Programming Exercise 4:

Neural Networks Learning

Machine Learning

Introduction

In this exercise, you will implement the backpropagation algorithm for neural networks and apply it to the task of hand-written digit recognition. Before starting on the programming exercise, we strongly recommend watching the video lectures and completing the review questions for the associated topics.

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.

Files included in this exercise

ex4.py – Python script that steps you through the exercise
ex4data1.mat - Training set of hand-written digits

ex4weights.mat - Neural network parameters for exercise 4

displayData.py - Function to help visualize the dataset

sigmoid.py - Sigmoid function

computeNumericalGradient.py - Numerically compute gradients
checkNNGradients.py - Function to help check your gradients
debugInitializeWeights.py - Function for initializing weights
predict.py - Neural network prediction function

[⋆] sigmoidGradient.py - Compute the gradient of the sigmoid function

[⋆] randInitializeWeights.py - Randomly initialize weights

[⋆] nnCostFunction.py - Neural network cost function

⋆ indicates files you will need to complete

Throughout the exercise, you will be using the script ex4.py. 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.py, will help you step through this exercise.

1.1 Visualizing the data

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

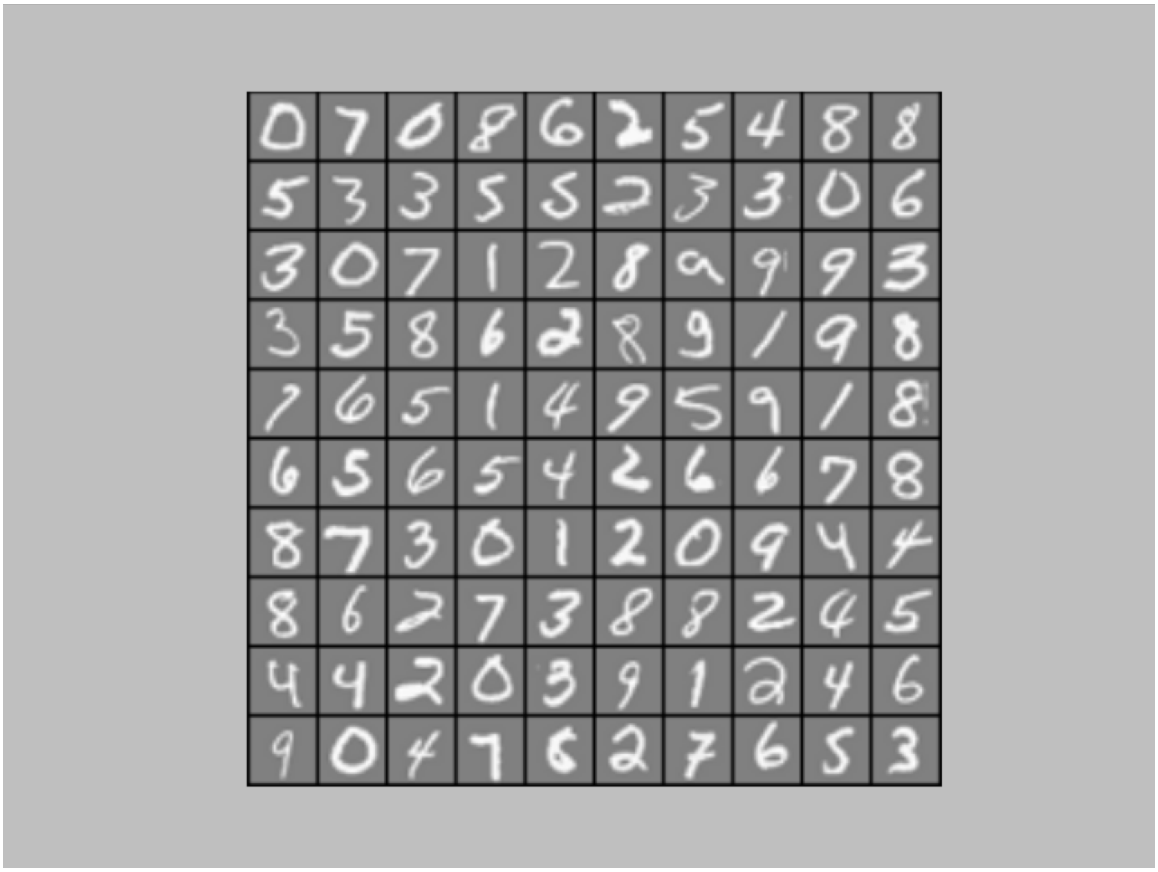

Figure 1: Examples from the dataset

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.

