FALL 22 ECE 661 – COMPUTER VISION

Homework 10

Sahithi Kodali – 34789866
kodali1@purdue.edu

TASK – 1: Face Recognition using PCA and LDA

This task involves using Principal Component Analysis (PCA) and Linear Discriminant Analysis (LDA) for dimensionality reduction and then using Nearest Neighbors (NN) for Face recognition.

1.1 Principal Component Analysis (PCA)

PCA works by the eigen decomposition of the covariance matrix of the input image vectors. An orthogonal set of vectors are computed by attaining the largest eigen vectors of image feature matrix to project the input image vectors on to. These eigen vectors are fewer in number than input images and hence can be termed as dimensionality reduction and are further used as feature vectors for face recognition using NN. The feature vectors using PCA can be obtained as below.

i. Vectorize an image *(i)* and normalize each vectorized image (x_i) by its size.

$$x_i\_norm = x_i / ||x_i||$$

ii. Estimate the mean of all images *(m)* and subtract this from the normalized images. These results are added as the columns of matrix X and the orthogonal matrix X^TX goes through eigen decomposition to give eigen vectors $(\lambda`_k)$ and eigen values.

$$m = \frac{1}{N} \sum_{i=1}^{N} x_i\_norm; \text{ where } N = \text{number of images}$$

$$\lambda`_k = [\lambda`_1 \mid \lambda`_2 \mid \lambda`_3 \mid \ldots\ldots \mid \lambda`_k]$$

iii. Sort the eigen vectors to pick the '*k'* largest eigen vectors (showing dimensionality reduction) and, compute the eigen vectors of image matrix X (λ_i) using these and normalize them.

$$\lambda_i = X \lambda`_i$$

$$\lambda_i\_norm = \lambda_i / || \lambda_i||$$

iv. Retain the *P* largest eigen vectors as column matrix of W_P and compute the feature vectors y_i for both training and testing image data.

$$y_i = W_P^T \ (x_i\_norm - m)$$

Once the feature vectors using PCA are obtained as above, an own Nearest Neighbors (NN) classification is implemented for face recognition. NN works by finding the Euclidean distance between a given test image and all train image vectors to find the class that is nearest and predict the class it classifies into. A single nearest neighbor is used (i.e., k = 1) for predicting the class.

1.2 Linear Discriminant Analysis (LDA)

LDA main goal is to find the maximal discriminating direction vector between the image classes rather than within an image class. It is done by maximizing the ratio of within class (S_w) is to between class (S_b) scatter of the feature vectors which is done by estimating the eigen vectors that maximize the Fisher Discriminant function (J), if S_w is non-singular.

$$J(w) = \frac{w^T S_B w}{w^T S_W w}$$

However, when S_w is singular this cannot be done. The following steps are used to obtain the feature vectors in this case.

i. Vectorize an image *(i)* and normalize each vectorized image (x_i) by its size.

$$x_i\_norm = x_i / ||x_i||$$

ii. Estimate the mean of all images (m_N) and the mean of images within a class C (m_C). Now compute the mean matrix *M* by subtracting these two means and adding them in each column of M for every class C.

$$m_N = \frac{1}{N} \sum_{i=1}^{N} x_i\_norm; \text{ where } N = \text{number of images}$$

$$m_C = \frac{1}{C} \sum_{i=1}^{C} x_i\_norm; \text{ where } C = \text{class of images}$$

$$M = m_C - m_N \text{ (for every Class C and as each column in M)}$$

iii. The orthogonal matrix M^TM goes through eigen decomposition to give eigen vectors and eigen values. Sort the eigen vectors and, compute the eigen vectors of image matrix M (λ_i) using these and normalize them.

$$\lambda_i = M \lambda`_i$$

$$\lambda_i\_norm = \lambda_i / || \lambda_i||$$

iv. Now, create a matrix (λ_k) with the '*k'* largest eigen vectors (showing dimensionality reduction) that are not nearly zero (i.e., greater than 1e-6) since singular matrix inverse does not exist.

$$\lambda_k = [\lambda`_1 | \lambda`_2 | \lambda`_3 | \ldots\ldots | \lambda`_k]$$

v. Now create a diagonal matrix (D) with the k nonzero eigen values as diagonal values. Compute *Z* and matrix *X* with each column having the within- class difference vectors $X_i = x_i\_norm - m_C$.

$$Z = \lambda_k D^{-0.5}$$

vi. Compute the eigenvalues and eigen vectors of (Z^TX) $(Z^TX)^T$ and sort the eigen vectors $(v`_i)$ based on the values. Now, compute the eigen vectors for the matrix Z. Retain the *P* largest eigen vectors as column matrix of W_P and compute the feature vectors y_i for both training and testing image data.

$$y_i = W_P^T (x_i\_norm - m_N)$$

Once the feature vectors are obtained for training and testing images using LCA as described above, NN classifier like in the PCA task is used to classify images and predict the class for face recognition task.

TASK – 2: Face Recognition using Autoencoder

An autoencoder which is a neural network-based model is provided for dimensionality reduction in this task. An own NN classification as stated in the previous task-1 is implemented to classify the images using the feature vectors obtained by the Autoencoder for p values = 3, 8, 16.

TASK – 3: Object Detection using Cascaded AdaBoost Classifiers

This task involves implementing the AdaBoost classifier using the Viola and Jones algorithm for object detection.

3.1 Feature Extraction

AdaBoost works by aggregating several weak classifiers to form a strong classifier for classification tasks. To perform such a classification, large number of features are required which is done by using Haar filters with different sizes and orientations. A horizontal filter of 1x2, 1x4, ….1x40 and a vertical filter of 2x2, 4x2....,20x2 are used to obtain 11900 features for the images given, each with size 20x40. An integral image is used for feature calculation for reducing computational costs.

