Machine Learning	**Programming Assignment 4**
Comp540 Spring 2015	**due 20 March 2015 at 8 pm**

Introduction

In this problem, you will implement one-vs-all logistic regression and neural networks to recognize hand-written digits. To get started, please download the code base `pa5.zip` from Owlspace. When you unzip the archive, you will see the following files.

Name	**Edit?**	**Read?**	**Description**
lrCostFunction.m	Yes	Yes	Logistic regression cost function
predict.m	Yes	Yes	neural network prediction function
predictOneVsAll.m	Yes	Yes	Predict using a one-vs-all multi-class classifier
oneVsAll.m	Yes	Yes	train a one-vs-all classifier
sigmoidGradient.m	Yes	Yes	Compute the gradient of the sigmoid function
nnCostFunction.m	Yes	Yes	Neural network cost function
ex4.m	No	Yes	Matlab script that will run your functions for Part 1
ex4_nn.m	No	Yes	Matlab script that will run your functions for Part 2
ex4_nnlearn.m	No	Yes	Matlab script that will run your functions for Part 3
ex4data1.mat	No	No	Training set of hand-written digits
ex4weights.mat	No	No	Initial weights for neural net
displayData.m	No	No	Function to visualize data set
sigmoid.m	No	No	Sigmoid function
fmincg.m	No	Yes	Optimization function using conjugate gradient
randInitialWeight.m	No	Yes	Randomly initialize weights
computeNumericalGradient.m	No	Yes	Numerically compute gradients
checkNNGradients.m	No	Yes	Function to help check gradients
debugInitialWeights.m	No	No	Function for initializing weights
pa4_2015.pdf	No	Yes	this document

Problem 1: Multi-class classification (20 points)

For this exercise, you will use logistic regression and 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 have learned can be used for this classification task. First, you will extend your previous implemention of logistic regression and apply it to one-vs-all classification.

The data set

You are given a data set in `ex4data1.mat` that contains 5000 training examples of handwritten digits. These are taken from the famous MNIST benchmark. The `.mat` format means that that the data has been saved in a native Matlab 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, matrices of the correct dimensions and values will appear in your programs memory. The matrices will already be named as `X` and `y`.

There are 5000 training examples in `ex4data1.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. To make things more compatible with 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.

Visualizing the data set

You will begin by visualizing a subset of the training set. In Part 1 of `ex4.m`, the code randomly selects 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. We have provided the `displayData` function, 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.

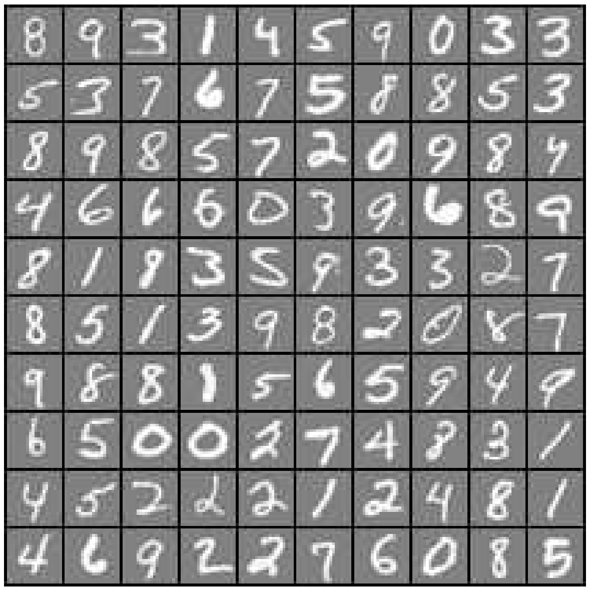

Figure 1: The training data

Vectorizing logistic regression

You will be using multiple one-vs-all logistic regression models to build a multi-class classifier. Since there are 10 classes, you will need to train 10 separate logistic regression classifiers. To make this training efficient, it is important to ensure that your code is well vectorized. In this section, you will implement a vectorized version of logistic regression that does not employ any `for` loops. You can use your previous code on logistic regression as a starting point for this exercise.

