Goncalo Mendes

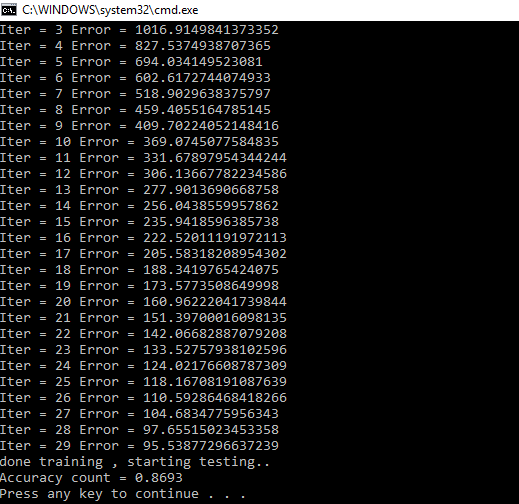
Deep Learning

In this assignment, we transferred our work and understanding of a Dense Neural Network into an Object-oriented program (OOP), that would allow us to easily change the configuration of the network without major implementation changes. Using the base architecture shown in class, I wrote down all the variables necessary to operate within a layer and all the variables to expand to the network once we have a layer of neurons created. Using the math derivations from the last assignment (3), I was able to make sure the dimensions of the arrays were correct for all the necessary parameters, such as the output of the sum, the activation function and the weights/biases, as well as the backpropagation elements, like the gradients.

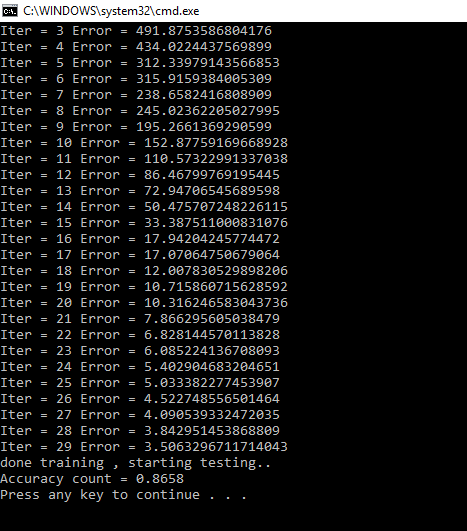
Starting with the layer, the information we need to operate in it is the inputs, which can be the data if we are in the first layer, or the previous layer outputs for each neuron. Then we define what we want each base element in a layer to do. In this case, that element is the neuron and what each must do, is compute the sum of the inputs multiplied by the weights plus the biases and apply an activation function on it. All these expressions were defined, including the possibilities of the 4 activation functions we have studied so far, Sigmoid, RELU, Tanh and Softmax. Using enumerations in classes we can call the computations within the layer with a parameter that specifies which activation function to use on that layer.

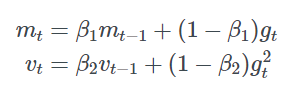
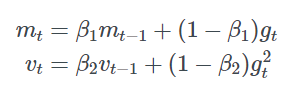
In the network, we just need to connect the layers computing the forward pass until we get to the loss and further create a method that does the backpropagation.

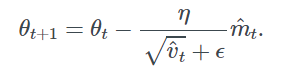
For the first part, with no enhancements or optimizers, for 30 epochs on the 10000 images of the MNIST data set, we get approximately the same results as in the previous assignment, which means the transition was smooth and nothing was unintentionally changed. For Sigmoid in all layers and softmax activation function in the last layer:



For Tanh the result was very close, but we can notice a good improvement on the error with the RELU activation function, with the necessary adjustment on the learning rate. We get a smaller error after the same 30 epochs, still with softmax in the last layer. The accuracy increases a bit too.

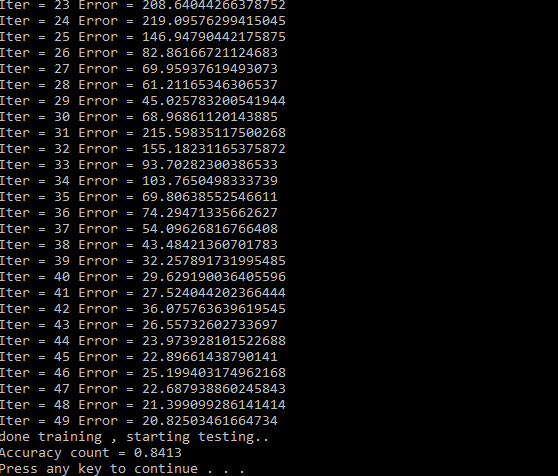


In the second part, we implemented the Adam Optimizer, using the information in the handout provided in the webpage. We start by estimating the mean and variance through the moments method ( mt and vt ) and then replace the same expressions within themselves to correct the running ones



This way, as opposed to Stochastic gradient descent that maintains a single learning rate for all weight updates and the learning rate, here a learning rate is maintained for each network weight (parameter) and separately adapted as learning unfolds.

