

# Chapter 5. Generic Object Detection for Industrial Applications

This chapter will introduce you to the world of generic object detection, with a closer look at the advantages that industrial applications yield compared to the standard academic research cases. As many of you will know, OpenCV 3 contains the well-known **Viola and Jones algorithm** (embedded as the CascadeClassifier class), which was specifically designed for robust face detection. However, the same interface can efficiently be used to detect any desired object class that suits your needs.

## Note

More information on the Viola and Jones algorithm can be found in the following publication:

Rapid object detection using a boosted cascade of simple features, Viola P. and Jones M., (2001). In Computer Vision and Pattern Recognition, 2001 (CVPR 2001). Proceedings of the 2001 IEEE Computer Society Conference on (Vol. 1, pp. I-511). IEEE.

This chapter assumes that you have a basic knowledge of the cascade classification interface of OpenCV 3. If not, here are some great starting points for understanding this interface and the basic usage of the supplied parameters and software:

- [http://docs.opencv.org/master/modules/objdetect/doc/cascade\\_classification.html](http://docs.opencv.org/master/modules/objdetect/doc/cascade_classification.html)
- [http://docs.opencv.org/master/doc/tutorials/objdetect/cascade\\_classifier/cascade\\_class](http://docs.opencv.org/master/doc/tutorials/objdetect/cascade_classifier/cascade_class)
- [http://docs.opencv.org/master/doc/user\\_guide/ug\\_traincascade.html](http://docs.opencv.org/master/doc/user_guide/ug_traincascade.html)

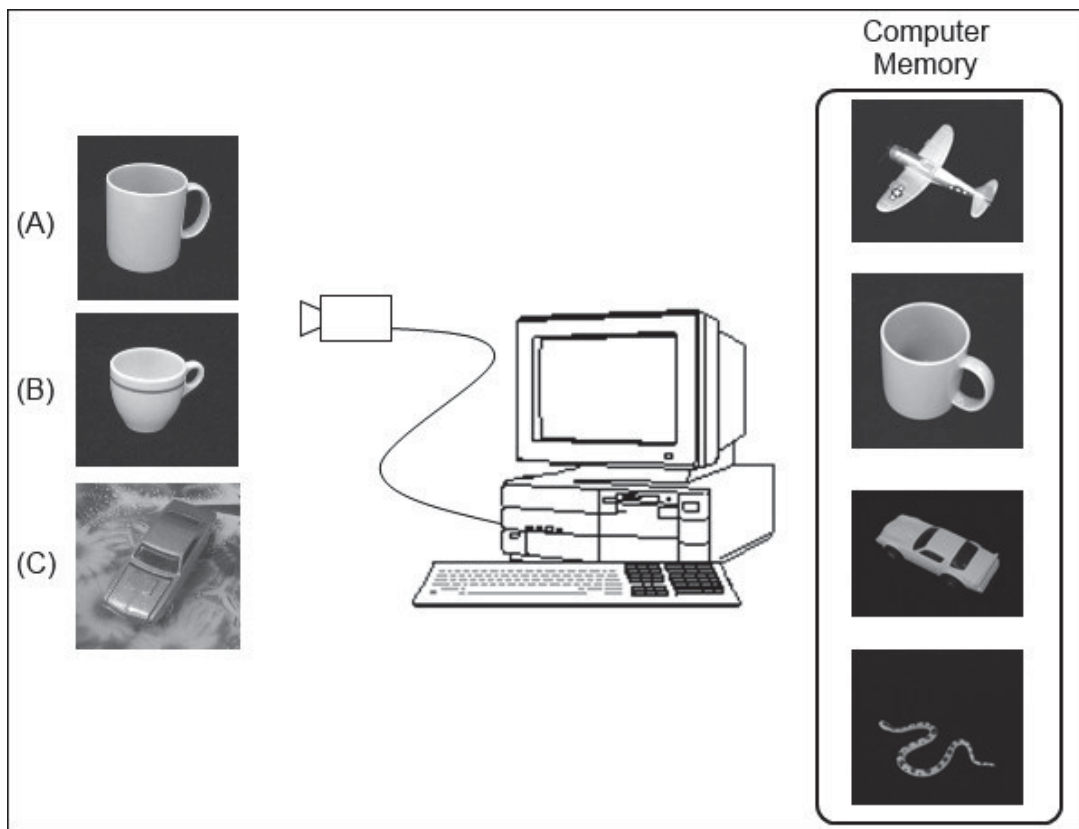
## Note

Or you can simply read one of the PacktPub books that discuss this topic in more detail such as [Chapter 3](#), *Training a Smart Alarm to Recognize the Villain and His Cat*, of the *OpenCV for Secret Agents* book by Joseph Howse.

In this chapter, I will take you on a tour through specific elements that are important when using the Viola and Jones face detection framework for generic object detection. You will learn how to adapt your training data to the specific situation of your setup, how to make your object detection model rotation invariant, and you will find guidelines on how to improve the accuracy of your detector by smartly using environment parameters and situational knowledge. We will dive deeper into the actual object class model and explain what happens, combined with some smart tools for visualizing the actual process of object detection. Finally, we will look at GPU possibilities, which will lead to faster processing times. All of this will be combined with code samples and example use cases of general object detection.

