Introduction to Face Detection and Face Recognition:

Last updated on 4th Feb, 2012 by Shervin Emami. Posted originally on 2nd June, 2010.

NOTE: Spanish translation is available, thanks to Maria Ramos from Webhostinghub.com.

NOTE: This page was written in 2010 and describes Face Detection & Face Recognition using OpenCV's old C interface. But in June 2012, OpenCV v2.4.1 came with "FaceRecognizer" functionality. FaceRecognizer is much easier to use & understand than this old code. I have written a tutorial & free project using the new FaceRecognizer in Chapter 8 of the book "Mastering OpenCV". My new code also does better image preprocessing & rotation than the code on this page, but the book code doesn't have an offline recognition mode. You should read the official FaceRecognizer tutorials or use my new code instead of this page, and read Chapter 8 of the Mastering OpenCV book if you want to understand how it works.

"Face Recognition" is a very active area in the Computer Vision and Biometrics fields, as it has been studied vigorously for 25 years and is finally producing applications in security, robotics, human-computer-interfaces, digital cameras, games and entertainment.

"Face Recognition" generally involves two stages:

1. **Face Detection**, where a photo is searched to find any face (shown here as a green rectangle), then image processing cleans up the facial image for easier recognition.
2. **Face Recognition**, where that detected and processed face is compared to a database of known faces, to decide who that person is (shown here as red text).

Since 2002, Face Detection can be performed fairly reliably such as with OpenCV's Face Detector, working in roughly 90-95% of clear photos of a person looking forward at the camera. It is usually harder to detect a person's face when they are viewed from the side or at an angle, and sometimes this requires 3D Head Pose Estimation. It can also be very difficult to detect a person's face if the photo is not very bright, or if part of the face is brighter than another or has shadows or is blurry or wearing glasses, etc.

However, Face Recognition is much less reliable than Face Detection, generally 30-70% accurate. Face Recognition has been a strong field of research since the 1990s, but is still far from reliable, and more techniques are being invented each year such as the ones listed at the bottom of this page (Alternatives to Eigenfaces such as 3D face recognition or recognition from video).

I will show you how to use Eigenfaces (also called "Principal Component Analysis" or PCA), a simple and popular method of 2D Face Recognition from a photo, as

opposed to other common methods such as Neural Networks or Fisher Faces.

To learn the theory of how Eigenface works, you should read Face Recognition With Eigenface from Servo Magazine (April 2007), and perhaps the mathematical algorithm.

First I will explain how to implement Eigenfaces for offline training from the command-line, based on the Servo Magazine tutorial and source-code (May 2007).

Once I have explained to you how offline training and offline face recognition works from the command-line, I will explain how this can be extended to online training directly from a webcam in realtime :-)

How to detect a face using OpenCV's Face Detector:

As mentioned above, the first stage in Face Recognition is Face Detection. The OpenCV library makes it fairly easy to detect a frontal face in an image using its Haar Cascade Face Detector (also known as the Viola-Jones method).

The function "cvHaarDetectObjects" in OpenCV performs the actual face detection, but the function is a bit tedious to use directly, so it is easiest to use this wrapper function:

```
// Perform face detection on the input image, using the given Haar Cascade.
// Returns a rectangle for the detected region in the given image.
CvRect detectFaceInImage(IplImage *inputImg, CvHaarClassifierCascade* cascade)
{
	// Smallest face size.
	CvSize minFeatureSize = cvSize(20, 20);
	// Only search for 1 face.
	int flags = CV_HAAR_FIND_BIGGEST_OBJECT | CV_HAAR_DO_ROUGH_SEARCH;
	// How detailed should the search be.
	float search_scale_factor = 1.1f;
	IplImage *detectImg;
	IplImage *greyImg = 0;
	CvMemStorage* storage;
	CvRect rc;
	double t;
	CvSeq* rects;
	CvSize size;
	int i, ms, nFaces;

	storage = cvCreateMemStorage(0);
	cvClearMemStorage( storage );


	// If the image is color, use a greyscale copy of the image.
	detectImg = (IplImage*)inputImg;
	if (inputImg->nChannels > 1) {
		size = cvSize(inputImg->width, inputImg->height);
		greyImg = cvCreateImage(size, IPL_DEPTH_8U, 1 );
		cvCvtColor( inputImg, greyImg, CV_BGR2GRAY );
		detectImg = greyImg;	// Use the greyscale image.
	}

	// Detect all the faces in the greyscale image.
	t = (double)cvGetTickCount();
	rects = cvHaarDetectObjects( detectImg, cascade, storage,
			search_scale_factor, 3, flags, minFeatureSize);
	t = (double)cvGetTickCount() - t;
	ms = cvRound( t / ((double)cvGetTickFrequency() * 1000.0) );
	nFaces = rects->total;
	printf("Face Detection took %d ms and found %d objects\n", ms, nFaces);

	// Get the first detected face (the biggest).
	if (nFaces > 0)
		rc = *(CvRect*)cvGetSeqElem( rects, 0 );
	else
		rc = cvRect(-1,-1,-1,-1);	// Couldn't find the face.

	if (greyImg)
		cvReleaseImage( &greyImg );
	cvReleaseMemStorage( &storage );
	//cvReleaseHaarClassifierCascade( &cascade );

	return rc;	// Return the biggest face found, or (-1,-1,-1,-1).
}
```

