Project 2: Human Detection

Abhedya N Khatiwala
University ID: N19478833
Net ID: ank434
Email ID: ank434@nyu.edu
Course: CS6643 Computer Vision and Scene Analysis

a) File Name of the source code:

The name of the python executable file is “HumanDetection.py”

The files that contain the HOG Descriptors for “crop001045b.bmp” and “crop001278a.bmp” are labelled “crop001045b.txt” and “crop001278a.txt” respectively.

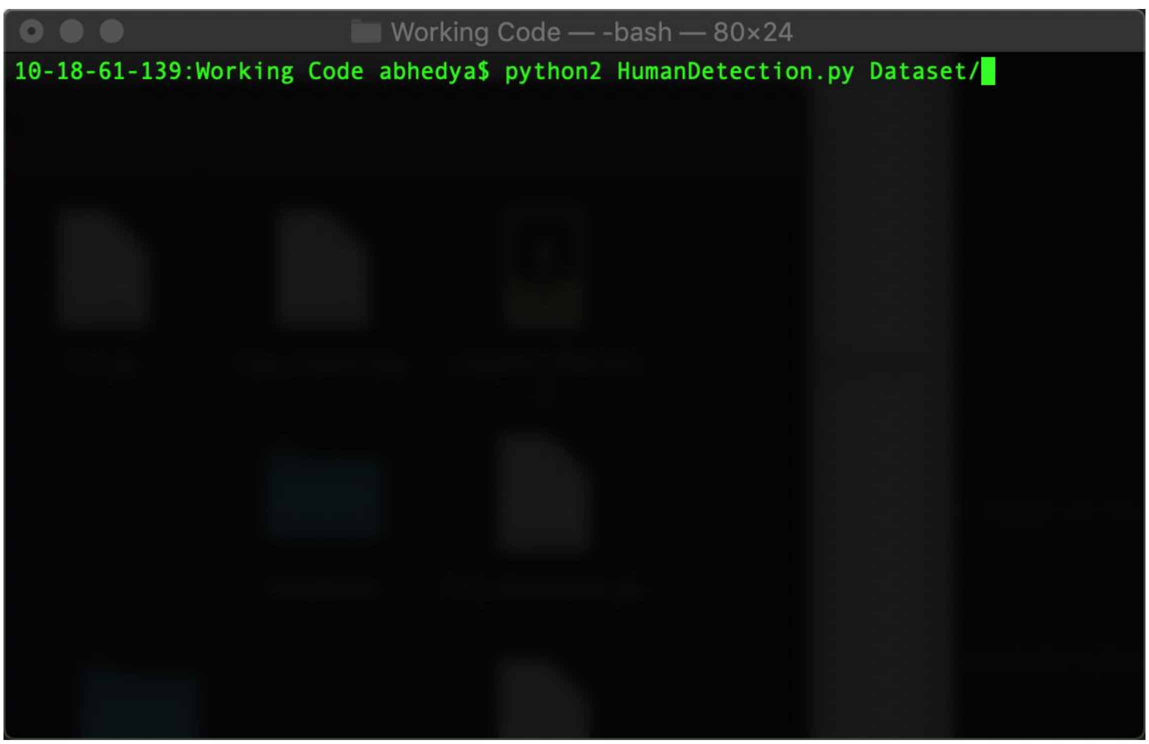
b) Instructions for running the program:

Initially, begin by pasting the folder containing the Dataset in the same folder as the python code. This is just for ease of executing the program.

Please note that, the Dataset folder must contain the Folders “Test_Neg”, “Test_Positive”, “Train_Positive”, “Train_Negative” with their correct respective images inside them.

For executing the code, type the following command on the terminal:

python2 HumanDetection.py Dataset/



Required Libraries: math, numpy, matplotlib, cv2, sys, os, glob

c)

Q1) How did you initialize the weight values of the network?

Ans1) I randomly initialized the weights using the np.random.rand() function, which generated random numbers.

