ECE661 Fall 2024: Homework 10

By – Aayush Bhadani

abhadani@purdue.edu

Theory Questions:

1. Overfitting to the training data refers to a situation where a machine learning model learns not only the underlying patterns and relationships in the data but also the noise and fluctuations that are specific to the training set. As a result, the model becomes highly specialized to the training data and performs exceptionally well on it but struggles to generalize to new, unseen data from the test set. In other words, the model memorizes the details of the training data rather than learning generalizable features.

Overfitting can occur when the model is too complex relative to the amount of training data available. A highly complex model with many parameters may fit the training data very closely, even capturing patterns that are irrelevant or random. While this might improve the model's performance on the training set, it leads to poor performance on new data because the model fails to recognize the broader patterns that apply in different contexts.

To prevent overfitting, techniques such as cross-validation, early stopping, regularization, dropout, and using a simpler model or more data are employed. Regularization methods like L1/L2 regularization help to penalize overly complex models and encourage learning more generalizable features. The goal is to find a model that balances the ability to fit the training data with the ability to generalize well to new, unseen examples.

2. The Reparameterization Trick is a technique used in Variational Autoencoders (VAEs) to make the stochastic sampling process differentiable, enabling backpropagation through the network during training. Instead of directly sampling from the latent distribution $N(\mu, \sigma^2)$, the trick expresses the latent variable as $z = \mu + \sigma \cdot \epsilon$, where μ and σ are the mean and standard deviation produced by the encoder, and ϵ is random noise sampled from a standard normal distribution. This allows the model to sample from the latent space in a differentiable manner by reparameterizing the random sampling as a deterministic function of the learned parameters and noise. By doing so, the VAE can backpropagate gradients through both the encoder and the decoder, facilitating the efficient optimization of the network.

Tasks:

PCA:

Principal Component Analysis (PCA) is a statistical method used for dimensionality reduction. It transforms high-dimensional data into a smaller number of dimensions while preserving as much variance (information) as possible. It achieves this by identifying new orthogonal axes, called principal components, which are linear combinations of the original features. The first principal component captures the direction of greatest variance, the second captures the second greatest variance, and so on. By projecting the data onto these new axes, PCA simplifies complex datasets, making them easier to analyze, visualize, and interpret.

Steps:

- First read the image data and transform each image into vector.
- Next, normalize/standardize the data by making the image vectors unit magnitude. This is important as PCA is sensitive to scales.
- Compute mean vector of all N normalized image data. It is given as,

$$\vec{m} = \frac{1}{N}\sum_{i=1}^{N} x_i$$

  Subtract this mean vector from all image vector data.

- Compute the covariance matrix which is given as,

$$C = \frac{1}{N}\sum_{i=1}^{N} x_i\, x_i^T$$

  Covariance matrix captures how the features in the data vary with respect to each other.

- Compute Eigen-values and Eigen-vectors of this covariance matrix.
  We use computational trick as image data can lead to very large covariance matrix. It is important to normalize the eigen-vectors obtained using this trick.
- Pick top k eigenvectors corresponding to largest eigenvalues.
  Eigenvectors with the highest eigenvalues are the most significant because they capture the most variance in the data.
- These top k eigenvectors form the lower dimension subspace on which we will project our data to reduce its dimension. Taking the projection is just a matrix multiplication operation between data matrix and matrix of selected eigenvectors. We transform both our train and test data to this lower dimension space.
- For classification we use 1-nearest neighbour approach, where we use the least distance (L2 norm) of projected test vector to projected train data points for assigning class label.
- Finally, the accuracy is calculated based on the comparison of predicted labels for the test data points and their ground truth.

LDA:

Linear Discriminant Analysis (LDA) is another powerful technique used for dimensionality reduction, particularly in classification problems. Unlike Principal Component Analysis (PCA), which focuses on preserving the overall variance in the data, LDA seeks to reduce dimensions while maximizing the separability between different classes. The method works by finding a linear combination of features that best discriminates between the classes. It does so by computing the within-class and between-class scatter matrices, then solving for the eigenvectors and eigenvalues that represent the directions with the most significant class separability. By projecting the data onto a subspace formed by the most important eigenvectors, LDA reduces the dimensionality while retaining the essential discriminative information.

Steps:

- Just like in PCA, we take the image data and process it.
- We define between-class scatter as,

$$S_b = \frac{1}{|C|}\sum_{i=1}^{|C|}(\vec{m}_i - \vec{m})(\vec{m}_i - \vec{m})^T$$

  Here, $|C|$ is the number of classes, $\vec{m}_i$ is mean of i^{th} class and $\vec{m}$ is the global mean.