Now you can simply call "detectFaceInImage" whenever you want to find a face in an image. You also need to specify the face classifier that OpenCV should use to detect the face. For example, OpenCV comes with several different classifiers for frontal face detection, as well as some profile faces (side view), eye detection, nose detection, mouth detection, whole body detection, etc. You can actually use this function with any of these other detectors if you want, or even create your own custom detector such as for car or person detection (read here), but since frontal face detection is the only one that is very reliable, it is the only one I will discuss.

For frontal face detection, you can chose one of these Haar Cascade Classifiers that come with OpenCV (in the "data\haarcascades\" folder):

- "haarcascade_frontalface_default.xml"
- "haarcascade_frontalface_alt.xml"
- "haarcascade_frontalface_alt2.xml"
- "haarcascade_frontalface_alt_tree.xml"

Each one will give slightly different results depending on your environment, so you could even use all of them and combine the results together (if you want the most detections). There are also some more eye, head, mouth and nose detectors in the downloads section of Modesto's page.

So you could do this in your program for face detection:

```
// Haar Cascade file, used for Face Detection.
char *faceCascadeFilename = "haarcascade_frontalface_alt.xml";
// Load the HaarCascade classifier for face detection.
CvHaarClassifierCascade* faceCascade;
```

```
faceCascade = (CvHaarClassifierCascade*)cvLoad(faceCascadeFilename, 0, 0, 0);
if( !faceCascade ) {
        printf("Couldnt load Face detector '%s'\n", faceCascadeFilename);
        exit(1);
}

// Grab the next frame from the camera.
IplImage *inputImg = cvQueryFrame(camera);

// Perform face detection on the input image, using the given Haar classifier
CvRect faceRect = detectFaceInImage(inputImg, faceCascade);

// Make sure a valid face was detected.
if (faceRect.width > 0) {
        printf("Detected a face at (%d,%d)!\n", faceRect.x, faceRect.y);
}

.... Use 'faceRect' and 'inputImg' ....

// Free the Face Detector resources when the program is finished
cvReleaseHaarClassifierCascade( &cascade );
```

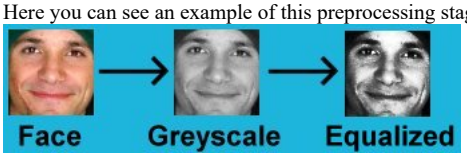
How to preprocess facial images for Face Recognition:

Now that you have detected a face, you can use that face image for Face Recognition. However, if you tried to simply perform face recognition directly on a normal photo image, you will probably get less than 10% accuracy!

It is extremely important to apply various image pre-processing techniques to standardize the images that you supply to a face recognition system. Most face recognition algorithms are extremely sensitive to lighting conditions, so that if it was trained to recognize a person when they are in a dark room, it probably wont recognize them in a bright room, etc. This problem is referred to as "lumination dependent", and there are also many other issues, such as the face should also be in a very consistent position within the images (such as the eyes being in the same pixel coordinates), consistent size, rotation angle, hair and makeup, emotion (smiling, angry, etc), position of lights (to the left or above, etc). This is why it is so important to use a good image preprocessing filters before applying face recognition. You should also do things like removing the pixels around the face that aren't used, such as with an elliptical mask to only show the inner face region, not the hair and image background, since they change more than the face does.

For simplicity, the face recognition system I will show you is Eigenfaces using greyscale images. So I will show you how to easily convert color images to greyscale (also called 'grayscale'), and then easily apply Histogram Equalization as a very simple method of automatically standardizing the brightness and contrast of your facial images. For better results, you could use color face recognition (ideally with color histogram fitting in HSV or another color space instead of RGB), or apply more processing stages such as edge enhancement, contour detection, motion detection, etc. Also, this code is resizing images to a standard size, but this might change the aspect ratio of the face. You can read my tutorial HERE on how to resize an image while keeping its aspect ratio the same.

Here you can see an example of this preprocessing stage:



Here is some basic code to convert from a RGB or greyscale input image to a greyscale image, resize to a consistent dimension, then apply Histogram Equalization for consistent brightness and contrast:

```
// Either convert the image to greyscale, or use the existing greyscale image.
IplImage *imageGrey;
if (imageSrc->nChannels == 3) {
        imageGrey = cvCreateImage( cvGetSize(imageSrc), IPL_DEPTH_8U, 1 );
        // Convert from RGB (actually it is BGR) to Greyscale.
        cvCvtColor( imageSrc, imageGrey, CV_BGR2GRAY );
}
else {
        // Just use the input image, since it is already Greyscale.
        imageGrey = imageSrc;
}

// Resize the image to be a consistent size, even if the aspect ratio changes.
IplImage *imageProcessed;
imageProcessed = cvCreateImage(cvSize(width, height), IPL_DEPTH_8U, 1);
// Make the image a fixed size.
// CV_INTER_CUBIC or CV_INTER_LINEAR is good for enlarging, and
// CV_INTER_AREA is good for shrinking / decimation, but bad at enlarging.
cvResize(imageGrey, imageProcessed, CV_INTER_LINEAR);

// Give the image a standard brightness and contrast.
cvEqualizeHist(imageProcessed, imageProcessed);

.....  Use 'imageProcessed' for Face Recognition ....

if (imageGrey)
        cvReleaseImage(&imageGrey);
if (imageProcessed)
        cvReleaseImage(&imageProcessed);
```

