Machine Learning: Programming Exercise 4

Neural Networks Learning

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

Files needed for this exercise

- `ex4.mlx` - MATLAB Live Script that steps you through the exercise
- `ex4data1.mat` - Training set of hand-written digits
- `ex4weights.mat` - Neural network parameters for exercise 4
- `submit.m` - Submission script that sends your solutions to our servers
- `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***

Confirm that your Current Folder is set correctly

Click into this section, then click the 'Run section' button above. This will execute the `dir` command below to list the files in your Current Folder. The output should contain all of the files listed above and the 'lib' folder. If it does not, right-click the 'ex4' folder and select 'Open' before proceding or see the instructions in `README.mlx` for more details.

```
dir
```

```
.                           debugInitializeWeights.m    ex4data1.mat                nnCostFunction.m
..                          displayData.m               ex4weights.mat              predict.m
checkNNGradients.m          ex4.mlx                     fmincg.m                    randInitializeWeights.
computeNumericalGradient.m  ex4_companion.mlx           lib                         sigmoid.m
```

Before you begin

The workflow for completing and submitting the programming exercises in MATLAB Online differs from the original course instructions. Before beginning this exercise, make sure you have read through the instructions in `README.mlx` which is included with the programming exercise files. `README` also contains solutions to the many common issues you may encounter while completing and submitting the exercises in MATLAB Online. Make sure you are following instructions in `README` and have checked for an existing solution before seeking help on the discussion forums.

Table of Contents

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.

1.1 Visualizing the data

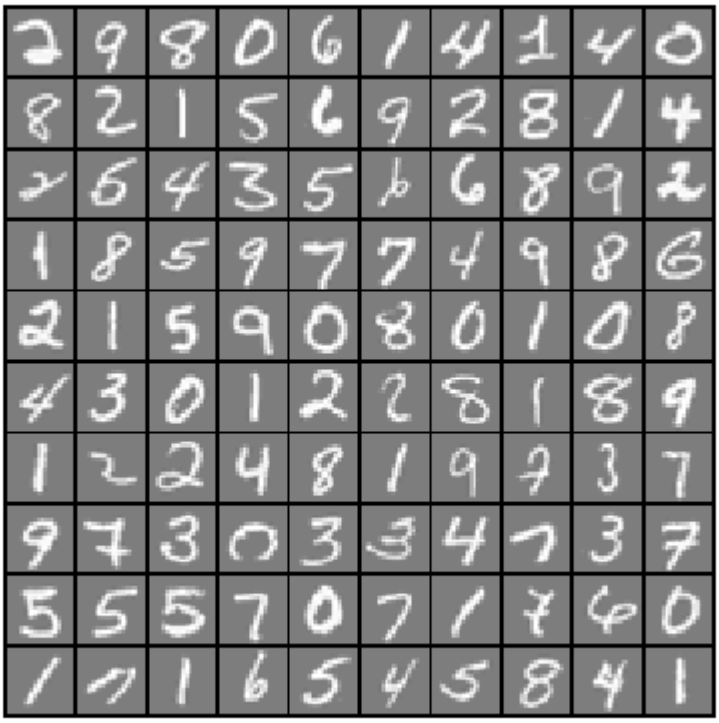
The code below will load the data and display it on a 2-dimensional plot (Figure 1) by calling the function `displayData`. This is the same dataset that you used in the previous exercise. Run the code below to load the training data into the variables `X` and `y`.

Figure 1: Examples from the dataset

```
load('ex4data1.mat');
m = size(X, 1);

% Randomly select 100 data points to display
sel = randperm(size(X, 1));
sel = sel(1:100);

displayData(X(sel, :));
```



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.

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

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 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 values of digit images. Since the images are of size 20 x 20, this gives us 400 input layer units (not counting the extra bias unit which always outputs +1).

