**CPEG 586 – DEEP LEARNING**

**HOMEWORK 4**

**Name:** Ivan Sangines Escrig

**ID#:** 968606

**Instructor:** Dr. Mahmood

**Date:** April 24, 2019

**TABLE OF CONTENT**

[**INTRODUCTION**](#_Toc446970371) 3

[**SCREENSHOTS:**](#_Toc446970373) 4

[**SOURCE CODE:**](#_Toc446970374) 13

[**CONCLUSION:**](#_Toc446970375) 25

**INTRODUCTION:**

The purpose of this assignment is to improve the previous assignment by converting our NN program to an object oriented view and also incorporating the zero out regularization. The zero out technique consists in “dropping out” some neurons, which means their contribution on training is temporary neutralized. This is done to prevent some neurons to depend on the neighbors in order to learn. Using an object oriented architecture, we will be able to use different activation functions as well as different approaches (MiniBatch, Stochastic etc.), by simple modifications in the calling function from the main. We will be able to specify what type of approach and activation function we want to use on the calling method.

The second part of the assignment will help us understand better how the ADAM algorithm work by implementing it in on our project. The goal of ADAM algorithm is to adjust the model parameters depending on previous adjustments on each weight. This means that we will be adjusting each gradient or learning rate of each weight differently, i.e. W1 might need to be adjusted in a higher rate than W2 then we should use the same learning rate for both of them.

The third and fourth part of the assignment consist in understand better and implement the Batch Normalization. We will first compute all the derivatives used on the backpropagation to further implement them in our project. The Batch normalization will work similarly to the regular NN approach with some extra components, which will normalize our data and scale it around the mean.

**SCREENSHOTS**

**Problem 1:**

In the following screenshots we can see and compare different results while applying different Activation Functions and approaches to our NN. The difference between this approach and the one from the previous assignment is that we now have an object oriented approach and the zero out or dropout. We are now able to compare the different activation functions by simply changing the arguments on the calling function.

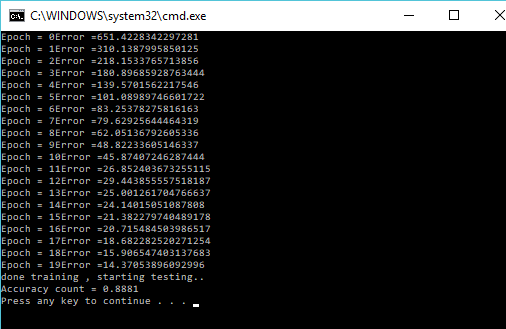
In order to use the different activation functions as well as approaches we just need a small change in the following calling functions:

NN = Network(trainX,trainY,numLayers, "SIGMOID", "SOFTMAX")

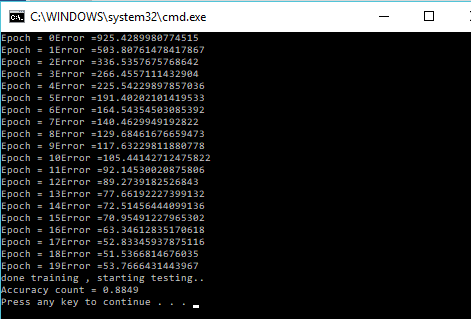
NN.Train(20, 0.1, "STOCHASTIC", " REGULAR ")

STOCHASTIC APPROACH:

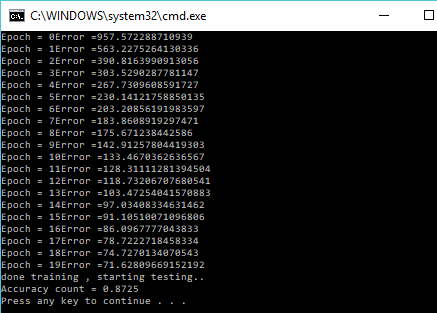
Sigmoid:



TanH:

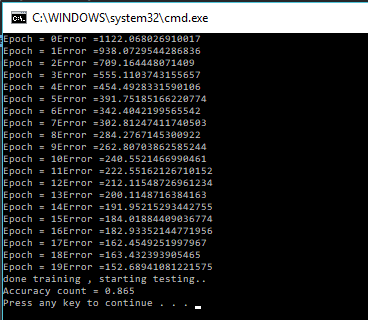


Relu:

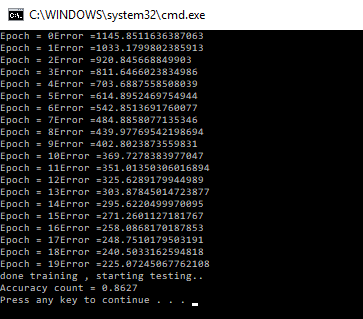


MINIBATCH APPROACH:

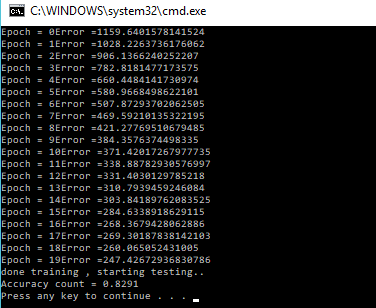
Sigmoid:



TanH:



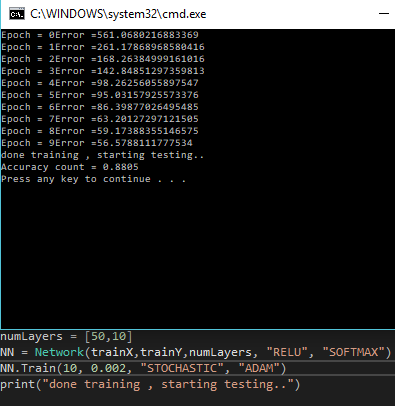
Relu:



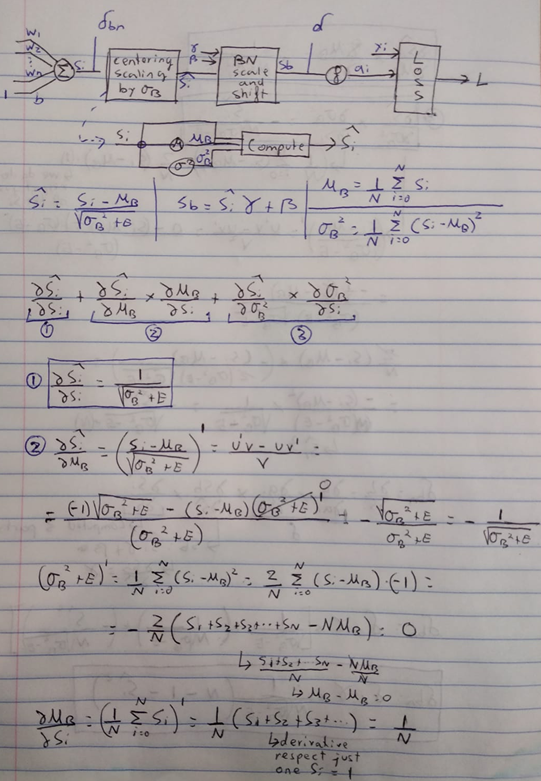
**Problem 2:**

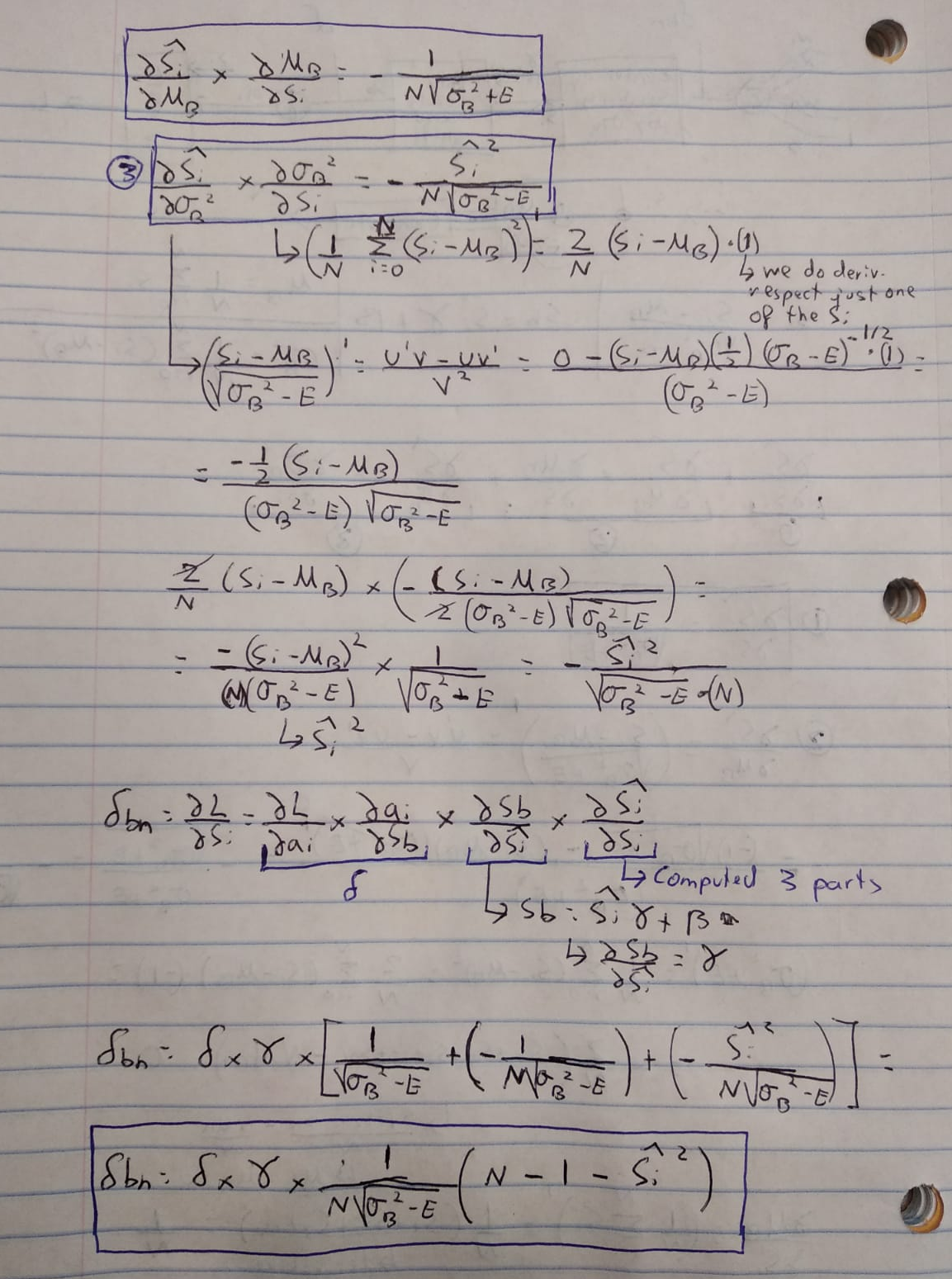
The following screenshots will show the results after applying ADAM optimization. As explained before, Adam optimizer consists in regulating the learning rate of each gradient independently. Not all the weights need to be adjusted using the same learning rate, all the weights are initialized randomly, and some of them might be far from the optimum value and some of them closer that is why we should not adjust all the weights with the same learning rate. The way Adam is implemented consists in adjusting the weights using a percentage of the previous gradient or momentum, the updates of a gradient will use a small percentage of the first updates and a biggest percentage from the last updates.

As we can see on the following output, we obtained an accuracy of 88% just a little bit higher than the one obtained with the STOCHASTIC approach (using RELU and SOFTMAX). However, it took us half of the epochs to achieve the same result



**Problem 3:**

****

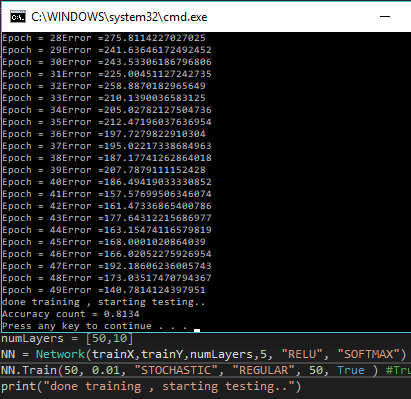
****

**Problem 4:**

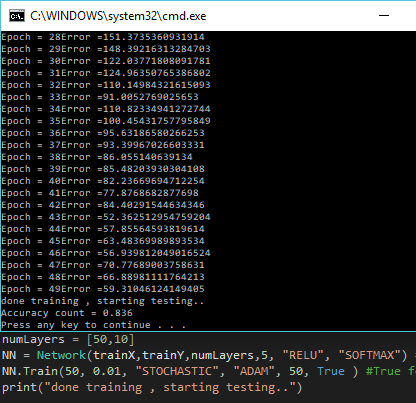
In the following screenshot we can see the results obtained after applying Batch Normalization to a NN.

As we can see from the results below, Batch Normalization also performs better with ADAM algorithm than without. I obtained an accuracy of 81% while not using ADAM and almost 84% of accuracy while using the ADAM regularization.

