Eigenfaces for Facial Recognition

Dylan Bao, Joseph Zawisa, and Max Zawisa

Overview

Computer vision involves the use of linear algebra to obtain and manipulate useful information from images. One specific domain of computer vision is facial recognition. Essentially, facial recognition is the use of computers to identify several unique images of one person's face as being images of the same person. Facial recognition is used in security and social media, among other applications.

The eigenface approach to facial recognition relies on concepts from linear algebra. In this approach, images are represented as matrices of pixels, and they are analyzed using singular value decomposition and projection onto eigenvectors. The eigenvectors used in facial recognition are often referred to as *eigenfaces*.

Algorithm

We implemented an eigenface algorithm using the MATLAB language and integrated development environment. An article by Wikipedia on eigenfaces[1] was the main source of the information we used to design this algorithm. The data we used was a set of different images each of 15 different people from the Yale Face Database[2].

```
% Settings
directory = 'faces/';
scale = 0.25; % Set image scaling
subjects = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]; % Which subjects should be used for training?
scenes = [1, 2, 3, 4, 5, 7, 8, 9, 10, 11]; % Which scenes should be used for training?
e = 0.8; % Set a threshold on the total variance
test.subject = 4;
test.scene = 6;
```

The settings in this portion of code allow users to specify the directory in which images are located and the factor by which to scale images. We can also choose which subjects and scenes are used to construct the training data set. Furthermore, this code allows us to set a threshold on total variance for principal component analysis and choose which image will be used for testing the algorithm.

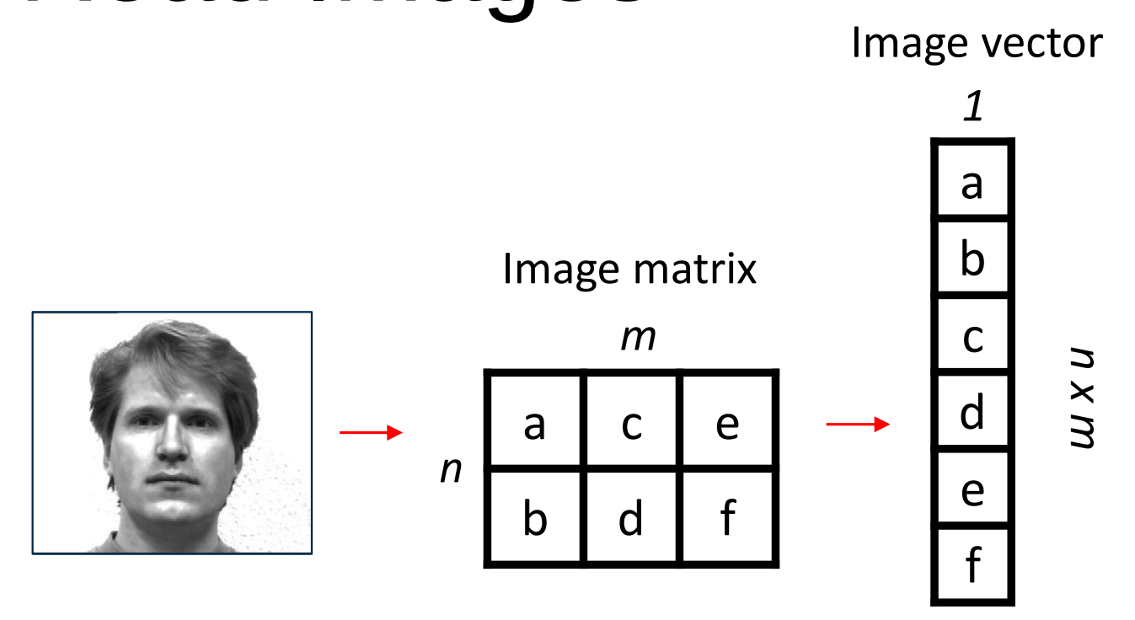
Assumptions

In order to implement this algorithm for facial recognition, we had to make several assumptions about the data being used:

- Faces are already in greyscale image
- Images are in GIF format and are all the same size
- Pictures contain only human faces
- People aren't making extreme facial expressions
- Each person's entire face is contained within the image and are in the same position
- People aren't wearing masks

