Volume 09 Issue 02 Published, May 19, 2005 ISSN 1535-864X DOI: 10.1535/itj.0902

Intel® Technology Journal

Compute-Intensive, Highly Parallel Applications and Uses

Learning-Based Computer Vision with Intel's Open Source Computer Vision Library

More information, including current and past issues of Intel Technology Journal, can be found at:
http://developer.intel.com/technology/itj/index.htm

Learning-Based Computer Vision with Intel's Open Source Computer Vision Library

Gary Bradski, Corporate Technology Group, Intel Corporation
Adrian Kaehler, Enterprise Platforms Group, Intel Corporation
Vadim Pisarevsky, Software and Solutions Group, Intel Corporation

Index words: computer vision, face recognition, road recognition, optimization, open source, OpenCV

ABSTRACT

The rapid expansion of computer processing power combined with the rapid development of digital camera capability has resulted in equally rapid advances in computer vision capability and use. Intel has long been at the forefront of enabling this advance on the computer hardware and software side. Computer vision software is supported by the free Open Source Computer Vision Library (OpenCV) that optionally may be highly optimized by loading the commercial Intel Integrated Performance Primitives (IPP). IPP now automatically supports OpenCV with no need to change or even recompile the user's source code. This functionality enables development groups to deploy vision and provides basic infrastructure to experts in vision.

OpenCV has long supported "geometric vision" from camera calibration, motion tracking in 2D, finding the camera location given a known 3D object, on up to producing depth maps from stereo vision. This paper describes using OpenCV for "learning-based vision," where objects such as faces, or patterns such as roads, are learned for segmentation and recognition.

INTRODUCTION

The Open Source Computer Vision Library (OpenCV) [1] is a free, open source collection of computer vision routines geared mainly towards human-computer interaction, robotics, security, and other vision applications where the lighting and context of use cannot be controlled. OpenCV is not geared towards factory machine vision where the environmental conditions can be controlled and one generally knows what one is looking for, although there is a large overlap.

OpenCV was designed for enablement and infrastructure. Many groups who could make use of vision were prevented from doing so due to lack of expertise; OpenCV enables these types of groups to add functionality such as face finding and tracking in a few lines of code. Other groups have vision expertise but were uselessly recreating vision algorithms that were already standard; OpenCV provides experts with a solid vision infrastructure and thereby allows experts to work at a higher level rather than have to worry about the basics. Because of the above, OpenCV's BSD type license is designed to promote free commercial and research use. Optionally, users may install the IPP libraries and benefit from highly optimized code without needing to recompile via the use of automatically selected optimized dynamic linked libraries.

OpenCV support for vision is extensive. It supports routines for input, display, and storage of movies and single images. Image-processing debug is supported by drawing and text display functions. Image processing itself is handled through convolution, thresholding, morphological operations, floodfills, histogramming, smoothing, pyramidal sub-sampling and a full suite of image algebra and arithmetic. Geometry is supported by Delaney triangulation, calibration, fundamental and essential matrices computation, image alignment, and stereo depth calculation. A full range of feature detection algorithms exists from corner detectors, Canny edge operators, blob finders, scale invariant features, and so on. Shape descriptors such as Hu moments, contour processing, Fourier descriptors, convex hulls, and connected components exist. Motion is covered via several types of optical flow, background learning and differencing, motion templates, and motion gradients. Learning-based vision is supported through feature histogram comparison, image statistics, template-based correlation, decision trees, and statistical boosting on up to convolutional neural networks.

OpenCV was released in Alpha in 2000, Beta in 2003, and will be released in official version 1.0 in Q4 2005. If the Intel Integrated Performance Primitives (IPP) library [2] is

optionally installed, OpenCV will automatically take advantage of and swap in the hand optimized routines in IPP providing a substantial speed-up to many vision routines.

In this paper, we describe computer vision routines based on learning. Learning-based vision has applications to image-based Web mining, image retrieval, video indexing, security, etc. OpenCV has strong and growing support for learning-based vision. We start out, however, by first discussing a recent change to OpenCV, full IPP support, and then move on to discuss two learning applications. We begin by describing automatic optimization of OpenCV using IPP and then we discuss using OpenCV for learned object finding and tracking (face), and end with abstract pattern segmentation (road finding).

AUTOMATIC OPTIMIZATION USING INTEGRATED PERFORMANCE PRIMITIVES

How to Make Use of IPP

Intel Integrated Performance Primitives (IPP) library is a large collection of low-level computational kernels highly optimized for Intel architectures, including the latest Pentium®, Itanium®, and XScale® processors. It consists of multiple domains that reside in separate dynamic libraries: signal and image processing, matrix processing, audio and video codecs, computer vision, speech recognition, cryptography, data compression, text processing, etc. It can be retrieved from http://www.intel.com/software/products/ipp [2]; full evaluation versions for Windows* and Linux*, and a free non-commercial version for Linux are available.

OpenCV is able to automatically detect and use the IPP library once the latter is installed; there is no need to recompile it. On Windows, make sure that the bin subdirectory of IPP is in the system path, for example, if IPP is installed to `"C:\Program Files\Intel\IPP"`, add `"C:\Program Files\Intel\IPP\bin"` to the path. On Linux the IPP dynamic libraries should be already in one of the standard paths after installation.

To check whether OpenCV has found IPP or not, the user application may call the `cvGetModuleInfo()` function:

® Pentium, Itanium, and XScale are all registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

* All other brands and names are the property of their respective owners.

```
const char* opencv_libraries = 0;
const char* addon_modules = 0;
cvGetModuleInfo( 0, &opencv_libraries,
                 &addon_modules );
printf( "OpenCV: %s\nAdd-on Modules: %s\n",
        opencv_libraries, addon_modules );
```

When IPP is detected, it will print something like this:

```
OpenCV: cxcore: beta 4.1 (0.9.7), cv: beta 4.1 (0.9.7)
Add-on modules: ippcv20.dll, ippi20.dll, ipps20.dll, ippvm20.dll
```

where ipp*.dll are names of IPP components, used by OpenCV: ippcv – computer vision, ippi – image processing, ipps – signal processing, ippvm – fast math functions.