Vectorizing the cost function

We will begin by writing a vectorized version of the cost function. Recall that in (unregularized) logistic regression, the cost function is

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\left(-y^{(i)}log(h_\theta(x^{(i)})) - (1-y^{(i)})log(1-h_\theta(x^{(i)}))\right)$$

To compute each element in the summation, we have to compute $h_\theta(x^{(i)})$ for every example $x^{(i)}$, where $h_\theta(x^{(i)}) = g(\theta^T x^{(i)})$ where $g(z) = \frac{1}{1+e^{-z}}$ is the sigmoid function. To compute the product $\theta^T x^{(i)}$ for every example, we simply need to perform the matrix multiplication of `X` and θ. You need to vectorize the rest of the computation of $J(\theta)$ so that it contains no `for` loops to implement the summation shown above. Hint: consider using element-wise multiplication (`.*` in Matlab) and the `sum` operation in Matlab.

Vectorizing the gradient

Recall that the gradient of the (unregularized) logistic regression cost is a vector where the j^{th} element is defined as

$$\frac{\partial J}{\partial \theta_j} = \frac{1}{m}\sum_{i=1}^{m}\left((h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}\right)$$

To vectorize this operation, consider constructing the vector $h_\theta(X) - y$ of size $m \times 1$, whose i^{th} element is $h_\theta(x^{(i)}) - y^{(i)}$. Then, $\frac{1}{m}X^T(h_\theta(X) - y)$ will allow you to compute the partial derivatives without any `for` loops. If you are comfortable with linear algebra, we encourage you to work through the matrix multiplications above to convince yourself that the vectorized version does the same computations.

Vectorizing code can sometimes be tricky. One common strategy for debugging is to print out the sizes of the matrices you are working with using the `size` function. For example, given a data matrix `X` of size 100×20 (100 examples, 20 features) and θ, a vector with dimensions 20×1, you can observe that `X`θ is a valid multiplication operation, while θ`X` is not. Furthermore, if you have a non-vectorized version of your code, you can compare the output of your vectorized code and non-vectorized code to make sure that they produce the same outputs.

Vectorizing regularized logistic regression (10 points)

We will now add regularization to the cost function. Recall that for regularized logistic regression, the cost function is defined as

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\left(-y^{(i)}log(h_\theta(x^{(i)})) - (1-y^{(i)})log(1-h_\theta(x^{(i)}))\right) + \frac{\lambda}{2m}\sum_{j=1}^{d}\theta_j^2$$

Note that you should not be regularizing θ_0 which is used for the bias term. Correspondingly, the partial derivative of regularized logistic regression cost for θ_j is defined as

$$\begin{aligned}\frac{\partial J}{\partial \theta_j} &= \frac{1}{m}\sum_{i=1}^{m}\left((h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}\right) \quad \text{for j} = 0\\ \frac{\partial J}{\partial \theta_j} &= \frac{1}{m}\sum_{i=1}^{m}\left((h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}\right) + \frac{\lambda}{m}\theta_j \quad \text{for j} \geq 1\end{aligned}$$

Now add code in `lrCostFunction.m` to compute the cost function and gradient for regularized logistic regression. Once again, you should not put any loops into your code.

One-vs-all Classification (5 points)

In this part of the exercise, you will implement one-vs-all classification by training multiple regularized logistic regression classifiers, one for each of the K classes in our dataset (Figure 1). In the handwritten digits dataset, K = 10, but your code should work for any value of K. You should now complete the code in `oneVsAll.m` to train one classifier for each class. In particular, your code should return all the classifier parameters in a matrix $\Theta \in \Re^{K\times(d+1)}$, where each row of Θ corresponds to the learned logistic regression parameters for one class. You can do this with a for-loop from 1 to K, training each classifier independently. Note that the `y` argument to this function is a vector of labels from 1 to 10, where we have mapped the digit 0 to the label 10 (to avoid confusions with indexing). When training the classifier for class $k \in \{1,\ldots,K\}$, you will want a m-dimensional vector of labels `y`, where $y^{(i)} \in \{0,1\}$, indicates whether the i^{th} training instance belongs to class k ($y^{(i)} = 1$), or if it belongs to a different class ($y^{(i)} = 0$). You may find logical arrays helpful for this task.

Furthermore, you will be using `fmincg` for this exercise (instead of `fminunc`). `fmincg` works similarly to `fminunc`, but is more more efficient for dealing with a large number of parameters. After you have correctly completed the code for `oneVsAll.m`, the script `ex4.m` will continue to use your `oneVsAll` function to train a multi-class classifier.

