Computer Vision (CE316 and CE866): Faces

Adrian F. Clark
⟨alien@essex.ac.uk⟩

Write the abostract!

Contents

1 Introduction **2**

2 Locating Faces by Colour **2**

3 Viola-Jones: Haar Features and Adaptive Boosting **2**

4 Processing Face Images **6**

5 Recognising Faces: Eigenfaces **7**

List of Figures

1 Successful detection of human skin by colour 2
2 Skin detection by colour is easily confused 2
3 Haar features 3
4 Haar feature that helps detect eyes 3
5 Principle of an integral image 4
6 Viola-Jones face location in action 5
7 Face dimensions that accord to the golden ratio are supposed to be more attractive 6
8 Model-based image coding 7
9 Illustration of principal component analysis 8
10 Some eigenfaces 10
11 The six ‘universal’ expressions 11

1 Introduction

Faces are interesting. Studies suggest that there is special 'circuitry' in the brain to locate faces in images, store them, and recognise them — that is one of the reasons we are able to 'see' faces in phenomena such as clouds, tea leaves, and so on. In this chapter, we shall consider how faces are located in images (including a dead end), and then take a look at one of the techniques that attempts to do face recognition. Largely for amusement, we shall also look briefly at assessing the beauty of faces and a cute approach to communicating people speaking.

There are several steps in processing images of faces, and getting to grips with their nomenclature is a good first step. Processing an image to identify where faces are to be found is known, not unreasonably, as *face location* or *face detection*. Having found a face, a common requirement is to scale it so that the eyes and mouth appear at known locations within the image; this is known as *face normalisation*. Finally, determining who a particular face belongs to is *face recognition*.

2 Locating Faces by Colour

One way to identify where a face lies in an image is by its colour. We know that the colour of skin is determined by a compound called *melanin*: the more of this that is present, the better the skin protects against ultra-violet radiation and the darker it appears. So, the argument goes, finding skin-coloured regions in images and determining whether they are roughly oval is one way of identifying faces.

Figure 1: Successful detection of human skin by colour (hue 300°–30°, saturation 30%–70%)

As the amount of melanin differs from person to person, we cannot look for a little region in the RGB colour space. However, if we convert the image to the HSV colour space, the fact that melanin is a specific colour should allow skin to be recognised by looking in a small range of hues and saturations. Implementing this is straightforward (see the routine `find_skin` in EVE), and Figure 1 shows that it can indeed find regions of skin.

However, there are major problems with this approach. Firstly, it clearly cannot work on monochrome ("black and white") images, while human vision obviously does. Secondly, the colour that skin appears is affected by the colour of its illumination, so the hue–saturation region changes as one moves from (say) daylight to fluorescent light. Thirdly and most crucially, there are compounds other than melanin that have similar colouring; in particular, some types of wood have rather similar hues to skin — Figure 2 gives an example failure. We are forced to conclude that using colour as a way of locating faces is far too naïve to be used in practice.

Figure 2: Skin by colour is easily confused; wood, for example, has a similar hue and saturation to skin (hue 300°–30°, saturation 30%–70%)

3 Viola-Jones: Haar Features and Adaptive Boosting

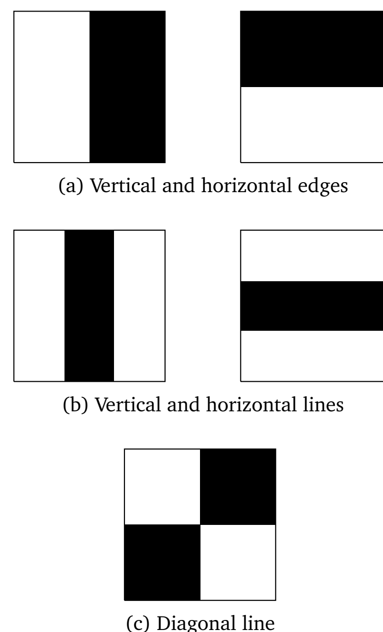
The face location technique due to Viola and Jones [Viola and Jones, 2004] is the *de facto* standard algorithm, present in practically every mobile 'phone and digital camera; the algorithm is patented, so anyone who uses it in commercial product has to pay a licence fee to Mitsubishi. As

we shall see, the approach that Viola and Jones developed is actually general, applicable to other types of feature detection and recognition. It is implemented in OpenCV in a way that makes performing face recognition straightforward, yet retains the ability to recognise other types of object.