- Within-class scatter is defined as,

$$S_w = \frac{1}{|C|}\sum_{i=1}^{|C|}\frac{1}{|C_i|}\sum_{k=1}^{|C_i|}\left(\vec{x}^i{}_k - \vec{m}_i\right)\left(\vec{x}^i{}_k - \vec{m}_i\right)^T$$

- The goal is to maximise the between-class variance and minimize within-class variance. This is captured by Fisher Discriminant Function which needs to be maximized and is given by,

$$J(\vec{w}) = \frac{\vec{w}^T S_b \vec{w}}{\vec{w}^T S_w \vec{w}}$$

  It must satisfy, $S_b\vec{w} = \lambda S_w\vec{w}$. Thus, it becomes a generalized eigenvalue problem.

- Eigenvectors are found using Yu-Yang approach as given in the paper 'A Direct LDA Algorithm for High-Dimensional Data–with Application to FaceRecognition'. Since we are dealing with image data which has large number of dimensions, we use the same computational trick for finding eigenvalues and their corresponding eigenvectors.
- After we obtain the eigenvectors which forms the lower dimensional space, we project our data onto it and do classification using 1-nearest neighbour. The approach here is similar to the one used in PCA.

AutoEncoder:

An autoencoder is a type of neural network which can be used dimensionality reduction and feature extraction by learning a compressed representation of input data. It comprises two components: an encoder, which reduces the input data to a lower-dimensional latent space, and a decoder, which reconstructs the original data from this compressed representation. The network is trained to minimize the reconstruction error, ensuring the encoded representation captures the most important features while eliminating redundant or irrelevant information. Unlike traditional methods like PCA, autoencoders can model complex, non-linear relationships, making them particularly effective for high-dimensional data such as images.

Once the autoencoder is trained, we can use encoder part to generate lower dimensional embeddings of the data. Here we have tried P = 3,8,16. Similar approach for classification has been used as the one described above.

Cascaded AdaBoost:

Cascaded AdaBoost, as introduced in the Viola-Jones framework, combines AdaBoost's ability to create strong classifiers from weak ones with a cascaded structure for efficiency. The cascade consists of sequential stages, each designed to reject negatives early while passing potential positives to subsequent stages for further scrutiny. AdaBoost boosts weak classifiers by iteratively focusing on harder examples, ensuring high accuracy. Early stages are simple and quick, rejecting the majority of negatives, while later stages are more complex, targeting challenging cases.

Steps:

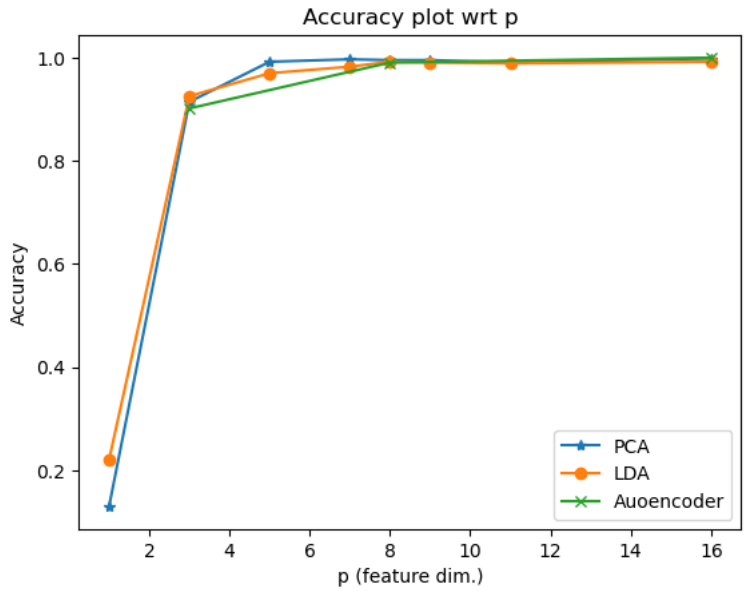
- Process the image data to compute integral images which will be useful for efficiently extracting Haar features. Next, extract features using Haar filters – horizontal and vertical type of various window sizes.
- Initialize the weights as $\frac{1}{2l}, \frac{1}{2m}$ for $y_i = 0,1$ resp. Here 'm' are number of negative examples and 'l' are number of positive examples.
- Create weak classifier:
    - Normalize the weights
    - Sort the features in ascending order of the values. Next, use feature values as thresholds.
    - Compute errors for different thresholds and select the one with least error.
    - Save the parameters (feature_index, threshold, polarity, error) for the feature with minimum error. After iteratively searching the feature with least error, get the best weak classifier.
- Strong weak-classifer:
    - For each best weak classifier obtained, compute:

$$\beta = \frac{\epsilon}{1-\epsilon} \quad ; \quad \epsilon \; is \; error$$
$$\alpha = \ln(1/\beta)$$