How Eigenfaces can be used for Face Recognition:

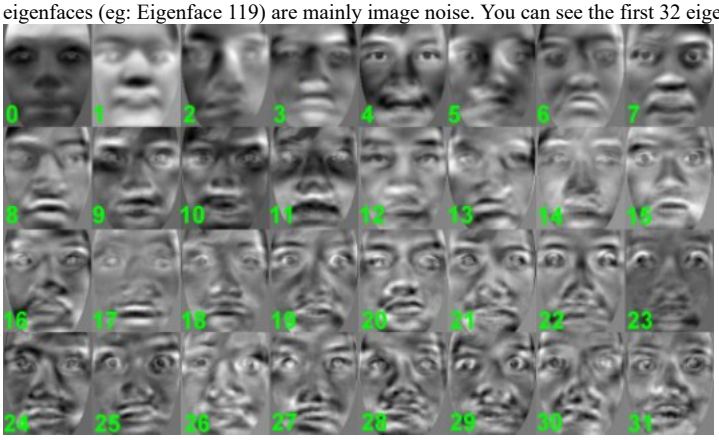
Now that you have a pre-processed facial image, you can perform Eigenfaces (PCA) for Face Recognition. OpenCV comes with the function "cvEigenDecomposite()", which performs the PCA operation, however you need a database (training set) of images for it to know how to recognize each of your people.

So you should collect a group of preprocessed facial images of each person you want to recognize. For example, if you want to recognize someone from a class of 10 students, then you could store 20 photos of each person, for a total of 200 preprocessed facial images of the same size (say 100x100 pixels).

The theory of Eigenfaces is explained in the two Face Recognition with Eigenface articles in Servo Magazine, but I will also attempt to explain it here.

Use "Principal Component Analysis" to convert all your 200 training images into a set of "Eigenfaces" that represent the main differences between the training images. First it will find the "average face image" of your images by getting the mean value of each pixel. Then the eigenfaces are calculated in comparison to this average face, where the first eigenface is the most dominant face differences, and the second eigenface is the second most dominant face differences, and so on, until you have about 50 eigenfaces that represent most of the differences in all the training set images.

In these example images above you can see the average face and the first and last eigenfaces that were generated from a collection of 30 images each of 4 people. Notice that the average face will show the smooth face structure of a generic person, the first few eigenfaces will show some dominant features of faces, and the last eigenfaces (eg: Eigenface 119) are mainly image noise. You can see the first 32 eigenfaces in the image below.



Explanation of Face Recognition using Principal Component Analysis:

To explain Eigenfaces (Principal Component Analysis) in simple terms, Eigenfaces figures out the main differences between all the training images, and then how to represent each training image using a combination of those differences.

So for example, one of the training images might be made up of:
(averageFace) + (13.5% of eigenface0) - (34.3% of eigenface1) + (4.7% of eigenface2) + ... + (0.0% of eigenface199).
Once it has figured this out, it can think of that training image as the 200 ratios:
{13.5, -34.3, 4.7, ..., 0.0}.

It is indeed possible to generate the training image back from the 200 ratios by multiplying the ratios with the eigenface images, and adding the average face. But since many of the last eigenfaces will be image noise or wont contribute much to the image, this list of ratios can be reduced to just the most dominant ones, such as the first 30 numbers, without effecting the image quality much. So now it's possible to represent all 200 training images using just 30 eigenface images, the average face image, and a list of 30 ratios for each of the 200 training images.

Interestingly, this means that we have found a way to compress the 200 images into just 31 images plus a bit of extra data, without loosing much image quality. But this tutorial is about face recognition, not image compression, so we will ignore that :-)

To recognize a person in a new image, it can apply the same PCA calculations to find 200 ratios for representing the input image using the same 200 eigenfaces. And once again it can just keep the first 30 ratios and ignore the rest as they are less important. It can then search through its list of ratios for each of its 20 known people in its database, to see who has their top 30 ratios that are most similar to the 30 ratios for the input image. This is basically a method of checking which training image is most similar to the input image, out of the whole 200 training images that were supplied.

Implementing Offline Training:

For implementation of offline training, where files are used as input and output through the command-line, I am using a similar method as the Face Recognition with Eigenface implementation in Servo Magazine, so you should read that article first, but I have made a few slight changes.