WITHOUT ADAM:



USING ADAM:



**SOURCE CODE:**

**Problem 1 and Problem 2:**

**Main:**

import os

import sys

import cv2

import numpy as np

from sklearn.utils import shuffle

from Network import \*

def main():

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

trainY = np.zeros((1000,10,1))

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

testY = np.zeros((10000,10,1)) # Load in the images

i = 0

for filename in os.listdir('C:/Users/ivans\_000/Desktop/MASTER/Spring2019/Deep\_Learning/Assignment2\_Sangines/Data/Training1000/'):

y = int(filename[0])

trainY[i,y] = 1.0

train[i] = cv2.imread('C:/Users/ivans\_000/Desktop/MASTER/Spring2019/Deep\_Learning/Assignment2\_Sangines/Data/Training1000/{0}'.format(filename),0)/255.0

i = i + 1

i = 0 # read test data

for filename in os.listdir('C:/Users/ivans\_000/Desktop/MASTER/Spring2019/Deep\_Learning/Assignment2\_Sangines/Data/Test10000'):

y = int(filename[0])

testY[i,y] = 1.0

test[i] = cv2.imread('C:/Users/ivans\_000/Desktop/MASTER/Spring2019/Deep\_Learning/Assignment2\_Sangines/Data/Test10000/{0}'.format(filename),0)/255.0

i = i + 1

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

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

numLayers = [50,10]

NN = Network(trainX,trainY,numLayers, "RELU", "SOFTMAX")

NN.Train(10, 0.002, "STOCHASTIC", "ADAM")

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

accuracyCount = 0

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

# do forward pass

a2 = NN.Compute(testX[i])

# 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))

if \_\_name\_\_ == "\_\_main\_\_":

sys.exit(int(main() or 0))

**Network:**

import numpy as np

import math

from sklearn.utils import shuffle

from Layer import \*

class Network(object):

"""description of class"""

def \_\_init\_\_(self, X, Y, neuronsLayer, activaitonF, activationLast ):

self.X = X

self.Y = Y

self.activationF = activaitonF

self.neuronsLayer = neuronsLayer #list with number of neurons per layer (each index represents one layer)

self.activationLast = activationLast #activation func in last layer

self.Layers = [] #will cointain all created layers (list of Layers objects)

#initializing Layers instances

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

if (i == 0): #First Layer

layer = (Layer(self.neuronsLayer[i], self.X.shape[1],self.activationF,0.8, False,))

elif (i == (len(self.neuronsLayer)-1)): #Last Layer

layer = (Layer(self.Y.shape[1], self.neuronsLayer[i-1],self.activationLast,1, True,))

else: #Mid Layer

layer =(Layer(self.neuronsLayer[i], self.neuronsLayer[i-1],self.activationF,0.8, False,))

self.Layers.append(layer)

def Compute(self, input):

self.Layers[0].Computations(input) #computing it separate because the input is X not a from prev layer

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

self.Layers[i].Computations(self.Layers[i-1].a) #inputs will be the outputs (a) of prev layers

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

def Train (self, epochs, learningRate, gradDesc, optimization="REGULAR", batchSize=1):

updates = 0

for i in range(epochs):

error = 0

self.X, self.Y = shuffle(self.X, self.Y) # shuffle data

for j in range(self.X.shape[0]): #will go throught all the trainin images

self.Compute(self.X[j]) #will create layers and doing forward pass

if (self.activationLast == "SOFTMAX"):

error += -0.5\*(self.Y[j] \* np.log(self.Layers[len(self.neuronsLayer)-1].a + 0.001)).sum()

else:

error += 0.5\*((self.Layers[len(self.neuronsLayer)-1].a - self.Y[j]) \* (self.Layers[len(self.neuronsLayer)-1].a - self.Y[j])).sum()

indexLayer = len(self.neuronsLayer) - 1

while (indexLayer>=0):

if (indexLayer == len(self.neuronsLayer)-1): #last Layer

if (self.activationLast == "SOFTMAX"):

self.Layers[indexLayer].delta = -self.Y[j] + self.Layers[indexLayer].a

else:

self.Layers[indexLayer].delta = -(self.Y[j] - self.Layers[indexLayer].a) \* self.Layers[indexLayer].derivativeAF

else: #mid Layer

#computing the missing part for delta

self.Layers[indexLayer].delta = np.dot(self.Layers[indexLayer+1].W.T, self.Layers[indexLayer+1].delta) \* self.Layers[indexLayer].derivativeAF

if (indexLayer>0):

