INF 5860 Exercises for week 7

Exercise 1 : Set up a computational graph and do backpropagation by hand

Consider a RELU activation node that takes as input $f=w_0x_0+w_1x_1$. The output of the RELU operation is: RELU=max(0,f)

a) Draw the computational graph
b) Do forward propagation for a sample (x_0=1, w_0=0.5, x_1=2, w_1=1,)
c) Do backward propagation to get the partial derivatives with respect to x_0, w_0, x_1, w_1)

Exercise 2: Sketch a simple net

A net is used to classify images of size 3x3 into a set of 6 different classes. The network has two hidden layers, with 3 nodes in hidden layer 1, and 2 nodes in hidden layer 2.

a) Draw the network with all connections and bias nodes.
b) How many Θ–matrices does this net require?
c) What is the dimension of each Θ–matrix?

Programming exercise: implement backpropagation

In this exercise, you will implement the backpropagation algorithm for a two-layer net. The focus is on implementing the backpropagation using a simple small dataset. Once your code works, you can proceed to use it to classify the MNIST-images.

This text will guide you through the implementation. In this exercise, we develop a straightforward implementation of a 2-layer net. Later, we will use library frameworks where it is easy to change the architecture, optimization, and all network parameters.

Please note that no solution code will be available. You will be required to implement backpropagation in mandatory exercise 1. At that point, you get a partial framework for also optimizing all network parameters so you can easily change the architecture to achieve as high accuracy as possible.

The following files are included in the exercise :

week7exercise.py
week7data1.mat
week7weights.mat
checkgrad.py
nncost.py

Part 1: Implement and test the sigmoid gradient

The gradient of the sigmoid function is given as $g'(z) = g(z)(1 - g(z))$. Implement the function sigmoidGradient(z). Check the output, for large positive or negative values, the gradient should be close to 0. When z=0 the gradient should be exactly 0.25. Your code should work for vectors as input, and return the gradient for everly element.

Part 2: Implement random initialization

To get good performance, it is important that a neural net is initialized with small different random numbers. Let us initialize weights by selecting random values in the range [-ei,ei]. Use ei=0.12. Implement the function randInitializeWeights in week7exercise.py by setting each weight

```
r *2 *ei - ei
```

where r is a different random number for each weight.

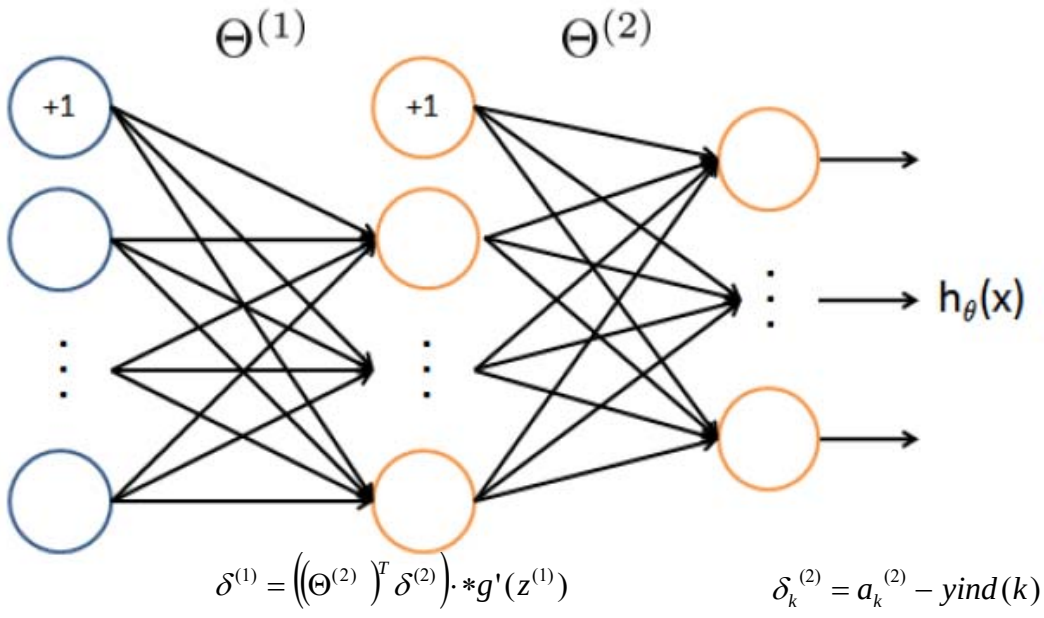
week7exercise.py use the mean and standard deviation is computed to check that the weights are small and different.

