**实验报告**

专业：计算机科学与技术

姓名：贺情怡

学号：3180105438

日期：2021/1/1

课程名称： 计算机视觉 指导老师： 宋明黎 成绩：

实验名称： HW#2:特征脸识别

**一、实验目的和要求**

自己写代码实现 Eigenface 人脸识别的训练与识别过程：

1. 假设每张人脸图像只有一张人脸，且两只眼睛位置已知（即可人工标注给出）。每张图像的眼睛位置存在相应目录下的一个与图像文件名相同但后缀名为 txt 的文本文件里，文本文件中用一行、以空格分隔的4个数字表示，分别对应于两只眼睛中心在图像中的位置；

2. 实现两个程序过程（两个执行文件），分别对应训练与识别；

3. 自己构建一个人脸库（至少 40 人，包括自己），课程主页提供一个人脸库可选用；

4. 不能直接调用 OpenCV 里面与 Eigenface 相关的一些函数，特征值与特征向量求解函数可以调用；只能用 C/C++/Python，不能用其他编程语言；GUI只能用 OpenCV 自带的 HighGUI，不能用QT或其他的；平台可以用 Win/Linux/MacOS，建议 Win 优先；

5. 训练程序格式大致为：“mytrain.exe <\*能量百分比>\* <\*model文件名> <其他参数>\*…”，用能量百分比决定取多少个特征脸，将训练结果输出保存到 model 文件中。同时将前 10 个特征脸拼成一张图像，然后显示出来；

6. 识别程序格式大致为：“mytest.exe <\*人脸图像文件名> <model文件名> <其他参数>\*…”，将 model 文件装载进来后，对输入的人脸图像进行识别，并将识别结果叠加在输入的人脸图像上显示出来，同时显示人脸库中跟该人脸图像最相似的图像。

**二、实验内容和原理**

**PCA**

PCA（Principal Components Analysis）是一种统计分析、简化数据集的方法。它利用正交变换来对一系列可能相关的变量的观测值进行线性变换，从而投影为一系列线性不相关变量的值，这些不相关变量称为主成分（Principal Components）。

（线性代数知识回顾：PCA的数学原理[http://blog.codinglabs.org/articles/pca-tutorial.html]）

**基本思想：**

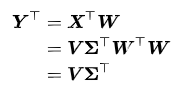
1. 将坐标轴中心移到数据的中心，然后旋转坐标轴，使得数据在C1轴上的方差最大，即全部n个数据个体在该方向上的投影最为分散。意味着更多的信息被保留下来。C1成为第一主成分。
2. C2第二主成分：找一个C2，使得C2与C1的协方差（相关系数）为0，以免与C1信息重叠，并且使数据在该方向的方差尽量最大。
3. 以此类推，找到第三主成分，第四主成分……第p个主成分。p个随机变量可以有p个主成分。

通过PCA，可以通过只保留对方差贡献最大的特征的方式来降低数据的维数（可以通过特征值的大小来判断贡献度）。PCA相当于找到了原有高维数据在低维度的投影钟，数据损失量最小的那个，作为提取出的主要特征。

**数学定义：**

PCA的数学定义是：一个正交化线性变换，把数据变换到一个新的坐标系统中，使得这一数据的任何投影的第一大方差在第一个坐标（称为第一主成分）上，第二大方差在第二个坐标（第二主成分）上，依次类推。

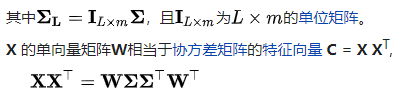
定义一个n × m的矩阵, XT为去平均值（以平均值为中心移动至原点）的数据，其行为数据样本，列为数据类别。则X的奇异值分解为X = WΣVT，其中m × m矩阵W是XXT的特征向量矩阵， Σ是m × n的非负矩形对角矩阵，V是n × n的XTX的特征向量矩阵。据此：



当 m < n − 1时，V 在通常情况下不是唯一定义的，而Y 则是唯一定义的。W 是一个正交矩阵，YTWT=XT，且YT的第一列由第一主成分组成，第二列由第二主成分组成，依此类推。

