Seeing With OpenCV

Finding Faces in Images

by Robin Hewitt

PART 2

Last month's article in this series introduced OpenCV – Intel's free, open-source computer vision library for C/C++ programmers. It covered the basics – downloading and installing OpenCV, reading and writing image files, capturing video, and working with the IplImage data structure.

This month, I'll show you how to use OpenCV to detect faces. I'll explain how the face detection algorithm works, and give you tips for getting the most out of it.

Background and Preliminaries

OpenCV uses a type of face detector called a Haar Cascade classifier. The sidebar, "How Face Detection Works, or What's a Haar Cascade Classifier, Anyhow?" explains what this mouthful means. Figure 1 shows an example of OpenCV's face detector in action.

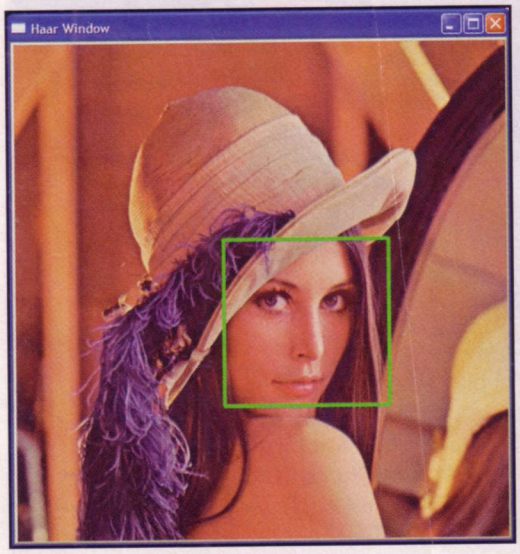


FIGURE 1. Face detection with OpenCV, using default parameters. The input image is lena.jpg, in the samples/c directory.

Given an image – which can come from a file or from live video – the face detector examines each image location and classifies it as "Face" or "Not Face." Classification assumes a fixed scale for the face, say 50 x 50 pixels. Since faces in an image might be smaller or larger than this, the classifier runs over the image several times, to search for faces across a range of scales. This may seem like an enormous amount of processing, but thanks to algorithmic tricks (explained in the sidebar), classification is very fast, even when it's applied at several scales.

The classifier uses data stored in an XML file to decide how to classify each image location. The OpenCV download includes four flavors of XML data for frontal face detection, and one for profile faces. It also includes three non-face XML files: one for full body (pedestrian) detection, one for upper body, and one for lower body.

You'll need to tell the classifier where to find the data file you want it to use. The one I'll be using is called haarcascade_frontal face_default.xml. In OpenCV version 1.0, it's located at:

```
[OPENCV_ROOT]/data/haarcascades/
haarcascade_frontalface_default.
xml
```

where `[OPENCV_ROOT]` is the path to your OpenCV installation. For example, if you're on Windows XP and you selected the default installation location, you'd use:

```
[OPENCV_ROOT] = "C:/Program Files/
OpenCV"
```

(If you're working with an older, 16-bit version of Windows, you'd use '\' as the directory separator, instead of '/'.)

It's a good idea to locate the XML file you want to use and make sure your path to it is correct before you code the rest of your face detection program.

You'll also need an image to process. The image lena.jpg is a good one to test with. It's located in the OpenCV samples/c directory. If you copy it to your program's working directory, you'll easily be able to compare your program's output with the output from the code in Figure 2.

Implementing Face Detection, Step by Step

Figure 2 shows the source code to load an image from a file, detect faces in it, and display the image with detected faces outlined in green. Figure 1 shows the display produced by this program when it's run from the command line, using:

```
DetectFaces lena.jpg
```

Initializing (and running) the Detector

The variable `CvHaarClassifier Cascade * pCascade` (Line 2) holds the data from the XML file you located earlier. To load the XML data into `pCascade`, you can use the `cvLoad()` function, as in Lines 11-13. `cvLoad()` is a general-purpose function for loading data from files. It takes up to