Q2) How many iterations (or epochs) through the training data did you perform?

Ans2) I performed 100 epochs.

Q3) How did you decide when to stop training?

Ans3) When the weights started to converge and the error between the actual value and predicted value became very small.

Q4) 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?

Ans4) If the predicted probability for a certain image was above 50%, the neural network classified the image as containing a Human. If the probability was below 50% the neural network classified the image as not containing a Human.

d and e) Following are the results achieved. Note the parameters mentioned below every image table. The values in the Column "Output Value" are the probabilities predicted by the neural network.

Test Image	Output value	Classification
crop_000010b	0.92160128	Human
crop001008b	0.60681299	Human
crop001028a	0.58773214	Human
crop001045b	0.87280838	Human
crop001047b	0.77229701	Human
00000053a_cut	0.09512104	Not Human
00000062a_cut	0.32254887	Not Human
00000093a_cut	0.32127101	Not Human
no_person__no_bike_213_cut	0.2817215	Not Human
no_person__no_bike_247_cut	0.2429066	Not Human

- Epochs: 100, Hidden Neurons: 1000

Test Image	Output value	Classification
crop_000010b	0.90775864	Human
crop001008b	0.62555973	Human
crop001028a	0.57816784	Human
crop001045b	0.90627158	Human
crop001047b	0.79141005	Human
00000053a_cut	0.11248245	Not Human
00000062a_cut	0.30224689	Not Human
00000093a_cut	0.33281027	Not Human
no_person__no_bike_213_cut	0.24704088	Not Human
no_person__no_bike_247_cut	0.22784613	Not Human

- Epochs: 100, Hidden Neurons: 500

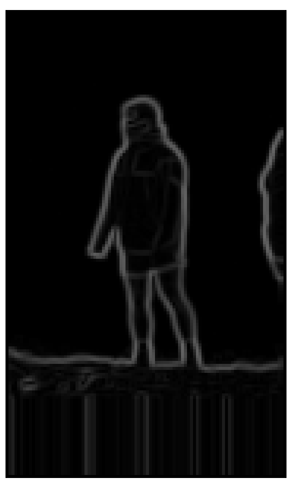
Test Image	Output value	Classification
crop_000010b	0.92358462	Human
crop001008b	0.64321735	Human
crop001028a	0.57564167	Human
crop001045b	0.90231501	Human
crop001047b	0.78677533	Human
00000053a_cut	0.08350137	Not Human
00000062a_cut	0.27950736	Not Human
00000093a_cut	0.36318688	Not Human
no_person__no_bike_213_cut	0.24066211	Not Human
no_person__no_bike_247_cut	0.23919398	Not Human

- Epochs: 100, Hidden Neurons: 250

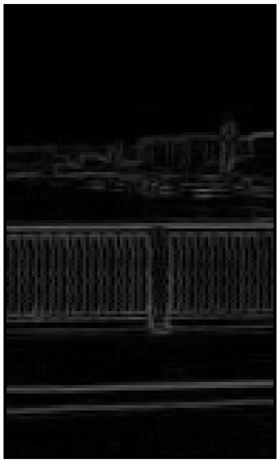
f) Normalized Gradient Magnitude Images for all the test images

