Face Recognition

EE5907 Pattern Recognition

Project Assignment 2

Name:

Yi Ming

Matriculation Number:

A0194507Y

Date Submitted:

17 November 2023

Table of Contents

Data Preparation

The data contains PIE image and own image. In PIE image, 25 random datasets are chosen while own image contains 10 pictures. All the images has been converted to gray-scale and resized to 32*32. Images are split into training set and testing set. The split ratio is set to be 70% and 30%. Below are some basic information of the data.

Chosen set	[1, 3, 4, 5, 6, 7, 9, 14, 17, 20, 24, 26, 28, 32, 34, 36, 42, 50, 51, 56, 57, 59, 64, 66, 67]
PIE image	4250
PIE image for training	2975
PIE image for testing	1275
Own image	10
Own image for training	7
Own image for testing	3

Table 1: Data preparation

PCA

In this part, and all part below the images are converted to 1024 length vector for further calculation. The following code gives the PCA transformation with desired dimension.

```
def PCA_nD(train_set,k):
    m = train_set.mean(axis=0)
    temp = train_set - m

    cov_mat = np.cov(temp.T)
    eigenvalues, eigenvectors = np.linalg.eig(cov_mat)
    eigenvectors = eigenvectors.real
    sorted_result = eigenvalues.argsort()[::-1]
    eigenvalues = eigenvalues[sorted_result]
    eigenvectors = eigenvectors[:, sorted_result]
    eigenvalues = eigenvalues[0:k]
    face = eigenvectors[:,:k]
    return eigenvalues, face
```

The following figures shows the PCA transformation result under 2D and 3D condition. Though the result cannot do classification under low dimensions, but it shows that 3D has a better result than 2D.