Training

Read Images



```
% Parameters:
%   directory - A string containing the directory to read files from
%   files - A list of the images available in the source directory
%   scale - The factor by which to scale images
%   scenes - A list of scene indices to read for each subject
%   subjects - A list of which subjects to read images from
%   numSubjects - The number of subjects available in the source directory
% Return Values:
%   images - A 2D matrix containing every image read as a column vector
function [images] = getImages(directory, files, scale, scenes, faces, subjects)
    % Iterate over every subject
    for subject = 1:length(faces)
        % Iterate over every scene
        for scene = 1:length(scenes)
            % Read in the image
            fprintf("Reading '%s'\n", files((faces(subject) - 1) * subjects + scenes(scene)).name);
            imageMatrix = imresize(imread(strcat(directory, files((faces(subject) - 1) * subjects + scenes(scene)).name)), scale);

            % Condense the image into a column vector and save it
            [rows, columns] = size(imageMatrix);
            images(:, (subject - 1) * length(scenes) + scene) = double(reshape(imageMatrix, [(rows * columns), 1]));
        end
    end
end
```

The first step, as in any computer program, is to get the data. We created a function to read images from the database. This function reads the appropriate images as specified by the settings earlier in the code. It is used to construct a matrix containing each of the training images as a column vector.

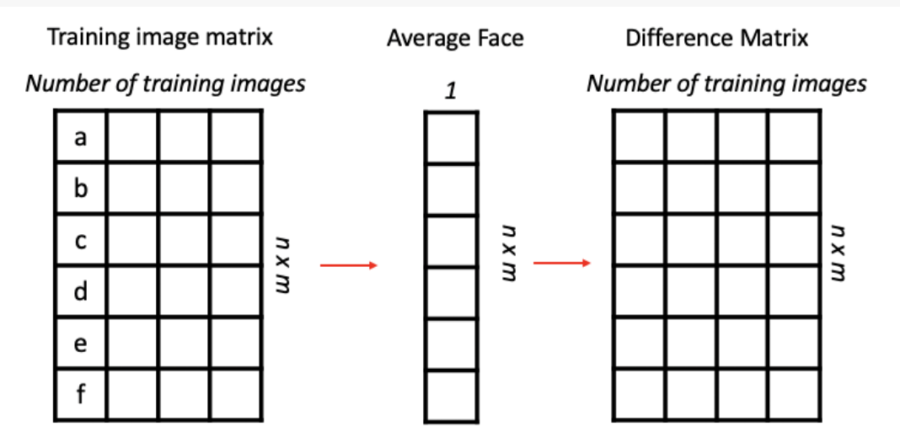
```
% Get a list of all the GIF images in the faces directory
files = dir(strcat(directory, '*.gif'));

% Read in the training images
trainingFaces = getImages(directory, files, scale, scenes, subjects, 11);
```

We call the function to obtain all the images in the directory of faces that have a GIF extension, and filter out any images that were not included in the settings for training subjects and scenes.

Average Face

```
% Compute the average face
averageFace = mean(trainingFaces, 2);
% Subtract the average face from each training face
for image = 1:(length(subjects) * length(scenes))
    trainingFaces(:, image) = trainingFaces(:, image) - averageFace;
end
```

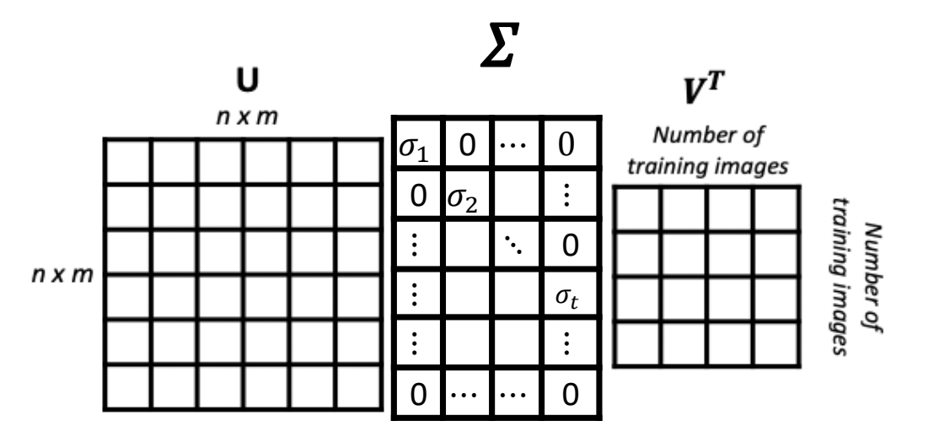
Now that we have a matrix with columns of image vectors, we take the element-wise mean of all the columns to get the *average face*. We then subtract this average face vector from every column In the training image matrix to get a matrix of differences from the average face, where each column vector is the difference between that image and the average face.



