**Human Detection using HOG Feature**

**Eashan Kaushik – ek3575, N19320245**

**Srijan Malhotra – sm9439, N18390405**

**Instruction to Run Code**

**Step-1:** Download files and put it in one directory. Files to be downloaded are:

* convolution.py
* main.py
* HOG.py

**Step-2:** Download the training the testing images and make the following structure:

|-train

|-------positive

|-------negative

|-test

|-------positive

|-------negative

|-Output

|-Magnitude

Save the positive and negative train and test images in the folder given above. Output and Magnitude folders are left empty.

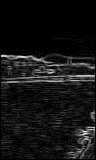
**Step-3:** Go to your terminal and go in the directory, run the command python main.py.

**Normalized Gradient Magnitude of 10 Test Images**

00000003a\_cut.bmp

****

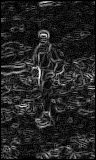
00000090a\_cut.bmp

****

00000118a\_cut.bmp

****

crop001034b.bmp

****

crop001070a.bmp

****

crop001278a.bmp

****

crop001500b.bmp

****

no\_person\_\_no\_bike\_258\_Cut.bmp

****

no\_person\_\_no\_bike\_264\_cut.bmp

****

person\_and\_bike\_151a.bmp

****

**Source Code**

* **model-test.py**

from convolution import SeConvolve

import numpy as np

import cv2

import os

import matplotlib.pyplot as plt

from PIL import Image

import math

import matplotlib.image as mpimg

import sys

class HOGDescriptor:

    def \_\_init\_\_(self):

        # Gradient x and y operations Prewitt's Operators

        self.\_convolution\_matrix\_gx = np.array([[-1, 0, 1], [-1, 0, 1], [-1, 0, 1]])

        self.\_convolution\_matrix\_gy = np.array([[1, 1, 1], [0, 0, 0], [-1, -1, -1]])

        #### Train ####

        self.training\_discript = dict()

        self.train\_images = list()

        self.number\_of\_train\_positive = 0

        self.image\_names\_train = list()

        self.gray\_train\_images = list()

        # Output of step 2

        self.\_gradient\_x\_train = list()

        self.\_gradient\_y\_train = list()

        # Output of step 3

        self.\_magnitude\_train = list()

        # Angle Output

        self.\_angle\_train = list()

        #### Test ####

        self.test\_discript = dict()

        self.test\_images = list()

        self.number\_of\_test\_positive = 0

        self.image\_names\_test = list()

        self.gray\_test\_images = list()

        # Output of step 2

        self.\_gradient\_x\_test = list()

        self.\_gradient\_y\_test = list()

        # Output of step 3

        self.\_magnitude\_test = list()

        # Angle Output

        self.\_angle\_test = list()

    # Function to fit the training data

    def fit(self):

        # read train positive images

        my\_list = os.listdir('train/positive')

        self.image\_names\_train.extend(my\_list)

        for current in my\_list:

            image = cv2.imread("train/positive/" + current)

            image = cv2.cvtColor(image, cv2.COLOR\_BGR2RGB)

            self.train\_images.append(image)

        self.number\_of\_train\_positive = len(self.train\_images)

        # read train negative images

        my\_list = os.listdir('train/negative')

        self.image\_names\_train.extend(my\_list)

        for current in my\_list:

            image = cv2.imread("train/negative/" + current)

            image = cv2.cvtColor(image, cv2.COLOR\_BGR2RGB)

            self.train\_images.append(image)

        # convert into numpy array

        self.train\_images = np.array(self.train\_images)

        # convert into grascale image

        self.to\_grayscale(self.train\_images, train=True)

        # gradient operation to calculate magnitude and angle

        self.gradient\_operation(self.gray\_train\_images, train=True)

        # calculate the hog descriptor

        self.hog\_feature(self.\_magnitude\_train, self.\_angle\_train, train=True)

    def test(self):

        # read test positve images

        my\_list = os.listdir('test/positive')

        self.image\_names\_test.extend(my\_list)

        for current in my\_list:

            image = cv2.imread("test/positive/" + current)

            image = cv2.cvtColor(image, cv2.COLOR\_BGR2RGB)

            self.test\_images.append(image)

        self.number\_of\_test\_positive = len(self.test\_images)