3.2 Training – Find the best weak classifiers and build a strong classifier

The main task of training is to iterate and update the parameters until the least possible False Positive Rate (FPR) is achieved. The following steps are followed to get the best weak classifiers and further get a cascade of strong classifier.

i. The weights for the positive and negative images are initialized and normalized. If P and N are the number of positive and negative images respectively, the weights initialized are $\frac{1}{2P}$ and $\frac{1}{2N}$ respectively.

ii. All the training image samples are sorted based on each of the feature vectors. Then the minimum error is computed as below using the two polarities.

$$min\_error = min\ (polarity1, polarity\ 2)$$

$$polarity\ 1 = S^{+} + T^{-} - S^{-}$$

$$polarity\ 2 = S^{-} + T^{+} - S^{+}$$

where, T^{+} is the sum of total positive weights

T^- is the sum of total negative weights

S^+ is the sum of positive weights below the current threshold sample

S^- is the sum of negative weights below the current threshold sample

iii. Repeating the above for all feature vectors until the overall minimum error is obtained. The weak classifier that is best for the current iteration *(t)* can be given based on this minimum error given as $h_t(x)$ for features f_t, polarity p_t and threshold that minimized the error θ_t.

$$h_t(x) = h\,(x, f_t, p_t, \theta_t)$$

iv. Compute the confidence parameters (α, β) in each iteration and update the weights for the next iteration ($w_{t+1,\,i}$) to get the next weak classifier of the cascade.

$$\beta = \frac{\epsilon}{1-\epsilon} \quad and\ \alpha = ln\left(\frac{1}{\beta}\right),\ in\ each\ iteration\ t$$

$$w_{t+1,\,i} = w_{t,\,i} * \beta^{\,1-\delta} \quad where,\ \delta = 0\ if\ sample\ x_i\ is\ correctly\ classified,\ else\ \delta = 0$$

v. The criterion for weak classifier is checked i.e., check if the weak classifier FPR is 0.5 and the true detection rate is 1. This is checked by multiplying α in previous step with the weak classifier. If the criteria check is not satisfied, the steps above (ii-iv) are repeated to find the next classifier.

vi. Finally, build the strong classifier C(x) using the weak classifiers threshold as below. The threshold is the minimum value of the positive images in training which is set to get the true positive rate as 1.

$$C(x) = \begin{cases} 1, & \sum_{t=1}^{T} \alpha_t h_t(x) \ \geq threshold \\ 0, & otherwise \end{cases}$$

vii. Once the strong classifier is built, check if the cumulative FPR is zero. If it is not zero, select the negative images only that are misclassified along with all the positive images for building the classifier as stated in the above steps.

3.3 Testing the classifier

For each of the testing data images, classify the images using one of the strong classifiers. The images classified as positive are sent into the next cascade for classification while the negative predicted images are not forwarded. Repeat this process using the next strong classifier until all the data is classified with no negative predicted images remaining.

Determine the performance metrics False Positive Rate (FPR) and False Negative Rate (FNR) for evaluating the classifier. These plots can be seen in the results section of this report.

RESULTS & OBSERVATIONS:

TASK 1 & 2:

The accuracy of classification using PCA, LDA and Autoencoder for the p values is plotted in the graph below.

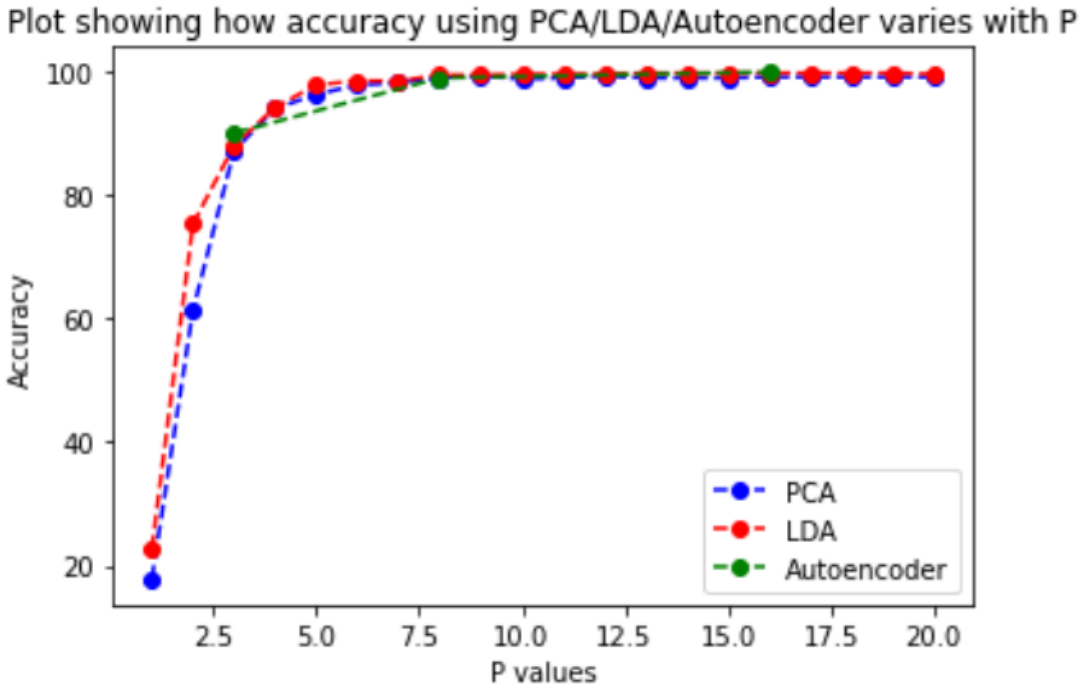

Figure 1: Plot between the accuracy obtained by PCA, LDA, Autoencoder for different p values.

Comparing PCA and LDA results:

Comparing the results of both PCA and LDA, we can observe that at every cascade LDA obtained higher accuracy than PCA. For example, at p =1, PCA obtained about 17.77% while LCA about 22.69% accuracy. Similarly, at every p we can observe the plotted points where accuracy is higher for LDA than PCA. Also, PCA reached a highest accuracy of 99.206 at p = 16 and stayed the same till p =20. However, LDA reached the highest accuracy of 99.841 at p = 13and stayed the same till p = 20.

Comparing PCA, LDA against Autoencoder results:

Comparing the plots of PCA/LDA with Autoencoder for p = [3, 8, 16] we can observe that at p= 3, autoencoder obtained highest accuracy of 90.157% as compared to PCA and LDA being <89%. In the same fashion, Autoencoder performed very well by attaining a 99% accuracy for p = 8 itself and later obtained a 100% accuracy for p =16 showing its dominance over PCA/LDA based image classification. Hence, looking at the results we can conclude that Autoencoder outperformed PCA and LDA results by achieving a 100% accuracy.