Singular Value Decomposition

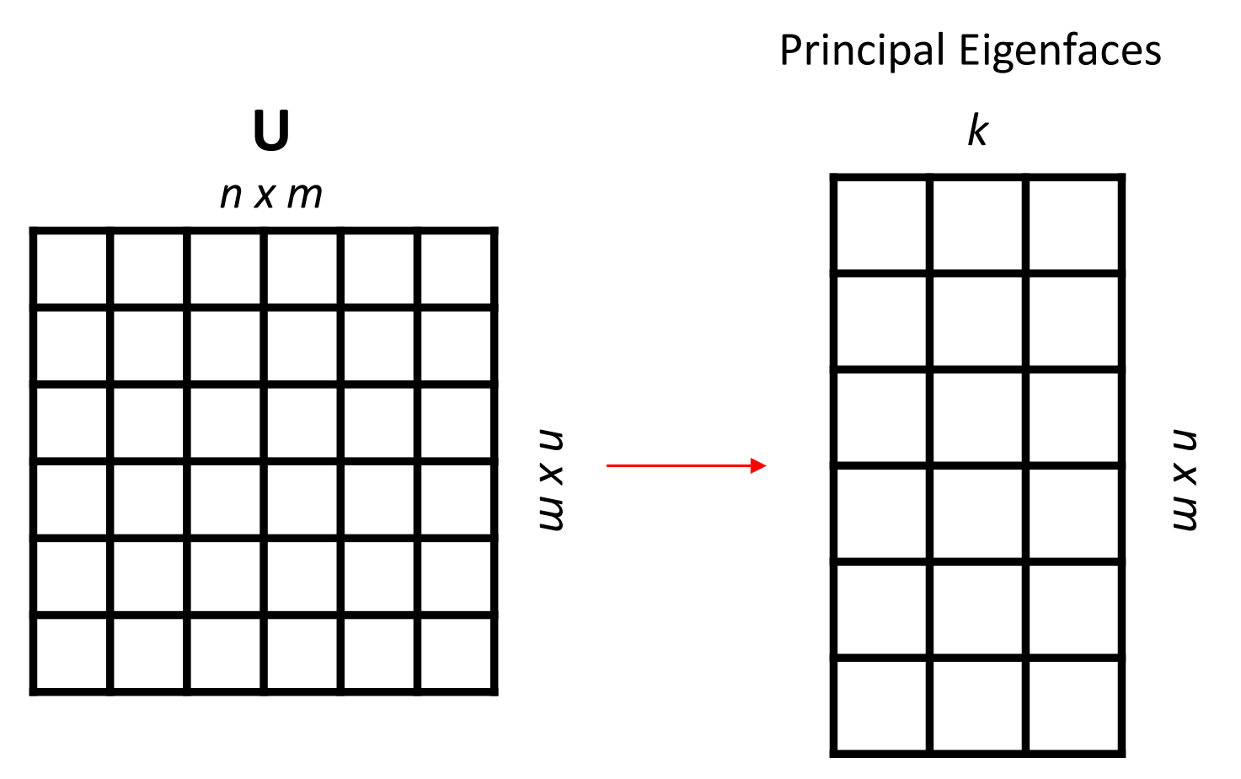
With a matrix containing the differences between each training image and the average face, we now perform *singular value decomposition* (SVD) on the matrix. MATLAB has a handy function designed specifically for SVD. Given a matrix A, it returns three matrices U, Σ, and V, such that $A = U\Sigma V^T$. The matrices U and V are square orthonormal matrices containing singular vectors of A.

```
% Decompose the matrix of training faces
[U, S, V] = svd(trainingFaces);
```



In this case, we are interested in the matrix U, because the vectors in U have the same dimension as the image vectors in our training dataset. The columns of U are *eigenfaces*. In principal, any face can be represented as a combination of eigenfaces. So we use eigenfaces to represent the characteristics of faces in our dataset.