You should be aware that the face detection technique was developed specifically for camera manufacturers so that cameras could auto-focus on faces, which are the parts of photographs that people normally want to be sharp. Hence, the algorithm works only for full-face images; it does not work for profile views and performs fairly poorly for three-quarter views. As we shall see, the algorithm involves processing rectangular regions, and these will differ in landscape and portrait orientations — this is why cameras that do face recognition always have orientation sensors.

The most important thing to realise about this technique is that it involves two phases. Firstly, a series of classifiers are trained how to detect faces using a database of positive (images containing faces at known locations) and negative (images without faces) examples, a very slow process. When trained, these classifiers can be applied to any image to locate any faces that are in it, and this executes quickly. The slow learning is not a problem as, in principle, it need only ever be done once; but it is important that the trained classifiers execute quickly.

To train the classifiers, features need to be extracted from images. The approach that Viola and Jones took was to reduce any input image region down to a fixed size (they used 24×24 pixels). Then, all possible Haar features are found from this image — even for such small images, there are 162,336 possible features, and this is one of the reasons that the learning algorithm is slow. A Haar feature is actually quite simple: it is the difference of the sum of pixels in rectangularly-shaped image regions. Figure 3 shows some example Haar features that correspond to edges and lines.



(a) Vertical and horizontal edges

(b) Vertical and horizontal lines

(c) Diagonal line

Figure 3: Some examples of Haar features. In each case, the feature is a single value calculated by subtracting the sum of pixels lying below the white rectangle from the sum of pixels lying below the black rectangle.

Most of the features calculated will be irrelevant but a few will help locate the face; for example, Figure 4 shows one that should help identify eyes. The learning algorithm finds which combination of features minimises the number of mis-classifications of face and non-face image regions. Each feature individually does not perform particularly well (it is a *weak classifier* in machine learning jargon) but all the weak classifiers in combination may yield a strong classifier. This procedure is known as *adaptive boosting* or *Adaboost*; it is actually a bit more complicated than this in detail but this is its principle. Viola and Jones found that about 200 of these features classified faces correctly with about 95% accuracy, while about 6,000 classified all of them.

Figure 4: Haar feature that helps detect eyes

Although the time taken to calculate 6,000 features is substantially less than the time required for 160,000, the process can still be made more efficient. Most of a typical image will not contain a face, so a quick test to ascertain whether a region definitely does not contain a face means that the other Haar features do not have to be calculated; for example, completely uniform regions are not faces. Only when an image region might potentially contain a face is it worth calculating some of the other features. Hence, Viola and Jones introduced the idea of a *cascade of classifiers*, with images potentially being rejected at each step along the cascade. Their

final face detection system contained 38 stages, the first five of which involve only 1, 10, 25, 25 and 50 features; the total number of features required by all 38 stages is a little over 6,000.

Speeding up Haar features

The computation of the rectangular regions used to form the Haar features can be speeded up significantly. This is done using a data structure known in the computer vision domain as an *integral image*; however the data structure actually arose for performing texture-mapping in computer graphics, where it is known as a *summed-area table*. Whatever its name, it is based on the algebraic identity

$$ab = (x+a)(y+b) - (x+a)y - x(y+b) + xy \qquad (1)$$

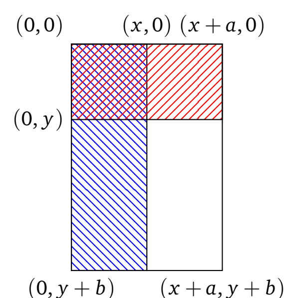


Figure 5: Principle of an integral image: the unshaded rectangle is found by subtracting the areas of the red and blue hatched rectangles from the outer rectangle, then adding in the cross-hatched rectangle

As ab is the area of a rectangle with sides a and b, this can be interpreted geometrically as shown in Figure 5. The area of the unshaded rectangle ab can be found by taking the area of the outer enclosing rectangle and subtracting from it the areas of the red and blue hatched regions. This means that the cross-hatched region has been subtracted twice, so it must then be added back in.

