CS 6643 **E. K. Wong**
Project 2: Human Detection **Fall 2018**

HOG based Human Detector

(a) File names of your source code and the two output HOG (.txt) files for (3) above.

- main_function.py
- neural_net.py
- helper.py
- hog_vector_crop001278a.bmp.txt
- hog_vector_crop001045b.bmp.txt

(b) Instructions on how to compile and run your program.

- Put all the files in a directory
- Download and put the Human directory containing the training and testing images
- Run the main_function.py
- Run the neural_net.py
  Note - if there are any import errors create an empty file and name it __init__.py

(c) Answers to the four questions below.

How did you initialize the weight values of the network?

- Random initialization led to no change in mean squared error in successive epochs, so I subtracted weights from 0.5 to create some positive and some negative weights for random initialization

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

- For 250 neurons
    - 601
- For 500 neurons
    - 401
- For 1000 neurons
    - 201

How did you decide when to stop training?

- When the mean squared error didn't change much in successive epochs

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?

- Classification criteria with 250 neurons
    - If O/P > 0.8 ----- Human Detected
    - If O/P < 0.8 ----- No Human Detected

- Classification criteria with 500 neurons
    - If O/P > 0.8 ----- Human Detected
    - If O/P < 0.8 ----- No Human Detected

Rajeev Joshi
rj1234

- Classification criteria with 1000 neurons
  - If O/P > 0.8 ----- Human Detected
  - If O/P < 0.8 ----- No Human Detected

(d) A table that contains the output value of the output neuron and the classification result (human or not-human) for all 10 test images (See table below)

250 neurons

Test Image	**Output value**	**Classification**
crop_000010b	0.9269852059318393	Human Detected
crop001008b	0.9917922747315159	Human Detected
crop001028a	6.5123803047228785e-06	No Human Detected
crop001045b	0.9999995360643198	Human Detected
crop001047b	0.9999997239546716	Human Detected
00000053a_cut	0.14879380517753005	No Human Detected
00000062a_cut	0.998592566727428	Human Detected
00000093a_cut	0.3085371725503404	No Human Detected
no_person__no_bike_213_cut	0.9999715657398482	Human Detected
no_person__no_bike_247_cut	0.6982196841970575	No Human Detected

500 neurons

Test Image	**Output value**	**Classification**
crop_000010b	0.00015564604375148053	No Human Detected
crop001008b	0.9999997724336278	Human Detected
crop001028a	0.01742642473790293	No Human Detected
crop001045b	0.999980946396591	Human Detected
crop001047b	0.0028003380403008107	No Human Detected
00000053a_cut	0.9999999999999865	Human Detected
00000062a_cut	0.9999886645474166	Human Detected

00000093a_cut	1.0830284055920957e-07	No Human Detected
no_person__no_bike_213_cut	0.9115076768529807	Human Detected
no_person__no_bike_247_cut	0.009700069422528905	No Human Detected

1000 neurons

Test Image	**Output value**	**Classification**
crop_000010b	0.00845928149946696	No Human Detected
crop001008b	0.9999999999999996	Human Detected
crop001028a	1.7617381617816827e-07	No Human Detected
crop001045b	0.9999973589453423	Human Detected
crop001047b	3.285006630101005e-10	No Human Detected
00000053a_cut	0.5442353122971582	No Human Detected
00000062a_cut	0.9999999293965783	Human Detected
00000093a_cut	2.801185291452647e-07	No Human Detected
no_person__no_bike_213_cut	9.350903752500737e-13	No Human Detected
no_person__no_bike_247_cut	0.9999950777003094	Human Detected

(e) Any other comments you may have about your program, training and testing of the neural network, and your results.

- With increasing number of neurons number of iterations/epochs to converge the mean squared error went down
- In order to better normalize and randomize the input, in each iteration i shuffle the training inputs and choose only top 16 inputs

(f) Normalized gradient magnitude images for all 10 test images (copy-and-paste from image files.)

