Computer Vision Project 2
HUMAN DETECTION

Aryaa Singh
NYU Id - N17541390
Net Id - as13538

INSTRUCTIONS TO COMPILE CODE:

1. The source file is a .py file, code was written in python, so to compile the code, just copy the code into any python text editor, Eg- sublime text or Jupyter notebook.
2. Include all the libraries which have been imported in the start of the code.
3. The image to be processed will be in the same folder as the python file.
4. Original implementation environment for this project was Jupyter notebook, the file was then converted to python .py file.
5. Copying the source code in a .txt file and word file has affected its indentation.

SOURCE CODE FILE NAME: humandetection.txt

EXECUTABLE CODE FILE NAME: humandetection.py

RESULTS

Results with 200 hidden layers

Epoch Count : 100 Iterations
Initialization Method : Random [0, 1]
Average Error HOG: 25.07 %
Average Error HOG+LBP = 26.4195 %

Input Image	**Correct Class**	**HOG only**		**HOG-LBP**	
200 hidden layer		**Output**	**Classification**	**Output**	**Classification**
crop001034b	Human	0.3544426	No human	0.321717	No human
crop001070a	Human	0.9513709	Human	0.9480844	Human
crop001278a	Human	0.9551899	Human	0.9534623	Human
crop001500b	Human	0.5206545	Borderline	0.6685937	Human
person_and_bike_151a	Human	0.9618364	Human	0.9697206	Human
00000003a_cut	No-human	0.1232204	No human	0.1164448	No human
00000090a_cut	No-human	0.0219161	No human	0.0306231	No human
00000118a_cut	No-human	0.2313648	No human	0.170281	No human
no_person_no_bike_258_cut	No-human	0.6353688	Human	0.7985	Human
no_person_no_bike_264_cut	No-human	0.2390461	No human	0.3876791	No human

Results with 400 hidden layers

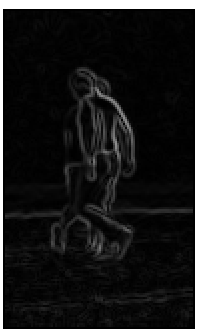
Epoch Count : 100 Iterations
Initialization Method : Random [0, 1]
Average Error HOG: 22.90 %
Average Error HOG+LBP = 23.79 %

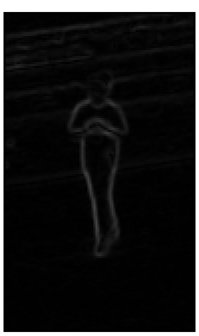
Input Image	Correct Class	HOG only		HOG-LBP	
400 hidden layer		**Output**	**Classification**	**Output**	**Classification**
crop001034b	Human	0.3966745	No human	0.3189758	No human
crop001070a	Human	0.9317146	Human	0.9494015	Human
crop001278a	Human	0.9651189	Human	0.9507023	Human
crop001500b	Human	0.7357599	Human	0.7971799	Human
person_and_bike_151a	Human	0.9719554	Human	0.9733341	Human
00000003a_cut	No-human	0.1520739	No human	0.1180313	No human
00000090a_cut	No-human	0.0132491	No human	0.0112043	No human
00000118a_cut	No-human	0.1355971	No human	0.1692405	No human
no_person_no_bike_258_cut	No-human	0.7068172	Human	0.750512	Human
no_person_no_bike_264_cut	No-human	0.2835926	No human	0.3193923	No human

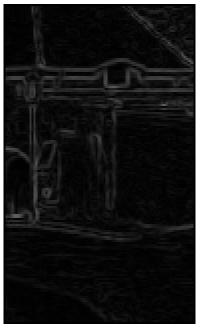
Average Error

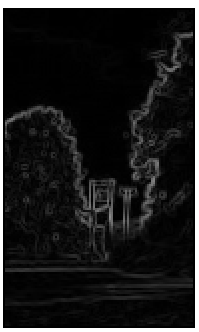
HOG 200	HOG-LBP 200	HOG 400	HOG-LBP 400
0.645557	0.678283	0.603325	0.681024
0.048629	0.051916	0.068285	0.050599
0.04481	0.046538	0.034881	0.049298
0.479345	0.331406	0.26424	0.20282
0.038164	0.030279	0.028045	0.026666
0.12322	0.116445	0.152074	0.118031
0.021916	0.030623	0.013249	0.011204
0.231365	0.170281	0.135597	0.16924
0.635369	0.7985	0.706817	0.750512
0.239046	0.387679	0.283593	0.319392