As described so far, this procedure involves more computation than just adding up all the relevant pixels. The speed-up comes from converting the image into the summed-area table or integral image data structure, so that its pixel value at (x, y) is the sum of all pixels in the original image above and to the left of it. If we call this integral image I, then the sum of all pixels in the rectangle with corners (x, y) and $(x+a, y+b)$ is

$$I(x+a, y+b) - I(x+a, y) - I(x, y+b) + I(x, y) \qquad (2)$$

which involves precisely four additions or subtractions. This means that the sum of any rectangular region of an image, *irrespective of its size*, can be calculated in constant time. This is a valuable property for any component of a real-time system, and has led to integral images appearing in a number of other vision operators, *e.g.* SURF.

Coding up the calculation of an integral image is quite straightforward, as the following code demonstrates:

```
def calc_sat (im):
    "Form an integral image from an image."
    ny, nx, nc = im.shape
    sat = numpy.zeros ((ny,nx,1))
    sat[0,0] = im[0,0]
    for x in range (1,nx):
        sat[0,x] = im[0,x] + sat[0,x-1]
    for y in range (1, ny):
        sat[y,0] = im[y,0] + sat[y-1,0]
        for x in range (1, nx):
            sat[y,x] = im[y,x] + sat[y,x-1] + sat[y-1,x] - sat[y-1,x-1]
    return sat
```

Having formed the integral image, calculating the area of a rectangular region of the original image can be encapsulated in a short routine. This is regrettably not a single line of code because the way that Python indexes arrays does not perfectly match (2).

```
def get_sat (sat, ylo, yhi, xlo, xhi):
    """Return the area of a rectangular region of an image from its
    integral image."""
    # We have to do a little fiddling around with indices because the way
    # Python indices does not quite match how an integral image is most
    # naturally indexed...so it is not quite constant in time.  Sigh.
    ylo -= 1
    xlo -= 1
    if ylo < 0:
        if xlo < 0:
           res = sat[yhi,xhi]
        else:
           res = sat[yhi,xhi] - say[yhi,xlo]
    elif xlo < 0:
        res = sat[yhi,xhi] - sat[ylo,xhi]
    else:
        res = sat[yhi,xhi,0] + sat[ylo,xlo,0] - sat[yhi,xlo,0] - sat[ylo,xhi,0]
    return res
```

The following short program shows how the routines are used:

```
# Create and fill an image.
im = eve.image ((10,10,1))
eve.set (im, 1)
# Work out the correct sum of a rectangular region of it.
correct = im[4:6,1:3].sum()
# Form the integral image.
sat = calc_sat (im)
# Print out the correct sum and the value obtained from
# the integral image.
print correct, get_sat (sat, 4, 5, 1, 2)
```

Using Viola-Jones in OpenCV

Viola-Jones feature detection is built into OpenCV and is fairly straightforward to use. You have to give it the Haar cascades that it is to use when identifying whatever type of object that is to be detected; this is done by means of an XML file. Files are supplied with OpenCV for identifying full faces; the code below uses one of them. There are also cascades for detecting (left) eyes, smiles. . . and Russian number plates. There are loads of cascades produced by other people on the Web.

The following code is a self-contained face detection routine, though it could be made more efficient. An example of a face detected by this routine is shown in Figure 6.