Note that the functions in ippi20.dll and the other '20' libraries do not contain the processing functions themselves. They are proxies for CPU-specific libraries (ippia6.dll, ippiw7.dll etc. for IPPI) that are loaded by the IPP dispatcher. The dispatcher mechanism and other concepts behind IPP are explained in detail in the IPP book [3].

How Automatic Use of Optimized IPP Works

The mechanism to swap in optimized code if found is simple. It uses function pointers and dynamic library-loading facilities provided by the operating system (OS). For every IPP function that OpenCV can use there is a pointer to a function that is initially set to null and which is assigned to a valid address when the corresponding IPP component is detected and loaded. So, while OpenCV can benefit from using IPP, it does not depend on it; the functionality is the same, regardless of whether IPP is installed or not. That is, for every IPP function there is a backup C code that is always included inside OpenCV binaries. So, a higher-level external OpenCV function loads a function pointer that calls either optimized IPP code or embedded low-level OpenCV C code depending on which is available.

Let's consider an example. The function cvPyrDown reduces image size down one level by employing Gaussian smoothing and sub-sampling. Smoothing is preformed prior to sub-sampling so that spurious frequencies are not introduced due to violations of the Nyquist sampling theorem. cvPyrDown supports multiple image types via several lower-level functions. In particular, 8-bit single-channel images are processed with icvPyrDownG5x5_8u_CnR. The corresponding IPP function for this type of images is ippiPyrDown_Gauss5x5_8u_C1R.

So we have the following code (simplified compared to the actual implementation):

```
// --- cvpyramids.cpp: ---

// declaration of the function type.

// Matches to the declaration in IPP header
files

typedef int (CV_STDCALL *
icvPyrDown_Gauss5x5_8u_C1R_t)( const uchar* src,
int src_step, uchar* dst, int dst_step, CvSize
size, void* buffer );

// pointer to the IPP function

icvPyrDown_Gauss5x5_8u_C1R_t

        icvPyrDown_Gauss5x5_8u_C1R_p = 0;


// C implementation, embedded in OpenCV

static int icvPyrDownG5x5_8u_CnR(

    const uchar* src, int src_step,

    uchar* dst, int dst_step, CvSize size,

    void* buffer, int cn )

{

   …

   return CV_OK;

}


// external high-level function

void cvPyrDown( const CvArr* src_arr,

                CvArr* dst_arr,

                int filter )

{

    …

    if( data_type == CV_8UC1 ) {

        if( icvPyrDown_Gauss5x5_8u_C1R_p )

            icvPyrDown_Gauss5x5_8u_C1R_p(…);

        else

            icvPyrDownG5x5_8u_CnR(…,1);

    }

    …

}
```

Also, the function pointer and the related information are stored to the joint table that is used by the OpenCV initialization procedure (a.k.a. switcher).

```
//cvswitcher.cpp:

…

{ (void**)&icvPyrDown_Gauss5x5_8u_C1R_p, 0,

  "ippiPyrDown_Gauss5x5_8u_C1R",
CV_PLUGINS1(CV_PLUGIN_IPPI), 0 },

…
```

That is, each entry of the table contains a pointer to the function pointer (so that the address could be changed by the switcher), the real function name, and the id of the module that contains the function (IPPI ~ "ippi20.dll" in this case). On start-up, the OpenCV initialization procedure tries to locate and load IPP modules:

```
…

plugins[CV_PLUGIN_IPPI].handle =

    LoadLibrary("ippi20.dll");
```

and retrieve the function pointers:

```
for(…;…;…) {

   void* handle =
plugins[func_table[i].plugin_id].handle;

   const char* fname=func_table[i].func_names;

   if( handle )

      *func_table[i].func_addr =

           GetProcAddress( handle, fname );

}
```

(on Linux `dlopen` is used instead of `LoadLibrary` and `dlsym` instead of `GetProcAddress`).

Functionality Coverage

Currently, OpenCV knows of and can use over 300 IPP functions. Below is a table of the major functions and the approximate speed-up (on a Pentium 4 processor) that a user could get by on using IPP. Note that the wide range of speed-up numbers in the table results from different potential image types. Byte images are faster to process than integer images which are faster than floating, for example. Another timing difference results from different kernel sizes. Small kernels are faster than large kernels, and some common kernels are "hard wired"–hand optimized.

Table 1: Approximate speed-ups using assembly optimized IPP over the embedded optimized C in OpenCV

Function	Speed-up range (OpenCV/IPP exec. time)
Gaussian Pyramids	~3
Morphology	~3-7
Median filter	~2.1-18
Linear convolution (with a small kernel)	~2-8
Template Matching	~1.5-4
Color Conversion (RGB to/from Grayscale, HSV, Luv)	~1-3
Image moments	~1.5-3
Distance transform	~1.5-2
Image affine and perspective transformations	~1-4
Corner detection	~1.8
DFT/FFT/DCT	~1.5-3
Math functions (exp, log, sin, cos …)	3-10

In OpenCV 1.0, support for more IPP functions, such as face detection and optical flow, will be added.

FACE DETECTION

Introduction/Theory

Object detection, and in particular, face detection is an important element of various computer vision areas, such as image retrieval, shot detection, video surveillance, etc. The goal is to find an object of a pre-defined class in a static image or video frame. Sometimes this task can be accomplished by extracting certain image features, such as edges, color regions, textures, contours, etc. and then using some heuristics to find configurations and/or combinations of those features specific to the object of interest. But for complex objects, such as human faces, it is hard to find features and heuristics that will handle the huge variety of instances of the object class (e.g., faces may be slightly rotated in all three directions; some people wear glasses; some have moustaches or beards; often one half of the face is in the light and the other is shadow, etc.). For such objects, a statistical model (classifier) may be trained instead and then used to detect the objects.