TASK 3:

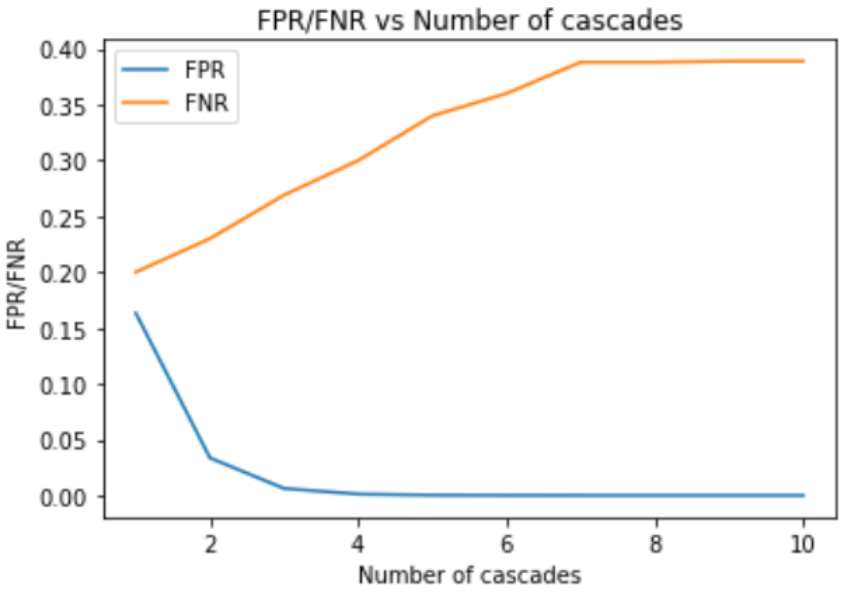

Figure 2: Plot showcasing how False Positive Rate (FPR) and False Negative Rate (FNR) vary with the different number of cascade values.

We can observe that with the decrease in FPR, the FNR increases which is the expected behavior expected by the AdaBoost classifier. This can be termed as the goal of decreasing FPR at the cost of increasing FNR which is how our classifier is designed to perform. This increase in FNR is due to the classifiers predicting the positive images as negative. An attempt to increase features to make the classifier learn better was made, but the memory storage and computational time taken in Jupyter notebook did not support the attempt, and hence only two Haar based kernels were used.