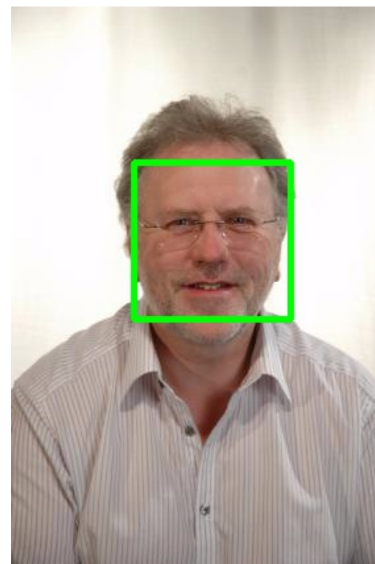


Figure 6: Viola-Jones face location in action

```
def detect (image):
    '''Detect a face using Haar cascades, as in Viola-Jones'''
    size = cv.GetSize (image)
    # Create a grey-scale version of the image, then use the Haar cascades
    # to detect faces.
    grayscale = cv.CreateImage (size, 8, 1)
    cv.CvtColor (image, grayscale, cv.CV_BGR2GRAY)
    storage = cv.CreateMemStorage (0)   # temporary storage
    cascade = cv.Load ('haarcascade_frontalface_alt.xml')
    faces = cv.HaarDetectObjects (grayscale, cascade, storage, 1.2, 2,
                                  cv.CV_HAAR_DO_CANNY_PRUNING)
```

```
# Outline any faces found with a green rectangle.
if faces:
    for i in faces:
        cv.Rectangle(image,
                     (i[0][0], i[0][1]),
                     (i[0][0] + i[0][2], i[0][1] + i[0][3]),
                     (0, 255, 0), 3, 8, 0)
```

4 Processing Face Images

Measuring 'attractiveness'

'Beauty,' the old adage goes, 'is in the eye of the beholder' — meaning that a face found attractive by one person may not be by others. Nevertheless, psychologists and others have found some evidence that certain face shapes and arrangements of features tend to be considered as more attractive than others, and this is supposed to be the case irrespective of gender and culture. There are two characteristics of attractive faces that are easily amenable to computer analysis and we shall consider them briefly.

Firstly, the more symmetric a face appears, the more attractive it is supposed to be. The difficulty here is identifying the axis of symmetry of a face, and one approach is to identify the locations of the eyes, nose and mouth and place the axis of symmetry mid-way between them. The degree of symmetry can be estimated by reflecting one half of the face and correlating it with the other half — though illumination needs to be fairly frontal for this to work.

Secondly, humans apparently find that things arranged according to the so-called *golden ratio*, ϕ, are pleasing. For $a > b > 0$, this is

$$\phi \equiv \frac{a+b}{a} = \frac{a}{b} \tag{3}$$

In other words, two quantities are in the golden ratio if their ratio is the same as the ratio of their sum to the larger of the two quantities. A bit of algebra will let you determine that

$$\phi = \frac{1+\sqrt{5}}{2} = 1.6180339887\cdots \tag{4}$$

(It is thought to be an irrational number.) Faces whose overall shape are in the ratio $\phi : 1$ are supposed to be the most attractive — see Figure 7 for an example. The same idea is supposed to apply for the positions of features within the face: for example, if the eyes are placed at the golden ratio position within the height of the face, it is supposed to be the most attractive position. There are various software offerings that perform this 'assessment' on the Internet; Figure 7 is taken from the website of one of them.

Clearly, these measures should be regarded only as an amusement. If your own face or the face of a loved one is neither particularly symmetrical nor has their features arranged according to the golden ratio, don't despair: Sophia Loren is reputed to be one of the most beautiful women

(a) Original image

(b) Superimposed mask

(c) "More beautiful" version according to the golden ratio

Figure 7: Face dimensions that accord to the golden ratio are supposed to be more attractive (from http://www.goldennumber.net/beauty/)

ever born and her face apparently scores poorly according to these criteria. In any case, remember that there is another old adage that 'beauty is more than skin deep.'

Model-based video coding

We have seen how it is possible to locate faces and facial features in images. With this capability, we can follow the motion of facial features from frame to frame of a video sequence and infer what the motion of a speaker's head is: for example, if both eyes are moving horizontally, the head must be rotating and, as the diameter of the head can be estimated from the imagery, we can work out what the angle of rotation must be.