# Difference between recognition, detection, and categorization

For completely understanding this chapter, it is important that you understand that the Viola and Jones detection framework based on cascade classification is actually an object categorization technique and that it differs a lot from the concept of object recognition. This leads to a common mistake in computer vision projects, where people do not analyze the problem well enough beforehand and thus wrongfully decide to use this technique for their problems. Take into consideration the setup described in the following figure, which consists of a computer with a camera attached to it. The computer has an internal description of four objects (plane, cup, car, and snake). Now, we consider the case where three new images are supplied to the camera of the system.



*A simple computer vision setup*

In the case that image A is presented to the system, the system creates a description of the given input image and tries to match it with the descriptions of the images in the computer memory database. Since that specific cup is in a slightly rotated position, the descriptor of the cup's memory image will have a closer match than the other object images in memory and thus this system is able to successfully recognize the known cup. This process is called **object recognition**, and is applied in cases where we know exactly which object we want to find in our input image.

*“The goal of object recognition is to match (recognize) a specific object or scene. Examples include recognizing a specific building, such as the Pisa Tower, or a specific painting, such as the Mona Lisa. The object is recognized despite changes in scale, camera viewpoint, illumination conditions and partial occlusion.”*

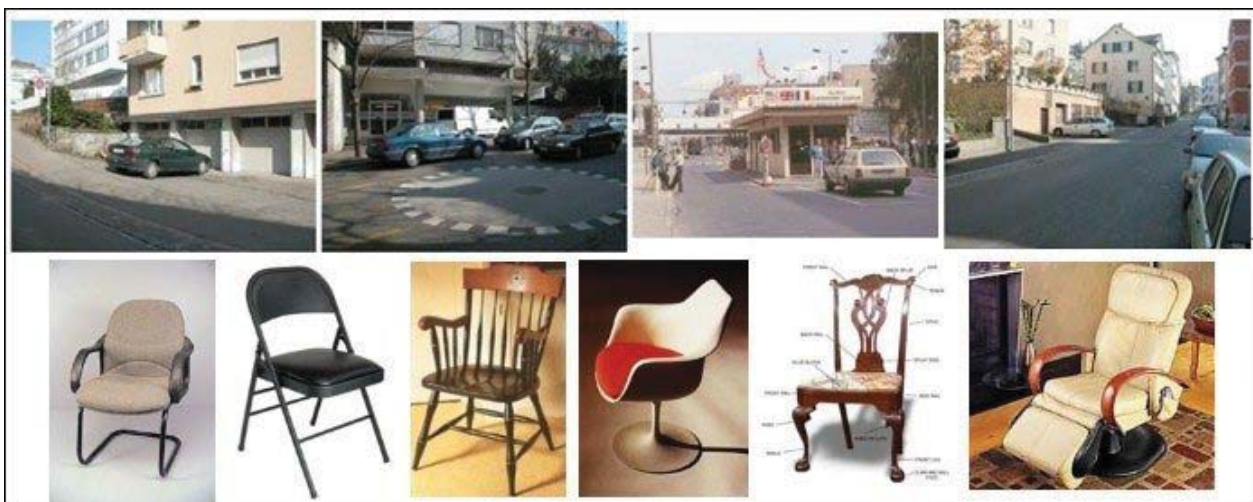
*—Andrea Vedaldi and Andrew Zisserman*

However, this technique has some downsides. If an object is presented to the system that doesn't have a description in the image database, the system will still return the closest match and thus the result could be very misleading. To avoid this we tend to put a threshold on the matching quality. If the threshold is not reached, we simply do not provide a match.

When image B is presented to the same system, we experience a new problem. The difference between the the given input image and the cup image in memory is so large (different size, different shape, different print, and so on) that the descriptor of image B will not be matched to the description of the cup in memory, again a large downside of object recognition. The problems even rise further, when image C is presented to the system. There, the known car from computer memory is presented to the camera system, but it is presented in a completely different setup and background than the one in memory. This could lead to the background influencing the object descriptor so much that the object is not recognized anymore.

**Object detection** goes a bit further; it tries to find a given object in varying setups by learning a more object specific description instead of just a description of the image itself. In a situation where the detectable object class becomes more complex, and the variation of an object is large over several input images—we are no longer talking about single object detection, but rather about detecting a class of objects—this is where **object categorization** comes into play.

With object categorization, we try to learn a generic model for the object class that can handle a lot of variation inside the object class, as shown in the following figure:



*An example of object classes with lots of variation: cars and chairs/sofas*

Inside such a single object class, we try to cope with different forms of variation, as seen

in the following figure:



*Variation within a single object class: illumination changes, object pose, clutter, occlusions, intra-class appearance, and viewpoint*

It is very important to make sure that your application actually is of the third and latter case if you plan to use the Viola and Jones object detection framework. In that case, the object instances you want to detect are not known beforehand and they have a large intra-class variance. Each object instance can have differences in shape, color, size, orientation, and so on. The Viola and Jones algorithm will model all that variance into a single object model that will be able to detect any given instance of the class, even if the object instance has never been seen before. And this is the large power of object categorization techniques, where they generalize well over a set of given object samples to learn specifics for the complete object class.

