Homework 2: Neural Networks*

Pattern Recognition and Machine Learning

September 21, 2016

Instructions

- **Submission:** 本次作业需提交的内容包括实验报告文档和代码, 实验报告文档主要描述原理和结果，请每位同学压缩打包, 压缩包的命名名字格式为“学号-班级-姓名-PRMLHW2.zip”. 请各班负责人（班长、学委等）统一将本班的所有作业收集并按班级和作业名的格式统一打包（如计算机1401班的命名为“计算机1401-PRMLHW2.zip”）, 在deadline前发到yxliang@csu.edu.cn. 未按时提交的则在deadline后自行按指定的命名格式打包后发到yxliang@csu.edu.cn.
- **Late homework policy:** 本次作业提交的deadline为2016.11.23 24:00, 约占平时成绩的1/3, 迟交则按每天0.8的比例进行计算, 如本来作业能打10分, 迟交1天则算8分, 迟交2天则算6.4分, 以此类推.
- **Collaboration policy:** Homeworks must be done individually, except where otherwise noted in the assignments. “Individually” means each student must hand in their own answers, and each student must write and use their own code in the programming parts of the assignment. It is acceptable for students to collaborate in figuring out answers and to help each other solve the problems, though you must in the end write up your own solutions individually, and you must list the names of students you discussed this with. We will be assuming that you will be taking the responsibility to make sure you personally understand the solution to any work arising from such collaboration.

Part I
Neural Networks: Feedforward Propagation

In this part of the exercise, you will implement a neural network 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. For this part, 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.

1 Dataset

You are given a data set in `ex3data1.mat` that contains 5000 training examples of handwritten digits[1]. The .mat format means that the data has been saved in a native Octave/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

*本次作业内容来自Andrew Ng's mlClass http://www.ml-class.org/course/class/index/.

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

load command. After loading, 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('ex3data1.mat');
% The matrices X and y will now be in your Octave environ
```

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.

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

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/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.

2 Visualizing the data

You will begin by visualizing a subset of the training set. In the first part of `ex3.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.

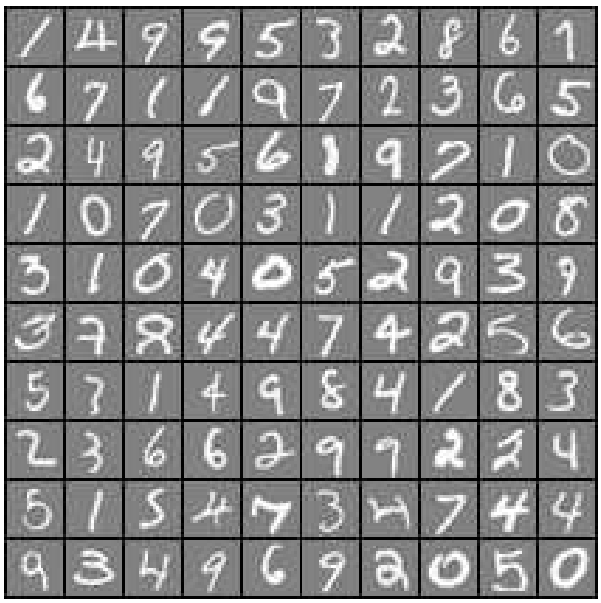

Figure 1: Examples from the dataset

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 20×20, 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 `ex3weights.mat` and will be loaded by `ex3_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). 注意到每个"神经元"的输入－输出映射关系其实就是一个逻辑回归(logistic regression). 虽然采用sigmoid函数，但你也可以选择双曲正切函数tanh: $g(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$. $\tanh(z)$ 函数是sigmoid函数的一种变体，它的取值范围为$[-1, 1]$，而不是sigmoid函数的$[0, 1]$。上次作业中推导了如果选择$g(z) = 1/(1 + \exp(-z))$，那么它的导数就是$g'(z) = g(z)(1 - g(z))$. 如果选择tanh 函数，那它的导数就是$g'(z) = 1 - (g(z))^2$，请根据tanh函数的定义自行推导这个等式。

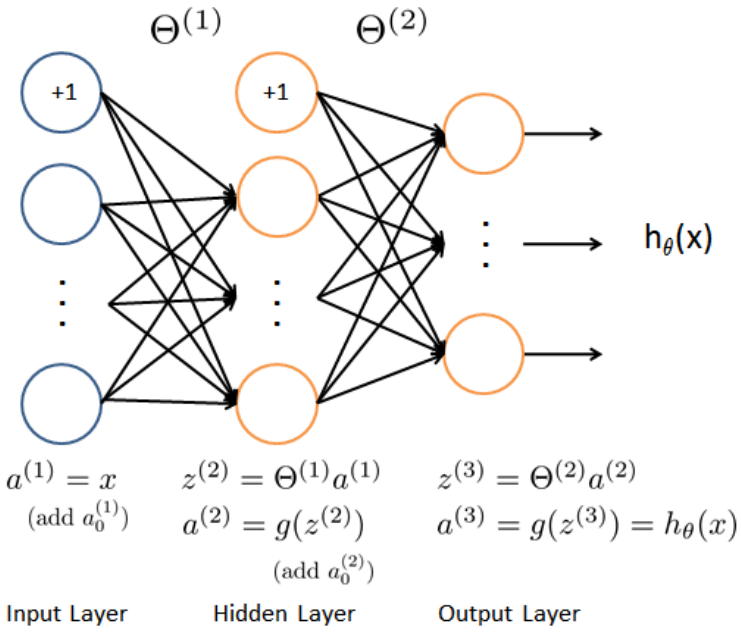

Figure 2: Neural network model