Choose Important Eigenfaces



After the SVD step, we could be left with an abundance of eigenfaces. But to represent a new or existing face, we may not need all the eigenfaces. Therefore, we choose the eigenfaces that are important and discard the rest.

From the matrix Σ, we can square each singular value to obtain the corresponding eigenvalue. The singular values (and therefore the corresponding eigenvalues as well) are listed in descending order,

such that $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_k$ and $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_k$. We set a threshold ε on the total variance (In this case, we found that a threshold around 0.8 was reasonable, and anything much less produced incorrect results).

$$\frac{(\lambda_1 + \lambda_2 + \cdots + \lambda_k)}{(\lambda_1 + \cdots + \lambda_{n \times m})} > \varepsilon$$

Next, we minimize k in the equation above (where $n \times m$ is the number of pixels in an image and the number of eigenfaces after SVD). Then we choose the first k eigenfaces in the matrix U and discard the rest.

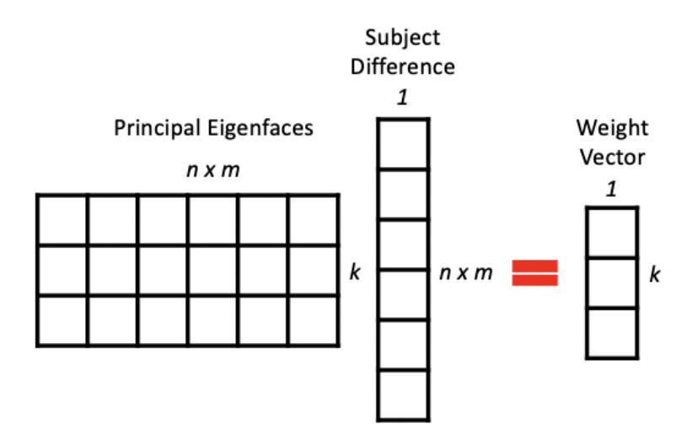
We wrote a function in MATLAB to simplify this step. The function computes the total variance with all eigenvalues and decrements the number of principal components (k) until it finds that the threshold would no longer be satisfied and returns the number of principal components. With this abstraction, we call the function from our driver program.

```
% Get the eigenvalues from the sigma (singular values) matrix
eigenvalues = diag(S).^2;

% Determine the number of important eigenfaces
k = important(eigenvalues, e);
% Shrink the matrix V to contain only important eigenfaces
% Transpose it so we can easily project images onto each eigenface
U = transpose(U(:, 1:k));
```

```
% Parameters:
%   eigenvalues - A 1D vector containing all the eigenvalues found in SVD
%   threshold - The threshold set on the total variance
% Return Values:
%   k - The number of important eigenfaces
function [k] = important(eigenvalues, threshold)
    % Determine the total variance with all components
    v = sum(eigenvalues);

    % Determine the number of important eigenfaces
    for k = length(eigenvalues):-1:1
        % Decrement k while we still satisfy the variance threshold
        if((sum(eigenvalues(1:k)) / v) < threshold)
            % Reset k to the last value that was valid
            k = k + 1;
            break;
        end
    end
end
```

Find Weight Vectors

At this point, we have a collection of the most important (principal) eigenfaces from our training dataset. However, the number of eigenfaces is not necessarily (or probably) the same as the number of people in our set of training images. To simplify the representation of a training image, we subtract the mean face from that image and project the difference onto each eigenface to get the weight of that eigenface. Matrix multiplication of the difference vector and a matrix with eigenfaces as rows will yield a *weight vector*.



```
% Compute the weights of the training images
for subject = 1:length(subjects)
    for scene = 1:length(scenes)
        % Compute the weight vector for this subject/scene
        subjectWeights(:, scene) = U * trainingFaces(:, (subject - 1) * length(scenes) + scene);
    end

    % Average this subject's weight vectors
    trainingAverageWeights(:, subject) = mean(subjectWeights, 2);
end
```

In our code, we computed the weight vector for each training image and then averaged the weight vectors of all the images for each person. This way, we only need to store a single average weight vector for an individual person in the training dataset. This portion of code constructs a matrix where each column is the average weight vector for an individual person in the training dataset.