SOURCE CODE:

```
#!/usr/bin/env python
# coding: utf-8

# <h2><center>ECE661 COMPUTER VISION</center></h2>
# <h3><center>Homework - 10 </center></h3>
# <h3><center>Sahithi Kodali - 34789866</center></h3>
# <h3><center>kodali1@purdue.edu</center></h3>

# In[49]:


# Import libraries needed
```

```
import numpy as np
import matplotlib.pyplot as plt
import cv2
import math
import copy
import pandas as pd
import os
import PIL
get_ipython().run_line_magic('matplotlib', 'inline')


# In[50]:


#load images from the folder
def load_images_from_folder(path):
    images = []
    img_names = []

    #read image
    for img_name in os.listdir(path):
        img_names.append(img_name)
        img = cv2.imread(os.path.join(path + '\\' + img_name))
        images.append(img)

    return images, img_names


# ### TASK-1

# In[51]:


#The implementation of this task is based/inspired from 2020 hw1 solution
#Vectorize and normalizing the image
def vectorize_img(image):

    if (len(image.shape) > 2):
        image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

    vec_img = np.ravel(image)
    vec_norm_img = vec_img/np.linalg.norm(vec_img)

    return vec_norm_img
```

```
# In[52]:


#obtain feature vectors using PCA
def compute_PCA_vecs(vec_imgs, size, N, k ):

    #find global mean of image vectors
    imgs_mean = np.mean(vec_imgs, axis = 1)
    imgs_mean_reshaped = np.reshape(imgs_mean, (size,1))

    #find the image matrix X and orthogonal matrix X^T X
    X = vec_imgs - imgs_mean_reshaped
    XT_X = np.dot(np.transpose(X), X)

    #find eigen values, vectors and sort in descending order
    w, v = np.linalg.eig(XT_X)
    sorted_v = [vec for val, vec in sorted(zip(w,v), key = lambda wv: wv[0],
reverse = True)]

    #get the top k eigen vectors for the image
    eigVecs = np.zeros((size, N))
    for i in range(N):
        eigVec = np.dot(X, sorted_v[i])
        eigVecs[:, i] = eigVec/np.linalg.norm(eigVec)

    topk_eigVec = eigVecs[:, 0:k]

    return imgs_mean_reshaped, topk_eigVec


# In[53]:


#obtain feature vectors using LDA
def compute_LDA_vecs(vec_imgs, size, k, num_C, imgs_perC ):

    #find global mean of image vectors
    imgs_mean = np.mean(vec_imgs, axis = 1)
    imgs_mean_reshaped = np.reshape(imgs_mean, (size,1))

    #find class mean for each class images
    Cimgs_mean = np.zeros((size, num_C))
    Cw_diff = np.zeros((size, num_C * imgs_perC))

    for C in range(num_C):
```

```
        C_mean = np.mean( vec_imgs[:, C*imgs_perC : (C+1)*imgs_perC], axis = 1)
        Cimgs_mean[:, C] = C_mean

        C_mean_reshaped = np.reshape(C_mean, (size,1))

        Cw_diff[:, C*imgs_perC : (C+1)*imgs_perC] = vec_imgs[:, C*imgs_perC :
(C+1)*imgs_perC] - C_mean_reshaped

    #find the image matrix M and orthogonal matrix M^T M
    M = Cimgs_mean - imgs_mean_reshaped
    MT_M = np.dot(np.transpose(M), M)

    #find eigen values, vectors and sort in descending order
    w, v = np.linalg.eig(MT_M)
    sorted_w = sorted(w, reverse = True)
    sorted_v = [vec for val, vec in sorted(zip(w,v), key = lambda wv: wv[0],
reverse = True)]

    #get rid of the eigen vector that makes the matrix singular
    eigVals = np.zeros(num_C-1)
    eigVecs = np.zeros((size, num_C-1))

    for c in range(num_C-1):
        eigVec = np.dot(M, sorted_v[c])
        eigVecs[:, c] = eigVec/np.linalg.norm(eigVec)
        eigVals[c] = sorted_w[c]

    #find Z and compute the matrix using this for eigen vectors
    D = np.sqrt(np.linalg.inv(np.eye(num_C-1) * eigVals))
    Z = np.dot(eigVecs, D)
    ZT_X = np.dot(np.dot(np.transpose(Z), Cw_diff),
np.transpose(np.dot(np.transpose(Z), Cw_diff)))

    #find eigen values, vectors and sort in descending order for the new matrix
    w, v = np.linalg.eig(ZT_X)
    sorted_v = [vec for val, vec in sorted(zip(w,v), key = lambda wv: wv[0],
reverse = True)]

    #get the top k eigen vectors for the image
    eigVecs = np.zeros((size, C-1))
    for i in range(k):
        eigVec = np.dot(Z, sorted_v[i])
        eigVecs[:, i] = eigVec/np.linalg.norm(eigVec)

    topk_eigVec = eigVecs[:, 0:k]
```

```
    return imgs_mean_reshaped, topk_eigVec


# In[54]:


def label_NN(y_train, y_test, num_C, imgs_perC, k_NN):
    cls_nn = np.zeros(num_C)
    cls_dist = np.zeros(num_C, np.uint16)
    final_cls = np.arange(1, num_C + 1)

    #determine euclidean distance
    d = np.sqrt(np.sum((y_train - y_test) ** 2, axis = 0))

    #find the k nearest neighbours for given k
    for n in range(k_NN):
        idx_dmin = np.argmin(d)
        idx_nn = int(idx_dmin/imgs_perC)

        cls_nn[idx_nn] += 1
        cls_dist[idx_nn] += d[idx_nn]
        d[idx_dmin] = float('inf')

    #predict the class as the one with maximum possibility for least distance as
neighbour for given image
    max_cls_nn = np.max(cls_nn)
    cls_dist = cls_dist[cls_nn == max_cls_nn]/max_cls_nn
    final_cls = final_cls[cls_nn == max_cls_nn]

    cls_pred = final_cls[np.argmin(cls_dist)]

    return cls_pred


# In[55]:


#data paths
faces_path = "D:\Purdue\ECE661_CV\HW10\FaceRecognition"
faces_train_path = os.path.join( faces_path + "\\train")
faces_test_path = os.path.join( faces_path + "\\test")


# In[56]:
```

```
#load data
train_faces, train_f_names = load_images_from_folder(faces_train_path)
test_faces, test_f_names = load_images_from_folder(faces_test_path)


# In[57]:


train_faces[0].shape


# In[58]:


#initiate constant values
size = 128*128
num_C = 30
imgs_perC = 21
k_NN = 1
k = 20


# In[59]:


all_eigVals = list(np.arange(1, k+1))


# In[60]:


all_eigVals


# In[61]:


N = len(train_faces)
train_faces_vecs = np.zeros((size, N))


# In[62]:


#get vectorized images
for i in range(N):
    tr_img = train_faces[i]
```

```
    train_faces_vecs[:,i] = vectorize_img(tr_img)


# In[65]:


#find feature vectors and predict class
PCA_acc = []