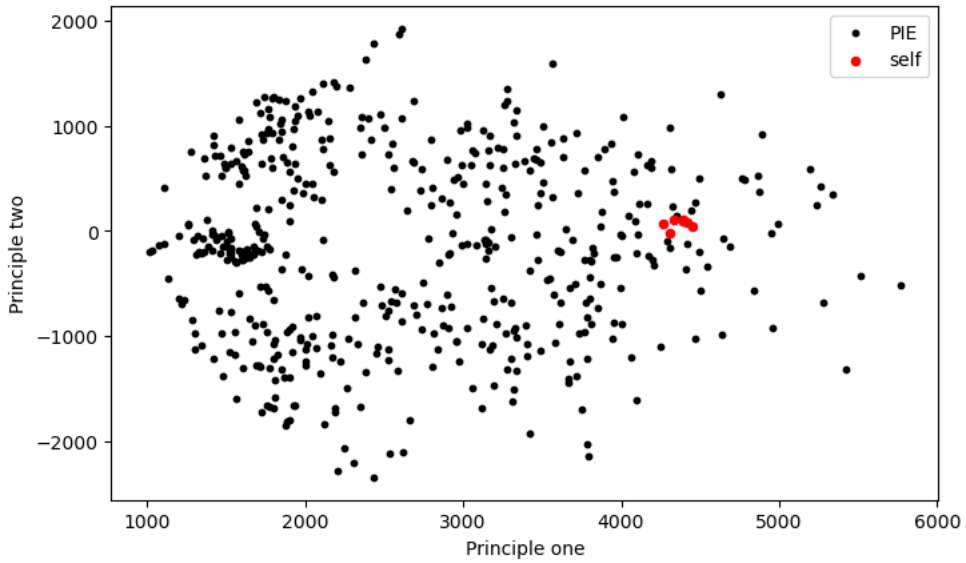

Figure 1: PCA 2D

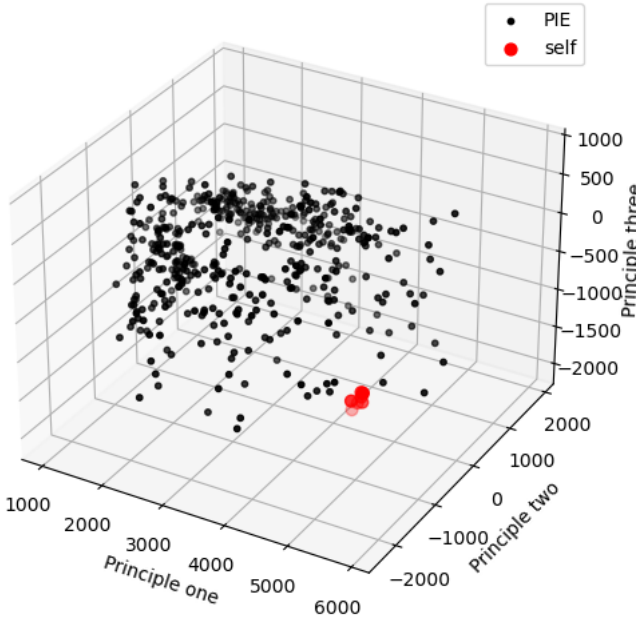

Figure 2: PCA 3D

The corresponding three eigenfaces are shown below.

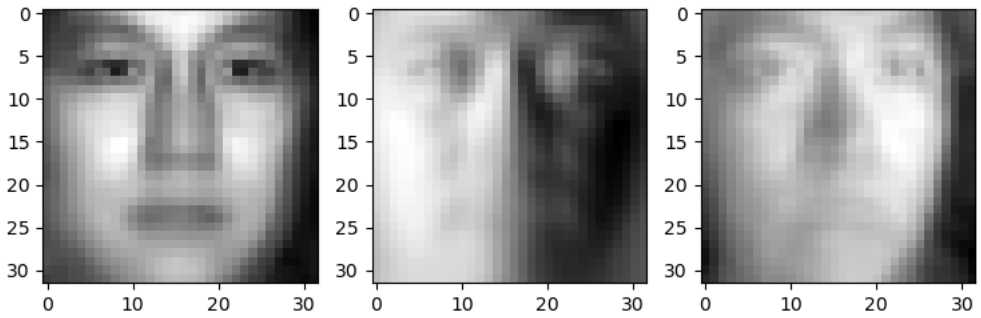

Figure 3: Eigenfaces

The following code provides the approach of nearest neighbour classification and provides the accuracy as well.

```
def euclidean_distance(x1, x2):
    return np.sqrt(np.sum((x1 - x2)**2))
def knn(X_train, y_train, X_test, k):
    y_pred = []
    for test_point in X_test:
        distances = [euclidean_distance(test_point, train_point) for train_point in X_train]
        sorted_indices = np.argsort(distances)
        k_nearest_neighbors = [y_train[i] for i in sorted_indices[:k]]
        y_pred.append(max(set(k_nearest_neighbors), key=k_nearest_neighbors.count))
    return np.array(y_pred)
def accuracy(y_true, y_pred):
    correct = np.sum(y_true == y_pred)
    total = len(y_true)
    return correct / total
```

The accuracy for different dimensions are shown in the following table. The results shows that the accuracy is increasing while increasing the dimension.

Dimension	PIE	Own image
40	81.92%	100%
80	85.45%	100%
200	87.72%	100%

LDA

Similar with PCA, LDA shares the same data processing and labeling process. The following graph gives the 2D and 3D dimension transformation. In the graph, we can see that the own image separates a lot from the PIE image. It may cause by the picture style difference. The own image has a much lighter background compared with the PIE image.

