```matlab
%% Machine Learning Online Class - Exercise 4 Neural Network Learning

%  Instructions
%  ------------
%
%  This file contains code that helps you get started on the
%  linear exercise. You will need to complete the following functions
%  in this exericse:
%
%     sigmoidGradient.m
%     randInitializeWeights.m
%     nnCostFunction.m
%
%  For this exercise, you will not need to change any code in this file,
%  or any other files other than those mentioned above.
%

%% Initialization
clear ; close all; clc

%% Setup the parameters you will use for this exercise
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 label 10)

%% =========== Part 1: Loading and Visualizing Data =============
%  We start the exercise by first loading and visualizing the dataset.
%  You will be working with a dataset that contains handwritten digits.
%

% Load Training Data
fprintf('Loading and Visualizing Data ...\n')

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, :));

fprintf('Program paused. Press enter to continue.\n');
pause;


%% ================ Part 2: Loading Parameters ================
% In this part of the exercise, we load some pre-initialized
% neural network parameters.

fprintf('\nLoading Saved Neural Network Parameters ...\n')
```

```
fprintf('\nLoading Saved Neural Network Parameters ...\n')

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

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

%% ================ Part 3: Compute Cost (Feedforward) ================
%  To the neural network, you should first start by implementing the
%  feedforward part of the neural network that returns the cost only. You
%  should complete the code in nnCostFunction.m to return cost. After
%  implementing the feedforward to compute the cost, you can verify that
%  your implementation is correct by verifying that you get the same cost
%  as us for the fixed debugging parameters.
%
%  We suggest implementing the feedforward cost *without* regularization
%  first so that it will be easier for you to debug. Later, in part 4, you
%  will get to implement the regularized cost.
%
fprintf('\nFeedforward Using Neural Network ...\n')

% 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, lambda);

fprintf(['Cost at parameters (loaded from ex4weights): %f '...
         '\n(this value should be about 0.287629)\n'], J);

fprintf('\nProgram paused. Press enter to continue.\n');
pause;

%% =============== Part 4: Implement Regularization ===============
%  Once your cost function implementation is correct, you should now
%  continue to implement the regularization with the cost.
%

fprintf('\nChecking Cost Function (w/ Regularization) ... \n')

% 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, lambda);

fprintf(['Cost at parameters (loaded from ex4weights): %f '...
         '\n(this value should be about 0.383770)\n'], J);
```

```matlab

fprintf('Program paused. Press enter to continue.\n');
pause;


%% ================ Part 5: Sigmoid Gradient  ================
%  Before you start implementing the neural network, you will first
%  implement the gradient for the sigmoid function. You should complete the
%  code in the sigmoidGradient.m file.
%

fprintf('\nEvaluating sigmoid gradient...\n')

g = sigmoidGradient([-1 -0.5 0 0.5 1]);
fprintf('Sigmoid gradient evaluated at [-1 -0.5 0 0.5 1]:\n  ');
fprintf('%f ', g);
fprintf('\n\n');

fprintf('Program paused. Press enter to continue.\n');
pause;


%% ================ Part 6: Initializing Pameters ================
%  In this part of the exercise, you will be starting to implment a two
%  layer neural network that classifies digits. You will start by
%  implementing a function to initialize the weights of the neural network
%  (randInitializeWeights.m)

fprintf('\nInitializing Neural Network Parameters ...\n')

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(:)];


%% =============== Part 7: Implement Backpropagation ===============
%  Once your cost matches up with ours, you should proceed to implement the
%  backpropagation algorithm for the neural network. You should add to the
%  code you've written in nnCostFunction.m to return the partial
%  derivatives of the parameters.
%
fprintf('\nChecking Backpropagation... \n');

%  Check gradients by running checkNNGradients
checkNNGradients;
```

```
fprintf('\nProgram paused. Press enter to continue.\n');
pause;


%% =============== Part 8: Implement Regularization ===============
%  Once your backpropagation implementation is correct, you should now
%  continue to implement the regularization with the cost and gradient.
%

fprintf('\nChecking Backpropagation (w/ Regularization) ... \n')

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

% Also output the costFunction debugging values
debug_J  = nnCostFunction(nn_params, input_layer_size, ...
                          hidden_layer_size, num_labels, X, y, lambda);

fprintf(['\n\nCost at (fixed) debugging parameters (w/ lambda = %f): %f ' ...
         '\n(for lambda = 3, this value should be about 0.576051)\n\n'], lambda, debug_J);

fprintf('Program paused. Press enter to continue.\n');
pause;


%% =================== Part 8: Training NN ===================
%  You have now implemented all the code necessary to train a neural
%  network. To train your neural network, we will now use "fmincg", which
%  is a function which works similarly to "fminunc". Recall that these
%  advanced optimizers are able to train our cost functions efficiently as
%  long as we provide them with the gradient computations.
%
fprintf('\nTraining Neural Network... \n')

%  After you have completed the assignment, change the MaxIter to a larger
%  value to see how more training helps.
options = optimset('MaxIter', 500);

%  You should also try different values of lambda
lambda = 5;

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