# LDA_acc = []

for i in range(1, k+1):
    print(i)

    #get the feature vectors using PCA/LDA and compute the train feature vector

    mean_vecs, eig_kvecs = compute_PCA_vecs(train_faces_vecs, size, N, i)
#     mean_vecs, eig_kvecs = compute_LDA_vecs(train_faces_vecs, size, i, num_C,
imgs_perC)

    y_train = np.dot(np.transpose(eig_kvecs), train_faces_vecs - mean_vecs)

    correct_pred = 0

    for j in range(N):
        #get ground truth of image
        gt_class = int(test_f_names[j].strip().split('_')[0])

        #vectorize image and find the test feature vector
        test_img_vec = vectorize_img(test_faces[j])
        test_img_vec_reshaped = np.reshape(test_img_vec, (size,1))
        y_test = np.dot(np.transpose(eig_kvecs), test_img_vec_reshaped -
mean_vecs)

        #predict the class of the test image
        pred_class = label_NN(y_train, y_test, num_C, imgs_perC, k_NN)

        if(pred_class == gt_class):
            correct_pred  += 1

    acc = (correct_pred/N)*100
    print(acc)
    PCA_acc.append(acc)
#     LDA_acc.append(acc)
```

```
# In[66]:


PCA_acc


# In[67]:


LDA_acc


# ### Task - 2

# In[1]:


import os
import numpy as np
import torch
from torch import nn, optim
from PIL import Image
from torch.autograd import Variable
from torch.utils.data import Dataset, DataLoader
from torchvision import transforms


# In[70]:


def euc_dist(x_train, x_test):
    d = np.sqrt(np.sum((x_train - x_test)**2))
    return d


# In[71]:


from scipy.stats import mode

def AE_NN(X_train, X_test, y_train, k):
    y_pred = []

    for img_vec in X_test:

        dist =[]
```

```
        for i in range(len(X_train)):
            d = euc_dist(np.array(X_train[i, :]), img_vec)
            dist.append(d)

        dist = np.array(dist)
        sort_dist = np.argsort(dist)[:k]

        cls = y_train[sort_dist]

        pred = mode(cls)
        pred = pred.mode[0]
        y_pred.append(pred)

    return y_pred


# In[2]:


class DataBuilder(Dataset):
    def __init__(self, path):
        self.path = path
        self.image_list = [f for f in os.listdir(path) if f.endswith('.png')]
        self.label_list = [int(f.split('_')[0]) for f in self.image_list]
        self.len = len(self.image_list)
        self.aug = transforms.Compose([
            transforms.Resize((64, 64)),
            transforms.ToTensor(),
        ])

    def __getitem__(self, index):
        fn = os.path.join(self.path, self.image_list[index])
        x = Image.open(fn).convert('RGB')
        x = self.aug(x)
        return {'x': x, 'y': self.label_list[index]}

    def __len__(self):
        return self.len


class Autoencoder(nn.Module):

    def __init__(self, encoded_space_dim):
        super().__init__()
```

```
        self.encoded_space_dim = encoded_space_dim
        ### Convolutional section
        self.encoder_cnn = nn.Sequential(
            nn.Conv2d(3, 8, 3, stride=2, padding=1),
            nn.LeakyReLU(True),
            nn.Conv2d(8, 16, 3, stride=2, padding=1),
            nn.LeakyReLU(True),
            nn.Conv2d(16, 32, 3, stride=2, padding=1),
            nn.LeakyReLU(True),
            nn.Conv2d(32, 64, 3, stride=2, padding=1),
            nn.LeakyReLU(True)
        )
        ### Flatten layer
        self.flatten = nn.Flatten(start_dim=1)
        ### Linear section
        self.encoder_lin = nn.Sequential(
            nn.Linear(4 * 4 * 64, 128),
            nn.LeakyReLU(True),
            nn.Linear(128, encoded_space_dim * 2)
        )
        self.decoder_lin = nn.Sequential(
            nn.Linear(encoded_space_dim, 128),
            nn.LeakyReLU(True),
            nn.Linear(128, 4 * 4 * 64),
            nn.LeakyReLU(True)
        )
        self.unflatten = nn.Unflatten(dim=1,
                                      unflattened_size=(64, 4, 4))
        self.decoder_conv = nn.Sequential(
            nn.ConvTranspose2d(64, 32, 3, stride=2,
                               padding=1, output_padding=1),
            nn.BatchNorm2d(32),
            nn.LeakyReLU(True),
            nn.ConvTranspose2d(32, 16, 3, stride=2,
                               padding=1, output_padding=1),
            nn.BatchNorm2d(16),
            nn.LeakyReLU(True),
            nn.ConvTranspose2d(16, 8, 3, stride=2,
                               padding=1, output_padding=1),
            nn.BatchNorm2d(8),
            nn.LeakyReLU(True),
            nn.ConvTranspose2d(8, 3, 3, stride=2,
                               padding=1, output_padding=1)
        )
```

```
    def encode(self, x):
        x = self.encoder_cnn(x)
        x = self.flatten(x)
        x = self.encoder_lin(x)
        mu, logvar = x[:, :self.encoded_space_dim], x[:, self.encoded_space_dim:]
        return mu, logvar

    def decode(self, z):
        x = self.decoder_lin(z)
        x = self.unflatten(x)
        x = self.decoder_conv(x)
        x = torch.sigmoid(x)
        return x

    @staticmethod
    def reparameterize(mu, logvar):
        std = logvar.mul(0.5).exp_()
        eps = Variable(std.data.new(std.size()).normal_())
        return eps.mul(std).add_(mu)


class VaeLoss(nn.Module):
    def __init__(self):
        super(VaeLoss, self).__init__()
        self.mse_loss = nn.MSELoss(reduction="sum")

    def forward(self, xhat, x, mu, logvar):
        loss_MSE = self.mse_loss(xhat, x)
        loss_KLD = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp())
        return loss_MSE + loss_KLD


def train(epoch):
    model.train()
    train_loss = 0

    for batch_idx, data in enumerate(trainloader):
        optimizer.zero_grad()
        mu, logvar = model.encode(data['x'])
        z = model.reparameterize(mu, logvar)
        xhat = model.decode(z)
        loss = vae_loss(xhat, data['x'], mu, logvar)
        loss.backward()
        train_loss += loss.item()
        optimizer.step()
```

```