These techniques allow us to train object detectors for more complex classes and thus make object categorization techniques ideal to use in industrial applications such as object inspection, object picking, and so on, where typically used threshold-based segmentation techniques seem to fail due this large variation in the setup.

If your application does not handle objects in these difficult situations, then consider using other techniques such as object recognition if it suits your needs!

Before we start with the real work, let me take the time to introduce to you the basic steps that are common in object detection applications. It is important to pay equal attention to all the steps and definitely not to try and skip some of them for gaining time. These would all influence the end result of the object detector interface:



1. **Data collection:** This step includes collecting the necessary data for building and testing your object detector. The data can be acquired from a range of sources going from video sequences to images captured by a webcam. This step will also make sure that the data is formatted correctly to be ready to be passed to the training stage.
2. **The actual model training:** In this step, you will use the data gathered in the first step to train an object model that will be able to detect that model class. Here, we will investigate the different training parameters and focus on defining the correct settings for your application.
3. **Object detection:** Once you have a trained object model, you can use it to try and detect object instances in the given test images.
4. **Validation:** Finally, it is important to validate the detection result of the third step, by comparing each detection with a manually defined ground truth of the test data. Various options for efficiency and accuracy validation will be discussed.

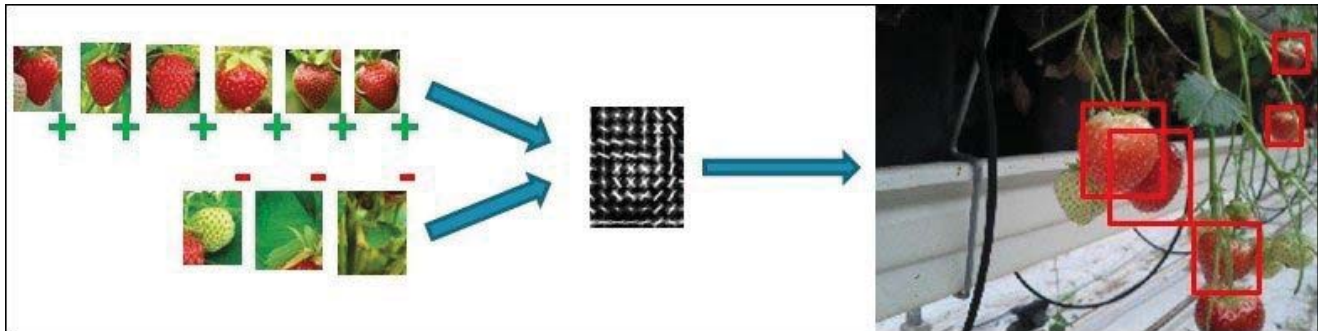
Let's continue by explaining the first step, the data collection in more detail, which is also the first subtopic of this chapter.



# Smartly selecting and preparing application specific training data

In this section, we will discuss how much training samples are needed according to the situational context and highlight some important aspects when preparing your annotations on the positive training samples.

Let's start by defining the principle of object categorization and its relation to training data, which can be seen in the following figure:



*An example of positive and negative training data for an object model*

The idea is that the algorithm takes a set of positive object instances, which contain the different presentations of the object you want to detect (this means object instances under different lighting conditions, different scales, different orientations, small shape changes, and so on) and a set of negative object instances, which contains everything that you do not want to detect with your model. Those are then smartly combined into an object model and used to detect new object instances in any given input image as seen in the figure above.

# The amount of training data

Many object detection algorithms depend heavily on large quantities of training data, or at least that is what is expected. This paradigm came to existence due to the academic research cases, mainly focusing on very challenging cases such as pedestrian and car detection. These are both object classes where a huge amount of intra-class variance exists, resulting in:

- A very large positive and negative training sample set, leading up to thousands and even millions of samples for each set.
- The removal of all information that pollutes the training set, rather than helping it, such as color information, and simply using feature information that is more robust to all this intra-class variation such as edge information and pixel intensity differences.

As a result, models were trained that successfully detect pedestrians and cars in about every possible situation, with the downside that training them required several weeks of processing.. However, when you look at more industrial specific cases, such as the picking of fruit from bins or the grabbing of objects from a conveyor belt, you can see that the amount of variance in objects and background is rather limited compared to these very challenging academic research cases. And this is a fact that we can use to our own advantage.

We know that the accuracy of the resulting object model is highly dependent on the training data used. In cases where your detector needs to work in all possible situations, supplying huge amounts of data seems reasonable. The complex learning algorithms will then decide which information is useful and which is not. However, in more confined cases, we could build object models by considering what our object model actually needs to do.

For example, the Facebook DeepFace application, used for detecting faces in every possible situation using the neural networks approach uses 4.4 million labeled faces.

## Note

More information on the DeepFace algorithm can be found in:

Deepface: Closing the gap to human-level performance in face verification, Taigman Y., Yang M., Ranzato M. A., and Wolf L. (2014, June). In Computer Vision and Pattern Recognition (CVPR), 2014, IEEE Conference on (pp. 1701-1708).