IMAGES









SOURCE CODE

```
# coding: utf-8

import glob              # To get all filenames of the imagess in the given folder
import cv2               # To get image reading and writing functions
import numpy as np       #  array manipulations
import math              # math functions, square root.

# Function to calculate RELU

def RELU(x):
   for i in range(x.shape[0]):
      for j in range(x.shape[1]):
         if x[i,j]<0:
            x[i,j]=0
   return x

# Function to calculate sigmoid

def sigmoid(x):
   return 1/(1+np.exp(-x))

# Function to convert the image to grayscale image

def grayScale(img):
   conversion = np.array([0.229,0.587,0.114])                    # List to multiply to get grayscale image
   gray_img_array = np.around(np.dot(img,conversion))              # Taking dot product and then
rounding off to get grayscale image
```

```
    return gray_img_array

# Function to proportionately divide gradient magnitude into histogram bins

def divide(mag, ang, x):
    temp = abs(x-ang)/20                    # Dividing the magnitude and then returning
    return temp*mag,(1-temp)*mag

# Function to compute gradient angle, and then wrapping it around -10 to 170

def angleCalc(y,x):
    ang = np.degrees(np.arctan2(y,x))              # To compute the gradient angle and then converting it into
degrees
    for i in range(ang.shape[0]):
        for j in range(ang.shape[1]):
            if ang[i,j]<-10:                        # Mapping negative angles
                ang[i,j]+=180
            elif ang[i,j]>=170:                      # Mapping angles greater than 170
                ang[i,j]-=180
    return ang


# FUNCTION FOR CALCULATING GRADIENT

def grCalc(img,fd,sd):
    # DEFINING PREWITT OPERATORS FOR GRADIENT CALCULATION
    x, y = np.array([[-1,0,1],[-2,0,2],[-1,0,1]]), np.array([[1,2,1],[0,0,0],[-1,-2,-1]])
    prfd, prsd = x.shape                          # Storing the dimensions of sobel masks into variables
```

```
    gx = np.zeros((fd,sd),dtype = np.float)                 # Defining gradient x array
    gy = np.zeros((fd,sd),dtype = np.float)                 # Defining gradient y array
    ngx = np.zeros((fd,sd),dtype = np.float)                # Defining the normalized gradient x array
    ngy = np.zeros((fd,sd),dtype = np.float)                # Defining the normalized gradient y array
    gm = np.zeros((fd,sd),dtype = np.float)                 # Defining the gradient magnitude array
    temp = np.zeros((prfd,prsd),dtype = np.float)     # Defining temporary array that will store the slice of smoothed image array for direct matrix multiplication
    for i in range(fd-prfd+1):
        for j in range(sd-prfd+1):
            temp = img[(i):(3+i),(j):(3+j)]       # Storing the slice of smoothed image array in temporary array
            gx[1+i,1+j] = np.sum(np.multiply(temp, x))  # Applying convolution for gradient x by directly multpilying the slice of matrix with sobel x operator
            gy[1+i,1+j] = np.sum(np.multiply(temp, y))  # Applying convolution for gradient y by directly multiplying the slice of matrix with sobel y operator
    ngx = np.absolute(gx)/4               # Forming normalized gradient x matrix from gradient x matrix by taking absolute value using np.absolute() and dividing by three
    ngy = np.absolute(gy)/4               # Forming normalized gradient y matrix from gradient y matrix by taking absolute value using np.absolute() and dividing by three
    gm = np.hypot(ngx,ngy)/np.sqrt(2)       # Forming the normalized gradient magnitude array by using np.hypot() which takes under root of sum of squares of normalized gradient x and normalized gradient y and then dividing by square root of 2 for normalization
    return np.around(gx),np.around(gy),np.around(ngx),np.around(ngy),np.around(gm)       # Returning gradient x, gradient y, normalized gradient x, normalized gradient y and normalized gradient magnitude


# Function to calculate the histogram of each 8x8 pixel cell, calling divide to split the gradient magnitude proportionally and
# then adding it to bins
def histCalc(ang,mag):
    hist = [0]*9
    for i in range(ang.shape[0]):
        for j in range(ang.shape[1]):
            if ang[i,j]<=0:
```

```
    mag1,mag2=divide(mag[i,j],ang[i,j],0)
    hist[8]+=mag1
    hist[0]+=mag2
elif ang[i,j]>=0 and ang[i,j]<=20:
    mag1,mag2=divide(mag[i,j],ang[i,j],20)
    hist[0]+=mag1
    hist[1]+=mag2
elif ang[i,j]>=20 and ang[i,j]<=40:
    mag1,mag2=divide(mag[i,j],ang[i,j],40)
    hist[1]+=mag1
    hist[2]+=mag2
elif ang[i,j]>=40 and ang[i,j]<=60:
    mag1,mag2=divide(mag[i,j],ang[i,j],60)
    hist[2]+=mag1
    hist[3]+=mag2
elif ang[i,j]>=60 and ang[i,j]<=80:
    mag1,mag2=divide(mag[i,j],ang[i,j],80)
    hist[3]+=mag1
    hist[4]+=mag2
elif ang[i,j]>=80 and ang[i,j]<=100:
    mag1,mag2=divide(mag[i,j],ang[i,j],100)
    hist[4]+=mag1
    hist[5]+=mag2
elif ang[i,j]>=100 and ang[i,j]<=120:
    mag1,mag2=divide(mag[i,j],ang[i,j],120)
    hist[5]+=mag1
    hist[6]+=mag2
elif ang[i,j]>=120 and ang[i,j]<=140:
    mag1,mag2=divide(mag[i,j],ang[i,j],140)
```

```
            hist[6]+=mag1
            hist[7]+=mag2
        elif ang[i,j]>=140 and ang[i,j]<=160:
            mag1,mag2=divide(mag[i,j],ang[i,j],160)
            hist[7]+=mag1
            hist[8]+=mag2
        elif ang[i,j]>=160:
            mag1,mag2=divide(mag[i,j],ang[i,j],160)
            hist[0]+=mag1
            hist[8]+=mag2
    return hist



# Function to calculate the L2 Norm of each histogram, taking 2x2 cells and returning 36xq vector


def normalize(histogram):
    total, hist = 0, []
    for i in range(2):
        for j in range(2):
            for k in range(9):
                total += (histogram[i,j,k]*histogram[i,j,k])            # Taking square sum of each value
                hist.append(histogram[i,j,k])
    lval = math.sqrt(total)                                 # Taking sqaure root of square sum
    hist = np.array(hist)                        # Converting list to numpy array for easier calculations
    if lval!=0:                                 # If not zero only then divide else let it be, it will remain 0
        hist = hist/lval
    return hist
```