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

The second part of the training set is a 5000-dimensional vector y that contains labels for the training set. Pay special attention that 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 values of 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 X and y by the ex4.py 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.py into Theta1 and Theta2. The parameters have dimensions that are sized for a neural network with 25 units in the second layer and 10 output units (corresponding to the 10 digit classes).

```
# Load the weights into variables Theta1 and Theta2

data = scipy.io.loadmat('ex3weights.mat')

Theta1 = data['Theta1']

Theta2 = data['Theta2']
```

```
# Theta1 has size 25*401

# Theta2 has size 10*26
```

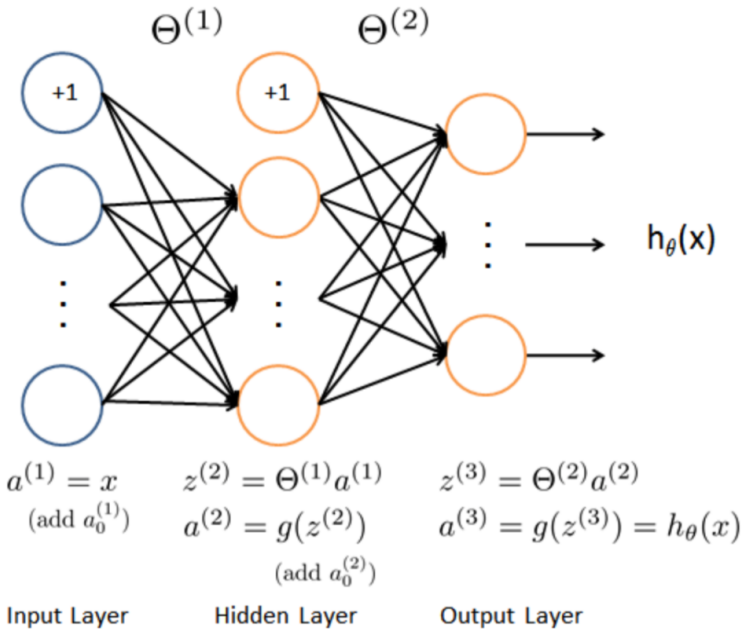

Figure 2: Neural network model.

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.py to return the cost.

Recall that the cost function for the neural network (without regularization) is

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

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 $h_\theta(x^{(i)})_k = a_k^{(i)}$ is the activation (output

value) of the k-th output unit. Also, recall that whereas the original labels (in the variable y) were 1, 2, ..., 10, for the purpose of training a neural network, we need to recode the labels as vectors containing only values 0 or 1, so that

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

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 ≥ 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 × 1 vector.) When you complete the code in nnCostFunction.py, 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.py will call your nnCostFunction using the loaded set of parameters for Theta1 and Theta2. You should see that the cost is about 0.287629.

1.4 Regularized cost function

The cost function for neural networks with regularization is given by

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

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.py and then later add the cost for the regularization terms.

Once you are done, ex4.py 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.

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.py 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 minimize(method='CG').

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

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

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

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 $\Theta^{(1)}$ uniformly in the range $[-\varepsilon_{init}, \varepsilon_{init}]$. You should use $\varepsilon_{init} = 0.12$.[2] This range of values ensures that the parameters are kept small and makes the learning more efficient.

2 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 $\Theta^{(l)}$.

