Project 2 – Human Detection Report

Course:	CS6643 – Computer Vision 2018 fall
Student name:	XIAOJIE SHI
Student ID:	xs857

(1) File names of your source code and the two output HOG (.txt) files

Source code: cv_proj2_source_code_xs857.py
Hog of crop001278a.bmp : crop001278a_hog.txt
Hog of crop001045b.bmp : crop001045b_hog.txt

(2) Instructions on how to compile and run your program

Step1: unzip my project submission
Step2: open the terminal, and cd the folder
Step2: enter the following command line: python cv_proj2_source_code_xs857.py

(3) Answers to the four questions below

(a) How did you initialize the weight values of the network?

Ans: For both hidden layer and output layer, I initialize them with random weights.

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

Ans: For the result in the table, I perform 34 iterations.

(c) How did you decide when to stop training?

Ans: Using hints from the professor, I set error delta which is the percent of changing squared error from the previous squared error. When the error delta smaller than or equal 0.05 and total squared error smaller than 0.003, the training stops. Because I did several experiments of training neural network with different iterations number, for example 20, 30, 88 and plot the accuracy of test image. After comparison, I found this condition is best for accuracy and training time.

(d) 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?

Ans: I set the threshold of 0.5. When the output is greater than 0.5, I label the picture as 1 which means detecting human. When the output smaller or equal 0.5, I label the picture as 0 which means no human detected.

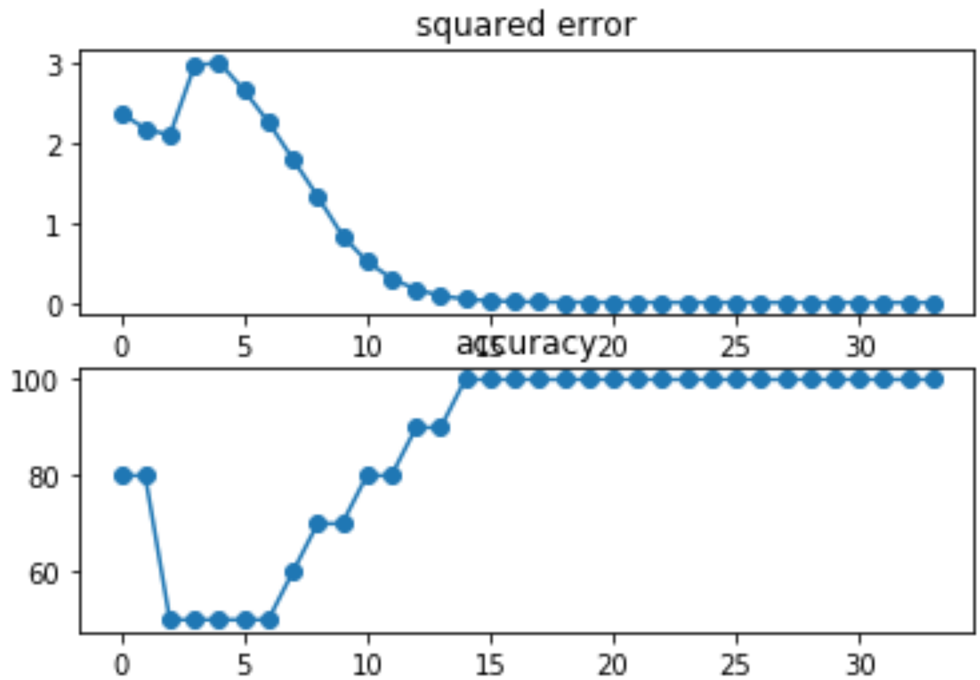
(4) A table that contains the output value of the output neuron and the classification result (human or not-human) for all 10 test images (See table below)

Test Image	**Output value**	**Classification**

crop_000010b	0.9601089303333407	1
crop001008b	0.8645913191279957	1
crop001028a	0.9883538413461832	1
crop001045b	0.6606970062400795	1
crop001047b	0.9388847336823392	1
00000053a_cut	0.09591986077922803	0
00000062a_cut	0.15227954897901244	0
00000093a_cut	0.01740267808195155	0
no_person__no_bike_213_cut	0.38566791831989644	0
no_person__no_bike_247_cut	0.1237177756315074	0

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

My neural network uses 250 hidden neurons and learning rate is 0.1. And the stop training condition is error delta smaller than or equal 0.05 and total squared error smaller than 0.003.
In most time there will be 100% accuracy on test sets. I also plot the total squared error and accuracy on the test set for each iteration as following graph. And the squared error of each iteration will be printed on the terminal too. And the final test result is align with the sequence of the file input.