```
                                   hidden_layer_size, ...
                                   num_labels, X, y, lambda);

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

% Obtain Theta1 and Theta2 back from nn_params
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));

fprintf('Program paused. Press enter to continue.\n');
pause;


%% ================= Part 9: Visualize Weights =================
%  You can now "visualize" what the neural network is learning by
%  displaying the hidden units to see what features they are capturing in
%  the data.

fprintf('\nVisualizing Neural Network... \n')

```

```
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
%

%% PRIMERO IMPLEMENTAMOS EL PREDICT QUE YA HICIMOS
X = [ones(m, 1) X]; % X = 5000 x 401
```

```
% a1 = 5000 x 401
a1 = X;
% a2 = 5000 x 25 --> 5000 x 26
% Theta1' = 401 x 25
a2 = sigmoid(a1 * Theta1');
a2 = [ones(m, 1) a2]; % añado col extra de 1s
% a3 = 5000 x 10 matrix
% Theta2' = 26 x 10 matrix
a3 = sigmoid(a2 * Theta2');

%% DESPUES LA COST FUNCTION SIMILAR A LA SIEMPRE PERO EN LA QUE h=a3
%% Y SE HA DE CONSIDERAR LA GENERALIZACION PROPIA DE LAS NN

% vectorizo y de manera que cada fila indica segun la posicion del 1
% cual es el dígito correcto (en lugar de llevar escrito el digito en decimal)
yVec = eye(num_labels)(y, :);
% coste unitario
cost = yVec.*log(a3)+(1-yVec).*log(1-a3);
% acumulamos los dos sumatorios (todas las clases y todos los training examples)
JsinReg = -sum(sum(cost,2))/m;
% calculamos la regularización. Asegurar dejar fuera los weights del bias
JRegTerm = sum(sum(Theta1(:,2:end).^2))+sum(sum(Theta2(:,2:end).^2));
J = JsinReg+lambda/(2*m)*JRegTerm; % Atención: es necesario sacar lambda/(2*m) aquí
% si lo dejamos en el la expresion anterior da un resultado distinto

% 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.
%

D1 = zeros(size(Theta1));
D2 = zeros(size(Theta2));

for i = 1:m, % para cada ejemplo en el training set

  % 1º FORWARD PROPAGATION
  ra1 = X(i, :)';

  rz2 = Theta1 * ra1;
  ra2 = sigmoid(rz2);
  ra2 = [1; ra2];

  rz3 = Theta2 * ra2;
  ra3 = sigmoid(rz3);

  % 2º CALCULOS ERRORES
  err3 = ra3 - yVec(i, :)'; % errores del output con los datos de training
  err2 = (Theta2' * err3)(2:end, 1) .* sigmoidGradient(rz2); % errores en la capa hidden, weighted average