        # read test negative images

        my\_list = os.listdir('test/negative')

        self.image\_names\_test.extend(my\_list)

        for current in my\_list:

            image = cv2.imread("test/negative/" + current)

            image = cv2.cvtColor(image, cv2.COLOR\_BGR2RGB)

            self.test\_images.append(image)

        # convert into numpy array

        self.test\_images = np.array(self.test\_images)

        # convert into grascale image

        self.to\_grayscale(self.test\_images, train=False)

        # gradient operation to calculate magnitude and angle

        self.gradient\_operation(self.gray\_test\_images, train=False)

        # calculate the hog descriptor

        self.hog\_feature(self.\_magnitude\_test, self.\_angle\_test, train=False)

    # function to evaluate on training images

    def evaluate(self, train=False):

        output\_class = dict()

        output\_dist = dict()

        output\_file = dict()

        # if train is False we evaluate on test descriptor

        if not train:

            discriptor = self.test\_discript

        else:

        # if train is True we evaluate on train descriptor

            discriptor = self.training\_discript

        for classes, disc in discriptor.items():

            distance = np.empty(0)

            class\_distance = np.empty(0)

            disc = np.array(disc)

            for classes\_tr, disc\_tr in self.training\_discript.items():

                disc\_tr = np.array(disc\_tr)

                # calculating the similarity index

                denominator = np.sum(disc\_tr)

                numerator = np.minimum(disc\_tr,disc)

                numerator = np.sum(numerator)

                # store the distance and the class the distance belongs to

                distance = np.append(distance, numerator/denominator)

                class\_distance = np.append(class\_distance, classes\_tr)

            c = {A: B for A, B in zip(class\_distance, distance)}

            c = dict(sorted(c.items(), key=lambda item: item[1], reverse=True))

            i = 0

            temp1 = list()

            temp2 = list()

            temp3 = list()

            while i < 3:

                cl = list(c.keys())[i]

                f = cl.split('-')[0]

                temp3.append(f)

                cl = cl.split('-')[-1]

                temp1.append(cl)

                di = list(c.values())[i]

                temp2.append(di)

                i += 1

            # output for evaluate, gives distance, classification and file name for 3NN

            output\_class[classes] = temp1

            output\_dist[classes] = temp2

            output\_file[classes] = temp3

        return output\_class, output\_dist, output\_file

    # calculates the HOG Feature

    def hog\_feature(self, magn, angle, train=True):

        for index, image in enumerate(magn):

            row = 0

            image\_discriptor = np.empty(0)

            while row <= 160:

                # block size 16 x 16 pixels

                row\_start = row

                row\_end = row + 16

                if not row\_end <= 160:

                    break

                col = 0

                while col <= 96:

                    count\_col = 0

                    column\_start = col

                    column\_end = col + 16

                    if column\_end <= 96:

                        # gets the magnitude and angle of current block

                        current\_block\_magnitude = magn[index][row\_start:row\_end, column\_start:column\_end]

                        current\_block\_angle = angle[index][row\_start:row\_end, column\_start:column\_end]

                        # calculate normalised descriptor of the block

                        hist = self.HOG\_cell(current\_block\_magnitude, current\_block\_angle)

                        # concatenate to find global descriptor

                        image\_discriptor = np.concatenate((image\_discriptor, hist), axis=0)

                    else:

                        break

                    col += 8

                row += 8

            # save for train or test images

            if train:

                if index < self.number\_of\_train\_positive:

                    self.training\_discript[self.image\_names\_train[index] + '-positive'] = image\_discriptor

                else:

                    self.training\_discript[self.image\_names\_train[index] + '-negative'] = image\_discriptor

            else:

                if index < self.number\_of\_test\_positive:

                    self.test\_discript[self.image\_names\_test[index] + '-positive'] = image\_discriptor

                else:

                    self.test\_discript[self.image\_names\_test[index] + '-negative'] = image\_discriptor

    # calculate HOG Descriptor for the block

    def HOG\_cell(self, current\_block\_magnitude, current\_block\_angle):

        # calculate HOG Descriptor for first HOG Cell

        hist1 = self.HOG\_cell\_histogram(current\_block\_magnitude[0:8, 0:8], current\_block\_angle[0:8, 0:8])

        # calculate HOG Descriptor for second HOG Cell

        hist2 = self.HOG\_cell\_histogram(current\_block\_magnitude[0:8, 8:16], current\_block\_angle[0:8, 8:16])