We therefore suggest using only meaningful positive and negative training samples for your object model by following a set of simple rules:

- For the positive samples, only use **natural occurring samples**. There are many tools out there that create artificially rotated, translated, and skewed images to turn a small training set into a large training set. However, research has proven that the resulting detector is less performant than simply collecting positive object samples that cover the actual situation of your application. Better use a small set of decent **high quality object samples**, rather than using a large set of low quality non-representative



samples for your case.

- For the negative samples, there are two possible approaches, but both start from the principle that you collect negative samples in the situation where your detector will be used, which is very different from the normal way of training object detectors, where just a large set of random samples not containing the object are being used as negatives.
  - Either point a camera at your scene and start grabbing random frames to sample negative windows from.
  - Or use your positive images to your advantage. Cut out the actual object regions and make the pixels black. Use those masked images as negative training data. Keep in mind that in this case the ratio between background information and actual object occurring in the window needs to be large enough. If your images are filled with object instances, cutting them will result in a complete loss of relevant background information and thus reduce the discriminative power of your negative training set.
- Try to use a very small set of negative windows. If in your case only 4 or 5 background situations can occur, then there is no need to use 100 negative images. Just take those five specific cases and sample negative windows from them.

Efficiently collecting data in this way ensures that you will end up with a very robust model for your specific application! However, keep in mind that this also has some consequences. The resulting model will not be robust towards different situations than the ones trained for. However, the benefit in training time and the reduced need of training samples completely outweighs this downside.

## Note

Software for negative sample generation based on OpenCV 3 can be found at [https://github.com/OpenCVBlueprints/OpenCVBlueprints/tree/master/chapter\\_5/source\\_cc](https://github.com/OpenCVBlueprints/OpenCVBlueprints/tree/master/chapter_5/source_cc)

You can use the negative sample generation software to generate samples like you can see in the following figure, where object annotations of strawberries are removed and replaced by black pixels.



*An example of the output of the negative image generation tool, where annotations are cut out and replaced by black pixels*

As you can see, the ratio between the object pixels and the background pixels is still large enough in order to ensure that the model will not train his background purely based on those black pixel regions. **Keep in mind that avoiding the approach of using these black pixelated images, by simply collecting negative images, is always better.** However, many companies forget this important part of data collection and just end up without a negative data set meaningful for the application. Several tests I performed proved that using a negative dataset from random frames from your application have a more discriminative negative power than black pixels cutout based images.

# Creating object annotation files for the positive samples

When preparing your positive data samples, it is important to put some time in your annotations, which are the actual locations of your object instances inside the larger images. Without decent annotations, you will never be able to create decent object detectors. There are many tools out there for annotation, but I have made one for you based on OpenCV 3, which allows you to quickly loop over images and put annotations on top of them.

## Note

Software for object annotation based on OpenCV 3 can be found at [https://github.com/OpenCVBlueprints/OpenCVBlueprints/tree/master/chapter\\_5/source\\_cc](https://github.com/OpenCVBlueprints/OpenCVBlueprints/tree/master/chapter_5/source_cc)

The OpenCV team was kind enough to also integrate this tool into the main repository under the apps section. This means that if you build and install the OpenCV apps during installation, that the tool is also accessible by using the following command:

```
/opencv_annotation -images <folder location> -annotations <output file>
```

Using the software is quite straightforward:

1. Start by running the CMAKE script inside the GitHub folder of the specific project. After running CMAKE, the software will be accessible through an executable. The same approach applies for every piece of software in this chapter. Running the CMAKE interface is quite straightforward:

```
cmakemake  
./object_annotation -images <folder location> -annotations <output  
file>
```

2. This will result in an executable that needs some input parameters, being the location of the positive image files and the output detection file.

## Note

Keep in mind to always assign the absolute path of all files!

3. First, parse the content of your positive image folder to a file (by using the supplied `folder_listing` software inside the object annotation folder), and then follow this by executing the annotation command:

```
./folder_listing -folder <folder> -images <images.txt>
```

