Face Recognition using EigenFaces

Group - 2

Deeprajsinh Gohil - AU2140164

Krishang Shah - AU2140035

Charmi Desai - AU2140167

Raiyan Diwan - AU2140111

Khwahish Patel - AU2140160

Introduction:

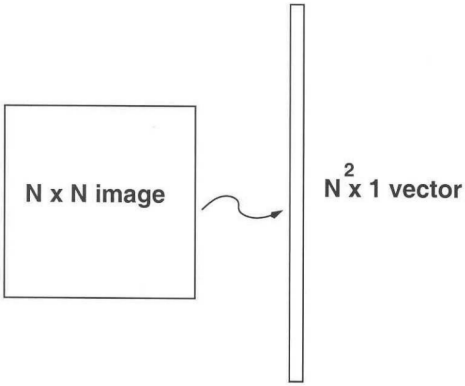
This report's objective is to draw attention to the use of linear algebra in the project area of face recognition and computer vision. The concept of "Face identification utilising eigen faces" will be presented in this study, along with a general overview of how linear algebra is used in computer vision. The goal of this work is to develop a method for leveraging eigen faces to detect and recognise faces. This use of linear algebra in the area of computer vision is very well thought out.

The need for Facial-Recognition

Facial recognition technology is becoming increasingly important in our lives. It is being used in a variety of applications such as security, identity verification, and automated customer service. It can be used to detect potential threats in public places, to speed up the check-in process at airports, and to identify people in photos or videos. Furthermore, it can help protect against identity theft and fraud, and also be used to create personalized marketing campaigns.

Face Recognition using eigen values:

A data matrix is on every face. Think of a grayscale image where each pixel is represented by a number. A pixel can be one if it is white, zero if it is black, or anything in between is considered to be grey. Therefore, the input of a dataset of photographs of a person whose face we need to recognise is our main focus. The most used method right now, as mentioned above, is recognition with eigenfaces. The covariance matrix's Eigenfaces, often known as ghostly pictures, are the Eigenvectors of the covariance matrix. The main benefit of using this style is that it makes it easier for us to effectively represent the input data. Each distinct face in this instance is modelled as a linear combination of eigen faces. Now consider a dataset with k unique individuals and multiple images of the same individual, each with a dimension of NxN. Here, each image from each sub-dataset can be categorised as unique because it contains images of the same face taken from various faces or angles. We first transform these images into N^2 vectors, which in turn will create N^2x1 column vectors. Now, from the photographs provided in the dataset, we find the mean faces for each individual. Now that we would have k such mean faces, this mean picture would also be a column vector of dimension N^2x1. We obtain a matrix made up of column vectors, which are nothing more than the mean faces, and having k columns and N^2 rows. The covariance matrix's Eigenvalues and vectors are our next duty.



Methodology:

1. Singular Value Decomposition:

SVD is used to determine the value of the primary components before doing Principal Component Analysis (PCA). A matrix's singular value decomposition is written as: A = UΣV^T; Where the diagonal matrix contains non-negative singular values and U and V are orthogonal matrices, and where the singular values are arranged in descending order in.

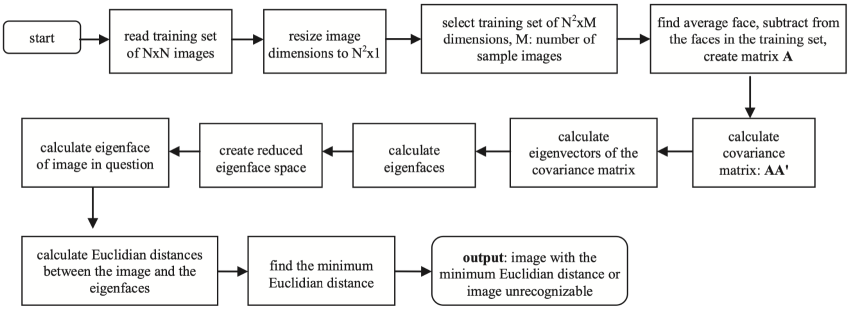
2. Covariance Matrix:

Now that we are familiar with SVD, it should be clear to us how PCA operates: by decomposing the Covariance matrix. Commonly, the covariance matrix is C1 = A^T*A. Now, the covariance matrix can be C1 = AA^T, but if we take the latter into consideration, it is obvious that our covariance matrix will be quite big because it will have a dimension of N^2xN^2. But if we think of our covariance matrix as an A^TA, we can see that it just depends on how many individuals are in our dataset, or k. As a result, we have the dimension kxk matrix.

3. EigenFaces:

Now that we have the covariance matrix, we must locate the Eigenfaces, which are nothing more than the covariance matrix's eigenvectors. This Eigenfaces' primary driving force is to: Represent the face photos more effectively by extracting key facial information. This also helps to decrease time and space complexity because just a few parameters are needed to represent each face. Thus each image is approximated using a subset of EigenValues.