        # calculate HOG Descriptor for third HOG Cell

        hist3 = self.HOG\_cell\_histogram(current\_block\_magnitude[8:16, 0:8], current\_block\_angle[8:16, 0:8])

        # calculate HOG Descriptor for forth HOG Cell

        hist4 = self.HOG\_cell\_histogram(current\_block\_magnitude[8:16, 8:16], current\_block\_angle[8:16, 8:16])

        # concatenate to get HOG Descriptor for block

        hist = np.concatenate((hist1, hist2, hist3, hist4), axis=0)

        # L2 Norm

        norm = math.sqrt(np.sum(np.power(hist, 2)))

        if norm > 0:

            # Divide by L2 Norm

            hist = hist / norm

        return hist

    # Calculate HOG Histogram for one 8x8 cell

    def HOG\_cell\_histogram(self, current\_block\_magnitude, current\_block\_angle):

        # bins 1-9

        bins = [0] \* 9

        i = 0

        while i <= current\_block\_magnitude.shape[0] - 1:

            j = 0

            while j <= current\_block\_magnitude.shape[1] - 1:

                angle = current\_block\_angle[i, j]

                # if angle is greater than 180 we subtract 180

                if angle >= 180:

                    angle = angle - 180

                # if angle is less than 0 we add 180

                if angle < 0:

                    angle = angle + 180

                # we split the block magnitude between two closest bins according to gradient angle

                if angle > 10 and angle <= 30:

                    bins[0] += current\_block\_magnitude[i, j] \* ((30 - angle) / 20)

                    bins[1] += current\_block\_magnitude[i, j] \* ((angle - 10) / 20)

                elif angle > 30 and angle <= 50:

                    bins[1] += current\_block\_magnitude[i, j] \* ((50 - angle) / 20)

                    bins[2] += current\_block\_magnitude[i, j] \* ((angle - 30) / 20)

                elif angle > 50 and angle <= 70:

                    bins[2] += current\_block\_magnitude[i, j] \* ((70 - angle) / 20)

                    bins[3] += current\_block\_magnitude[i, j] \* ((angle - 50) / 20)

                elif angle > 70 and angle <= 90:

                    bins[3] += current\_block\_magnitude[i, j] \* ((90 - angle) / 20)

                    bins[4] += current\_block\_magnitude[i, j] \* ((angle - 70) / 20)

                elif angle > 90 and angle <= 110:

                    bins[4] += current\_block\_magnitude[i, j] \* ((110 - angle) / 20)

                    bins[5] += current\_block\_magnitude[i, j] \* ((angle - 90) / 20)

                elif angle > 110 and angle <= 130:

                    bins[5] += current\_block\_magnitude[i, j] \* ((130 - angle) / 20)

                    bins[6] += current\_block\_magnitude[i, j] \* ((angle - 110) / 20)

                elif angle > 130 and angle <= 150:

                    bins[6] += current\_block\_magnitude[i, j] \* ((150 - angle) / 20)

                    bins[7] += current\_block\_magnitude[i, j] \* ((angle - 130) / 20)

                elif angle > 150 and angle <= 170:

                    bins[7] += current\_block\_magnitude[i, j] \* ((170 - angle) / 20)

                    bins[8] += current\_block\_magnitude[i, j] \* ((angle - 150) / 20)

                elif angle >= 0 and angle <= 10:

                    bins[0] += current\_block\_magnitude[i, j] \* ((angle + 10) / 20)

                    bins[8] += current\_block\_magnitude[i, j] \* ((10 - angle) / 20)

                elif angle > 170 and angle <= 180:

                    bins[8] += current\_block\_magnitude[i, j] \* ((190 - angle) / 20)

                    bins[0] += current\_block\_magnitude[i, j] \* ((angle - 170) / 20)

                else:

                    print('some error')

                j += 1

            i += 1

        return np.array(bins)

    # convert to grayscale

    def to\_grayscale(self, images, train=True):

        # for train

        if train:

            for image in images:

                # extract R G B channels

                R, G, B = image[:,:,0], image[:,:,1], image[:,:,2]

                # weighted formulae to convert to grayscale

                new\_image = np.round((0.299 \* R) + (0.587 \* G) + (0.114 \* B))

                self.gray\_train\_images.append(new\_image)

            self.gray\_train\_images = np.array(self.gray\_train\_images)

        else:

        # for test

            for image in images:

                # extract R G B channels

                R, G, B = image[:,:,0], image[:,:,1], image[:,:,2]

                # weighted formulae to convert to grayscale

                new\_image = np.round((0.299 \* R) + (0.587 \* G) + (0.114 \* B))

                self.gray\_test\_images.append(new\_image)

            self.gray\_test\_images = np.array(self.gray\_test\_images)

    # gradient operation

    def gradient\_operation(self, images, train=True):

        # if train

        if train:

            for image in images:

                # Convolution done on the image\_matrix w.r.t gradient x

                gradient\_x = SeConvolve(image, self.\_convolution\_matrix\_gx, mode='gradient')

                gradient\_x\_train, \_ = gradient\_x.convolution()

                self.\_gradient\_x\_train.append(gradient\_x\_train)

                # Convolution done on the image\_matrix w.r.t gradient y

                gradient\_y = SeConvolve(image, self.\_convolution\_matrix\_gy, mode='gradient')

                gradient\_y\_train, \_ = gradient\_y.convolution()

                self.\_gradient\_y\_train.append(gradient\_y\_train)

                # We compute gradient magnitude, gradient angle

                magnitude\_x\_train = self.calcuate\_magnitude(gradient\_x\_train, gradient\_y\_train)

                angle\_x\_train = self.calculate\_angle(gradient\_x\_train, gradient\_y\_train)

                self.\_magnitude\_train.append(magnitude\_x\_train)

                self.\_angle\_train.append(angle\_x\_train)

            # convert to numpy array

            self.\_gradient\_x\_train = np.array(self.\_gradient\_x\_train)

            self.\_gradient\_y\_train = np.array(self.\_gradient\_y\_train)

            self.\_magnitude\_train = np.array(self.\_magnitude\_train)

            self.\_angle\_train = np.array(self.\_angle\_train)

        else:

            for image in images:

                # Convolution done on the image\_matrix w.r.t gradient x

                gradient\_x = SeConvolve(image, self.\_convolution\_matrix\_gx, mode='gradient')

                gradient\_x\_train, \_ = gradient\_x.convolution()

                self.\_gradient\_x\_test.append(gradient\_x\_train)

                # Convolution done on the image\_matrix w.r.t gradient y

                gradient\_y = SeConvolve(image, self.\_convolution\_matrix\_gy, mode='gradient')

                gradient\_y\_train, \_ = gradient\_y.convolution()

                self.\_gradient\_y\_test.append(gradient\_y\_train)

                # We compute gradient magnitude, gradient angle and edge angle

                magnitude\_x\_train = self.calcuate\_magnitude(gradient\_x\_train, gradient\_y\_train)

                angle\_x\_train = self.calculate\_angle(gradient\_x\_train, gradient\_y\_train)

                self.\_magnitude\_test.append(magnitude\_x\_train)

                self.\_angle\_test.append(angle\_x\_train)

            # convert to numpy array

            self.\_gradient\_x\_test = np.array(self.\_gradient\_x\_test)

            self.\_gradient\_y\_test = np.array(self.\_gradient\_y\_test)

            self.\_magnitude\_test = np.array(self.\_magnitude\_test)

            self.\_angle\_test = np.array(self.\_angle\_test)

    def calcuate\_magnitude(self, gradient\_x, gradient\_y):

        height, width = gradient\_x.shape

        # After gaussing smoothing and gradient computation we have lost a total of 8 rows and 8 columns

        magnitude = np.zeros((height - 2, width - 2))

        # looping over the desired matrix

        for i in range(1,height - 1):

            for j in range(1,width - 1):

                # gradient calculated using root(gx\*\*2 + gy\*\*2)

                temp = (gradient\_x[i, j] \*\* 2) + (gradient\_y[i, j] \*\* 2)

                magnitude[i - 1, j - 1] = math.sqrt(temp)

        # same size as original image

        magnitude = np.pad(magnitude, 1, mode='constant')

        # normalize the magnitude between [0, 255]

        magnitude = ((magnitude - magnitude.min()) \* (1/(magnitude.max() - magnitude.min()) \* 255)).astype(int)

        return magnitude

    def calculate\_angle(self, gradient\_x, gradient\_y):

        height, width = gradient\_x.shape

        # After gaussing smoothing and gradient computation we have lost a total of 8 rows and 8 columns

        angle = np.zeros((height - 2, width - 2))

        edge\_angle = np.zeros((height - 2, width - 2))

