1. Source code file names:
   1. create\_data.py
   2. HoG.py
   3. neural\_net.py
   4. run\_NN.py
2. Instructions on how to compile and run the code:
   1. First place above four files in a folder.
   2. Inside this folder create four folders as ‘positive\_train’, ‘negative\_train’, ‘positive\_test’, and ‘negative\_test’.
   3. Place all the positive train images in positive train folder and likewise place the other images in the respective folders.
   4. First run the create\_data.py file. This file will create two csv files ‘train.csv’ and ‘test.csv’.
   5. Then run the ‘run\_NN.py’ file. It will display the accuracy and the predictions as outputs.
   6. Or you can run the CV\_Project2.bat batch file which will do steps 4 and 5 for you.
3. Answers:
   1. How did you initialize the weight values of the network?

🡪 Random weights from normal distribution, 250 hidden neurons, and learning rate of 0.01

* 1. How many iterations (or epochs) through the training data did you perform

🡪 After trying various combination, epoch was decided to be 400.

* 1. How did you decide when to stop training?

🡪 When either number of iterations (epochs) are completed or the difference between successive errors is less than 10^-6

* 1. Based on the output value of the output neuron, how did you decide on how to classify the input image into human or not-human?

🡪 The output neuron uses sigmoid as activation function so if the value is less than 0.5 the image is non-human else it is human.

1. Classification results on test images:

|  |  |  |
| --- | --- | --- |
| **Test Image** | **Output value** | **Classification** |
| crop\_000010b | 0.97051742 | 1 |
| crop001008b | 0.98690437 | 1 |
| crop001028a | 0.78469989 | 1 |
| crop001045b | 0.88945977 | 1 |
| crop001047b | 0.91908398 | 1 |
| 00000053a\_cut | 0.41748846 | 0 |
| 00000062a\_cut | 0.22092658 | 0 |
| 00000093a\_cut | 0.03816118 | 0 |
| no\_person\_\_no\_bike\_213\_cut | 0.47318405 | 0 |
| no\_person\_\_no\_bike\_247\_cut | 0.08421976 | 0 |

1. Comments:

Training the neural network may take some time depending upon the size of data and the terminating conditions.

Above is the best accuracy I got while trying different hyper parameters with random initial weights.

1. Normalized test images:

A picture containing photo, indoor, white

Description automatically generated A picture containing indoor, photo

Description automatically generated A picture containing sitting

Description automatically generated  

 A person in a dark room

Description automatically generated A picture containing white, photo, black

Description automatically generated A picture containing indoor

Description automatically generated A picture containing white, outdoor

Description automatically generated

1. Code:
   1. HoG.py:

# importing required libraries

import numpy as np

from skimage.io import imread

from PIL import Image

import math

import sys

class HOG:

def \_\_init\_\_(self, image\_name):

self.image\_name = image\_name

# function to convert color image to greyscale

def greyscale\_operation(self, ip\_img):

op\_img = np.zeros((ip\_img.shape[0], ip\_img.shape[1]))

for i in range(ip\_img.shape[0]):

for j in range(ip\_img.shape[1]):

op\_img[i][j] = np.round\_(0.299\*ip\_img[i][j][0] + 0.587\*ip\_img[i][j][1] + 0.114\*ip\_img[i][j][2])

return op\_img

# function for implementing gradient operation using prewitt's edge detector

def gradient\_operation(self, ip\_img):

# prewitt's vertical and horizontal kernels

Gx\_kernel = ([[-1, 0, 1]] \* 3)

Gy\_kernel = ([1, 1, 1], [0, 0, 0], [-1, -1, -1])

Gx = np.zeros((ip\_img.shape[0], ip\_img.shape[1]))

Gy = np.zeros((ip\_img.shape[0], ip\_img.shape[1]))

G = np.zeros((ip\_img.shape[0], ip\_img.shape[1]))

Theta = np.zeros((ip\_img.shape[0], ip\_img.shape[1]))

# calculating Gx and Gy using prewitt's edge detector

for i in range(ip\_img.shape[0]):

for j in range(ip\_img.shape[1]):

# pixels for which part of the prewitt's mask goes outside of the image border

if i < 1 or j < 1 or ip\_img.shape[0] - i <= 1 or ip\_img.shape[1] - j <= 1:

continue

else:

arr1 = ip\_img[i - 1:i + 2, j - 1:j + 2]

Gx[i][j] = np.sum(np.multiply(arr1, Gx\_kernel))/3

Gy[i][j] = np.sum(np.multiply(arr1, Gy\_kernel))/3

G[i][j] = round(math.sqrt(Gx[i][j]\*\*2 + Gy[i][j]\*\*2)/math.sqrt(2))

if Gx[i][j] == 0 and Gy[i][j] == 0:

Theta[i][j] = 0

else:

theta = np.arctan2(Gy[i][j], Gx[i][j])\*180/np.pi

if theta < 0:

theta += 180

if theta >= 170:

theta -= 180

Theta[i][j] = theta

return G, Theta

# HoG implementation

def hog\_operation(self, ip\_img, ip\_theta):