Basically, to create a facerec database from training images, you create a text file that lists the image files and which person each image file represents.
For example, you could put this into a text file called "4_images_of_2_people.txt":

```
1 Shervin data\Shervin\Shervin1.bmp
1 Shervin data\Shervin\Shervin2.bmp
1 Shervin data\Shervin\Shervin3.bmp
1 Shervin data\Shervin\Shervin4.bmp
2 Chandan data\Chandan\Chandan1.bmp
2 Chandan data\Chandan\Chandan2.bmp
2 Chandan data\Chandan\Chandan3.bmp
2 Chandan data\Chandan\Chandan4.bmp
```

This will tell the program that person 1 is named "Shervin", and the 4 preprocessed facial photos of Shervin are in the "data\Shervin" folder, and person 2 is called "Chandan" with 4 images in the "data\Chandan" folder. The program can then loaded them all into an array of images using the function "loadFaceImgArray()". Note that for simplicity, it doesn't allow spaces or special characters in the person's name, so you might want to enable this, or replace spaces in a person's name with underscores (such as Shervin_Emami).

To create the database from these loaded images, you use OpenCV's "cvCalcEigenObjects()" and "cvEigenDecomposite()" functions, eg:

```
// Tell PCA to quit when it has enough eigenfaces.
CvTermCriteria calcLimit = cvTermCriteria( CV_TERMCRIT_ITER, nEigens, 1);

// Compute average image, eigenvectors (eigenfaces) and eigenvalues (ratios).
cvCalcEigenObjects(nTrainFaces, (void*)faceImgArr, (void*)eigenVectArr,
        CV_EIGOBJ_NO_CALLBACK, 0, 0, &calcLimit,
        pAvgTrainImg, eigenValMat->data.fl);

// Normalize the matrix of eigenvalues.
cvNormalize(eigenValMat, eigenValMat, 1, 0, CV_L1, 0);

// Project each training image onto the PCA subspace.
CvMat projectedTrainFaceMat = cvCreateMat( nTrainFaces, nEigens, CV_32FC1 );
int offset = projectedTrainFaceMat->step / sizeof(float);
for(int i=0; i<nTrainFaces; i++) {
        cvEigenDecomposite(faceImgArr[i], nEigens, eigenVectArr, 0, 0,
                pAvgTrainImg, projectedTrainFaceMat->data.fl + i*offset);
}
```

You now have:

- the average image "pAvgTrainImg",
- the array of eigenface images "eigenVectArr[]" (eg: 200 eigenfaces if you used nEigens=200 training images),
- the matrix of eigenvalues (eigenface ratios) "projectedTrainFaceMat" of each training image.

These can now be stored into a file, which will be the face recognition database. The function "storeTrainingData()" in the code will store this data into the file "facedata.xml", which can be reloaded anytime to recognize people that it has been trained for. There is also a function "storeEigenfaceImages()" in the code, to generate the images shown earlier, of the average face image to "out_averageImage.bmp" and eigenfaces to "out_eigenfaces.bmp".

Implementing Offline Recognition:

For implementation of the offline recognition stage, where the face recognition system will try to recognize who is the face in several photos from a list in a text file, I am also using an extension of the Face Recognition with Eigenface implementation in Servo Magazine.
The same sort of text file that is used for offline training can also be used for offline recognition. The text file lists the images that should be tested, as well as the correct person in that image. The program can then try to recognize who is in each photo, and check the correct value in the input file to see whether it was correct or not, for generating statistics of its own accuracy.

The implementation of the offline face recognition is almost the same as offline training:

1. The list of image files (preprocessed faces) and names are loaded into an array of images, from the text file that is now used for recognition testing (instead of training). This is performed in code by "loadFaceImgArray()".
2. The average face, eigenfaces and eigenvalues (ratios) are loaded from the face recognition database file "facedata.xml", by the function "loadTrainingData()".
3. Each input image is projected onto the PCA subspace using the OpenCV function "cvEigenDecomposite()", to see what ratio of eigenfaces is best for representing this input image.
4. But now that it has the eigenvalues (ratios of eigenface images) to represent the input image, it looks for the original training image that had the most similar ratios. This is done mathematically in the function "findNearestNeighbor()" using the "Euclidean Distance", but basically it checks how similar the input image is to each training image, and finds the most similar one: the one with the least distance in Euclidean Space. As mentioned in the Servo Magazine article, you might get better results if you use the Mahalanobis space (define USE_MAHALANOBIS_DISTANCE in the code).
5. The distance between the input image and most similar training image is used to determine the "confidence" value, to be used as a guide of whether someone was actually recognized or not. A confidence of 1.0 would mean a good match, and a confidence of 0.0 or negative would mean a bad match. But beware that the confidence formula I use in the code is just an extremely basic confidence metric that isn't reliable, so if you need something more reliable you should look for "Face Verification" algorithms. If you find that it gives misleading values for your images, you should ignore it or disable it in the code (eg: set the confidence always to 1.0).

