CS 6643
Computer Vision

Name: Aditya Jain

University ID Number: N14198456

NetID: aj2529

Contents

Project description. In this project, you will implement a program that uses *HOG* (*Histograms of Oriented Gradients*) and *LBP* (*Local Binary Pattern*) features to detect human in images. First, you will use the HOG feature only to detect humans. Next, you will combine the HOG feature with the LBP feature to form an augmented feature (HOG-LBP) to detect human. A *Two-Layer Perceptron* (feedforward neural network) will be used to classify the input feature vector into *human* or *no-human.*

Conversion to grayscale: The inputs to your program are color sub-images cut out from larger images. First, convert the color images into grayscale using the formula $I = Round(0.299R + 0.587G + 0.114B)$ where *R, G* and *B* are the pixel values from the red, green and blue channels of the color image, respectively, and *Round* is the round off operator.

Gradient operator: Use the **Sobel's operator** for the computation of horizontal and vertical gradients. Use formula $M(i,j) = \sqrt{G_x^2 + G_y^2}$ to compute gradient magnitude, where G_x and G_y are the horizontal and vertical gradients. Normalize and round off the results to integers within the range [0, 255]. Next, compute the gradient angle (with respect to the positive *x* axis that points to the right.) For image locations where the templates go outside of the borders of the image, assign a value of 0 to both gradient magnitude and gradient angle. Also, if both G_x and G_y are 0, assign a value of 0 to both gradient magnitude and gradient angle.

HOG feature: Refer to the lecture slides for the computation of the HOG feature. Use the unsigned representation and quantize the gradient angle into one of the 9 bins as shown in the table below. If the gradient angle is within the range [170, 350), simply subtract by 180 first. Use the following parameter values in your implementation: *cell size* = 8 x 8 pixels, *block size* = 16 x 16 pixels (or 2 x 2 cells), *block overlap* or *step size* = 8 pixels (or 1 cell.) Use *L2* norm for block normalization. Leave the histogram and final feature values as floating point numbers. Do not round off to integers.

Histogram Bins		
Bin #	Angle in degrees	Bin center
1	[-10,10)	0
2	[10,30)	20
3	[30,50)	40
4	[50,70)	60
5	[70,90)	80
6	[90,110)	100
7	[110,130)	120
8	[130,150)	140
9	[150,170)	160

LBP feature: For the computation of the LBP feature, first divide the input image into non-overlapping blocks of size 16×16. Next, compute LBP patterns (refer to lecture slides) at each pixel location inside the blocks and convert the 8-bit patterns into decimals within the range [0, 255]. Then, form a histogram of the LBP patterns for each block. To reduce the dimension of the histogram, we create separate bins for uniform patterns and a single bin for all non-uniform patterns. An 8-bit LBP pattern is called uniform if the binary pattern contains at most two 0-1 or 1-0 transitions if we go around the pattern in circle. For example, 00010000 (2 transitions) is a uniform pattern, but 01010100 (6 transitions) is not. By putting all non-uniform patterns into a single bin, the dimension of the histogram is reduced from 256 to 59. The 58 uniform binary patterns correspond to the integers 0, 1, 2, 3, 4, 6, 7, 8, 12, 14, 15, 16, 24, 28, 30, 31, 32, 48, 56, 60, 62, 63, 64, 96, 112, 120, 124, 126, 127, 128, 129, 131, 135, 143, 159, 191, 192, 193, 195, 199, 207, 223, 224, 225, 227, 231, 239, 240, 241, 243, 247, 248, 249, 251, 252, 253, 254 and 255, and all other integers belong to non-uniform patterns. Let the 1st to 58th bins of your histogram be assigned to the uniform patterns according to the order above, and the 59th bin be assigned to non-uniform patterns. For pixels in the first and last rows, and first and last columns of the image, we cannot compute their LBP patterns since some of their 8-neigbors are outside of the borders of the image. Simply assign a LBP value of 5 at these pixel locations and they will be assigned to the 59th bin of the histogram for non-uniform patterns. Finally, concatenate the histograms from all blocks (in left to right, then top to bottom order) to form a single feature vector.

HOG-LBP feature: To form the combined HOG-LBP feature, simply concatenate the HOG and LBP feature vectors together to form a long vector.

Two-layer perceptron: Implement a fully-connected two-layer perceptron with an input layer of size *N*, with *N* being the size of the input feature vector, a hidden layer of size *H* and an output layer of size 1. Let $H = 200$ and 400 and report the training and classification results for each. (Optional: you can try other hidden layer sizes and report the results if you get better results than the two above.) Use the *ReLU* activation function for neurons in the hidden layer and the *Sigmoid* function for the output neuron. The Sigmoid function will ensure that the output is within the range [0,1], which can be interpreted as the probability of having detected *human* in the image. Use the weight updating rules we covered in lecture for the training of the two-layer perceptron. Use random initialization to initialize the weights of the perceptron. Assign an output label of 1.0 for training images containing human and 0.0 for training images with no human. You can experiment with and decide on the learning rate to use (can try 0.1 first.) After each epoch of training, compute the *average error* from the errors of individual training samples. The error for an individual training sample $= |\text{correct output} - \text{network output}|$, with the correct output equals 1.0 for positive samples and 0.0 for negative samples. You can stop training when the change in average error between consecutive epochs is less than some threshold (e.g., 0.1) or when the number of epochs is more than some maximum (e.g., 1000.) After training, you can use the perceptron to classify the test images. Use the following rules for classification:

Perceptron output	**Classification**
≥ 0.6	human
> 0.4 and < 0.6	borderline
≤ 0.4	no-human

Training and test images: A set of 20 training images and a set of 10 test images in *.bmp* format will be provided. The training set contains 10 positive (human) and 10 negative (no human) samples and the test set contains 5 positive and 5 negative samples. All images are of size 160 (height) X 96 (width). You can download the images from:

https://drive.google.com/drive/folders/1Lk7FLJ4fIpBZ708pOwEI-RWaGa-I35ky?usp=sharing

To access, you need to log on Google Drive with your NYU NetID.