为了得到一种降低数据维度的有效办法，我们可以利用WL把 X 映射到一个只应用前面L个向量的低维空间中去：





在欧几里得空间给定一组点数，第一主成分对应于通过多维空间平均点的一条线，同时保证各个点到这条直线距离的平方和最小。去除掉第一主成分后，用同样的方法得到第二主成分。依此类推。在Σ中的奇异值均为矩阵 XXT的特征值的平方根。每一个特征值都与跟它们相关的方差是成正比的，而且所有特征值的总和等于所有点到它们的多维空间平均点距离的平方和。

**特征脸**

可以通过在一大组描述不同人脸的图像上进行主成分分析（PCA）获得。任意一张人脸图像都可以被认为是这些标准脸的组合。例如，一张人脸图像可能是特征脸1的10%，加上特征脸2的55%，在减去特征脸3的3%。值得注意的是，它不需要太多的特征脸来获得大多数脸的近似组合。另外，由于人脸是通过一系列向量（每个特征脸一个比例值）而不是数字图像进行保存，可以节省很多存储空间。

**实现：**

1. 准备一个训练集的人脸图像。构成训练集的图片需要在相同的照明条件下拍摄的，并将所有图像的眼睛和嘴对齐。他们还必须在预处理阶段就重采样到一个共同的像素分辨率（R×C）。现在，简单地将原始图像的每一行的像素串联在一起，产生一个具有R×C个元素的行向量，每个图像被视为一个向量。现在，假定所有的训练集的图像被存储在一个单一的矩阵T中，矩阵的每一列是一个图像。
2. 减去均值向量. 均值向量a要首先计算，并且T中的每一个图像都要减掉均值向量。
3. 计算协方差矩阵S的特征值和特征向量。每一个特征向量的维数与原始图像的一致，因此可以被看作是一个图像。因此这些向量被称作特征脸。他们代表了图像与均值图像差别的不同方向。通常来说，这个过程的计算代价很高（如果可以计算的话）。
4. 选择主成分。一个D x D的协方差矩阵会产生D个特征向量，每一个对应R × c图像空间中的一个方向。具有较大特征值的特征向量会被保留下来，一般选择最大的N个，或者按照特征值的比例进行保存，如保留前95%。

这些特征脸现在可以用于标识已有的和新的人脸：我们可以将一个新的人脸图像（先要减去均值图像）投影到特征脸上，以此来记录这个图像与平均图像的偏差。每一个特征向量的特征值代表了训练集合的图像与均值图像在该方向上的偏差有多大。将图像投影到特征向量的子集上可能丢失信息，但是通过保留那些具有较大特征值的特征向量的方法可以减少这个损失。

**三、实验步骤与分析**

**输入图像的预处理**

1. 因为之后也会需要对自己的照片进行处理，所以直接忽略了给出的数据库中的眼睛信息，使用了python自带的人脸识别库进行头像裁剪。
2. 对裁剪出的头像做一遍直方图均衡化，平衡不同光照的情况

|  |
| --- |
| img = cv.imread(path, -1)  FACE = cv.CascadeClassifier(r'haarcascade\_frontalface\_default.xml')  Scan = FACE.detectMultiScale(img,scaleFactor=1.1,minNeighbors=3)  if len(Scan) == 1 :  for (x,y,w,h) in Scan:  cropped = img[y:y+h, x:x+w]  # cropped is where face is  cv.rectangle(img, (x,y), (x+w, y+h), (255, 255, 255), 1)  cropped = cv.equalizeHist(cropped)  cropped = cv.resize(cropped, (250, 250))  # change it to 250 \* 250 (then will be 62500)  cv.imshow("ori",img)  cv.imshow("figure",cropped)  cv.waitKey(1)  print(path)  # change it to 1-D and append it to T  cropped = cropped.reshape(cropped.size, 1)  T.append(cropped) |

**PCA**

在实验中我自己实现了一遍PCA，可以通过输入能量来选择前若干个特征脸。但最终还是决定使用OpenCV自带的PCACompute来运算。（在实验结果章节中会对比计算结果）