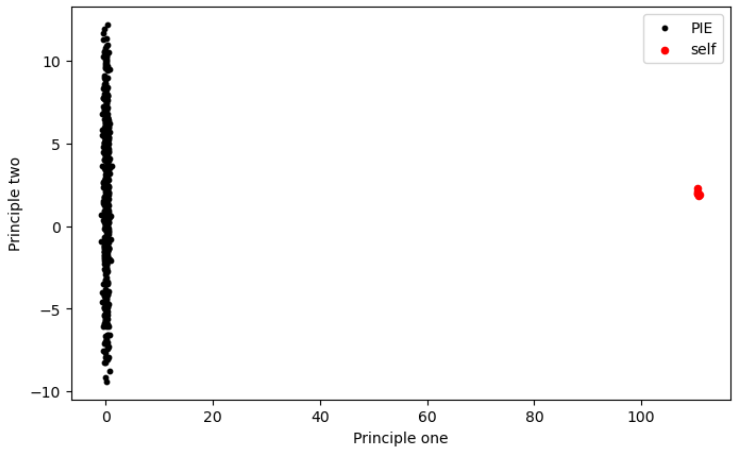

Figure 4: LDA 2D

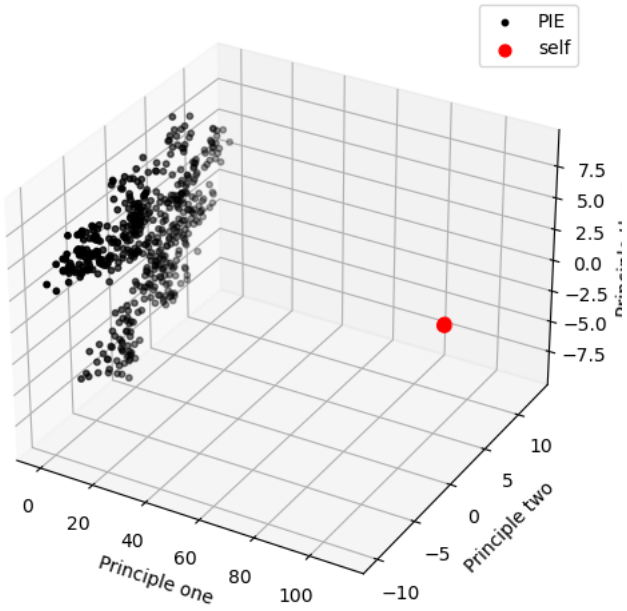

Figure 5: LDA 3D

The following table shows the LDA classification result with dimension 2,3,9. It shows that with the dimensionality increasing, the classification accuracy increases significantly.

Dimension	PIE	Own image
2	18.15%	100%
3	43.51%	100%
9	91.78%	100%

SVM

The following code shows the implementation of SVM.

```
from libsvm.svmutil import *
from numpy import ctypeslib
def SVM_calculation(train_data, train_labels, test_data, test_labels, cost_parameter):
    train_labels_list = train_labels.T[0].tolist()
    train_data_list = train_data.tolist()
    test_labels_list = test_labels.T[0].tolist()
    test_data_list = test_data.tolist()
    training_problem = svm_problem(train_labels_list, train_data_list)
    svm_config = '-t 0 -b 1 -c ' + str(cost_parameter)
    svm_params = svm_parameter(svm_config)
    trained_model = svm_train(training_problem, svm_params)
    predicted_labels, accuracy, _ = svm_predict(test_labels_list, test_data_list, trained_model, '-b 1')

    return accuracy[0]
```

Under the pre-processed by PCA, and given penalty parameter set as 0.01, 0.1, 1. The results are shown in the following table. Under lower dimension, the lower penalty gives a slightly lower accuracy.

C	40	80	200
0.01	98.59%	98.43%	98.59%
0.1	98.51%	98.51%	98.51%
1	98.51%	98.51%	98.51%

CNN

In this part the images are loaded as (number, height, weight, channel). Noted that the channel equals to 3 for the RGB image. The following code gives the construction of the CNN.

```
class CNN(nn.Module):
    def __init__(self, num_classes=4):
        super(CNN, self).__init__()
        self.l1 = nn.Sequential(
            nn.Conv2d(in_channels=3, out_channels=20, kernel_size=5, stride=1, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2))
        self.l2 = nn.Sequential(
            nn.Conv2d(in_channels=20, out_channels=50, kernel_size=5, stride=1, padding=1),
            nn.BatchNorm2d(50),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2))
        self.l3 = nn.Sequential(
            nn.Conv2d(in_channels=50, out_channels=500, kernel_size=5, stride=1, padding=1),
            nn.BatchNorm2d(500),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2))
        self.fc1 = nn.Sequential(
            nn.Dropout(0.2),
            nn.Linear(2000, 2000),
            nn.ReLU())
```

```
        self.fc2 = nn.Sequential(
            nn.Dropout(0.2),
            nn.Linear(2000, 2000),
            nn.ReLU())
        self.fc3 = nn.Linear(2000,26)
    def forward(self,x):
        x = self.l1(x)
        x = self.l2(x)
        x = self.l3(x)
        x = x.reshape(x.size(0), -1)
        x = self.fc1(x)
        x = self.fc2(x)
        out = self.fc3(x)
        return out
```