# HoG bins table

bins\_table = {0:[-10,10], 20:[10,30], 40:[30,50], 60:[50,70], 80:[70,90], 100:[90,110], 120:[110,130], 140:[130, 150], 160:[150,170]}

histo\_center = np.zeros((ip\_img.shape[0], ip\_img.shape[1]))

# First the bin centers for each pixel are identified

for i in range(ip\_theta.shape[0]):

for j in range(ip\_theta.shape[1]):

for x in bins\_table.keys():

if bins\_table[x][0] <= ip\_theta[i][j] < bins\_table[x][1]:

histo\_center[i][j] = x

# hog histograms for each cell are created

histo\_list = np.zeros((int(ip\_img.shape[0]/8), int(ip\_img.shape[1]/8), 9))

a = 0

k = 0

for x in range(int(ip\_theta.shape[0]/8)):

b = 0

c = 0

for y in range(int(ip\_theta.shape[1]/8)):

for i in range(a, a+8):

for j in range(b, b+8):

bin\_no = int(histo\_center[i][j]/20)

if ip\_theta[i][j] == histo\_center[i][j]:

histo\_list[k][c][bin\_no] += ip\_img[i][j]

else:

diff1 = abs(bins\_table[histo\_center[i][j]][0] - ip\_theta[i][j])

diff2 = abs(bins\_table[histo\_center[i][j]][1] - ip\_theta[i][j])

if ip\_theta[i][j] < histo\_center[i][j]:

histo\_list[k][c][bin\_no] += (diff2/20)\*ip\_img[i][j]

histo\_list[k][c][bin\_no - 1] += (diff1/20)\*ip\_img[i][j]

else:

histo\_list[k][c][bin\_no] += (diff1/20)\*ip\_img[i][j]

if bin\_no == 8:

histo\_list[k][c][0] += (diff2/20)\*ip\_img[i][j]

else:

histo\_list[k][c][bin\_no + 1] += (diff2/20)\*ip\_img[i][j]

b += 8

c += 1

a += 8

k += 1

# block\_array will store the normalized histogram block wise

block\_array = np.zeros((histo\_list.shape[0] - 1, histo\_list.shape[1] - 1, 36))

# hog\_discriptor will create a one dimensional array of all the normalized hirtograms

hog\_discriptor\_op = []

for i in range(histo\_list.shape[0] - 1):

for j in range(histo\_list.shape[1] - 1):

l1 = histo\_list[i][j]

l2 = histo\_list[i+1][j]

l3 = histo\_list[i][j+1]

l4 = histo\_list[i+1][j+1]

norm\_factor = math.sqrt(np.sum(np.square(l1)) + np.sum(np.square(l2)) + np.sum(np.square(l3)) + np.sum(np.square(l4)))

np.seterr(divide='ignore', invalid='ignore')

l1\_, l2\_, l3\_, l4\_ = np.true\_divide(l1, norm\_factor), np.true\_divide(l2, norm\_factor), np.true\_divide(l3, norm\_factor), np.true\_divide(l4, norm\_factor)

block\_array[i][j] = np.concatenate((l1\_, l2\_, l3\_, l4\_))

hog\_discriptor\_op.extend(l1\_.tolist())

hog\_discriptor\_op.extend(l2\_.tolist())

hog\_discriptor\_op.extend(l3\_.tolist())

hog\_discriptor\_op.extend(l4\_.tolist())

return np.array(hog\_discriptor\_op).reshape((1, len(hog\_discriptor\_op)))

# main function

def hog(self):

image\_name = self.image\_name

img = imread(image\_name)

gs\_img = self.greyscale\_operation(img)

grad\_img, theta = self.gradient\_operation(gs\_img)

hog\_discriptor = self.hog\_operation(grad\_img, theta)

return hog\_discriptor

* 1. create\_data.py:

import HoG as HoG

import numpy as np

import os

# function to create data

def create\_data():

files = os.listdir('./positive\_train')

for i in range(len(files)):

hog\_obj = HoG.HOG('./positive\_train/'+files[i])

hog = hog\_obj.hog()

if i == 0:

train\_X\_1 = hog

else:

train\_X\_1 = np.vstack((train\_X\_1, hog))

train\_y\_1 = np.full((train\_X\_1.shape[0],1), 1)

files = os.listdir('./negative\_train')

for i in range(len(files)):

hog\_obj = HoG.HOG('./negative\_train/'+files[i])

hog = hog\_obj.hog()

if i == 0:

train\_X\_0 = hog

else:

train\_X\_0 = np.vstack((train\_X\_0, hog))

train\_y\_0 = np.full((train\_X\_0.shape[0],1), 0)

train\_x = np.vstack((train\_X\_1, train\_X\_0))

train\_x = np.nan\_to\_num(train\_x)

train\_y = np.vstack((train\_y\_1, train\_y\_0))

train = np.append(train\_x, train\_y, 1)

