Assignment Two Analysis (Neural Network Gradient Descent)

by Haoyu Wang (haoyuw2)

Analysis:

By implementing the multiple channel convolution nerual network , I produce a 94.83% accuracy model when testing the model by training data. To implement the model, I follow the steps proposed in the course note. First, I randomly generate 3 matrics, corresponding to the b, K, and W representing the bias, convolution matrix and weighted matrix for fully connect network. Then, I define a variable called *ite* to follow the iterations in the training process in 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 (Convolution, ReLU, softMax) to get the forward_x. By utilizing the backpropagation method described in the class note, I derive the derivative of objective function with respect to each parameter - b, K and W. Subtracting the multiplication of learning rate and the deepest gradient descent from *model['b'], model['K'], model['W'],* one iteration is finished. Learning rate is defined as 0.05 and *ite_max* is defined as 55000 by trial and error. One thing that is important to notice is that the running time of convolution is relativly slow compared with SGD and Neural network, which usually takes 10 seconds per thought iteration. I therefore vectorize matrix to reduce the running time of whole model.

Code:

Attached in the next page.

Assignment_3_Convolution_Network

February 14, 2019

In [ ]:
```
import numpy as np
import h5py
import time
import copy
from random import randint
import random
np.set_printoptions(threshold=np.nan)
#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):
    return x_input*(x_input>0)

#Implementation of stochastic gradient descent algorithm
#number of inputs
num_inputs = 28*28
#number of outputs
num_outputs = 10
#number of filter width
width_filter = 10
#channel of the filter
channel = 3
#R(Width*d) = D-width+1
inter_width = 28-width_filter+1
#Initialize model dictionary
```

```
model = {}

#model 'K' is convolution filter, dimension is R(width_filter*width_filter*width_filter)
model['K'] = np.random.randn(width_filter, width_filter, channel) / np.sqrt(num_inputs)
#model 'W' is the weight matrix of the single layer of fully con , dimension is R(Width*
model['W'] = np.random.randn(num_outputs, inter_width, inter_width, channel) / np.sqrt(n
#model 'b' is the layer bias, dimension is R(K)
model['b'] = np.random.randn(num_outputs)

##############################Start the Convolution Network process###################
#define learning_rate to be 0.0085 600000 0.005
learning_rate = 0.05
ite_max = 55000
for ite in range(ite_max):
    #pick a random point
    if(ite%1000==0):
        print(ite)
    rand_num = random.randint(0,len(x_train)-1)
    x_rand = x_train[rand_num].reshape(28,28)
    y_rand = y_train[rand_num]

    #Acquire the forward x by forward function
    Z = np.zeros(inter_width*inter_width*channel).reshape(inter_width,inter_width,channe
    for z_dim in range(channel):
        for y_dim in range(inter_width):
            for x_dim in range(inter_width):
                x_area = x_rand[y_dim:y_dim+width_filter,x_dim:x_dim+width_filter]
                Z[y_dim][x_dim][z_dim] = np.sum(np.multiply(x_area, model['K'][:,:,z_dim

    #Take the ReLU function
    H = ReLU(Z)

    #Get the U by a fully connected network
    U = np.zeros(num_outputs)
    for x_dim in range(num_outputs):
        U[x_dim] = np.sum(np.multiply(model['W'][x_dim,:,:,:],H))+model['b'][x_dim]

    forward_x = softMax(U)

    #Calculate the rho with respect to 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])

    #Calculate the delta
```

```
        delta = np.zeros(inter_width*inter_width*channel).reshape(inter_width,inter_width,ch
        for z_dim in range(channel):
            for y_dim in range(inter_width):
                for x_dim in range(inter_width):
                    delta[y_dim][x_dim][z_dim] = rho_wrt_U@model['W'][:,y_dim,x_dim,z_dim]

        #Calculate rho with respect to W
        rho_wrt_W = np.zeros(num_outputs*inter_width*inter_width*channel).reshape(num_output
        for x_dim in range(num_outputs):
            rho_wrt_W[x_dim,:,:,:] = rho_wrt_U[x_dim] * H

        #Rho with respect to b
        rho_wrt_b = rho_wrt_U

        #Calculate the derivative of Z
        der_Z = 1*(Z>1)

        #Calculate the rho with respect to K
        dot_product = np.multiply(der_Z,delta)
        rho_wrt_K = np.zeros(width_filter*width_filter*channel).reshape(width_filter,width_f
        for z_dim in range(channel):
            for y_dim in range(width_filter):
                for x_dim in range(width_filter):
                    x_area = x_rand[y_dim:y_dim+inter_width,x_dim:x_dim+inter_width]
                    rho_wrt_K[y_dim,x_dim,z_dim]= np.sum(np.multiply(x_area, dot_product[:,:

        model['b'] = model['b'] - learning_rate*rho_wrt_b
        model['K'] = model['K'] - learning_rate*rho_wrt_K
        model['W'] = model['W'] - learning_rate*rho_wrt_W



    print("FINISH")
```

In [118]:
```
#forward to get the f(x;theta)
def forward(x_input,model):
    #Acquire the forward x by forward function
    x_input = x_input.reshape(28,28)
    Z = np.zeros(inter_width*inter_width*channel).reshape(inter_width,inter_width,chan
    for z_dim in range(channel):
        for y_dim in range(inter_width):
            for x_dim in range(inter_width):
                x_area = x_input[x_dim:x_dim+width_filter,y_dim:y_dim+width_filter]
                Z[x_dim][y_dim][z_dim] = np.sum(np.multiply(x_area, model['K'][:,:,z_d
```

```
    #Take the ReLU function
    H = ReLU(Z)

    #Get the U by a fully connected network
    U = np.zeros(num_outputs)
    for x_dim in range(num_outputs):
        U[x_dim] = np.sum(np.multiply(model['W'][x_dim,:,:,:],H))+model['b'][x_dim]

    forward_x = softMax(U)
    return forward_x


######################################################
#test data
total_correct = 0
for n in range(len(x_test)):
    if(n%1000==0):
        print(n)
    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) ) )
print("FINISH")
```

```
0.9483
FINISH
```