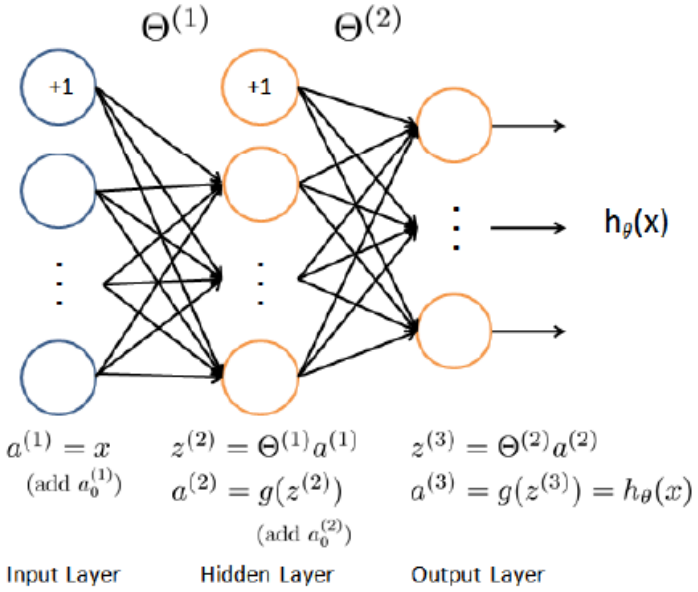


Figure 2: Neural network model.

You have been provided with a set of network parameters $(\Theta^{(1)}, \Theta^{(2)})$ already trained by us. These are stored in `ex4weights.mat`. Run the code below to load them 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
load('ex4weights.mat');
```

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 $h_\theta(x^{(i)})_k = a_k^{(3)}$ is the activation (output value) of the *k*-th output unit. Also, recall that whereas the original labels (in the variable y) were $1, 2, \ldots, 10,$ for the purpose of training a neural network, we need to recode the labels as vectors containing only values 0 or 1, so that

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

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* x 1 vector.) When you complete the code in `nnCostFunction.m`, you will need to add the column of 1's to the `X` matrix. The parameters for each unit in the neural network is represented in `Theta1` and `Theta2` as one row. 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. We suggest implementing the feedforward cost *without* regularization first so that it will be easier for you to debug. Later, you will get to implement the regularized cost.

Once you are done, run the code below to call your `nnCostFunction` using the loaded set of parameters for `Theta1` and `Theta2`. You should see that the cost is about 0.287629.

```
input_layer_size  = 400;  % 20x20 Input Images of Digits
hidden_layer_size = 25;   % 25 hidden units
num_labels = 10;          % 10 labels, from 1 to 10 (note that we have mapped "0" to la

% Unroll parameters
nn_params = [Theta1(:) ; Theta2(:)];

% Weight regularization parameter (we set this to 0 here).
lambda = 0;

J = nnCostFunction(nn_params, input_layer_size, hidden_layer_size, num_labels, X, y, la

fprintf('Cost at parameters (loaded from ex4weights): %f', J);
```

```
Cost at parameters (loaded from ex4weights): 0.287629
```

You should now submit your solutions. Enter **`submit`** *at the command prompt, then enter or confirm your login and token when prompted.*

1.4 Regularized cost function

The cost function for neural networks with regularization is given by

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

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.m` and then later add the cost for the regularization terms. Once you are done, run the code below to 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.

```
% Weight regularization parameter (we set this to 1 here).
lambda = 1;

J = nnCostFunction(nn_params, input_layer_size, hidden_layer_size, num_labels, X, y, la
fprintf('Cost at parameters (loaded from ex4weights): %f', J);
```

```
Cost at parameters (loaded from ex4weights): 0.383770
```

You should now submit your solutions. Enter/confirm your login and token in the command window when prompted.

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 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.

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

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

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

```
% Call your sigmoidGradient function
sigmoidGradient(0)
```

```
ans = 0.2500
```

You should now submit your solutions. Enter **`submit`** *at the command prompt, then enter or confirm your login and token when prompted.*

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 $[-\epsilon_{int}, \epsilon_{init}]$. You should use $\epsilon_{init} = 0.12$*. This range of values ensures that the parameters are kept small and makes the learning more efficient.

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;
```

When you are done, run the code below to call `randInitialWeights` and initialize the neural network parameters.

```
initial_Theta1 = randInitializeWeights(input_layer_size, hidden_layer_size);
initial_Theta2 = randInitializeWeights(hidden_layer_size, num_labels);

% Unroll parameters
initial_nn_params = [initial_Theta1(:) ; initial_Theta2(:)];
```

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

You do not need to submit any code for this part of the exercise.