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

Test positive:

(1) crop_000010b.bmp

(2) crop001008b.bmp

(3) crop001028a.bmp

(4) crop001045b.bmp

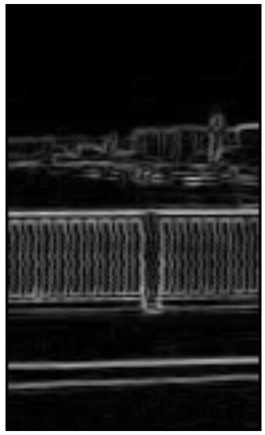
(5) crop001047b.bmp

Test negative:

(1) 00000053a_cut.bmp

(2) 00000062a_cut.bmp

(3) 00000093a_cut.bmp



(4) no_person__no_bike_213_cut.bmp

(5) no_person__no_bike_247_cut.bmp

(7) The source code of your program

```
import cv2 as cv
import numpy as np
import math
import string
import sys
from numpy import arctan2
```

```
import random
import math
import glob as gb
import matplotlib.pyplot as plt

# operator
PREWITT_GX = np.array([[-1,0,1],
                [-1,0,1],
                [-1,0,1]])

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

# each cell has 8 * 8 pixels
CELL_ROW = int(8)
CELL_COL = int(8)


WINDOW_ROW = 160
WINDOW_COL = 96

# the number of cell in each window

# row: 20
CELL_ROW_PER_WINDOW = int(WINDOW_ROW / CELL_ROW)
# col: 12
CELL_COL_PER_WINDOW = int(WINDOW_COL / CELL_COL)

# each block has 2 * 2 cells
BLOCK_ROW = 2
BLOCK_COL = 2

# the number of block in each window

# row: 19
BLOCK_ROW_PER_WINDOW = int(CELL_ROW_PER_WINDOW - BLOCK_ROW + 1)
# col: 11
BLOCK_COL_PER_WINDOW = int(CELL_COL_PER_WINDOW - BLOCK_COL + 1)

# img_size
IMG_ROW = 160
IMG_COL = 96
```

```
# for plot the graph
# total error after each iteration
error_list = []
# accuracy on test sets after each iteration
accuracy = []


# step1: convert color image into gray value

'''
description:
parameter:
return:
'''

# ********image preprocess********
'''
description: convert an color image into gray image
parameter:
   img: color image
return:
   gray_imag: gray image
'''
def color2gray(img):
   gray = 0.299*img[:,:,0] + 0.587*img[:,:,1] + 0.114*img[:,:,2]
   gray_img = gray.astype(np.uint8)
   return gray_img


'''
description: normalize the image within the range[0,255]
'''
def normalize(ndarr):
   result = np.zeros([IMG_ROW, IMG_COL])
   result = np.abs(ndarr)
   max = np.max(result)

   if max > 255:
      # take the max value greater than 255, get normalization ratio
      ratio = math.ceil(max / 255)
      result = np.rint(result / ratio)

   return result.astype(np.uint8)
```

```
# return if it is undefined area
def isCal(i, j, bound):
    if i >= bound and \
    i < IMG_ROW - bound and \
    j >= bound and \
    j < IMG_COL - bound:
        return True
    return False

'''
description: convolution
steps:
    (a) slice the matrix according to the center
    (b) np.multiply
'''
def conv(i,j,img,kernel):
    kernel_size = kernel.shape[0]
    kernel_bound = int(kernel_size / 2);

    up = i - kernel_bound
    down = i + kernel_bound + 1
    left = j - kernel_bound
    right = j + kernel_bound + 1

    sliced = img[up:down, left:right]
    return int(np.sum(sliced * kernel))

'''
description: calculate the gradient operator
steps:
    do the convolution applying Prewitt's operator
return:
    resultGX: gradient value x
    resultGY: gradient value y
'''
def gradient_operator(gimg):
    resultGX = np.zeros([IMG_ROW, IMG_COL])
    resultGY = np.zeros([IMG_ROW, IMG_COL])

    kernel_size = PREWITT_GX.shape[0]
    kernel_bound = int(kernel_size / 2);

    bound = kernel_bound
    count = 0
```

```
    # apply Prewitt to the image
    for i in range(IMG_ROW):
        for j in range(IMG_COL):
            if isCal(i, j, bound):
                resultGX[i,j] = conv(i,j,gimg, PREWITT_GX)
                resultGY[i,j] = conv(i,j,gimg, PREWITT_GY)
            else:
                resultGX[i,j] = 0
                resultGY[i,j] = 0
    return resultGX, resultGY

# calculate the magnitude
def magnitude(resultGX,resultGY):
    resultMG = np.zeros([IMG_ROW, IMG_COL])