4. The folder listing tool should generate a file, which looks exactly like this:

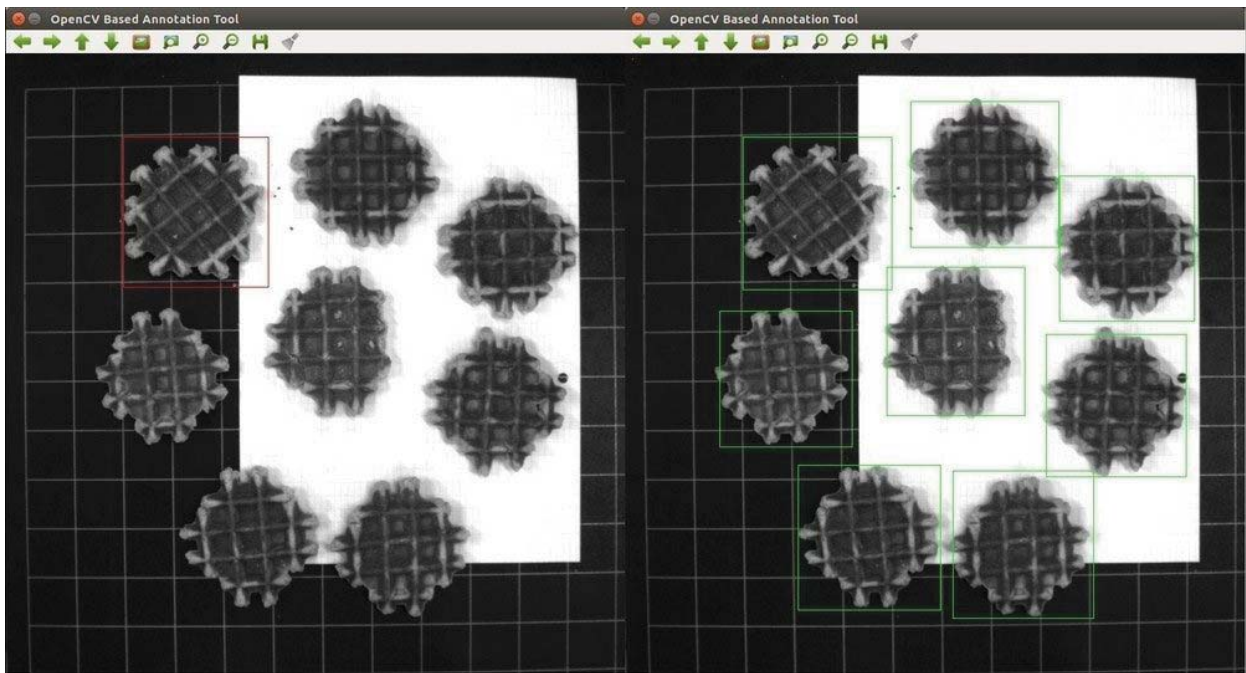
```
positives.txt x
1 /home/usr/data/image1.png
2 /home/usr/data/image2.png
3 ...
4 /home/usr/data/imageN.png
```

*A sample positive samples file generated by the folder listing tool*

5. Now, fire up the annotation tool with the following command:

```
./object_annotation -images <images.txt> -annotations <annotations.txt>
```

6. This will fire up the software and give you the first image in a window, ready to apply annotations, as shown in the following figure:



*A sample of the object annotation tool*

7. You can start by selecting the top-left corner of the object, then moving the mouse until you reach the bottom right corner of the object, which can be seen in the left part of the preceding figure. However, the software allows you to start your annotation from each possible corner. If you are unhappy with the selection, then reapply this step, until the annotation suits your needs.
8. Once you agree on the selected bounding box, press the button that confirms a selection, which is key *C* by default. This will confirm the annotation, change its color from red to green, and add it to the annotations file. Be sure only to accept an annotation if you are 100% sure of the selection.
9. Repeat the preceding two steps for the same image until you have annotated every single object instance in the image, as seen in the right part of the preceding example



image. Then press the button that saves the result and loads in the following image, which is the *N* key by default.

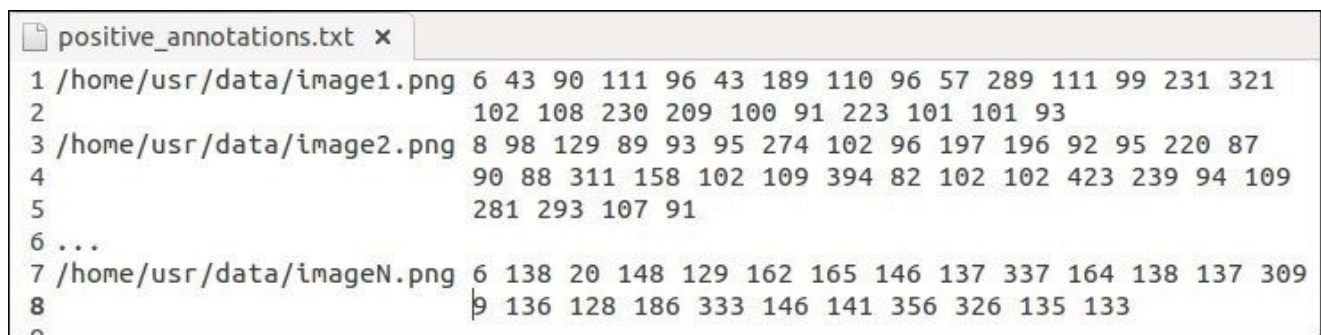
10. Finally, you will end up with a file called `annotations.txt`, which combines the location of the image files together with the ground truth locations of all object instances that occur inside the training images.

## Note

If you want to adapt the buttons that need to be pressed for all the separate actions, then open up the `object_annotation.cpp` file and browse to line 100 and line 103. There you can adapt the ASCII values assigned to the button you want to use for the operation.

An overview of all ASCII codes assigned to your keyboard keys can be found at <http://www.asciitable.com/>.