Experiments: You need to perform experiments with hidden layer sizes of 200 and 400 in the perceptron, and for each hidden layer size, use the HOG only feature and then the combined HOG-LBP feature (a total of four experiments.) **(a) HOG only feature.** Given the image size of 160×96 and the parameters given above for HOG computation, you should have 20 X 12 cells and 19 X 11 blocks. The size of your feature vector (and the size of the input layer of your perceptron) is therefore 7,524. (b) **Combined HOG-LBP feature.** With the parameters given above for the LBP feature, there are 10×6 blocks in the input image and the size of the LBP feature is $10 \times 6 \times 59 = 3{,}540$. The combined HOG-LBP feature therefore has size $7524 + 3540 = 11{,}064$.

Implementation: You need to write program code to implement the *HOG* and *LBP* features, and the *two-layer perceptron*. You can use Python, C++/C, Java or Matlab to implement your program. If you would like to use a different language, send me an email first. You are not allowed to use any built-in library function for any of the tasks that you are required to implement, including the Sobel's operator, computation of the HOG and LBP features, and the two-layer perceptron. The only library functions you are allowed to use are those for the reading and writing of image files, matrix and vector arithmetic, and certain other commonly used mathematical functions.

1. File names for your source code and the HOG and LBP feature files for image crop001034b.
Main.py

Inside HOG Descriptor: crop001034b.txt (uploaded as name "HOG.txt" under nyu classes)
Inside LBP Descriptor: crop001034b.txt (uploaded as name "LBP.txt" under nyu classes)
Inside HOG-LBP Descriptor: crop001034b.txt (uploaded as name "HOG-LBP.txt" under nyu classes)

2. Instruction on how to run your program, and instruction on how to compile your program if your program requires compilation.

Steps:
- Install NumPy and OpenCV
- Save the Data Images directory in google drive link in the same directory
- python3 Main.py

3. Method you used to initialize the weight values of your perceptron (e.g., random initialization with values within range [0.0, 1.0].)

-> np.random() -->np = numpy

4. Criteria you used to stop training (e.g., when change in average error between consecutive epochs is less than 0.1 or when number of epochs reaches 1000.)

0.00004 is the average error between consecutive epochs

5. The number of iterations (or epochs) required to train your perceptron. Report for each of the four experiments: hidden layer sizes of 200 and 400 -- HOG only and combined HOG-LBP.

	Number of hidden neurons	**Number of epochs**
For HOG only	**200**	**117**
	400	**115**
For HOG - LBP	**200**	**117**
	400	**62**

6. For hidden layer sizes 200 and 400, create separate tables (see below) that contain the output values of the output neuron and the classification results (human, borderline or no-human) for HOG feature only and for the combined HOG-LBP feature. Use the rules above (in Two-layer perceptron section) for classification. Report results for all 10 test images in the table. Also, compute and report the average error for the 10 test images. The error for a test sample is computed as .

FOR 200

Test Image	*Correct Class*	*HOG only*		*HOG-LBP*	
		Output	*Classification*	*Output*	*Classification*
crop001034b	*Human*	*0.42*	Borderline	*0.49*	Borderline
crop001070a	*Human*	*0.79*	*Human*	*0.89*	*Human*
crop001278a	*Human*	*0.84*	*Human*	*0.79*	*Human*
crop001500b	*Human*	*0.68*	*Human*	*0.62*	*Human*
person_and_bike_151a	*Human*	*0.87*	*Human*	*0.80*	*Human*
00000003a_cut	*No-human*	*0.23*	*No-human*	*0.16*	*No-human*
00000090a_cut	*No-human*	*0.12*	*No-human*	*0.09*	*No-human*
00000118a_cut	*No-human*	*0.27*	*No-human*	*0.13*	*No-human*
no_person_no_bike_258_cut	*No-human*	*0.61*	*No-human*	*0.50*	Borderline

no_person_no_bike_264_cut	*No-human*	*0.33*	*No-human*	*0.59*	**Borderline**

FOR 400

Test Image	*Correct Class*	*HOG only*		*HOG-LBP*	
		Output	*Classification*	*Output*	*Classification*
crop001034b	*Human*	*0.44*	Borderline	*0.65*	*Human*
crop001070a	*Human*	*0.80*	*Human*	*0.97*	*Human*
crop001278a	*Human*	*0.86*	*Human*	*0.99*	*Human*
crop001500b	*Human*	*0.67*	*Human*	*0.81*	*Human*
person_and_bike_151a	*Human*	*0.85*	*Human*	*0.92*	*Human*
00000003a_cut	*No-human*	*0.24*	*No-human*	*0.15*	*No-human*
00000090a_cut	*No-human*	*0.12*	*No-human*	*0.04*	*No-human*
00000118a_cut	*No-human*	*0.30*	*No-human*	*0.05*	*No-human*
no_person_no_bike_258_cut	*No-human*	*0.60*	Borderline	*0.63*	*Human*
no_person_no_bike_264_cut	*No-human*	*0.32*	*No-human*	*0.59*	Borderline