Test Positives

Normalized Gradient Magnitude crop_000010b.bmp_image

Normalized Gradient Magnitude crop001008b.bmp_image

Normalized Gradient Magnitude crop001028a.bmp_image

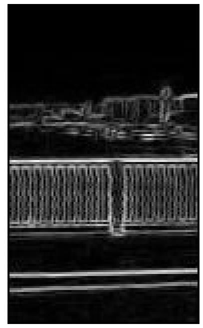
Normalized Gradient Magnitude crop001045b.bmp_image

Normalized Gradient Magnitude crop001047b.bmp_image

Test Negatives


Normalized Gradient Magnitude 00000053a_cut.bmp_image


Normalized Gradient Magnitude 00000062a_cut.bmp_image


Normalized Gradient Magnitude 00000093a_cut.bmp_image


Normalized Gradient Magnitude no_person__no_bike_213_cut.bmp_image


Normalized Gradient Magnitude no_person__no_bike_247_cut.bmp_image

(g) The source code of your program (copy-and-paste from source code file.)

1 Main_function.py

- Reads the images
- Builds the appropriate input and outputs for the neural network
  - Computes the horizontal and vertical gradients of the image
  - Computes the gradient magnitude and normalized gradient magnitude
  - Computes the gradient angles
  - Computes the HOG vector

```
from helper import *
import numpy as np
```

```
import matplotlib.pyplot as plt
from neural_net import *
import pathlib
import os

def main():
   helper = Helper()

   def preprocess(images, fileame, path, data = "", data_input = [], data_output = []):
       for c in range(len(images)):
           img = images[c]
           img = np.array(img, dtype=float)
           name = fileame[c]
           print("\n################### "+name+" ###################\n")
           """ Conver to grayscale """
           gray_img = helper.greyscale_Image(img)

           """ Computing horizontal and vertical gradients for the gray image """
           horizontal_gradient_img, vertical_gradient_img = helper.gradient(gray_img)
           # print(horizontal_gradient_img[22])
           # print(vertical_gradient_img[22])

           """ Compute the gradient magnitude """
           # print("Computing the Gradient Magnitude")
           gradient_magnitude_img = np.sqrt(np.power(horizontal_gradient_img, 2) +
np.power(vertical_gradient_img, 2))

           """ Normalizing for the range -> 0 - 255 """
           # print("Normalizing the Gradient Magnitude")
           normalized_gradient_magnitude_img =(gradient_magnitude_img /
np.max(gradient_magnitude_img)) * 255

           # print("Computing the Gradient angles")
           gradient_angle = np.arctan(vertical_gradient_img/horizontal_gradient_img )* 180 /
np.pi

           # print("Replacing the NAN values in the gradient angles with 0")
           gradient_angle[np.isnan(gradient_angle)] = 0
           for i in range(len(gradient_angle)):
               for j in range(len(gradient_angle[i])):
                   if gradient_angle[i][j] < 0:
                       gradient_angle[i][j] += 360

           # print("Computing the HOG Vector for "+name+" image")
           HOG_vector, HOG_image = helper.HOG(normalized_gradient_magnitude_img,
gradient_angle)

