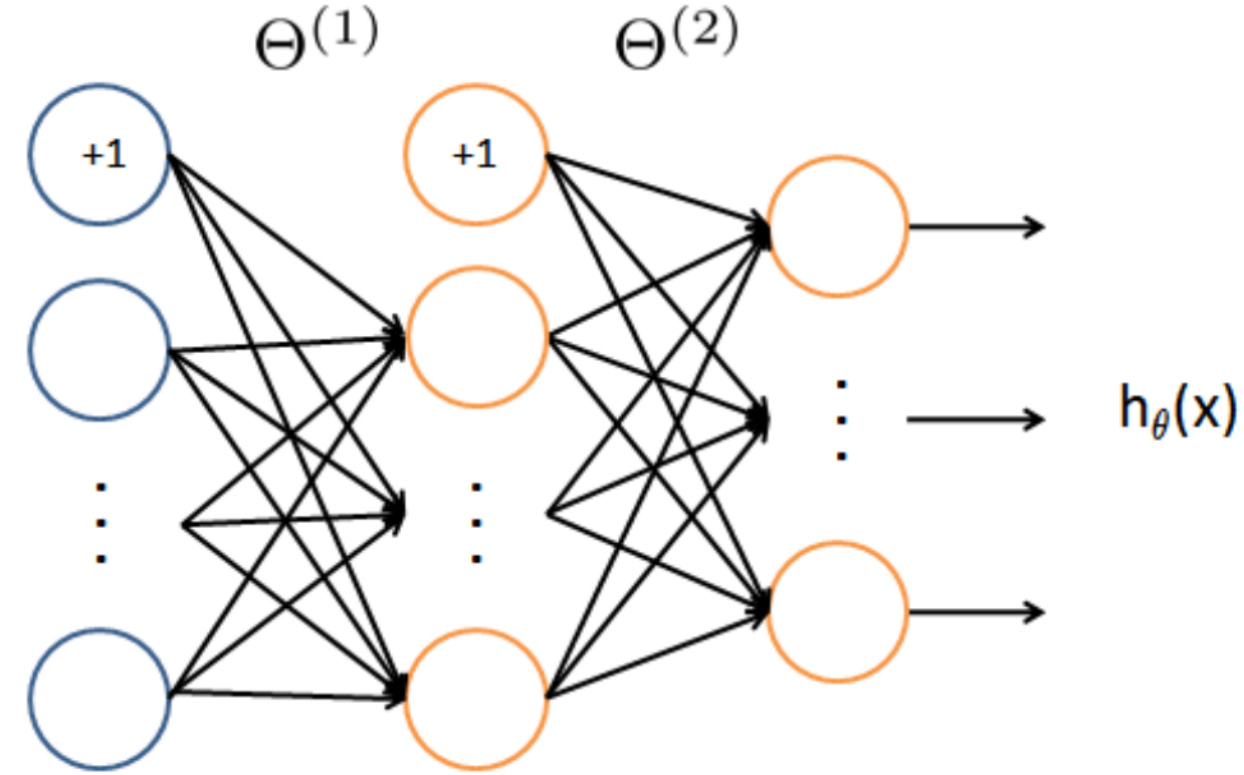
神经网络



$a^{(1)} = x$ (add $a_0^{(1)}$)

$z^{(2)} = \Theta^{(1)} a^{(1)}$

$a^{(2)} = g(z^{(2)})$ (add $a_0^{(2)}$)