Testing

In the training process, we picked a small number of images in the Yale database[2] to exclude from our training data set. From these images that were excluded during training, we choose one to test the algorithm. This image contains the face of one of the people in our training dataset, but this unique image was not included in training.

Read Image and Subtract Average Face

After training our model we read in a test image and concatenate the pixels into a column vector, exactly as we did in the training process. The next step is to subtract the average face from the image vector to get the difference vector for our test image.

```
% Read in the test image
testFace = getImages(directory, files, scale, test.scene, test.subject, 11);
```

Test Image Weight Vector

After we processed the training images, we were left with a matrix containing column vectors that represent the average weight vector for each person in the training dataset. Similarly, we compute the weight vector for the test image by projecting it onto the matrix of eigenfaces.

```
% Subtract the average face
testFace = testFace - averageFace;
% Compute the test face's weight vector
testWeight = U * testFace;
```

Now we have a good way to compare the test image with each of the people from our training dataset.

Find the Closest Training Weight Vector

We now have a weight vector for the test image, representing the weights of each eigenface in this image. In the training data, we have the average weight vector for each person in the database. To find which person's face is in the test image, we iterate over all the individuals in the database, and find which person's average weight vector is closest to the weight vector for the test image. As we iterate through the database, we compute the Euclidean distance between the current person's average weight vector and the test image's weight vector. For example, if we had weight vectors **q** and **p** with k elements each, the distance between these vectors would be given by:

$$d = \sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2 + \cdots + (q_k - p_k)^2}$$

Since we are operating under the assumption that the test image contains the face of a person in our training dataset, we conclude that the person in the test image is the person whose average weight vector is closest to that of the test image.

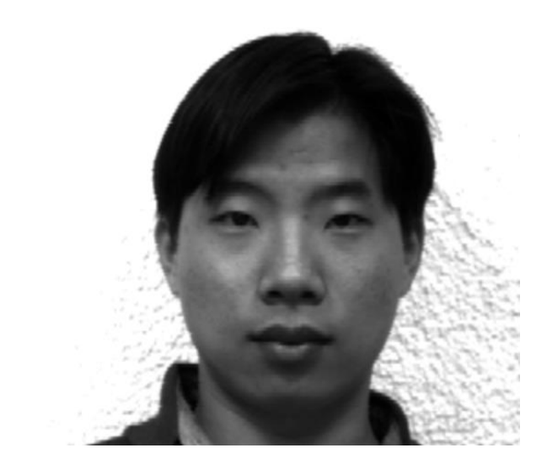
We wrote a function to simplify this part of the algorithm. It returns the index in the list of subjects of the person whose face is in the test image.

```
% Parameters:
%   subjects - The number of subjects in the training dataset
%   scenes - The number of scenes in the training dataset
%   testWeight - The weight vector of the test image
%   trainingWeights - A 2D matrix with each subject's average weight vector
%                     as a column vector
function [closest] = identify(subjects, scenes, testWeight, trainingWeights)
    % Assume the largest distance possible
    shortestDistance = realmax;
    % Find the closest weight vector in the database
    for subject = 1:subjects
            % Get the Euclidean distance between the current training
            % image's weight vector and the test image's weight vector
            distance = norm(testWeight - trainingWeights(:, subject));

            % Update the closest weight vector
            if distance < shortestDistance
                shortestDistance = distance;
                closest = subject;
            end
    end
end
```

```
% Find the closest weight vector in the database
closest = subjects(identify(length(subjects), length(scenes), testWeight, trainingAverageWeights));
% Print the result
fprintf("The test face was identified as subject %d.\n", closest);
```

Results

The algorithm we implemented works with a variety of training/test image sets from the Yale database[2]. Here, we show the test with the settings written in code in the beginning of the Algorithm section. Trained on 10 different images of each of the 15 people on the left, this code correctly identified the image on the right.



References

[1] "Eigenface." *Wikipedia*, Wikimedia Foundation, 1 Apr. 2019, en.wikipedia.org/wiki/Eigenface.

[2] *MIT Media Lab: VisMod Group*, vismod.media.mit.edu/vismod/classes/mas622-00/datasets/.