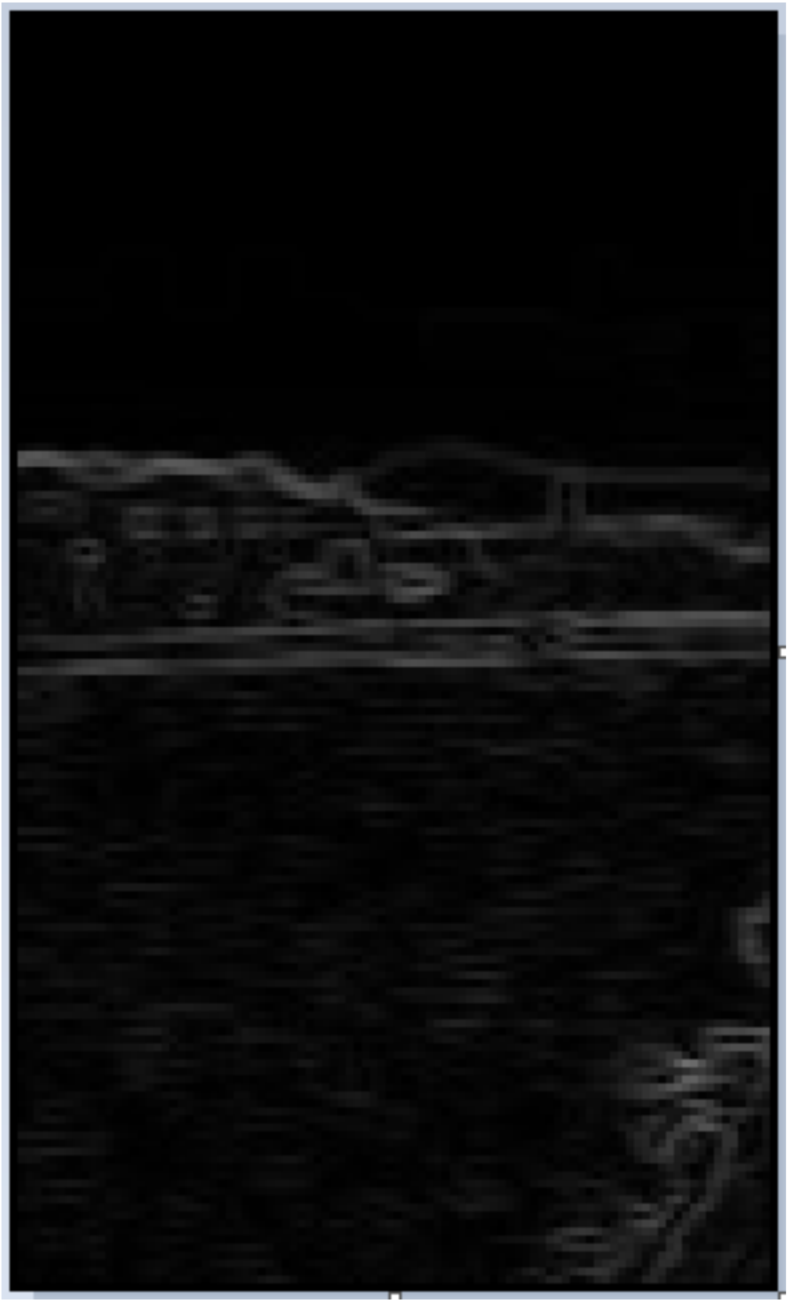
7. Any other comments you may have on your program, training and testing of the perceptron and your results.

Initially, read all the training images either positive or negative.
Made an array which consists of values
1 if the image contains human
0 if the image doesn't contain human.
This helped in adjusting prediction as well as weight bias.

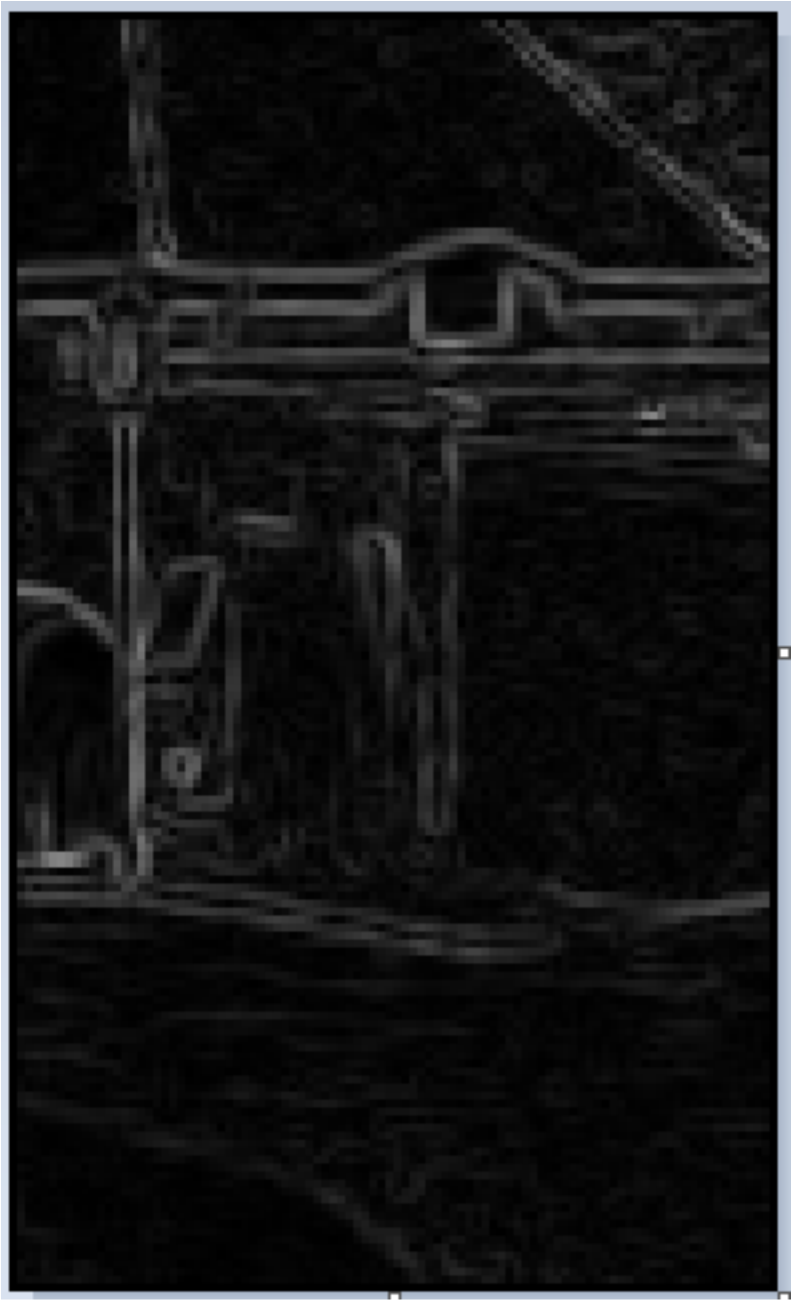
After each iteration, maintain updated weights as well as bias values. Number of neurons in hidden layer generally makes an impact on prediction. Greater the number of neurons more will be the accuracy but this comes with a cost of more amount of resource consumption.