Positive Test Images



Negative Test Images





g) Source Code

```
#The author of this code is "Abhedya N Khatiwala"
#CS Grad at NYU School of Engineering


import math
import numpy as np
from PIL import Image
from sys import argv
import cv2
import matplotlib.pyplot as plt
from matplotlib.pyplot import imshow, show, subplot, figure, gray, title, axis, savefig, imread
import sys
import os
import glob

class HistogramOfGradients:




	def findImages(self, dataFile, delimeter): #This function takes data folders and returns a
list of images and another list which contains actual output for that image
		PathList = []
		dataOut = []

		for dataFolder in dataFile.keys():
			for directoryName, subDirectory, fileL in os.walk(dataFolder):
				for imageFile in fileL:
					imageP = dataFolder + delimeter + imageFile
					PathList.append(imageP)

					dataOut.append([dataFile[dataFolder]])

		return PathList, dataOut

	def RGBtoGray(self, image):	#this function converts the given RGB image in to a
Grayscale Image
		return  np.round(np.dot(image[...,:3], [0.299, 0.587, 0.114]))
```

```
	def convolveP(self, image, kernel):	#this function performs the convolution 
operation.

		iN = len(image)
		jN = len(image[0])
		image_new = np.zeros((iN, jN), dtype = np.float) #initializing a matrix with 
float(0)
		for a in range(0, iN - 2):
			for b in range(0, jN - 2):
				image_new[a+1][b+1] = (np.sum(image[a: a + 3,
					b: b + 3] * kernel))/3 #multiplication of two matrices for 
convolution

		return image_new




	def prewittGradient(self, image):

		i = len(image)
		j = len(image[0])


		op1 = np.array(([-1, 0, 1],
						[-1, 0, 1],
						[-1, 0, 1])) #Prewitt's Operator

		op2 = np.array(([1, 1, 1],
						[0, 0, 0],
						[-1, -1, -1]))	#Prewitt's Operator
		Gx = self.convolveP(image, op1)
		#Gx = Gx/3
		Gy = self.convolveP(image, op2)
		#Gy/3
		gradient = np.zeros((i, j), dtype = np.float)

		grad_angle = np.zeros((i, j), dtype = np.float)

		for m in range(i):
			for n in range(j):
				gradient[m][n] = (np.sqrt(np.square(Gx[m][n]) + 
np.square(Gy[m][n])))/np.sqrt(2) # computing gradient for every pixel value and normalizing it
```

```
				#here we compute the grad angle by and check for negative
angles
				grad_angle[m][n] = 0
				if(Gx[m][n] == 0):
					if(Gy[m][n] == 0):
						grad_angle[m][n] = 0
				#	grad_angle[m][n] = 0
					else:
						if(Gy[m][n] > 0):
							grad_angle[m][n] = 90
						else:
							grad_angle[m][n] = -90

				else:
					grad_angle[m][n] =
math.degrees(np.arctan(Gy[m][n]/Gx[m][n])) #computing gradient angle for each pixel

				if(grad_angle[m][n] < 0):
					grad_angle[m][n] = grad_angle[m][n] + 180

		return gradient, grad_angle


	def histogram(self, grad_angle, gradient):  #this function computes the histogram and
returns the descriptor.
		i = len(gradient)
		j = len(gradient[0])

		featureVec = []
		cell_size = (8, 8)	#defining the cell size
		block_size = (16, 16)	#defining the block size
		step_size = (8, 8)	#defining the bin size
		grad_angle = ((grad_angle)/20.0)	#calculating the the


		i_cell_size = cell_size[0]		#cell size assigned for row
		j_cell_size = cell_size[1]				#cell size assigned for columm

		i_cell_count = i//cell_size[0]	#calcualting the cell count
		j_cell_count = j//cell_size[1]
```

i_cells_per_block, j_cells_per_block = np.array(block_size)/np.array(cell_size) #calculating the number of cells per block

i_cells_per_step, j_cells_per_step = np.array(step_size)/np.array(step_size)

i_block_count = (i_cell_count - i_cells_per_block) / i_cells_per_step + 1
j_block_count = (j_cell_count - j_cells_per_block) / j_cells_per_step + 1

bin_histograms = np.zeros((i_cell_count, j_cell_count, 9)) #we create an array to store the histogram of all cells.