The output from the software is a list of object detections in a `*.txt` file for each folder of positive image samples, which has a specific structure as seen in the following figure:



```
positive_annotations.txt x
1 /home/usr/data/image1.png 6 43 90 111 96 43 189 110 96 57 289 111 99 231 321
2 102 108 230 209 100 91 223 101 101 93
3 /home/usr/data/image2.png 8 98 129 89 93 95 274 102 96 197 196 92 95 220 87
4 90 88 311 158 102 109 394 82 102 102 423 239 94 109
5 281 293 107 91
6 ...
7 /home/usr/data/imageN.png 6 138 20 148 129 162 165 146 137 337 164 138 137 309
8 9 136 128 186 333 146 141 356 326 135 133
9
```

### *An example of an object annotation tool*

It starts with the absolute file location of each image in the folder. There was a choice of not using relative paths since the file will then be fully dependent on the location where it is stored. However, if you know what you are doing, then using relative file locations in relation to the executable should work just fine. Using the absolute path makes it more universal and more failsafe. The file location is followed by the number of detections for that specific image, which allows us to know beforehand how many ground truth objects we can expect. For each of the objects, the (x, y) coordinates are stored to the top-left corner combined with the width and the height of the bounding box. This is continued for each image, which is each time a new line appears in the detection output file.

## Tip

It is important for further model training that each set of ground truth values captured from other annotation systems is first converted to this format in order to ensure the decent working of the cascade classifier software embedded in OpenCV 3.

A second point of attention when processing positive training images containing object instances, is that you need to pay attention to the way you perform the actual placement of



the bounding box of an object instance. A good and accurately annotated ground truth set will always give you a more reliable object model and will yield better test and accuracy results. Therefore, I suggest using the following points of attention when performing object annotation for your application:

- Make sure that the bounding box contains the complete object, but at the same time avoid as much background information as possible. The ratio of object information compared to background information should always be larger than 80%. Otherwise, the background could yield enough features to train your model on and the end result will be your detector model focusing on the wrong image information.
- Viola and Jones suggests using squared annotations, based on a 24x24 pixel model, because it fits the shape of a face. However, this is not mandatory! If your object class is more rectangular like, then do annotate rectangular bounding boxes instead of squares. It is observed that people tend to push rectangular shaped objects in a square model size, and then wonder why it is not working correctly. Take, for example, the case of a pencil detector, where the model dimensions will be more like 10x70 pixels, which is in relation to the actual pencil dimensions.
- Try doing concise batches of images. It is better to restart the application 10 times, than to have a system crash when you are about to finish a set of 1,000 images with corresponding annotations. If somehow, the software or your computer fails it ensures that you only need to redo a small set.

# Parsing your positive dataset into the OpenCV data vector

Before the OpenCV 3 software allows you to train a cascade classifier object model, you will need to push your data into an OpenCV specific data vector format. This can be done by using the provided sample creation tool of OpenCV.

## Note

The sample creation tool can be found at <https://github.com/Itseez/opencv/tree/master/apps/createsamples/> and should be built automatically if OpenCV was installed correctly, which makes it usable through the `opencv_createsamples` command.

Creating the sample vector is quite easy and straightforward by applying the following instruction from the command line interface:

```
./opencv_createsamples -info annotations.txt -vec images.vec -bg negatives.txt -num amountSamples -w model_width -h model_height
```

This seems quite straightforward, but it is very important to make no errors in this step of the setup and that you carefully select all parameters if you want a model that will actually be able to detect something. Let's discuss the parameters and instruct where to focus on:

- **-info:** Add here the annotation file that was created using the object annotation software. Make sure that the format is correct, that there is no empty line at the bottom of the file and that the coordinates fall inside the complete image region. This annotation file should only contain positive image samples and **no negative image samples** as some online tutorials suggest. This would train your model to recognize negative samples as positives, which is not what we desire.
- **-vec:** This is the data format OpenCV will use to store all the image information and is the file that you created using the create samples software provided by OpenCV itself.
- **-num:** This is the actual number of annotations that you have inside the vector file over all the images presented to the algorithm. If you have no idea anymore how many objects you have actually annotated, then run the annotation counter software supplied.

## Note

The sample counting tool can be found at [https://github.com/OpenCVBlueprints/OpenCVBlueprints/tree/master/chapter\\_5/source](https://github.com/OpenCVBlueprints/OpenCVBlueprints/tree/master/chapter_5/source) and can be executed by the following command:

```
./count_samples -file <annotations.txt>
```

- **-w and -h:** These are the two parameters that specify the final model dimensions. Keep in mind that these dimensions will immediately define the smallest object that you will be able to detect. Keep the size of the actual model therefore smaller than

the smallest object you want to detect in your test images. Take, for example, the Viola and Jones face detector, which was trained on samples of 24x24 pixels, and will never be able to detect faces of 20x20 pixels.

When looking to the OpenCV 3 documentation on the “create samples” tool, you will see a wide range of extra options. These are used to apply artificial rotation, translation, and skew to the object samples in order to create a large training dataset from a limited set of training samples. This only works well when applying it to objects on a clean single color background, which can be marked and passed as the transparency color. Therefore, we suggest not to use these parameters in real-world applications and provide enough training data yourself using all the rules defined in the previous section.

## Tip

If you have created a working classifier with, for example, 24x24 pixel dimensions and you still want to detect smaller objects, then a solution could be to upscale your images before applying the detector. However, keep in mind that if your actual object is, for example, 10x10 pixels, then upscaling that much will introduce tons of artifacts, which will render your model detection capabilities useless.

