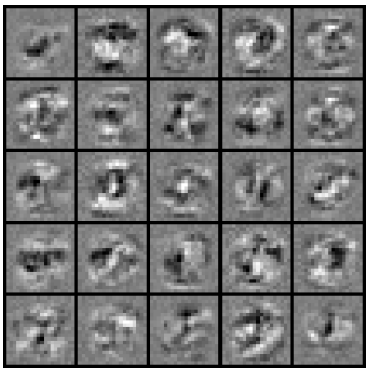

Figure 4: Visualization of Hidden Units.

[2] It turns out that this is equivalent to finding the input that gives the highest activation for the hidden unit, given a "norm" constraint on the input $\|x\|_2 \leq 1$

3.1. Optional (ungraded) exercise

In this part of the exercise, you will get to try out different learning settings for the neural network to see how the performance of the neural network varies with the regularization parameter λ and number of training steps (the `MaxIter` option when using `fmincg`).

Neural networks are very powerful models that can form highly complex decision boundaries. Without regularization, it is possible for a neural net- work to "overfit" a training set so that it obtains close to 100% accuracy on the training set but does not as well on new examples that it has not seen before. You can set the regularization λ to a smaller value and the `MaxIter` parameter to a higher number of iterations to see this for yourself.

You will also be able to see for yourself the changes in the visualizations of the hidden units when you change the learning parameters λ and `MaxIter`.

You do not need to submit any solutions for this optional (ungraded) exercise.

Submission

Submit the following files on Canvas, under Assignments from "LAB 4-2".

[*] `sigmoidGradient.m`- Compute the gradient of the sigmoid function
[*] `randInitializeWeights.m`- Randomly initialize weights
[*] `nnCostFunction.m` - Neural network cost function