$z^{(3)} = \Theta^{(2)} a^{(2)}$

$a^{(3)} = g(z^{(3)}) = h_\theta(x)$

Input Layer | Hidden Layer | Output Layer

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

nnCostFunction.m - Neural network cost function

```
% 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
```

```
function [J grad] = nnCostFunction(nn_params, ...
                                   input_layer_size, ...
                                   hidden_layer_size, ...
                                   num_labels, ...
                                   X, y, lambda)
%NNCOSTFUNCTION Implements the neural network cost function for a two layer
%neural network which performs classification
%   [J grad] = NNCOSTFUNCTON(nn_params, hidden_layer_size, num_labels, ...
%   X, y, lambda) computes the cost and gradient of the neural network. The
%   parameters for the neural network are "unrolled" into the vector
%   nn_params and need to be converted back into the weight matrices.
%
%   The returned parameter grad should be a "unrolled" vector of the
%   partial derivatives of the neural network.
%

% Reshape nn_params back into the parameters Theta1 and Theta2, the weight matrices
% for our 2 layer neural network
Theta1 = reshape(nn_params(1:hidden_layer_size * (input_layer_size + 1)), ...
                 hidden_layer_size, (input_layer_size + 1));

Theta2 = reshape(nn_params((1 + (hidden_layer_size * (input_layer_size + 1))):end), ...
                 num_labels, (hidden_layer_size + 1));

% Setup some useful variables
m = size(X, 1);

% You need to return the following variables correctly
J = 0;
Theta1_grad = zeros(size(Theta1));
Theta2_grad = zeros(size(Theta2));

% ====================== YOUR CODE HERE ======================
% Instructions: You should complete the code by working through the
```

```
%               following parts.
%
% Part 1: Feedforward the neural network and return the cost in the
%         variable J. After implementing Part 1, you can verify that your
%         cost function computation is correct by verifying the cost
%         computed in ex4.m
%
% Part 2: Implement the backpropagation algorithm to compute the gradients
%         Theta1_grad and Theta2_grad. You should return the partial derivatives of
%         the cost function with respect to Theta1 and Theta2 in Theta1_grad and
%         Theta2_grad, respectively. After implementing Part 2, you can check
%         that your implementation is correct by running checkNNGradients
%
%         Note: The vector y passed into the function is a vector of labels
%               containing values from 1..K. You need to map this vector into a
%               binary vector of 1's and 0's to be used with the neural network
%               cost function.
%
%         Hint: We recommend implementing backpropagation using a for-loop
%               over the training examples if you are implementing it for the
%               first time.
%
% Part 3: Implement regularization with the cost function and gradients.
%
%         Hint: You can implement this around the code for
%               backpropagation. That is, you can compute the gradients for
%               the regularization separately and then add them to Theta1_grad
%               and Theta2_grad from Part 2.
%


%Part1
完成前向传递算法
a1 = [ones(m,1) X];
给数据集X（5000*400）的每一组数据加上一个偏差项1，变成5000*401
K = num_labels;
num_labels是输出的元素数，在这里是数字0-9，因此K=10，另外由于对象为数字1-10，因此指定0代表10（但在编程过程中不用考虑这一因素）
layerB = sigmoid(a1 * Theta1');
计算a1（5000*401）和 Theta1（25*401）的sigmoid值（逻辑激励方程），得出第二层（隐藏层）的激励值a2（5000*25）
```

layerB = [ones(m,1) layerB];

给a2加上偏差项1，变成5000*26

layerC = sigmoid(layerB * Theta2');

同上，计算a2（5000*26）和Theta2（10*26）的sigmoid值，得出第三层（输出层）的激励值a3（5000*10）

y_vec = zeros(m,K);

初始化输出结果向量y_vec（5000*10），每一项均为0

for i = 1:m,

  y_vec(i, y(i)) = 1;