Part 3: Implement backpropagation

This exercise extends the function nnCostFunction that you implemented last week (if not, start with this and check last week's exercise that you get correct cost value with and without regularization on the pre-loaded weights).

Before starting on this, verify that the cost on the preloaded weights with data week7exercise1.dat is J=0.2876 without regularization (lambda=0) , and J=0.3877 with lambda=1.

Once it returns the correct cost, modify it by adding code compute the derivative of J with respect to all parameters in $\Theta^{(1)}$ and $\Theta^{(2)}$.



Recall that to do backpropagation, you should start with a forward pass of training sample (xi,yi) to get the outputs $a_k{}^{(2)}$

In the function checkNNgradient, a small network with 5 samples is set up to help your implementation. It might be useful to sketch the net with connections and bias nodes to assist your debugging and verify dimensions.

Implement step 1-4 in a for loop over the training samples:

1. Do a forward pass of sample x_i $a^{(0)}$ to get $z^{(1)}$, $a^{(1)}$, $z^{(2)}$ and $a^{(2)}$. Remember to add bias nodes to the correct $a^{(l)}$ vectors.
2. For each output node in the output layer set $\delta_k{}^{(2)} = a_k{}^{(2)} - yind(k)$
   This should be a vector of length num_labels.
3. For hidden layer l=1, set

$$\delta^{(1)} = \left(\left(\Theta^{(2)}\right)^T \delta^{(2)}\right) .* g'(z^{(1)})$$

Hint : $\left(\left(\Theta^{(2)}\right)^T \delta^{(2)}\right)$ must have the same dimension as $g'(z^{(1)})$, which does not have any bias node, so discard the bias nodes when computing $\left(\left(\Theta^{(2)}\right)^T \delta^{(2)}\right)$
And note $\cdot *$, elementwise product.

4. Accumulate the gradients in:

$$\text{Set } \Delta_{ij}^{(l)} = \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$$

Hints:

Note that $\Delta_{ij}^{(l)}$ is a matrix with the same size as $\Theta^{(l)}$
So $a_j^{(l)} \delta_i^{(l+1)}$ is also a matrix of the same size

5. Obtain the unregularized gradients outside the for-loop by normalizing:

$$D_{ij}^{(l)} = \frac{1}{m} \Delta_{ij}^{(l)}$$

After you have implemented the backpropagation, we will run gradient checking on it.

Part 4: Gradient checking

In the function checkNNGradients, estimate numerically each gradient in Theta1 and Theta2 as:

.....

$$\frac{\partial J}{\partial \theta_1} = \frac{J(\theta_1 + \varepsilon, \theta_2, ... \theta_n) - J(\theta_1 - \varepsilon, \theta_2, ... \theta_n)}{2\varepsilon}$$

$$\frac{\partial J}{\partial \theta_2} = \frac{J(\theta_1, \theta_2 + \varepsilon, ... \theta_n) - J(\theta_1, \theta_2 - \varepsilon, ... \theta_n)}{2\varepsilon}$$

$$\vdots$$

$$\frac{\partial J}{\partial \theta_n} = \frac{J(\theta_1, \theta_2, ... \theta_n + \varepsilon) - J(\theta_1, \theta_2, ... \theta_n - \varepsilon)}{2\varepsilon}$$

Compare the numerical gradient to the corresponding backprop-gradient element by element, and verify that the difference is small (1e-5 or smaller). Let ε=1e-4

Do this on the small example, not on MNIST during ordinary training.

Part 5: include the regularization term in the gradient computation

After your backprop-algorithm pass the gradient check, include the regularization term:

$$D_{ij}^{(l)} = \frac{1}{m}\Delta_{ij}^{(l)} + \lambda\Theta_{ij}^{(l)}, \text{if } j \neq 0$$

$$D_{ij}^{(l)} = \frac{1}{m}\Delta_{ij}^{(l)}, \qquad \text{if } j = 0$$

Avoid regularizing the first column of Θ(j) which is used for the bias term.

Step 6: implement gradient descent, train the network on MNIST data

When your code works, you can add code in week7exercise.py to do gradient descent on Theta1 and Theta2. It will take a while to train, so try a higher learning rate to get some results in a limited time. See how the convergence changes if you use a learning rate of 0.01, 0.1, 1 or 3. Monitor changes in the cost function. Can you beet 95% accuracy on the training data? You might also try different regularization parameters.