With this information, Münevver Köküer and the author were able to do video-telephony at *extremely* low data rates. Rather than sending the actual video, we sent only the rotation angles *etc*. that the head is undergoing and then animated a 3D head model at the receiver — in our work, we used the Candide model discussed in the notes on stereo vision. The appearance of the 3D model can be improved by texture-mapping the subject's face onto the 3D model. Although this may seem a little bizarre, we were able to produce a system in the early 1990s that was accurate enough for deaf people to lip-read the animated model!

(a) Original image

(b) Coded image

Figure 8: Model-based image coding. By analysing the motion of a human head in 2D over the frames of a video, it is possible to infer the motion in 3D from (a) and use that to animate a 3D model, as shown in (b).

5 Recognising Faces: Eigenfaces

You will recall that the standard deviation (or its square, the variance) gives us a single number that describes the variation present in an image. It would be useful if we could find a similarly straightforward formula that encapsulates the *similarity between two images* in a single number. An inspection of the equation defining the variance suggests something. Given two images, $P(x, y)$ and $Q(x, y)$, let us calculate

$$\mathrm{Cov}(P,Q) = \frac{1}{MN} \sum_{x,y} \left(P(x,y) - \langle P \rangle\right) \left(Q(x,y) - \langle Q \rangle\right). \tag{5}$$

where $\langle \cdot \rangle$ represents the mean value. If $P(x, y) > \langle P \rangle$ and $Q(x, y) > \langle Q \rangle$, we will get a positive contribution to $\mathrm{Cov}(P,Q)$; likewise, we will get a positive contribution to $\mathrm{Cov}(P,Q)$ when $P(x, y) < \langle P \rangle$ and $Q(x, y) < \langle Q \rangle$. Conversely, when $P(x, y) - \langle P \rangle$ and $Q(x, y) - \langle Q \rangle$ have opposite signs, we will get a negative contribution to $\mathrm{Cov}(P,Q)$. So $\mathrm{Cov}(P,Q)$ is actually a fairly good measure of how well P and Q vary 'in step' with each other. Because of this and how similar it is to the variance, $\mathrm{Cov}(P,Q)$ is known as the *covariance* of P and Q. It is easy to show that if P and Q are uncorrelated or independent, then $\mathrm{Cov}(P,Q) \approx 0$.

Looking back in the notes from earlier lectures at the definition of the correlation coefficient, you can now see that it is the same as

$$r = \frac{\mathrm{Cov}(P,Q)}{\sqrt{\sigma_P^2 \, \sigma_Q^2}} \tag{6}$$

In other words, the correlation coefficient is nothing more than a suitably normalized covariance.

Covariance is useful if we have two images but what if we have more than two? The normal approach is to form a matrix of covariances; if we have N images $P_1, P_2, \dots P_N$, the covariance matrix $\mathbf{C}$ is

$$\begin{pmatrix} \text{Cov}(P_1, P_1) & \text{Cov}(P_1, P_2) & \cdots & \text{Cov}(P_1, P_N) \\ \text{Cov}(P_2, P_1) & \text{Cov}(P_2, P_2) & \cdots & \text{Cov}(P_2, P_N) \\ \vdots & \vdots & \ddots & \vdots \\ \text{Cov}(P_N, P_1) & \text{Cov}(P_N, P_2) & \cdots & \text{Cov}(P_N, P_N) \end{pmatrix} \quad (7)$$

In other words, each element represents the covariance between a pair of images. Diagonal elements, $\text{Cov}(P_i, P_i)$, are of course simply variances. A few moments' thought will tell you that all elements of the covariance matrix are non-negative and that the matrix is symmetric, as $\text{Cov}(P_i, P_j) = \text{Cov}(P_j, P_i)$.