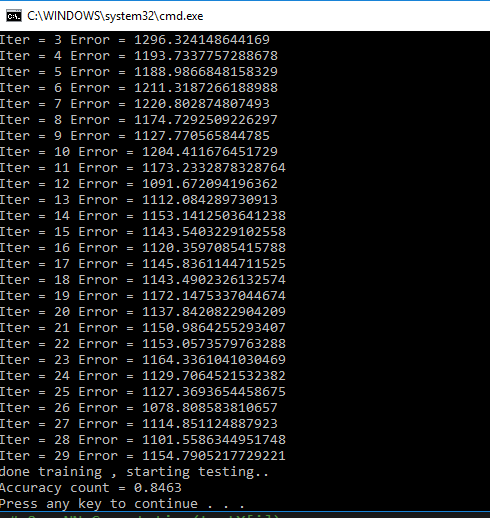
The method computes individual adaptive learning rates for different parameters from estimates of first and second moments of the gradients, and updates them accordingly. The resulting accuracy rounds the 84%.



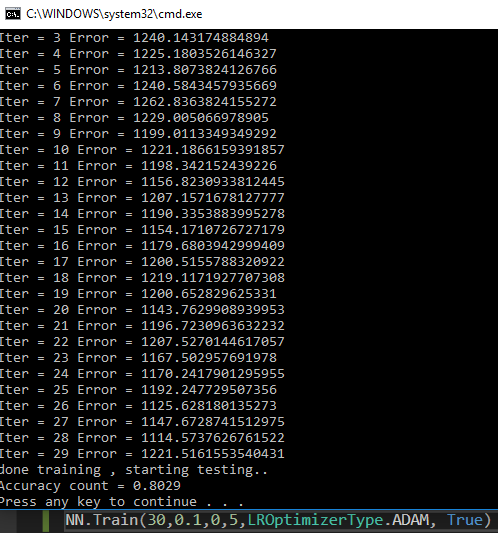
For the final part, we derived the math shown in class for Batch Normalization, to include it as a new block inside our neuron. The main goal is to scale the data points in relation to each other and further center them around the mean and we use the backpropagation algorithm idea to learn the scaling factors. Below are the derivatives for the batch normalization. Normalization by definition is diving the components of the data by its norm, to balance out sizes towards the same scale and equally distance them from the origin.

[ScreenShots of the Math] !!!

The results with Batch Normalization with ADAM Optimization



Results for Batch Normalization without ADAM Optimization



Source Code :

class ActivationType(object):

SIGMOID = 1

RELU = 2

SOFTMAX = 3

TANH = 4

class GradDescType(object):

STOCHASTIC = 1

BATCH = 2

MINIBATCH = 3

class LROptimizerType(object):

NONE = 1

ADAM = 2

class BatchNormMode(object):

TRAIN = 1

TEST = 2

Layer.py

import numpy as np

from BatchNormMode import BatchNormMode

from LROptimizerType import LROptimizerType

from ActivationType import ActivationType

class Layer(object):

def \_\_init\_\_(self, numNeurons, numNeuronsPrevLayer, batchsize,LastLayer = False, dropOut = 0.2, activationType=ActivationType.SIGMOID):

## initialize all the model parameters declared above, necessary for the NN. ( Weights, Biases, Gradients ).

self.numNeurons = numNeurons

self.LastLayer = LastLayer

self.numNeuronsPrevLayer = numNeuronsPrevLayer

self.activationFunction = activationType

self.w = np.random.uniform(low = -0.1, high = 0.1, size=(numNeurons, numNeuronsPrevLayer))

self.b = np.random.uniform( low = -1, high = 1, size=(numNeurons))

self.delta = np.zeros((numNeurons, numNeuronsPrevLayer))

