The OpenCV User Guide

Release 2.4.13.3

September 25, 2017

CHAPTER

ONE

OPERATIONS WITH IMAGES

Input/Output

Images

Load an image from a file:

```
Mat img = imread(filename)
```

If you read a jpg file, a 3 channel image is created by default. If you need a grayscale image, use:

```
Mat img = imread(filename, 0);
```

Note: format of the file is determined by its content (first few bytes)

Save an image to a file:

```
imwrite(filename, img);
```

Note: format of the file is determined by its extension.

Note: use `imdecode` and `imencode` to read and write image from/to memory rather than a file.

XML/YAML

TBD

Basic operations with images

Accessing pixel intensity values

In order to get pixel intensity value, you have to know the type of an image and the number of channels. Here is an example for a single channel grey scale image (type 8UC1) and pixel coordinates x and y:

```
Scalar intensity = img.at<uchar>(y, x);
```

`intensity.val[0]` contains a value from 0 to 255. Note the ordering of `x` and `y`. Since in OpenCV images are represented by the same structure as matrices, we use the same convention for both cases - the 0-based row index (or y-coordinate) goes first and the 0-based column index (or x-coordinate) follows it. Alternatively, you can use the following notation:

```
Scalar intensity = img.at<uchar>(Point(x, y));
```

Now let us consider a 3 channel image with `BGR` color ordering (the default format returned by `imread`):

```
Vec3b intensity = img.at<Vec3b>(y, x);
uchar blue = intensity.val[0];
uchar green = intensity.val[1];
uchar red = intensity.val[2];
```

You can use the same method for floating-point images (for example, you can get such an image by running Sobel on a 3 channel image):

```
Vec3f intensity = img.at<Vec3f>(y, x);
float blue = intensity.val[0];
float green = intensity.val[1];
float red = intensity.val[2];
```

The same method can be used to change pixel intensities:

```
img.at<uchar>(y, x) = 128;
```

There are functions in OpenCV, especially from calib3d module, such as `projectPoints`, that take an array of 2D or 3D points in the form of `Mat`. Matrix should contain exactly one column, each row corresponds to a point, matrix type should be 32FC2 or 32FC3 correspondingly. Such a matrix can be easily constructed from `std::vector`:

```
vector<Point2f> points;
//... fill the array
Mat pointsMat = Mat(points);
```

One can access a point in this matrix using the same method `Mat::at` :

```
Point2f point = pointsMat.at<Point2f>(i, 0);
```

Memory management and reference counting

`Mat` is a structure that keeps matrix/image characteristics (rows and columns number, data type etc) and a pointer to data. So nothing prevents us from having several instances of `Mat` corresponding to the same data. A `Mat` keeps a reference count that tells if data has to be deallocated when a particular instance of `Mat` is destroyed. Here is an example of creating two matrices without copying data:

```
std::vector<Point3f> points;
// .. fill the array
Mat pointsMat = Mat(points).reshape(1);
```

As a result we get a 32FC1 matrix with 3 columns instead of 32FC3 matrix with 1 column. `pointsMat` uses data from `points` and will not deallocate the memory when destroyed. In this particular instance, however, developer has to make sure that lifetime of `points` is longer than of `pointsMat`. If we need to copy the data, this is done using, for example, `Mat::copyTo` or `Mat::clone`:

```
Mat img = imread("image.jpg");
Mat img1 = img.clone();
```

To the contrary with C API where an output image had to be created by developer, an empty output `Mat` can be supplied to each function. Each implementation calls `Mat::create` for a destination matrix. This method allocates data for a matrix if it is empty. If it is not empty and has the correct size and type, the method does nothing. If, however, size or type are different from input arguments, the data is deallocated (and lost) and a new data is allocated. For example:

```
Mat img = imread("image.jpg");
Mat sobelx;
Sobel(img, sobelx, CV_32F, 1, 0);
```

Primitive operations

There is a number of convenient operators defined on a matrix. For example, here is how we can make a black image from an existing greyscale image `img`:

```
img = Scalar(0);
```

Selecting a region of interest:

```
Rect r(10, 10, 100, 100);
Mat smallImg = img(r);
```

A convertion from `Mat` to C API data structures:

```
Mat img = imread("image.jpg");
IplImage img1 = img;
CvMat m = img;
```

Note that there is no data copying here.