Noted that in the fully connected layer, the drop out function is added to avoid over-fitting problem. The model summary is shown below.

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1          [-1, 20, 30, 30]           1,520
              ReLU-2          [-1, 20, 30, 30]               0
         MaxPool2d-3          [-1, 20, 15, 15]               0
            Conv2d-4          [-1, 50, 13, 13]          25,050
       BatchNorm2d-5          [-1, 50, 13, 13]             100
              ReLU-6          [-1, 50, 13, 13]               0
         MaxPool2d-7            [-1, 50, 6, 6]               0
            Conv2d-8           [-1, 500, 4, 4]         625,500
       BatchNorm2d-9           [-1, 500, 4, 4]           1,000
             ReLU-10           [-1, 500, 4, 4]               0
        MaxPool2d-11           [-1, 500, 2, 2]               0
          Dropout-12                [-1, 2000]               0
           Linear-13                [-1, 2000]       4,002,000
             ReLU-14                [-1, 2000]               0
          Dropout-15                [-1, 2000]               0
           Linear-16                [-1, 2000]       4,002,000
             ReLU-17                [-1, 2000]               0
           Linear-18                  [-1, 26]          52,026
================================================================
Total params: 8,709,196
Trainable params: 8,709,196
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.01
Forward/backward pass size (MB): 0.81
Params size (MB): 33.22
Estimated Total Size (MB): 34.04
----------------------------------------------------------------
```

Based on this construction the results is shown below. Finally we get a result of accuracy equals to 94.4%

```
epoch=99       train loss=0.061207   train accuracy=0.982  test accuracy
=0.944
```

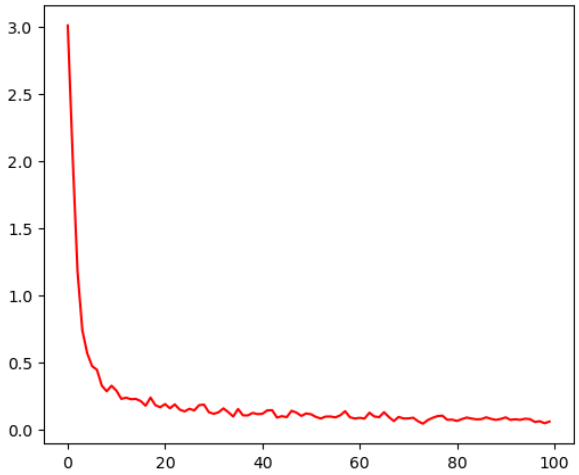

Figure 6: CNN train loss

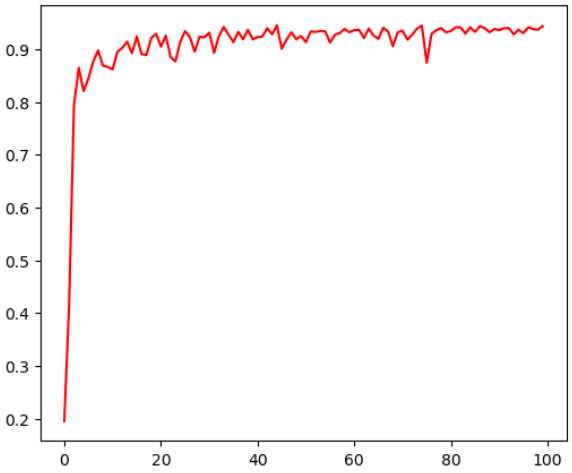

Figure 7: CNN accuracy