2.3 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 $(l+1)$.

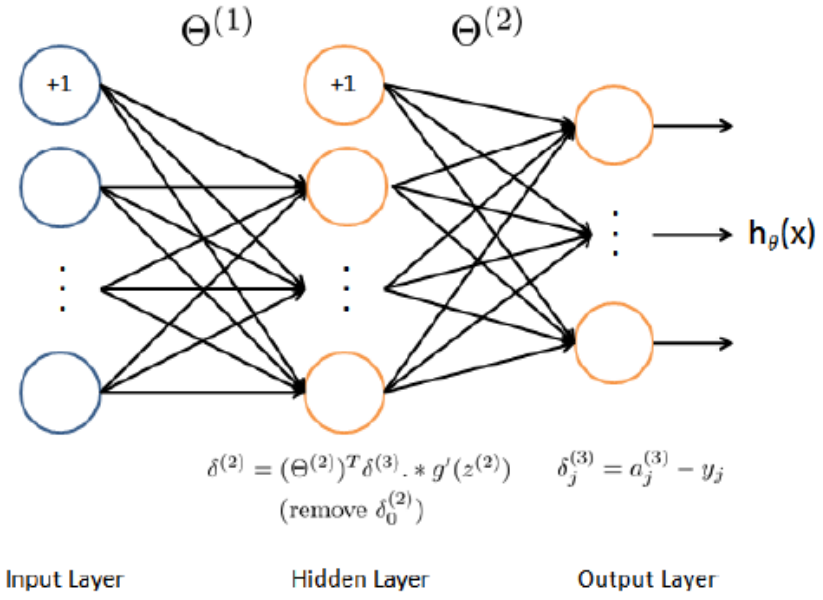


Figure 3: Backpropagation Updates.

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 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 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: $\Delta^{(l)} = \Delta^{(l)} + \delta^{(l+1)}(a^{(l)})^T$. Note that you should skip or remove $\delta_0^{(2)}$. In MATLAB, removing $\delta_0^{(2)}$ corresponds to `delta_2 = delta_2(2:end)`.
5. Obtain the (unregularized) gradient for the neural network cost function by dividing the accumulated gradients by $\frac{1}{m}$: $\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = D_{ij}^{(l)} = \frac{1}{m} \Delta_{ij}^{(l)}$

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.

After you have implemented the backpropagation algorithm, the code in the next section will 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.

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

So, $\theta^{(i+)}$ is the same as θ, except its *i*-th element has been incremented by #. Similarly, $\theta^{(i-)}$is the corresponding vector with the *i*-th element decreased by #. 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 signicant 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.

The code below 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 dierence that is less than 1e-9.

```
checkNNGradients;
```

```
    -0.0093   -0.0093
     0.0089    0.0089
    -0.0084   -0.0084
     0.0076    0.0076
    -0.0067   -0.0067
    -0.0000   -0.0000
     0.0000    0.0000
    -0.0000   -0.0000
     0.0000    0.0000
    -0.0000   -0.0000
    -0.0002   -0.0002
     0.0002    0.0002
    -0.0003   -0.0003
     0.0003    0.0003
    -0.0004   -0.0004
    -0.0001   -0.0001
     0.0001    0.0001
    -0.0001   -0.0001
     0.0002    0.0002
    -0.0002   -0.0002
     0.3145    0.3145
     0.1111    0.1111
     0.0974    0.0974
     0.1641    0.1641
     0.0576    0.0576
     0.0505    0.0505
     0.1646    0.1646
     0.0578    0.0578
     0.0508    0.0508
     0.1583    0.1583
     0.0559    0.0559
     0.0492    0.0492
     0.1511    0.1511
     0.0537    0.0537
     0.0471    0.0471
     0.1496    0.1496
     0.0532    0.0532
     0.0466    0.0466

The above two columns you get should be very similar.
(Left-Your Numerical Gradient, Right-Analytical Gradient)

If your backpropagation implementation is correct, then
the relative difference will be small (less than 1e-9).

Relative Difference: 2.2366e-11
```

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 condent 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).