One-vs-all prediction (5 points)

After training your one-vs-all classifier, you can now use it to predict the digit contained in a given image. For each input, you should compute the probability that it belongs to each class using the trained logistic regression classifiers. Your one-vs-all prediction function will pick the class for which the corresponding logistic regression classifier outputs the highest probability and return the class label (1, 2,..., or K) as the prediction for the input example. You should now complete the code in `predictOneVsAll.m` to use the one-vs-all classifier to make predictions. Once you are done, `ex4.m` will call your `predictOneVsAll` function using the learned value of Θ. You should see that the training set accuracy is about 95.96%

Problem 2: Neural networks (10 points)

In the previous problem you implemented multi-class logistic regression to recognize handwritten digits. However, logistic regression cannot form more complex hypotheses as it is only a linear classifier. In this part of the exercise, you will implement a neural network to recognize handwritten digits using the same training set as before. The neural network will be able to represent complex models that form non-linear hypotheses. In this problem, you will be using parameters from a neural network that we have already trained. Your goal is to implement the feedforward propagation algorithm to use our weights for prediction. In the next problem, you will write the backpropagation algorithm for learning the neural network parameters. The provided script, `ex4_nn.m`, will help you step through this exercise.

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 $+1$). As before, the training data will be loaded into the variables `X` and `y`. 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_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).

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

Feedforward Propagation and Prediction (10 points)

Now you will implement feedforward propagation for the neural network. You will need to complete the code in `predict.m` to return the neural networks prediction. You should implement the feedforward computation that computes $h_\theta(x^{(i)})$ for every example $x^{(i)}$ and returns the associated predictions. Similar to the one-vs-all classification strategy, the prediction from the neural network will be the label that has the largest output $(h_\theta(x))_k$.

Once you are done, `ex4_nn.m` will call your `predict` function using the loaded set of parameters for `Theta1` and `Theta2`. You should see that the accuracy is about 97.52%. After that, an interactive

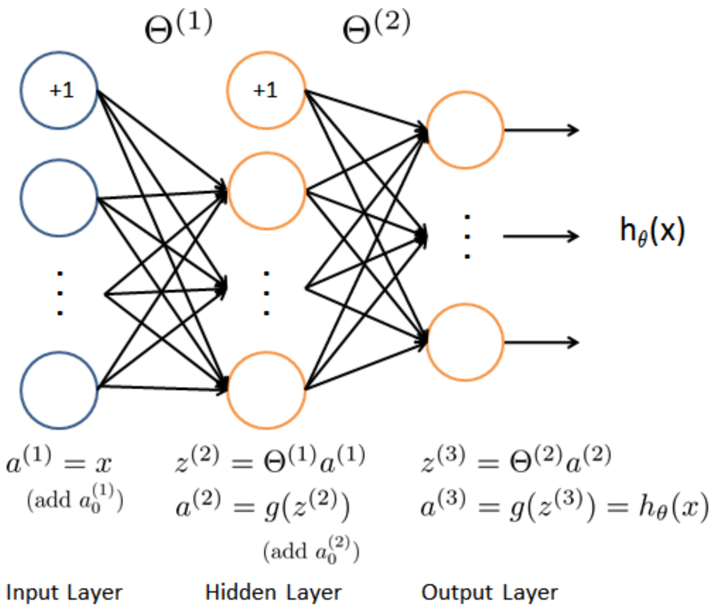

Figure 2: The neural network

sequence will launch displaying 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.

Problem 3: Neural network learning (30 points)

In the previous problem, you implemented feedforward propagation for neural networks and used it to predict handwritten digits with the weights we provided. In this problem, you will implement the backpropagation algorithm to learn the parameters for the neural network.

Throughout the problem, you will be using the script `ex4_nnlearn.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.

Feedforward and cost function (5 points)

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 Figure 2 and $h_\theta(x^{(i)})_k$ is the output of the k^{th} unit at the output layer. $K = 10$ is the number of labels. Also note that the y's are recoded as boolean vectors of size K. So, for example if $x^{(i)}$ is a image of the digit 7, the corresponding $y^{(i)}$ is the vector $[0, 0, 0, 0, 0, 0, 1, 0, 0, 0]^T$ with a 1 in the 7^{th} position. The y vector corresponding to an image of the digit zero is a boolean vector of length 10 with a 1 in the 10^{th} position.

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 three labels). Note that the matrix `X` contains the examples in rows (i.e., X(i,:) is the i^{th} training example $x^{(i)}$, expressed as a $d \times 1$ vector.) When you complete the code in `nnCostFunction.m`, you will need to add the column of 1s 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 and over the output labels to compute the cost.

Once you are done, `ex4_nnlearn.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.