  % 3º ACUMULO LA DESVIACIÓN
  D1 = D1 + err2 * ra1'; % Delta acumuladores
  D2 = D2 + err3 * ra2'; % Delta acumuladores
end

% Theta1_grad = D1 / m; % primero sin regularizar
```

```matlab
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)); % para matrices debe hacerse elemento a elemento

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

end
```

```matlab
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
%

% utilizamos valores pequeños que aseguran que los parámetros quedarán pequeños
% y el training será más eficiente

epsilon_init=0.12;
W = rand(L_out, 1+ L_in)* 2*epsilon_init - epsilon_init;

%
```

```matlab
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);
```

```matlab
function numgrad = computeNumericalGradient(J, theta)
%COMPUTENUMERICALGRADIENT Computes the gradient using "finite differences"
%and gives us a numerical estimate of the gradient.
%   numgrad = COMPUTENUMERICALGRADIENT(J, theta) computes the numerical
%   gradient of the function J around theta. Calling y = J(theta) should
%   return the function value at theta.

% Notes: The following code implements numerical gradient checking, and
%        returns the numerical gradient.It sets numgrad(i) to (a numerical
%        approximation of) the partial derivative of J with respect to the
%        i-th input argument, evaluated at theta. (i.e., numgrad(i) should
%        be the (approximately) the partial derivative of J with respect
%        to theta(i).)
%

numgrad = zeros(size(theta));
perturb = zeros(size(theta));
e = 1e-4;
for p = 1:numel(theta)
    % Set perturbation vector
    perturb(p) = e;
    loss1 = J(theta - perturb);
    loss2 = J(theta + perturb);
    % Compute Numerical Gradient
    numgrad(p) = (loss2 - loss1) / (2*e);
    perturb(p) = 0;
end

end
```

```
function [h, display_array] = displayData(X, example_width)
%DISPLAYDATA Display 2D data in a nice grid
%   [h, display_array] = DISPLAYDATA(X, example_width) displays 2D data
%   stored in X in a nice grid. It returns the figure handle h and the
%   displayed array if requested.

% Set example_width automatically if not passed in
if ~exist('example_width', 'var') || isempty(example_width)
  example_width = round(sqrt(size(X, 2)));
end

% Gray Image
colormap(gray);

% Compute rows, cols
[m n] = size(X);
example_height = (n / example_width);

% Compute number of items to display
display_rows = floor(sqrt(m));
display_cols = ceil(m / display_rows);

% Between images padding
pad = 1;

% Setup blank display
display_array = - ones(pad + display_rows * (example_height + pad), ...
                       pad + display_cols * (example_width + pad));

% Copy each example into a patch on the display array
curr_ex = 1;
for j = 1:display_rows
  for i = 1:display_cols
    if curr_ex > m,
      break;
    end
    % Copy the patch

    % Get the max value of the patch
    max_val = max(abs(X(curr_ex, :)));
    display_array(pad + (j - 1) * (example_height + pad) + (1:example_height), ...
              pad + (i - 1) * (example_width + pad) + (1:example_width)) = ...
          reshape(X(curr_ex, :), example_height, example_width) / max_val;
    curr_ex = curr_ex + 1;
  end
  if curr_ex > m,
    break;
  end
end
```

```
end

% Display Image
h = imagesc(display_array, [-1 1]);

% Do not show axis
axis image off

drawnow;

```

```
function p = predict(Theta1, Theta2, X)
%PREDICT Predict the label of an input given a trained neural network
%   p = PREDICT(Theta1, Theta2, X) outputs the predicted label of X given the
%   trained weights of a neural network (Theta1, Theta2)

% Useful values
m = size(X, 1);
num_labels = size(Theta2, 1);

% You need to return the following variables correctly
p = zeros(size(X, 1), 1);

h1 = sigmoid([ones(m, 1) X] * Theta1');
h2 = sigmoid([ones(m, 1) h1] * Theta2');
[dummy, p] = max(h2, [], 2);

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


end
```