           HOG_vector = np.array(HOG_vector)

           # print("Flattening the HOG vector to form a large 7524 x 1 vector")
```

```
            flat_HOG_vector = HOG_vector.reshape(HOG_vector.shape[0]*HOG_vector.shape[1])

            # Save HOG vector to the output folder
            # path = helper.hog_vectors_file+"hog_vectors\\"
            # print(path)
            # pathlib.Path(path).mkdir(parents=True, exist_ok=True)
            # with open(path+name+'.txt', 'a') as the_file:
            #     for i in flat_HOG_vector:
            #         the_file.write(str(i)+"\n")

            # print("Apeending the Final HOG Vector to the Train input Array")
            data_input.append(flat_HOG_vector)
            if data == "pos":
                data_output.append([1])
            else:
                data_output.append([0])

            # helper.show([HOG_image], ["HOG Image"])
            # helper.save([gray_img, gradient_magnitude_img,
normalized_gradient_magnitude_img, HOG_image],
            #             ["Gray Image "+name, "Gradient Magnitude "+name, "Normalized
Gradient Magnitude "+name, "Hog Gradient image "+name],
            #              path)

        return data_input, data_output

    """ Read the Image """
    train_images_pos, filename_pos = helper.read_images(helper.train_pos_images)
    train_images_neg, filename_neg = helper.read_images(helper.train_neg_images)

    test_images_pos, test_filename_pos = helper.read_images(helper.test_pos_images)
    test_images_neg, test_filename_neg = helper.read_images(helper.test_neg_images)

    # """ Build the Train Input Array with dimensions -> 20 x 7524 """
    pos_train_input, pos_train_output = preprocess(train_images_pos, filename_pos,
helper.train_pos_images, data="pos", data_input=[], data_output=[])
    final_train_input, final_train_output = preprocess(train_images_neg, filename_neg,
helper.train_neg_images, data="neg", data_input=pos_train_input,
data_output=pos_train_output)
    final_train_input = np.array(final_train_input)
    final_train_output = np.array(final_train_output)

    final_train_input, final_train_output = helper.unison_shuffled_copies(final_train_input,
final_train_output)

    post_test_input, pos_test_output = preprocess(test_images_pos, test_filename_pos,
helper.test_pos_images, data="pos", data_input=[], data_output=[])
    final_test_input, final_test_output = preprocess(test_images_neg, test_filename_neg,
helper.test_neg_images, data="neg", data_input=post_test_input, data_output=pos_test_output)
    final_test_input = np.array(final_test_input)
```

```
    final_test_output = np.array(final_test_output)

    # Write the Train and Test data to file
    np.save("train_input.npy", final_train_input)
    np.save("train_output.npy", final_train_output)
    np.save("test_input.npy", final_test_input)
    np.save("test_output.npy", final_test_output)

if __name__ == "__main__":
    main()
```