```
% Load saved matrices from file
load('ex3weights.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. Similar to the one-vs-all classification strategy, the prediction from the neural network will be the label that has the largest output $(h_\Theta(x))_k$.

Implementation Note: 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. Specifically, 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 you index (and if necessary, transpose) X correctly so that you get $a^{(l)}$ as a column vector.

Once you are done, `ex3_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 97.5%. After that, an interactive 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`.

Part II
Neural Networks: Backpropagation

In the previous part, 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 part.

The training data will be loaded into the variables X and y by the `ex4.m` script. 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
```

5 Feedforward and cost function

5.1 Unregularized 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 (注意到这里的$h_\Theta(x)$同前面的linear regression和logistic regression中的$h_\Theta(x)$的不同, 显然更为复杂) and $K = 10$ is the total number of possible labels. Note that $h_\Theta(x^{(i)})_k = a_k^{(3)}$ 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

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

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 the matrices `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.

5.2 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 the weight decay parameter $\lambda = 1$. You should see that the cost is about 0.383770.

6 Backpropagation

In this section, 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.

6.1 Sigmoid gradient

You will first implement the sigmoid gradient function. As described before, the gradient for the sigmoid function can be computed as $g'(z) = g(z)(1-g(z))$. 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 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 $[-\epsilon_{init}, \epsilon_{init}]$. 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.

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 Backpropagation

Now, you will implement the backpropagation algorithm. Recall that the intuition behind the backpropagation algorithm is as follows. Given a training example $(x^{(t)}, y^{(t)})$, we will first run a "forward pass" to compute all the activations throughout the network, including the output value of the hypothesis $h_\Theta(x)$. Then, for each node j in layer l, we would like to compute an"error term" $\delta_j^{(l)}$ that measures how much that node was "responsible" for any errors in our output.

For an output node, we can directly measure the difference between the network's activation and the true target value, and use that to define $\delta_j^{(3)}$ (since layer 3 is the output layer). For the hidden units, you will compute $\delta_j^{(l)}$ based on a weighted average of the error terms of the nodes in layer $(l+1)$.

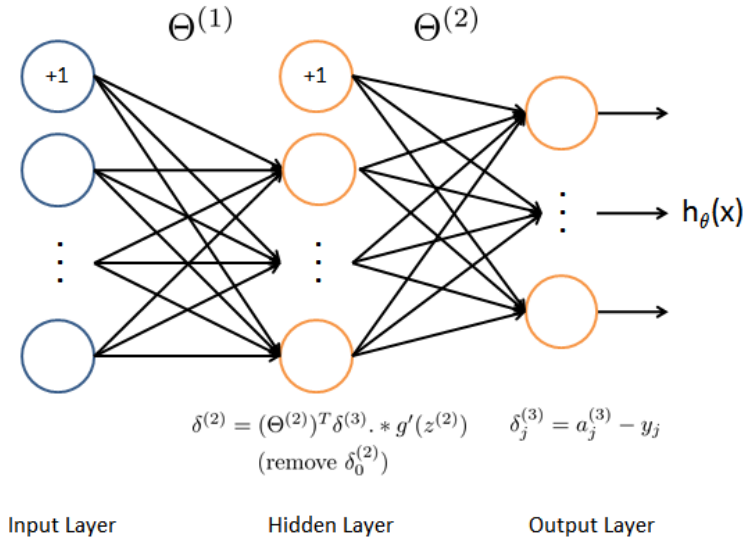