end

y是输出结果的标签号（即1，2，3，...，10），现在要将其转换为一个类似于[0,0,0,1,0,0,0,0,0,0]（即y=4）的向量

方法是另y_vec的第i行（即每一组数据）的第y(i)项（例如y=10就表示为[0,0,...,0,0,1]）为1

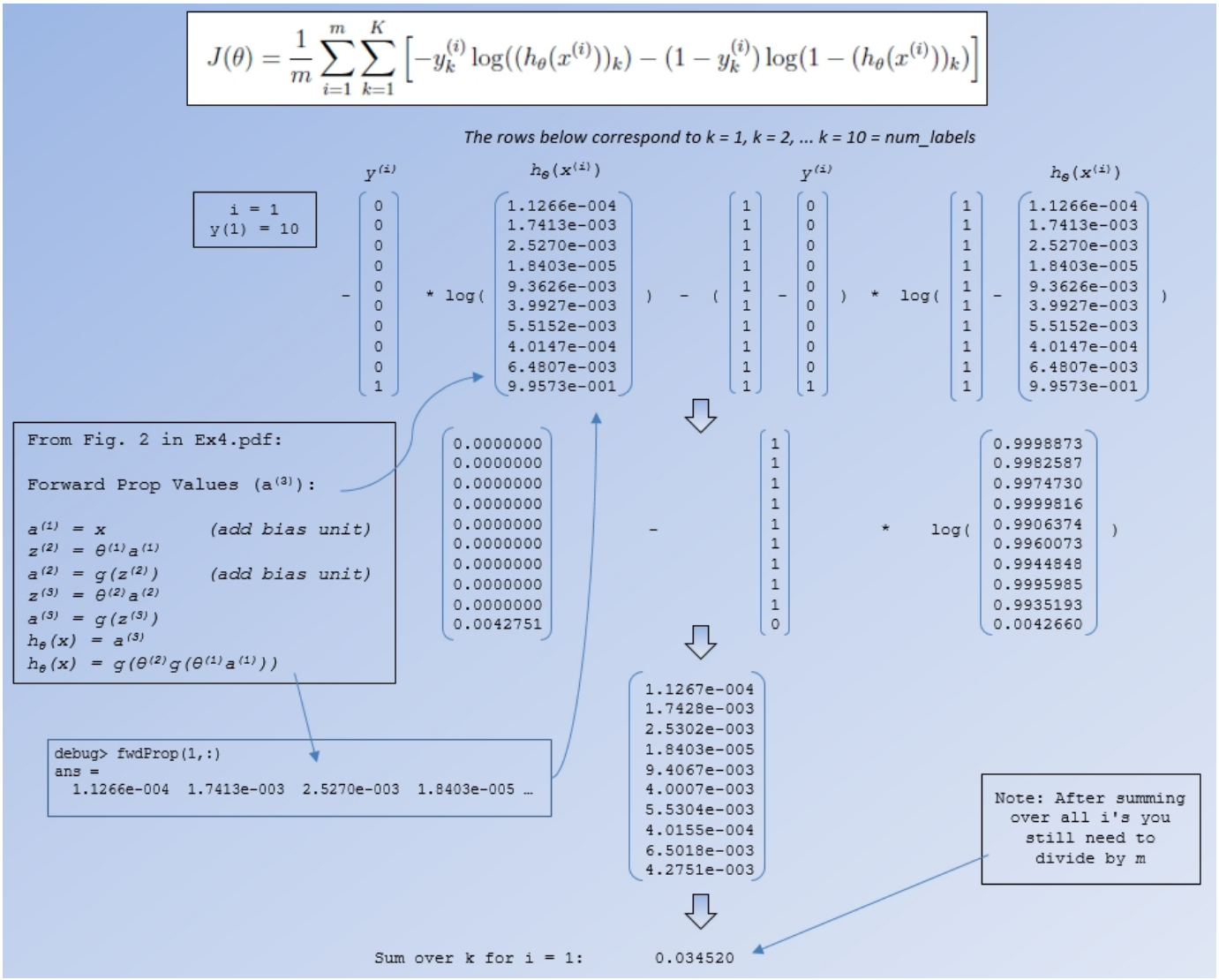
这样就可以将输出结果全部转化为binary形式（非0即1）的结果

J = (1/m) * sum(sum(-y_vec .* log(layerC) - (1 - y_vec) .* log(1 - layerC)));

根据公式，这里需要计算cost function

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

具体内涵参考下图



公式中的y即y_vec（5000*10），h即a3（5000*10），代表了输出结果的偏差值（error），这里利用y和1-y分别计算了y为0和y为1情况下的偏差值