        # looping over the desired matrix

        for i in range(1,height - 1):

            for j in range(1,width - 1):

                if gradient\_x[i, j]  != 0:

                    # returns angle between -180 to 180

                    angle[i - 1, j - 1] = math.degrees(math.atan2(gradient\_y[i, j], gradient\_x[i, j]))

        # same size as original image

        angle = np.pad(angle, 1, mode='constant')

        return angle

* **main.py**

from HOG import HOGDescriptor

import matplotlib.pyplot as plt

# main block

if \_\_name\_\_ == '\_\_main\_\_':

    # call HOG class

    hg = HOGDescriptor()

    # Fit on training data

    hg.fit()

    # Transform test data

    hg.test()

    # evaluate the data on test images

    print(f'For Test Images{hg.evaluate()}')

    # evaluate the data on train images

    print(f'\n\n\nFor Train Images{hg.evaluate(train=True)}')

    # HOG ASCII Derscriptors

    for name in ['crop001028a.bmp-positive', 'crop001030c.bmp-positive', '00000091a\_cut.bmp-negative']:

        with open('Output/' + name.split('-')[0] + '.txt', 'w',encoding='utf-8') as file:

            file.writelines( "%s\n" % item for item in hg.training\_discript[name])

    for name in ['crop001278a.bmp-positive', 'crop001500b.bmp-positive', '00000090a\_cut.bmp-negative']:

        with open('Output/' + name.split('-')[0] + '.txt', 'w',encoding='utf-8') as file:

            file.writelines( "%s\n" % item for item in hg.test\_discript[name] )

    # Gradient Magnitude of Test Images

    for i in range(0, hg.\_magnitude\_test.shape[0]):

        plt.imsave('Magnitude/' + hg.image\_names\_test[i], hg.\_magnitude\_test[i], cmap='gray')

* **convolution.py**

####################################################

## Developed by: Eashan Kaushik & Srijan Malhotra ##

## Project Start: 29th October 2021               ##

####################################################

import numpy as np

class SeConvolve:

    def \_\_init\_\_(self, image\_matrix, kernel, mode='gradient'):

        # input image matrix

        self.image\_matrix = image\_matrix

        # inpur kernel

        self.kernel = kernel

        # output after convolution with the kerne

        self.\_output = None

        # normalized output

        self.\_output\_norm = None

        # mode smoothing or gradient

        self.mode = mode

    #######################

    ## Getter and Setter ##

    #######################

    @property

    def output(self):

        return self.\_output

    @property

    def output\_norm(self):

        return self.\_output\_norm

    #######################

    #######################

    def convolution(self):

      height, width = self.image\_matrix.shape

      if self.mode == 'gradient':

        # code to find the gradients

        self.\_output = np.zeros((height - 2, width - 2))

        # looping over the desired matrix

        for i in range(1,height - 1):

          for j in range(1,width - 1):

            # martix multiplication for gradient computation

            # prewit convolution after gaussian smoothing leads to loss of 4 rows and 4 columns thats why we start from i-4 and j-4

            self.\_output[i - 1,j - 1] = (self.kernel[0, 0] \* self.image\_matrix[i - 1, j - 1]) + (self.kernel[0, 1] \* self.image\_matrix[i - 1, j]) + (self.kernel[0, 2] \* self.image\_matrix[i - 1, j + 1]) + \

                      (self.kernel[1, 0] \* self.image\_matrix[i, j - 1]) + (self.kernel[1, 1] \* self.image\_matrix[i, j]) + (self.kernel[1, 2] \* self.image\_matrix[i, j + 1]) + (self.kernel[2, 0] \* self.image\_matrix[i + 1, j - 1]) + \

                      (self.kernel[2, 1] \* self.image\_matrix[i + 1, j]) + (self.kernel[2, 2] \* self.image\_matrix[i + 1, j + 1])

      # we call the normalize function to normalize the output

      self.normalize()

      self.\_output = np.pad(self.\_output, 1, mode='constant')

      return self.\_output, self.\_output\_norm

    # normalize

    def normalize(self):

      if self.mode == 'gradient':

        # normalize using sum of absolute values

        temp\_output = np.absolute(self.\_output)

        self.\_output\_norm = temp\_output / 3

        self.\_output\_norm = np.pad(self.\_output\_norm, 1, mode='constant')

**Classification Table**

