Programming Exercise 4
Neural Networks Learning

Machine Learning (Stanford University)

September 2020

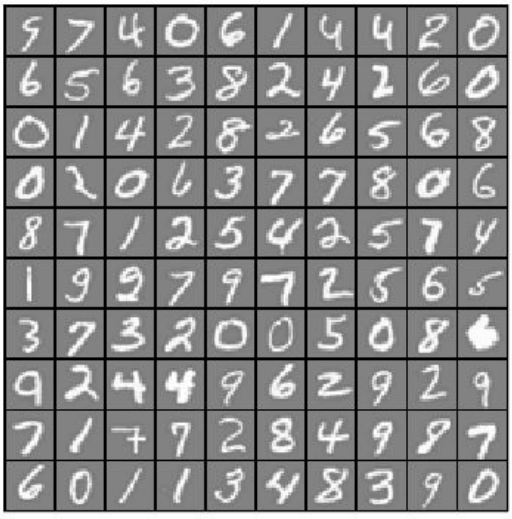
This exercise shows the implementation of the backpropagation algorithm for neural networks applied to the task of handwritten digit recognition.

Specifically,

- **sigmoidGradient.m**
  - Compute the gradient of the sigmoid function
- **randInitializeWeights.m**
  - Randomly initialize weights
- **nnCostFunction.m (Neural Network cost function)**
  - Feedforward and Cost Function
  - Regularized Cost Function
  - Neural Net Gradient Function (Backpropagation)
  - Regularized Gradient

1 Neural Networks

1.1 Visualizing the data



*This is the same dataset in the previous exercise.

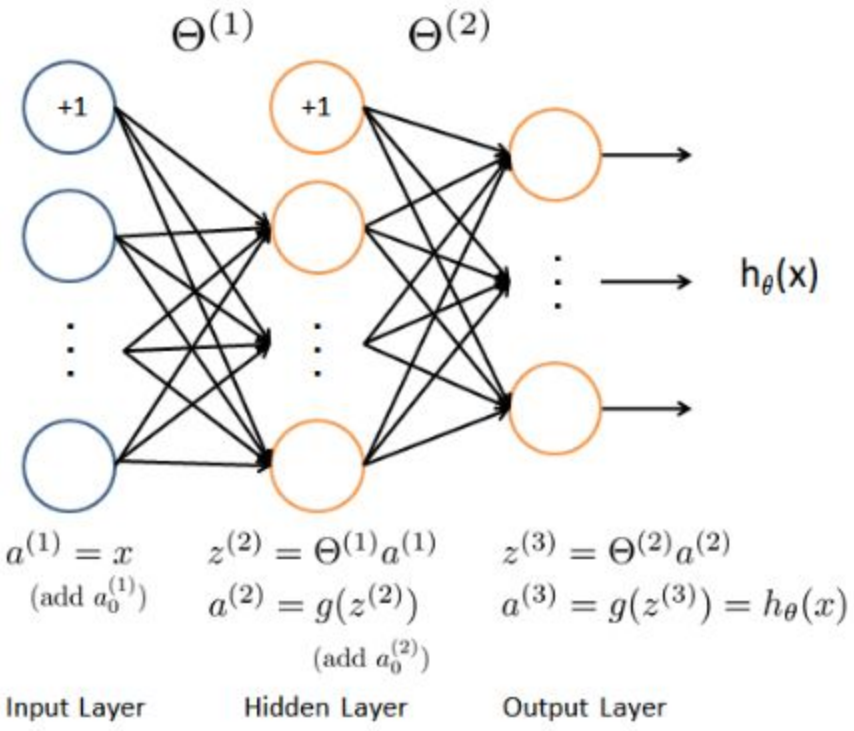
Training Set (X)

- Originally
  - 5000 training examples
  - Each training example is 20 x 20 pixels (grayscale)
  - Each pixel is represented by a floating point number indicating the grayscale intensity at that location
- Flattening
  - 20 x 20 pixels is “unrolled”/”flattened” into a 400 x 1 dimensional vector.
  - Each of these training examples becomes a single row in our data matrix X
- Resulting data
  - Therefore, this gives us a 5000 by 400 matrix X where every row is a training example of a handwritten digit image

Ground Truth Labels

- 5000 x 1 dimensional vector y

1.2 Model Presentation



Neural Network Model

*This was given.

```
% 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
```

1.3 Feedforward and cost function

Instruction:

You will implement the cost function and gradient for the neural network. First, complete the code in nnCostFunction.m to return the cost [without regularization].

Recall:

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

Implementation notes:

- Note that $h_\theta(x^{(i)})_k = a_k^{(3)}$ is the activation (output value) of the k-th output unit.
- *K* is the number of classes
- 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

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

- 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).
- The matrix X contains the examples in rows (i.e., X(i,:)' is the i-th training example $x^{(i)}$, expressed as a $n \times 1$ 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 are represented in Theta1 and Theta2 as one row. Specifically, the first row of Theta1 corresponds to the first hidden unit in the second layer.

  Cost should be about 0.287629.

Implementation:

```
function [J grad] = nnCostFunction(nn_params, ...
                                   input_layer_size, ...
                                   hidden_layer_size, ...
                                   num_labels, ...
                                   X, y, lambda)

% 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));

% ====================== Part 1 ======================

%Feedforward
a_1 = [ones(m,1) X ]

z_2 = a_1*Theta1'; % result is 16x4
a_2 = sigmoid(z_2);

a_2 = [ones(size(a_2,1),1) a_2]

z_3 = a_2*Theta2';
a_3 = sigmoid(z_3);

h = a_3


% cost J
y_proc = (1:num_labels)==y
J = (1/m) * sum(sum((-y_proc.*log(h))-((1-y_proc).*log(1-h))));
```

This is like the same as before except that the cost function now is different.