    print('====> Epoch: {} Average loss: {:.4f}'.format(
        epoch, train_loss / len(trainloader.dataset)))


# In[85]:


###################################
# Change these
p = 3  # [3, 8, 16]
training = False
TRAIN_DATA_PATH = 'D:/Purdue/ECE661_CV/HW10/FaceRecognition/train'
EVAL_DATA_PATH = 'D:/Purdue/ECE661_CV/HW10/FaceRecognition/test'
LOAD_PATH = f'D:/Purdue/ECE661_CV/HW10/weights/model_{p}.pt'
OUT_PATH = 'D:/Purdue/ECE661_CV/HW10/AE_ops'
###################################


# In[86]:


model = Autoencoder(p)

if training:
    epochs = 100
    log_interval = 1
    trainloader = DataLoader(
        dataset=DataBuilder(TRAIN_DATA_PATH),
        batch_size=12,
        shuffle=True,
    )
    optimizer = optim.Adam(model.parameters(), lr=1e-3)
    vae_loss = VaeLoss()
    for epoch in range(1, epochs + 1):
        train(epoch)
    torch.save(model.state_dict(), os.path.join(OUT_PATH, f'model_{p}.pt'))
else:
    trainloader = DataLoader(
        dataset=DataBuilder(TRAIN_DATA_PATH),
        batch_size=1,
    )
    model.load_state_dict(torch.load(LOAD_PATH))
    model.eval()

    X_train, y_train = [], []
```

```
    for batch_idx, data in enumerate(trainloader):
        mu, logvar = model.encode(data['x'])
        z = mu.detach().cpu().numpy().flatten()
        X_train.append(z)
        y_train.append(data['y'].item())
    X_train = np.stack(X_train)
    y_train = np.array(y_train)

    testloader = DataLoader(
        dataset=DataBuilder(EVAL_DATA_PATH),
        batch_size=1,
    )
    X_test, y_test = [], []
    for batch_idx, data in enumerate(testloader):
        mu, logvar = model.encode(data['x'])
        z = mu.detach().cpu().numpy().flatten()
        X_test.append(z)
        y_test.append(data['y'].item())
    X_test = np.stack(X_test)
    y_test = np.array(y_test)

#     print(X_train[5])
#     print(y_train.shape)
#     print(X_test.shape)
#     print(y_test.shape)
    ###################################
    # Your code starts here

    ###################################
    y_pred = AE_NN(X_train, X_test, y_train, 1)
    count = 0
    N = len(y_test)

    for i in range(N):
        if (y_test[i] == y_pred[i]):
            count += 1

    acc_3 = (count/N)*100
#     acc_8 = (count/N)*100
#     acc_16 = (count/N)*100


# In[89]:
```

```
acc_8


# In[92]:


#plot the acuuracy vs P values
plt.plot(all_eigVals, PCA_acc, linestyle='--', marker='o', color='blue',  label =
'PCA')
plt.plot(all_eigVals, LDA_acc, linestyle='--', marker='o', color='red', label =
'LDA')
plt.plot([3, 8, 16], [acc_3, acc_8, acc_16], linestyle='--', marker='o',
color='green', label = 'Autoencoder')

plt.legend()

plt.title('Plot showing how accuracy using PCA/LDA/Autoencoder varies with P')
plt.xlabel('P values')
plt.ylabel('Accuracy')
plt.show()

# ### TASK-3

#!/usr/bin/env python
# coding: utf-8

# In[1]:


# Import libraries needed

import numpy as np
import matplotlib.pyplot as plt
import cv2
import math
import copy
import pandas as pd
import os
import PIL
get_ipython().run_line_magic('matplotlib', 'inline')


# In[2]:
```

```
#All the functions below for implementing adaboost are based/inspired from 2018
pdf1 solution
#get the integral image for each image
def integral_img(img):
    ih, iw = img.shape
    ig_img = np.zeros((ih, iw))

    for j in range(1, ih+1):
        for i in range(1, iw+1):
            ig_img[j-1, i-1] = np.sum(img[:j,:i])
                return ig_img


# In[3]:


#build kernel 1

def haar1(img):
    ih, iw = img.shape
    kernels = []

    for w in range(1, int(iw/2)+1):
        z, o = np.zeros((1,w)), np.ones((1,w))
        ker = np.hstack((z,o))
        kernels.append(ker)

    return kernels


# In[4]:


#build kernel 2

def haar2(img):
    ih, iw = img.shape
    kernels = []

    for h in range(1, int(ih/2)+1):
        o, z = np.ones((h,2)), np.zeros((h,2))
        ker = np.vstack((o,z))
        kernels.append(ker)

    return kernels
```

```
# In[5]:


#compute features using kernel 1
def comp_feat1(ig_img, kernel):

    feats = []
    ih, iw = ig_img.shape

    for k in kernel:
        kh, kw = k.shape

        for j in range(-1, ih - kh):
            for i in range(-1, iw - kw):
                lx0, ly0, rx0, ry0 = i, j, i + int(kw/2), j + kh
                lx1, ly1, rx1, ry1 = i + int(kw/2), j, i + kw, j + kh

                A = ig_img[ly0, lx0] if ly0 > -1 and lx0 > -1 else 0
                B = ig_img[ly1, lx1] if ly1 > -1 and lx1 > -1 else 0
                C = ig_img[ry0, lx0] if ry0 > -1 and lx0 > -1 else 0
                D = ig_img[ry0, rx0] if ry0 > -1 and rx0 > -1 else 0
                E = ig_img[ly1, rx1] if ly1 > -1 and rx1 > -1 else 0
                F = ig_img[ry1, rx1] if ry1 > -1 and rx1 > -1 else 0

                f = -A + 2*B + C - 2*D - E + F

                feats.append(f)
    return feats


# In[6]:


#compute features using kernel 2
def comp_feat2(ig_img, kernel):

    feats = []
    ih, iw = ig_img.shape

    for k in kernel:
        kh, kw = k.shape

        for j in range(-1, ih - kh):
            for i in range(-1, iw - kw):
                lx1, ly1, rx1, ry1 = i, j, i + kw, j + int(kh/2)
```

```
                lx0, ly0, rx0, ry0 = i, j + int(kh/2), i + kw, j + kh

                A = ig_img[ly1, lx1] if ly1 > -1 and lx1 > -1 else 0
                B = ig_img[ly1, rx1] if ly1 > -1 and rx1 > -1 else 0
                C = ig_img[ly0, lx0] if ly0 > -1 and lx0 > -1 else 0
                D = ig_img[ry1, rx1] if ry1 > -1 and rx1 > -1 else 0
                E = ig_img[ry0, lx0] if ry0 > -1 and lx0 > -1 else 0
                F = ig_img[ry0, rx0] if ry0 > -1 and rx0 > -1 else 0

                f = A - B - 2*C + 2*D + E - F

                feats.append(f)
    return feats


# In[7]:


#get the best weak classifiers
def find_best_wc(train_f, train_l, normW, num_pos):