    for i in range(IMG_ROW):
        for j in range(IMG_COL):
            resultMG[i,j] = math.sqrt(math.pow(resultGX[i,j],2)+ \
                                      math.pow(resultGY[i,j],2))
    return resultMG


# ********histogram of gradient********

# unsigned the magnitude angle
def get_orientation(Gx, Gy):
    return np.abs((arctan2(Gy, Gx) * 180 / np.pi))

# caculate the histogram for each cell
def get_histogram(magnitude_slice, orientation_slice):
    # init the histogram
    hist = np.zeros(9,dtype = np.float);

    for i in range(CELL_ROW):
        for j in range(CELL_COL):
            # get the two bins number around the orientation angle
            divide_res = orientation_slice[i,j] / 20
            left_bin_num = (math.floor(divide_res)) % 9
            right_bin_num = (math.ceil(divide_res)) % 9

            # calculate the ratio
            left_bin_ratio = (orientation_slice[i,j] - left_bin_num * 20) / 20
            right_bin_ratio = 1 - left_bin_ratio

            hist[left_bin_num] += magnitude_slice[i,j] * left_bin_ratio
```

```
        hist[right_bin_num] += magnitude_slice[i,j] * right_bin_ratio
    return hist

# get the window, and calculate the cell inside, get 20 * 12 * 9

'''
description: go over the picture, calculate the histogram for each bin
parameter:
    window_magnitude: image magnitude
    window_orientation: image orientation
return:
    window_cell is a 3d array [x,y,z]. x, y are the coordinate of the cell, z store the histogram
'''
def get_window_cell(window_magnitude, window_orientation):
    window_cell = np.zeros([CELL_ROW_PER_WINDOW,CELL_COL_PER_WINDOW,9], dtype =
np.float)

    for i in range(CELL_ROW_PER_WINDOW):
        for j in range(CELL_COL_PER_WINDOW):
            up = i * CELL_ROW
            down = up + CELL_ROW
            left = j * CELL_COL
            right = left + CELL_COL
            window_cell[i,j] = get_histogram(window_magnitude[up:down,left:right], \
                                window_orientation[up:down,left:right])
    return window_cell

# L2 normalize over block
def L2_norm(block):
    norm = np.sqrt(np.sum(np.square(block)))
    if norm == 0:
        return block
    return block / norm

# moving 2*2 block mask over the window_cell to form the block, and do the normalization
def normalize_over_block(window_cell):
    final_descriptor = np.zeros([BLOCK_ROW_PER_WINDOW,BLOCK_COL_PER_WINDOW,36],
dtype = np.float)
    for i in range(BLOCK_ROW_PER_WINDOW):
        for j in range(BLOCK_COL_PER_WINDOW):
            # row number of upperbound
            up = i
            # row number of downbound
            down = i + BLOCK_ROW
```

```
            # col number of upperbound
            left = j
            # col number of downbound
            right = j + BLOCK_COL
            block = window_cell[up:down, left:right].flatten()
            final_descriptor[i,j] = L2_norm(block)
    return final_descriptor.flatten().tolist()

'''
description: get the descriptor of the image
parameter:
    img_path: the path of the image
return:
    final_descriptor: descriptor of the image
'''
def get_descriptor(img_path):
    image = cv.imread(img_path, 1)
    gray_image = color2gray(image)
    Gx, Gy = gradient_operator(gray_image)
    mag_img = magnitude(Gx, Gy)
    orientation_img = get_orientation(Gx, Gy)
    window_cell = get_window_cell(mag_img, orientation_img)
    final_descriptor = normalize_over_block(window_cell)
    return final_descriptor

# ********training and testing data process********

def get_sets(original_path):
    train_sets = []
    count_p = 0
    for i in range(len(original_path)):
        img_path = gb.glob(original_path[i])
        for path in img_path:
            sub_train_set = []