For any set of images that are related in some way, for example different images of the same scene or a set of images of people's faces, off-diagonal elements are non-zero as there is usually some correlation (and hence covariance) between images. If we can manipulate the covariance matrix to remove this correlation, we can 'concentrate' the variation in a set of images into just a few images...and this leads us to the concept of *principal component analysis* (PCA), a powerful analysis technique for sets of related images which involves a transformation of the covariance matrix. The technique is also known as the Hotelling transform and the Karhünen-Loève transform; the latter name is often used in, for example remote sensing.

It is possible to show that any real, symmetric, non-singular matrix $\mathbf{C}$ can be decomposed into the form

$$\mathbf{CK} = \lambda \mathbf{K} \quad (8)$$

where the rows of $\mathbf{K}$ are known as the *eigenvectors* of $\mathbf{C}$ and $\lambda = (\lambda_1, \lambda_2, \dots \lambda_N)^T$ the corresponding *eigenvalues*. These eigenvectors are necessarily orthonormal; if $\mathbf{A}$ and $\mathbf{B}$ are two eigenvectors, then $|\mathbf{A}|^2 = |\mathbf{B}|^2 = 1$ and

$$\mathbf{A.B} = A_1 B_1 + A_2 B_2 + \cdots + A_n B_N = 0. \quad (9)$$

The matrix $\mathbf{K}$ is the kernel of the PCA, which is evaluated using

$$\mathbf{Y} = \mathbf{K}(\mathbf{P} - \langle \mathbf{P} \rangle) \quad (10)$$

Applying $\mathbf{K}$ to data analyses it in terms of the principal components.

If we calculate $\mathbf{KCK}^T$, we end up with a matrix where the diagonal elements are the eigenvalues and the off-diagonal elements are zero:

$$\begin{pmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \lambda_N \end{pmatrix} \quad (11)$$

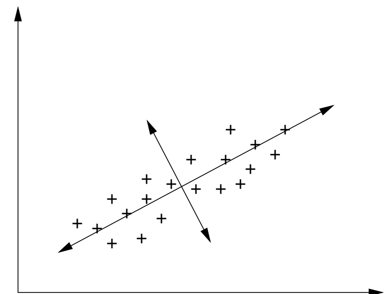


Figure 9: Illustration of principal component analysis

This mathematical analysis is fairly abstract until one gives it a physical interpretation (Figure 9). If we have a cloud of points in an N-dimensional space, what PCA does is as follows. Firstly, it identifies the direction of

maximum variation; this direction becomes the first principal component (eigenvector) and the corresponding variance its eigenvalue. The second principal component must be perpendicular to the first one; in Figure 9 this forces to lie in the direction shown but in three dimensions it could be anywhere on a circle perpendicular to the first principal component. The particular direction is again chosen to maximize the variance. The same approach is used for the remaining components.

It is normally found that the first few eigenvalues are much larger than the remainder. As these correspond to the variances (*i.e.,* importances or signals) of the various principal components, they show how the useful variation has been compressed into the first few 'eigen images' while the remainder contain much less significant variation (and often simply noise).

So why is PCA useful? Given an arbitrary image $R(x, y)$ which was part of the set of images used to determine the PCA, we can represent R as a weighted combination of the various eigen images:

$$R(x, y) = r_1 Y_1(x, y) + r_2 Y_2(x, y) + \cdots + r_n Y_n(x, y) \tag{12}$$

In other words, $r_1, r_2, \ldots r_N$ is a unique *feature vector* for the image $R(x, y)$ in terms of the principal components. For an image $R'(x, y)$ which was not part of the set of images underwent eigen decomposition, the above relationship is usually approximately true. . . and this means it is possible to represent any image in terms of its feature vector. At this point, we can apply any clustering technique to identify groups in PCA space, and these correspond to similar images. Moreover, the various principal components represent the most important modes of variation.

A *caveat* is in order here, however. If an image is found which is significantly different to those that formed the set that underwent eigen decomposition, it will not be represented well as a linear combination of the principal components. Put another way, PCA is good at interpolating between the various principal components but less good at extrapolating. Hence, it is important that the set of images used to obtain the principal components exhibits all of the possible modes of variation that are able to occur in practice.