Your job is to complete randInitializeWeights.py 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 = np.random.rand(L_out, 1 + L_in)*(2*epsilon_init) - epsilon_init
```

2.3 Backpropagation

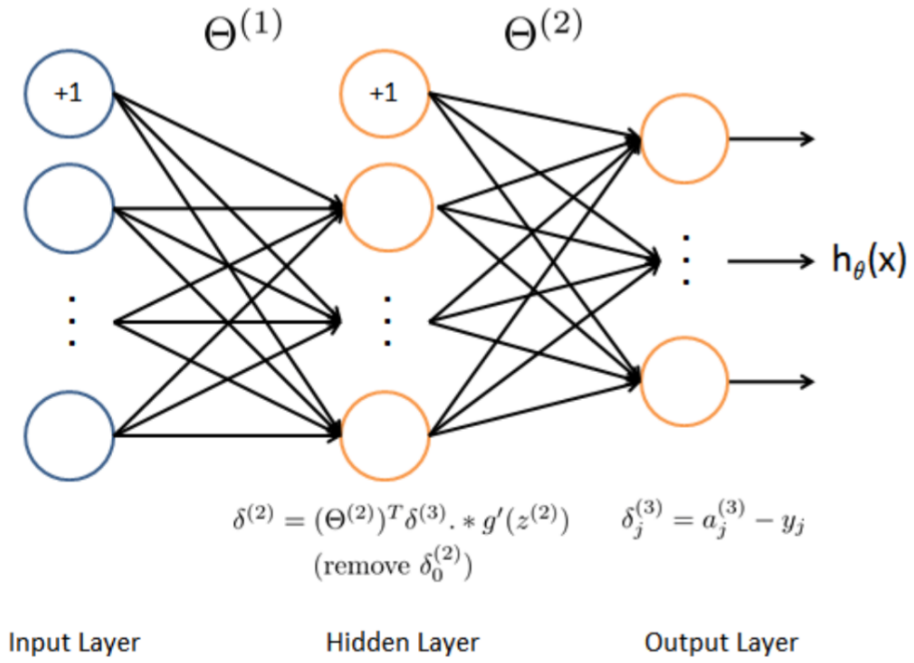

Figure 3: Backpropagation Updates.

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 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 $(l+1)$.

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 tth iteration performing the calculation on the tth 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 layers 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.
2. For each output unit k in layer 3 (the output layer), set

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

where $y_k \in \{0,1\}$ includes 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 (explained in the previous programming exercise).

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 $\delta_0^{(2)}$.

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

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

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

After you have implemented the backpropagation algorithm, the script ex4.py 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 $\frac{\partial}{\partial \theta_i} J(\theta)$;you'd like to check if f_i is outputting correct derivative values.

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

So, $\theta^{(i+)}$ is the same as θ, except its i-th element has been incremented by ϵ. Similarly, $\theta^{(i-)}$ is the corresponding vector with the i-th element decreased by ϵ. You can now numerically verify $f_i(\theta)$'s correctness by checking, for each i, that:

$$f_i(\theta) \approx \frac{J\left(\theta^{(i+)}\right) - J\left(\theta^{(i-)}\right)}{2\epsilon}.$$

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.py. 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.py 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.py function to check if your gradient implementations for the other exercises are correct too (e.g., logistic regression's cost function).

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

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

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

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

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

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

2.6 Learning parameters using minimize(method='CG')

After you have successfully implemented the neural network cost function and gradient computation, the next step of the ex4.py script will use minimize(method='CG') to learn a good set of parameters.

After the training completes, the ex4.py 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 95.8% (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 $\left(a_i^{(l)}\right)$ close to 1). For the neural network you trained, notice that the ith row of $\Theta^{(1)}$

is a 401-dimensional vector that represents the parameter for the ith

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.py 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 unit corresponds roughly to detectors that look for strokes and other patterns in the input.

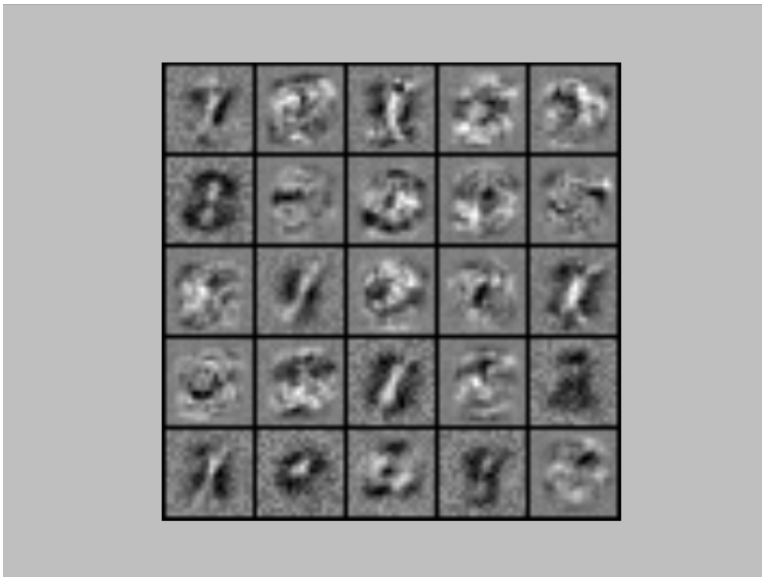

Figure 4: Visualization of Hidden Units.

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 and number of training steps (the maxiter option when using minimize(method='CG')).

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 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 and maxiter.