Conversion from color to grey scale:

```
Mat img = imread("image.jpg"); // loading a 8UC3 image
Mat grey;
cvtColor(img, grey, CV_BGR2GRAY);
```

Change image type from 8UC1 to 32FC1:

```
src.convertTo(dst, CV_32F);
```

Visualizing images

It is very useful to see intermediate results of your algorithm during development process. OpenCV provides a convenient way of visualizing images. A 8U image can be shown using:

```
Mat img = imread("image.jpg");

namedWindow("image", CV_WINDOW_AUTOSIZE);
imshow("image", img);
waitKey();
```

A call to `waitKey()` starts a message passing cycle that waits for a key stroke in the `"image"` window. A `32F` image needs to be converted to `8U` type. For example:

```
Mat img = imread("image.jpg");
Mat grey;
cvtColor(img, grey, CV_BGR2GRAY);

Mat sobelx;
Sobel(grey, sobelx, CV_32F, 1, 0);

double minVal, maxVal;
minMaxLoc(sobelx, &minVal, &maxVal); //find minimum and maximum intensities
Mat draw;
sobelx.convertTo(draw, CV_8U, 255.0/(maxVal - minVal), -minVal * 255.0/(maxVal - minVal));

namedWindow("image", CV_WINDOW_AUTOSIZE);
imshow("image", draw);
waitKey();
```

CHAPTER

TWO

FEATURES2D

Detectors

Descriptors

Matching keypoints

The code

We will start with a short sample `opencv/samples/cpp/matcher_simple.cpp`:

```
Mat img1 = imread(argv[1], CV_LOAD_IMAGE_GRAYSCALE);
Mat img2 = imread(argv[2], CV_LOAD_IMAGE_GRAYSCALE);
if(img1.empty() || img2.empty())
{
    printf("Can't read one of the images\n");
    return -1;
}

// detecting keypoints
SurfFeatureDetector detector(400);
vector<KeyPoint> keypoints1, keypoints2;
detector.detect(img1, keypoints1);
detector.detect(img2, keypoints2);

// computing descriptors
SurfDescriptorExtractor extractor;
Mat descriptors1, descriptors2;
extractor.compute(img1, keypoints1, descriptors1);
extractor.compute(img2, keypoints2, descriptors2);

// matching descriptors
BruteForceMatcher<L2<float> > matcher;
vector<DMatch> matches;
matcher.match(descriptors1, descriptors2, matches);

// drawing the results
namedWindow("matches", 1);
Mat img_matches;
drawMatches(img1, keypoints1, img2, keypoints2, matches, img_matches);
imshow("matches", img_matches);
waitKey(0);
```

The code explained

Let us break the code down.

```
Mat img1 = imread(argv[1], CV_LOAD_IMAGE_GRAYSCALE);
Mat img2 = imread(argv[2], CV_LOAD_IMAGE_GRAYSCALE);
if(img1.empty() || img2.empty())
{
    printf("Can't read one of the images\n");
    return -1;
}
```

We load two images and check if they are loaded correctly.:

```
// detecting keypoints
FastFeatureDetector detector(15);
vector<KeyPoint> keypoints1, keypoints2;
detector.detect(img1, keypoints1);
detector.detect(img2, keypoints2);
```

First, we create an instance of a keypoint detector. All detectors inherit the abstract `FeatureDetector` interface, but the constructors are algorithm-dependent. The first argument to each detector usually controls the balance between the amount of keypoints and their stability. The range of values is different for different detectors (For instance, *FAST* threshold has the meaning of pixel intensity difference and usually varies in the region *[0,40]*. *SURF* threshold is applied to a Hessian of an image and usually takes on values larger than *100*), so use defaults in case of doubt.

```
// computing descriptors
SurfDescriptorExtractor extractor;
Mat descriptors1, descriptors2;
extractor.compute(img1, keypoints1, descriptors1);
extractor.compute(img2, keypoints2, descriptors2);
```

We create an instance of descriptor extractor. The most of OpenCV descriptors inherit `DescriptorExtractor` abstract interface. Then we compute descriptors for each of the keypoints. The output `Mat` of the `DescriptorExtractor::compute` method contains a descriptor in a row *i* for each *i*-th keypoint. Note that the method can modify the keypoints vector by removing the keypoints such that a descriptor for them is not defined (usually these are the keypoints near image border). The method makes sure that the ouptut keypoints and descriptors are consistent with each other (so that the number of keypoints is equal to the descriptors row count).

```
// matching descriptors
BruteForceMatcher<L2<float> > matcher;
vector<DMatch> matches;
matcher.match(descriptors1, descriptors2, matches);
```

Now that we have descriptors for both images, we can match them. First, we create a matcher that for each descriptor from image 2 does exhaustive search for the nearest descriptor in image 1 using Euclidean metric. Manhattan distance is also implemented as well as a Hamming distance for Brief descriptor. The output vector `matches` contains pairs of corresponding points indices.

```
// drawing the results
namedWindow("matches", 1);
Mat img_matches;
drawMatches(img1, keypoints1, img2, keypoints2, matches, img_matches);
imshow("matches", img_matches);
waitKey(0);
```

The final part of the sample is about visualizing the matching results.