With all this under our belt, we can now see how it applies to face recognition. The basic idea is that one takes a set of training images of faces, performs PCA on them, and then for any other face applies the kernel to it to obtain its feature vector in terms of the 'eigenface' images. The hope is that images of each individual will produce a unique cluster in feature space.

However, there are several things that need to be done for this approach to work well. The first is that one needs to pre-process all the images so that images to be transformed are the same size with the eyes and mouth appearing at the same pixel locations — this is the *face normalization* process that was alluded to in the introduction to this chapter. If this is not done, the most significant principal component images encode this misalignment rather than any variation that is useful for face recognition (*e.g.,* nose shape).

The second important point concerns how PCA is applied. Normally in

PCA, the number of samples needs to be much larger than the size of the resulting feature vector (we are, after all, taking averages). If we want to treat each pixel in an image as a separate thing that may vary in PCA space then even small images end up with huge covariance matrices: for example, a 100×100-pixel image results in a 10^4-dimensional feature space and a covariance matrix of $10^4 \times 10^4 = 10^8$ elements! However, most elements of this huge covariance matrix will be zero: if there are N training examples, there will be at most $N - 1$ eigenvectors with non-zero eigenvalues.

Kirby and Sirovich were the first to find a way around this problem [Kirby and Sirovich, 1990], though they were interested in image coding rather than face recognition. They realised that if the number of training examples is smaller than the dimensionality of the images, the principal components can be computed more easily. If $\mathbf{T}$ is a matrix of pre-processed training examples, each row of which contains a mean-subtracted image, the covariance matrix can be computed as $\mathbf{C} = \mathbf{T}\mathbf{T}^T$ and the eigen decomposition of $\mathbf{C}$ is given by

$$\mathbf{C}\mathbf{v}_i = \mathbf{T}^T\mathbf{T}\mathbf{v}_i = \lambda\mathbf{v}_i \tag{13}$$

However, $\mathbf{T}^T\mathbf{T}$ is the huge matrix alluded to above; if we perform the eigen decomposition of

$$\mathbf{T}\mathbf{T}^T\mathbf{u}_i = \lambda\mathbf{u}_i \tag{14}$$

then we notice that pre-multiplying both sides of the equation by $\mathbf{T}^T$ gives us

$$\mathbf{T}^T\mathbf{T}\mathbf{T}^T\mathbf{u}_i = \lambda\mathbf{u}_i. \tag{15}$$

So if $\mathbf{u}_i$ is an eigenvector of $\mathbf{T}\mathbf{T}^T$, then $\mathbf{v}_i = \mathbf{T}^T\mathbf{u}_i$ is an eigenvector of $\mathbf{C}$. If we have a training set of 300 images, each 100×100 pixels, the matrix $\mathbf{T}\mathbf{T}^T$ is 300×300 rather than $10^4 \times 10^4$, much more manageable. (However, note that the $\mathbf{v}_i$ are not normalized; that must be done separately.) After Kirby and Sirovich realised this, Turk and Pentland [1991] quickly came up with their 'eigenfaces' scheme, and that inspired the vast amount of subsequent research into face recognition.

Figure 10: Some eigenfaces. Top left is the mean face image, and the other two images on the top row are the first two principal components ("eigenfaces"). The bottom row shows the three smallest principal components: these supposedly contain the least amount of information. [*Scholarpedia article on eigenfaces* 2009]

To illustrate this, consider Figure 10 (taken from the Scholarpedia article on eigenfaces), which is a PCA decomposition of a well-known face database. The leftmost in the first row is the mean face, while the other two images on the top row are the top two eigenfaces; the second row shows eigenfaces with the three smallest eigenvalues. It is apparent that although the eyes have been aligned well (they do not appear blurred in the mean face), there is still significant variation around the eye region in the least significant principal components. The first principal component appears to encode the variation between asian and european face shapes; however, if this is the case, it is purely a consequence of the faces that form the database.

