**SIAMESE USING CNN**

**MASTER PROJECT**

**Name:** Ivan Sangines Escrig

**ID#:** 968606

**Instructor:** Dr. Mahmood

**Date:** May 31, 2019

**TABLE OF CONTENT**

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

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

[**SOURCE CODE:**](#_Toc446970373) 9

[**CONCLUSION:**](#_Toc446970375) 16

**INTRODUCTION:**

The purpose of this assignment is to understand better how to work with TensorFlow library by creating a Siamese network, which uses a CNN in order to obtain the features of the incoming images. Once we obtain the features from the inputs (and after flattening), the embedding will occur by using the flattened features as an input of the embedding NN.

In order to train this Siamese Network, we have to divide the training in different parts:

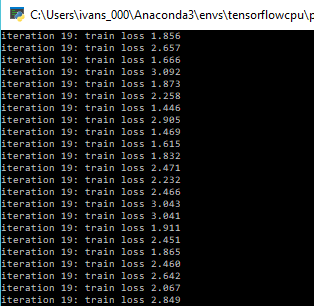
* The first training will consists in training the CNN with the NN in order to be able to extract good and meaningful features for the image to further create a strong and unique embedding.
* Second part of the training will consist in training the Siamese Network for classification. Once the Siamese learns how to extract features from the images and how to create a good embedding, we will fix the weights, biases and kernels for the CNN and NN. Then, we will run the training data set through the network and using a Cross Entropy loss function, we will train the Siamese Network for classification.

After training the Siamese Network, we are ready to test it by running the test data set and computing accuracy to verify the Siamese Network was properly trained.

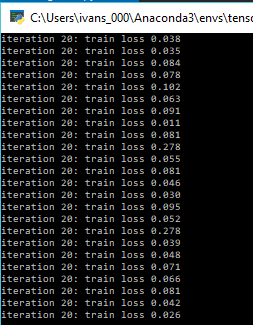
The second part of my Master’s project will consists in training the Siamese Network using a different embedding, which will be a combination of both outputs, and a new loss function. The results will be compared in order to show improvements between both architectures.

**SCREENSHOTS**

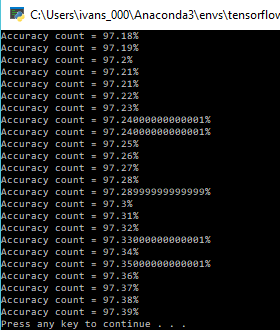
On the following screenshot, we can see the training error for the first part of the training. Here we are training the Siamese Network to extract meaningful features from the incoming images as well as, learning how to create a strong and unique embedding. The initial error for this part of the training was around 15, even though I was able to reduce the error to 2, it is still a little bit high since the MNIST data set is a simple data set with no much information (simple digits). Better error can be achieved if a data set with more information is used (i.e. faces data set).



On this screenshot, we can see the loss for the classification training. This error started around 5 and dropped down to 0.02, which is a pretty low error, which will give us a high accuracy at the end.



On this last screenshot, we will see the accuracy obtained after the Siamese training is completed. As mentioned before, in order to obtain the accuracy of the Siamese Network, after the training is completed, the weights, biases and kernels are fixed to further test the network using our test data set. After testing the Siamese Network, I obtained an accuracy of 97.4%, which is higher than all the other examples tried during the course. One of the main reasons is that in this project we are using a CNN in order to obtain features, which are the meaningful parts of our images, and then using them as inputs for the embedding. In other words, we are just using the meaningful part of the images in order to create a stronger and unique embedding.