其中使用自己实现的PCA算法的代码为train\_my.py

|  |
| --- |
| ENERGY = float(input("input the energy(float in 0~1)"))  Sum = 0  for i in range(vs.shape[0]):  Sum += vs[i]/sum(vs)  if Sum >= ENERGY:  break  NUM\_EIGEN\_FACES = max([i, NUM\_EIGEN\_FACES])  vs = vs[0:NUM\_EIGEN\_FACES].copy()  Vs = Vs[:, 0:NUM\_EIGEN\_FACES].copy()  print(Vs.shape) |

直接使用cv2库中的PCA算法（此处是用户自定义输入采用前NUM\_EIGEN\_FACES个脸）：

|  |
| --- |
| Mean, Vs = cv.PCACompute(T, mean=None, maxComponents=NUM\_EIGEN\_FACES) |

**EigenFace**

下面的代码用于保存和显示EigenFace

|  |
| --- |
| for V in Vs:  eigenFace = V.reshape((250, 250))  # here we will show the eigenface  cv.imshow('figure', eigenFace)  cv.waitKey(1)  eigenFaces.append(eigenFace)  eigenFaces = np.array(eigenFaces)  eigenFace\_mean = eigenFaces.mean(axis=0)  Mean = Mean.reshape((250, 250))  plt.figure()  plt.title('Mean')  plt.subplot(1, 2, 1)  plt.imshow(Mean, cmap = plt.get\_cmap('gray'))  plt.subplot(1, 2, 2)  plt.imshow(eigenFace\_mean, cmap = plt.get\_cmap('gray'))  plt.show() |

储存recognize所需的矩阵信息

|  |
| --- |
| np.save('save\_database\\save\_Mean', Mean)  np.save('save\_database\\save\_A', T)  np.save('save\_database\\eigenface', eigenFaces) |

**识别**

读取图像及预处理的相关操作和train时一样，不再赘述（这里应该注意，必须对图像进行一样的预处理）

识别前需进行的矩阵运算：

1. PRO矩阵用于存储将图像映射到若干特征脸的空间时，每张图像对应的坐标
2. 将需要识别的图像也映射到这个空间中（都是乘上特征脸矩阵即可）

|  |
| --- |
| PRO = A.dot(Vectors)  # …  # to 1-D  Array = cropped.reshape(cropped.size,1)  Array = np.mat(np.array(Array)).squeeze()  # get the difference and project it to the eiganface space  meanVector = Mean.flatten()  meanVector = meanVector.squeeze()  diff = Array - meanVector  diff = diff.squeeze()  pro = diff.dot(Vectors) |

然后计算空间中离需要识别的图像距离最近的点

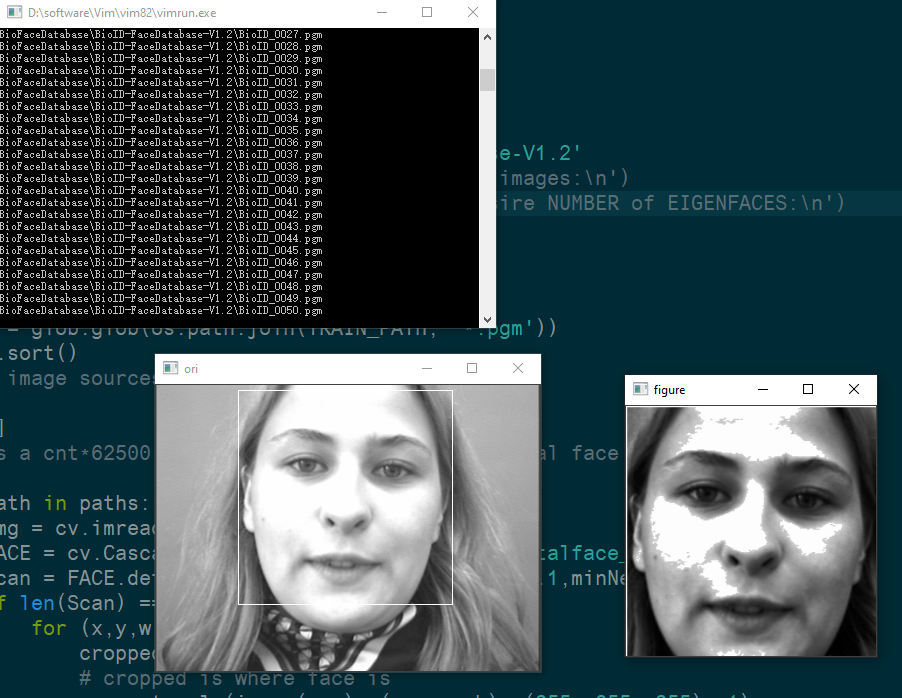
|  |
| --- |
| distance = []  for i in range(0, A.shape[0]):  cur = PRO[i,:]  temp = np.linalg.norm(pro - cur)  distance.append(temp)  print('No.' + str(i) + ' = ' + str(distance[i]))  minDistance = min(distance)  index = distance.index(minDistance) - 1 |