#this fills the histogram array by going through the gradient angles.
for I in range(i):
for J in range(j):

c_angle = grad_angle[I][J] #calculating the edge magnitude at current pixel
c_mag = gradient[I][J] #calculating the gradient angle at current pixel

l_bin = np.floor(c_angle) #calculating the bins
h_bin = np.ceil(c_angle)

l_dist = c_angle - l_bin #calculating the angle distance to it's nearby bins
h_dist = c_angle - h_bin

l_val = c_mag * h_dist #calcuating the distance based on from the center of two bins.
h_val = c_mag * l_dist

bin_histograms[int(I // i_cell_size)][int(J/ j_cell_size)][int(l_bin)] += l_val
bin_histograms[int(I // i_cell_size)][int(J/ j_cell_size)][int(h_bin%9)] += h_val

bin_histograms = bin_histograms.astype(np.float)

#here we perform Block Level Normalization

```
		for iWc in range(0, i_block_count, i_cells_per_step):
			for jWc in range(0, j_block_count, j_cells_per_step):

				c_block = bin_histograms[iWc: iWc + i_cells_per_block, jWc: jWc +
j_cells_per_block].reshape(-1)
				c_block_l2 = sum(c_block**2)**0.5

				if c_block_l2 != 0:
					c_block /= c_block_l2
				featureVec.append(abs(c_block))

		featureVec = np.array(featureVec).reshape(-1, 1)

		return featureVec



		#This function is like a main function in this Class. It takes as input, a list which
contains images and gives back
		#an array of feature vectors.
	def hog(self, im_path):

		features = []
		for images in im_path:
			inputImage = self.GetImages(images)
			if('png' in sys.argv[1]):
				inputImage *= 255


			grayscale = self.RGBtoGray(inputImage)  #this function call returns a
Grayscale image
			#gsResult = Image.fromarray(grayscale)

			gradient, angle = self.prewittGradient(grayscale) #computing the gradient
and the angle and using Prewitt's Operator

			#gradientResult = Image.fromarray(gradient)

			cellRes = self.histogram(angle, gradient)

			features.append(cellRes)


		return features
```

```
	def GetImages(self, im_path):	#this function reads image from a file
		return np.array(plt.imread(im_path))


#This class cosists of all the functions required to initialize, train and test the Neural Network.

class NeuralNetwork:
	#initializing the neural net parameters
	# arch = (7524, 250, 1) indicates the input neurons, hidden neurons, the epochs and the 
learning_rate
	# for changing the number of hidden neurons, change the  value 250/500/1000
	def __init__(self, arch = (7524, 250, 1), epochs = 100, learning_rate = 0.01):

		self.arch = arch

		#we initialize the weights with Random Values
		self.weights1 = np.random.randn(arch[1], arch[0]) * 0.01
		self.weights2 = np.random.randn(arch[2], arch[1]) * 0.01

		#we initialize the bias with zeroes
		self.bias1 = np.zeros((arch[1], 1))
		self.bias2 = np.zeros((arch[2], 1))

		self.layer1 = self.layer2 = None

		self.d_weights1 = self.d_weights2 = None
		self.epochs = epochs
		self.learning_rate = learning_rate



	#this function performs the forward propogation.
	def feedforward(self, training_data):
		a1 = self.weights1.dot(training_data) + self.bias1
		self.layer1 = self.ReLU(a1)
		self.layer2 = self.sigmoid(self.weights2.dot(self.layer1) + self.bias2)


	#this function calculates the error on every epoch
	def error(self, actual_output):
		return 0.5*np.square(self.layer2 - actual_output).sum()
```

```
	#this function performs the back propogation after the error is calculated and rectifies
the weights
	def backprop(self, training_data, actual_output):
		diff = self.layer2 - actual_output
		z2 = diff * self.sigmoid_derivative(self.layer2)
		self.d_weights2 = np.dot(z2, self.layer1.T)

		z1 = np.dot(self.weights2.T, z2) * self.dReLU(self.layer1)
		self.d_weights1 = np.dot(z1, training_data.T)

		self.d_bias2 = np.sum(z2, axis = 1, keepdims = True)
		self.d_bias1 = np.sum(z1, axis = 1, keepdims = True)



	#this function correclty updates the weights.
	def update(self):