FIGURE 2. Source code to detect faces in one image. Usage: Detect Faces <image file>.

```
1 // declarations
2 CvHaarClassifierCascade * pCascade = 0;       // the face detector
3 CvMemStorage * pStorage = 0;                  // expandable memory buffer
4 CvSeq * pFaceRectSeq;                         // list of detected faces
5 int i;
6
7 // initializations
8 IplImage * pInpImg = (argc > 1) ?
9    cvLoadImage(argv[1], CV_LOAD_IMAGE_COLOR) : 0;
10 pStorage = cvCreateMemStorage(0);
11 pCascade = (CvHaarClassifierCascade *)cvLoad
12   ((OPENCV_ROOT"/data/haarcascades/haarcascade_frontalface_default.xml"),
13   0, 0, 0 );
14
15 // validate that everything initialized properly
16 if( !pInpImg || !pStorage || !pCascade )
17 {
18    printf("Initialization failed: %s \n",
19      (!pInpImg)?    "didn't load image file" :
20      (!pCascade)?   "didn't load Haar cascade -- "
21                     "make sure path is correct" :
22      "failed to allocate memory for data storage");
23    exit(-1);
24 }
25
26 // detect faces in image
27 pFaceRectSeq = cvHaarDetectObjects
28    (pInpImg, pCascade, pStorage,
29    1.1,                        // increase search scale by 10% each pass
30    3,                          // drop groups of fewer than three detections
31    CV_HAAR_DO_CANNY_PRUNING,   // skip regions unlikely to contain a face
32    cvSize(0,0));               // use XML default for smallest search scale
33
34 // create a window to display detected faces
35 cvNamedWindow("Haar Window", CV_WINDOW_AUTOSIZE);
36
37 // draw a rectangular outline around each detection
38 for(i=0;i<(pFaceRectSeq? pFaceRectSeq->total:0); i++ )
39 {
40    CvRect * r = (CvRect*)cvGetSeqElem(pFaceRectSeq, i);
41    CvPoint pt1 = { r->x, r->y };
42    CvPoint pt2 = { r->x + r->width, r->y + r->height };
43    cvRectangle(pInpImg, pt1, pt2, CV_RGB(0,255,0), 3, 4, 0);
44 }
45
46 // display face detections
47 cvShowImage("Haar Window", pInpImg);
48 cvWaitKey(0);
49 cvDestroyWindow("Haar Window");
50
51 // clean up and release resources
52 cvReleaseImage(&pInpImg);
53 if(pCascade) cvReleaseHaarClassifierCascade(&pCascade);
54 if(pStorage) cvReleaseMemStorage(&pStorage);
```

three input parameters. For this example, you'll only need the first parameter. This is the path to an XML file containing a valid Haar Cascade. Here, I've loaded the default frontal face detector included with OpenCV. If you're coding in C, set the remaining parameters to 0. If you're coding in C++, you can simply omit the unused parameters from your function call.

Before detecting faces in images, you'll also need to instantiate a `CvMemStorage` object (`pStorage`, declared at Line 3). This is a memory buffer that expands automatically, as needed. The face detector will put the list of detected faces into this buffer. Since the buffer is expandable, you won't need to worry about overflowing it. All you'll have to do is create it (Line 10), then release it when you're finished (Line 54).

You'll often need to load data from files with OpenCV. Since it's easy to get a path wrong, it's a good idea to insert a quick check to make sure everything loaded and initialized properly. Lines 16-24 do a simple error check, print a diagnostic message, and exit if initialization fails.

Running the Detector

Lines 27-32 call `cvHaarDetectObjects()` to run the face detector. This function takes up to seven parameters. The first three are the image pointer, XML data, and memory buffer. The remaining four parameters are set to their C++ defaults. These last four parameters are described below, in the section, "Parameters and Tuning."

Viewing the Results

A quick way to check if your program works is to display the results in an OpenCV window. You can create a display window using the `cvNamedWindow()` function, as in Line 35. The first parameter is a string, with a window name. The second, `CV_WINDOW_AUTOSIZE`, is a flag that tells the window to automatically resize itself to fit the image you give it to display.

To pass an image for display, call `cvShowImage()` with the name you previously assigned the window, and the image you want it to display. The `cvWaitKey()` call at Line 48 pauses the application until you close the window. If the window fails to close by clicking its close icon, click inside the window's display area, then press a keyboard key. Also, make sure your program calls `cvDestroyWindow()` (Line 49) to close the window.

Face detections are stored as a list of `CvRect struct` pointers. Lines 38-44 access each detection rectangle and add its outline to the `pInpImg` variable, which holds the in-memory image loaded from the file.