#self.a = np.zeros((numNeurons,1))

self.df = np.zeros((numNeurons))

self.wgrad = np.zeros((numNeurons, numNeuronsPrevLayer)) ## Matrix W in Notes

self.bgrad = np.zeros((numNeurons))

self.zeroout = None

self.dropOut = dropOut

#-------------BATCH NORMALIZATION--------------

self. mu = np.zeros((numNeurons)) # batch mean

self.sigma2 = np.zeros((numNeurons)) # sigma^2 for batch

self.epsilon = 1e-6

self.gamma = np.random.rand(1)

self.beta= np.random.rand(1)

self.S = np.zeros((numNeurons,numNeuronsPrevLayer))

self.Shat = np.zeros((numNeurons,numNeuronsPrevLayer))

self.Sb = np.zeros((numNeurons,numNeuronsPrevLayer))

self.runningmu = np.zeros((numNeurons))

self.runningsigma2 = np.zeros((numNeurons))

self.dgamma = np.zeros((numNeurons))

self.dbeta = np.zeros((numNeurons))

self.delta = np.zeros((numNeurons,numNeuronsPrevLayer))

self.deltabn = np.zeros((numNeurons,numNeuronsPrevLayer))

#----------------------------------------------------------------

#----------following for implementing ADAM-----------------------

self.mtw = np.zeros((numNeurons,numNeuronsPrevLayer))

self.mtb = np.zeros((numNeurons))

self.vtw = np.zeros((numNeurons,numNeuronsPrevLayer))

self.vtb = np.zeros((numNeurons))

#----------------------------------------------------------------

def sigmoid(self,x):

return 1 / (1 + np.exp(-x)) ##SIGMOID

def TanH(self, x):

return np.tanh(x) ##TANH

def Relu(self, x):

return np.maximum(0,x) ##RELU

def Softmax(self, x):

if (x.shape[0] == x.size):

ex = np.exp(x)

return ex/ex.sum()

ex = np.exp(x)

for i in range(ex.shape[0]):

denom = ex[i,:].sum()

ex[i,:] = ex[i,:]/denom

return ex

def Computation(self,indata, doBatchNorm=False, batchMode = BatchNormMode.TRAIN):

self.S = np.dot(indata,self.w.T) + self.b

if( doBatchNorm == True):

if(batchMode == BatchNormMode.TRAIN):

self.mu = np.mean(self.S, axis=0) # batch mean

self.sigma2 = np.var(self.S,axis=0) # batch sigma^2

self.runningmu = 0.9 \* self.runningmu + (1 - 0.9)\* self.mu

self.runningsigma2 = 0.9 \* self.runningsigma2 + (1 - 0.9)\* self.sigma2

else:

self.mu = self.runningmu

self.sigma2 = self.runningsigma2

self.Shat = (self.S - self.mu)/np.sqrt(self.sigma2 + self.epsilon)

self.Sb = self.Shat \* self.gamma + self.beta

sum = self.Sb

else:

sum = self.S

if( self.activationFunction == ActivationType.SIGMOID ) : ## SIGMOID ACTIVATION FUNCTION

self.a = self.sigmoid(sum)

self.df = self.a \* (1- self.a)

if( self.activationFunction==ActivationType.RELU) : ## RELU ACTIVATION FUNCTION

self.a = self.Relu(sum)

#self.derivAF = 1.0 \* (self.a > 0)

epsilon=1.0e-6

self.df = 1. \* (self.a > epsilon)

self.df[self.df == 0] = epsilon

if( self.activationFunction==ActivationType.SOFTMAX) : ## SOFTMAX ACTIVATION FUNCTION

self.a = self.Softmax(sum)

self.df = None

if( self.activationFunction==ActivationType.TANH) : ## TANH ACTIVATION FUNCTION

self.a = self.TanH(sum)

self.df = (1 - self.a \* self.a)

if (self.LastLayer == False):

self.zeroout = np.random.binomial(1,self.dropOut,(self.numNeurons))/self.dropOut

self.a = self.a \* self.zeroout

self.df = self.df \* self.zeroout

def ClearWBGrads(self): # Used after updating gradients to clear accumulation and restart new batch.

self.wgrad = np.zeros((self.numNeurons, self.numNeuronsPrevLayer))

self.bGrad = np.zeros((self.numNeurons,1))

Network.py

import math

import numpy as np

from Layer import \*

from GradDescType import GradDescType

from BatchNormMode import BatchNormMode

from LROptimizerType import LROptimizerType

from sklearn.utils import shuffle

class Network(object):

def \_\_init\_\_(self,TrainX,TrainY, numLayers,batchsize,dropOut = 1.0, activationFunction=ActivationType.SIGMOID, lastLayerActivationFunction = ActivationType.SIGMOID):

self.TrainX = TrainX

self.TrainY = TrainY

self.numLayers = numLayers

self.batchsize = batchsize

self.Layers = [] ## Python List with all Layers.

