Handwritten Digit Recognition

```
%load data into X and y
load('ex3data1.mat');
```

There are 5000 training examples in the dataset, where each example is a 20 pixel by 20 pixel grayscale image of the digit . This 20 by 20 pixels is unrolled into a 400 dimensional vector.Thus X is a 5000 by 400 matrix

Visualization

The code randomly selects 100 examples from the training examples

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

displayData(sel);
```

1.Regularized Logistics Regression

We will be using multiple one vs all logistic regression models to build a multi-class classifier. Since there are 10 classes , we will train 10 separate logistic classifiers.

```
theta_t = [-2; -1; 1; 2];
X_t = [ones(5,1) reshape(1:15,5,3)/10];
y_t = ([1;0;1;0;1] >= 0.5);
lambda_t = 3;
[J, grad] = lrCostFunction(theta_t, X_t, y_t, lambda_t);

fprintf('Cost: %f | Expected cost: 2.534819\n',J);
fprintf('Gradients:\n'); fprintf('%f\n',grad);
fprintf('Expected gradients:\n 0.146561\n -0.548558\n 0.724722\n 1.398003');
```

1.a.One-vs-all Classification

We will implement one-vs-all classification by training multiple regularized logistic regression classifiers, one for each of the 10 classes in our dataset

```
num_labels = 10; % 10 labels, from 1 to 10
lambda = 0.1;
[all_theta] = oneVsAll(X, y, num_labels, lambda);
```

1.b.One-vs-all Prediction

The 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 $(1, 2, ..., \text{ or } K)$ as the prediction for the input example.

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

2.Neural Networks

Logistic regression being a linear classifier cannot implement complex functions. One can add more features like polynomial features to it but its training becomes expensive .

Our model is a three layered neural network containing one input, one hidden and one output layer.

The parameters $(\Theta^{(1)}, \Theta^{(2)})$ are already provided.

```
% Load the weights into variables Theta1 and Theta2
load('ex4weights.mat');
```

2.a.Feedforward and regularized cost function

```
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 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);
```

3.Backpropagation

We will implement the backpropagation algorithm to compute the gradient for the neural network cost function.

3.a. Sigmoid gradient

We 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

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

3.b.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.

```
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(:)];
```

3.c. Backpropagation

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.

3.d. Gradient checking

Suppose we have a function $f_i(\theta)$ that purportedly computes $\frac{\partial}{\partial \theta_i} J(\theta)$; we'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 ϵ. Similarly, $\theta^{(i-)}$ is the corresponding vector with the i-th element decreased by ϵ. We 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 $\epsilon = 10^{-4}$, we'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).

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;
```

2.5 Regularized neural networks

To account for regularization, we can add this as an additional term after computing the gradients using backpropagation. After we have computed $\Delta_{ij}^{(l)}$ using backpropagation, we 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 we 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)

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