1.4 Regularized cost function

Instruction:

Implement the cost function with regularization as given:

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

Implementation Notes:

- 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.**

  The cost should be about 0.383770.

Implementation:

Using the J before:

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

2 Backpropagation

Instruction:

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(Θ) 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

Instruction:
To help you get started with this part of the exercise, you will first implement the sigmoid gradient function.

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

where

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

Note that this is used in computing the errors.

Implementation Notes:
- Values of 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.

Implementation:

```
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));
```

2.2 Random initialization

Advice:

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

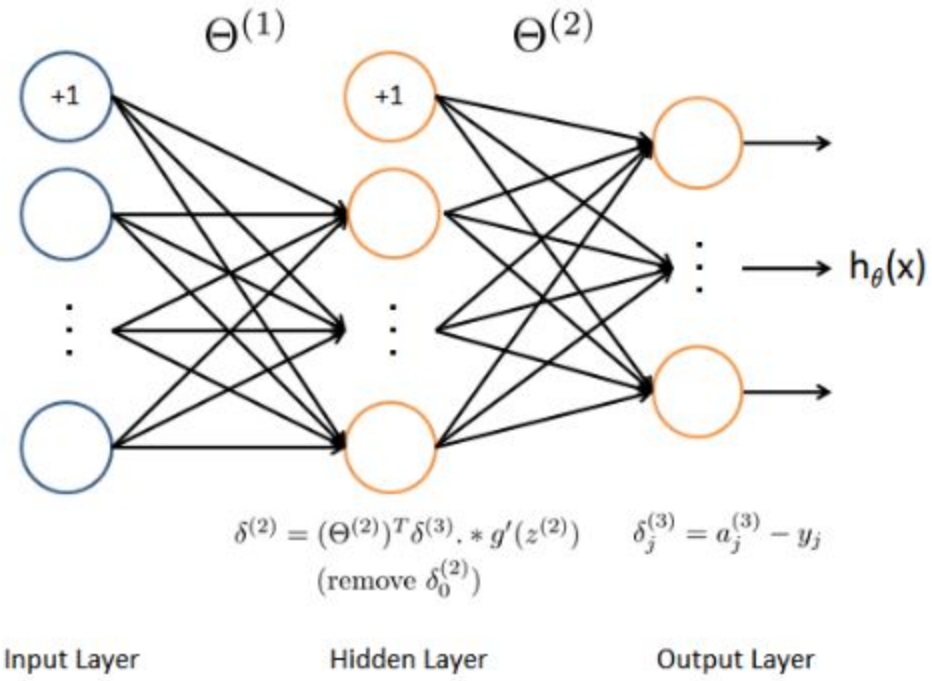
[2]One effective strategy for choosing ϵ_{init} is to base it on the number of units in the network. A good choice of ϵ_{init} is $\epsilon_{init} = \frac{\sqrt{6}}{\sqrt{L_{in}+L_{out}}}$, where $L_{in} = s_l$ and $L_{out} = s_{l+1}$ are the number of units in the layers adjacent to $\Theta^{(l)}$.

Implementation: (This was actually given)

```
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);

% 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;

end
```

2.3 Backpropagation

Instruction:
Implement the backpropagation algorithm. A guide is given below,



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 Octave/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

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

4. Accumulate the gradient from this example using the following formula. Note that you should skip or remove $\delta_0^{(2)}$. In Octave/MATLAB, removing $\delta_0^{(2)}$ corresponds to `delta_2 = delta_2(2:end)`.

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

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

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

Implementation:
The implementation is straightforward.

```
% input
a_1 = [ones(m,1) X ]

% forward prop
z_2 = a_1*Theta1';
a_2 = sigmoid(z_2);

a_2 = [ones(size(a_2,1),1) a_2]

z_3 = a_2*Theta2';
a_3 = sigmoid(z_3);

h = a_3; y_proc = (1:num_labels)==y;

delta_3 = h - y_proc;
delta_2 = (delta_3*Theta2).*[ones(size(z_2,1),1) sigmoidGradient(z_2)];
delta_2 = delta_2(:,2:end);

big_delta_2 = delta_3'*a_2;
big_delta_1 = delta_2'*a_1;

% get D
Theta2_grad = (1/m)*(big_delta_2);
Theta1_grad = (1/m)*(big_delta_1);
```

Then gradient checking was done after implementing the unregularized gradient calculations. After this, gradient checking should be turned off since it will cause the program to run slow.

2.4 Regularized Neural Networks

Instruction:
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.**

Recall:

This is done by:

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

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

Caution:
Note that you should not be regularizing the first column of $\Theta^{(l)}$ which is used for the bias term.

Implementation:

```
113  Theta2_grad = Theta2_grad  + (lambda/m)*([zeros(size(Theta2,1),1) Theta2(:,2:end)]);
114  Theta1_grad = Theta1_grad + (lambda/m)*([zeros(size(Theta1,1),1) Theta1(:,2:end)]);
```

This seems to just be the second case but we also already take into account the first case since we only selected second columns and above of Theta1 and Theta2.

3 Scores

```
==
==                                  Part Name |     Score | Feedback
==                                  --------- |     ----- | --------
==                Feedforward and Cost Function |  30 /  30 | Nice work!
==                    Regularized Cost Function |  15 /  15 | Nice work!
==                              Sigmoid Gradient |   5 /   5 | Nice work!
==   Neural Network Gradient (Backpropagation) |  40 /  40 | Nice work!
==                           Regularized Gradient |  10 /  10 | Nice work!
==                                  ---------------------------------
==                                            | 100 / 100 |
```