Feature Based Face Detection and Recognition for Augmented Reality

Realised by
Romain Leprêtre - 49039

Computer Vision and Mixed Reality

Presented to
Pedro Mendes Jorge

ISEL
Master in Informatics and Multimedia Engineering
Thursday 3rd June, 2021

1 Introduction

The main goal of the application is to recognize the user standing in front of the camera and put a virtual object on their face depending on who that person is. In order to do so we will go through multiple steps. The first thing to do is to create a database of normalized pictures of the users that will be recognized and subsequently train a neural network that will be used for live face recognition.

Once our algorithm is trained we can start working on the live video capture. First, we will use Haar Cascade Classifier[1] in order to detect the eyes which will not only help us normalize the current frame in order to pass it to the nearest neighbor algorithm but also helps us to align the virtual object correctly with the user's face. The following sections of this report will be dedicated to explain each process more thoroughly.

2 Creating the face database

In order to create a training set for the KNN[1], I asked 7 to 10 selfies to my 3 flatmates and used myself as the fourth class. I then created a script that would normalize them to a 56 by 46 grayscale version. As stated in the instruction, both eyes, right and left, were supposed to be "perfectly aligned horizontally, and located in line 24, columns 16 and 31, respectively."

2.1 The normalize() function

The first job of the function is to detect the coordinates of the center of the eyes on the picture for which I used Haar Cascade Classifier. Note, that I had to tweak the second argument of the ye_cascade.detectMultiScale() function in order for it to scale down the image resolution and get less false positives on what it thought to be eyes.

```
eyes = eye_cascade.detectMultiScale(roi_gray, 1.3)
eyes_centers = []

for (ex,ey,ew,eh) in eyes:
    eyes_centers.append(np.array((x+(ex + ew//2),y + (ey + eh//2))))

if  len(eyes_centers) >= 2 and eyes_centers is not None:
    eye_one = np.array([eyes_centers[0][0],eyes_centers[0][1],1])
    eye_two = np.array([eyes_centers[1][0],eyes_centers[1][1],1])
```

Once the eyes are found, it is important to determine which is the left and the right eye. Since we need to rotate the image to align the eyes horizontally it is important to always pick the same eye as the center of rotation, in my case, the left one.

The math was fairly simple, considering h, the hypotenuse and θ the angle between the eyes and the horizontal, x_1 and y_1 the left eye coordinates and x_2 and y_2 the right eye coordinates

$$h = \| \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} - \begin{pmatrix} x_2 \\ y_2 \end{pmatrix} \|$$

$$\theta = -\arcsin(\frac{y_1 - y_2}{h})$$

[1] k-nearest neighbors algorithm

Knowing this we can create the Rotation matrix R:

$$R = \begin{bmatrix} \alpha & \beta & (1-\alpha) * x_1 - beta * y_1 \\ -\beta & \alpha & \beta * x_1 + (1 - alpha) * y_1 \end{bmatrix}$$

with $\alpha = cos(\theta)$ and $\beta = sin(\theta)$.

Thanks to this matrix and the warpAffine() function from opencv, we can rotate the whole image by an angle θ and a rotation center $\begin{pmatrix} x_1 \\ y_1 \end{pmatrix}$. We can also find the rotated position of the right eye[2]

$$\begin{pmatrix} x_1' \\ y_1' \end{pmatrix} = R \cdot \begin{pmatrix} x_2 \\ y_2 \\ 1 \end{pmatrix}$$

Next up, we need to determine the scaling ratio. Since we want the eyes to be 15 pixels from each other we can safely assume:

$$ratio = \frac{15}{\| \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} - \begin{pmatrix} x_2 \\ y_2 \end{pmatrix} \|} = \frac{15}{h}$$

Once we applied the ratio to all matrices and vectors, we can crop our image in a way that places the eyes to the correct position.

```
normalized_img =
    scaled_gray[ int(left_eye[1]-24): int(left_eye[1]+32),
     int(left_eye[0]-16): int(right_eye[0]+15)]
```

2.2 The classes.csv file

Once we have all the images normalized, we have to create a csv file that will match every image file name to a certain class number. This way we can simply load the csv file in order to give the training answers to our neural network.