CHAPTER

THREE

KINECT AND OPENNI

Using Kinect and other OpenNI compatible depth sensors

Depth sensors compatible with OpenNI (Kinect, XtionPRO, ...) are supported through `VideoCapture` class. Depth map, BGR image and some other formats of output can be retrieved by using familiar interface of `VideoCapture`.

In order to use depth sensor with OpenCV you should do the following preliminary steps:

1. Install OpenNI library (from here http://www.openni.org/downloadfiles) and PrimeSensor Module for OpenNI (from here https://github.com/avin2/SensorKinect). The installation should be done to default folders listed in the instructions of these products, e.g.:

```
OpenNI:
    Linux & MacOSX:
        Libs into: /usr/lib
        Includes into: /usr/include/ni
    Windows:
        Libs into: c:/Program Files/OpenNI/Lib
        Includes into: c:/Program Files/OpenNI/Include
PrimeSensor Module:
    Linux & MacOSX:
        Bins into: /usr/bin
    Windows:
        Bins into: c:/Program Files/Prime Sense/Sensor/Bin
```

   If one or both products were installed to the other folders, the user should change corresponding CMake variables `OPENNI_LIB_DIR`, `OPENNI_INCLUDE_DIR` or/and `OPENNI_PRIME_SENSOR_MODULE_BIN_DIR`.

2. Configure OpenCV with OpenNI support by setting `WITH_OPENNI` flag in CMake. If OpenNI is found in install folders OpenCV will be built with OpenNI library (see a status `OpenNI` in CMake log) whereas PrimeSensor Modules can not be found (see a status `OpenNI PrimeSensor Modules` in CMake log). Without PrimeSensor module OpenCV will be successfully compiled with OpenNI library, but `VideoCapture` object will not grab data from Kinect sensor.

3. Build OpenCV.

VideoCapture can retrieve the following data:

1. **data given from depth generator:**

    - `CV_CAP_OPENNI_DEPTH_MAP` - depth values in mm (CV_16UC1)
    - `CV_CAP_OPENNI_POINT_CLOUD_MAP` - XYZ in meters (CV_32FC3)
    - `CV_CAP_OPENNI_DISPARITY_MAP` - disparity in pixels (CV_8UC1)
    - `CV_CAP_OPENNI_DISPARITY_MAP_32F` - disparity in pixels (CV_32FC1)

- CV_CAP_OPENNI_VALID_DEPTH_MASK - mask of valid pixels (not ocluded, not shaded etc.) (CV_8UC1)

2. **data given from BGR image generator:**

   - CV_CAP_OPENNI_BGR_IMAGE - color image (CV_8UC3)
   - CV_CAP_OPENNI_GRAY_IMAGE - gray image (CV_8UC1)

In order to get depth map from depth sensor use `VideoCapture::operator >>`, e. g.

```
VideoCapture capture( CV_CAP_OPENNI );
for(;;)
{
    Mat depthMap;
    capture >> depthMap;

    if( waitKey( 30 ) >= 0 )
        break;
}
```

For getting several data maps use `VideoCapture::grab` and `VideoCapture::retrieve`, e.g.

```
VideoCapture capture(0); // or CV_CAP_OPENNI
for(;;)
{
    Mat depthMap;
    Mat bgrImage;

    capture.grab();

    capture.retrieve( depthMap, CV_CAP_OPENNI_DEPTH_MAP );
    capture.retrieve( bgrImage, CV_CAP_OPENNI_BGR_IMAGE );

    if( waitKey( 30 ) >= 0 )
        break;
}
```