    - Update the weight by multiplying $\beta^{1-(label-pred)}$
    - Compute prediction for best weak classifier in a stage by multiplying predictions with α. Save α with classifier parameters as well.
    - Get the final classification using the α.
    - Keep track pf performance metrics like TPR,FPR,TNR,FNR. Stop when required metric condition is met.
    - Finally remove TN examples from the dataset at the end of the stage.
- Cascaded stages:
    - Repeat the stages in cascaded manner to get to a low FPR with satisfactory accuracy. Training can be stopped early, if the set desired performance conditions are met.
- Test:
    - Pass test data examples through the cascaded framework in a similar fashion as done in training to obtain outputs by thresholding.
      Remove examples accordingly from the dataset with each stage.
    - Compute:

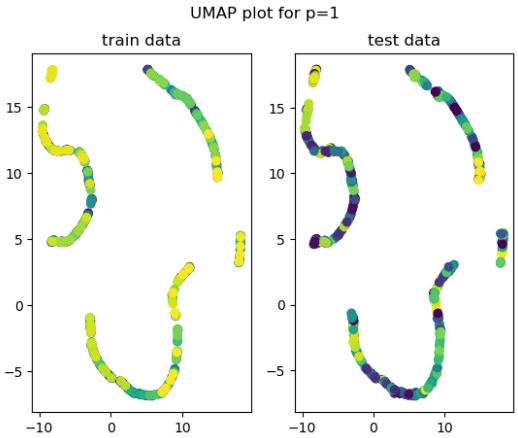
$$FPR = \frac{\#of \; misclassified \; negative \; image}{\#of \; negative \; images}$$
$$FNR = \frac{\#of \; misclassified \; positve \; image}{\#of \; positive \; images}$$

Results:

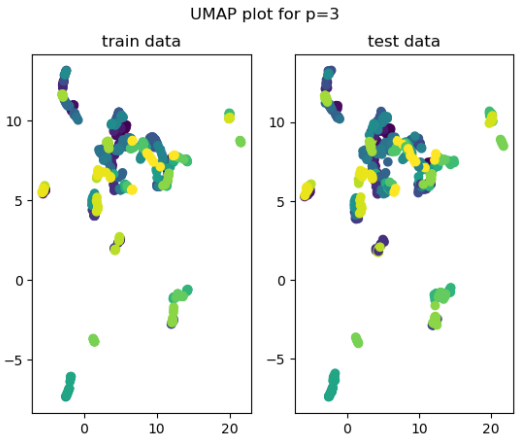


Accuracies and UMAP for PCA approach:

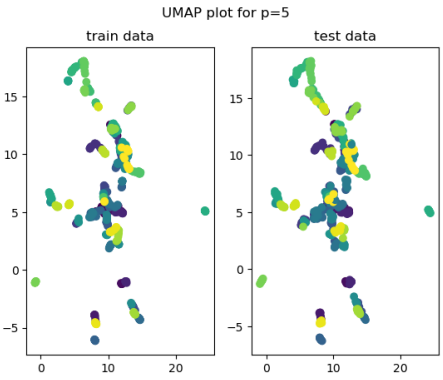
Accuracy with p=1 : 0.13015873015873017



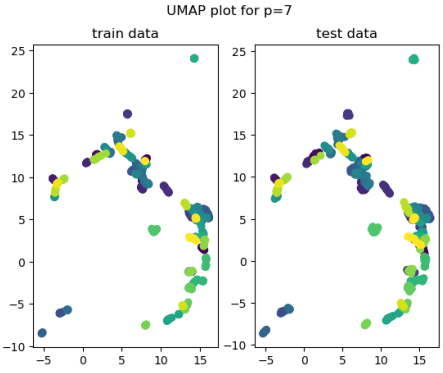
Accuracy with p=3 : 0.9142857142857143



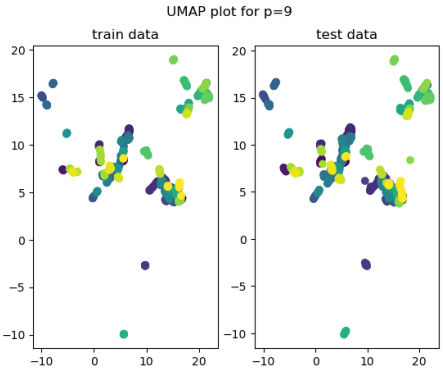
Accuracy with p=5 : 0.9920634920634921



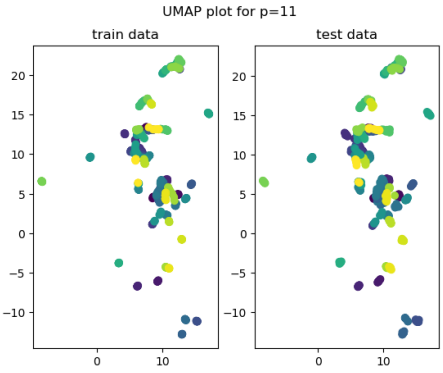
Accuracy with p=7 : 0.9968253968253968



Accuracy with p=9 : 0.9952380952380953

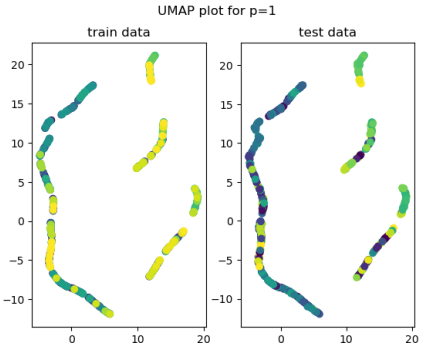


Accuracy with p=11 : 0.9904761904761905

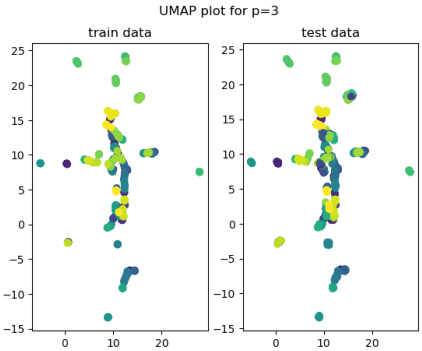


Accuracies and UMAP for LDA approach:

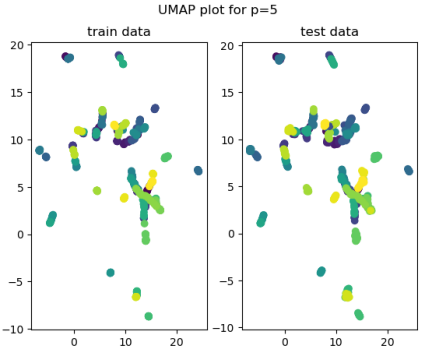
Accuracy with p=1 : 0.22063492063492063



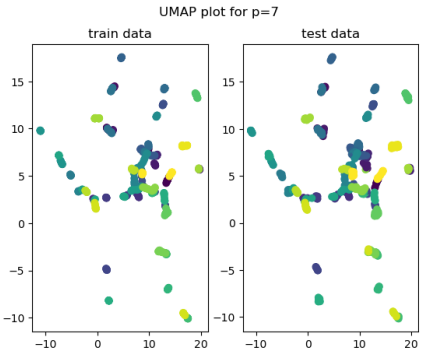
Accuracy with p=3 : 0.9253968253968254



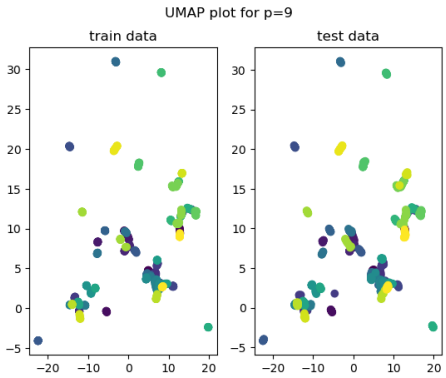
Accuracy with p=5 : 0.9698412698412698



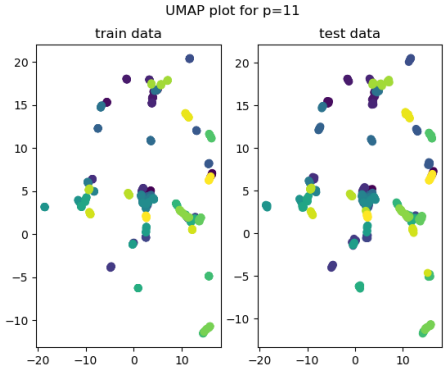
Accuracy with p=7 : 0.9825396825396825



Accuracy with p=9 : 0.9904761904761905



Accuracy with p=11 : 0.9888888888888889



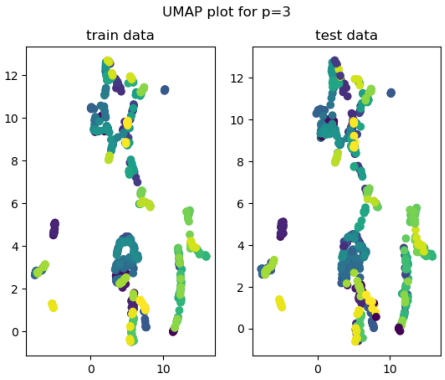
Discussion:

Both PCA and LDA are effective in reducing dimensionality while maintaining good accuracy, as seen from the accuracy plot with respect to p. Although LDA performed better in low dimension, their performances are quite similar in high dimension space with PCA being slightly better.

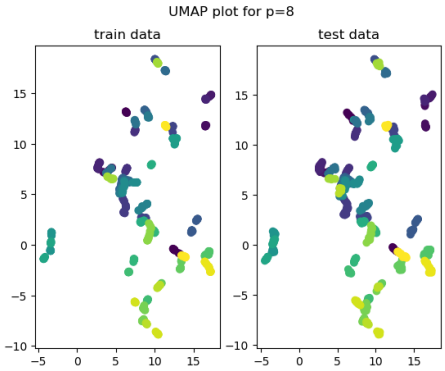
Observing the distinct patterns in the UMAPs for PCA and LDA, one can say that the two methods differently project the data in the reduced dimensional space. In the lower dimensions, it is difficult to distinguish between classes represented by different colors and thus accuracy is also low. In higher dimensions however, we can start to make out different classes which are colored differently and so the accuracy is also high.