这里用矢量的方法完全涵盖了所有数据集的情况，因此不需要再用for文的方式来对每一组数据集进行处理

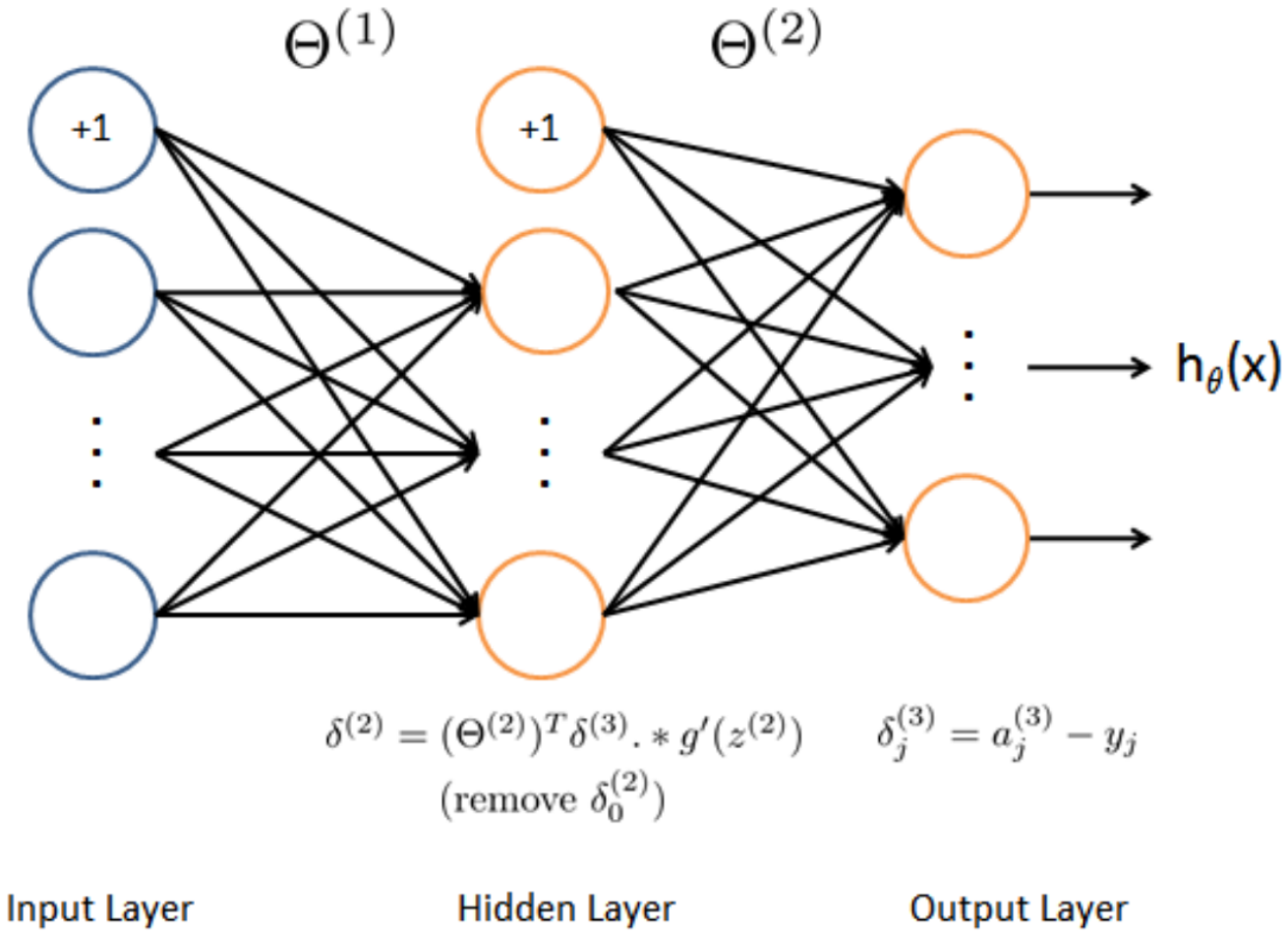
J = J + (lambda/(2*m)) * (sum(sum(Theta1(:,2:end).^2)) + sum(sum(Theta2(:,2:end).^2)));

最后对cost function进行正规化处理，这里需要排除偏差项，因此Theta1(:,2:end)即表示取第2列开始到最后为止所有的元素

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

%Part2

完成逆向传递算法



for t = 1:m,
步骤1到4在for文中进行

第一步，将X的数据代入a1并添加偏差项1，得到矢量a1（401*1），根据Theta1（25*401），Theta2（10*26），
计算出z2（25*1），a2(26*1)，z3(10*1)，a3(10*1)，其中a2也需要添加偏差项

```
a1 = [1; X(t,:)'];
z2 = Theta1 * a1;
a2 = [1;sigmoid(z2)];
z3 = Theta2 * a2;
a3 = sigmoid(z3);
```

第二步，对每个输出结果，计算偏差值delta（error），这里y_vec3 = ([1:num_labels] == y(t))';原理如下：
首先[1:num_labels]创建了一个矢量[1,2,3,4,5,6,7,8,9,10]，然后y(t)即第t组数据的输出结果，[1:num_labels] == y(t)会返回一个形式为[0,0,0,1,0,0,0,0,0,0]的矩阵，
最后将矩阵转置，得到一个10*1且仅由0或1组成的矢量
然后用a3减去y_vec3，得到输出层的误差值delta3（10*1）

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

```
y_vec3 = ([1:num_labels] == y(t))';
delta3 = a3 - y_vec3;
```

第三步，对于隐藏层（这里仅有一层即第二层）的误差值delta进行计算，公式如下：

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

Theta2的转置（26*10）和delta3（10*1）相乘，再将每一项乘以sigmoidGradient，由于sigmoidGradient本身不包含偏差项，因此需要手动加上

```
delta2 = Theta2' * delta3 .* [1; sigmoidGradient(z2)];
```

第四步，计算每一层的Δ，这里要搞清楚，δ代表当前层的所有unit个数，如果是hidden layer的话，还包括了bias unit。
Δ的最终结果是当前层的theta的偏导数，和当前层的theta维数一致，通过δ计算Δ的时候，由于后层的δ中的bias unit并不是由当前层的theta计算到的，
反向计算theta的偏导数Δ的时候则要把其中的bias unit去掉（在这里即delta2 = delta2(2:end);）

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

```
delta2 = delta2(2:end);
Theta1_grad = Theta1_grad + delta2 * a1';
Theta2_grad = Theta2_grad + delta3 * a2';
end
```