Figure 3: Backpropagation Updates

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

[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 units in the layers adjacent to $\Theta^{(l)}$.

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 Octave/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 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
$$\delta^{(2)} = (\Theta^{(2)})^T \delta^{(3)} .* g'(z^{(2)})$$

4. Accumulate the gradient from this example using the following formula. Note that you should skip or remove $\delta_0^{(0)}$. In Octave/Matlab, removing $\delta_0^{(0)}$ 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 $\frac{1}{m}$
$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = D_{ij}^{(l)} = \frac{1}{m} \Delta_{ij}^{(l)}$$

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).

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.

6.4 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 $\frac{\partial}{\partial \theta_i} J(\theta)$; you'd like to check if f_i is outputting correct derivative values. Let

$$\theta^{(i+)} = \theta + \begin{bmatrix} 0 \\ 0 \\ \vdots \\ \epsilon \\ \vdots \\ 0 \end{bmatrix}, \text{ and } \theta^{(i-)} = \theta - \begin{bmatrix} 0 \\ 0 \\ \vdots \\ \epsilon \\ \vdots \\ 0 \end{bmatrix}$$

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 $\epsilon = 10^{-4}$, 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 con dent that your gradient computations are correct, you should turn o 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).

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.

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

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

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

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

$$\Theta^{(l)} = \begin{bmatrix} \Theta_{10}^{(l)} & \Theta_{11}^{(l)} & \cdots \\ \Theta_{20}^{(l)} & \Theta_{21}^{(l)} & \\ \vdots & & \ddots \end{bmatrix}$$

.

Somewhat confusingly, indexing in Octave/Matlab starts from 1 (for both i and j), thus `Theta1(2, 1)` actually corresponds to $\Theta_{20}^{(l)}$ (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 `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.

6.6 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.

7 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 nd 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

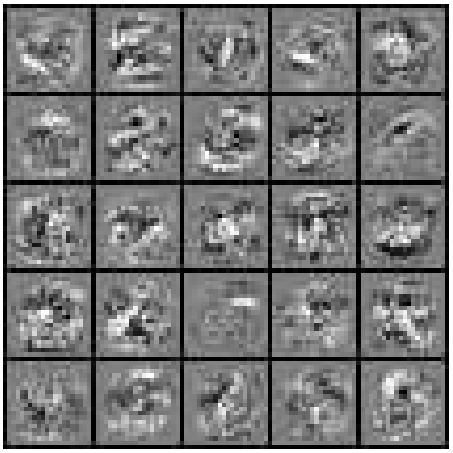

Figure 4: Visualization of Hidden Units.

by the hidden unit is to reshape this 400 dimensional vector into a 20×20 image and display it[3]. 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 corresponds roughly to detectors that look for strokes and other patterns in the input.

7.1 Choosing the regularization parameter λ

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 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 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`.

[3]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 (i.e., $\|x\|_2 \leq 1$).