prevOut = self.Layers[indexLayer-1].a #output from the layer before

else:

prevOut = self.X[j] #whenever it is first layer, we will use the inputs X

self.Layers[indexLayer].gradW += np.dot(self.Layers[indexLayer].delta, prevOut.T)

self.Layers[indexLayer].gradB += self.Layers[indexLayer].delta

indexLayer = indexLayer - 1

#For ADAM Algorithm

updates = updates + 1

#Cheching gradient descent type in order to update gradients differently

if (gradDesc == "MINIBATCH"):

if(j%batchSize == 0):

self.UpdateGradBias(learningRate, batchSize, updates, optimization )

if (gradDesc == "STOCHASTIC"):

self.UpdateGradBias(learningRate, batchSize, updates, optimization )

#For Batch, we will update weights once we finished each epoch. BatchSize is the total number of images

if (gradDesc == "BATCH"):

self.UpdateGradBias(learningRate, self.X.shape[0], updates, optimization)

print("Epoch = " + str(i) + "Error =" + str(error))

def UpdateGradBias (self, learningRate, batchSize, updates, optimization="REGULAR"):

beta1 = 0.9

beta2 = 0.999

epsilon = 1e-8

if (optimization=="REGULAR"):

for layer in range(len(self.neuronsLayer)):

self.Layers[layer].W = self.Layers[layer].W - learningRate \* (1/batchSize) \* self.Layers[layer].gradW

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

self.Layers[layer].ClearGradients()

elif (optimization=="ADAM"):

for lay in range(len(self.neuronsLayer)):

self.Layers[lay].mtw = beta1 \* self.Layers[lay].mtw + (1 - beta1) \* self.Layers[lay].gradW

self.Layers[lay].mtb = beta1 \* self.Layers[lay].mtb + (1 - beta1) \* self.Layers[lay].gradB

self.Layers[lay].vtw = beta2 \* self.Layers[lay].vtw + (1 - beta2) \* self.Layers[lay].gradW \* self.Layers[lay].gradW

self.Layers[lay].vtb = beta2 \* self.Layers[lay].vtb + (1 - beta2) \* self.Layers[lay].gradB \* self.Layers[lay].gradB

mtwFinal = self.Layers[lay].mtw / (1 - beta1\*\*updates )

mtbFinal = self.Layers[lay].mtb / (1 - beta1\*\*updates )

vtwFinal = self.Layers[lay].vtw / (1 - beta2\*\*updates )

vtbFinal = self.Layers[lay].vtb / (1 - beta2\*\*updates )

self.Layers[lay].W = self.Layers[lay].W - learningRate \* (1/batchSize) \* mtwFinal/((vtwFinal\*\*0.5) + epsilon)

self.Layers[lay].b = self.Layers[lay].b - learningRate \* (1/batchSize) \* mtbFinal/((vtbFinal\*\*0.5) + epsilon)

self.Layers[lay].ClearGradients()

**Layer:**

import numpy as np

class Layer(object):

"""description of class"""

def \_\_init\_\_(self, numNeurons, numPrevNeurons, activationFunc, dropRate=0.2, isLastLayer=False):

self.numNeurons = numNeurons

self.numPrevNeurons = numPrevNeurons

self.activationFunc = activationFunc

self.lastLayer = isLastLayer

#Creating matrix needed

self.W = np.random.uniform(-0.1,0.1,(numNeurons,numPrevNeurons))

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

self.delta = np.zeros((numNeurons,1)) #will contain derivativeAF \* all the remaining derivatives

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

self.gradW = np.zeros((numNeurons,numPrevNeurons))

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

self.derivativeAF = np.zeros((numNeurons,1)) #will be part of delta, a\*deriv of activation func. Used in back propagation

self.dropRate = dropRate

self.dropOut = None

#--------------------ADAM variables-------------------------------------------

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

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

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

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

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

def Computations(self, input):

sum = np.dot(self.W,input) + self.b

if (self.activationFunc == "SIGMOID"):

self.a = 1/(1 + np.exp(-sum)) #applying Sigmoid func to each Sum for each neuron

self.derivativeAF = self.a \* (1 - self.a) #used in back propagation, part of the delta

elif (self.activationFunc == "TANH"):

self.a = np.tanh(sum)

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

elif (self.activationFunc == "RELU"):

self.a = np.maximum(0,sum)

self.derivativeAF = 1.0 \* (self.a>0) #1 \* TRUE=1 and 1\*FALSE = 0

elif (self.activationFunc == "SOFTMAX"):

exp = np.exp(sum)

self.a = exp/exp.sum()

self.derivativeAF = None

#DropOut

if (self.lastLayer == False):