2 Neural_net.py

- Runs a neural net with two layers

```
from numpy import exp, array, random, dot
import numpy as np
from helper import *

class NeuronLayer():
    def __init__(self, number_of_neurons, number_of_inputs_per_neuron):
        self.synaptic_weights = (0.5 - np.random.random((number_of_inputs_per_neuron,
number_of_neurons)) )


class NeuralNetwork():
    def __init__(self, layer1, layer2):
        self.layer1 = layer1
        self.layer2 = layer2

    def __sigmoid(self, x):
        """
        The Sigmoid function, which describes an S shaped curve.
        """
        return 1 / (1 + np.exp(-x))

    def __sigmoid_derivative(self, x):
        """
        The derivative of the Sigmoid function.
        """
        return x * (1 - x)

    def __relu(self, x):
```

```
        """
        The Rectified Linear Units Function.
        """
        return np.maximum(0.0, x)

    def __relu_derivative(self, x):
        """
        The derivative of the Rectified Linear Units Function.
        """
        x[ x<=0 ] = 0
        x[ x>0]  = 1
        return x

    # We train the neural network through a process of trial and error.
    # Adjusting the synaptic weights each time.
    def train(self, training_set_inputs, training_set_outputs, number_of_training_iterations,
learning_rate = 0.01):
        helper = Helper()

        for iteration in range(number_of_training_iterations):
            training_set_inputs, training_set_outputs =
helper.unison_shuffled_copies(training_set_inputs, training_set_outputs)
            training_set_inputs = training_set_inputs[:16]
            training_set_outputs = training_set_outputs[:16]

            # Pass the training set through our neural network
            output_from_layer_1, output_from_layer_2 = self.think(training_set_inputs)

            # Calculate the error for layer 2 (The difference between the desired output
            # and the predicted output).
            layer2_error = training_set_outputs - output_from_layer_2
            layer2_delta = layer2_error * self.__sigmoid_derivative(output_from_layer_2)

            # Calculate the error for layer 1 (By looking at the weights in layer 1,
            # we can determine by how much layer 1 contributed to the error in layer 2).
            layer1_error = layer2_delta.dot(self.layer2.synaptic_weights.T)
            layer1_delta = layer1_error * self.__relu_derivative(output_from_layer_1)

            # Calculate how much to adjust the weights by
            layer1_adjustment = learning_rate * (training_set_inputs.T.dot(layer1_delta))
            layer2_adjustment = learning_rate * (output_from_layer_1.T.dot(layer2_delta))

            # Adjust the weights.
            self.layer1.synaptic_weights += layer1_adjustment
```

```
            self.layer2.synaptic_weights += layer2_adjustment

            if iteration % 100 == 0:
                learning_rate = learning_rate / 1.1
                print(str(iteration )+ "  Error :
"+str(np.mean(self.calculate_loss(training_set_outputs, output_from_layer_2))))


    # The neural network calculates the error (Squared Error)
    def calculate_loss(self, ground_truth, predicted_output):
        return np.square(ground_truth - predicted_output)/2

    # The neural network thinks
    def think(self, inputs):
        output_from_layer1 = self.__relu(dot(inputs, self.layer1.synaptic_weights))
        output_from_layer2 = self.__sigmoid(dot(output_from_layer1,
self.layer2.synaptic_weights))
        return output_from_layer1, output_from_layer2

    # The neural network prints its weights
    def print_weights(self):
        print ("    Layer 1 Weights Shape : ")
        print (self.layer1.synaptic_weights.shape)
        print ("    Layer 2 Weights Shape :")
        print (self.layer2.synaptic_weights.shape)

    # The neural network saves its weights
    def save_weights(self):
        np.save("hidden_layer_weights.npy", self.layer1.synaptic_weights)
        np.save("output_layer_weights.npy", self.layer2.synaptic_weights)


if __name__ == "__main__":
    # load the train input and output matrices
    final_train_input = np.load("train_input.npy")
    final_train_output = np.load("train_output.npy")
    final_test_input = np.load("test_input.npy")
    final_test_output = np.load("test_output.npy")

    #Seed the random number generator (for reproducable results)
    random.seed(15)
    no_of_neurons_in_hidden_layer = 1000
    epochs = 201
```

```
# Create layer 1
layer1 = NeuronLayer(no_of_neurons_in_hidden_layer, final_test_input[0].shape[0])

# Create layer 2
layer2 = NeuronLayer(1, no_of_neurons_in_hidden_layer)

# Combine the layers to create a neural network
neural_network = NeuralNetwork(layer1, layer2)

# save weights
neural_network.save_weights()

print ("Stage 1) Random starting synaptic weights: ")
neural_network.print_weights()

# The training set. We have 7 examples, each consisting of 3 input values
# and 1 output value.
training_set_inputs = final_train_input
training_set_outputs = final_train_output

# Train the neural network using the training set.
neural_network.train(training_set_inputs, training_set_outputs, epochs)

print ("Stage 2) New synaptic weights after training: ")
neural_network.print_weights()

# Test the neural network with a new situation.
print ("Stage 3) Considering the test input: ")
hidden_state, output = neural_network.think(final_test_input)

final_output = []
for i in output:
    print(i[0])
    temp = []
    if i[0] > 0.8:
        temp.append("1 : Person Detected")
    else:
        temp.append("0 : No Person Detected")
    final_output.append(temp)
print (output, "\n",final_output,"\n" ,final_test_output)
```