A last point of attention is how you can decide which is an effective model size for your purpose. On the one hand, you do not want it to be too large so that you can detect small object instances, on the other hand, you want enough pixel information so that separable features can be found.

## Note

You can find a small code snippet that can help you out with defining these ideal model dimensions based on the average annotation dimensions at the following location:

[https://github.com/OpenCVBlueprints/OpenCVBlueprints/tree/master/chapter\\_5/source\\_cc](https://github.com/OpenCVBlueprints/OpenCVBlueprints/tree/master/chapter_5/source_cc)

Basically, what this software does is it takes an annotation file and processes the dimensions of all your object annotations. It then returns an average width and height of your object instances. You then need to apply a scaling factor to assign the dimensions of the smallest detectable object.

For example:

- Take a set of annotated apples in an orchard. You got a set of annotated apple tree images of which the annotations are stored in the apple annotation file.
- Pass the apple annotation file to the software snippet which returns you the average apple width and apple height. For now, we suppose that the dimensions for  $[w_{average} \ h_{average}]$  are  $[60 \ 60]$ .
- If we would use those  $[60 \ 60]$  dimensions, then we would have a model that can only detect apples equal and larger to that size. However, moving away from the tree will result in not a single apple being detected anymore, since the apples will become smaller in size.
- Therefore, I suggest reducing the dimensions of the model to, for example,  $[30 \ 30]$ .

This will result in a model that still has enough pixel information to be robust enough and it will be able to detect up to half the apples of the training apples size.

- Generally speaking, the rule of thumb can be to take half the size of the average dimensions of the annotated data and ensure that your largest dimension is not bigger than 100 pixels. This last guideline is to ensure that training your model will not increase exponentially in time due to the large model size. If your largest dimension is still over 100 pixels, then just keep halving the dimensions until you go below this threshold.

You have now prepared your positive training set. The last thing you should do is create a folder with the negative images, from which you will sample the negative windows randomly, and apply the folder listing functionality to it. This will result in a negative data referral file that will be used by the training interface.





# Parameter selection when training an object model

Once you have built a decent training samples dataset, which is ready to process, the time has arrived to fire up the cascade classifier training software of OpenCV 3, which uses the Viola and Jones cascade classifier framework to train your object detection model. The training itself is based on applying the boosting algorithm on either Haar wavelet features or Local Binary Pattern features. Several types of boosting are supported by the OpenCV interface, but for convenience, we use the frequently used AdaBoost interface.

## Note

If you are interested in knowing all the technical details of the feature calculation, then have a look at the following papers which describe them in detail:

- **HAAR:** Papageorgiou, Oren and Poggio, “A general framework for object detection”, International Conference on Computer Vision, 1998.
- **LBP:** T. Ojala, M. Pietikäinen, and D. Harwood (1994), “Performance evaluation of texture measures with classification based on Kullback discrimination of distributions”, Proceedings of the 12th IAPR International Conference on Pattern Recognition (ICPR 1994), vol. 1, pp. 582 - 585.

This section will discuss several parts of the training process in more detail. It will first elaborate on how OpenCV runs its cascade classification process. Then, we will take a deeper look at all the training parameters provided and how they can influence the training process and accuracy of the resulting model. Finally, we will open up the model file and look in more detail at what we can find there.

# Training parameters involved in training an object model

It is important to pay attention when carefully selecting your training parameters. In this subsection, we will discuss the relevance of some of the training parameters used when training and suggest some settings for general testing purposes. The following subsection will then discuss the output and the quality of the resulting classifier.

First, start by downloading and compiling the cascade classifier training application which is needed for generating an object detection model using the Viola and Jones framework.

## Note

The cascade classification training tool can be found at <https://github.com/Itseez/opencv/tree/master/apps/traincascade/>. If you build the OpenCV apps and installed them, then it will be directly accessible by executing the `./train_cascade` command.

If you run the application without the parameters given, then you will get a print of all the arguments that the application can take. We will not discuss every single one of them, but focus on the most delicate ones that yield the most problems when training object models using the cascade classifier approach. We will give some guidelines as how to select correct values and where to watch your steps. You can see the output parameters from the cascade classifier training interface in the following figure:

```
Usage: opencv_traincascade
  -data <cascade_dir_name>
  -vec <vec_file_name>
  -bg <background_file_name>
  [-numPos <number_of_positive_samples = 2000>]
  [-numNeg <number_of_negative_samples = 1000>]
  [-numStages <number_of_stages = 20>]
  [-precalcValBufSize <precalculated_vals_buffer_size_in_Mb = 256>]
  [-precalcIdxBufSize <precalculated_idx_buffer_size_in_Mb = 256>]
  [-baseFormatSave]
--cascadeParams--
  [-stageType <BOOST(default)>]
  [-featureType <{HAAR(default), LBP, HOG}>]
  [-w <sampleWidth = 24>]
  [-h <sampleHeight = 24>]
--boostParams--
  [-bt <{DAB, RAB, LB, GAB(default)}>]
  [-minHitRate <min_hit_rate> = 0.995]
  [-maxFalseAlarmRate <max_false_alarm_rate = 0.5>]
  [-weightTrimRate <weight_trim_rate = 0.95>]
  [-maxDepth <max_depth_of_weak_tree = 1>]
  [-maxWeakCount <max_weak_tree_count = 100>]
--haarFeatureParams--
  [-mode <BASIC(default) | CORE | ALL]
--lbpFeatureParams--
--HOGFeatureParams--
```