Once it knows which training image is most similar to the input image, and assuming the confidence value is not too low (it should be atleast 0.6 or higher), then it has figured out who that person is, in other words, it has recognized that person!

Implementing Realtime Recognition from a Camera:

It is very easy to use a webcam stream as input to the face recognition system instead of a file list. Basically you just need to grab frames from a camera instead of from a file, and you run forever until the user wants to quit, instead of just running until the file list has run out. OpenCV provides the 'cvCreateCameraCapture()' function (also known as 'cvCaptureFromCAM()') for this.

Grabbing frames from a webcam can be implemented easily using this function:

```
// Grab the next camera frame. Waits until the next frame is ready, and
// provides direct access to it, so do NOT modify or free the returned image!
// Will automatically initialize the camera on the first frame.
IplImage* getCameraFrame(CvCapture* &camera)
{
        IplImage *frame;
        int w, h;

        // If the camera hasn't been initialized, then open it.
        if (!camera) {
                printf("Acessing the camera ...\n");
                camera = cvCreateCameraCapture( 0 );
                if (!camera) {
                        printf("Couldn't access the camera.\n");
                        exit(1);
                }
                // Try to set the camera resolution to 320 x 240.
                cvSetCaptureProperty(camera, CV_CAP_PROP_FRAME_WIDTH, 320);
                cvSetCaptureProperty(camera, CV_CAP_PROP_FRAME_HEIGHT, 240);
                // Get the first frame, to make sure the camera is initialized.
                frame = cvQueryFrame( camera );
                if (frame) {
                        w = frame->width;
```

```
                        h = frame->height;
                        printf("Got the camera at %dx%d resolution.\n", w, h);
                }
                // Wait a little, so that the camera can auto-adjust its brightness.
                Sleep(1000);    // (in milliseconds)
        }

        // Wait until the next camera frame is ready, then grab it.
        frame = cvQueryFrame( camera );
        if (!frame) {
                printf("Couldn't grab a camera frame.\n");
                exit(1);
        }
        return frame;
}
```

This function can be used like this:

```
CvCapture* camera = 0;  // The camera device.
while ( cvWaitKey(10) != 27 ) { // Quit on "Escape" key.
        IplImage *frame = getCameraFrame(camera);
        ...
}
// Free the camera.
cvReleaseCapture( &camera );
```

Note that if you are developing for MS Windows, you can grab camera frames twice as fast as this code by using the videoInput Library v0.1995 by Theo Watson. It uses hardware-accelerated DirectShow, whereas OpenCV uses VFW that hasn't changed in 15 years!

Putting together all the parts that I have explained so far, the face recognition system runs as follows:

1. Grab a frame from the camera (as I mentioned here).
2. Convert the color frame to greyscale (as I mentioned here).
3. Detect a face within the greyscale camera frame (as I mentioned here).
4. Crop the frame to just show the face region (using cvSetImageROI() and cvCopyImage()).
5. Preprocess the face image (as I mentioned here).
6. Recognize the person in the image (as I mentioned here).

Implementing Online Training from a Camera:

Now you have a way to recognize people in realtime using a camera, but to learn new faces you would have to shutdown the program, save the camera images as image files, update the training images list, use the offline training method from the command-line, and then run the program again in realtime camera mode. So in fact, this is exactly what you can do programmatically to perform online training from a camera in realtime!

So here is the easiest way to add a new person to the face recognition database from the camera stream without shutting down the program:

1. Collect a bunch of photos from the camera (preprocessed facial images), possibly while you are performing face recognition also.
2. Save the collected face images as image files onto the hard-disk using cvSaveImage().
3. Add the filename of each face image onto the end of the training images list file (the text file that is used for offline training).
4. Once you are ready for online training the new images (such as once you have 20 faces, or when the user says that they are ready), you "retrain" the database from all the image files. The text file listing the training image files has the new images added to it, and the images are stored as image files on the computer, so online training works just like it did in offline training.
5. But before retraining, it is important to free any resources that were being used, and re-initialize the variables, so that it behaves as if you shutdown the program and restarted. For example, after the images are stored as files and added to the training list text file, you should free the arrays of eigenfaces, before doing the equivalent of offline training (which involves loading all the images from the training list file, then finding the eigenfaces and ratios of the new training set using PCA).

This method of online training is fairly inefficient, because if there was 50 people in the training set and you add one more person, then it will train again for all 51 people, which is bad because the amount of time for training is exponential with more users or training images. But if you are just dealing with a few hundred training images in total then it shouldn't take more than a few seconds.

Download OnlineFaceRec:

The software and source-code is available here (open-source freeware), to use on Windows, Mac, Linux, Android, iPhone, etc as you wish for educational or personal purposes, but NOT for commercial, criminal-detection, or military purposes (because this code is way too simple & unreliable for critical applications such as criminal detection, and also I no longer support any military).

For Windows:

Click here to download "OnlineFaceRec" for Windows (eg: Win98, WinXP, WinNT, Win7): onlineFaceRec.zip
(0.07MB file including C/C++ source code, VS2008 project files and the compiled Win32 program, created 4th Feb 2012).