You should now submit your solutions. Enter **`submit`** *at the command prompt, then enter or confirm your login and token when prompted.*

2.5 Regularized neural networks

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

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

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

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

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

Somewhat confusingly, indexing in MATLAB starts from 1 (for both *i* and *j*), thus `Theta1(2, 1)` actually corresponds to $\Theta_{2,0}^{(l)}$ (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, run the code below 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.

```
%  Check gradients by running checkNNGradients
lambda = 3;
checkNNGradients(lambda);
```

```
   -0.0093   -0.0093
    0.0089    0.0089
   -0.0084   -0.0084
    0.0076    0.0076
   -0.0067   -0.0067
   -0.0168   -0.0168
    0.0394    0.0394
    0.0593    0.0593
    0.0248    0.0248
   -0.0327   -0.0327
   -0.0602   -0.0602
   -0.0320   -0.0320
    0.0249    0.0249
    0.0598    0.0598
    0.0386    0.0386
   -0.0174   -0.0174
   -0.0576   -0.0576
   -0.0452   -0.0452
    0.0091    0.0091
    0.0546    0.0546
    0.3145    0.3145
    0.1111    0.1111
    0.0974    0.0974
    0.1187    0.1187
    0.0000    0.0000
    0.0337    0.0337
    0.2040    0.2040
    0.1171    0.1171
    0.0755    0.0755
    0.1257    0.1257
   -0.0041   -0.0041
    0.0170    0.0170
    0.1763    0.1763
    0.1131    0.1131
    0.0862    0.0862
    0.1323    0.1323
   -0.0045   -0.0045
    0.0015    0.0015

The above two columns you get should be very similar.
(Left-Your Numerical Gradient, Right-Analytical Gradient)

If your backpropagation implementation is correct, then
the relative difference will be small (less than 1e-9).

Relative Difference: 2.17629e-11
```

```
% Also output the costFunction debugging value
% This value should be about 0.576051
debug_J  = nnCostFunction(nn_params, input_layer_size, hidden_layer_size, num_labels, X
fprintf('Cost at (fixed) debugging parameters (w/ lambda = 3): %f', debug_J);
```

```
Cost at (fixed) debugging parameters (w/ lambda = 3): 0.576051
```

You should now submit your solutions. Enter **`submit`** *at the command prompt, then enter or confirm your login and token when prompted.*