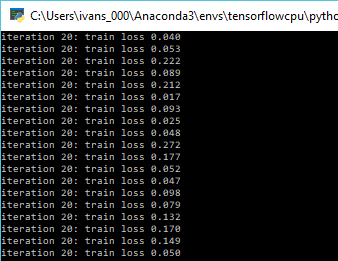
****

**SECOND PART:**

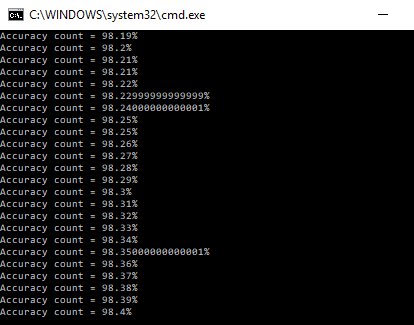
The second part of the Master’s project consisted in create a Siamese Network using a different loss function, while training the embedding. The idea of this new loss function consists in creating a better and stronger embedding by combining both outputs from the “two” CNN.

When we are training the Siamese Network in order to learn how to create meaningful and unique embedding, we will now combine the two training outputs by doing de difference between them both to obtain a final embedding. Once we obtain the new embedding, we will now use it as the output to train the network with the new loss function.

The following screenshot shows the loss training for the Siamese Network using the new loss function. The final error obtain for this new training is 0.05 which is about the same if we compare it with the previous loss error. However, the main reason might be that the MNIST data set used is not very meaningful, that is because this type of data set represents simple digits with no many “important” features.

****

On the following screenshot, we can see the accuracy after applying the new loss function to our Siamese Network. As we can see, we now obtained an accuracy of 98% compared to the previous 97%. Even though we obtain an improvemen of 1%t, a better data set could have sown us better results.



**SOURCE CODE:**

**Main:**

from tensorflow.examples.tutorials.mnist import input\_data

import tensorflow as tf

import matplotlib

import matplotlib.pyplot as plt

from Siamese import SiameseCustom

import numpy as np

from keras.utils import to\_categorical

import os

import cv2

def visualize(embed, labels):

labelset = set(labels.tolist())

fig = plt.figure(figsize=(8,8))

ax = fig.add\_subplot(111)

for label in labelset:

indices = np.where(labels == label)

ax.scatter(embed[indices,0], embed[indices,1], label = label, s = 20)

ax.legend()

plt.show()

plt.close()

def readInputData():

train = np.empty((60000,28,28),dtype="float64")

trainYOH = np.zeros((60000,10)) # one hot

trainY = np.zeros((60000)) # just digit i.e, 5 or 7 etc..

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

testYOH = np.zeros((10000,10))

testY = np.zeros((10000))

i = 0

for filename in os.listdir("C:/Users/ivans\_000/Desktop/MASTER/Spring2019/Deep\_Learning/Assignment2\_Sangines/Data/TrainingAll60000"):

y = int(filename[0])

trainYOH[i,y] = 1.0

trainY[i] = y

train[i] = cv2.imread("C:/Users/ivans\_000/Desktop/MASTER/Spring2019/Deep\_Learning/Assignment2\_Sangines/Data/TrainingAll60000/{0}".format(filename),0)/255.0 # for color, use 1

i = i + 1

if i%100 == 0:

print(i)

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])

testYOH[i,y] = 1.0

testY[i] = y

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

if i%100 == 0:

print(i)

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

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

#return trainX, trainY, trainYOH, testX, testY, testYOH

return train, trainY, trainYOH, test, testY, testYOH

def main():

# Load MNIST dataset

trainX, trainY, trainYOH, testX, testY, testYOH = readInputData() # file based input data

print("Loaded data")

siamese = SiameseCustom() # for custom input data read from images files

siamese.trainSiamese(trainX, trainY,20,100)

embed = siamese.test\_model(testX)

embed = embed.reshape([-1, 2])

siamese.trainSiameseForClassification(trainX, trainY,25,100)

visualize(embed, testY)

siamese.computeAccuracy(testX, testY)

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

main()

**Siamese class:**

import tensorflow as tf

import os

from tensorflow.examples.tutorials.mnist import input\_data

from keras.utils import to\_categorical

import numpy as np

from sklearn.utils import shuffle

class SiameseCustom(object):

def \_\_init\_\_(self):