第五步，计算偏导数并进行正规化，由于偏差项不需要正规化，因此取Theta的第二列以后，并在其前面手动添加一列1

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

```
Theta1_grad = (1/m) * Theta1_grad + [zeros(size(Theta1_grad,1),1) (lambda/m) *
```

```
Theta1(:, 2:end)];
            Theta2_grad = (1/m) * Theta2_grad + [zeros(size(Theta2_grad,1),1) (lambda/m) *
Theta2(:, 2:end)];

% -------------------------------------------------------------

%
=========================================================================

% Unroll gradients
grad = [Theta1_grad(:) ; Theta2_grad(:)];

end
```

sigmoidGradient.m Compute the gradient of the sigmoid function

$$g'(z) = \frac{d}{dz}g(z) = g(z)(1 - g(z))$$

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

```
function g = sigmoidGradient(z)
%SIGMOIDGRADIENT returns the gradient of the sigmoid function
%evaluated at z
%   g = SIGMOIDGRADIENT(z) computes the gradient of the sigmoid function
%   evaluated at z. This should work regardless if z is a matrix or a
%   vector. In particular, if z is a vector or matrix, you should return
%   the gradient for each element.

g = zeros(size(z));

% ====================== YOUR CODE HERE ======================
```

```
% Instructions: Compute the gradient of the sigmoid function evaluated at
%               each value of z (z can be a matrix, vector or scalar).

g = sigmoid(z) .* (1 - sigmoid(z));
```
计算激励值的梯度

```
% =============================================================

end
```

randInitializeWeights.m Randomly initialize weights

```
function W = randInitializeWeights(L_in, L_out)
%RANDINITIALIZEWEIGHTS Randomly initialize the weights of a layer with L_in
%incoming connections and L_out outgoing connections
%   W = RANDINITIALIZEWEIGHTS(L_in, L_out) randomly initializes the weights
%   of a layer with L_in incoming connections and L_out outgoing
%   connections.
%
%   Note that W should be set to a matrix of size(L_out, 1 + L_in) as
%   the first column of W handles the "bias" terms
%

% You need to return the following variables correctly
W = zeros(L_out, 1 + L_in);

% ====================== YOUR CODE HERE ======================
% Instructions: Initialize W randomly so that we break the symmetry while
%               training the neural network.
%
% Note: The first column of W corresponds to the parameters for the bias unit
%
% Randomly initialize the weights to small values
epsilon_init = 0.12;
W = rand(L_out, 1 + L_in) * 2 * epsilon_init - epsilon_init;
```
对初始权重进行随机选择，以保证神经网络的效果

```
%
```

```
=========================================================================

end
```

checkNNGradients.m - Function to help check your gradients

```
function checkNNGradients(lambda)
%CHECKNNGRADIENTS Creates a small neural network to check the
%backpropagation gradients
%   CHECKNNGRADIENTS(lambda) Creates a small neural network to check the
%   backpropagation gradients, it will output the analytical gradients
%   produced by your backprop code and the numerical gradients (computed
%   using computeNumericalGradient). These two gradient computations should
%   result in very similar values.
%

if ~exist('lambda', 'var') || isempty(lambda)
    lambda = 0;
end

input_layer_size = 3;
hidden_layer_size = 5;
num_labels = 3;
m = 5;

% We generate some 'random' test data
Theta1 = debugInitializeWeights(hidden_layer_size, input_layer_size);
Theta2 = debugInitializeWeights(num_labels, hidden_layer_size);
% Reusing debugInitializeWeights to generate X
X  = debugInitializeWeights(m, input_layer_size - 1);
y  = 1 + mod(1:m, num_labels)';

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

% Short hand for cost function
```

```
costFunc = @(p) nnCostFunction(p, input_layer_size, hidden_layer_size, ...
                               num_labels, X, y, lambda);

[cost, grad] = costFunc(nn_params);
numgrad = computeNumericalGradient(costFunc, nn_params);

% Visually examine the two gradient computations.  The two columns
% you get should be very similar.
disp([numgrad grad]);
fprintf(['The above two columns you get should be very similar.\n' ...
         '(Left-Your Numerical Gradient, Right-Analytical Gradient)\n\n']);

% Evaluate the norm of the difference between two solutions.
% If you have a correct implementation, and assuming you used EPSILON = 0.0001
% in computeNumericalGradient.m, then diff below should be less than 1e-9
diff = norm(numgrad-grad)/norm(numgrad+grad);

fprintf(['If your backpropagation implementation is correct, then \n' ...
         'the relative difference will be small (less than 1e-9). \n' ...
         '\nRelative Difference: %g\n'], diff);

end
```