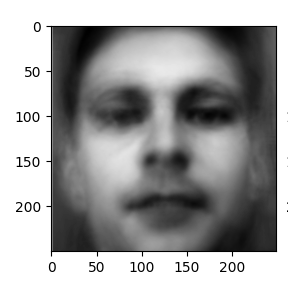
再打开数据库对应的文件夹，找到对应图像并显示即可

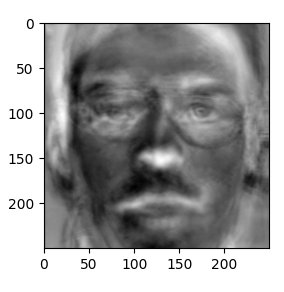
|  |
| --- |
| # load the train-data to find the most similar photo  TRAIN\_PATH = 'BioFaceDatabase\BioID-FaceDatabase-V1.2'  TrainFiles = os.listdir(TRAIN\_PATH)  paths = glob.glob(os.path.join(TRAIN\_PATH, '\*.pgm'))  paths.sort()  # …  result = cv.imread(paths[index], -1)  testImg = cv.putText(testImg, "Most Similar: ", (40, 50), cv.FONT\_HERSHEY\_PLAIN, 1.0, (0, 0, 0), 2)  testImg = cv.putText(testImg, paths[index].split ( "\\" ) [ -1 ], (40, 100), cv.FONT\_HERSHEY\_PLAIN, 1.0, (0, 0, 0), 2)  cv.imshow("ori",testImg)  cv.imshow("recognize result",result)  cv.waitKey(0) |