8. Normalized gradient magnitude images for the 10 test images (Copy-and-paste from output image files.)

00000090a_cut.bmp



00000003a_cut.bmp



00000118a_cut.bmp

no_person__no_bike_258_Cut.bmp

no_person__no_bike_264_cut.bmp

Crop001278a.bmp

crop001034b.bmp

crop001500b.bmp

Person_and_bike_151a.bmp

9. The source code of your program (Copy-and-paste from source code file. This is in addition to the source code file that you need to hand in.)

Main.py

```
import os
import random
from typing import Union, List

import numpy as np
import cv2
import math
from numpy.core.multiarray import ndarray
```

```
def compute_gradient_magnitude_angle(gx, gy):
    gradient_magnitude = np.zeros((gx.shape[ 0 ], gx.shape[ 1 ]))
    gradient_angle = np.zeros((gx.shape[ 0 ], gx.shape[ 1 ]))

    for row in range(gx.shape[ 0 ]):
        for col in range(gx.shape[ 1 ]):
            gradient_magnitude[ row, col ] = math.sqrt(
                (gx[ row, col ] * gx[ row, col ]) + (gy[ row, col ] * gy[ row, col ]))
            gradient_magnitude[ row, col ] = gradient_magnitude[ row, col ] / np.sqrt(2)
            if (gx[ row, col ] == 0) and (gy[ row, col ] == 0):
                gradient_angle[ row, col ] = 0
            elif gx[ row, col ] == 0:
                if gy[ row, col ] > 0:
                    gradient_angle[ row, col ] = 90
                else:
                    gradient_angle[ row, col ] = -90
            else:
                gradient_angle[ row, col ] = math.degrees(np.arctan(gy[ row, col ] / gx[ row, col ]))

            if gradient_angle[ row, col ] < 0:
                gradient_angle[ row, col ] = 180 + gradient_angle[ row, col ]

            if gradient_angle[ row, col ] == 0:
                gradient_angle[ row, col ] = 0

    return gradient_magnitude, gradient_angle


def convolution(image: object, g: object) -> object:
    rows, cols = image.shape
    height_g, width_g = g.shape[ 0 ] // 2, g.shape[ 1 ] // 2
    image_convoluted: ndarray = np.zeros(image.shape)
    for i in range(1, rows - 1):
        for j in range(1, cols - 1):
            image_convoluted[ i, j ] = 0
            for k in range(-height_g, height_g + 1):
                for m in range(-width_g, width_g + 1):
                    image_convoluted[ i, j ] = image_convoluted[ i, j ] + (
                            g[ height_g + k, width_g + m ] * image[ i + k, j + m ])
            image_convoluted[ i, j ] = image_convoluted[ i, j ] / 3    # normalizing gradients
    return image_convoluted


def sobel(image):
    return convolution(image, np.array([ [ -1, 0, 1 ], [ -2, 0, 2 ], [ -1, 0, 1 ] ])), convolution(image, np.array(
```

```
        [ [ 1, 2, 1 ], [ 0, 0, 0 ], [ -1, -2, -1 ] ]))


def calc_cell_histogram(image: object, gradient_magnitude: object, gradient_angle: object) -> object:
    height, width = image.shape
    row: Union[ float, int ] = math.floor(height / 8)
    col: Union[ float, int ] = math.floor(width / 8)
    row_hist: int = 0
    col_hist: int = 0
    cell_histogram = np.zeros((row, col, 9))
    for r in range(0, height, 8):
        for c in range(0, width, 8):
            i_row = r
            limit_i_row = i_row + 8
            histogram = [ 0 ] * 9
            for i in range(i_row, limit_i_row):
                j_col = c
                limit_j_col = j_col + 8

                for j in range(j_col, limit_j_col):
                    if gradient_angle[ i, j ] == 0 or gradient_angle[ i, j ] == 180:
                        histogram[ 0 ] += gradient_magnitude[ i, j ]
                    elif 0 < gradient_angle[ i, j ] < 20:
                        histogram[ 0 ] += ((20 - gradient_angle[ i, j ]) / 20) * gradient_magnitude[ i, j
]
                        histogram[ 1 ] += ((gradient_angle[ i, j ] - 0) / 20) * gradient_magnitude[ i, j ]
                    elif gradient_angle[ i, j ] == 20:
                        histogram[ 1 ] += gradient_magnitude[ i, j ]
                    elif 20 < gradient_angle[ i, j ] < 40:
                        histogram[ 1 ] += ((40 - gradient_angle[ i, j ]) / 20) * gradient_magnitude[ i, j
]
                        histogram[ 2 ] += ((gradient_angle[ i, j ] - 20) / 20) * gradient_magnitude[ i, j
]
                    elif gradient_angle[ i, j ] == 40:
                        histogram[ 2 ] += gradient_magnitude[ i, j ]
                    elif 40 < gradient_angle[ i, j ] < 60:
                        histogram[ 2 ] += ((60 - gradient_angle[ i, j ]) / 20) * gradient_magnitude[ i, j
]
                        histogram[ 3 ] += ((gradient_angle[ i, j ] - 40) / 20) * gradient_magnitude[ i, j
]
                    elif gradient_angle[ i, j ] == 60:
                        histogram[ 3 ] += gradient_magnitude[ i, j ]
                    elif 60 < gradient_angle[ i, j ] < 80:
                        histogram[ 3 ] += ((80 - gradient_angle[ i, j ]) / 20) * gradient_magnitude[ i, j
]
                        histogram[ 4 ] += ((gradient_angle[ i, j ] - 60) / 20) * gradient_magnitude[ i, j
]
```

```
                elif gradient_angle[ i, j ] == 80:
                    histogram[ 4 ] += gradient_magnitude[ i, j ]
                elif 80 < gradient_angle[ i, j ] < 100:
                    histogram[ 4 ] += ((100 - gradient_angle[ i, j ]) / 20) * gradient_magnitude[ i, j
]
                    histogram[ 5 ] += ((gradient_angle[ i, j ] - 80) / 20) * gradient_magnitude[ i, j
]