Accuracies and UMAP for Autoencoder approach:

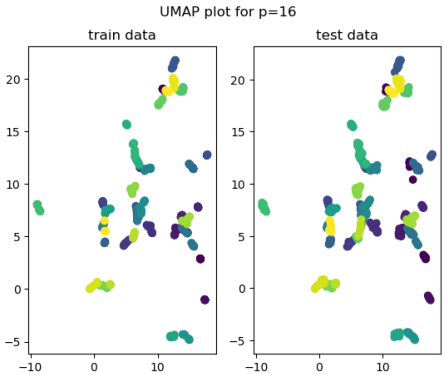
Accuracy with p=3 : 0.9015873015873016



Accuracy with p=8 : 0.9904761904761905



Accuracy with p=16 : 1.0



Discussion:

The performance of autoencoder based approach was as effective as PCA and LDA. It also resulted in the best accuracy score overall. However, in low dimension it lagged slightly behind.

We have distinct UMAP projection patterns compared to PCA and LDA. Although similar to the other two approaches we can start to distinguish different classes (and thus high accuracy) with higher dimensions.

```
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


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,trainloader,optimizer,vae_loss):
    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)))


##################################
```

```
# Change these
# p = 3  # [3, 8, 16]
def autoencoder_model(p):
    training = False
    TRAIN_DATA_PATH = 'FaceRecognition/train'#'path/to/train/set'
    EVAL_DATA_PATH = 'FaceRecognition/test'#'path/to/test/set'
    LOAD_PATH = f'weights\model_{p}.pt'#f'path/to/model_{p}.pt'
    OUT_PATH = 'autoencoder'#'path/to/exp'
    ###################################

    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,model,trainloader,optimizer,vae_loss)
        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)

        ###################################
        # Your code starts here
        return X_train,y_train,X_test,y_test
        ###################################
```

```
# %%
import numpy as np
import matplotlib.pyplot as plt
import cv2 as cv
import os
import umap
import autoencoder


# %%
faceRec_dataPath = 'FaceRecognition'
carDet_dataPath = 'CarDetection'

# %%
def load_data(root,dirName):
    data_path = os.path.join(root,dirName)
    data_dir = sorted(os.listdir(data_path))
    img_data = []
    labels = []
    for img_name in data_dir:
        img_path = os.path.join(data_path,img_name)
        img = cv.imread(img_path)
        labels.append(int(img_name.split('_')[0]))
        img_data.append(img)

    return np.array(labels), np.array(img_data)


# %%
faceRec_labels,faceRec_img_data = load_data(faceRec_dataPath,'train')

# %%
def PCA(X,p):
    # X = [x.flatten() for x in X]
    # X = np.array(X)
    # X = X/(np.linalg.norm(X,axis=1,keepdims=True))
    X_m = np.mean(X,axis=0)
    X = X - X_m
    X = X.T
    cov_mat = (X.T)@X
    u,d,v = np.linalg.svd(cov_mat)
    W = X@v
    W = W/(np.linalg.norm(W,axis=0,keepdims=True))
    return W[:,:p],X_m



# %%

# aaa = PCA(faceRec_img_data,p=1)

# %%
def LDA(X,Y,p):
    # X = [x.flatten() for x in X]
    # X = np.array(X)
    # X = X/(np.linalg.norm(X,axis=1,keepdims=True))
    X_m = np.mean(X,axis=0)
    X = X - X_m
    num_classes = len(np.unique(Y))
    class_means = []
    data_mean_diff = []
    for label in np.unique(Y):
        X_c = X[Y==label]
        class_mean = np.mean(X_c,axis=0)
        class_means.append(class_mean)
        X[Y==label] = (X_c - class_mean)
    # X -= X_m
    class_means = np.array(class_means)
    mean_vec_diff = (class_means-X_m).T
    S_B = mean_vec_diff.T@(mean_vec_diff)
    u,d,v = np.linalg.svd(S_B)
    W = (mean_vec_diff)@v
    W = W/(np.linalg.norm(W,axis=0,keepdims=True))
    D_B = d[:-1]
    Y = W[:,:-1]
```

```
    Z = Y@(np.sqrt(np.linalg.inv(np.diag(D_B))))
    zt_Sw_z = ((X@Z).T)@(X@Z) #(Z.T)@X.T@X@Z
    u,d,v = np.linalg.svd(zt_Sw_z)
    # v = ((X@Z))@vt  #
    # v = v/(np.linalg.norm(v,axis=0,keepdims=True))  #
    # print((X@Z).shape,v.shape,Z.shape)
    W = Z@v[:,1:]#Z@v[:,1:]
    W = W/(np.linalg.norm(W,axis=0,keepdims=True))
    # print(W[:,-p:].shape)
    return W[:,-p:],X_m  #W[:,-p:]


# %%
# ############################################################
# faceRec_img_data = [x.flatten() for x in faceRec_img_data]
# faceRec_img_data = np.array(faceRec_img_data)
# faceRec_img_data = faceRec_img_data/(np.linalg.norm(faceRec_img_data,axis=1,keepdims=True))
# aaa=LDA(faceRec_img_data,faceRec_labels,p=1)

# %%
def NearestNeighbor(test_vec,feature_space,y):