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

self.a = self.dropOut \* self.a

self.derivativeAF = self.dropOut \* self.derivativeAF

def ClearGradients(self): #In order to set gradients to 0 after each computed Image

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

self.gradW = np.zeros((self.numNeurons,self.numPrevNeurons))

**Problem 4:**

**Main:**

import os

import sys

import cv2

import numpy as np

from sklearn.utils import shuffle

from Network import \*

def main():

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/ivans\_000/Desktop/MASTER/Spring2019/Deep\_Learning/Assignment2\_Sangines/Data/Training1000/'):

y = int(filename[0])

trainY[i,y] = 1.0

train[i] = cv2.imread('C:/Users/ivans\_000/Desktop/MASTER/Spring2019/Deep\_Learning/Assignment2\_Sangines/Data/Training1000/{0}'.format(filename),0)/255.0

i = i + 1

i = 0 # read test data

for filename in os.listdir('C:/Users/ivans\_000/Desktop/MASTER/Spring2019/Deep\_Learning/Assignment2\_Sangines/Data/Test10000'):

y = int(filename[0])

testY[i,y] = 1.0

test[i] = cv2.imread('C:/Users/ivans\_000/Desktop/MASTER/Spring2019/Deep\_Learning/Assignment2\_Sangines/Data/Test10000/{0}'.format(filename),0)/255.0

i = i + 1

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

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

isBatchNorm = True

numLayers = [50,10]

NN = Network(trainX,trainY,numLayers,5, "RELU", "SOFTMAX") # try SOFTMAX

NN.Train(30, 0.1, "STOCHASTIC", "ADAM", 50, True )

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

accuracyCount = 0

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

# do forward pass

a2 = NN.Compute(testX[i], isBatchNorm, False)

# 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))

if \_\_name\_\_ == "\_\_main\_\_":

sys.exit(int(main() or 0))

**Network:**

import numpy as np

import math

from sklearn.utils import shuffle

from Layer import \*

class Network(object):

"""description of class"""

def \_\_init\_\_(self, X, Y, neuronsLayer,batchSize, activaitonF, activationLast ):

self.X = X

self.Y = Y

self.activationF = activaitonF

self.neuronsLayer = neuronsLayer #list with number of neurons per layer (each index represents one layer)

self.activationLast = activationLast #activation func in last layer

self.Layers = [] #will cointain all created layers (list of Layers objects)

#initializing Layers instances

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

if (i == 0): #First Layer

layer = (Layer(self.neuronsLayer[i], self.X.shape[1],self.activationF,batchSize,0.8, False,))

elif (i == (len(self.neuronsLayer)-1)): #Last Layer

layer = (Layer(self.Y.shape[1], self.neuronsLayer[i-1],self.activationLast,batchSize, True,))

else: #Mid Layer

layer =(Layer(self.neuronsLayer[i], self.neuronsLayer[i-1],self.activationF,batchSize,0.8, False,))

self.Layers.append(layer)

def Compute(self, input, isBatch = False, isTrain = False):

self.Layers[0].Computations(input, isBatch, isTrain) #computing it separate because the input is X not a from prev layer

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

self.Layers[i].Computations(self.Layers[i-1].a, isBatch, isTrain) #inputs will be the outputs (a) of prev layers

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

def Train (self, epochs, learningRate, gradDesc, optimization="REGULAR", batchSize=1, isBatchNorm = False):

updates = 0

for i in range(epochs):

error = 0

self.X, self.Y = shuffle(self.X, self.Y) # shuffle data

for j in range(0,self.X.shape[0], batchSize ): #will go throught all the trainin images

#--------------------IF NOT DOING BATCH, X and Y WILL JUST HAVE 1 IMAGE. ---------------------

#-----------WHEN DOING BATCHNORM, X AND Y WILL HAVE AS MANY IMAGES AS THE BATCH SIZE----------

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

y\_train\_mini = self.Y[j:j + batchSize]