Statistical model-based training takes multiple instances of the object class of interest, or "positive" samples, and multiple "negative" samples, i.e., images that do not contain objects of interest. Positive and negative samples together make a training set. During training, different features are extracted from the training samples and distinctive features that can be used to classify the object are selected. This information is "compressed" into the statistical model parameters. If the trained classifier does not detect an object (misses the object) or mistakenly detects the absent object (i.e., gives a false alarm), it is easy to make an adjustment by adding the corresponding positive or negative samples to the training set.

OpenCV uses such a statistical approach for object detection, an approach originally developed by Viola and Jones [4] and then analyzed and extended by Lienhart [5, 6]. This method uses simple Haar-like features (so called because they are computed similar to the coefficients in Haar wavelet transforms) and a cascade of boosted tree classifiers as a statistical model. In [4] and in OpenCV this method is tuned and primarily used for face detection. Therefore, we discuss face detection below, but a classifier for an arbitrary object class can be trained and used in exactly the same way.

The classifier is trained on images of fixed size (Viola uses 24x24 training images for face detection), and detection is done by sliding a search window of that size through the image and checking whether an image region at a certain location "looks like a face" or not. To detect faces of different size it is possible to scale the image, but the classifier has the ability to "scale" as well.

Fundamental to the whole approach are Haar-like features and a large set of very simple "weak" classifiers that use a single feature to classify the image region as face or non-face.

Each feature is described by the template (shape of the feature), its coordinate relative to the search window origin and the size (scale factor) of the feature. In [3], eight different templates were used, and in [5, 6] the set was extended to 14 templates, as shown in Figure 1.

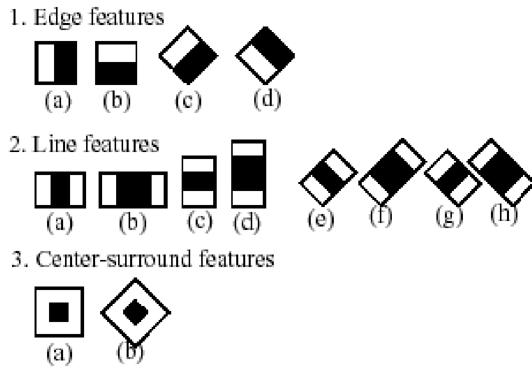


Figure 1: Extended set of Haar-like features

Each feature consists of two or three joined "black" and "white" rectangles, either up-right or rotated by 45°. The Haar feature's value is calculated as a weighted sum of two components: The pixel sum over the black rectangle and the sum over the whole feature area (all black and white rectangles). The weights of these two components are of opposite signs and for normalization, their absolute values are inversely proportional to the areas: for example, the black feature 3(a) in Figure 1 has $weight_{black} = -9 \times weight_{whole}$.

In real classifiers, hundreds of features are used, so direct computation of pixel sums over multiple small rectangles would make the detection very slow. But Viola [4] introduced an elegant method to compute the sums very fast. First, an integral image, Summed Area Table (*SAT*), is computed over the whole image *I*, where

$$SAT(X,Y) = \sum_{x<X,\, y<Y} I(x,y)\,.$$

The pixel sum over a rectangle $r=\{(x,y), x_0 \le x < x_0+w, y_0 \le y < y_0+h\}$ can then be computed using *SAT* by using just the corners of the rectangle regardless of size:

$$RecSum(r) = SAT(x_0+w, y_0+h) - SAT(x_0+w, y_0) - SAT(x_0, y_0+h) + SAT(x_0, y_0)$$

This is for up-right rectangles. For rotated rectangles, a separate "rotated" integral image must be used.

The computed feature value $x_i = w_{i,0} RecSum(r_{i,0}) + w_{i,1} RecSum(r_{i,1})$ is then used as input to a very simple decision tree classifier that usually has just two terminal nodes, that is:

$$f_i = \begin{cases} +1, & x_i \ge t_i \\ -1, & x_i < t_i \end{cases}$$

or three terminal nodes:

$$f_i = \begin{cases} +1, & t_{i,0} \le x_i < t_{i,1} \\ -1 & else \end{cases}$$

where the response +1 means the face, and −1 – means the non-face. Every such classifier, called a weak classifier, is not able to detect a face; rather, it reacts to some simple feature in the image that may relate to the face. For example, in many face images eyes are darker than the surrounding regions, and so feature 3a in Figure 1, centered at one of the eyes and properly scaled, will likely give a large response (assuming that $weight_{black} < 0$).

In the next step, a complex and robust classifier is built out of multiple weak classifiers using a procedure called boosting, introduced by Freund and Schapire [7].

The boosted classifier is built iteratively as a weighted sum of weak classifiers:

$$F = sign(c_1 f_1 + c_2 f_2 + \ldots + c_n f_n)$$

On each iteration, a new weak classifier f_i is trained and added to the sum. The smaller the error f_i gives on the training set, the larger is the coefficient c_i that is assigned to it. The weight of all the training samples is then updated, so that on the next iteration the role of those samples that are misclassified by the already built *F* are emphasized. It is proven in [7] that if f_i is even slightly more selective than just a random guess, then *F* can achieve an arbitrarily high (<1) hit rate and an arbitrarily small (>0) false alarm rate, if the number of weak classifiers in the sum (ensemble) is large enough. However, in practice, that would require a very large training set as well as a very large number of weak classifiers, resulting in a slow processing speed.

Instead, Viola [4] suggests building several boosted classifiers F_k with constantly increasing complexity and chaining them into a cascade with the simpler classifiers going first. During the detection stage, the current search window is analyzed subsequently by each of the F_k classifiers that may reject it or let it go through, as depicted in Figure 2.