3 Projecting the training set

The face recognition algorithm is based on feature vectors. Since their size is considerably smaller than a full picture, it makes it a lot easier for the computer to execute computationally heavy tasks.

The idea is to use the Fisherfaces algorithm to create the feature vector since it produces very good results when trying to classify feature vectors. It was not an easy task since there are a lot of huge matrices of various dimensions and it was quite complex to keep track of everything.

[2]We do not need to this for the left eye as it is the center of rotation and does not move

The idea is to find the W matrices for the Principal Component Analysis[3] and the FLD[3]. By multiplying a normalized image vector by both matrices we can project them on a smaller space that will help us perform our calculations more easily.

First, we read our image files and build a x matrix made out x_i column vectors representing each image. This can be easily done using numpy.vstack and transposing or simply numpy.hstack after flattening each image 56 by 46 matrix. Next, we need to compute μ the mean of all the image vectors which will be useful in both the Eigenfaces and Fisherfaces algorithm.

3.1 Eigenfaces algorithm

In order to define the W_{fld} matrix, we need to apply the Eigenfaces algorithm to x.

1. Find $A = x - \mu$
2. Since A's dimension is 2576 by the amount of pictures in the set, N, we rather compute the eigenvectors of lower dimension $(N \times N)$ matrix $R = AA^T$
3. We use numpy.linalg.eig() on R to obtain the eigenvectors, eigenvalue pairs.
4. We keep the eigenvectors corresponding to the $N - c$ highest eigenvalues.[4]
5. Finally we find $W_{pca} = \frac{A \cdot V}{\|A \cdot V\|}$

3.2 Fisherfaces algorithm

Once we have W_{pca}, we still need to find W_{fld} with the Fisherfaces algorithm to complete the projection.

1. We need to find the S_b and S_w, respectively then inter and intra scatter matrix.

$$S_b = \sum_{i=1}^{c} n_i(\mu_i - \mu)(\mu_i - \mu)^T$$

$$S_w = \sum_{i=1}^{c} (x - \mu_i)(x - \mu_i)^T$$

Which can be translated into code like this:

```
for c in  range( len(my_classes_set)):
    class_index = np.nonzero(my_classes == c+1)
    class_means[:,c] = np.mat(np.mean(x[:,class_index[0]], axis = 1))

    S_w = S_w +
    np.dot((x[:,class_index[0]] - np.mat(class_means[:,c]).T),
    (x[:,class_index[0]] - np.mat(class_means[:,c]).T).T)

    S_b = S_b +
    class_index[0].size*np.dot((np.mat(class_means[:,c]).T - mu),
```

[3] Fisher Linear Discriminant

[4] c, the amount of classes in our training set

```
(np.mat(class_means[:,c]).T - mu).T)
```

2. From there we can find $\tilde{S_b} = W_{pca}^T S_b W_{pca}$ and $\tilde{S_w} = W_{pca}^T S_w W_{pca}$

3. Compute the eigenvectors and eigenvalue pairs of the resulting matrix $\frac{\tilde{S_b}}{\tilde{S_w}}$

4. And finally we get W_{fld} the eigenvectors corresponding to the $c-1$ highest eigenvalues.

3.3 Obtaining the projected training set

Once we get both projection matrices W_{pca} and W_{fld} we can project our training set in the smaller space and get:

$$Y = W_{fld}^T W_{pca}^T (x - \mu)$$

Y contains the N projected vectors of dimension $c-1$.

4 Training the neural network

Since opencv does most of the work for us, training the network wasn't the most complex task. Once the knn object is initialized, we need to use the train function and give it three arguments. The training set's projected vectors, the Y matrix, whether the vectors are organized in rows or columns and the corresponding class for each vector.

```
Y_train, W_pca, W_fld,  my_training_classes, mu  =
fisherFaces('./faceDB_20_21/training/')

knn = cv2.ml.KNearest_create()
knn.train(Y_train, cv2.ml.ROW_SAMPLE, my_training_classes)
```

5 The face recognition process

We are now getting to the live part of our application, everything else is computed beforehand. Since it is live we will use a *while* loop that will read each frame from the camera. For each loop the application will execute the following code.

```
img, left_eye, right_eye, theta, eyes_dist = normalize(frame)
img = img.reshape(2576,1).astype(np.float32)
projected_img = np.linalg.multi_dot([W_fld.T, W_pca.T, img - mu])
ret, results, neighbours, dist = knn.findNearest(projected_img.T, 1)
```

The first step is normalizing the frame that we get from the camera to the same grayscale 56 by 46 images, then flattening it. Afterwards we can use our projection matrices W_{pca} and W_{fld} to project the vector to the smaller space. And finally we ask our nearest neighbor algorithm to find which class is the vector closer to.