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

self.Compute(X\_train\_mini, isBatchNorm, True) #will call TRAIN in batch

if (self.activationLast == "SOFTMAX"):

error += -0.5\*(y\_train\_mini \* np.log(self.Layers[len(self.neuronsLayer)-1].a + 0.001)).sum()

else:

error += 0.5\*((self.Layers[len(self.neuronsLayer)-1].a - y\_train\_mini) \* (self.Layers[len(self.neuronsLayer)-1].a - y\_train\_mini)).sum()

indexLayer = len(self.neuronsLayer) - 1

while (indexLayer>=0):

if (indexLayer == len(self.neuronsLayer)-1): #last Layer

if (self.activationLast == "SOFTMAX"):

self.Layers[indexLayer].delta = -y\_train\_mini + self.Layers[indexLayer].a

else:

self.Layers[indexLayer].delta = -(y\_train\_mini - self.Layers[indexLayer].a) \* self.Layers[indexLayer].derivativeAF

else: #mid Layer

#computing the missing part for delta

self.Layers[indexLayer].delta = np.dot(self.Layers[indexLayer+1].delta, self.Layers[indexLayer+1].W) \* self.Layers[indexLayer].derivativeAF

#---------------------CHECKING IF BATCHNORM-------------------------------------------

#will compute derivatives and deltas for BATCHNORM

if (isBatchNorm):

#COMPUTING DERIVATIVES

self.Layers[indexLayer].derivBeta = np.sum(self.Layers[indexLayer].delta)

self.Layers[indexLayer].derivGama = np.sum(self.Layers[indexLayer].delta \* self.Layers[indexLayer].Shat)

self.Layers[indexLayer].deltaBatch = (self.Layers[indexLayer].delta \* self.Layers[indexLayer].gamma)/(batchSize \* np.sqrt(self.Layers[indexLayer].sigma2+self.Layers[indexLayer].epsilon))\*(batchSize - 1 - self.Layers[indexLayer].Shat\*self.Layers[indexLayer].Shat)

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

if (indexLayer>0):

prevOut = self.Layers[indexLayer-1].a #output from the layer before

else:

prevOut = X\_train\_mini #whenever it is first layer, we will use the inputs X

if(isBatchNorm):

#UPDATES FOR BATCHNORM

self.Layers[indexLayer].gradW = np.dot(self.Layers[indexLayer].deltaBatch.T, prevOut)

self.Layers[indexLayer].gradB = self.Layers[indexLayer].deltaBatch.sum()

else:

self.Layers[indexLayer].gradW += np.dot(self.Layers[indexLayer].delta, prevOut.T)

self.Layers[indexLayer].gradB += self.Layers[indexLayer].delta

indexLayer = indexLayer - 1

#For ADAM Algorithm

updates = updates + 1

#Cheching gradient descent type in order to update gradients differently

if (gradDesc == "MINIBATCH"):

if(j%batchSize == 0):

self.UpdateGradBias(learningRate, batchSize, updates, optimization, isBatchNorm )

if (gradDesc == "STOCHASTIC"):

self.UpdateGradBias(learningRate, batchSize, updates, isBatchNorm, optimization )

#For Batch, we will update weights once we finished each epoch. BatchSize is the total number of images

if (gradDesc == "BATCH"):

self.UpdateGradBias(learningRate, self.X.shape[0], updates, optimization, isBatchNorm)

print("Epoch = " + str(i) + "Error =" + str(error))

def UpdateGradBias (self, learningRate, batchSize, updates, isBatchNorm, optimization="REGULAR"):

beta1 = 0.9

beta2 = 0.999

epsilon = 1e-8

for lay in range(len(self.neuronsLayer)):

if (optimization=="REGULAR"):

self.Layers[lay].W = self.Layers[lay].W - learningRate \* (1/batchSize) \* self.Layers[lay].gradW

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

self.Layers[lay].ClearGradients()

elif (optimization=="ADAM"):

self.Layers[lay].mtw = beta1 \* self.Layers[lay].mtw + (1 - beta1) \* self.Layers[lay].gradW

self.Layers[lay].mtb = beta1 \* self.Layers[lay].mtb + (1 - beta1) \* self.Layers[lay].gradB

self.Layers[lay].vtw = beta2 \* self.Layers[lay].vtw + (1 - beta2) \* self.Layers[lay].gradW \* self.Layers[lay].gradW

self.Layers[lay].vtb = beta2 \* self.Layers[lay].vtb + (1 - beta2) \* self.Layers[lay].gradB \* self.Layers[lay].gradB

mtwFinal = self.Layers[lay].mtw / (1 - beta1\*\*updates )

mtbFinal = self.Layers[lay].mtb / (1 - beta1\*\*updates )

vtwFinal = self.Layers[lay].vtw / (1 - beta2\*\*updates )