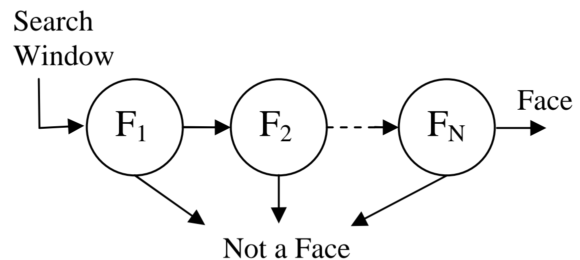


Figure 1: Object (face) detection cascade of classifiers where rejection can happen at any stage

That is, F_k (k=1..N)'s are subsequently applied to the face candidate until it gets rejected by one of them or until it passes them all. In experiments, about 70-80% of candidates are rejected in the first two stages that use the simplest features (about 10 weak classifiers each), so this technique speeds up detection greatly. Most of the detection time, therefore, is spent on real faces. Another advantage is that each of the stages need not be perfect; in fact, the stages are usually biased toward higher hit-rates rather than towards small false-alarm rates. By choosing the desired hit-rate and false-alarm rate at every stage and by choosing the number of stages accurately, it is possible to achieve very good detection performance. For example, if each of the stages gives a 0.999 hit-rate and a 0.5 false-alarm rate, then by stacking 20 stages into a cascade, we will be able to get a hit-rate of 0.999^{20}=0.98 and a false-alarm rate of 0.5^{20}~10^{-6}!

Face Detection with OpenCV

OpenCV provides low-level and high-level APIs for face/object detection. A low-level API allows users to check an individual location within the image by using the classifier cascade to find whether it contains a face or not. Helper functions calculate integral images and scale the cascade to a different face size (by scaling the coordinates of all rectangles of Haar-like features) etc. Alternatively, the higher-level function `cvDetectObjects` does this all automatically, and it is enough in most cases. Below is a sample of how to use this function to detect faces in a specified image:

```
// usage: facedetect --cascade=<path> image_name
#include "cv.h"
#include "highgui.h"
#include <string.h>

int main( int argc, char** argv )
{
    CvHaarClassifierCascade* cascade;

    // face sequence will reside in the storage
    CvMemStorage* storage=cvCreateMemStorage(0);
    IplImage *image;
    CvSeq* faces;
    int optlen = strlen("--cascade=");
    int i;

    if( argc != 3 ||
        strncmp(argv[1], "--cascade=", optlen) )
        return -1;

    // load classifier cascade from XML file
    cascade = (CvHaarClassifierCascade*)
        cvLoad( argv[1] + optlen );
    // load image from the specified file
    image = cvLoadImage( argv[2], 1 );

    if( !cascade || !image )
        return -1;


// get the sequence of face rectangles
faces = cvHaarDetectObjects( image,
          cascade, storage,
          1.2, // scale the cascade
               // by 20% after each pass
          2, // groups of 3 (2+1) or more
neighbor face rectangles are joined into a
single "face", smaller groups are rejected
          CV_HAAR_DO_CANNY_PRUNING, // use Canny
edge detector to reduce number of false alarms
          cvSize(0, 0) // start from the minimum
face size allowed by the particular classifier
    );

    // for each face draw the bounding rectangle
    for(i=0;i<(faces ? faces->total:0); i++ ) {
        CvRect* r = (CvRect*)
             cvGetSeqElem( faces, i );
        CvPoint pt1 = { r->x, r->y };
        CvPoint pt2 = { r->x + r->width,
                        r->y + r->height };
        cvRectangle( image, pt1, pt2,
                     CV_RGB(255,0,0), 3, 8, 0 );
    }

    // create window and show the image with
outlined faces
    cvNamedWindow( "faces", 1 );
    cvShowImage( "faces", image );
    cvWaitKey();
    // after a key pressed, release data
    cvReleaseImage( &image );
    cvReleaseHaarClassifierCascade( &cascade );
    cvReleaseMemStorage( &storage );
    return 0;
}
```

If the above program is built as facedetect.exe, it may be invoked as (type it in a single line):

```
facedetect.exe --cascade="c:\program
files\opencv\data\haarcascades\haarcascade_
frontalface_default.xml" "c:\program
files\opencv\samples\c\lena.jpg"
```

assuming that OpenCV is installed in c:\program files\opencv. Figure 3 shows example results of using a trained face detection model that ships with OpenCV.

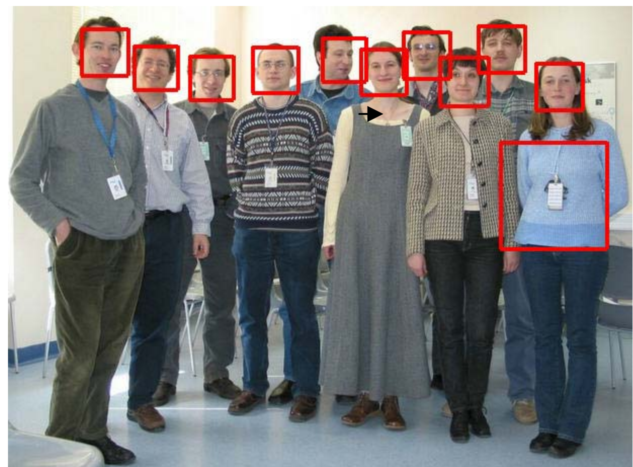


Figure 3: Results of a trained face detection model that ships with OpenCV. All faces were detected, one false positive detection resulted.

A detailed description of object detection functions can be found in the OpenCV reference manual (opencvref_cv.htm, *Object Detection* section).

Training the Classifier Cascade

Once there is a trained classifier cascade stored in an XML file, it can be easily loaded using the `cvLoad` function and then used by `cvHaarDetectObjects` or by low-level object detection functions. The question remains as to how to create such a classifier, if/when the standard cascades shipped with OpenCV fail on some images or one wants to detect some different object classes, like eyes, cars, etc. OpenCV includes a `haartraining` application that creates a classifier given a training set of positive and negative samples. The usage scheme is the following (for more details, refer to the haartraining reference manual supplied with OpenCV):

1. Collect a database of positive samples. Put them into one or more directories and create an index file that has the following format:

   filename_1 count_1 x11 y11 w11 h11 x12 y12 ...

   filename_2 count_2 x21 y21 w21 h21 x22 y22 ...

   ...