    distances = np.linalg.norm(feature_space-test_vec,axis=1)
    nearest_idx = np.argmin(distances)
    y_pred = y[nearest_idx]
    return y_pred


# %%
def classifer(x_train,y_train,x_test,y_test,p,classifier_type='PCA',UMAP_plot=False):
    if classifier_type!='autoencoder':
        if classifier_type == 'PCA':
            l_dim_space,X_m = PCA(x_train,p)
            # x_train -= X_m
            # x_test -= X_m
        elif classifier_type == 'LDA':
            l_dim_space,X_m = LDA(x_train,y_train,p)

        x_train -= X_m
        x_test -= X_m

        x_train_proj = x_train@l_dim_space
        x_test_proj = x_test@l_dim_space

    elif classifier_type=='autoencoder':
        x_train_proj = x_train
        x_test_proj = x_test

    y_pred = []
    for test_vec in x_test_proj:
        y_pred.append(NearestNeighbor(test_vec,x_train_proj,y_train))
    y_pred = np.array(y_pred)
    accuracy = np.sum(y_pred==y_test)/len(y_test)
    print(f'Accuracy with p={p} : ',accuracy)

    if UMAP_plot:
        # train_reducer = umap.UMAP()
        # train_embedding = train_reducer.fit_transform(x_train_proj)
        # test_reducer = umap.UMAP()
        # test_embedding = test_reducer.fit_transform(x_test_proj)
        reducer = umap.UMAP()
        train_embedding = reducer.fit_transform(x_train_proj)
        test_embedding = reducer.transform(x_test_proj)
        fig,axes = plt.subplots(1,2)
        plt.suptitle(f'UMAP plot for p={p}')
        axes[0].scatter(train_embedding[:,0],train_embedding[:,1],c=y_train)
        axes[0].set_title('train data')
        axes[1].scatter(test_embedding[:,0],test_embedding[:,1],c=y_pred)
        axes[1].set_title('test data')
        plt.show()

    return accuracy



# %%
```

```
def faceRec_classification(faceRec_dataPath):
    y_train,x_train = load_data(faceRec_dataPath,'train')
    y_test,x_test = load_data(faceRec_dataPath,'test')

    x_train = [x.flatten() for x in x_train]
    x_train = np.array(x_train)
    x_train = x_train/(np.linalg.norm(x_train,axis=1,keepdims=True))
    x_test = [x.flatten() for x in x_test]
    x_test = np.array(x_test)
    x_test = x_test/(np.linalg.norm(x_test,axis=1,keepdims=True))

    p_list = [1,3,5,7,8,9,11,16]
    PCA_acc_list = []
    LDA_acc_list = []
    autoencoder_acc_list = []
    print('PCA')
    for p in p_list:
        PCA_acc = classifer(x_train,y_train,x_test,y_test,p,classifier_type='PCA',UMAP_plot=True)
        PCA_acc_list.append(PCA_acc)

    print('LDA')
    for p in p_list:
        LDA_acc = classifer(x_train,y_train,x_test,y_test,p,classifier_type='LDA',UMAP_plot=True)
        LDA_acc_list.append(LDA_acc)

    print('Autoencoder')
    for p in [3, 8, 16]:
        X_train,Y_train,X_test,Y_test = autoencoder.autoencoder_model(p)
        autoencoder_acc = classifer(X_train,Y_train,X_test,Y_test,p,classifier_type='autoencoder',UMAP_plot=True)
        autoencoder_acc_list.append(autoencoder_acc)

    plt.plot(p_list,PCA_acc_list,marker='*',label='PCA')
    plt.plot(p_list,LDA_acc_list,marker='o',label='LDA')
    plt.plot([3, 8, 16],autoencoder_acc_list,marker='x',label='Auoencoder')
    plt.title('Accuracy plot wrt p')
    plt.xlabel('p (feature dim.)')
    plt.ylabel('Accuracy')
    plt.legend()
    plt.show()
    # print(PCA_acc_list)

# %%
faceRec_classification(faceRec_dataPath)

# %%
```

```
# -*- coding: utf-8 -*-
"""cascaded_AdaBoost.ipynb

Automatically generated by Colab.

Original file is located at
    https://colab.research.google.com/drive/1gEa89fc9EzLOhEFpwCgga_-zFGA3JVsC

Task 3
"""

!unzip /content/CarDetection.zip



import numpy as np
import matplotlib.pyplot as plt
import cv2 as cv
import os

carDet_dataPath = ''

def load_data(root,dirName,category):
    data_path = os.path.join(root,dirName,category)
    data_dir = sorted(os.listdir(data_path))