For setting and getting some property of sensor‘ data generators use `VideoCapture::set` and `VideoCapture::get` methods respectively, e.g.

```
VideoCapture capture( CV_CAP_OPENNI );
capture.set( CV_CAP_OPENNI_IMAGE_GENERATOR_OUTPUT_MODE, CV_CAP_OPENNI_VGA_30HZ );
cout << "FPS    " << capture.get( CV_CAP_OPENNI_IMAGE_GENERATOR+CV_CAP_PROP_FPS ) << endl;
```

Since two types of sensor’s data generators are supported (image generator and depth generator), there are two flags that should be used to set/get property of the needed generator:

- CV_CAP_OPENNI_IMAGE_GENERATOR – A flag for access to the image generator properties.
- CV_CAP_OPENNI_DEPTH_GENERATOR – A flag for access to the depth generator properties. This flag value is assumed by default if neither of the two possible values of the property is not set.

Some depth sensors (for example XtionPRO) do not have image generator. In order to check it you can get `CV_CAP_OPENNI_IMAGE_GENERATOR_PRESENT` property.

```
bool isImageGeneratorPresent = capture.get( CV_CAP_PROP_OPENNI_IMAGE_GENERATOR_PRESENT ) != 0; // or == 1
```

Flags specifing the needed generator type must be used in combination with particular generator property. The following properties of cameras available through OpenNI interfaces are supported:

- For image generator:

  - CV_CAP_PROP_OPENNI_OUTPUT_MODE – Three output modes are supported: CV_CAP_OPENNI_VGA_30HZ used by default (image generator returns images in VGA resolution with 30 FPS), CV_CAP_OPENNI_SXGA_15HZ (image generator returns images in SXGA resolution with 15 FPS) and CV_CAP_OPENNI_SXGA_30HZ (image generator returns images in SXGA resolution with 30 FPS, the mode is supported by XtionPRO Live); depth generator's maps are always in VGA resolution.

- For depth generator:

  - CV_CAP_PROP_OPENNI_REGISTRATION – Flag that registers the remapping depth map to image map by changing depth generator's view point (if the flag is "on") or sets this view point to its normal one (if the flag is "off"). The registration process's resulting images are pixel-aligned,which means that every pixel in the image is aligned to a pixel in the depth image.

    Next properties are available for getting only:

  - CV_CAP_PROP_OPENNI_FRAME_MAX_DEPTH – A maximum supported depth of Kinect in mm.
  - CV_CAP_PROP_OPENNI_BASELINE – Baseline value in mm.
  - CV_CAP_PROP_OPENNI_FOCAL_LENGTH – A focal length in pixels.
  - CV_CAP_PROP_FRAME_WIDTH – Frame width in pixels.
  - CV_CAP_PROP_FRAME_HEIGHT – Frame height in pixels.
  - CV_CAP_PROP_FPS – Frame rate in FPS.

- Some typical flags combinations "generator type + property" are defined as single flags:

  - CV_CAP_OPENNI_IMAGE_GENERATOR_OUTPUT_MODE = CV_CAP_OPENNI_IMAGE_GENERATOR + CV_CAP_PROP_OPENNI_OUTPUT_MODE
  - CV_CAP_OPENNI_DEPTH_GENERATOR_BASELINE = CV_CAP_OPENNI_DEPTH_GENERATOR + CV_CAP_PROP_OPENNI_BASELINE
  - CV_CAP_OPENNI_DEPTH_GENERATOR_FOCAL_LENGTH = CV_CAP_OPENNI_DEPTH_GENERATOR + CV_CAP_PROP_OPENNI_FOCAL_LENGTH
  - CV_CAP_OPENNI_DEPTH_GENERATOR_REGISTRATION = CV_CAP_OPENNI_DEPTH_GENERATOR + CV_CAP_PROP_OPENNI_REGISTRATION

For more information please refer to the example of usage openni_capture.cpp in opencv/samples/cpp folder.

CHAPTER

FOUR

CASCADE CLASSIFIER TRAINING

Introduction

The work with a cascade classifier includes two major stages: training and detection. Detection stage is described in a documentation of `objdetect` module of general OpenCV documentation. Documentation gives some basic information about cascade classifier. Current guide is describing how to train a cascade classifier: preparation of a training data and running the training application.

Important notes

There are two applications in OpenCV to train cascade classifier: `opencv_haartraining` and `opencv_traincascade`. `opencv_traincascade` is a newer version, written in C++ in accordance to OpenCV 2.x API. But the main difference between this two applications is that `opencv_traincascade` supports both Haar *[Viola2001]* and LBP *[Liao2007]* (Local Binary Patterns) features. LBP features are integer in contrast to Haar features, so both training and detection with LBP are several times faster then with Haar features. Regarding the LBP and Haar detection quality, it depends on training: the quality of training dataset first of all and training parameters too. It's possible to train a LBP-based classifier that will provide almost the same quality as Haar-based one.