2.6 Learning parameters using `fmincg`

After you have successfully implemented the neural network cost function and gradient computation, run the code below to use `fmincg` to learn a good set of parameters. After the training completes, the code will 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.

```
options = optimset('MaxIter', 50);
lambda = 1;

% Create "short hand" for the cost function to be minimized
costFunction = @(p) nnCostFunction(p, input_layer_size, hidden_layer_size, num_labels,

% Now, costFunction is a function that takes in only one argument (the
% neural network parameters)
[nn_params, ~] = fmincg(costFunction, initial_nn_params, options);
```

```
Iteration     1 | Cost: 3.305808e+00
Iteration     2 | Cost: 3.244591e+00
Iteration     3 | Cost: 3.224528e+00
Iteration     4 | Cost: 2.728529e+00
Iteration     5 | Cost: 2.392188e+00
Iteration     6 | Cost: 2.104558e+00
Iteration     7 | Cost: 1.878274e+00
Iteration     8 | Cost: 1.649402e+00
Iteration     9 | Cost: 1.525610e+00
Iteration    10 | Cost: 1.385040e+00
Iteration    11 | Cost: 1.262909e+00
Iteration    12 | Cost: 1.191624e+00
Iteration    13 | Cost: 1.125654e+00
Iteration    14 | Cost: 1.037664e+00
Iteration    15 | Cost: 9.941855e-01
Iteration    16 | Cost: 9.601111e-01
Iteration    17 | Cost: 9.172971e-01
Iteration    18 | Cost: 8.881248e-01
Iteration    19 | Cost: 8.730900e-01
Iteration    20 | Cost: 8.280842e-01
Iteration    21 | Cost: 7.868730e-01
Iteration    22 | Cost: 7.475196e-01
Iteration    23 | Cost: 7.077484e-01
Iteration    24 | Cost: 6.858086e-01
Iteration    25 | Cost: 6.619652e-01
Iteration    26 | Cost: 6.342661e-01
Iteration    27 | Cost: 6.199128e-01
Iteration    28 | Cost: 6.152493e-01
Iteration    29 | Cost: 5.962388e-01
Iteration    30 | Cost: 5.891749e-01
Iteration    31 | Cost: 5.853484e-01
Iteration    32 | Cost: 5.741881e-01
Iteration    33 | Cost: 5.673138e-01
Iteration    34 | Cost: 5.579343e-01
Iteration    35 | Cost: 5.490590e-01
Iteration    36 | Cost: 5.421337e-01
Iteration    37 | Cost: 5.358088e-01
Iteration    38 | Cost: 5.304875e-01
Iteration    39 | Cost: 5.238108e-01
Iteration    40 | Cost: 5.148360e-01
```

```
Iteration    41 | Cost: 5.110084e-01
Iteration    42 | Cost: 5.081119e-01
Iteration    43 | Cost: 5.040794e-01
Iteration    44 | Cost: 4.989980e-01
Iteration    45 | Cost: 4.969157e-01
Iteration    46 | Cost: 4.962145e-01
Iteration    47 | Cost: 4.932027e-01
Iteration    48 | Cost: 4.912594e-01
Iteration    49 | Cost: 4.900989e-01
Iteration    50 | Cost: 4.850150e-01
```

```
% Obtain Theta1 and Theta2 back from nn_params
Theta1 = reshape(nn_params(1:hidden_layer_size * (input_layer_size + 1)), hidden_layer_
Theta2 = reshape(nn_params((1 + (hidden_layer_size * (input_layer_size + 1))):end), num

pred = predict(Theta1, Theta2, X);
fprintf('\nTraining Set Accuracy: %f\n', mean(double(pred == y)) * 100);
```

```
Training Set Accuracy: 95.160000
```

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 x 20 image and display it.*

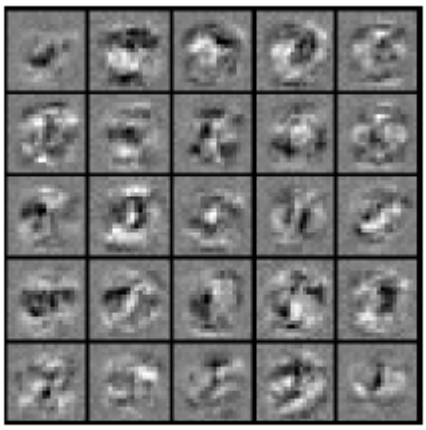
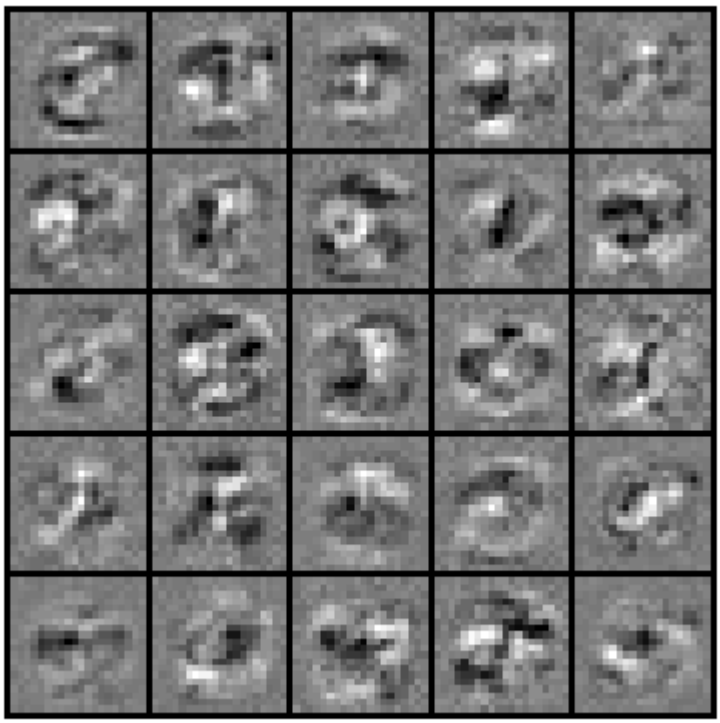