**四、实验结果**

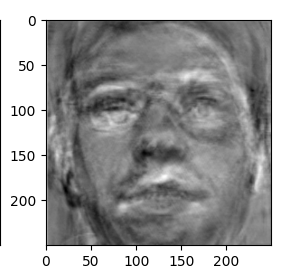
**Train**

预处理：

平均脸：

使用cv2自带的PCACompute算出的特征脸叠加（前十张）：

使用自己实现的PCA算出的特征脸叠加（能量值0.8）：



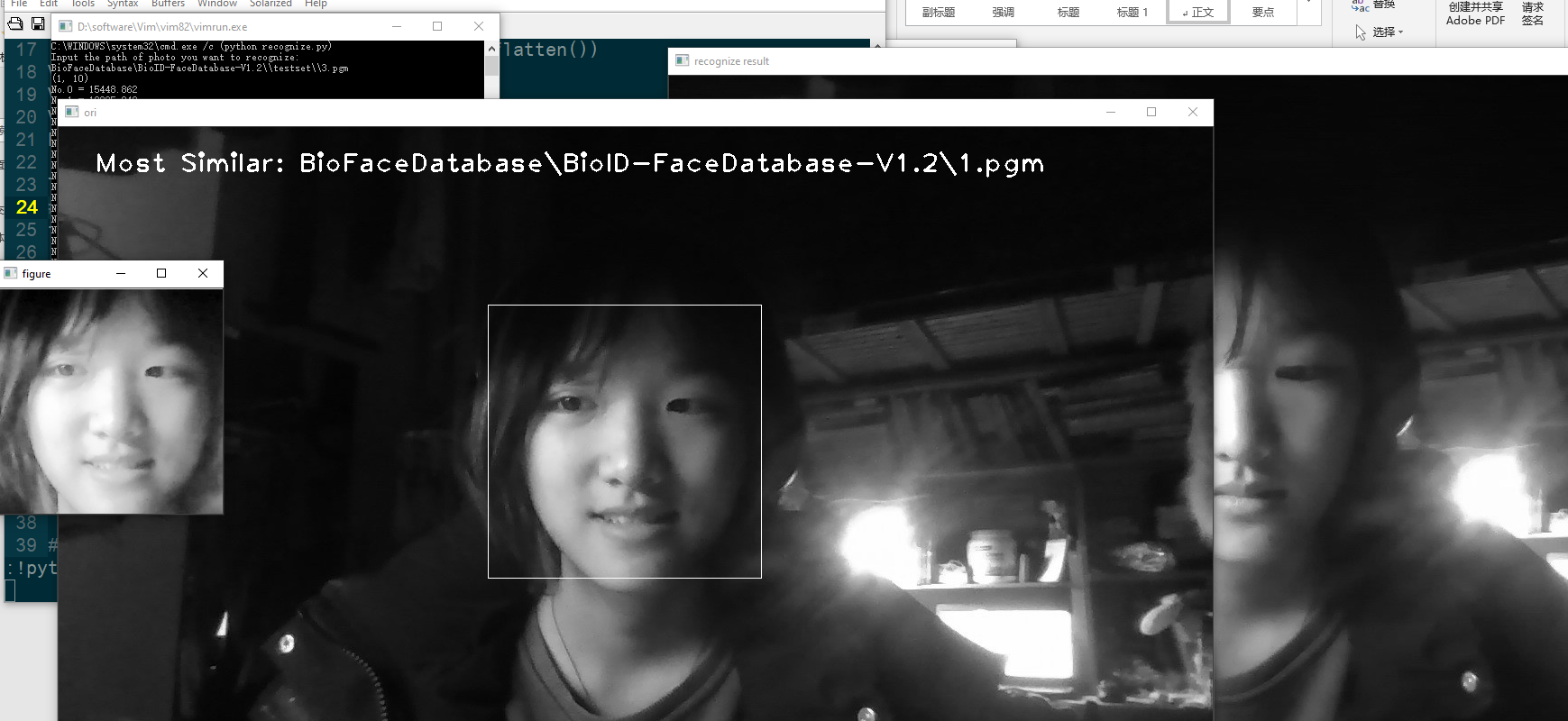
随便选取的一张特征脸：



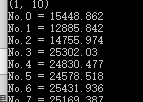
**识别：**

Cv2自带的PCA的结果：

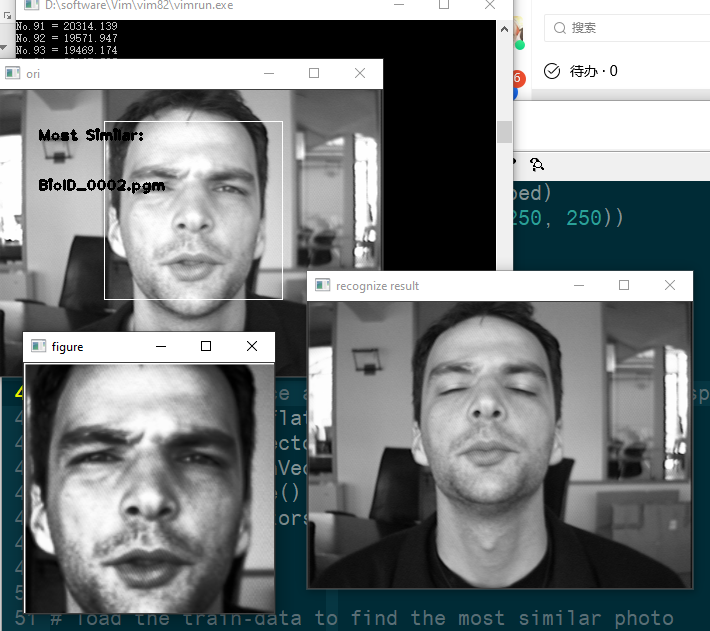
1. 自己的照片



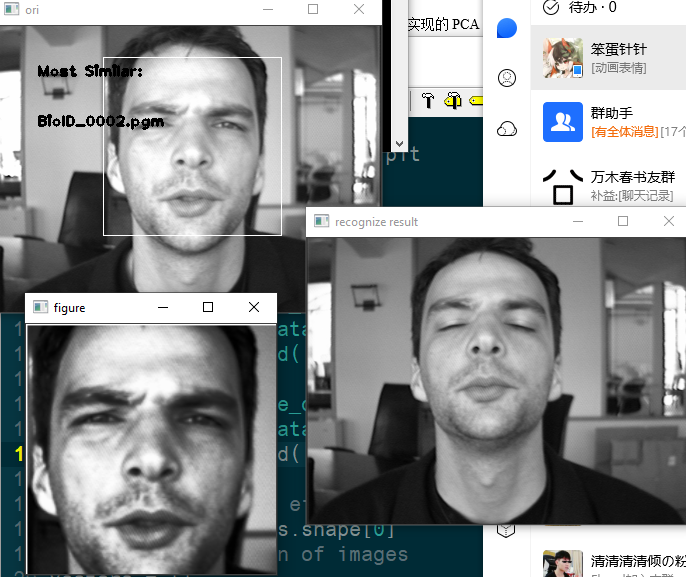
可以看出，不同人的照片之间的距离还是相当大的

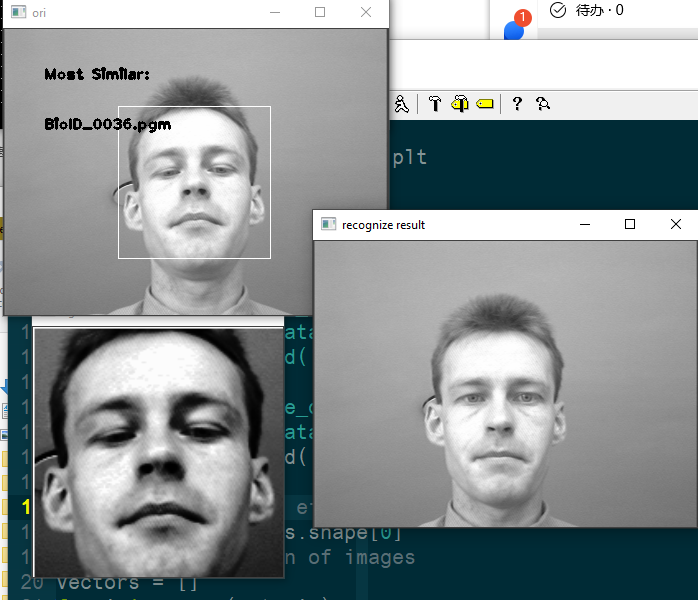


1. 数据库中给出的照片



**自己实现的PCA的识别效果：**





**五、心得体会**

**算法理解：**

其实这个算法非常符合直觉，就是降低数据的维度达到加速的目的。

把在height\*width维的空间中进行的比较变换到了维度相当低的空间中，而PCA的目的就是找到最恰当的投影空间（使损失的信息尽量少）

**实验体会：**

Python的语法真的好厉害……

DDL晚上发现对算法理解和实现有误……在调试过程中python坏掉了（看起来是管理员权限之类的坏了），卸载重装python搞了好久还是没办法……于是只好迟交了orz

虽然当时线性代数学得比较认真，但过了两年还是失去了很多记忆XD。

刚开始以为求特征值会对size\*size的矩阵来做，于是用OpenCV自带的PCA在写，之后发生好像用N\*N（N为图片数目）也行，于是自己实现了一下

**用openCV自带的PCA程序一直失败**

（但是它刚开始莫名其妙报错……调了一晚上甚至重装python也没有成功……之后新建文件备份代码，发现就可以跑通了……迷茫

Train.py运行完会报错shell returned -1073740791，查了很多资料没有发现原因，但是已经运行完了（而且在别人16G内存的电脑上没有报错）于是就不管了QwQ（可能是因为内存炸了？但是明明运行完了的QAQ）