3 helper.py

Rajeev Joshi
rj1234

- All of the helper functions that do actual computations

```
import matplotlib.pyplot as plt
import cv2
import math
import numpy as np
import os
import pathlib
from os import listdir

class Helper:
   def __init__(self):
       """
       Specify the Image directory(should only contain the images
       and one other directory named 'output' to save output images)
       """
       self.train_pos_images = ".\\Human\\Train_Positive\\"
       self.train_neg_images = ".\\Human\\Train_Negative\\"
       self.test_pos_images = ".\\Human\\Test_Positive\\"
       self.test_neg_images = ".\\Human\\Test_Neg\\"
       self.hog_vectors_file = ".\\Human\\"
       # self.image_dir = ".\\Human\\Train_Positive\\"

       # HOG Stuff
       self.cell_size = 8              # 8 x 8 pixels
       self.block_size = 16            # 16 x 16 pixels
       self.bin_size = 9               # Number of bins per cell
       self.angle_unit = 180 // self.bin_size  # to find the appropriate index in the cell
histogram

   def read_images(self, source_path):
       '''
       read the image in a specific directory
       :param source_path: path to the directory contains images
       :return: a list of image objects
       '''
       imgs = []
       files = []
       for file in listdir(source_path):
           if file.split("_")[0] == "outputs":
               pass
           else:
               img = cv2.imread(source_path + file)
               imgs.append(img)
               files.append(file)
       return imgs, files

   def HOG(self, gradient_magnitude, gradient_angle):
       height, width = gradient_magnitude.shape
```

```
        # create a cell gradient vector and initialize with zeros
        cell_gradient_vector = np.zeros((height // self.cell_size, width // self.cell_size,
self.bin_size))

        # calculate the cell histogram for each cell in the cell gradient vector
        for i in range(cell_gradient_vector.shape[0]):
            for j in range(cell_gradient_vector.shape[1]):
                cell_magnitude = gradient_magnitude[i * self.cell_size:(i + 1) *
self.cell_size,
                                 j * self.cell_size:(j + 1) * self.cell_size]
                cell_angle = gradient_angle[i * self.cell_size:(i + 1) * self.cell_size,
                             j * self.cell_size:(j + 1) * self.cell_size]
                cell_gradient_vector[i][j] = self.cell_gradient(cell_magnitude, cell_angle)

        hog_image = self.plot_gradient(np.zeros([height, width]), cell_gradient_vector)
        hog_vector = []
        for i in range(cell_gradient_vector.shape[0] - 1):
            for j in range(cell_gradient_vector.shape[1] - 1):
                # creating a block vector
                block_vector = []
                block_vector.extend(cell_gradient_vector[i][j])
                block_vector.extend(cell_gradient_vector[i][j + 1])
                block_vector.extend(cell_gradient_vector[i + 1][j])
                block_vector.extend(cell_gradient_vector[i + 1][j + 1])
                # normaling the block vector
                mag = lambda vector: math.sqrt(sum(i ** 2 for i in vector))
                magnitude = mag(block_vector)
                if magnitude != 0:
                    normalize = lambda block_vector, magnitude: [element / magnitude for
element in block_vector]
                    block_vector = normalize(block_vector, magnitude)
                hog_vector.append(block_vector)
        return hog_vector, hog_image

    def cell_gradient(self, cell_magnitude, cell_angle):
        orientation_centers = [0] * self.bin_size
        for i in range(cell_magnitude.shape[0]):
            for j in range(cell_magnitude.shape[1]):
                gradient_strength = cell_magnitude[i][j]
                gradient_angle = cell_angle[i][j]
                min_angle, max_angle, mod = self.get_closest_bins(gradient_angle)
                orientation_centers[min_angle] += (gradient_strength * (1 - (mod /
self.angle_unit)))
                orientation_centers[max_angle] += (gradient_strength * (mod /
self.angle_unit))
        return orientation_centers

    def get_closest_bins(self, gradient_angle):
        """
```

```
        calculates the ratio by which the magnitude will be divided in the two bins