- If you dont have the OpenCV 2.0 SDK then you can just get the Win32 DLLs and HaarCascade for running this program (including 'cvaux200.dll' and 'haarcascade_frontalface_alt.xml'): onlineFaceRec_OpenCVbinaries.7z (1.7MB 7-Zip file).
- And if you want to run the program but dont have the Visual Studio 2008 runtime installed then you can just get the Win32 DLLs ('msvcr90.dll', etc): MS_VC90_CRT.7z (0.4MB 7-Zip file).
- To open Zip or 7z files you can use the freeware 7-Zip program (better than WinZip and WinRar in my opinion) from HERE.

For Linux:

Click here to download "OnlineFaceRec" for Linux (eg: Linux Mint, Ubuntu, Debian, Fedora, OpenSUSE, Arch): onlineFaceRec_Linux.zip
(0.003MB file including C/C++ source code and a compiled Linux program, created 30th Dec 2011).

- To build the code on Linux, first install OpenCV development packages (eg: follow the Official OpenCV Linux installation tutorial).
- You also need to put the file "haarcascade_frontalface_alt.xml" into the same folder as your program, such as by running this:

```
sudo apt-get install libopencv-dev
cp /usr/share/opencv/haarcascades/haarcascade_frontalface_alt.xml .
```

- Then build the code using GCC and the OpenCV headers and libraries. eg:

```
g++ OnlineFaceRec.cpp -I/usr/local/include/opencv -lopencv_core -lopencv_imgproc -lopencv_highgui -lopencv_legacy -lopencv_objdetect -o OnlineFa
```

The code was tested with MS Visual Studio 2008 using OpenCV v2.0 and on Linux with GCC 4.2 using OpenCV v2.3.1, but I assume it works with other versions & compilers fairly easily, and it should work the same in all versions of OpenCV before v2.2. Students also ported this code to Dev-C++ at https://sourceforge.net/projects/facerec/.

There are two different ways you can use this system:

1. As a realtime program that performs face detection and online face recognition from a web camera.
2. As a command-line program to perform offline face recognition using text files, just like the eigenface program in Servo Magazine.

How to use the realtime webcam FaceRec system:

If you have a webcam plugged in, then you should be able to test this program by just double-clicking the EXE file in Windows (or compile the code and run it if you are using Linux or Mac). Hit the Escape key on the GUI window when you want to quit the program.

After a few seconds it should show the camera image, with the detected face highlighted. But at first it wont have anyone in its face rec database, so you will need to create it by entering a few keys. Beware that to use the keyboard, you have to click on the DOS console window before typing anything (because if the OpenCV window is highlighted then the code wont know what you typed).

1. In the console window, hit the 'n' key on your keyboard when a person is ready for training. This will add a new person to the facerec database. Type in the person's name (without any spaces) and hit Enter.
2. It will begin to automatically store all the processed frontal faces that it sees. Get a person to move their head around a bit until it has stored about 20 faces of them. (The facial images are stored as PGM files in the "data" folder, and their names are appended to the text file "train.txt").
3. Get the person in front of the camera to move around a little and move their face a little, so that there will be some variance in the training images.
4. Then when you have enough detected faces for that person, ideally more than 30 for each person, hit the 't' key in the console window to begin training on the images that were just collected. It will then pause for about 5-30 seconds (depending on how many faces and people are in the database), and finally continue once it has retrained with the extra person. The database file is called "facedata.xml".
5. It should print the person's name in the console whenever it recognizes them.
6. Repeat this again from step 1 whenever you want to add a new person, even after you have shutdown the program.
7. If you hit the Escape key in the console (or GUI) then it will shutdown safely. You can then run the program again later and it should already be trained to recognize anyone that was added.

Depending on the speed of your computer and camera, it might be recording faces too fast for you (ie: when you click 'n', it saves 100 faces when you only want about 20 faces). So you might want to modify the code to just run at about 1 frame per second (adjust the "cvWaitKey(10)" code to something like "cvWaitKey(1000)").

Note that the code currently doesn't let you delete a previous person or add more training images to an existing known person, etc. These features aren't too hard to add (deleting a person means deleting the image files and lines from the training file, adding more images to an existing person requires keeping track of how many images where already loaded for that person). But there will always be more features that could be added!

How to use the offline cmd-based FaceRec system:

If you want to use the command-line mode for offline facerec (based on the eigenface program in Servo Magazine), such as for testing, here are the instructions:

1. First, you need some face images. You can find many face databases at the Essex page http://peipa.essex.ac.uk/benchmark/databases/index.html. I used "ORL / AT&T: The Database of Faces": Cambridge_FaceDB.zip (3.7MB).
2. List the training and test face images you want to use into text files. If you downloaded the ORL database, you can use my sample text files: facerecExample_ORL.zip (1kB). To use these input files exactly as provided, unzip the facerecExample_ORL.zip file in your folder with OnlineFaceRec.exe and then unzip the ORL face database there (eg: 'Cambridge_DB\s1\1.pgm').
3. To run the learning phase of eigenface, enter in the command prompt:
   OnlineFaceRec train lower2.txt
   That will create a database file "facedata.xml" with just 2 faces each from 10 people (a total of 20 faces). It will also generate "out_averageImage.bmp" and "out_eigenfaces.bmp" for you to look at.
4. To run the recognition phase, enter:
   OnlineFaceRec test upper6.txt
   (That will test the database with 6 faces each from 10 people (a total of 60 faces).
   It should give a surprisingly high recognition rate of 95% correct, from just 2 photos of each person!