		self.weights1 = self.weights1 - self.learning_rate * self.d_weights1
		self.bias1 = self.bias1 - self.learning_rate * self.d_bias1

		self.weights2 = self.weights2 - self.learning_rate * self.d_weights2
		self.bias2 = self.bias2 - self.learning_rate * self.d_bias2




	#this function trains the neural net an invokes the self.function defined in this class
	def trainNN(self, trainImages, training_data_in, training_data_out):
		trainLen = len(training_data_in)
		for epoch in range(self.epochs):
			ep_error = 0.0		#we initialize the epoch_error intially as zero
			for data_count, train_data in enumerate(training_data_in):
				self.feedforward(train_data)			#for ever image in the
dataset, we call the feedforward function
				error = self.error(training_data_out[data_count])		#we
compute the error by calling the error function defined in this class
				ep_error += error				#we update the error
				self.backprop(train_data, training_data_out[data_count])	#then
we call the backprop function defined in this class to update the weights
				self.update()	#this function correctly updates the weights that
we figured out using backpropogation
```

```
				print("Epoch Count: " + str(epoch), "Average Error: ", ep_error/trainLen)



	#this function tests the neural net and returns the misclassified count.
	#this function basically predicts the probability of wether a human is present in an
image.
	#oif the probability is above 0.5, then it classifies the image as human.
	def testNN(self, testImages, testing_data_in, testing_data_out):
		misclassify = 0

		positiveList = []
		negativeList = []

		for data_count, test_data in enumerate(testing_data_in):
			self.feedforward(test_data)
			print("Predicted Probability: " + str(self.layer2), "Actual Probability Value:
" + str(testing_data_out[data_count]))

			cPrediction = np.round(self.layer2.sum())

			if cPrediction:
				positiveList.append([testImages[data_count],
str(self.layer2.sum())])
			else:
				negativeList.append([testImages[data_count],
str(self.layer2.sum())])

			misclassify += (float(cPrediction - testing_data_out[data_count]) == 0)

		#Fig = plt.figure()

		print(str(float(misclassify) / float(len(testing_data_out)) * 100) + " % Prediction
Accuracy")
		#print(len(testing_data_out))


		#sigmoid function
	def sigmoid(self, t):
		return 1/(1+np.exp(-t))


	#this function calcultes the sigmoid derivative required for backpropogation
	def sigmoid_derivative(self, p):
```

```
		return p * (1 - p)

	#this function is the RELU activation function
	def ReLU(self, t):
		return np.maximum(t, 0)

	#this function is the derivative of ReLU used for backpropogation
	def dReLU(self, t):
		return 1*(t>0)



#The following code basically extracts the images present in the Dataset folder, and calls the respective functions.
if __name__ == "__main__":
	if len(sys.argv) >= 2:
		dataPath = sys.argv[1]
	else:
		dataPath = raw_input("No Path to Data Found. Enter the path to the data directory: ")

	positiveTrainDataFolder = 'Train_Positive'
	negativeTrainDataFolder = 'Train_Negative'

	positiveTestDataFolder = 'Test_Positive'
	negativeTestDataFolder = 'Test_Neg'
	delimeter = '/'

	positiveTrainPath = dataPath + delimeter + positiveTrainDataFolder
	negativeTrainPath = dataPath + delimeter + negativeTrainDataFolder
	positiveTestPath = dataPath + delimeter + positiveTestDataFolder
	negativeTestPath = dataPath + delimeter + negativeTestDataFolder

	trainDataList = {positiveTrainPath: 1, negativeTrainPath: 0}
	testDataList = {positiveTestPath: 1, negativeTestPath: 0}

	h = HistogramOfGradients()

	trainImages, trainDataLabels = h.findImages(trainDataList, delimeter)
	testImages, testDataLabels = h.findImages(testDataList, delimeter)

	trainDataIn = np.array((h.hog(trainImages)))
	testDataIn = np.array((h.hog(testImages)))
```

```
NN = NeuralNetwork()
NN.trainNN(trainImages, trainDataIn, trainDataLabels)
NN.testNN(testImages, testDataIn, testDataLabels)
```