Releasing Resources

Lines 52-54 release the resources used by the input image, the XML data, and the storage buffer. If you'll be detecting faces in multiple images, you don't need to release the

How Face Detection Works

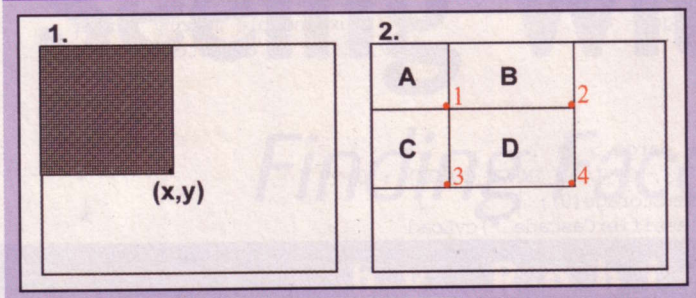


FIGURE B. The Integral Image trick. After integrating, the pixel at (x,y) contains the sum of all pixel values in the shaded rectangle. The sum of pixel values in rectangle D is (x4, y4) – (x2, y2) – (x3, y3) + (x1, y1).

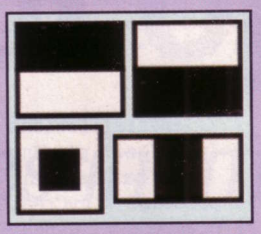


FIGURE A. Examples of the Haar features used in OpenCV.

OpenCV's face detector uses a method that Paul Viola and Michael Jones published in 2001. Usually called simply the Viola-Jones method, or even just Viola-Jones, this approach to detecting objects in images combines four key concepts:

- Simple rectangular features, called Haar features.
- An Integral Image for rapid feature detection.
- The AdaBoost machine-learning method.
- A cascaded classifier to combine many features efficiently.

The features that Viola and Jones used are based on Haar wavelets. Haar wavelets are single-wavelength square waves (one high interval and one low interval). In two dimensions, a square wave is a pair of adjacent rectangles — one light and one dark.

The actual rectangle combinations used for visual object detection are not true Haar wavelets. Instead, they contain rectangle combinations better suited to visual recognition tasks. Because of that difference, these features are called Haar features, or Haarlike features, rather than Haar wavelets. Figure A shows the features that OpenCV uses.

The presence of a Haar feature is determined by subtracting the average dark-region pixel value from the average light-region pixel value. If the difference is above a threshold (set during learning), that feature is said to be present.

To determine the presence or absence of hundreds of Haar features at every image location and at several scales efficiently, Viola and Jones used a technique called an Integral Image. In general, "integrating" means adding small units together. In this case, the small units are pixel values. The integral value for each pixel is the sum of all the pixels above it and to its left. Starting at the top left and traversing to the right and down, the entire image can be integrated with a few integer operations per pixel.

As Figure B1 shows, after integration, the value at each pixel location, (x,y) contains the sum of all pixel values within a rectangular region that has one corner at the top left of the image and the other at location (x,y). To find the average pixel value in this rectangle, you'd only need to divide the value at (x,y) by the rectangle's area.

But what if you want to know the summed values for some other rectangle, one that doesn't have one corner at the upper left of the image? Figure B2 shows the solution to that problem. Suppose you want the summed values in D. You can think of that as being the sum of pixel values in the combined rectangle, A+B +C+D, minus the sums in rectangles A+B and A+C, plus the sum of pixel values in A. In other words,

$$D = A+B+C+D - (A+B) - (A+C) + A.$$

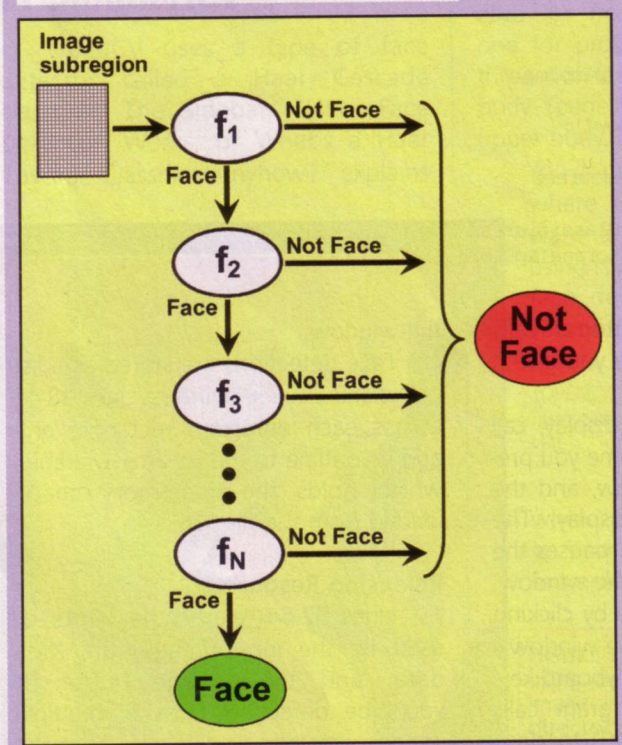
Conveniently, A+B+C+D is the Integral Image's value at location 4, A+B is the value at location 2, A+C is the value at location 3, and A is the value at location 1. So, with an Integral Image, you can find the sum of pixel values for any rectangle in the original image with just three integer operations:

$$(x4, y4) - (x2, y2) - (x3, y3) + (x1, y1).$$

To select the specific Haar features to use and to set threshold levels, Viola and Jones use a machine-learning method called AdaBoost. AdaBoost combines many "weak" classifiers to create one "strong" classifier. "Weak" here means the classifier only gets the right answer a little more often than random guessing would. That's not very good. But if you had a whole lot of these weak classifiers and each one "pushed" the final answer a little bit in the right direction, you'd have a strong, combined force for arriving at the correct solution. AdaBoost selects a set of weak classifiers to combine and assigns a weight to each. This weighted combination is the strong classifier.

Viola and Jones combined weak classifiers as a filter chain, shown in Figure C, that's especial-

FIGURE C. The classifier cascade is a chain of single-feature filters. Image subregions that make it through the entire cascade are classified as "Face." All others are classified as "Not Face."



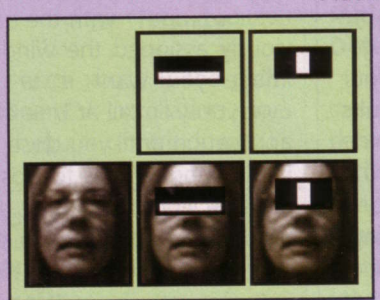


FIGURE D. The first two Haar features in the original Viola-Jones cascade.

ly efficient for classifying image regions. Each filter is a weak classifier consisting of one Haar feature. The threshold for each filter is set low enough that it passes all, or nearly all, face examples in the training set. (The training set is a large database of faces, maybe a thousand or so.) During use, if any one of these filters fails to pass an image region, that region is immediately classified as "Not Face." When a filter passes an image region, it goes to the next filter in the chain. Image regions that pass through all filters in the chain are classified as "Face." Viola and Jones dubbed this filtering chain a cascade.

The order of filters in the cascade is determined by weights that AdaBoost assigns. The more heavily weighted filters come first, to eliminate non-face image regions as quickly as possible. Figure D shows the first two features from the original Viola-Jones cascade superimposed on my face. The first one keys off the cheek area being lighter than the eye region. The second uses the fact that the bridge of the nose is lighter than the eyes.

XML data or the buffer until after you're done detecting faces.

Parameters and Tuning

There are several parameters you can adjust to tune the face detector for your application.

Minimum Detection Scale

The seventh parameter in the call to `cvHaarDetectObjects()` is the size of the smallest face to search for. In C, you can select the default for this by setting the scale to 0x0, as in Figure 2, Line 32. (In C++, simply omit this parameter to use the default.) But what is the default? You can find out by opening the XML file you'll be using. Look for the `<size>` tag. In the default frontal face detector, it's:

```
<size> 24 24 </size>.
```

So, for this cascade, the default minimum scale is 24 x 24.

Depending on the resolution you're using, this default size may be a very small portion of your overall image. A face image this small may not be meaningful or useful, and detecting it takes up CPU cycles you could use for other purposes. For these reasons — and also to minimize the number of face detections your own code needs to process — it's best to set the minimum detection scale only as small as you truly need.

To set the minimum scale higher than the default value, set this parameter to the size you want. A good rule of thumb is to use some fraction of your input image's width or height as the minimum scale — for example, 1/4 of the image width. If you specify a minimum scale other than the default, be sure its aspect ratio (the ratio of width to height) is the same as the default's. In this case, aspect ratio is 1:1.

Minimum Neighbors Threshold

One of the things that happens "behind the scenes" when you call the face detector is that each positive face region actually generates many hits from the Haar detector. Figure 3 shows OpenCV's internal rectangle list for the example image, lena.jpg. The face region itself generates the largest cluster of rectangles. These largely overlap. In addition, there's one small detection to the (viewer's) left, and two larger detections slightly above and left of the main face cluster.

Usually, isolated detections are false detections, so it makes sense to discard these. It also makes sense to somehow merge the multiple detections for each face region into a single detection. OpenCV does both these before returning its list of detected faces. The merge step first groups rectangles that contain a large amount of overlap, then finds the average rectangle for the group. It then replaces all rectangles in the group with the average rectangle.