   That is, each line starts with a file name (including subdirectories) of an image followed by the number of objects in it and bounding rectangles for every object (x and y coordinates of top-left corner, width and height in pixels). For example, if a database of eyes resides in a directory eyes_base, the index file eyes.idx may look like this:

   ```
   eyes_base/eye_000.jpg  2 30 100  15  10 55 100 15 10
   eyes_base/eye_001.jpg 4 15 20 10 6 30 20 10 6 …
   …
   ```

   Notice that the performance of a trained classifier strongly depends on the quality of the database used. For example, for face detection, faces need to be aligned so that the relative locations of eyes–the most distinctive features–are the same. The eyes need to be on the same horizontal level (i.e., faces are properly rotated) etc. Another example is the detection of profile faces. These are non-symmetric, and it is reasonable to train the classifier only on right profiles (so that variance inside the object class is smaller) and at the detection stage to run it twice–once on the original images and a second time on the flipped images.

2. Build a vec-file out of the positive samples using the `createsamples` utility. While the training procedure might be repeated many times with different parameters, the same vec-file may be re-used.

   Example:

   ```
   createsamples –vec eyes.vec \
   –info eyes.idx –w 20 –h 15
   ```

   The above builds eyes.vec out of the database, described in eyes.idx (see above): all the positive samples are extracted from images, normalized and resized to the same size (20x15 in this case). `createsamples` can also creates a vec file out of a single positive sample (e.g., some company logo) by applying different geometrical transformations, adding noise, altering colors, etc. See *haartraining* in the OpenCV reference html manual for details.

3. Collect a database of negative samples. Make sure the database does not contain instances of the object class of interest. You can make negative samples out of arbitrary images, for example. They can be downloaded from the Internet, bought on CD, or shot by your digital camera. Put the images into one or more directories, and make an index file: that is, a plain list of image filenames, one per line. For example, an image index file called "backgrounds.idx" might contain:

   ```
   backgrounds/img0001.jpg
   backgrounds/my_img_02.bmp
   backgrounds/the_sea_picture.jpg
   …
   ```

4. Run haartraining. Below is an example (type it in command-line prompt as a single line or create a batch file):

   ```
   haartraining
   –data eyes_classifier_take_1
   –vec eyes.vec –w 20 –h 15
   –bg backgrounds.idx
   –nstages 15
   –nsplits 1
   [–nonsym]
   –minhitrate 0.995
   –maxfalsealarm 0.5
   ```

In this example, a classifier will be stored in eyes_classifier_take1.xml. eyes.vec is used as a set of positive samples (of size 20x15), and random images from background.idx are used as negative samples. The cascade will consist of 15 (-nstages) stages; every stage is trained

to have the specified hit-rate (-minhitrate) or higher, and a false-alarm rate (-maxfalsealarm) or lower. Every weak classifier will have just 1 (-nsplits) non-terminal node (1 split trees are called "stumps").

The training procedure may take several hours to complete even on a fast machine. The main reason is that there are quite a lot of different Haar features within the search window that need to be tried. However, this is essentially a parallel algorithm and it can benefit (and does benefit) from SMP-aware implementations. Haartraining supports OpenMP via the Intel Compiler and this parallel version is shipped with OpenCV.

We discussed use of an object detection/recognition algorithm built into OpenCV. In the next section, we discuss using OpenCV functions to recognize abstract objects such as roads.

ROAD SEGMENTATION

The Intel OpenCV library has been used for the vision system of an autonomous robot. This robot is built from a commercial off-road vehicle and the vision system is used to detect and follow roads. In this system, the problem was to use a close-by road, identified by scanning laser range finders to initialize vision algorithms that can extend the initial road segmentation out as far as possible. The roads in question were not limited to marked and paved streets; they were typically rocky trails, fire roads, and other poor quality dirt trails. Figure 4 shows "easy" and "hard" roads.

Figure 4: Example roads: a less difficult power-line access road (top) and a more difficult "off-road" trail (bottom)

Based on laser scanner point clouds in the near field, it was possible to estimate what sections of nearby visible terrain might be road. In this case, "near" is approximately ten meters (varying by terrain type and other ambient conditions). Once projected into the visual camera images, this region could then be used to train a classifier that would extrapolate out beyond the range of the lasers into the far visual field. In many cases the method is successful at extrapolating the road all of the way to the horizon. This amounts to a practical range of as much as one hundred meters. The ability to see into the visual far field is crucial for path planning for high-speed operation.

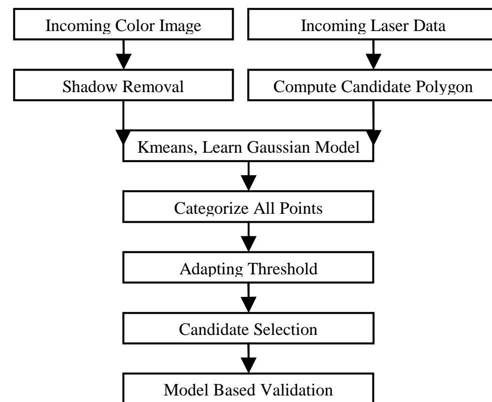


Figure 5: Overview of data flow

The core algorithm outlined in Figure 5 is as follows. First, the flat terrain that the lasers find immediately in front of the vehicle is converted to a single polygon and projected into the camera co-ordinates. Shadow regions are marked out of consideration as shown in Figure 6. The projected polygon represents our best starting guess for determining what pixels in the overall image contribute to road. It is of course possible that there is no road at all. The method is to extrapolate out to find the largest patch to which we can extend what the lasers have given us, and only thereafter to ask if that patch might be a road. This final determination will be made based on the shape of the area found. Only a relatively small set of shapes can correspond to a physical road of approximately constant width, disappearing into the distance.