Figure 4: Visualization of Hidden Units.

The code below 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.

```
% Visualize Weights
displayData(Theta1(:, 2:end));
```



**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.,* $\|x\|_2 \leq 1$*).*

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 `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 do 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`.

```
% Change lambda and MaxIter to see how it affects the result
lambda = 0.3;
MaxIter = 150;

options = optimset('MaxIter', MaxIter);

% Create "short hand" for the cost function to be minimized
```

```
costFunction = @(p) nnCostFunction(p,input_layer_size, hidden_layer_size, num_labels, X

% Now, costFunction is a function that takes in only one argument (the neural network p
[nn_params, ~] = fmincg(costFunction, initial_nn_params, options);
```

```
Iteration     1 | Cost: 3.301983e+00
Iteration     2 | Cost: 3.240785e+00
Iteration     3 | Cost: 3.220654e+00
Iteration     4 | Cost: 2.720471e+00
Iteration     5 | Cost: 2.380253e+00
Iteration     6 | Cost: 2.086888e+00
Iteration     7 | Cost: 1.866808e+00
Iteration     8 | Cost: 1.638195e+00
Iteration     9 | Cost: 1.506057e+00
Iteration    10 | Cost: 1.366032e+00
Iteration    11 | Cost: 1.231452e+00
Iteration    12 | Cost: 1.149599e+00
Iteration    13 | Cost: 1.041395e+00
Iteration    14 | Cost: 9.860013e-01
Iteration    15 | Cost: 9.437212e-01
Iteration    16 | Cost: 8.831735e-01
Iteration    17 | Cost: 8.579205e-01
Iteration    18 | Cost: 8.427001e-01
Iteration    19 | Cost: 7.905778e-01
Iteration    20 | Cost: 7.452837e-01
Iteration    21 | Cost: 7.007467e-01
Iteration    22 | Cost: 6.471476e-01
Iteration    23 | Cost: 6.280961e-01
Iteration    24 | Cost: 6.024878e-01
Iteration    25 | Cost: 5.892976e-01
Iteration    26 | Cost: 5.829625e-01
Iteration    27 | Cost: 5.645135e-01
Iteration    28 | Cost: 5.561900e-01
Iteration    29 | Cost: 5.467557e-01
Iteration    30 | Cost: 5.372367e-01
Iteration    31 | Cost: 5.199188e-01
Iteration    32 | Cost: 5.000246e-01
Iteration    33 | Cost: 4.793058e-01
Iteration    34 | Cost: 4.677513e-01
Iteration    35 | Cost: 4.586934e-01
Iteration    36 | Cost: 4.417482e-01
Iteration    37 | Cost: 4.286718e-01
Iteration    38 | Cost: 4.233509e-01
Iteration    39 | Cost: 4.180942e-01
Iteration    40 | Cost: 4.112258e-01
Iteration    41 | Cost: 4.066567e-01
Iteration    42 | Cost: 4.022555e-01
Iteration    43 | Cost: 3.952143e-01
Iteration    44 | Cost: 3.898429e-01
Iteration    45 | Cost: 3.881583e-01
Iteration    46 | Cost: 3.834527e-01
Iteration    47 | Cost: 3.776924e-01
Iteration    48 | Cost: 3.708190e-01
Iteration    49 | Cost: 3.656852e-01
Iteration    50 | Cost: 3.629993e-01