Between isolated rectangles and large groupings are smaller groupings that may be faces, or may be false detections. The minimum-neighbors threshold sets the cutoff level for discarding or keeping rectangle groups based on how many raw detections are in the group. The C++ default for this parameter is three, which means to merge groups of three or more and discard groups with fewer rectangles. If you find that your face detector is missing a lot of faces, you might try lowering this threshold to two or one.

If you set it to 0, OpenCV will return the complete list of raw detections from the Haar classifier. While you're tuning your face detector, it's helpful to do this just to see what's going on inside OpenCV. Viewing the raw detections will improve your intuition about the effects of changing other parameters, which will help you tune them.

Scale Increase Rate

The fourth input parameter to `cvHaarDetectObjects()` specifies how quickly OpenCV should increase the scale for face detections with each pass it makes over an image. Setting this higher makes the detector run faster (by running fewer passes), but if it's too high, you may jump too quickly between scales and miss faces. The default in OpenCV is 1.1, in other words, scale increases by a factor of 1.1 (10%) each pass.

Canny Pruning Flag

The sixth parameter to `cvHaarDetectObjects()` is a flag variable. There are currently only two options: 0 or `CV_HAAR_DO_CANNY_PRUNING`. If the Canny Pruning option is selected, the detector skips image regions that are unlikely to contain a face, reducing computational overhead and possibly

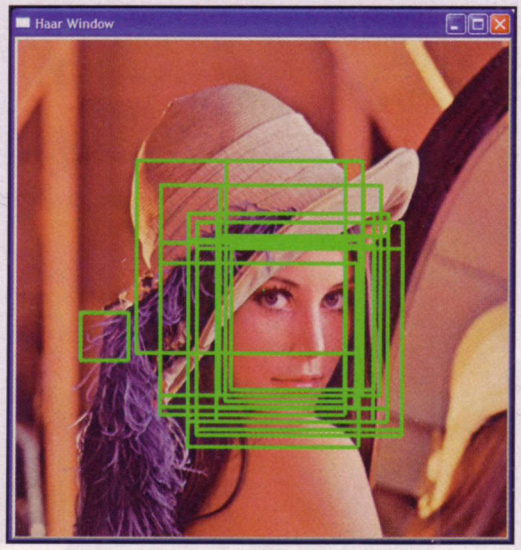


FIGURE 3. OpenCV's internal detection rectangles. To see these, use min_neighbors = 0.

References and Resources

- *OpenCV on Sourceforge* **http://sourceforge.net/projects/opencvlibrary**
- *Official OpenCV usergroup* **http://tech.groups.yahoo.com/group/OpenCV**
- G. Bradski, A. Kaehler, and V. Pisarevsky, "Learning-Based Computer Vision with Intel's Open Source Computer Vision Library," *Intel Technology Journal*, Vol 9(1), May 19, 2005. **www.intel.com/technology/itj/2005/volume09issue02/art03_learning_vision/p01_abstract.htm**
- R.E. Schapire, "A Brief Introduction to Boosting," *Joint Conference on Artificial Intelligence*, Morgan Kauman, San Francisco, pp. 1401-1406, 1999.
- P. Viola and M.J. Jones, "Rapid Object Detection using a Boosted Cascade of Simple Features," CVPR, 2001.

eliminating some false detections. The regions to skip are identified by running an edge detector (the Canny edge detector) over the image before running the face detector.

Again, the choice of whether or not to set this flag is a tradeoff between speed and detecting more faces. Setting this flag speeds processing, but may cause you to miss some faces. In general, you can do well with it set, but if you're having difficulty detecting faces, clearing this flag may allow you to detect more reliably. Setting the minimum-neighbors threshold to 0 so you can view the raw detections will help you better gauge the effect of using Canny Pruning.

The Haar Cascade

There are several frontal face detector cascades in OpenCV. The best choice for you will depend on your set-up. It's easy to switch between them – just change the file name. Why not try each?

It's also possible to create your own, custom XML file using the HaarTraining application, in OpenCV's apps directory. Using that application is beyond the scope of this article. However, the instructions are in OpenCV's apps/haartraining/docs directory.

Coming Up ...

Now that you've found a face, you might want to follow it around. Next month, I'll show you how to use Camshift, OpenCV's face tracking method, to do just that. Be seeing you! SV

About the Author

Robin Hewitt is an independent software consultant working in the areas of computer vision and robotics. She has worked as a Computer Vision Algorithm Developer at Evolution Robotics and is a member of SO(3), a computer-vision research group at UC San Diego. She is one of the original developers of SodaVision, an experimental face-recognition system at UC San Diego. SodaVision was built with OpenCV.