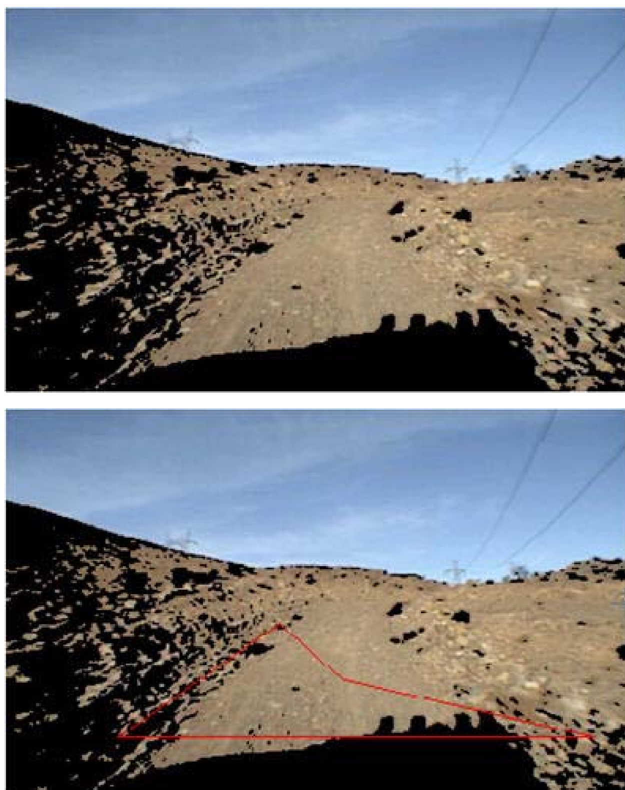


Figure 6: Starting with the bottom image of Figure 4, shadows are eliminated–blacked out– (top), and the polygon is projected from the laser data (bottom)

The red, green, blue (RGB) values of the individual pixels in the marked polygon are used as features to describe individual pixels in the marked region (giving a total of three dimensions). These pixels are kept in a FIFO circular buffer of sufficient depth to hold pixels from many frames. At every frame the OpenCV function cvKMeans2 is called with settings to find three clusters. These clusters are modeled with three multivariate Gaussian models trained from the pixel density distribution in the 3-dimensional color space. This algorithm achieves the same results as fitting three multivariate Gaussians using expectation maximization, but is faster and takes advantage of the built in cvKMeans2 in OpenCV. Three clusters are used because empirically it was found that ruts and rocks in the road tend to give dirt roads a tri-tonal color profile.

After generating the above model, we then use it to score the remaining pixels in the scene by using the OpenCV function cvMahalanobis to find the minimum Mahalanobis distance R_m from the candidate pixel to the means of the three learned multi-modal Gaussian distributions. Pixels for which R_m is less than 1 are assigned a score of 1.0; those for which R_m is greater than 1 are given a score of $1/R_m$. The resulting "image" contains a kind of confidence score for every pixel being road or not. This scoring is essentially equivalent to a log-likelihood score for a pixel being in the road distribution, but again is fast and convenient to compute using cvMahalanobis.

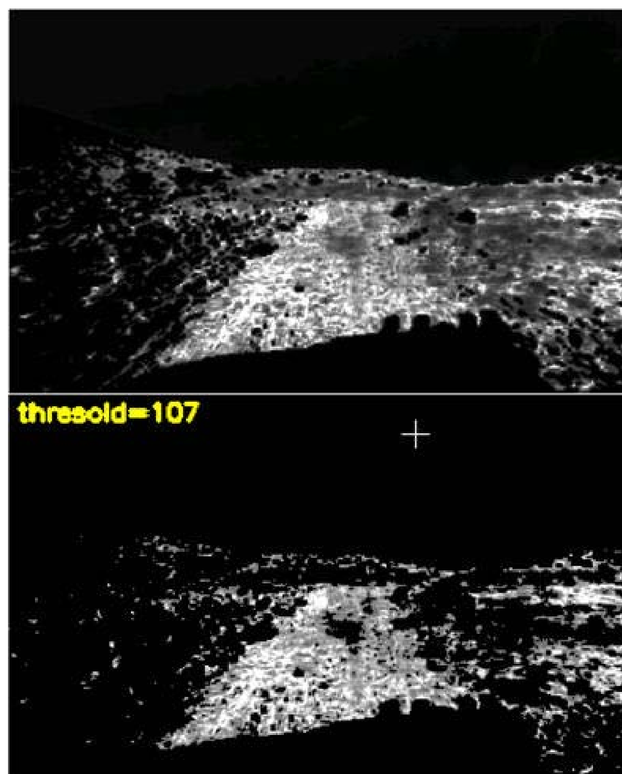


Figure 7: Probability map based on individual pixel's distance from the model mean (top), and image with low probability pixels removed (bottom). In the thresholded bottom image, the threshold is set so as to keep constant the proportion of non-zero pixels in the original

This method does not guarantee that all pixels in the polygon training area will be scored as road. OpenCV function cvThreshold is used adaptively such that no less than 80% of the pixels in the training area are retained after the threshold is applied, as shown in Figure 7. A morphological function cvDilate is then applied with the result that the identified pixels can be clustered into large connected regions. As we are only interested in extrapolating the already identified space found by the lasers, clusters are next rejected if they do not connect with the polygonal training area cluster. The retained clusters are then used to mask the original confidence image. This process is illustrated in Figure 8.