`opencv_traincascade` and `opencv_haartraining` store the trained classifier in different file formats. Note, the newer cascade detection interface (see `CascadeClassifier` class in `objdetect` module) support both formats. `opencv_traincascade` can save (export) a trained cascade in the older format. But `opencv_traincascade` and `opencv_haartraining` can not load (import) a classifier in another format for the further training after interruption.

Note that `opencv_traincascade` application can use TBB for multi-threading. To use it in multicore mode OpenCV must be built with TBB.

Also there are some auxiliary utilities related to the training.

- `opencv_createsamples` is used to prepare a training dataset of positive and test samples. `opencv_createsamples` produces dataset of positive samples in a format that is supported by both `opencv_haartraining` and `opencv_traincascade` applications. The output is a file with *.vec extension, it is a binary format which contains images.
- `opencv_performance` may be used to evaluate the quality of classifiers, but for trained by `opencv_haartraining` only. It takes a collection of marked up images, runs the classifier and reports the performance, i.e. number of found objects, number of missed objects, number of false alarms and other information.

Since `opencv_haartraining` is an obsolete application, only `opencv_traincascade` will be described further. `opencv_createsamples` utility is needed to prepare a training data for `opencv_traincascade`, so it will be described too.

`opencv_createsamples` utility

An `opencv_createsamples` utility provides functionality for dataset generating, writing and viewing. The term *dataset* is used here for both training set and test set.

Training data preparation

For training we need a set of samples. There are two types of samples: negative and positive. Negative samples correspond to non-object images. Positive samples correspond to images with detected objects. Set of negative samples must be prepared manually, whereas set of positive samples is created using `opencv_createsamples` utility.

Negative Samples

Negative samples are taken from arbitrary images. These images must not contain detected objects. Negative samples are enumerated in a special file. It is a text file in which each line contains an image filename (relative to the directory of the description file) of negative sample image. This file must be created manually. Note that negative samples and sample images are also called background samples or background samples images, and are used interchangeably in this document. Described images may be of different sizes. But each image should be (but not necessarily) larger then a training window size, because these images are used to subsample negative image to the training size.

An example of description file:

Directory structure:

```
/img
  img1.jpg
  img2.jpg
bg.txt
```

File bg.txt:

```
img/img1.jpg
img/img2.jpg
```

Positive Samples

Positive samples are created by `opencv_createsamples` utility. They may be created from a single image with object or from a collection of previously marked up images.

Please note that you need a large dataset of positive samples before you give it to the mentioned utility, because it only applies perspective transformation. For example you may need only one positive sample for absolutely rigid object like an OpenCV logo, but you definitely need hundreds and even thousands of positive samples for faces. In the case of faces you should consider all the race and age groups, emotions and perhaps beard styles.

So, a single object image may contain a company logo. Then a large set of positive samples is created from the given object image by random rotating, changing the logo intensity as well as placing the logo on arbitrary background. The amount and range of randomness can be controlled by command line arguments of `opencv_createsamples` utility.

Command line arguments:

- -vec <vec_file_name>

  Name of the output file containing the positive samples for training.

- -img <image_file_name>

Source object image (e.g., a company logo).

- `-bg <background_file_name>`

  Background description file; contains a list of images which are used as a background for randomly distorted versions of the object.

- `-num <number_of_samples>`

  Number of positive samples to generate.

- `-bgcolor <background_color>`

  Background color (currently grayscale images are assumed); the background color denotes the transparent color. Since there might be compression artifacts, the amount of color tolerance can be specified by `-bgthresh`. All pixels withing `bgcolor-bgthresh` and `bgcolor+bgthresh` range are interpreted as transparent.

- `-bgthresh <background_color_threshold>`
- `-inv`

  If specified, colors will be inverted.

- `-randinv`

  If specified, colors will be inverted randomly.

- `-maxidev <max_intensity_deviation>`

  Maximal intensity deviation of pixels in foreground samples.

- `-maxxangle <max_x_rotation_angle>`
- `-maxyangle <max_y_rotation_angle>`
- `-maxzangle <max_z_rotation_angle>`

  Maximum rotation angles must be given in radians.