In order to find which distance should be the threshold for the unknown class I used the training set that we were given and added my own normalized pictures to the testing set. After that I computed the mean value of the distance for all the false results.

```
knn.train(Y_train, cv2.ml.ROW_SAMPLE, my_training_classes)
ret, results, neighbours, dist = knn.findNearest(Y_test, 1)
index = np.nonzero((results.T == my_classes) == False)
print(np.mean(dist[index[1]]))
```

Since the mean was around 410 000 I decided to give my threshold some margin and tried it with 450 000. It was still not tolerant enough and after some tweaking I found 600 000 to be working quite well since it was successfully recognizing the known classes and not getting a false positive for an unknown person.

6 Adding the virtual object

First we want to get the virtual object to the right size, in order to do that we need to take some measurement on the initial image. Either using Photoshop or Gimp, we need to determine where would the center of the eyes be on the virtual object and what the distance between the eyes would be. This way we can easily find the scaling ratio.

$$ratio = \frac{EyesDistance}{VirtualObjectEyesDistance}$$

and resize it accordingly,

```
virtual_object = cv2.resize(virtual_object, (scaled_height,scaled_width),
                 interpolation = cv2.INTER_AREA)
```

After that we need to create a mask for the chroma keying[2] and it's invert from the the original image. The image background is RGB(0, 255, 0) green so we can exclude that color.

```
lower_green = np.array([0, 100, 0])      ##[R value, G value, B value]
upper_green = np.array([120, 255, 100])
mask = cv2.inRange(virtual_object, lower_green, upper_green)
```

After that we need to repeat the mask on 3 axis to match the dimension of the real image, invert it and finally multiply both by 255 since before that their values' range is from 0 to 1.

```
mask = np.repeat(mask[:, :, np.newaxis], 3, axis=2)
inv_mask = np.bitwise_not(mask)
mask = mask*255
inv_mask = inv_mask*255
```

The rotation on the virtual object is also applied using the warpAffine method and a rotation matrix which will rotate the object using the center of the eyes on the virtual object as the rotation center.

After the rotation we can finally place our virtual object on the image. Since we want to align the mask center with the eyes' center we can substract the local coordinates of the mask center to the global coordinates of the eyes' center and obtain the position of the top left corner of the virtual object.

```
y = int(eyes_center[1] - mask_center[1])
x = int(eyes_center[0] - mask_center[0])
```

```
h, w = virtual_object.shape[0], virtual_object.shape[1]

img[y:y+h, x:x+w] = inv_mask * virtual_object + img[y:y+h, x:x+w] * mask
```

7 Conclusion

In the end the project was practically working, there are a few improvements that could be made around the mask part, so far the application crashes if the mask has to get out of the frame since its index would be out of bounds.

I have also had trouble getting the virtual object to rotate correctly, it tends to translate as well which is an undesired side effect. At first I thought it was because I picked the left eye as a rotation center but switching to the center of the eyes didn't improve anything and gave me the exact same results.

In conclusion, it was a very interesting and practical project, I am glad that I got it working even though it is not perfect. The neural network has a 70% success rate which is quite ok since the Fisherfaces algorithm was by far the hardest part of this whole project.

I uploaded the project on this github repository `https://github.com/rlepretre/face-recognition` .

References

[1] *Cascade Classifier*. URL: https://docs.opencv.org/3.4/db/d28/tutorial_cascade_classifier.html. (accessed: 01.05.2020).

[2] Teja Kummarikuntla. *Blue or Green Screen Effect with OpenCV [Chroma keying]*. URL: https://medium.com/fnplus/blue-or-green-screen-effect-with-open-cv-chroma-keying-94d4a6ab2743. (accessed: 26.05.2020).

[3] Wikipedia. *Principal component analysis*. URL: https://en.wikipedia.org/wiki/Principal_component_analysis. (accessed: 26.05.2020).