Regularized cost function (5 points)

The cost function for neural networks with regularization is given by

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

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 units. 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_nnlearn.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.

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 $J(\theta)$ 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.

Sigmoid gradient (5 points)

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

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

When you are done, try testing a few values by calling `sigmoidGradient(z)` at the Matlab 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.

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 $[-\epsilon_{init}, +\epsilon_{init}]$. You should use $\epsilon_{init} = 0.12$. This range of values ensures that the parameters are kept small and makes the learning more efficient. We have implemented this procedure in `randInitializeWeights.m`.

The backpropagation algorithm (10 points)

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 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. 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, computing the activations $z^{(2)}$, $a^{(2)}$, $z^{(3)}$, and $a^{(3)}$ 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 term. In Matlab, if `a1` 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

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

   where `.*` stands for element-wise product of two vectors.

4. Accumulate the gradient from this example using the following formula. Note that you should skip or remove $\delta_0^{(2)}$.

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

5. Obtain the (unregularized) gradient for the neural network cost function by multiplying the accumulated gradients by $\frac{1}{m}$.

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

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 in Matlab. After you have implemented the backpropagation algorithm, the script `ex4_nnlearn.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.

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 $\Theta^{(1)}$ and $\Theta^{(2)}$ 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 computes $\frac{\partial}{\partial\theta}J(\theta)$; you would like to check if $f_i$is outputting correct derivative values. Let,

$$\begin{aligned} \theta^{(i+)} &= \theta + [0, \ldots, 0, \epsilon, 0, \ldots, 0]^T \\ \theta^{(i-)} &= \theta - [0, \ldots, 0, \epsilon, 0, \ldots, 0]^T \end{aligned}$$

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'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 $\epsilon = 10^4$, you willl 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_nnlearn.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.

Regularized neural networks (5 points)

After you have successfully implemeted 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 $\Delta_{ij}^{(l)}$ using backpropagation, you should add regularization using

$$\begin{aligned}\frac{\partial}{\partial\Theta_{ij}^{(l)}} &= \frac{1}{m}\Delta_{ij}^{(l)} \text{ for } j = 0 \\ &= \frac{1}{m}\Delta_{ij}^{(l)} + \frac{\lambda}{m}\Theta_{ij}^{(l)} \quad \text{for } j \geq 1\end{aligned}$$

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

Now modify your code that computes grad in `nnCostFunction` to account for regularization. After you are done, the `ex4_nnlearn.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.

Learning parameters using fmincg

After you have successfully implemented the neural network cost function and gradient computation, the next step of the `ex4_nnlearn.m` script will use `fmincg` to learn a good set parameters. After the training completes, the `ex4_nnlearn.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.

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. The next step of `ex4_nnlearn.m` does this by using the `displayData` function and it will show you an image (similar to Figure 3) 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.

Varying λ and Maxiter

Now you can 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 network 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 youself. 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`.

What to turn in

Please zip up all the files in the archive (including files that you did not modify) and submit it as `pa4_netid`.zip on Owlspace before the deadline. Include a PDF file in the archive that presents your plots and your discussion of results from the problems above.

Acknowledgment

These problems are from Andrew Ng's exercise on logistic regression and neural nets.

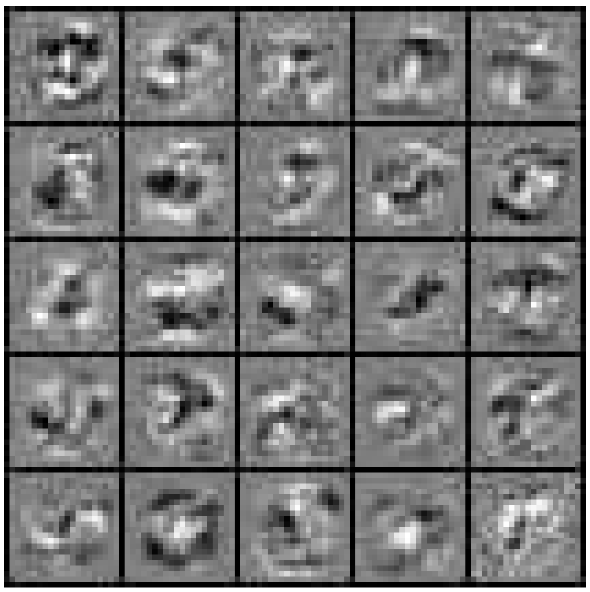

Figure 3: Visualization of hidden units