# saving training data to csv file

np.savetxt("train.csv",train,delimiter=",")

files = os.listdir('./positive\_test')

for i in range(len(files)):

hog\_obj = HoG.HOG('./positive\_test/'+files[i])

hog = hog\_obj.hog()

if i == 0:

test\_X\_1 = hog

else:

test\_X\_1 = np.vstack((test\_X\_1, hog))

test\_y\_1 = np.full((test\_X\_1.shape[0],1), 1)

files = os.listdir('./negative\_test')

for i in range(len(files)):

hog\_obj = HoG.HOG('./negative\_test/'+files[i])

hog = hog\_obj.hog()

if i == 0:

test\_X\_0 = hog

else:

test\_X\_0 = np.vstack((test\_X\_0, hog))

test\_y\_0 = np.full((test\_X\_0.shape[0],1), 0)

test\_x = np.vstack((test\_X\_1, test\_X\_0))

test\_x = np.nan\_to\_num(test\_x)

test\_y = np.vstack((test\_y\_1, test\_y\_0))

test = np.append(test\_x, test\_y, 1)

# saving testing data to csv file

np.savetxt("test.csv",test,delimiter=",")

create\_data()

* 1. neural\_net.py

import numpy as np

class MLP:

def \_\_init\_\_(self, w1, b1, w2, b2, lr):

self.fc1 = FCLayer(w1, b1, lr)

self.rel = ReLU()

self.fc2 = FCLayer(w2, b2, lr)

self.sig = Sigmoid()

# function to calculate mean squared error

def MSE(self, prediction, target):

return (0.5\*(target-prediction)\*\*2).sum()

# function to calculate the error

def MSEGrad(self, prediction, target):

return -(target - prediction)

# training neural network

def train(self, X, y, steps):

stop = False

prev\_loss = 0.0

s = 0

# training will end when either epoch iterations are completed or when error is very low

while stop != True and s != steps:

i = s % y.size

xi = np.expand\_dims(X[i], axis=0)

yi = np.expand\_dims(y[i], axis=0)

pred = self.fc1.forward(xi)

pred = self.rel.forward(pred)

pred = self.fc2.forward(pred)

pred = self.sig.forward(pred)

loss = self.MSE(pred, yi)

if round(abs(loss - prev\_loss), 6) == 0.0:

print("Epochs:", s/y.size+1)

stop = True

break

prev\_loss = loss

grad = self.MSEGrad(pred, yi)

grad = self.sig.backward(grad)

grad = self.fc2.backward(grad)

grad = self.rel.backward(grad)

grad = self.fc1.backward(grad)

s += 1

# prediction using trained NN

def predict(self, X):

pred = self.fc1.forward(X)

pred = self.rel.forward(pred)

pred = self.fc2.forward(pred)

pred = self.sig.forward(pred)

# pred = np.round(pred)

return np.ravel(pred)

class FCLayer:

def \_\_init\_\_(self, w, b, lr):

self.lr = lr

self.w = w

self.b = b

# forward pass

def forward(self, input):

self.input = input

h = np.dot(input, self.w) + self.b

return h

# backward pass

def backward(self, gradients):

input = self.input

x\_ = np.dot(gradients, self.w.T)

self.w = self.w - np.dot(input.T, gradients)\*self.lr

self.b = self.b - gradients\*self.lr

return x\_

class Sigmoid:

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

None

def sigmoid\_func(self, a):

return 1/(1+np.exp(-a))

#forward pass

def forward(self, input):

self.input = input

sig\_val = self.sigmoid\_func(input)

return sig\_val

# backward pass

def backward(self, gradients):

input = self.input

sig\_val\_back = gradients\*(1 - self.sigmoid\_func(input))\*self.sigmoid\_func(input)

return sig\_val\_back

class ReLU:

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

None

# forward pass

def forward(self, input):

self.input = input

input[input<0] = 0

return input

# backward pass

def backward(self, gradients):

input = self.input

input[input < 0] = 0

input[input > 0] = 1

input \*= gradients

return input

* 1. run\_NN.py

import numpy as np

import neural\_net as model

# function to load saved data

def load\_data(path):

data = np.genfromtxt(path, delimiter=',', dtype=float)

return data[:,:-1], data[:,-1].astype(int)

train\_x, train\_y = load\_data("train.csv")

test\_x, test\_y = load\_data("test.csv")

# MLP Training

# learning rate

lr = 0.01

# random weight initialization

w1 = np.random.normal(0, .1, size=(train\_x.shape[1], 250))

w2 = np.random.normal(0, .1, size=(250,1))

b1 = np.random.normal(0, .1, size=(1,250))

b2 = np.random.normal(0, .1, size=(1,1))

mlp = model.MLP(w1, b1, w2, b2, lr)

# set epoch values

epoch = 400

steps = epoch\*train\_y.size

# training neural network

mlp.train(train\_x, train\_y, steps)

# evaluation function to calculate accuracy

def evaluate(solutions, real):

if(solutions.shape != real.shape):

raise ValueError("Output is wrong shape.")

predictions = np.array(solutions)

labels = np.array(real)

return (predictions == labels).sum() / float(labels.size)

# predicting on test data

solutions = mlp.predict(test\_x)

# printing NN output

print("NN Output:")

print(solutions)

# printing predictions

print("Predictions")

solutions = np.round(solutions)

print(solutions)

# printing evaluation accuracy

print(evaluate(solutions, test\_y))