CS 547 / IE 534 Deep Learning, Fall 2019
Homework 1

Author: Vardhan Dongre (vdongre2)

Implemented and trained a neural network from scratch in Python for the MNIST dataset. The neural network was trained on the Training Set using stochastic gradient descent.

This implementation achieved an accuracy of **97.63%** on the Test Set.

Implementation:

The code can be understood in the following subheads:

1. **Data:** MNIST database is a database of handwritten digits which contains 60,000 training images and 10,000 testing images. The images are pre-processed to fit in 28X28 pixel box. In this code the training images are stored in x_train and x_test and the corresponding labels are in y_train and y_test respectively.

2. **Model:** The model/Learning Algorithm adopted is a Neural Network with single hidden layer. The number of input to the model are 784 and the models learns a mapping to an output size of 10 [$f(x;\theta): R^d \rightarrow R^k$] Where d = 784 and k =10. The network consists a single hidden layer. The number of units in the hidden layer is taken as 100. The weights and biases for each layer are randomly initialized, normalized and stored in a dictionary {W, b1, C, b2} where,
   $W \in R^{100X784}$
   $b1 \in R^{100X1}$
   $C \in R^{10X100}$
   $b2 \in R^{10X1}$
   ReLU function is used which adds a layer of non-linearity. Forward propagation is performed on randomly sampled data points in training set. Backward propagation is implemented to calculate the gradients of parameters which are required for computing the step that is used to optimize the model. Stochastic Gradient Descent has been implemented for optimization where we are using single data samples for computing the update direction

3. **Training:** This model is trained for 20 epochs using a Learning rate schedule with 0.01 as the beginning learning rate. The Learning rate is defined using a piecewise learning rate schedule which decreases from 0.01 to 0.00001 by an order of 10^{-1} with every 5 epochs. Predictions of the model ($F_{softmax}(U)$) are compared with the actual labels for the training data and the accuracy of the model is calculated. It was observed that with each iteration the accuracy of the prediction increased, and model achieved an accuracy of 98% on the training set after 7 epochs.

4. **Testing:** The model is tested on the Test set and an accuracy of 97.63% is obtained. (Objective was to achieve an accuracy of 97 – 98 % on the Test set)

In [23]:
```
#-------------------- Neural Network for Classification ------------------#
#                                                                          #
# Implementation of Single hidden layer Neural Network for classifying    #
# MNIST dataset containing hand-written digits (0-9) using Stochastic     #
# Gradient Descent. Target accuracy on Test Set was 97% - 98%, This       #
# implementation achieved 97.63% accuracy with the follwing hyper-        #
# parameters:                                                             #
# Units in hidden layer =100, activation function = RelU                  #
#                                                                          #
# Created by: Vardhan Dongre                                              #
# [ Based on code provided for Logisitic Regression in CS 547 (Fall 19) ] #
#-------------------------------------------------------------------------#

import numpy as np
import h5py
import time
import copy
from random import randint

#load MNIST data
MNIST_data = h5py.File('MNISTdata.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()

###########################################################################
#Implementation of stochastic gradient descent algorithm
#number of inputs
num_inputs = 28*28
#number of outputs
num_outputs = 10
# number of hidden units
hidden = 100

model = {}
model['W'] = np.random.randn(hidden,num_inputs) / np.sqrt(num_inputs)
model['b1'] = np.random.randn(hidden,1) / np.sqrt(hidden)
model['C'] = np.random.randn(num_outputs,hidden) / np.sqrt(hidden)
model['b2'] = np.random.randn(num_outputs,1) / np.sqrt(hidden)
model_grads = copy.deepcopy(model)

def activation(Z,type = 'ReLU',deri = False):
        # implement the activation function
        if type == 'ReLU':
            if deri == True:
                return np.array([1 if i>0 else 0 for i in np.squeeze(Z)])
            else:
                return np.array([i if i>0 else 0 for i in np.squeeze(Z)])
        elif type == 'Sigmoid':
            if deri == True:
                return 1/(1+np.exp(-Z))*(1-1/(1+np.exp(-Z)))
            else:
                return 1/(1+np.exp(-Z))
        elif type == 'tanh':
```

```
            if deri == True:
                return
            else:
                return 1-(np.tanh(Z))**2
        else:
            raise TypeError('Invalid type!')

def softmax_function(z):
    ZZ = np.exp(z)/np.sum(np.exp(z))
    return ZZ

def cross_entropy_error(v,y):
    return -np.log(v[y])


def forward(x,y, model):
    Z = np.matmul(model['W'],x).reshape((hidden,1)) + model['b1']
    H = np.array(activation(Z, deri = False)).reshape((hidden,1))
    U = np.matmul(model['C'],H).reshape((num_outputs,1)) + model['b2']
    predicted = np.squeeze(softmax_function(U))
    p = predicted.reshape((1,num_outputs))
    error = cross_entropy_error(predicted,y)
    results = {
        'Z': Z,
        'H': H,
        'U': U,
        'p':p,
        'error': error
    }
    return results

def backward(x,y,forward_results, model, model_grads):
    E = np.array([0]*num_outputs).reshape((1,num_outputs))
    E[0][y] = 1
    dU = (-(E - forward_result['p'])).reshape((num_outputs,1))
    model_grads['b2'] = copy.copy(dU)
    model_grads['C'] = np.matmul(dU, forward_results['H'].transpose())
    delta = np.matmul(second_layer['C'].transpose(),dU)
    model_grads['b1'] = delta.reshape(hidden,1)*activation(forward_results[
    model_grads['W'] = np.matmul(model_grads['b1'].reshape((hidden,1)),x.re
    return model_grads

import time
time1 = time.time()
LR = .01
num_epochs = 20
for epochs in range(num_epochs):
    if (epochs > 5):
        LR = 0.001
    if (epochs > 10):
        LR = 0.0001
    if (epochs > 15):
        LR = 0.00001

    total_correct_train = 0
    for n in range( len(x_train)):
        n_random = randint(0,len(x_train)-1 )
```

```
            y = y_train[n_random]
            x = x_train[n_random][:]
            forward_result = {}
            forward_result = forward(x, y, model)
            p_values = forward_result['p']
            prediction = np.argmax(p_values)
            if (prediction == y):
                total_correct_train += 1
            model_grads = backward(x,y,forward_result, model, model_grads)
            model['C'] -= LR*model_grads['C']
            model['b2'] -= LR*model_grads['b2']
            model['b1'] -= LR*model_grads['b1']
            model['W'] -= LR*model_grads['W']

        print(total_correct_train/np.float(len(x_train) ) )


time2 = time.time()
print(time2-time1)

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

```
0.8864
0.9369
0.9479166666666666
0.95695
0.9606166666666667
0.9636
0.98255
0.9850833333333333
0.9865833333333334
0.9866
0.98675
0.9882666666666666
0.9882
0.9878333333333333
0.9885833333333334
0.98805
0.9892166666666666
0.9881333333333333
0.9881166666666666
0.9886333333333334
554.1279056072235
0.9763
```

Accuracy for Training set
No. of epochs = 20
LR (begin) = 0.01
LR schedule provided

Accuracy for Test Set (97.63%)