#----set up place holders for inputs and labels for the siamese network---

# two input placeholders for Siamese network

self.tf\_inputA = tf.placeholder(tf.float32, [None, 28,28], name = "inputA")

self.tf\_inputB = tf.placeholder(tf.float32, [None, 28,28], name = "inputB")

# labels for the image pair # 1: similar, 0: dissimilar

self.tf\_Y = tf.placeholder(tf.float32, [None,], name = "Y")

self.tf\_YOneHot = tf.placeholder(tf.float32, [None,10], name = "YoneHot")

# outputs, loss function and training optimizer

#self.outputA, self.outputB = self.siameseNetwork()

self.outputA, self.outputFinal = self.siameseNetwork()

self.output = self.siameseNetworkWithClassification()

self.loss = self.contastiveLoss()

self.lossCrossEntropy = self.crossEntropyLoss()

self.optimizer = self.optimizer\_initializer()

self.optimizerCrossEntropy = self.optimizer\_initializer\_crossEntropy()

self.saver = tf.train.Saver()

# Initialize tensorflow session

self.sess = tf.Session()

self.sess.run(tf.global\_variables\_initializer())

def layerCNN(self,input\_data, numChannels, numFeatureMaps, kernelSize, pool\_shape, name):

# setup the filter input shape for tf.nn.conv\_2d

# for the first CNN layer, the number of feature maps will be the number of color channles

# 1 for gray scale, 3 for RGB

# for the second CNN layer, numChannels will be the number of feature maps in previous CNN layer

filterShape = [kernelSize[0], kernelSize[1], numChannels, numFeatureMaps] #numChaneels = 3 for RGB

# initialize weights and bias for the filter

weights = tf.Variable(tf.truncated\_normal(filterShape, stddev=0.03), name=name+'\_W')

bias = tf.Variable(tf.truncated\_normal([numFeatureMaps]), name=name+'\_b')

# setup the convolutional layer operation

out\_layer = tf.nn.conv2d(input\_data, weights, [1, 1, 1, 1], padding='SAME')

out\_layer += bias

# apply activation function

output\_layer = tf.nn.relu(out\_layer)

# max pooling on output of feature maps

# ksize = size of the max pooling window (i.e. the area over which the max pooling is done

# It needs to be 4D to match the data tensor - 1x2x2x1

ksize = [1, pool\_shape[0], pool\_shape[1], 1]

# strides defines how the max pooling area is applied on the image - It also matches the data tensor

# max pooling areas starting at x=0, x=2, x=4 etc. through your image. If the stride is 1, we will get max pooling

# strides of 2 in the x and y directions.

strides = [1, 2, 2, 1]

output\_layer\_pooling = tf.nn.max\_pool(output\_layer, ksize=ksize, strides=strides, padding='SAME')

return output\_layer\_pooling

def layer(self, tf\_input, num\_hidden\_units, variable\_name, trainable=True):

# tf\_input: batch\_size x n\_features

# num\_hidden\_units: number of hidden units

tf\_weight\_initializer = tf.random\_normal\_initializer(mean = 0, stddev = 0.01)

num\_features = tf\_input.get\_shape()[1]

W = tf.get\_variable(

name = variable\_name + "\_W",

dtype = tf.float32,

shape = [num\_features, num\_hidden\_units],

initializer = tf\_weight\_initializer,

trainable=trainable

)

b = tf.get\_variable(

name = variable\_name + "\_b",

dtype = tf.float32,

shape = [num\_hidden\_units],

trainable=trainable

)

out = tf.add(tf.matmul(tf\_input, W), b)

return out

def network(self, tf\_input, trainable=True):

# Setup FNN

#fc1 = self.layer(tf\_input = tf\_input, num\_hidden\_units = 512,trainable=trainable, variable\_name = "fc1")

#ac1 = tf.nn.relu(fc1)

#fc2 = self.layer(tf\_input = ac1, num\_hidden\_units = 512, trainable=trainable,variable\_name = "fc2")

#ac2 = tf.nn.relu(fc2)