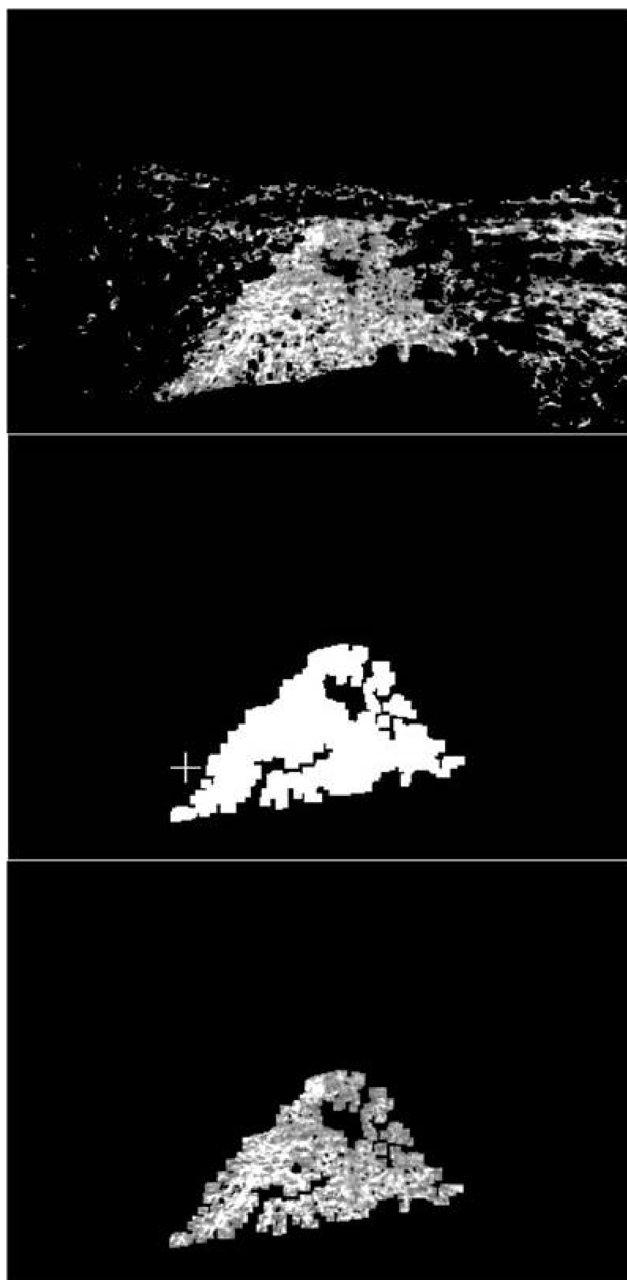
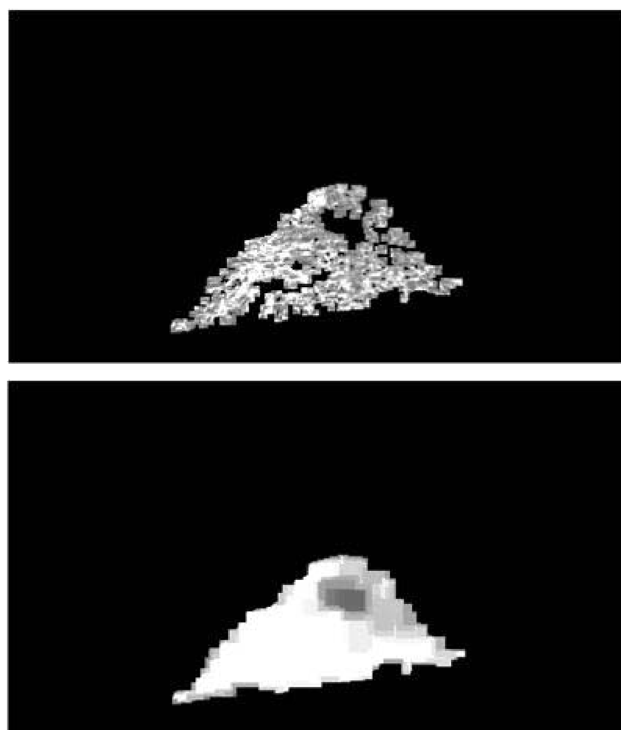


Figure 8: Starting with the thresholded image (top), the contiguous patch that intersects the training polygon is identified (middle) and clipped from the original probability map (bottom)

Finally, in Figure 9, the resulting clusters are compared with the expected shape of a road projected into the camera coordinates. This is done by fitting the right and left sides of the cluster(s) to individual lines, and by computing an effective horizon which is the location of the highest identified road pixel. If these fits are sufficiently good, the fit lines intersect at the horizon, and the angles of the lines are consistent with the projection of a road into the camera coordinates, then the system reports the confidence image. This image, once projected into ground-based coordinates, is used to mark the road ahead for the subsequent path-planning operations.

By building this model, it is possible to reject anomalies that might arise from dirt, facing into the sun, reflections, and other sources. Such anomalies are not uncommon, and the ability to build a geometrically possible model and then score that model allows the system to report correctly when the found pixels are something other than a road. In some cases this occurs when no road is present at all, and the lasers find only a smooth patch of ground that is not part of an actual road. In three hours of test video varying widely over many terrain types containing both road and no road, the algorithm was not seen to give a false positive signal.



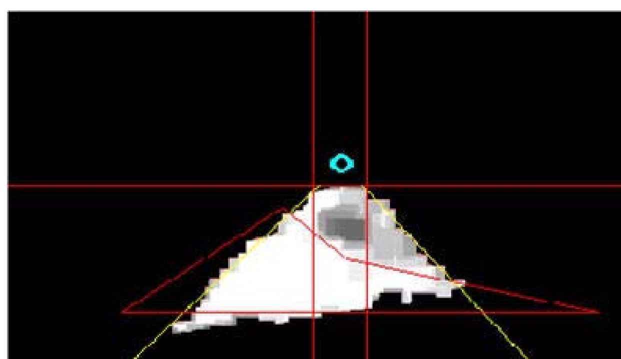


Figure 9: Starting with the clipped probability map (one), the image is first blurred (two), and the edges are fit to straight lines (three). Combining these lines with the effective horizon, a model for the road is constructed (four). A confidence score is assigned based on the number of pixels correctly classified by the implied road structure (four).

The algorithm presented has been tested against terrain types of widely varying difficulty. All of the images shown here were from a 320x240 camera image, and they were capable of being generated at ten frames per second on a computer based on the 1.7 GHz Mobile Intel Pentium 4 processor – M that was selected for its low-power requirements in the robot. For roads that are well defined, flat, and have a relatively uniform surface texture, the method allows extrapolation of the entire road to the horizon.

DISCUSSION

We started out describing recent developments with the OpenCV library [1] that allow it to be automatically optimized by using OpenCV with the IPP libraries [2], and we described how automatic optimization works through the cvswitcher function. To recap, the switcher is given a table of function identifiers and pointers to those functions. During launch, the switcher looks for the appropriate optimized IPP code for the machine it is running on and swaps in the address of those functions if the correct IPP module is found. Otherwise it retains the embedded optimized C functions. Because the source code for the switcher is available in the file cvswitcher.cpp, one may also override this functionality with custom functionality–by pointing it to one's own functions or alternate functions, depending on the processor used.