                elif gradient_angle[ i, j ] == 100:
                    histogram[ 5 ] += gradient_magnitude[ i, j ]
                elif 100 < gradient_angle[ i, j ] < 120:
                    histogram[ 5 ] += ((120 - gradient_angle[ i, j ]) / 20) * gradient_magnitude[ i, j
]
                    histogram[ 6 ] += ((gradient_angle[ i, j ] - 100) / 20) * gradient_magnitude[ i, j
]
                elif gradient_angle[ i, j ] == 120:
                    histogram[ 6 ] += gradient_magnitude[ i, j ]
                elif 120 < gradient_angle[ i, j ] < 140:
                    histogram[ 6 ] += ((140 - gradient_angle[ i, j ]) / 20) * gradient_magnitude[ i, j
]
                    histogram[ 7 ] += ((gradient_angle[ i, j ] - 120) / 20) * gradient_magnitude[ i, j
]
                elif gradient_angle[ i, j ] == 140:
                    histogram[ 7 ] += gradient_magnitude[ i, j ]
                elif 140 < gradient_angle[ i, j ] < 160:
                    histogram[ 7 ] += ((160 - gradient_angle[ i, j ]) / 20) * gradient_magnitude[ i, j
]
                    histogram[ 8 ] += ((gradient_angle[ i, j ] - 140) / 20) * gradient_magnitude[ i, j
]
                elif gradient_angle[ i, j ] == 160:
                    histogram[ 8 ] += gradient_magnitude[ i, j ]
                elif gradient_angle[ i, j ] > 160:
                    histogram[ 8 ] += ((180 - gradient_angle[ i, j ]) / 20) * gradient_magnitude[ i, j
]
                    histogram[ 0 ] += ((gradient_angle[ i, j ] - 160) / 20) * gradient_magnitude[ i, j
]

            cell_histogram[ row_hist, col_hist ] = histogram
            col_hist = col_hist + 1
        row_hist = row_hist + 1
        col_hist = 0
    return cell_histogram, row, col


# calculate feature vector which contains hog descriptor of the image.
def calc_feature_vector(cell_histogram: object, image_height: object, image_width: object) -> object:
    feature_vector = np.zeros(1)
    for row in range(0, image_height - 1):
```

```
        for col in range(0, image_width - 1):
            s: float = 0.0
            # create a temporary block of size 36
            block: ndarray = np.zeros(1)
            block = np.append(block, cell_histogram[ row, col ])
            block = np.append(block, cell_histogram[ row, col + 1 ])
            block = np.append(block, cell_histogram[ row + 1, col ])
            block = np.append(block, cell_histogram[ row + 1, col + 1 ])
            block = block[ 1: ]
            # l2-normalization
            for k in range(0, 36):
                s = s + np.square(block[ k ])
            l2_norm_factor = np.sqrt(s)
            for k in range(0, 36):
                if l2_norm_factor == 0:
                    continue
                block[ k ] = block[ k ] / l2_norm_factor    # l2 normalization.
            feature_vector = np.append(feature_vector, block)
    return feature_vector[ 1: ]


def calc_hog(image: object, gradient_magnitude: object, gradient_angle: object) -> object:
    cell_histogram, image_height, image_width = calc_cell_histogram(image, gradient_magnitude,
gradient_angle)
    return calc_feature_vector(cell_histogram, image_height, image_width)


def lbp_value(image: object, x: object, y: object) -> object:
    lbp: List[ int ] = [ get_pixel(image, image[ x ][ y ], x + 1, y + 1), get_pixel(image, image[ x ][ y ], x + 1,
y),
                        get_pixel(image, image[ x ][ y ], x + 1, y - 1), get_pixel(image, image[ x ][ y
], x, y + 1),
                        get_pixel(image, image[ x ][ y ], x, y - 1), get_pixel(image, image[ x ][ y ], x
- 1, y + 1),
                        get_pixel(image, image[ x ][ y ], x - 1, y), get_pixel(image, image[ x ][ y ], x
- 1, y - 1) ]

    power_val: List[ int ] = [ 1, 2, 4, 8, 16, 32, 64, 128 ]
    val: int = 0
    for i in range(len(lbp)):
        val += lbp[ i ] * power_val[ i ]
    return val


def calc_lbp(image: object):
    height, width = image.shape
    blocks = [ ]
```

```
    for j in range(0, width, 16):
        for i in range(0, height, 16):
            blocks.append(image[ i:i + 16, j:j + 16 ])
    blocks = np.array(blocks)
    lbp = np.zeros((10, 6, 59), np.uint8)
    for block in blocks:
        hist = {0: 0, 1: 0, 2: 0, 3: 0, 4: 0, 6: 0, 7: 0, 8: 0, 128: 0, 12: 0, 14: 0, 15: 0, 16: 0, 131: 0, 24: 0,
                28: 0, 30: 0, 31: 0, 32: 0, 240: 0, 129: 0, 193: 0, 135: 0, 255: 0, 48: 0, 56: 0, 159: 0, 60:
0, 192: 0,
                62: 0, 191: 0,
                64: 0, 224: 0, 195: 0, 199: 0, 207: 0, 248: 0, 251: 0, 143: 0, 223: 0, 96: 0, 225: 0, 227: 0,
256: 0,
                231: 0, 252: 0,
                239: 0, 112: 0, 241: 0, 243: 0, 254: 0, 247: 0, 120: 0, 249: 0, 63: 0, 124: 0, 253: 0, 126: 0,
127: 0}
        for i in range(16):
            for j in range(16):
                if i == 0 or i == 15 or j == 0 or j == 15:
                    val = 5
                else:
                    val = lbp_value(block, i, j)
                if val in hist:
                    hist[ val ] += 1
                else:
                    hist[ 256 ] += 1
        temp = [ ]
        for k in sorted(hist.keys()):
            temp.append(hist[ k ])
        lbp = np.append(lbp, temp)
        return lbp[ 59: ]


def get_pixel(img: object, center: object, x: object, y: object) -> object:
    new_value = 0
    try:
        if img[ x ][ y ] >= center:
            new_value = 1
    except:
        pass
    return new_value