# reshape the input to a 4D tensor. The first value = batch size, last = number of channels, 3 for color

x4D\_input = tf.reshape(tf\_input, [-1, 28, 28, 1])

# create cnn layers

numFeatureMapsLayer1 = 8

numFeatureMapsLayer2 = 16

cnnLayer1Output = self.layerCNN(x4D\_input, 1, numFeatureMapsLayer1, [5, 5], [2, 2], name='cnnLayer1')

cnnLayer2Output = self.layerCNN(cnnLayer1Output, numFeatureMapsLayer1, numFeatureMapsLayer2, [5, 5], [2, 2], name='cnnLayer2')

flattened = tf.reshape(cnnLayer2Output, [-1, 7 \* 7 \* numFeatureMapsLayer2]) # first dim is batch size

# after maxpooling through two layers, the output will be 7x7 x num featuremaps in second CNN layer

# Create NN layer for embeding

fc3 = self.layer(tf\_input = flattened, num\_hidden\_units = 100, trainable=trainable,variable\_name = "fc3")

return fc3

def networkWithClassification(self, tf\_input):

# Setup FNN

fc3 = self.network(tf\_input, trainable=False)

ac3 = tf.nn.relu(fc3)

fc4 = self.layer(tf\_input = ac3, num\_hidden\_units = 80, trainable=True,variable\_name = "fc4")

ac4 = tf.nn.relu(fc4)

fc5 = self.layer(tf\_input = ac4, num\_hidden\_units = 10, trainable=True,variable\_name = "fc5")

return fc5

def siameseNetwork(self):

# Initialze neural network

with tf.variable\_scope("siamese") as scope:

outputA = self.network(self.tf\_inputA)

# share weights

scope.reuse\_variables()

outputB = self.network(self.tf\_inputB)

#For new loss function

outputFin = tf.reduce\_sum(tf.abs(tf.subtract(outputA,outputB)))

return outputA, outputFin

#return outputA, outputB

def siameseNetworkWithClassification(self):

# Initialze neural network

with tf.variable\_scope("siamese",reuse=tf.AUTO\_REUSE) as scope:

#with tf.variable\_scope(&quot;siamese&quot;) as scope:

output = self.networkWithClassification(self.tf\_inputA)

return output

def contastiveLoss(self, margin = 5.0):

with tf.variable\_scope("siamese") as scope:

labels = self.tf\_Y

#---------Using new loss function--------------------------

labels = self.tf\_Y

p = tf.add(tf.sigmoid(self.outputFinal), 1e-6)

l = tf.log(tf.add(tf.subtract(1.0,p),1e-6))

lossSimilar = tf.multiply(tf.subtract(1.0,labels),l)

lossDissimilar = tf.multiply(labels, tf.log(p))

loss = tf.reduce\_mean(tf.add(lossSimilar,lossDissimilar))

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

# Euclidean distance squared

#dist = tf.pow(tf.subtract(self.outputA, self.outputB), 2, name = "Dw")

#Dw = tf.reduce\_sum(dist, 1)

# add a small value 1e-6 to increase the stability of calculating the gradients for sqrt

#Dw2 = tf.sqrt(Dw + 1e-6, name = "Dw2")

# Loss function

#lossSimilar = tf.multiply(labels, tf.pow(Dw2,2), name = "constrastiveLoss\_1")

#lossDissimilar = tf.multiply(tf.subtract(1.0, labels),

#tf.pow(tf.maximum(tf.subtract(margin, Dw2), 0), 2), name = "constrastiveLoss\_2")

#loss = tf.reduce\_mean(tf.add(lossSimilar, lossDissimilar), name ="constrastiveLoss")

return loss

def crossEntropyLoss(self):

labels = self.tf\_YOneHot

lossd = tf.reduce\_mean(tf.nn.softmax\_cross\_entropy\_with\_logits(logits=self.output,labels=labels))

return lossd

def optimizer\_initializer(self):

LEARNING\_RATE = 0.01

RAND\_SEED = 0 # random seed

tf.set\_random\_seed(RAND\_SEED)