    img_data = []
    for img_name in data_dir:
        img_path = os.path.join(data_path,img_name)
        img = cv.imread(img_path,cv.IMREAD_GRAYSCALE)
        img_data.append(img)
    labels = np.ones(len(img_data)) if category=='positive' else np.zeros(len(img_data))
    return labels, np.array(img_data)

y_train_pos, x_train_pos = load_data(carDet_dataPath,'train','positive')
y_train_neg, x_train_neg = load_data(carDet_dataPath,'train','negative')
y_test_pos, x_test_pos = load_data(carDet_dataPath,'test','positive')
y_test_neg, x_test_neg = load_data(carDet_dataPath,'test','negative')

x_train_pos_INT = np.array([(cv.integral(x_train_pos[i]))[1:,1:] for i in range(len(x_train_pos))])
x_train_neg_INT = np.array([(cv.integral(x_train_neg[i]))[1:,1:] for i in range(len(x_train_neg))])
x_test_pos_INT = np.array([(cv.integral(x_test_pos[i]))[1:,1:] for i in range(len(x_test_pos))])
x_test_neg_INT = np.array([(cv.integral(x_test_neg[i]))[1:,1:] for i in range(len(x_test_neg))])

def window_sum_val(img_INT,i,j,h,w):
    I1 = img_INT[i,j]
    I2 = img_INT[i+h-1,j]
    I3 = img_INT[i,j+w-1]
    I4 = img_INT[i+h-1,j+w-1]
    return I4-I2-I3+I1

def Haar_features(img_INT):
    #using 2 rectangle features

    #feature type1 - horizontal
    H,W = img_INT.shape
    feature_list1 = []
    for i in range(1,H):
        for j in range(1,W):
            for h in range(1,H-i+1):
            # h=1
              for w in range(1,((W-j+1)//2)):
                  S1 = window_sum_val(img_INT,i,j,h,w)
                  S2 = window_sum_val(img_INT,i,j+w,h,w)
                  feature_list1.append(S1-S2)

    #feature type2 - vertical
```

```
    feature_list2 = []
    for i in range(1,H):
        for j in range(1,W):
            for w in range(1,W-j+1):
            # w=1
              for h in range(1,((H-i+1)//2)):
                  S1 = window_sum_val(img_INT,i,j,h,w)
                  S2 = window_sum_val(img_INT,i+h,j,h,w)
                  feature_list2.append(S2-S1)

    feature_list = feature_list1 + feature_list2

    return np.array(feature_list)

def features_dataset(INT_dataset,fname):
    features_list = []
    for img_INT in INT_dataset:
        features_list.append(Haar_features(img_INT))
    features_list = np.array(features_list)
    np.savez_compressed(f'Haar_features_data/{fname}',features_list)
    return features_list

x_train_pos_F = features_dataset(x_train_pos_INT,'x_train_pos_F')
x_train_neg_F = features_dataset(x_train_neg_INT,'x_train_neg_F')
x_test_pos_F = features_dataset(x_test_pos_INT,'x_test_pos_F')
x_test_neg_F = features_dataset(x_test_neg_INT,'x_test_neg_F')

X_train_F = np.vstack((x_train_pos_F,x_train_neg_F))
y_train_F = np.hstack((y_train_pos,y_train_neg))
X_test_F = np.vstack((x_test_pos_F,x_test_neg_F))
y_test_F = np.hstack((y_test_pos,y_test_neg))

def get_weak_classifier(X,y,wts):
    wts = wts/np.sum(wts)
    min_err_seen = np.inf
    for f_idx in range(X.shape[1]):
        feature = X[:,f_idx]
        sort_idx = np.argsort(feature)
        sorted_feature = feature[sort_idx]
        sorted_wts = wts[sort_idx]
        sorted_y = y[sort_idx]

        pos_cumsum = np.cumsum(sorted_wts*sorted_y)
        neg_cumsum = np.cumsum(sorted_wts) - pos_cumsum
        total_pos_wts = np.sum(sorted_wts[sorted_y==1])
        total_neg_wts = np.sum(sorted_wts[sorted_y==0])

        error1 = pos_cumsum + total_neg_wts - neg_cumsum    #error assosciated with polarity 1
        error2 = neg_cumsum + total_pos_wts - pos_cumsum    #error assosciated with polarity -1

        min_error_idx = np.argmin(np.minimum(error1,error2))
        min_error = np.min(np.minimum(error1,error2))
        thres = sorted_feature[min_error_idx]

        if min_error<min_err_seen:
            if error1[min_error_idx]<=error2[min_error_idx]:
                p=1
                pred = (feature>=thres).astype(int)
                err = error1[min_error_idx]
            else:
                p=-1