# sigmoid function
def sigmoid(x: object) -> object:
    return 1.0 / (1.0 + np.exp(-x))
```

```
# derivative of sigmoid function
def d_sigmoid(x: object) -> object:
    return x * (1 - x)


# relu function
def relu(x):
    return x * (x > 0)


# Derivative of relu function
def derivative_relu(x):
    return 1. * (x > 0)


def train_neural_network(x: object, actual_training_label_list: object, number_of_hidden_layer_neurons:
object) -> object:
    np.random.seed(1)
    #    random initialization of weight and bias
    w1 = np.random.randn(number_of_hidden_layer_neurons, len(x[ 0 ])) * 0.01
    b1 = np.zeros((number_of_hidden_layer_neurons, 1))
    w2 = np.random.randn(1, number_of_hidden_layer_neurons) * 0.01
    b2 = np.zeros((1, 1))

    weight_bias_dict = {}    # This will contain updated weight and bias.
    old_cost = 0.0

    # This neural network trains maximum up to 200 epoch.
    # If cost between two epochs < 0.02, stop
    # weights do not change much
    for i in range(0, 1000):
        cost = 0.0
        # print(len(X))
        for j in range(0, len(x)):
            q = x[ j ]    # getting feature vector from the list.
            # Neural network train
            # forward pass
            z1 = w1.dot(q) + b1
            a1 = relu(z1)
            z2 = w2.dot(a1) + b2
            a2 = sigmoid(z2)
            cost += (1.0 / 2.0) * (np.square((a2 - actual_training_label_list[ j ])))    # findng the cost of
the every image and sum their cost.

            # Backward Propagation
            dz2 = (a2 - actual_training_label_list[ j ]) * d_sigmoid(a2)
```

```
            dw2 = np.dot(dz2, a1.T)
            db2 = np.sum(dz2, axis=1, keepdims=True)
            dz1 = w2.T.dot(dz2) * derivative_relu(a1)
            dw1 = np.dot(dz1, q.T)
            db1 = np.sum(dz1, axis=1, keepdims=True)

            # updating weights. Here 0.01 is the learning rate
            w1 = w1 - 0.01 * dw1
            w2 = w2 - 0.01 * dw2
            b1 = b1 - 0.01 * db1
            b2 = b2 - 0.01 * db2

        cost_avg = cost / len(x)    # taking average cost
        print("Epoch = ", i + 1, "cost_avg = ", cost_avg[ 0 ][ 0 ])
        weight_bias_dict = {'w1': w1, 'b1': b1, 'w2': w2, 'b2': b2}    # save updated weights.
        # if cost between two epochs < 0.0001, stop.
        # Because we know that weights do not change too much.
        if abs(old_cost - cost_avg) <= 0.00004:
            return weight_bias_dict
        else:
            old_cost = cost_avg
    return weight_bias_dict


def accuracy(nn_output, y_test):
    count = 0
    for no, ao in zip(nn_output, y_test):
        # if neural network's output is > 0.5,
        # it means neural network has detected that
        # there is a human in the image other wise there is not human in the image
        if no[ 0 ] > 0.5:
            count += abs(1.0 - ao[ 0 ])
        else:
            count += abs(0.0 - ao[ 0 ])

    return (((len(y_test) - count) / len(y_test)) * 100)[ 0 ]


def save_model_file(dictionary, file_name):
    np.save(str(file_name) + ".npy", dictionary)
    print("Saved model file as", str(file_name), ".npy")


def loadModelFile(name):
    print("Loading model file")
    print(name)
    dictionary = np.load(str(name) + ".npy", allow_pickle=True)
```

```
    print("Successfully loaded model files")
    return dictionary[ () ]


# Predict the newly seen data
def predict(x_test, trained_model_parameter_dict):

    w1, w2, b1, b2 = trained_model_parameter_dict[ 'w1' ], trained_model_parameter_dict[ 'w2' ],
trained_model_parameter_dict[ 'b1' ], trained_model_parameter_dict[
        'b2' ]
    z1 = w1.dot(x_test) + b1
    a1 = relu(z1)
    z2 = w2.dot(a1) + b2
    a2 = sigmoid(z2)
    return a2


def calculateFeatureVectorImg_HOG(img_path):
    """
    @param 1: img_path, full path of the image
    @return feature_vector, contains features which is used as an input to neural network. dimension [7524
x 1]
    """
    img_c = cv2.imread(img_path)
    img_gray_scale = np.round(0.299 * img_c[ :, :, 2 ] + 0.587 * img_c[ :, :, 1 ] + 0.114 * img_c[ :, :,0 ])
    gx, gy = sobel(img_gray_scale)
    gradient_magnitude, gradient_angle = compute_gradient_magnitude_angle(gx, gy)

    img_path = img_path.split('/')

    # save gradient magnitude files for test images.
    if "Test_" in img_path[ 1 ]:
        if not os.path.exists("Gradient Magnitude Test Images"):
            os.makedirs("Gradient Magnitude Test Images")
        cv2.imwrite("Gradient Magnitude Test Images" + "/" + str(img_path[ 2 ]), gradient_magnitude)

    feature_vector = calc_hog(img_gray_scale, gradient_magnitude,
                              gradient_angle)    # calculate hog descriptior

    feature_vector2 = calc_lbp(img_gray_scale)
    feature_vector = feature_vector.reshape(feature_vector.shape[ 0 ],
                                            1)    # reshaping vector. making dimension [7524 x 1]
    # this below code is used to store the feature vector of crop001278a.bmp and crop001278a.bmp into txt
file.
    # feature_vector2 = feature_vector2.reshape(feature_vector2.shape[0], 1)
    # # print(feature_vector.shape,feature_vector2.shape)
    # feature_vector1 = np.append(feature_vector,feature_vector2)
```

```
        # feature_vector1 = feature_vector1.reshape(feature_vector1.shape[0], 1)
        if img_path[ 2 ] == "crop001034b.bmp":