# Initialize optimizer

optimizer = tf.train.GradientDescentOptimizer(LEARNING\_RATE).minimize(self.loss)

return optimizer

def optimizer\_initializer\_crossEntropy(self):

LEARNING\_RATE = 0.01

RAND\_SEED = 0 # random seed

tf.set\_random\_seed(RAND\_SEED)

# Initialize optimizer

optimizer = tf.train.AdamOptimizer(LEARNING\_RATE).minimize(self.lossCrossEntropy)

return optimizer

def trainSiamese(self, trainX, trainY, numIterations, batchSize=100):

# Train the network for embeddings via contrastive loss

for i in range(numIterations):

trainX, trainY = shuffle(trainX, trainY, random\_state=0)

for j in range(0, trainX.shape[0], batchSize\*2):# get (X, y) for current minibatch/chunk

input1 = trainX[j:j + batchSize]

y1 = trainY[j:j + batchSize]

input2 = trainX[j+batchSize:j + batchSize+batchSize]

y2 = trainY[j+batchSize:j + batchSize+batchSize]

label = (y1 == y2).astype("float")

\_, trainingLoss = self.sess.run([self.optimizer, self.loss],

feed\_dict = {self.tf\_inputA: input1, self.tf\_inputB: input2, self.tf\_Y:label})

if i % 1 == 0:

print("iteration %d: train loss %.3f" % (i, trainingLoss))

return trainingLoss

def trainSiameseForClassification(self, trainX, trainY,numIterations, batchSize=10):

# Train the network for classification via softmax

for i in range(numIterations):

trainX, trainY = shuffle(trainX, trainY, random\_state=0)

for j in range(0, trainX.shape[0], batchSize):# get (X, y) for current minibatch/chunk

input1 = trainX[j:j + batchSize]

y1 = trainY[j:j + batchSize]

y1c = to\_categorical(y1) # convert labels to one hot

labels = np.zeros(batchSize)

\_, trainingLoss = self.sess.run([self.optimizerCrossEntropy,self.lossCrossEntropy],

feed\_dict = {self.tf\_inputA: input1, self.tf\_inputB: input1,

self.tf\_YOneHot: y1c, self.tf\_Y:labels})

if i % 5 == 0:

print("iteration %d: train loss %.3f" % (i, trainingLoss))

return trainingLoss

def test\_model(self, input):

# Test the trained model

output = self.sess.run(self.outputA, feed\_dict = {self.tf\_inputA: input})

return output

def computeAccuracy(self,testX, testY):

labels = np.zeros(100)

yonehot = np.zeros((100,10))

aout = self.sess.run(self.output, feed\_dict={self.tf\_inputA: testX,self.tf\_inputB: testX,

self.tf\_YOneHot: yonehot, self.tf\_Y:labels})

accuracyCount = 0

testY = to\_categorical(testY) # one hot labels

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

# determine index of maximum output value

maxindex = aout[i].argmax(axis = 0)

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

accuracyCount = accuracyCount + 1

print("Accuracy count = " + str(accuracyCount/testY.shape[0]\*100) + "%")

**CONCLUSION:**

This project helped me understand better how to properly work with TensorFlow library by having to modify the structure of a regular Siamese to a Siamese Network using a CNN. After completing this project, I also understood the importance and difference between a regular Siamese (using NN) and a Siamese using CNN. I realized the importance of trying to focus on the important or meaningful features of the images. Even though, this project was done using the MNIST data set, which does not have too many meaningful features since the images are simple digits with not many information in it, I could see an improvement of a little bit more than 1% while comparing with the different regular Siamese Networks used in class.

On the other hand, one of the main things that could improve this project and make the comparisons more meaningful is to use a different data set. As mention before, MNIST data set is very simple and does not have many meaningful features, that is why all the errors and accuracies in the different networks are really close to each other. However, if we would use a face data set, which will have more information and better features to learn, we would be able to see better improvements and even get an error really close to 0 and an accuracy of 99%.