You can try other combinations, such as "**OnlineFaceRec train all10.txt**" and "**OnlineFaceRec test all10.txt**", to get even higher accuracy. Just remember that these photos are already processed, and from a very fixed lab environment. If you try this with images from real-world conditions, you should expect just 40-80% correct!

How to improve the Face Recognition accuracy:

To improve the recognition performance, there are MANY things that can be improved here (look at commercial Face Recognition systems such as SPOTR for examples), and some improvements can be fairly easy to implement. For example, you could add color processing, edge detection, etc.

You can usually improve the face recognition accuracy by using more input images, atleast 50 per person, by taking more photos of each person, particularly from different angles and lighting conditions. If you cant take more photos, there are several simple techniques you could use to obtain more training images, by generating new images from your existing ones:

- You could create mirror copies of your facial images, so that you will have twice as many training images and it wont have a bias towards left or right.
- You could translate or resize or rotate your facial images slightly to produce many alternative images for training, so that it will be less sensitive to exact conditions.
- You could add image noise to have more training images that improve the tolerance to noise.

Remember that it is important to have a lot of variation of conditions for each person, so that the classifier will be able to recognize the person in different lighting conditions and positions, instead of looking for specific conditions. But it's also important to make sure that a set of images for a person is not too varied, such as if you rotated some images by 90 degrees. This would make the classifier to be too generic and also give very bad results, so if you think you will have a set of images with too much variance (such as rotation more than 20 degrees), then you could create separate sets of training images for each person. For example, you could train a classifier to recognize "John_Facing_Forward" and another one for "John_Facing_Left" and other ones "Mary_Facing_Forward", "Mary_Facing_Left", etc. Then each classifier can have a lot of variance but not too much, and you simply need to associate the different classifiers for each person with that one person (ie: "John" or "Mary").

That's why you can often get very bad results if you don't use good preprocessing on your images. As I mentioned earlier, Histogram Equalization is a very basic image preprocessing method that can make things worse in some situations, so you will probably have to combine several different methods until you get decent results.

And something important to understand is that at the heart of the algorithm, it is matching images by basically doing the equivalent of subtracting the testing image with a training image to see how similar they are. This would work fairly well if a human performed it, but the computer just thinks in terms of pixels and numbers. So if you imagine that it is looking at one pixel in the test image, and subtracting the greyscale value of that pixel with the value of the pixel in the EXACT same location of each training image, and the lower the difference then the better the match. So if you just move an image by a few pixels across, or use an image that is just a few pixels bigger or has a few more pixels of the forehead showing than the other image, etc, then it will think they are completely different images! This is also true if the background is different, because the code doesn't know the difference between background and foreground (face), which is why it's important to crop away as much of the background as you can, such as by only using a small section inside the face that doesn't include any background at all.

Since the images should be almost perfectly aligned, it actually means that in many cases, using small low-res images (such as by shrinking the images to thumbnail size) can give better recognition results than large hi-res images! (Because they will be better aligned if they are shrunk).

Also, even if the images are perfectly aligned, if the testing image is a bit brighter than the training image then it will still think there is not much of a match. Histogram Equalization can help in many cases but it can also make things worse in other cases, so differences in lighting is a difficult & common problem. There are also issues such as if there was a shadow on the left of the nose in the training image and on the right in the test image then it will often cause a bad match, etc.

That's why face recognition is relatively easy to do in realtime if you are training on someone and then instantly trying to recognize them after, since it will be the same camera, and background will be the same, their expressions will be almost the same, the lighting will be the same, and the direction you are viewing them from will be the same. So you will often get good recognition results at that moment. But once you try to recognize them from a different direction or from a different room or outside or on a different time of the day, it will often give bad results!

So it's important to do a lot of experimentation if you want better results, and if you still can't get good results after trying many things, perhaps you will need a more complicated face recognition algorithm than PCA (Eigenfaces), such as 3D Face Recognition or Active Appearance Models, mentioned below.

Alternative techniques to Eigenfaces for Face Recognition:

Something you should know is that Eigenfaces is considered the simplest method of accurate face recognition, but many other (much more complicated) methods or combinations of multiple methods are slightly more accurate. So if you have tried the hints above for improving your training database and preprocessing but you still need more accuracy, you will probably need to learn some more complicated methods, or for example you could figure out how to combine separate Eigenface models for the eyes, nose & mouth.