self.lastLayerActivationFunction = lastLayerActivationFunction

self.gradDescType = 1

self.LROptimization = LROptimizerType.ADAM

self.BatchNormMode = BatchNormMode

for i in range(len( self.numLayers )):

if (i == 0): # first layer

layer = Layer(numLayers[i],TrainX.shape[1],batchsize,False,dropOut, activationFunction)

elif (i == len(numLayers)-1): # last layer

layer = Layer(TrainY.shape[1],numLayers[i-1], batchsize,True, dropOut, lastLayerActivationFunction)

else: # intermediate layers

layer = Layer(numLayers[i],numLayers[i-1],batchsize,False,dropOut, activationFunction)

self.Layers.append(layer);

def Computation(self,indata, doBatchNorm=False, batchMode= BatchNormMode.TEST): # Goes through all layers and executes method inside layer called Computation

self.Layers[0].Computation(indata, doBatchNorm, batchMode)

for i in range(1,len(self.numLayers)):

self.Layers[i].Computation(self.Layers[i-1].a, doBatchNorm, batchMode)

return self.Layers[len(self.numLayers)-1].a

def Train(self, epochs, learningRate, lambda1, batchsize=1, LROptimization=LROptimizerType.NONE, doBatchNorm=False):

iter=0

for i in range (epochs) :

loss = 0

self.TrainX, self.TrainY = shuffle(self.TrainX, self.TrainY, random\_state=0)

for j in range(0,self.TrainX.shape[0],batchsize):

# get (X, y) for current minibatch/chunk

X\_train\_mini = self.TrainX[j:j + batchsize]

Y\_train\_mini = self.TrainY[j:j + batchsize]

self.Computation(X\_train\_mini, doBatchNorm, batchMode=BatchNormMode.TRAIN)

if( self.lastLayerActivationFunction == ActivationType.SOFTMAX):

loss += -(Y\_train\_mini \* np.log(self.Layers[len(self.numLayers)-1].a+0.001)).sum() ## LOSS FUNCTION

else:

loss += ((self.Layers[len(self.numLayers)-1].a - Y\_train\_mini) \* (self.Layers[len(self.numLayers)-1].a - Y\_train\_mini)).sum()

k = len(self.numLayers) -1

## Compute Deltas on all layers and all Wgrads and Bgrads for all layers

while(k >= 0):

if (k == len(self.numLayers)-1): # Last Layer

if (self.lastLayerActivationFunction == ActivationType.SOFTMAX):

self.Layers[k].delta = -Y\_train\_mini+ self.Layers[k].a

else:

self.Layers[k].delta = -(Y\_train\_mini-self.Layers[k].a) \* self.Layers[k].df

else: # intermediate layer

self.Layers[k].delta = np.dot(self.Layers[k+1].delta,self.Layers[k+1].w) \* self.Layers[k].df

if (doBatchNorm == True):

self.Layers[k].dbeta = np.sum(self.Layers[k].delta,axis=0)

self.Layers[k].dgamma = np.sum(self.Layers[k].delta \* self.Layers[k].Shat,axis=0)

self.Layers[k].deltabn = (self.Layers[k].delta \* self.Layers[k].gamma)/(batchsize\*np.sqrt(self.Layers[k].sigma2 + self.Layers[k].epsilon )) \* (batchsize -1 - (self.Layers[k].Shat \* self.Layers[k].Shat))

if(k > 0):

prevOut = self.Layers[k-1].a

else:

prevOut = X\_train\_mini

if( doBatchNorm == True):

self.Layers[k].wgrad = np.dot(self.Layers[k].deltabn.T,prevOut)

self.Layers[k].bgrad = self.Layers[k].deltabn.sum(axis=0)

else:

self.Layers[k].wgrad = np.dot(self.Layers[k].delta.T,prevOut)

self.Layers[k].bgrad = self.Layers[k].delta.sum(axis=0)

k = k - 1

iter = iter+1

self.UpdateGradsBiases(learningRate, lambda1, batchsize, LROptimization, iter, doBatchNorm)

print("Iter = " + str(i) + " Error = "+ str(loss))

def UpdateGradsBiases(self, learningRate, lambda1, batchSize, LROptimization, iter, doBatchNorm): # update weights and biases for all layers

beta1 = 0.9

beta2 = 0.999

epsilon = 1e-8

for ln in range(len(self.numLayers)):

if (LROptimization == LROptimizerType.NONE):

self.Layers[ln].w = self.Layers[ln].w - learningRate \* (1/batchSize) \* self.Layers[ln].wgrad - learningRate \* lambda1 \* self.Layers[ln].w.sum()