            train_pos_des = get_descriptor(path)
            sub_train_set.append(train_pos_des)
            print("--{0:d}--".format(count_p))
            print("mean:{0:f}, std_dev{1:f}, path:{2:s}".format(np.mean(np.array(train_pos_des)),
np.std(np.array(train_pos_des)), path))
            sub_train_set.append([1 if i == 0 else 0])
            train_sets.append(sub_train_set)
            count_p += 1
    return train_sets
```

```
def get_trainning_set():
   print("********Training Sets********")
   return get_sets(["train_data/train_positive/*.bmp", \
               "train_data/train_negative/*.bmp"])

def get_test_set():
   print("********Test Sets********")
   return get_sets(["train_data/test_positive/*.bmp", \
               "train_data/test_negative/*.bmp"])

def get_magnitude_picture(original_path):
   train_sets = []
   count_p = 0
   for i in range(len(original_path)):
      img_path = gb.glob(original_path[i])
      for path in img_path:
         test_img = cv.imread(path)
         test_gray_img = color2gray(test_img)
         gx,gy = gradient_operator(test_gray_img)
         test_mag = magnitude(gx,gy)
         test_mag_norm = normalize(test_mag)
         entries = path.split('/')
         img_name,postfix = entries[2].split('.')
         img_name = img_name + "_hog"
         cv.imwrite("additional_file/{0:s}.{1:s}".format(img_name,postfix),test_mag_norm)

# neural network
# number hidden_neurons: 250
# learning rate: 0.1
# output weight: 0.01 * random
# hidden weight: 0.1 * random

"""
init:
   num_inputs:
   num_hidden:
   num_outputs:
attribute:
      LEARNING_RATE: the speed of update the weight
"""
class NeuralNetwork:
   LEARNING_RATE = 0.1
   def __init__(self, num_inputs, num_hidden, num_outputs):
      self.num_inputs = num_inputs
```

```
        self.num_hidden = num_hidden
        self.hidden_layer = NeuronLayer(num_hidden, 0, num_inputs)
        self.output_layer = NeuronLayer(num_outputs, 1, num_hidden)

    # forward inputs -> hidden layer -> output layer
    def forward_propagation(self, inputs):
        hidden_layer_outputs = self.hidden_layer.forward_propagation(inputs)
        return self.output_layer.forward_propagation(hidden_layer_outputs)

    # train neural network
    # v3
    def train(self, training_inputs, training_outputs):
        self.forward_propagation(training_inputs)
        # Step1 get get output layer partial deriavate with respect to inpit
        # there is only one output
        # p_e_o_in partial derivate of error to output neuron net input
        pd_e_o_in = [0]
        pd_e_o_in[0] =
self.output_layer.neurons[0].calculate_pd_error_wrt_total_net_input(training_outputs[0])

       #  Step1 get hidden layer partial deriavate with respect to input
        pd_e_h_in = [0] * len(self.hidden_layer.neurons)
        for h in range(len(self.hidden_layer.neurons)):
            pd_e_h_out = pd_e_o_in[0] * self.output_layer.neurons[0].weights[h]
            pd_e_h_in[h] = pd_e_h_out *
self.hidden_layer.neurons[h].calculate_pd_total_net_input_wrt_input()

        # Step2 update output neuron weights
        for w_ho in range(len(self.output_layer.neurons[0].weights)):
            delta_o_weight = pd_e_o_in[0] *
self.output_layer.neurons[0].calculate_pd_total_net_input_wrt_weight(w_ho)
            self.output_layer.neurons[0].weights[w_ho] -= self.LEARNING_RATE * delta_o_weight

        # Step2 update hidden neuron weights
        # delta_h_weight: the update value of weight
        for h in range(len(self.hidden_layer.neurons)):
            for w_ih in range(len(self.hidden_layer.neurons[h].weights)):
                delta_h_weight = pd_e_h_in[h] *
self.hidden_layer.neurons[h].calculate_pd_total_net_input_wrt_weight(w_ih)