- `-show`

  Useful debugging option. If specified, each sample will be shown. Pressing `Esc` will continue the samples creation process without.

- `-w <sample_width>`

  Width (in pixels) of the output samples.

- `-h <sample_height>`

  Height (in pixels) of the output samples.

- `-pngoutput`

  With this option switched on `opencv_createsamples` tool generates a collection of PNG samples and a number of associated annotation files, instead of a single `vec` file.

The `opencv_createsamples` utility may work in a number of modes, namely:

- **Creating training set from a single image and a collection of backgrounds:**
  - with a single `vec` file as an output;
  - with a collection of JPG images and a file with annotations list as an output;
  - with a collection of PNG images and associated files with annotations as an output;
- Converting the marked-up collection of samples into a `vec` format;

- Showing the content of the vec file.

Creating training set from a single image and a collection of backgrounds with a single vec file as an output

The following procedure is used to create a sample object instance: The source image is rotated randomly around all three axes. The chosen angle is limited my -max?angle. Then pixels having the intensity from [bg_color-bg_color_threshold; bg_color+bg_color_threshold] range are interpreted as transparent. White noise is added to the intensities of the foreground. If the -inv key is specified then foreground pixel intensities are inverted. If -randinv key is specified then algorithm randomly selects whether inversion should be applied to this sample. Finally, the obtained image is placed onto an arbitrary background from the background description file, resized to the desired size specified by -w and -h and stored to the vec-file, specified by the -vec command line option.

Creating training set as a collection of PNG images

To obtain such behaviour the -img, -bg, -info and -pngoutput keys should be specified. The file name specified with -info key should include at least one level of directory hierarchy, that directory will be used as the top-level directory for the training set. For example, with the opencv_createsamples called as following:

```
opencv_createsamples -img /home/user/logo.png -bg /home/user/bg.txt -info /home/user/annotations.lst -pngoutput
```

The output will have the following structure:

```
/home/user/
    annotations/
        0001_0107_0099_0195_0139.txt
        0002_0107_0115_0195_0139.txt
        ...
    neg/
        <background files here>
    pos/
        0001_0107_0099_0195_0139.png
        0002_0107_0115_0195_0139.png
        ...
    annotations.lst
```

With *.txt files in annotations directory containing information about object bounding box on the sample in a next format:

```
Image filename : "/home/user/pos/0002_0107_0115_0195_0139.png"
Bounding box for object 1 "PASperson" (Xmin, Ymin) - (Xmax, Ymax) : (107, 115) - (302, 254)
```

And annotations.lst file containing the list of all annotations file:

```
/home/user/annotations/0001_0109_0209_0195_0139.txt
/home/user/annotations/0002_0241_0245_0139_0100.txt
```

Creating test set as a collection of JPG images

This variant of opencv_createsamples usage is very similar to the previous one, but generates the output in a different format; To obtain such behaviour the -img, -bg and -info keys should be specified. For example, with the opencv_createsamples called as following:

```
opencv_createsamples -img /home/user/logo.png -bg /home/user/bg.txt -info annotations.lst -maxxangle 0.1 -maxyang
```

Directory structure:

```
info.dat
img1.jpg
img2.jpg
```

File info.dat:

```
img1.jpg  1  140 100 45 45
img2.jpg  2  100 200 50 50   50 30 25 25
```

Converting the marked-up collection of samples into a `vec` format

Positive samples also may be obtained from a collection of previously marked up images. This collection is described by a text file similar to background description file. Each line of this file corresponds to an image. The first element of the line is the filename. It is followed by the number of object instances. The following numbers are the coordinates of objects bounding rectangles (x, y, width, height).

An example of description file:

Directory structure:

```
/img
  img1.jpg
  img2.jpg
info.dat
```

File info.dat:

```
img/img1.jpg  1  140 100 45 45
img/img2.jpg  2  100 200 50 50   50 30 25 25
```

Image img1.jpg contains single object instance with the following coordinates of bounding rectangle: (140, 100, 45, 45). Image img2.jpg contains two object instances.

In order to create positive samples from such collection, `-info` argument should be specified instead of `-img`:

- `-info <collection_file_name>`

  Description file of marked up images collection.

The scheme of samples creation in this case is as follows. The object instances are taken from images. Then they are resized to target samples size and stored in output vec-file. No distortion is applied, so the only affecting arguments are `-w`, `-h`, `-show` and `-num`.