                if not os.path.exists("HOG descriptor"):
                        os.makedirs("HOG descriptor")

                # saving hog descriptor value. Here,%10.14f will store upto 14 decimal of value
                np.savetxt("HOG descriptor" + "/" + str(img_path[ 2 ][ :-3 ]) + "txt", feature_vector,
fmt="%10.14f")
        # np.savetxt("HOG-LBP descriptor" + "/" + str(img_path[2][:-3]) + "txt", feature_vector1,
fmt="%10.14f")
        # np.savetxt("LBP descriptor" + "/" + str(img_path[2][:-3]) + "txt", feature_vector2, fmt="%10.14f")
        return feature_vector


def calculateFeatureVectorImg_LBP(img_path):
        img_c = cv2.imread(img_path)
        img_gray_scale = np.round(
                0.299 * img_c[ :, :, 2 ] + 0.587 * img_c[ :, :, 1 ] + 0.114 * img_c[ :, :,
                                                                                  0 ])    # converting
image into grayscale.
        gx, gy = sobel(img_gray_scale)    # finding horizontal gradient and vertical gradient.
        gradient_magnitude, gradient_angle = compute_gradient_magnitude_angle(gx,
                                                                                      gy)    # finding
gradient magnitude and gradient angle.

        img_path = img_path.split('/')

        # save gradient magnitude files for test images.
        if ("Test_" in img_path[ 1 ]):
                if not os.path.exists("Gradient Magnitude Test Images"):
                        os.makedirs("Gradient Magnitude Test Images")
                cv2.imwrite("Gradient Magnitude Test Images" + "/" + str(img_path[ 2 ]), gradient_magnitude)

        feature_vector = calc_hog(img_gray_scale, gradient_magnitude,
                                  gradient_angle)    # calculate hog descriptior

        feature_vector2 = calc_lbp(img_gray_scale)
        feature_vector = feature_vector.reshape(feature_vector.shape[ 0 ],
                                                1)    # reshaping vector. making dimension [7524 x 1]
        # this below code is used to store the feature vector of crop001278a.bmp and crop001278a.bmp into txt
file.
        feature_vector2 = feature_vector2.reshape(feature_vector2.shape[ 0 ], 1)
        # print(feature_vector.shape,feature_vector2.shape)
        feature_vector1 = np.append(feature_vector, feature_vector2)
        feature_vector1 = feature_vector1.reshape(feature_vector1.shape[ 0 ], 1)
        if img_path[ 2 ] == "crop001034b.bmp":
                if not os.path.exists("HOG descriptor"):
```

```
            os.makedirs("HOG descriptor")
        if not os.path.exists("HOG-LBP descriptor"):
            os.makedirs("HOG-LBP descriptor")
        if not os.path.exists("LBP descriptor"):
            os.makedirs("LBP descriptor")
        # saving hog descriptor value. Here,%10.14f will store upto 14 decimal of value
        # np.savetxt("HOG descriptor" + "/" + str(img_path[2][:-3]) + "txt", feature_vector,
fmt="%10.14f")
        np.savetxt("HOG-LBP descriptor" + "/" + str(img_path[ 2 ][ :-3 ]) + "txt", feature_vector1,
fmt="%10.14f")
        np.savetxt("LBP descriptor" + "/" + str(img_path[ 2 ][ :-3 ]) + "txt", feature_vector2,
fmt="%10.14f")
    return feature_vector1


# Preprocessing. Getting the folders where the images are stored.
TRAIN_PATH = [ "Image Data/Training images (Neg)", "Image Data/Training images (Pos)" ]
TEST_PATH = [ "Image Data/Test images (Pos)", "Image Data/Test images (Neg)" ]
y_train = [ ]    # contains training samples label.
y_test = [ ]    # contains testing samples label.

train_images_feature_vector_list = [ ]    # contrains training samples feature vector.
test_images_feature_vector_list = [ ]    # contrains testing samples feature vector.
print("FOR HOG")
print("#########Start finding feature vector for training samples#############")
ind = 0
for path in TRAIN_PATH:
    for root, dirs, files in os.walk(path):
        for name in files:
            # calculating hog descriptor of the all train images and store it into
train_images_feature_vector_list.
            train_images_feature_vector_list.append(calculateFeatureVectorImg_HOG(path + "/" +
str(name)))
            y_train.append(np.array([ [ ind ] ]))    # if human is present in the image we label as 1
otherwise 0.
    ind = 1
print("#########Finished finding feature vector for training samples###########")

test_img_path = [ ]    # storing path of the test images

print("#########Start finding feature vector for testing samples#############")
ind = 1
for path in TEST_PATH:
    for root, dirs, files in os.walk(path):
        for name in files:
            # storing path of the test images.
            test_img_path.append(path + '/' + str(name))
```

```
			# calculating hog descriptor of the all train images and store it into 
train_images_feature_vector_list.
			test_images_feature_vector_list.append(calculateFeatureVectorImg_HOG(path + "/" + 
str(name)))
			y_test.append(np.array([ [ ind ] ]))    # if human is present in the image we label as 1 
otherwise 0.
	ind = 0
print("#########Finished finding feature vector for testing samples###########")

# Shuffle the data
combine = list(zip(train_images_feature_vector_list, y_train))
random.shuffle(combine)
train_images_feature_vector_list, y_train = zip(*combine)
# Testing the trained neural network
for no_hidden_neurons in [ 200, 400 ]:
	print("###################Start training where ", no_hidden_neurons, " hidden 