                self.hidden_layer.neurons[h].weights[w_ih] -= self.LEARNING_RATE * delta_h_weight


    # pass every training sets input into neural network
    # calculate the total squared error of all training sets
```

```
    def calculate_total_error(self, training_sets):
        total_error = 0
        for t in range(len(training_sets)):
            training_inputs, training_outputs = training_sets[t]
            self.forward_propagation(training_inputs)
            for o in range(len(training_outputs)):
                total_error += self.output_layer.neurons[o].calculate_error(training_outputs[o])
        return total_error


    # pass every test sets input into neural network
    # calculate the accuracy
    def test(self, test_sets):
        correct = 0
        for t in range(len(test_sets)):
            training_inputs, training_outputs = test_sets[t]
            self.forward_propagation(training_inputs)
            if abs(training_outputs[0] - self.output_layer.neurons[0].output) < 0.5:
                correct += 1
        accuracy.append(correct * 10)

    # print the test result of trained model
    def test_print_final(self, test_sets):
        print("********Threshold is 0.5********")
        for t in range(len(test_sets)):
            training_inputs, training_outputs = test_sets[t]
            self.forward_propagation(training_inputs)
            prob = self.output_layer.neurons[0].output
            prediction = 1 if prob > 0.5 else 0
            print("target", training_outputs[0], \
                  "probability:", prob, \
                 "prediction:", prediction)
'''
init:
    num_neurons
    choose whether it is hidden or output
attributes:
    neurons(list):
    bias: the b in the formula y =wx + b
method:
    forward_propagation(self, inputs): get output of every neuron
'''
class NeuronLayer:
    def __init__(self, num_neurons, hidden_0_output_1, num_weights):
```

```
        self.neurons = []
        self.bias = random.randint(-1,1) * random.random()
        if hidden_0_output_1 == 0:
            for i in range(num_neurons):
                self.neurons.append(Hidden_Neuron(self.bias, num_weights))
        else:
            for i in range(num_neurons):
                self.neurons.append(Output_Neuron(self.bias, num_weights))

    def forward_propagation(self, inputs):
        outputs = []
        for neuron in self.neurons:
            outputs.append(neuron.calculate_output(inputs))
        return outputs

'''
procedure:
    inputs -> |net input| neuron(activate) -> output -> error
attribute:
    weights(list): each neuron's weights
    inputs(list): input value
    output(value): output value
method:
    calculate_output: assign the value to the output
    calculate_total_net_input(self):
    activate:
        (a)hidden layer: ReLU
        (b)output layer: sigmoid
    calculate_pd_error_wrt_total_net_input(self, target_output): (a) * (b)
    (a) calculate_pd_error_wrt_output(target_output): partial_Error / partial_Output
    (b) calculate_pd_total_net_input_wrt_input(self): partial_Output / partial_Netinput
    (c) calculate_pd_total_net_input_wrt_weight(self, index): partial_Netinput /partial_wi
'''
'''
Neuron: parent class
'''
class Neuron:
    def calculate_output(self, inputs):
        self.inputs = inputs
        self.output = self.activate(self.calculate_total_net_input())
        return self.output

    # net input follow the formula y = wx + b
    def calculate_total_net_input(self):
```

```
        total = 0
        for i in range(len(self.inputs)):
            total += self.inputs[i] * self.weights[i]
        return total + self.bias

    # partial derivate squared error with respect to the net input