Function to calculate descriptor of all images, it calls histCalc to calculate histograms of all 8x8 cells, normalize to do L2

normalization

```
def HOG(ang,mag):
    x = int(ang.shape[0]/8)
    y = int(ang.shape[1]/8)                    # Getting histogram
    hist = np.zeros((x,y,9))
    idx = [0,0]
    for i in range(x):
        for j in range(y):
            temp = histCalc(ang[idx[0]:(idx[0]+8),idx[1]:(idx[1]+8)],mag[idx[0]:(idx[0]+8),idx[1]:(idx[1]+8)])
            hist[i,j]=temp
            idx[1] += 8
        idx[0] += 8
        idx[1] = 0
    hist_norm = []                              # Histogram equalization
    for i in range(x-1):
        for j in range(y-1):
            temp = normalize(hist[i:(i+2),j:(j+2)])
            hist_norm.extend(temp.tolist())
    hist_norm = np.array(hist_norm)
    return hist_norm


def bp(block):
    hist={}
    a=[0, 1, 2, 3, 4, 5, 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, 255]       # Bin patterbns
hist = {el:0 for el in a}
for row in range(block.shape[0]):                               # Calculating binary patterns
  for col in range(block.shape[1]):
    arr=[]
    if (row == 0 or col ==0 or row == block.shape[0]-1 or col == block.shape[1]-1):
      if hist[5] == 0:
        hist[5]=1
      else:
        hist[5]+=1
    else:
      for i in range(row-1, row+2):
        for j in range(col-1, col+2):
          if block[i][j] > block[row][col]:
            arr.append(1)
          else:
            arr.append(0)
      arr.pop(4)
      arr.reverse()
      warr=np.where(arr)[0]
      if len(warr)>=1:
        num=0
        for n in warr:
          num+=2**n
      else:
        num=0
      if num in a and num != 5:
```

```
                if hist[num]==0:
                    hist[num]=1
                else:
                    hist[num]+=1
    return hist

 #function to calculate local binary pattern
def lbp(img, r, c):
    x=int(r/16)
    y=int(c/16)
    index=[0,0]
    histogram=np.zeros((x,y,59))
    for i in range(x):
        for j in range(y):
            temp=bp(img[index[0]:(index[0]+16), index[1]:index[1]+16])
            temp=list(temp.values())
            histogram[i,j]=temp
            index[1]+=16
        index[0]+=16
        index[1]=0
    flt = []
    for i in range(x):                                  # Normalize the histogram
        for j in range(y):
            ssum=0
            norm_histo=[]
            for k in range(59):
                ssum+=(histogram[i,j,k]*histogram[i,j,k])
                norm_histo.append(histogram[i,j,k])
            lval=math.sqrt(ssum)
```

```
        norm_histo=np.array(norm_histo)
        if lval!=0:
            norm_histo=norm_histo/lval
        norm_histo = norm_histo.tolist()
        flt.extend(norm_histo)
    flt = np.array(flt)
    return flt
```