Iteration    51 | Cost: 3.572680e-01
Iteration    52 | Cost: 3.489970e-01
Iteration    53 | Cost: 3.406556e-01
Iteration    54 | Cost: 3.368727e-01
Iteration    55 | Cost: 3.362117e-01
Iteration    56 | Cost: 3.339604e-01
Iteration    57 | Cost: 3.324136e-01
Iteration    58 | Cost: 3.313074e-01
Iteration    59 | Cost: 3.296817e-01
```

```
Iteration    60 | Cost: 3.281677e-01
Iteration    61 | Cost: 3.266352e-01
Iteration    62 | Cost: 3.235286e-01
Iteration    63 | Cost: 3.192550e-01
Iteration    64 | Cost: 3.097224e-01
Iteration    65 | Cost: 2.966382e-01
Iteration    66 | Cost: 2.897823e-01
Iteration    67 | Cost: 2.884965e-01
Iteration    68 | Cost: 2.841433e-01
Iteration    69 | Cost: 2.832676e-01
Iteration    70 | Cost: 2.823501e-01
Iteration    71 | Cost: 2.812302e-01
Iteration    72 | Cost: 2.781653e-01
Iteration    73 | Cost: 2.747287e-01
Iteration    74 | Cost: 2.713446e-01
Iteration    75 | Cost: 2.677658e-01
Iteration    76 | Cost: 2.661008e-01
Iteration    77 | Cost: 2.657772e-01
Iteration    78 | Cost: 2.618590e-01
Iteration    79 | Cost: 2.608162e-01
Iteration    80 | Cost: 2.604301e-01
Iteration    81 | Cost: 2.599099e-01
Iteration    82 | Cost: 2.590716e-01
Iteration    83 | Cost: 2.560278e-01
Iteration    84 | Cost: 2.523629e-01
Iteration    85 | Cost: 2.510464e-01
Iteration    86 | Cost: 2.509886e-01
Iteration    87 | Cost: 2.496638e-01
Iteration    88 | Cost: 2.491901e-01
Iteration    89 | Cost: 2.486522e-01
Iteration    90 | Cost: 2.469514e-01
Iteration    91 | Cost: 2.430370e-01
Iteration    92 | Cost: 2.406146e-01
Iteration    93 | Cost: 2.385059e-01
Iteration    94 | Cost: 2.342497e-01
Iteration    95 | Cost: 2.255177e-01
Iteration    96 | Cost: 2.165772e-01
Iteration    97 | Cost: 2.128827e-01
Iteration    98 | Cost: 2.107394e-01
Iteration    99 | Cost: 2.079825e-01
Iteration   100 | Cost: 2.069266e-01
Iteration   101 | Cost: 2.069030e-01
Iteration   102 | Cost: 2.059616e-01
Iteration   103 | Cost: 2.056229e-01
Iteration   104 | Cost: 2.052427e-01
Iteration   105 | Cost: 2.045830e-01
Iteration   106 | Cost: 2.041269e-01
Iteration   107 | Cost: 2.038857e-01
Iteration   108 | Cost: 2.032791e-01
Iteration   109 | Cost: 2.027183e-01
Iteration   110 | Cost: 2.023796e-01
Iteration   111 | Cost: 2.016734e-01
Iteration   112 | Cost: 2.000854e-01
Iteration   113 | Cost: 1.990504e-01
Iteration   114 | Cost: 1.981079e-01
Iteration   115 | Cost: 1.967009e-01
Iteration   116 | Cost: 1.961368e-01
Iteration   117 | Cost: 1.955403e-01
Iteration   118 | Cost: 1.952025e-01
Iteration   119 | Cost: 1.947970e-01
Iteration   120 | Cost: 1.945320e-01
Iteration   121 | Cost: 1.941636e-01
Iteration   122 | Cost: 1.934487e-01
Iteration   123 | Cost: 1.930432e-01
Iteration   124 | Cost: 1.925531e-01
```

