Assignment Two Analysis (Neural Network Gradient Descent)

by Haoyu Wang (haoyuw2)

Analysis:

By implementing the One hidden layer Neural Network method, I reduce the error of ojective function and produce a 97.25% accuracy model when testing the model by y_train. To implement the model, I follow the steps proposed in the course note. First, I randomly generate 4 matrix, corresponding to the W, b1, C and b2. Then, I define a variable called *ite* to follow the iterations in the training process and a while loop for *ite_max* iterations to train the parameters. During the training process, I create a random integer number every iteration and derive a randomly selected *x_rand* and *y_rand* data from set *x_train* and set *y_train*. Then I forward the x_rand to the model (ReLU, softMax) to get the forward_x. By utilizing the backpropagation method described in the class note, I derive the derivate of objective function with respect to each parameter. Subtracting the multiplication of learning rate and the deepest gradient descent from *model[W], model[b1], model[C], model[b2],* one iteration is finished. Learning rate is defined as 0.005 and *ite_max* is defined as 600000 by trial and error.

Code:

Attached in the next page.

Assignment2_DeepLearning

February 6, 2019

In [9]:
```
import numpy as np
import h5py
import time
import copy
from random import randint
import random
#load MNIST data
MNIST_data = h5py.File('data.hdf5', 'r')
x_train = np.float32(MNIST_data['x_train'][:] )
y_train = np.int32(np.array(MNIST_data['y_train'][:,0]))
x_test = np.float32( MNIST_data['x_test'][:] )
y_test = np.int32( np.array( MNIST_data['y_test'][:,0] ) )
MNIST_data.close()
####################################################################################

#define softmax for f(x;theta)
def softMax(x):
    out_put = np.exp(x)/np.sum(np.exp(x),axis =0)
    return out_put

#define ReLU fucnction that act as hidden layer in the training process.
def ReLU(x_input):
    for i in range(len(x_input)):
        if (x_input[i] < 0):
            x_input[i] = 0
    return x_input




#Implementation of stochastic gradient descent algorithm
#number of inputs
num_inputs = 28*28
#number of outputs
num_outputs = 10
#number of hidden units
num_hidden_unit = 150
```

```
model = {}
#model 'W' is weight first layer, dimension is R(D)->R(dH) (input->output)
model['W'] = np.random.randn(num_hidden_unit,num_inputs) / np.sqrt(num_inputs)
#model 'b1' is the first layer bias , dimension is R(dH)
model['b1'] = np.random.randn(num_hidden_unit)
#model 'C' is weight of next layer, dimension is R(dH)->R(K)
model['C'] = np.random.randn(num_outputs,num_hidden_unit)
#model 'b2' is the second layer bias, dimension is R(K)
model['b2'] = np.random.randn(num_outputs)

##############################start the Neural Network gradient descent process#########
#define learning_rate to be 0.0085
learning_rate = 0.005
ite_max = 600000
for ite in range(ite_max):
    #pick a random point
    rand_num = random.randint(0,len(x_train)-1)
    x_rand = x_train[rand_num]
    y_rand = y_train[rand_num]

    #Acquire the forward x by forward function
    Z = model['W']@x_rand + model['b1']
    H1 = ReLU(Z)
    U =  model['C']@H1 + model['b2']
    forward_x = softMax(U)

    rho_wrt_U = np.zeros(num_outputs)
    for i in range(num_outputs):
        if i==y_rand:
            rho_wrt_U[i] =  -(1-forward_x[i])
        else:
            rho_wrt_U[i] =  -(-forward_x[i])

    #No sure we still need to tranpose over here
    delta = model['C'].T@rho_wrt_U
    der_Z = np.copy(H1)
    #Find the derivative
    der_Z[der_Z>0] = 1


    rho_wrt_b1 = np.multiply(delta,der_Z)
    rho_wrt_W = rho_wrt_b1.reshape(num_hidden_unit,1)@x_rand.reshape(1,784)

    model['C'] = model['C'] - learning_rate*rho_wrt_U.reshape(num_outputs,1)@H1.T.reshap
    model['b2'] = model['b2'] - learning_rate*rho_wrt_U
    model['b1'] = model['b1'] - learning_rate*rho_wrt_b1
    model['W'] = model['W'] - learning_rate*rho_wrt_W
```

```
        print("FINISH")
```

```
FINISH
```

```
In [10]: #forward to get the f(x;theta)
         def forward(x_input,model):
             Z = model['W']@x_input + model['b1']
             H1 = ReLU(Z)
             U =  model['C']@H1 + model['b2']
             return softMax(U)


         ######################################################
         #test data
         total_correct = 0
         for n in range(len(x_test)):
             y = y_test[n]
             x = x_test[n][:]
             p = forward(x, model)
             prediction = np.argmax(p)
             if (prediction == y):
                 total_correct += 1
         print(total_correct/np.float(len(x_test) ) )
```

```
0.9725
```