Function to implament neural network, to train it.

```
def neural_net(inp,output,hNeurons):
    aplha = 0.1                                   # Initializing the learning rate
    col = inp.shape[1]
    w1 = np.random.randn(col,hNeurons)                        # Weight for layer 1
    w1 = np.multiply(w1,math.sqrt(2/int(col+hNeurons)))          # Faactoring the weight
    w2 = np.random.randn(hNeurons,1)                         # Weight for layer 2
    w2 = np.multiply(w2,math.sqrt(2/int(hNeurons+1)))          # Factoring the weight
    w1bias = np.random.randn(hNeurons)                        # Bias for layer 1
    w1bias = np.multiply(w1bias,math.sqrt(2/int(hNeurons)))
    w2bias = np.random.randn(1)                          # Bias for layer 2
    w2bias = np.multiply(w2bias,math.sqrt(2/int(1)))
    err_curve=np.zeros((100,1))                          # Error array for each epoch
    epoch = 0
    while epoch<100:                               # Doing forward and backward propogation for each epoch
        for i in range(inp.shape[0]):
            x = inp[i,:].reshape([1,-1])
            z = RELU((x.dot(w1)+w1bias))                   # Computing values for hidden layer
            y = sigmoid((z.dot(w2)+w2bias))                # Computing values for output layer
            err = output[i]-y                          # Error for output layer
```

```
            sqerr = 0.5*err*err                              # Square error
            del_out=(-1*err)*(1-y)*y
            del_layer2=z.T.dot(del_out)
            del_layer20=np.sum(del_out,axis=0)
            zz=np.zeros_like(z)
            for k in range(hNeurons):

              if(z[0][k]>0):
                zz[0][k]=1
              else:
                zz[0][k]=0
            del_hNeurons= del_out.dot(w2.T)*zz
            del_layer1=x.T.dot(del_hNeurons)
            delta_layer10=np.sum(del_hNeurons,axis=0)

            w2-= aplha*del_layer2
            w2bias-= aplha*del_layer20
            w1-= aplha*del_layer1
            w1bias-= aplha*delta_layer10
            err_curve[epoch] = sqerr/inp.shape[0]
        print('Epoch %d: err %f'%(epoch,np.mean(sqerr)/inp.shape[0]))
        epoch +=1
    return w1,w1bias,w2,w2bias,err_curve



# Function to predict values for my neural network


def predict(w,wb,v,vb,Output_descriptor):
    Number_of_test_image,number_of_attribute=Output_descriptor.shape
```

```
    predict=[]
    for k in range(Number_of_test_image):
        x=Output_descriptor[k,:].reshape([1,-1])
        z=RELU((x.dot(w)+wb))
        y=sigmoid(z.dot(v)+vb)
        predict.append(y)
    return predict

# Main function that calls every other function

TrainPath1 = 'train_images_pos'
TrainPath2 = 'train_images_neg'
TestPath1 = 'test_images_pos'
TestPath2 = 'test_images_neg'

ImgTrain = []          # List to store all training images
OutTrain = []          # List to store all training output
ImgTest = []           # List to store all testing images
OutTest = []           # List to store all training output

for filename in glob.glob(TrainPath1+'/*.bmp'):     # Getting all the filenames of positive images
    img = np.array(cv2.imread(filename, cv2.IMREAD_COLOR))
    ImgTrain.append(grayScale(img))                 # Appending to the train image array
    OutTrain.append(1)                  # Appending to the test array, the value 1 for positive

for filename in glob.glob(TrainPath2+'/*.bmp'):     # Getting all file names of negative images
    img = np.array(cv2.imread(filename, cv2.IMREAD_COLOR))
    ImgTrain.append(grayScale(img))                 # Appending to train image array