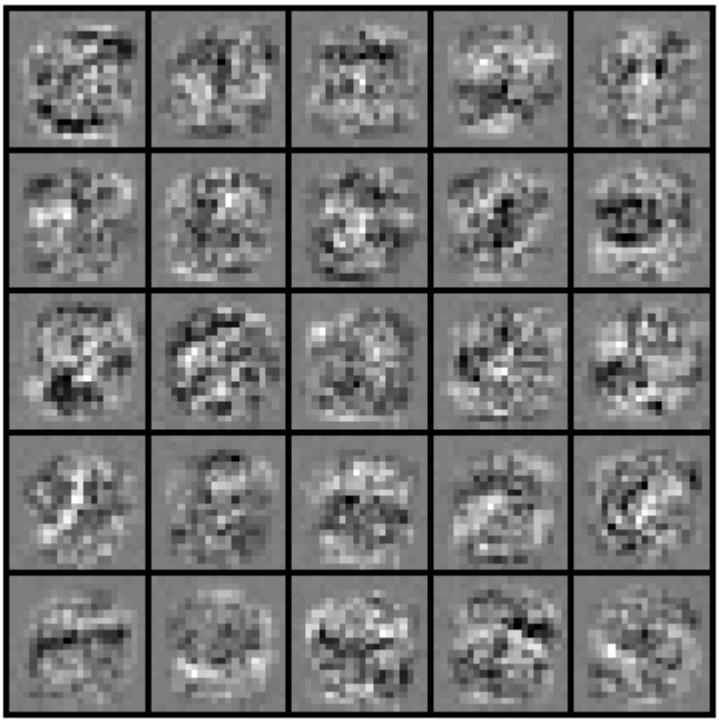
```
Iteration   125 | Cost: 1.922166e-01
Iteration   126 | Cost: 1.918787e-01
Iteration   127 | Cost: 1.908447e-01
Iteration   128 | Cost: 1.900198e-01
Iteration   129 | Cost: 1.890584e-01
Iteration   130 | Cost: 1.876851e-01
Iteration   131 | Cost: 1.870569e-01
Iteration   132 | Cost: 1.868665e-01
Iteration   133 | Cost: 1.866069e-01
Iteration   134 | Cost: 1.864487e-01
Iteration   135 | Cost: 1.863254e-01
Iteration   136 | Cost: 1.860582e-01
Iteration   137 | Cost: 1.858473e-01
Iteration   138 | Cost: 1.857686e-01
Iteration   139 | Cost: 1.856571e-01
Iteration   140 | Cost: 1.853994e-01
Iteration   141 | Cost: 1.850809e-01
Iteration   142 | Cost: 1.847397e-01
Iteration   143 | Cost: 1.844635e-01
Iteration   144 | Cost: 1.841400e-01
Iteration   145 | Cost: 1.832333e-01
Iteration   146 | Cost: 1.822319e-01
Iteration   147 | Cost: 1.812408e-01
Iteration   148 | Cost: 1.797780e-01
Iteration   149 | Cost: 1.789767e-01
Iteration   150 | Cost: 1.778561e-01
```

```
% Obtain Theta1 and Theta2 back from nn_params
Theta1 = reshape(nn_params(1:hidden_layer_size * (input_layer_size + 1)), hidden_layer_
Theta2 = reshape(nn_params((1 + (hidden_layer_size * (input_layer_size + 1))):end), num

pred = predict(Theta1, Theta2, X);
fprintf('\nTraining Set Accuracy: %f\n', mean(double(pred == y)) * 100);
```

```
Training Set Accuracy: 99.640000
```

```
% Visualize Weights
displayData(Theta1(:, 2:end));
```



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

Submission and Grading

After completing various parts of the assignment, be sure to use the submit function system to submit your solutions to our servers. The following is a breakdown of how each part of this exercise is scored.

Part	Submitted File	Points
Feedforward and Cost Function	`nnCostFunction.m`	30 points
Regularized Cost Function	`nnCostFunction.m`	15 points
Sigmoid Gradient	`sigmoidGradient.m`	5 points
Neural Net Gradient Function (Backpropagation)	`nnCostFunction.m`	40 points
Regularized Gradient	`nnCostFunction.m`	10 points
Total Points		100 points

You are allowed to submit your solutions multiple times, and we will take only the highest score into consideration.