    #get the features and labels
    feats, tot = train_f.shape
    num_neg = tot - num_pos

    labels = np.asarray(train_l, dtype = int)

    #initiate variables
    best_wc = []
    best_cls = np.zeros(tot)
    best_inc = np.zeros(tot, dtype = int)
    best_err = math.inf

    #get total positive and negative images weights
    TP = np.sum(normW[:num_pos])
    TN = np.sum(normW[num_pos:])

    for f in range(feats):
        feats_all = train_f[f].tolist()
        ini_idx = list(range(tot))
        gt_labels = labels.tolist()
        normW_lst = normW.tolist()

        feats_all, ini_idx, gt_labels, normW_lst = zip(*sorted(zip(feats_all,
ini_idx, gt_labels, normW_lst), key = lambda xyz: xyz[0]))
```

```
    feats_all = np.array(feats_all)
    ini_idx = np.array(ini_idx)
    gt_labels = np.array(gt_labels)
    normW_lst = np.array(normW_lst)

    #get cumulative sum of postive and negitive weights
    SP = np.cumsum(normW_lst * gt_labels)
    SN = np.cumsum(normW_lst) - SP

    pol1 = SP + (TN - SN)
    pol2 = SN + (TP - SP)

    min_err = np.minimum(pol1, pol2)
    min_err_idx = np.argmin(min_err)

    if pol1[min_err_idx] <= pol2[min_err_idx]:
        pol = 1

        cls = train_f[f] >= feats_all[min_err_idx]
        cls = np.asarray(cls, dtype = int)

        inc = np.asarray(cls != labels, dtype = int)
        sum_inc = int(np.sum(inc))

        wc = [f, feats_all[min_err_idx], pol, pol1[min_err_idx], sum_inc]

    else:
        pol = -1

        cls = train_f[f] < feats_all[min_err_idx]
        cls = np.asarray(cls, dtype = int)

        inc = np.asarray(cls != labels, dtype = int)
        sum_inc = int(np.sum(inc))

        wc = [f, feats_all[min_err_idx], pol, pol2[min_err_idx], sum_inc]

    #choose the best weak classifier
    if min_err[min_err_idx] < best_err:
        best_wc = wc
        best_cls = cls
        best_inc = inc
        best_err = min_err[min_err_idx]
```

```
    return best_wc, best_cls, best_inc



# In[26]:


#build cascade of strong classifiers from the best weak classifiers obtained
def build_cascade(train_f, train_l, num_pos, casc_id, FPR_thresh, TPR_thresh, T,
cascades):
    feats, tot = train_f.shape
    num_neg = tot - num_pos

    #initialize weights
    w_pos = np.ones(num_pos)/ (2*num_pos)
    w_neg = np.ones(num_neg)/ (2*num_neg)

    w_norm = np.hstack((w_pos, w_neg))/1.0

    best_wc_casc = []
    alphas = []
    lst_hx = []
    FPR_lst = []
    TPR_lst = []

    for t in range(T):
        print(t)
        w_norm = w_norm/np.sum(w_norm)

        best_wc, best_cls, best_inc = find_best_wc(train_f, train_l, w_norm,
num_pos)
        best_wc_casc.append(best_wc)

        #calculate the confidence params
        eps = best_wc[3]

        beta = eps/(1 - eps + 0.00001)
        alpha = math.log(1/(beta + 0.00001))
        w_norm = w_norm * np.power(beta, 1- best_inc)

        alphas.append(alpha)
        lst_hx.append(best_cls)

        arr_alphas = np.array([alphas]).T
        arr_lst_hx = np.array(lst_hx).T
```

```
        temp = np.matmul(arr_lst_hx, arr_alphas)
        alpha_thresh = np.min(temp[:num_pos])

        casc = temp >= alpha_thresh
        casc = np.asarray(casc, dtype = int)

        #compute FPR, FNR
        FPR = np.sum(casc[num_pos:])/num_neg
        TNR = 1-FPR

        TPR = np.sum(casc[:num_pos])/num_pos
        FNR = 1-TPR

        print(FPR)
        print(TPR)

        FPR_lst.append(FPR)
        TPR_lst.append(TPR)

        #check for criteria
        if TPR >= TPR_thresh and FPR <= FPR_thresh:
            break

    #update params
    new_feats = train_f[:, :num_pos]

    for n in range(num_neg):
        neg_idx = n + num_pos

        if casc[neg_idx] > 0:
            inc_neg = train_f[:, neg_idx]
            inc_neg = np.expand_dims(inc_neg, axis = 1)
            new_feats = np.hstack((new_feats, inc_neg))

    rem_neg = new_feats.shape[1] - num_pos
    new_num_neg = num_neg - rem_neg

    new_labels = np.ones(num_pos + rem_neg)
    new_labels[num_pos:] = 0

    casc_dict = copy.deepcopy(cascades)
    casc_dict[casc_id] = { 'T':t+1, 'tpr': TPR, 'fpr' : FPR, 'neg_red':
new_num_neg, 'neg_rem':rem_neg, 'alphas':alphas, 'best_wc_cls': best_wc_casc }
```

```
    return casc_dict, new_feats, new_labels


# In[30]:


#testing cascade using the testing features and the cascade of strong classifiers
def test_cascade(test_f, cascade):

    T = cascade['T']
    TPR, FPR = cascade['tpr'], cascade['fpr']
    neg_red = cascade['neg_red']
    neg_rem = cascade['neg_rem']
    alphas = cascade['alphas']
    best_wc_cls = cascade['best_wc_cls']

    feats, tot = test_f.shape

    lst_hx = []

    for t in range(T):
        f_id = best_wc_cls[t][0]
        thresh = best_wc_cls[t][1]
        pol = best_wc_cls[t][2]

        f_all = test_f[f_id]
        if pol == 1:
            cls = np.asarray(f_all >= thresh, dtype = int)
        else:
            cls = np.asarray(f_all < thresh, dtype = int)

        lst_hx.append(cls)

    alphas_arr = np.array([alphas]).T
    hx_arr = np.array(lst_hx).T

    temp = np.matmul(hx_arr, alphas_arr)
    alpha_thresh = np.sum(alphas_arr) * 0.5

    casc = temp >= alpha_thresh
    casc = np.asarray(casc, dtype=int)

    return casc
```

```
# In[9]:


#load images from the folder
def load_images(path):
    images = []
    img_names = []

    #read image
    for img_name in os.listdir(path):
        img_names.append(img_name)
        img = cv2.imread(os.path.join(path + '\\' + img_name))
        img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
        images.append(img)

    return images, img_names


# In[10]:


#data paths
cars_path = "D:\Purdue\ECE661_CV\HW10\CarDetection"

cars_tr_pos_path = os.path.join(cars_path + "\\train\\positive")
cars_tr_neg_path = os.path.join(cars_path + "\\train\\negative")