Showing the content of the `vec` file

`opencv_createsamples` utility may be used for examining samples stored in positive samples file. In order to do this only `-vec`, `-w` and `-h` parameters should be specified.

Note that for training, it does not matter how vec-files with positive samples are generated. But `opencv_createsamples` utility is the only one way to collect/create a vector file of positive samples, provided by OpenCV.

Example of vec-file is available here `opencv/data/vec_files/trainingfaces_24-24.vec`. It can be used to train a face detector with the following window size: `-w 24 -h 24`.

Cascade Training

The next step is the training of classifier. As mentioned above `opencv_traincascade` or `opencv_haartraining` may be used to train a cascade classifier, but only the newer `opencv_traincascade` will be described further.

Command line arguments of `opencv_traincascade` application grouped by purposes:

1. Common arguments:

   - `-data <cascade_dir_name>`

     Where the trained classifier should be stored.

   - `-vec <vec_file_name>`

     vec-file with positive samples (created by `opencv_createsamples` utility).

   - `-bg <background_file_name>`

     Background description file.

   - `-numPos <number_of_positive_samples>`

   - `-numNeg <number_of_negative_samples>`

     Number of positive/negative samples used in training for every classifier stage.

   - `-numStages <number_of_stages>`

     Number of cascade stages to be trained.

   - `-precalcValBufSize <precalculated_vals_buffer_size_in_Mb>`

     Size of buffer for precalculated feature values (in Mb).

   - `-precalcIdxBufSize <precalculated_idxs_buffer_size_in_Mb>`

     Size of buffer for precalculated feature indices (in Mb). The more memory you have the faster the training process.

   - `-baseFormatSave`

     This argument is actual in case of Haar-like features. If it is specified, the cascade will be saved in the old format.

   - `-acceptanceRatioBreakValue`

     This argument is used to determine how precise your model should keep learning and when to stop. A good guideline is to train not further than 10e-5, to ensure the model does not overtrain on your training data. By default this value is set to -1 to disable this feature.

2. Cascade parameters:

   - `-stageType <BOOST(default)>`

     Type of stages. Only boosted classifier are supported as a stage type at the moment.

   - `-featureType<{HAAR(default), LBP}>`

     Type of features: `HAAR` - Haar-like features, `LBP` - local binary patterns.

   - `-w <sampleWidth>`

   - `-h <sampleHeight>`

     Size of training samples (in pixels). Must have exactly the same values as used during training samples creation (`opencv_createsamples` utility).

3. Boosted classifer parameters:

   - -bt <{DAB, RAB, LB, GAB(default)}>

     Type of boosted classifiers: DAB - Discrete AdaBoost, RAB - Real AdaBoost, LB - LogitBoost, GAB - Gentle AdaBoost.

   - -minHitRate <min_hit_rate>

     Minimal desired hit rate for each stage of the classifier. Overall hit rate may be estimated as (min_hit_rate ^ number_of_stages) *[Viola2004]*.

   - -maxFalseAlarmRate <max_false_alarm_rate>

     Maximal desired false alarm rate for each stage of the classifier. Overall false alarm rate may be estimated as (max_false_alarm_rate ^ number_of_stages) *[Viola2004]*.

   - -weightTrimRate <weight_trim_rate>

     Specifies whether trimming should be used and its weight. A decent choice is 0.95.

   - -maxDepth <max_depth_of_weak_tree>

     Maximal depth of a weak tree. A decent choice is 1, that is case of stumps.

   - -maxWeakCount <max_weak_tree_count>

     Maximal count of weak trees for every cascade stage. The boosted classifier (stage) will have so many weak trees (<=maxWeakCount), as needed to achieve the given -maxFalseAlarmRate.

4. Haar-like feature parameters:

   - -mode <BASIC (default) | CORE | ALL>

     Selects the type of Haar features set used in training. BASIC use only upright features, while ALL uses the full set of upright and 45 degree rotated feature set. See *[Rainer2002]* for more details.

5. Local Binary Patterns parameters:

   Local Binary Patterns don't have parameters.

After the opencv_traincascade application has finished its work, the trained cascade will be saved in cascade.xml file in the folder, which was passed as -data parameter. Other files in this folder are created for the case of interrupted training, so you may delete them after completion of training.

Training is finished and you can test you cascade classifier!

CHAPTER

FIVE

SENZ3D AND INTEL PERCEPTUAL COMPUTING SDK

Using Creative Senz3D and other Intel Perceptual Computing SDK compatible depth sensors

Depth sensors compatible with Intel Perceptual Computing SDK are supported through `VideoCapture` class. Depth map, BGR image and some other formats of output can be retrieved by using familiar interface of `VideoCapture`.