    OutTrain.append(0)                  # Appending to the test array, the value 0 for negative
```

```
for filename in glob.glob(TestPath1+'/*.bmp'):   # Getting all the filenames of positive images
    img = np.array(cv2.imread(filename, cv2.IMREAD_COLOR))
    ImgTest.append(grayScale(img))                        # Appending to the train image array
    OutTest.append(1)                         # Appending to the test array, the value 1 for positive

for filename in glob.glob(TestPath2+'/*.bmp'):  # Getting all file names of negative images
    img = np.array(cv2.imread(filename, cv2.IMREAD_COLOR))
    ImgTest.append(grayScale(img))                       # Appending to train image array
    OutTest.append(0)                         # Appending to the test array, the value 0 for negative


fvector = np.zeros((20,7524))                 # Creating HOG descriptor for training images
fvector2 = np.zeros((10,7524))                # Creating HOG descriptor for test images
newfvector = np.zeros((20,11064))
newfvector2=np.zeros((10,11064))


for i in range(len(ImgTrain)):
    gx,gy,gxn,gyn,gm = grCalc(ImgTrain[i],ImgTrain[i].shape[0],ImgTrain[i].shape[1])    # Calculating
gradient magnitude of all training images
    ga = angleCalc(gy,gx)                        # Calculating gradient angle of all training images
    fvector[i] = HOG(ga,gm)        # Storing HOG descriptor for all images
    lvector = lbp(ImgTrain[i] ,ImgTrain[i].shape[0],ImgTrain[i].shape[1])
    newfvector[i] = np.concatenate((fvector[i],lvector))


for i in range(len(ImgTest)):
```

```
    gx,gy,gxn,gyn,gm = grCalc(ImgTest[i],ImgTest[i].shape[0],ImgTest[i].shape[1])  # Calculating gradient
magnitude of all testing images
    ga = angleCalc(gy,gx)                    # Calculating gradient angle of all training images
    cv2.imwrite('gradients{}.bmp'.format(i),gm) # Writing the images to directory
    fvector2[i] = HOG(ga,gm)      # Storing HOG descriptor for all images
    lvector2 = lbp(ImgTest[i] ,ImgTest[i].shape[0],ImgTest[i].shape[1])
    newfvector2[i] = np.concatenate((fvector2[i],lvector2))


hNeurons = [200,400]    # List with values of Hidden neurons

#HOG
for i in range(len(hNeurons)):   # running neural networks for different values of hidden neurons
    print('HIDDEN LAYER = %d'%(hNeurons[i]))
    print('\n\n')
    w1,w1bias,w2,w2bias,err_curve = neural_net(fvector,np.array(OutTrain),hNeurons[i])
    predicted_output=predict(w1,w1bias,w2,w2bias,fvector2)
    pre=[]

    for check in predicted_output:
        if(check >=0.5):
            pre.append(1)
        else:
            pre.append(0)
        print(check)

    print(len(pre))

    correct=0
```

```
    wrong=0

    for i in range(len(pre)):
        if(pre[i]==OutTest[i]):
            correct+=1
        else:
            wrong+=1

    print('correct = %d'%(correct))
    print('wrong = %d'%(wrong))
    print(pre)
    print(OutTest)
    print('\n\n\n')

# HOG + LBP
for i in range(len(hNeurons)):   # running neural networks for different values of hidden neurons
    print('Hidden Layers = %d'%(hNeurons[i]))
    print('\n\n')
    w1,w1bias,w2,w2bias,err_curve = neural_net(newfvector,np.array(OutTrain),hNeurons[i])
    predicted_output=predict(w1,w1bias,w2,w2bias,newfvector2)
    pre=[]

    for check in predicted_output:
        if(check >=0.5):
            pre.append(1)
        else:
            pre.append(0)
        print(check)
```

```
print(len(pre))


correct=0
wrong=0


for i in range(len(pre)):
    if(pre[i]==OutTest[i]):
        correct+=1
    else:
        wrong+=1


print('Correct = %d'%(correct))
print('Wrong = %d'%(wrong))
print(pre)
print(OutTest)
print('\n\n\n')
```