neurons################")
	print(len(train_images_feature_vector_list))
	model = train_neural_network(train_images_feature_vector_list, y_train, no_hidden_neurons)
	print("Saving model in data", str(no_hidden_neurons), ".npy file")
	save_model_file(model, "data" + str(no_hidden_neurons))    # save model file. we can use it later for 
prediction.
	print("successfully trained neural network containing ", no_hidden_neurons, " hidden neurons.")

print("################################################################################################")

""" Let's test trained neural network."""
for no_hidden_neurons in [ 200, 400 ]:
	neural_network_output = [ ]    # storing predicted value of the test image
	model = loadModelFile(
		"data" + str(no_hidden_neurons))    # load model file for getting weights and bias.

	print("Predicted value of the test images where number of neurons = ", no_hidden_neurons)

	# getting all images from the list of test images and print output value of the neural network.
	for test_img, test_img_name in zip(test_images_feature_vector_list, test_img_path):
		neural_network_output.append(predict(test_img, model))
		print(test_img_name, "   Predicted value = ", neural_network_output[ -1 ][ 0 ][ 0 ])


print("###############################################################################################")
	print(
		"Accuracy = ", accuracy(neural_network_output, y_test))
	print("Finished prediction of the neural network where number of neurons in hidden layers = ", 
no_hidden_neurons)

print("FOR HOG_LBP")
```

```
print("#########Start finding feature vector for training samples############")
ind = 0
TRAIN_PATH = [ "Image Data/Training images (Neg)", "Image Data/Training images (Pos)" ]
TEST_PATH = [ "Image Data/Test images (Pos)", "Image Data/Test images (Neg)" ]
y_train = [ ]    # contains training samples label.
y_test = [ ]    # contains testing samples label.

train_images_feature_vector_list = [ ]
test_images_feature_vector_list = [ ]
for path in TRAIN_PATH:
        for root, dirs, files in os.walk(path):
                for name in files:
                        # calculating hog descriptor of the all train images and store it into
train_images_feature_vector_list.
                        train_images_feature_vector_list.append(calculateFeatureVectorImg_LBP(path + "/" +
str(name)))
                        y_train.append(np.array([ [ ind ] ]))    # if human is present in the image we label as 1
otherwise 0.
        ind = 1
print("#########Finished finding feature vector for training samples##########")

test_img_path = [ ]    # storing path of the test images

print("#########Start finding feature vector for testing samples############")
ind = 1
for path in TEST_PATH:
        for root, dirs, files in os.walk(path):
                for name in files:
                        # storing path of the test images.
                        test_img_path.append(path + '/' + str(name))
                        # calculating hog descriptor of the all train images and store it into
train_images_feature_vector_list.
                        test_images_feature_vector_list.append(calculateFeatureVectorImg_LBP(path + "/" +
str(name)))
                        y_test.append(np.array([ [ ind ] ]))    # if human is present in the image we label as 1
otherwise 0.
        ind = 0
print("#########Finished finding feature vector for testing samples##########")

# Shuffle the data. It's a good thing to shuffle data.
combine = list(zip(train_images_feature_vector_list, y_train))
random.shuffle(combine)
train_images_feature_vector_list, y_train = zip(*combine)

"""Let's train neural network."""
for no_hidden_neurons in [ 200, 400 ]:
```

```
    print("#####################Start training where ", no_hidden_neurons, " hidden
neurons##################")
    print(len(train_images_feature_vector_list))
    model = train_neural_network(train_images_feature_vector_list, y_train, no_hidden_neurons)
    print("Saving model in data", str(no_hidden_neurons), ".npy file")
    save_model_file(model, "data" + str(no_hidden_neurons))    # save model file. we can use it later for
prediction.
    print("successfully trained neural network containing ", no_hidden_neurons, " hidden neurons.")

print("##########################################################################################")

# Testing the trained network
for no_hidden_neurons in [ 200, 400 ]:
    neural_network_output = [ ]    # storing predicted value of the test image
    model = loadModelFile(
        "data" + str(no_hidden_neurons))    # load model file for getting weights and bias.

    print("Predicted value of the test images where number of neurons = ", no_hidden_neurons)

    # getting all images from the list of test images and print output value of the neural network.
    for test_img, test_img_name in zip(test_images_feature_vector_list, test_img_path):
        neural_network_output.append(predict(test_img, model))
        print(test_img_name, "    Predicted value = ", neural_network_output[ -1 ][ 0 ][ 0 ])


print("#########################################################################################")
    print(
        "Accuracy = ", accuracy(neural_network_output, y_test))    # print accuracy of neural network.
    print("Finished prediction of the neural network where number of neurons in hidden layers = ",
no_hidden_neurons)
```