        returns bin 1 and bin 2 along with the ratio for distribution
        """
        if gradient_angle > 170:
            gradient_angle = (gradient_angle - 180)
        idx = int(gradient_angle / self.angle_unit)
        mod = gradient_angle % self.angle_unit
        if idx == self.bin_size:
            return idx - 1, (idx) % self.bin_size, mod
        return idx, (idx + 1) % self.bin_size, mod

    def plot_gradient(self, image, cell_gradient):
        cell_width = self.cell_size / 2
        max_mag = np.array(cell_gradient).max()
        for x in range(cell_gradient.shape[0]):
            for y in range(cell_gradient.shape[1]):
                cell_grad = cell_gradient[x][y]
                cell_grad /= max_mag
                angle = 0
                angle_gap = self.angle_unit
                for magnitude in cell_grad:
                    angle_radian = math.radians(angle)
                    x1 = int(x * self.cell_size + magnitude * cell_width *
math.cos(angle_radian))
                    y1 = int(y * self.cell_size + magnitude * cell_width *
math.sin(angle_radian))
                    x2 = int(x * self.cell_size - magnitude * cell_width *
math.cos(angle_radian))
                    y2 = int(y * self.cell_size - magnitude * cell_width *
math.sin(angle_radian))
                    cv2.line(image, (y1, x1), (y2, x2), int(255 * math.sqrt(magnitude)))
                    angle += angle_gap
        return image

    def convolution(self, image, kernel):
        image_height = image.shape[0]
        image_width  = image.shape[1]

        kernel_height = kernel.shape[0]
        kernel_width = kernel.shape[1]

        height = (kernel_height - 1) // 2
        width = (kernel_width - 1) // 2

        output = np.zeros((image_height, image_width))

        for i in np.arange(height, image_height - height):
            for j in np.arange(width, image_width - width):
                sum = 0
                for k in np.arange(-height, height+1):
```

```
                for l in np.arange(-width, width+1):
                    a = image[i+k, j+l]
                    p = kernel[height+k, width+l]
                    sum += (p * a)
            output[i, j] = sum

    return output


def unison_shuffled_copies(self, a, b):
    assert len(a) == len(b)
    p = np.random.permutation(len(a))
    return a[p], b[p]



def greyscale_Image(self, rgb):
    """
    Returns a gray scale version of the input image
    """
    r, g, b = rgb[:,:,0], rgb[:,:,1], rgb[:,:,2]
    gray = 0.2989 * r + 0.5870 * g + 0.1140 * b

    return gray

def gradient(self, img):
    """
    Returns the horizonal and vertical gradient for the input image
    """
    prewitt_hor = np.array([[-1 ,0 ,1],
                            [-1 ,0 ,1],
                            [-1 ,0 ,1]])

    prewitt_ver = np.array([[-1 ,-1 ,-1],
                            [ 0 , 0 , 0],
                            [ 1 , 1 , 1]])

    hor_gradient = np.copy(img)
    # print("Running horizonal gradient operation using prewitt_hor Kernel")
    hor_gradient = self.convolution(hor_gradient, prewitt_hor)
    # print("DONE")

    ver_gradient = np.copy(img)
    # print("Running vertical gradient operation using prewitt_ver Kernel")
    ver_gradient = self.convolution(ver_gradient, prewitt_ver)
    # print("DONE")

    return hor_gradient, ver_gradient
```

```python
def save(self, images, names, image_dir):
    """
    Takes 2 lists
        images : list of images as numpy arrays
        names  : list of names as a string
    """
    for i in range(len(names)):
        # print("saving : "+names[i])
        # print(names[i])
        path = image_dir+"outputs_"+names[i].split(" ")[-1]+"\\"
        pathlib.Path(path).mkdir(parents=True, exist_ok=True)
        cv2.imwrite(os.path.join(path , str(names[i])+'_image.png'), images[i])

def show(self, images, names):
    """
    Takes 2 lists
        images : list of images as numpy arrays
        names  : list of names as a string
    """
    for i in range(len(names)):
        if str(images[i]) == 'gray_img':
            plt.imshow(images[i], cmap = plt.get_cmap('gray'))
        else:
            plt.imshow(images[i])
        print("showing : "+str(names[i]))
        plt.show()
```