Approach



Original Images

Mean Face

eigrnfaces

First, we open the image using the Pillow library's Image module. Once we convert the image to grayscale, it is obvious that the amount of information saved per pixel will be significantly decreased. We use the ImageOps module to accomplish this. We change it from being a NxN matrix to a b column matrix that is N^2x1.

The images that are represented as pixels are found in the columns of the b matrix. We now locate the mean matrix and give it the name "bMean" given the matrix "b". The covariance matrix is then located using the method b^T * b.' Jacobi's method is then used to determine the eigenvalues and eigenvectors. The eigenFace matrix is made up of the eigenVectors of the covariance matrix.We determine the projection of each image into the eigenFace space using this eigenFace matrix. This projection matrix is contrasted with the input image's projection matrix.

Snippets

In [ ]:

```
    class EigenFaces(object):
        def train(self, training_images):
            self.projected_classes = []
            self.count_captures = 0
            self.count_timer=0
            self.list_array_images, self.list_label, \
                fclass_samples_list = \
                    read_images(training_images)

            image_matrix = np.array([np.array(Image.fromarray(img)).flatten()
                  for img in self.list_array_images],'f')

            self.eigen_matrix, variance, self.mean_image = pca.pca(image_matrix)

            for class_sample in fclass_samples_list:
                class_weights = self.project_image(class_sample)
                self.projected_classes.append(class_weights.mean(0))

        def project_image(self, X):
            X = X - self.mean_image
            return np.dot(X, self.eigen_matrix.T)

        def predict_face(self, X):
            min_class = -1
            min_distance = np.finfo('float').max
            projected_target = self.project_image(X)
            projected_target = np.delete(projected_target, -1)
            for i in range(len(self.projected_classes)):
                distance = np.linalg.norm(projected_target - np.delete(self.projecte
                if distance < min_distance:
                    min_distance = distance
                    min_class = self.list_label[i]
            return min_class

        def __repr__(self):
            return "PCA (num_components=%d)" % (self.numerical_comp)
```

In [ ]:

```
def read_images(path, sz=None):

    class_samples_list = []
    class_matrix = []
    images, image_labels = [], []
    for dirname, dirnames, filenames in os.walk(path):
        for subdir in dirnames:
            path = os.path.join(dirname, subdir)
            class_samples_list = []
            for filename in os.listdir(path):
                if filename != ".DS_Store":
                    try:
                        im = Image.open(os.path.join(path, filename))
                        if (sz is not None):
                            im = im.resize(sz, Image.ANTIALIAS)
                        images.append(np.asarray(im, dtype = np.uint8))
                    except IOError as e:
                        errno, strerror = e.args
                        print("I/O error({0}): {1}".format(errno, strerror))
                    except:
                        print("Unexpected error:", sys.exc_info()[0])
                        raise

                    class_samples_list.append(np.asarray(im, dtype = np.uint8))

            class_samples_matrix = np.array([img.flatten()
                for img in class_samples_list],'f')

            class_matrix.append(class_samples_matrix)

            image_labels.append(subdir)

    return images, image_labels, class_matrix
```

In [ ]:

```
def pca(X):
    num_data, dim = X.shape
    mean_X = X.mean(axis=0)
    X = X - mean_X

    if dim > num_data:
        M = np.dot(X,func.Transpose(X))
        e,EV = np.linalg.eigh(M)
        tmp = func.Transpose(np.dot(func.Transpose(X),EV))
        V = tmp[::-1]
        S = np.sqrt(e[::-1])

        for i in range(V.shape[1]):
            V[:,i] /= S
    else:
        U, S, V = np.linalg.svd(X)
        V = V[:num_data]
    return V, S, mean_X
```

References:

Rosebrock, Adrian. “OpenCV Eigenfaces for Face Recognition.” PyImageSearch, 12 May 2021, https://pyimagesearch.com/2021/05/10/opencv-eigenfaces-for-face-recognition/ (https://pyimagesearch.com/2021/05/10/opencv-eigenfaces-for-face-recognition/).

navydhara79-zz. “Navydhara79-ZZ/Face-Recognition-Using-Eigen-Faces: Implementation of Facial Recognition Using Eigenfaces Technique.” GitHub, https://github.com/navydhara79-zz/Face-Recognition-using-eigen-faces (https://github.com/navydhara79-zz/Face-Recognition-using-eigen-faces).

Eigenfaces for Face Detection/Recognition - Johns Hopkins University. http://vision.jhu.edu/teaching/vision08/Handouts/case_study_pca1.pdf (http://vision.jhu.edu/teaching/vision08/Handouts/case_study_pca1.pdf).