    def calculate_pd_error_wrt_total_net_input(self, target_output):
        return self.calculate_pd_error_wrt_output(target_output) *
self.calculate_pd_total_net_input_wrt_input();

    # squared error
    def calculate_error(self, target_output):
        return 0.5 * ((target_output - self.output)**2)

    # partial derivate squared error with respect to the output
    def calculate_pd_error_wrt_output(self, target_output):
        return -(target_output - self.output)

    # partial derivate net input with respect to weights
    def calculate_pd_total_net_input_wrt_weight(self, index):
        return self.inputs[index]

'''
inherited class Hidden Neuron
'''
class Hidden_Neuron(Neuron):
    def __init__(self, bias, num_weights):
        self.bias = bias
        self.weights = []
        for i in range(num_weights):
            self.weights.append(0.1 * random.random()*random.randint(-1,1))
    # ReLU function
    def activate(self, total_net_input):
        return max(0, total_net_input)

    # partial derivate neuron output with respect to the net input
    # As known as the derivate of the ReLU
    def calculate_pd_total_net_input_wrt_input(self):
        if self.output > 0:
            return self.output
        else:
            return 0

'''
```

```
inherited class Output Neuron
'''
class Output_Neuron(Neuron):

    def __init__(self, bias, num_weights):
        self.bias = bias
        self.weights = []
        for i in range(num_weights):
            self.weights.append(0.01 * random.random()*random.randint(-1,1))

    # sigmoid function
    def activate(self, total_net_input):
        try:
            return 1 / (1 + math.exp(-total_net_input))
        except:
            print("ERROR", total_net_input)

    # partial derivate neuron output with respect to the net input
    # As known as the derivate of the sigmoid function
    def calculate_pd_total_net_input_wrt_input(self):
        return self.output * (1 - self.output)


# ***get report file***(do once)
# normalized gradient magnitude images for 10 test image
# get_magnitude_picture(["train_data/test_positive/*.bmp", \
#                   "train_data/test_negative/*.bmp"])

# get 2 hog
# crop001278a = np.array(get_descriptor("train_data/train_positive/crop001278a.bmp"))
# np.savetxt("additional_file/crop001278a_hog.txt",crop001278a,newline='\n',fmt = '%10.16f')

# crop001045b = np.array(get_descriptor("train_data/test_positive/crop001045b.bmp"))
# np.savetxt("additional_file/crop001045b_hog.txt",crop001045b,newline='\n',fmt = '%10.16f')


# main
# get training_sets
training_sets = get_trainning_set()
# get test_sets
test_sets = get_test_set()

# hidden neuron number
h_n_num = 250
```

```
# iteration number
iteration_num = 20

# new neural network
nn = NeuralNetwork(len(training_sets[0][0]), h_n_num, len(training_sets[0][1]))
iteration_round = 0
e_index = -1
while True:
    for j in range(20):
        training_inputs, training_outputs = training_sets[j]
        nn.train(training_inputs, training_outputs)
    error_list.append(round(nn.calculate_total_error(training_sets), 9))
    e_index += 1
    print("***",iteration_round, round(nn.calculate_total_error(training_sets), 9))
    iteration_round += 1
    nn.test(test_sets)

    if len(error_list) >=2:
        delta_e = abs((error_list[e_index] -error_list[e_index-1]) / error_list[e_index-1])
        print("error delta:",delta_e)
        if error_list[e_index] <= 0.003 and delta_e <= 0.05:
            break

nn.test_print_final(test_sets)

k = range(0,iteration_round)

plt.subplot(2, 1, 1)
plt.plot(k,error_list,'o-')
plt.title("squared error")


plt.subplot(2, 1,2)
plt.plot(k,accuracy, 'o-')
plt.title("accuracy")

plt.show()
```