You might be pleased to learn that the eigenfaces approach has been implemented in OpenCV, and a web search will quickly turn up the documentation and example programs. There are also implementations of related techniques, for example the use of linear discriminant analysis to yield so-called "Fisherfaces" rather than eigenfaces — Fisherfaces is almost always better at recognition than eigenfaces. There are theoretical

and practical advantages to some of these alternative techniques but exploring them would regrettably take us beyond the scope of this lecture course.

Expression recognition and affective computing

Having obtained some idea of how eigenfaces works, it is interesting to consider some other applications of the approach. There are many, as PCA is a general technique which is used in many subject domains; in vision, one of the more interesting is expression recognition.

We all know when a person is smiling — in fact, human analysis is usually good enough to ascertain whether or not a smile is genuine, a subtlety well beyond computer analysis. If software were able to identify when a computer user is happy by recognising him or her smile, then in principle one could make the software adapt to this — and similarly with other emotions. This idea of having software adapt to the emotional state of its user is known as *affective computing* [Picard, 2000]. Of course, there are other ways that emotion can be measured: skin conductivity (as in a lie detector), alpha and other signals in brain–computer interfaces, jauntiness of walking as measured by accelerometers, and so on.

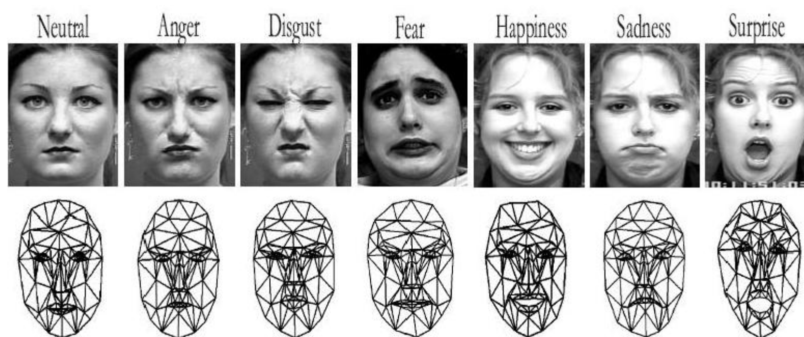


Figure 11: The six 'universal' expressions (from http://www.eecs.qmul.ac.uk/~ioannisp/ralis.htm)

Most expression work focusses on the so-called 'universal' emotions, illustrated in Figure 11 — though there is not universal agreement on them, several researchers adding 'contempt' to those shown in the figure.

Expression recognition works in essentially the same way as face recognition: a training set of images of each expression is captured and its principal components found. An image whose expression is to be determined is processed by the same PCA kernel and its best match identifies the expression class to which it belongs.

Some reasonably impressive results have been reported in the literature, though this author treats them with some caution because the datasets he has examined seem to have ludicrously exaggerated expressions. Perhaps no-one he has seen has been quite as surprised as the people pictured in the datasets, who presumably have just been told that expression recognition works... However, please do take a look at the online resources and decide for yourself.

As a footnote to this brief foray into affective computing, it is worth noting that the vast majority of published work is expended towards recognising emotional state; there seems to be little, if any, work towards

adapting interactions based on knowledge of the perceived state — a good topic for a PhD, perhaps?

References

Kirby, M. and L. Sirovich (1990). "Application of the Karhünen-Loève procedure for the characterization of human faces". In: *IEEE Transactions on Pattern analysis and Machine Intelligence* 12.1, pp. 103–108.

Picard, Rosalind W. (2000). *Affective Computing*. MIT Press.

Scholarpedia article on eigenfaces (2009). `http://www.scholarpedia.org/article/Eigenfaces`.

Turk, M. and A. Pentland (1991). "Eigenfaces for recognition". In: *Journal of Cognitive Neuroscience* 3.1, pp. 71–86.

Viola, Paul and Michael J. Jones (2004). "Robust Real-Time Face Detection". In: *International Journal of Computer Vision* 57.2, pp. 137–154.