                pred = (feature<thres).astype(int)
                err = error2[min_error_idx]
            classifier_param = [f_idx,thres,p,err,pred]
            min_err_seen = min_error
    return classifier_param
```

```
def cascade_stage(X,y,set_TPR,set_FPR,n_est=50):
    weight_scalar = 0.5
    l = int(np.sum(y))
    m = len(y) - l
    wts = np.append(np.ones(l)/(2*l),np.ones(m)/(2*m))
    classifier_list=[]
    sum_aH = np.zeros(len(y))
    a_thres = 0
    for est in range(n_est):
        classifier_param = get_weak_classifier(X,y,wts)
        pred = classifier_param[-1]
        et = classifier_param[-2]
        beta_t = et/(1-et+1e-6)
        correct_pred_vec = np.abs(pred-y)
        wts = wts*(beta_t**(1-correct_pred_vec))
        alpha_t = np.log(1/(beta_t+1e-6))
        classifier_param.append(alpha_t)
        classifier_list.append(classifier_param)

        sum_aH = sum_aH + (alpha_t*pred)
        a_thres = a_thres + (alpha_t*weight_scalar)
        strong_H = sum_aH>=a_thres

        TPR = np.sum(strong_H[:l])/l
        FPR = np.sum(strong_H[l:])/m
        TNR = 1-FPR
        FNR = 1-TPR
        rep_FPR = np.sum(strong_H[l:])/1758
        if TPR>=set_TPR and FPR<=set_FPR:
            break

    perf_metrics = [TPR,FPR,rep_FPR,TNR,FNR]
    neg_idx_rm = (y==0) & (strong_H==0)
    X_rev = np.delete(X,neg_idx_rm,axis=0)
    y_rev = np.delete(y,neg_idx_rm,axis=0)

    return X_rev,y_rev,classifier_list,perf_metrics

def cascaded_AdaBoost():
  #train
  X,y = X_train_F,y_train_F
  num_pos_ex = int(np.sum(y))
  num_cascade_stages = 10
  stage_classifier_list = []
  perf_metric_list = []
  for i in range(num_cascade_stages):
      print(f'Stage : {i+1}')
      X,y,classifier_list,perf_metrics = cascade_stage(X,y,set_TPR=0.99,set_FPR=0.45,n_est=20)
      stage_classifier_list.append(classifier_list)
      perf_metric_list.append(perf_metrics)
      print(f'metrics : {perf_metrics}')
      if len(y[num_pos_ex:])==0:
          print('Training stopped')
          break

  perf_metric_list=np.array(perf_metric_list)
  stage_classifier_list=np.array(stage_classifier_list,dtype=object)
  np.savez_compressed(f'Haar_features_data/tranied_clf',x=stage_classifier_list,y=perf_metric_list)
  plt.plot(perf_metric_list[:,1],label='FPR')
  plt.plot(perf_metric_list[:,-1],label='FNR')
  plt.xlabel('stages')
  plt.ylabel('Perf. rate')
  plt.legend()
  plt.show()
```

```
  #test
  pred_list = []
  weight_scalar = 0.5
  l = int(np.sum(y_test_F))   #num_positive
  m = len(y_test_F) - l       #num_negetive
  sum_aH = np.zeros(len(y_test_F))
  a_thres = 0
  test_FPR_list,test_FNR_list= [],[]
  T_FN,T_TN = 0,0
  for classifiers in stage_classifier_list:
      # weak_pred_list = []
      for clf in classifiers:
          f_idx,thres,p,_,_,alpha_t = clf
          feature = X_test_F[:,f_idx]
          test_pred = (feature>=thres if p==1 else feature<thres).astype(int)
          # weak_pred_list.append(test_pred)
          sum_aH = sum_aH + (alpha_t*test_pred)
          a_thres = a_thres + (alpha_t*weight_scalar)
      strong_H = (sum_aH>=a_thres).astype(int)

      #only index for positive images as we remove negative images in each stage
      num_TP = np.sum(strong_H[:l]==1)
      T_FN = T_FN + l - num_TP
      num_FP = np.sum(strong_H[l:]==1)
      new_num_TN = m - num_FP
      T_TN = T_TN + new_num_TN
      test_FNR_list.append(T_FN/l)
      test_FPR_list.append(1-(T_TN/m))
      #positively classified samples for the next cascade stage
      X_test_F = X_test_F [np.where(strong_H == 1),:][0]
      y_test_F = y_test_F[np.where(strong_H==1)[0]]
      l = np.sum(y_test_F == 1)
      m = np.sum(y_test_F == 0)
      if (l == 0):
          print ('No more positive images')
          break
  print (test_FPR_list)
  print (test_FNR_list)

  plt.plot(test_FPR_list,label='FPR')
  plt.plot(test_FNR_list,label='FNR')
  plt.xlabel('stages')
  plt.ylabel('Perf. rate')
  plt.legend()
  plt.show()

cascaded_AdaBoost()
```