cars_tt_pos_path = os.path.join(cars_path + "\\test\\positive")
cars_tt_neg_path = os.path.join(cars_path + "\\test\\negative")


# In[11]:


#load data
train_pos_cars, train_carp_names = load_images(cars_tr_pos_path)
train_neg_cars, train_carn_names = load_images(cars_tr_neg_path)

test_pos_cars, test_carp_names = load_images(cars_tt_pos_path)
test_neg_cars, test_carn_names = load_images(cars_tt_neg_path)


# In[12]:


#get feature vectors for train positive images
```

```
for idx, img in enumerate(train_pos_cars):
    ig_img = integral_img(img)
    k1 = haar1(img)
    k2 = haar2(img)

    feats1 = comp_feat1(ig_img, k1)
    feats2 = comp_feat2(ig_img, k2)

    feats = np.array(feats1 + feats2)
    feats = np.expand_dims(feats, axis = 1)

    train_f_pos = feats if idx == 0 else np.hstack((train_f_pos, feats))

train_l_pos = np.ones((len(train_pos_cars)))

PATH = 'D:/Purdue/ECE661_CV/HW10/Outputs/'
np.savez( os.path.join(PATH,'feats_tr_pos.npz'), train_f_pos, train_l_pos)


# In[13]:


#get feature vectors for train negative images

for idx, img in enumerate(train_neg_cars):
    ig_img = integral_img(img)
    k1 = haar1(img)
    k2 = haar2(img)

    feats1 = comp_feat1(ig_img, k1)
    feats2 = comp_feat2(ig_img, k2)

    feats = np.array(feats1 + feats2)
    feats = np.expand_dims(feats, axis = 1)

    train_f_neg = feats if idx == 0 else np.hstack((train_f_neg, feats))

train_l_neg = np.ones((len(train_neg_cars)))

PATH = 'D:/Purdue/ECE661_CV/HW10/Outputs/'
np.savez( os.path.join(PATH,'feats_tr_neg.npz'), train_f_neg, train_l_neg)


# In[14]:
#get feature vectors for test positive images
```

```
for idx, img in enumerate(test_pos_cars):
    ig_img = integral_img(img)
    k1 = haar1(img)
    k2 = haar2(img)

    feats1 = comp_feat1(ig_img, k1)
    feats2 = comp_feat2(ig_img, k2)

    feats = np.array(feats1 + feats2)
    feats = np.expand_dims(feats, axis = 1)

    test_f_pos = feats if idx == 0 else np.hstack((test_f_pos, feats))

test_l_pos = np.ones((len(test_pos_cars)))

PATH = 'D:/Purdue/ECE661_CV/HW10/Outputs/'
np.savez( os.path.join(PATH,'feats_tt_pos.npz'), test_f_pos, test_l_pos)


# In[15]:


#get feature vectors for test negative images
for idx, img in enumerate(test_neg_cars):
    ig_img = integral_img(img)
    k1 = haar1(img)
    k2 = haar2(img)

    feats1 = comp_feat1(ig_img, k1)
    feats2 = comp_feat2(ig_img, k2)

    feats = np.array(feats1 + feats2)
    feats = np.expand_dims(feats, axis = 1)

    test_f_neg = feats if idx == 0 else np.hstack((test_f_neg, feats))

test_l_neg = np.ones((len(test_neg_cars)))

PATH = 'D:/Purdue/ECE661_CV/HW10/Outputs/'
np.savez( os.path.join(PATH,'feats_tt_neg.npz'), test_f_neg, test_l_neg)


# In[16]:
#training
#load the train features
```

```
PATH = 'D:/Purdue/ECE661_CV/HW10/Outputs/'
tr_pos = np.load( os.path.join(PATH,'feats_tr_pos.npz'))
tr_f_pos, tr_l_pos = tr_pos['arr_0'], tr_pos['arr_1']

tr_neg = np.load( os.path.join(PATH,'feats_tr_neg.npz'))
tr_f_neg, tr_l_neg = tr_neg['arr_0'], tr_neg['arr_1']

num_pos, num_neg = tr_l_pos.shape[0], tr_l_neg.shape[0]
num_feats, tot = tr_f_pos.shape[0], num_pos + num_neg

train_f = np.hstack((tr_f_pos, tr_f_neg ))
train_l = np.hstack((tr_l_pos, tr_l_neg ))


# In[29]:

#initilize variables for number of weak classifiers and th estrong classifier
cascades
#initlize FPR and TPR targets and thresholds
T = 100
S = 10

FPR_target = 0.000001
TPR_target = 1

FPR_thresh = 0.5
TPR_thresh = 1

FPR = 1
TPR = 0

all_FPR = []
all_TPR = []
casc_idx = []

casc_dict = {}

for c_id in range(1, S+1):
    c_dict, new_feats, new_labels = build_cascade(train_f, train_l, num_pos,
c_id, FPR_thresh, TPR_thresh, T, casc_dict)

    casc_dict = copy.deepcopy(c_dict)
    train_f = copy.deepcopy(new_feats)
    train_l = copy.deepcopy(new_labels)
```

```
    rem_neg = casc_dict[c_id]['neg_rem']
    red_neg = casc_dict[c_id]['neg_red']
    tpr, fpr = casc_dict[c_id]['tpr'], casc_dict[c_id]['fpr']

    FPR *= fpr
    TPR *= tpr

    all_FPR.append(FPR)
    all_TPR.append(TPR)
    casc_idx.append(c_id)

    if(TPR >= TPR_target and FPR <= FPR_target) or rem_neg == 0:
        break


# In[37]:


#testing
#load the test features

PATH = 'D:/Purdue/ECE661_CV/HW10/Outputs/'
tt_pos = np.load( os.path.join(PATH,'feats_tt_pos.npz'))
tt_f_pos, tt_l_pos = tt_pos['arr_0'], tt_pos['arr_1']

tt_neg = np.load( os.path.join(PATH,'feats_tt_neg.npz'))
tt_f_neg, tt_l_neg = tt_neg['arr_0'], tt_neg['arr_1']

#initilize features and labels
num_pos, num_neg = tt_l_pos.shape[0], tt_l_neg.shape[0]
test_f = np.hstack((tt_f_pos, tt_f_neg ))

all_FPR = []
all_FNR = []
num_FP, num_FN = 0, 0

#compute FPR and FNR using cascade for each testing image
for id, (k, casc) in enumerate(casc_dict.items()):
    C = test_cascade(test_f, casc)

    #get False negatives
    curr_pos = 0
    for p in range(num_pos):
```

```
        if C[p, 0] == 1:
            pos_ex = test_f[:, p]
            pos_ex = np.expand_dims(pos_ex, axis = 1)
            new_test_f = np.hstack((new_test_f, pos_ex)) if p>0 else pos_ex

            curr_pos += 1

    num_FN += (num_pos - curr_pos)

    #get False Positives
    curr_neg = 0
    for n in range(num_neg):
        label = num_pos + n
        if C[label, 0] == 1:
            neg_ex = test_f[:, label]
            neg_ex = np.expand_dims(neg_ex, axis = 1)
            new_test_f = np.hstack((new_test_f, neg_ex))

            curr_neg += 1

    num_FP = curr_neg

    #get FPR and FNR list
    all_FPR.append(num_FP/num_neg)
    all_FNR.append(num_FN/num_pos)

    if num_pos == 0:
        break

# In[38]:


#plot for FPR and FNR wrt cascade
casc_idx = list(casc_dict)
plt.plot(casc_idx, all_FPR, label = 'FPR')
plt.plot(casc_idx, all_FNR, label = 'FNR')

plt.title('FPR/FNR vs Number of cascades')
plt.xlabel('Number of cascades')
plt.ylabel('FPR/FNR')
plt.legend()
plt.show()
```

*****