We then focused on a small part of OpenCV's support for learning-based vision. The Haar-feature-based boosted classifier cascade, based on the work of Viola and others [4, 5, 6] that is directly implemented in OpenCV, was described. When you download OpenCV, this classifier comes with a file that is trained to recognize faces and may be run by building and running the facedetect.c code that comes in the samples directory, which will normally be placed in C:\Program Files\OpenCV\samples\c on Windows machines when you install OpenCV. The procedure for using your own data to train a classifier cascade to recognize any object was then described. Using this, one can create tree, car, cat, or person detectors.

Note that one may make the classifier more flexible by using instead of, or in addition to, learning features from raw images; image processing can be used to enhance or pre-select images of features such as gradient magnitudes to be learned. This can help reduce lighting sensitivity, for example.

We then shifted gears to describe a project aimed at detecting and segmenting an abstract object–a road. This involved using laser range data to identify a polygon of a nearby road to use as a seed for learning and modeling clusters of road-colored pixels. This learned model was then used to segment the road beyond the range of the laser range finders. This is essentially a local model of what pixels "look" like road. Global geometric constraints are then used to only accept road segmentations that in fact look like the shape of a road as in Figure 9 bottom. The segmented road pixels are then projected along with the laser data into a bird's eye view 2D path planner. Vision significantly extends the range of road detection out from the lasers and so allows the robot to travel at much higher speeds. Using the local and global constraints together, false positive road identification was completely eliminated.

In the above, we just touched on the learning-based vision possibilities of OpenCV. Many other techniques are supported. For example OpenCV contains methods for gathering and matching histograms of image features; learning the means and variances of pixels over a moving window of time; Kalman and Particle ("Condensation") filters for tracking, and data that can be clustered or tracked with the Meanshift or CAMShift [8] algorithms. In Release version 1.0 several more machine-learning algorithms for computer vision should become available such as convolutional neural networks, general back propagation, general decision trees, several methods of statistical boosting, and random forests. These new algorithms will enable powerful approaches to learning-based vision.

CONCLUSION

Computer vision, unlike for example factory machine vision, happens in unconstrained environments, potentially with changing cameras and changing lighting and camera views. Also, some "objects" such as roads, rivers, bushes, etc. are just difficult to describe. In these situations, engineering a model *a-priori* can be difficult.

With learning-based vision, one just "points" the algorithm at the data and useful models for detection, segmentation, and identification can often be formed. Learning can often easily fuse or incorporate other sensing modalities such as sound, vibration, or heat. Since cameras and sensors are becoming cheap and powerful and learning algorithms have a vast appetite for computational threads, Intel is very interested in enabling geometric and learning-based vision routines in its OpenCV library since such routines are vast consumers of computational power.

REFERENCES

[1] Open Source Computer Vision Library: http://www.intel.com/research/mrl/research/opencv

[2] Intel® Integrated Performance Primitives http://www.intel.com/software/products/perflib

[3] Stewart Taylor, "Intel® Integrated Performance Primitives," in *How to Optimize Software Applications Using Intel® IPP*" http://www.intel.com/intelpress/sum_ipp.htm

[4] Paul Viola and Michael J. Jones, "Rapid Object Detection using a Boosted Cascade of Simple Features," *IEEE CVPR*, 2001.

[5] Rainer Lienhart and Jochen Maydt, "An Extended Set of Haar-like Features for Rapid Object Detection," Submitted to *ICIP2002*.

[6] Alexander Kuranov, Rainer Lienhart, and Vadim Pisarevsky, "An Empirical Analysis of Boosting Algorithms for Rapid Objects With an Extended Set of Haar-like Features," *Intel Technical Report* MRL-TR-July02-01, 2002.

[7] Freund, Y. and Schapire, R. E. (1996b), "Experiments with a new boosting algorithm," in *Machine Learning: Proceedings of the Thirteenth International Conference*, Morgan Kauman, San Francisco, pp. 148-156, 1996.

[8] Bradski, G., "Computer Vision Face Tracking For Use in a Perceptual User Interface," *Intel Technology Journal*,http://developer.intel.com/technology/itj/q21998/articles/art_2.htm, Q2 1998.

AUTHORS' BIOGRAPHIES

Gary Rost Bradski is a principal engineer and manager of the Machine Learning group for Intel Research. His current interests are learning-based vision and sensor fusion in world models. Gary received a B.S. degree from U.C. Berkeley in May, 1981. He received his Ph.D. degree in Cognitive and Neural Systems (mathematical modeling of biological perception) in May, 1994 from Boston University Center for Adaptive Systems. He started and was the technical content director of OpenCV working closely with Vadim. Currently, he consults on OpenCV and machine learning content with the performance primitives group. His e-mail is Garybradski at gmail.com.

Adrian Kaehler is a senior software engineer working in the Enterprise Platforms Group. His interests include machine learning, statistical modeling, and computer vision. Adrian received a B.A. degree in Physics from the University of California at Santa Cruz in 1992 and his Ph.D. degree in Theoretical Physics from Columbia University in 1998. Currently, Adrian is involved with a variety of vision-related projects in and outside of Intel. His e-mail is Adrian.l.Kaehler at intel.com.

Vadim Pisarevsky is a software engineer in the Computational Software Lab in Intel. His interests are in image processing, computer vision, machine learning, algorithm optimization, and programming languages. Vadim received a Masters degree in Mathematics from the Nizhny Novgorod State University, in 1998. He has been involved in different software projects related to multimedia processing since 1996. In 2000, he joined Intel Russia Research Center where he led the OpenCV development team for over four years. Currently, he is working in the software department and continues to improve OpenCV and its integration with Intel Integrated Performance Primitives. His e-mail is Vadim.Pisarevsky at intel.com.

Copyright © Intel Corporation 2005. This publication was downloaded from http://developer.intel.com/.

Legal notices at http://www.intel.com/sites/corporate/tradmarx.htm.

For further information visit:

developer.intel.com/technology/itj/index.htm

Copyright © 2005 Intel Corporation. All rights reserved.
Intel is a trademark or registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.
For a complete listing of trademark information visit: www.intel.com/sites/corporate/tradmarx.htm