vtbFinal = self.Layers[lay].vtb / (1 - beta2\*\*updates )

self.Layers[lay].W = self.Layers[lay].W - learningRate \* (1/batchSize) \* mtwFinal/((vtwFinal\*\*0.5) + epsilon)

self.Layers[lay].b = self.Layers[lay].b - learningRate \* (1/batchSize) \* mtbFinal/((vtbFinal\*\*0.5) + epsilon)

self.Layers[lay].ClearGradients()

if (isBatchNorm == True):

self.Layers[lay].beta = self.Layers[lay].beta - learningRate \*self.Layers[lay].derivBeta

self.Layers[lay].gamma = self.Layers[lay].gamma - learningRate \* self.Layers[lay].derivGama

**Layer:**

import numpy as np

class Layer(object):

"""description of class"""

def \_\_init\_\_(self, numNeurons, numPrevNeurons, activationFunc, batchSize=1, dropRate=0.2, isLastLayer=False):

self.numNeurons = numNeurons

self.numPrevNeurons = numPrevNeurons

self.activationFunc = activationFunc

self.lastLayer = isLastLayer

self.batchSize = batchSize #used for dimentions

#Creating matrix needed

self.W = np.random.uniform(-0.1,0.1,(numNeurons,numPrevNeurons))

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

self.delta = np.zeros((numNeurons)) #will contain derivativeAF \* all the remaining derivatives

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

self.gradW = np.zeros((numNeurons,numPrevNeurons))

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

self.derivativeAF = np.zeros((numNeurons)) #will be part of delta, a\*deriv of activation func. Used in back propagation

self.dropRate = dropRate

self.dropOut = None

#--------------------ADAM variables-------------------------------------------

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

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

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

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

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

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

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

self.sigma2 = np.zeros((self.numNeurons))

self.epsilon = 1e-6

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

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

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

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

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

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

self.sigma2Run = np.zeros((self.numNeurons))

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

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

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

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

def Computations(self, input, isBatch = False, isTrain = False):

self.S = np.dot(input, self.W.T) + self.b

if (isBatch):

#We have to diferenciate between Train and Test

#-----------------------TRAIN------------------------------------

if(isTrain):

self.mu = np.mean(self.S)

self.sigma2 = np.var(self.S)

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

self.sigma2Run = 0.9 \* self.sigma2Run + (1 - 0.9)\* self.sigma2

#------------------TEST-------------------------------------------

else:

self.mu = self.muRun

self.sigma2 = self.sigma2Run

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.activationFunc == "SIGMOID"):

self.a = 1/(1 + np.exp(-sum)) #applying Sigmoid func to each Sum for each neuron

self.derivativeAF = self.a \* (1 - self.a) #used in back propagation, part of the delta

elif (self.activationFunc == "TANH"):

self.a = np.tanh(sum)

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

elif (self.activationFunc == "RELU"):

self.a = np.maximum(0,sum)

#self.derivativeAF = 1.0 \* (self.a>0) #1 \* TRUE=1 and 1\*FALSE = 0

epsilon = 1.06e-6

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

self.derivativeAF[self.derivativeAF == 0] = epsilon

elif (self.activationFunc == "SOFTMAX"):

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

ex = np.exp(sum)

self.a = ex/ex.sum()

else:

ex = np.exp(sum)

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

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

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

self.a = ex

self.derivativeAF = None

#DropOut

if (self.lastLayer == False):

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

self.a = self.a\*self.dropOut

self.derivativeAF = self.dropOut \* self.derivativeAF

def ClearGradients(self): #In order to set gradients to 0 after each computed Image

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

self.gradW = np.zeros((self.numNeurons,self.numPrevNeurons))

**CONCLUSION:**

This assignment really helped me improve my knowledge about NN by converting the previous assignment to an Object Oriented architecture. This assignment also helped me to understand how ADAM regularization works. I realized the ADAM algorithm adjusts the weights and biases using a percentage of the previous gradients or momentums, giving more importance or taking more percentage from the nearest previous momentums or gradients. As we can see in the results, ADAM seems to improve the performance of the NN since not all the weights and biases need to me adjusted using the same rate.

The last part of the assignment helped me to understand Batch Normalization by doing the changes needed to the already created NN. Writing down the derivative and a picture of the NN using Batch Normalization, really helped me to understand the new implementations better.

After this assignment, I can understand better the differences between Batch Normalization and the regular NN, since the idea of Batch Normalization is to normalize and scale our data around the mean.