self.Layers[ln].b = self.Layers[ln].b - learningRate \* (1/batchSize) \* self.Layers[ln].bgrad

elif (LROptimization == LROptimizerType.ADAM):

gtw = self.Layers[ln].wgrad

gtb = self.Layers[ln].bgrad

self.Layers[ln].mtw = beta1 \* self.Layers[ln].mtw + (1 - beta1) \* gtw

self.Layers[ln].mtb = beta1 \* self.Layers[ln].mtb + (1 - beta1) \* gtb

self.Layers[ln].vtw = beta2 \* self.Layers[ln].vtw + (1 - beta2) \* gtw\*gtw

self.Layers[ln].vtb = beta2 \* self.Layers[ln].vtb + (1 - beta2) \* gtb\*gtb

mtwhat = self.Layers[ln].mtw/(1 - beta1\*\*iter)

mtbhat = self.Layers[ln].mtb/(1 - beta1\*\*iter)

vtwhat = self.Layers[ln].vtw/(1 - beta2\*\*iter)

vtbhat = self.Layers[ln].vtb/(1 - beta2\*\*iter)

self.Layers[ln].w = self.Layers[ln].w - learningRate \* (1/batchSize) \* mtwhat /((vtwhat\*\*0.5) + epsilon)

self.Layers[ln].b = self.Layers[ln].b - learningRate \* (1/batchSize) \* mtbhat /((vtbhat\*\*0.5) + epsilon)

if (doBatchNorm == True):

self.Layers[ln].beta = self.Layers[ln].beta - learningRate \* self.Layers[ln].dbeta

self.Layers[ln].gamma = self.Layers[ln].gamma - learningRate \* self.Layers[ln].dgamma

Main:

import os

import sys

import cv2

import matplotlib.pyplot as plt

import numpy as np

from Layer import \*

from Network import \*

from BatchNormMode import BatchNormMode

from LROptimizerType import LROptimizerType

from GradDescType import GradDescType

from sklearn.utils import shuffle

from ActivationType import ActivationType

train = np.empty((1000,28,28),dtype='float64')

trainY = np.zeros((1000,10))

test = np.empty((10000,28,28),dtype='float64')

testY = np.zeros((10000,10)) # Load in the images

i = 0

for filename in os.listdir('C:/Users/Windows User/Desktop/UB Spring 2019/Deep Learning/Assignment 3/Data/Training1000/'):

y = int(filename[0])

trainY[i,y] = 1.0

train[i] = cv2.imread('C:/Users/Windows User/Desktop/UB Spring 2019/Deep Learning/Assignment 3/Data/Training1000/{0}'.format(filename),0)/255.0 # for color, use 1

i = i + 1

i = 0 # read test data

for filename in os.listdir('C:/Users/Windows User/Desktop/UB Spring 2019/Deep Learning/Assignment 3/Data/Test10000'):

y = int(filename[0])

testY[i,y] = 1.0

test[i] = cv2.imread('C:/Users/Windows User/Desktop/UB Spring 2019/Deep Learning/Assignment 3/Data/Test10000/{0}'.format(filename),0)/255.0

i = i + 1

trainX = train.reshape(train.shape[0],train.shape[1]\*train.shape[2])

testX = test.reshape(test.shape[0],test.shape[1]\*test.shape[2])

numLayers = [10,50]

doBatchNorm=True

NN = Network(trainX,trainY,numLayers,10,1.0,ActivationType.RELU, ActivationType.SOFTMAX) # try SOFTMAX

#NN.Train(30,0.1,0.1, GradDescType.STOCHASTIC,1)

#NN.Train(50,0.1,0.0,50,LROptimizerType.NONE, doBatchNorm) # with BatchNorm = True, try with and without ADAM

#(self, epochs, learningRate, lambda1, batchsize=1, LROptimization=LROptimizerType.NONE, doBatchNorm=False):

NN.Train(30,0.1,0,5,LROptimizerType.ADAM, True) # with doBatchNorm = False,90%

print("done training , starting testing..")

accuracyCount = 0

for i in range(testY.shape[0]):

# do forward pass

#a2 = NN.Computation(testX[i])

a2 = NN.Computation(testX[i], doBatchNorm, BatchNormMode.TEST) # determine index of maximum output value

maxindex = a2.argmax(axis = 0)

if (testY[i,maxindex] == 1):

accuracyCount = accuracyCount + 1

print("Accuracy count = " + str(accuracyCount/10000.0))