In order to use depth sensor with OpenCV you should do the following preliminary steps:

1. Install Intel Perceptual Computing SDK (from here http://www.intel.com/software/perceptual).
2. Configure OpenCV with Intel Perceptual Computing SDK support by setting `WITH_INTELPERC` flag in CMake. If Intel Perceptual Computing SDK is found in install folders OpenCV will be built with Intel Perceptual Computing SDK library (see a status `INTELPERC` in CMake log). If CMake process doesn't find Intel Perceptual Computing SDK installation folder automatically, the user should change corresponding CMake variables `INTELPERC_LIB_DIR` and `INTELPERC_INCLUDE_DIR` to the proper value.
3. Build OpenCV.

VideoCapture can retrieve the following data:

1. **data given from depth generator:**
   - `CV_CAP_INTELPERC_DEPTH_MAP` - each pixel is a 16-bit integer. The value indicates the distance from an object to the camera's XY plane or the Cartesian depth. (CV_16UC1)
   - `CV_CAP_INTELPERC_UVDEPTH_MAP` - each pixel contains two 32-bit floating point values in the range of 0-1, representing the mapping of depth coordinates to the color coordinates. (CV_32FC2)
   - `CV_CAP_INTELPERC_IR_MAP` - each pixel is a 16-bit integer. The value indicates the intensity of the reflected laser beam. (CV_16UC1)
2. **data given from BGR image generator:**
   - `CV_CAP_INTELPERC_IMAGE` - color image. (CV_8UC3)

In order to get depth map from depth sensor use `VideoCapture::operator >>`, e. g.

```
VideoCapture capture( CV_CAP_INTELPERC );
for(;;)
{
    Mat depthMap;
    capture >> depthMap;

    if( waitKey( 30 ) >= 0 )
        break;
}
```

For getting several data maps use `VideoCapture::grab` and `VideoCapture::retrieve`, e.g.

```
VideoCapture capture(CV_CAP_INTELPERC);
for(;;)
{
    Mat depthMap;
    Mat image;
    Mat irImage;

    capture.grab();

    capture.retrieve( depthMap, CV_CAP_INTELPERC_DEPTH_MAP );
    capture.retrieve(    image, CV_CAP_INTELPERC_IMAGE );
    capture.retrieve(  irImage, CV_CAP_INTELPERC_IR_MAP);

    if( waitKey( 30 ) >= 0 )
        break;
}
```

For setting and getting some property of sensor‘ data generators use `VideoCapture::set` and `VideoCapture::get` methods respectively, e.g.

```
VideoCapture capture( CV_CAP_INTELPERC );
capture.set( CV_CAP_INTELPERC_DEPTH_GENERATOR | CV_CAP_PROP_INTELPERC_PROFILE_IDX, 0 );
cout << "FPS    " << capture.get( CV_CAP_INTELPERC_DEPTH_GENERATOR+CV_CAP_PROP_FPS ) << endl;
```

Since two types of sensor’s data generators are supported (image generator and depth generator), there are two flags that should be used to set/get property of the needed generator:

- CV_CAP_INTELPERC_IMAGE_GENERATOR – a flag for access to the image generator properties.
- CV_CAP_INTELPERC_DEPTH_GENERATOR – a flag for access to the depth generator properties. This flag value is assumed by default if neither of the two possible values of the property is set.

For more information please refer to the example of usage intelperc_capture.cpp in `opencv/samples/cpp` folder.

[Viola2001] Paul Viola, Michael J. Jones. *Rapid Object Detection using a Boosted Cascade of Simple Features*. Conference on Computer Vision and Pattern Recognition (CVPR), 2001, pp. 511-518.

[Viola2004] Paul Viola, Michael J. Jones. *Robust real-time face detection*. International Journal of Computer Vision, 57(2):137–154, 2004.

[Rainer2002] Rainer Lienhart and Jochen Maydt. *An Extended Set of Haar-like Features for Rapid Object Detection*. Submitted to ICIP2002.

[Liao2007] Shengcai Liao, Xiangxin Zhu, Zhen Lei, Lun Zhang and Stan Z. Li. *Learning Multi-scale Block Local Binary Patterns for Face Recognition*. International Conference on Biometrics (ICB), 2007, pp. 828-837.