Most tutorials on face recognition are for basic Neural Networks, which usually don't work as well as Eigenfaces does. And unfortunately there are only some basic explanations for better type of face recognition than Eigenfaces, such as recognition from video and other techniques at the "Face Recognition Homepage", or 3D Face Recognition and Active Appearance Models. But for other techniques, you should read some recent computer vision research papers from CVPR and other computer vision conferences. Most computer vision or machine vision conferences include new advances in face detection and face recognition that give slightly better accuracy. So for example you can look for the CVPR10 and CVPR09 conferences at http://www.cvpapers.com/. There are also several better Face Recognition implementations inside the CSU Face Identification Evaluation System.

But remember that when you are implementing state-of-art techniques such as this, you usually won't be able to find any information or help about it except for what is in the 1 or 2 research papers about the technique. So you should only do it if you are confident at getting it to work by yourself! If you had trouble understanding this tutorial then you will probably have much more trouble understanding a conference paper about a more complicated technique..

Troubleshooting errors:

Error messages when compiling the project 'OnlineFaceRec':

- If you try to compile the source-code and get this error:
  'CV_HAAR_FIND_BIGGEST_OBJECT' : undeclared identifier
  'CV_HAAR_DO_ROUGH_SEARCH' : undeclared identifier
  It means that you are using OpenCV v1.1 or older, which doesnt know about those 2 parameters for Face Detection. So just comment out these 2 parameters in the code and it should be fine. They only make Face Detection run about 2% faster, so it's not a big difference.

- If you try to compile the source-code and get this error:
  Cannot open include file: 'vector': No such file or directory
  It means you are probably using a C compiler (such as Turbo C), but you need a C++ compiler (such as Visual Studio, Eclipse, C++ Builder, DevCpp, Code::Blocks, Xcode, MinGW or GNU g++).

- If you try to compile the source-code and get this error:
  Cannot open include file: 'cv.h': No such file or directory
  It means your compiler can't find the OpenCV header files. Either you haven't installed and built OpenCV yet, or you haven't configured your project

properties to know where the OpenCV includes folder is on your computer (eg: 'C:\OpenCV\include\opencv\'). See the Official OpenCV Install Guide for more info.

- If you try to compile the source-code and get this error:
  cannot open input file 'cv200.lib'
  It means your compiler can't find the OpenCV lib files. Either you haven't installed and built OpenCV yet, or you haven't configured your project properties to know where the OpenCV 'lib' folder is on your computer (eg: 'C:\OpenCV\lib'). See the Official OpenCV Install Guide for more info.
- If you try to compile the source-code and get this error:
  error LNK2001: Unsolved external symbol _cvEigenDecomposite
  It means you need to add the lib file 'cvaux200.lib' to you linker settings, just like 'cv200.lib', 'cxcore200.lib' and 'highgui200.lib'. In VS2008, this can be added in project Properties -> Linker -> Input -> Additional Dependencies.

Error messages when running the program 'OnlineFaceRec.exe':

- If you try to run the program and get this error:
  This application has failed to start because cv200.dll was not found. Re-installing the application may fix this problem.
  It means it can't find the OpenCV library on your computer. If you have installed OpenCV 2.0, then make sure you have set your Windows PATH correctly. Otherwise, you can just download and put the files from 'onlineFaceRec_OpenCVbinaries.7z' above into the same folder as 'OnlineFaceRec.exe'.

- If you try to run the program and get this error:
  Can't open training database file 'facedata.xml'.
  Don't worry because it just means you don't have an existing trained database, so the program will create a new one from scratch for you.

- If you try to run the program and get this error:
  ERROR in recognizeFromCam(): Could not load Haar cascade Face detection classifier in 'haarcascade_frontalface_alt.xml'.
  It means it can't find the Haar Cascade XML file. So make sure you copy the file 'haarcascade_frontalface_alt.xml' (from the folder 'data\haar_cascades\' in OpenCV) to the same folder as 'OnlineFaceRec.exe'.

- If you try to run the program and get a missing file error such as any of these:
  Can't open file <>
  Can't load image from <>
  ERROR in recognizeFromCam(): Bad input image!
  It means it can't find an image file that you put in one of the input files (such as 'train.txt' or 'test.txt'). For example, if you run 'OnlineFaceRec.exe train my_database.txt' and your file 'my_database.txt' says '1 Shervin ..\pic1.bmp', then make sure your pictures such as pic1.bmp is in the parent directory from 'OnlineFaceRec.exe', or wherever the path is that you wrote in your input text file.

About Me:

- I have spent almost every day of the past 23 years making robots or electronic inventions or computer programs. I also have a Masters degree in Robotics, a BSc in Computer Science, and a BEngg in Mechatronics Engineering.
- I have worked in Australia, USA (California & Florida), UAE (Dubai) and Philippines on state-of-the-art robotics & computer vision technologies, from self-balancing robots that can climb stairs, to humanoid robots that can talk & recognize faces, to some of the fastest 3D software-renderers & 3D Augmented Reality mobile apps available.
- To read more about me, contact me, subscribe to my RSS, or add me on Google+ or Twitter, click HERE.

- St