- -data: This parameter contains the folder where you will output your training results. Since the creation of folders is OS specific, OpenCV decided that they will let users handle the creation of the folder. If you do not make it in advance, training results will not be stored correctly. The folder will contain a set of XML files, one for the training parameters, one for each trained stage and finally a combined XML file containing the object model.
- -numPos: This is the amount of positive samples that will be used in training each stage of weak classifiers. Keep in mind that this number is not equal to the total amount of positive samples. The classifier training process (discussed in the next subtopic) is able to reject positive samples that are wrongfully classified by a certain stage limiting further stages to use that positive sample. A good guideline in selecting this parameter is to multiply the actual amount of positive samples, retrieved by the sample counter snippet, with a factor of 0.85.
- -numNeg: This is the amount of negative samples used at each stage. However, this is not the same as the amount of negative images that were supplied by the negative data. The training samples negative windows from these images in a sequential order at the model size dimensions. Choosing the right amount of negatives is highly dependent on your application.
  - If your application has close to no variation, then supplying a small number of windows could simply do the trick because they will contain most of the background variance.
  - On the other hand, if the background variation is large, a huge number of samples would be needed to ensure that you train as much random background noise as possible into your model.
  - A good start is taking a ratio between the number of positive and the number of negative samples equaling 0.5, so double the amount of negative versus positive windows.
  - Keep in mind that each negative window that is classified correctly at an early stage will be discarded for training in the next stage since it cannot add any extra value to the training process. Therefore, you must be sure that enough unique windows can be grabbed from the negative images. For example, if a model uses 500 negatives at each stage and 100% of those negatives get correctly classified at each stage, then training a model of 20 stages will need 10,000 unique negative samples! Considering that the sequential grabbing of samples does not ensure uniqueness, due to the limited pixel wise movement, this amount can grow drastically.
- -numStages: This is the amount of weak classifier stages, which is highly dependent on the complexity of the application.
  - The more stages, the longer the training process will take since it becomes harder at each stage to find enough training windows and to find features that

correctly separate the data. Moreover, the training time increases in an exponential manner when adding stages.

- Therefore, I suggest looking at the reported acceptance ratio that is outputted at each training stage. Once this reaches values of  $10^{-5}$ , you can conclude that your model will have reached the best descriptive and generalizing power it could get, according to the training data provided.
- Avoid training it to levels of  $10^{-5}$  or lower to avoid overtraining your cascade on your training data. Of course, depending on the amount of training data supplied, the amount of stages to reach this level can differ a lot.
- -bg: This refers to the location of the text file that contains the locations of the negative training images, also called the negative samples description file.
- -vec: This refers to the location of the training data vector that was generated in the previous step using the create\_samples application, which is built-in to the OpenCV 3 software.
- -precalcValBufSize and -precalcIdxBufSize: These parameters assign the amount of memory used to calculate all features and the corresponding weak classifiers from the training data. If you have enough RAM memory available, increase these values to 2048 MB or 4096 MB, which will speed up the precalculation time for the features drastically.
- -featureType: Here, you can choose which kind of features are used for creating the weak classifiers.
  - HAAR wavelets are reported to give higher accuracy models.
  - However, consider training test classifiers with the LBP parameter. It decreases training time of an equal sized model drastically due to the integer calculations instead of the floating point calculations.
- -minHitRate: This is the threshold that defines how much of your positive samples can be misclassified as negatives at each stage. The default value is 0.995, which is already quite high. The training algorithm will select its stage threshold so that this value can be reached.
  - Making it 0.999, as many people do, is simply impossible and will make your training stop probably after the first stage. It means that only 1 out of 1,000 samples can be wrongly classified over a complete stage.
  - If you have very challenging data, then lowering this, for example, to 0.990 could be a good start to ensure that the training actually ends up with a useful model.
- -maxFalseAlarmRate: This is the threshold that defines how much of your negative samples need to be classified as negatives before the boosting process should stop adding weak classifiers to the current stage. The default value is 0.5 and ensures that a stage of weak classifier will only do slightly better than random guessing on the negative samples. Increasing this value too much could lead to a single stage that already filters out most of your given windows, resulting in a very slow model at detection time due to the vast amount of features that need to be validated for each

window. This will simply remove the large advantage of the concept of early window rejection.

The parameters discussed earlier are the most important ones to dig into when trying to train a successful classifier. Once this works, you can increase the performance of your classifier even more, by looking at the way boosting forms its weak classifiers. This can be adapted by the `-maxDepth` and `-maxWeakCount` parameters. However, for most cases, using **stump weak classifiers** (single layer decision trees) on single features is the best way to start, ensuring that single stage evaluation is not too complex and thus fast at detection time.



# The cascade classification process in detail

Once you select the correct training parameters, you can start the cascade classifier training process, which will build your cascade classifier object detection model. In order to fully understand the cascade classification process that builds up your object model, it is important to know how OpenCV does its training of the object model, based on the boosting process.

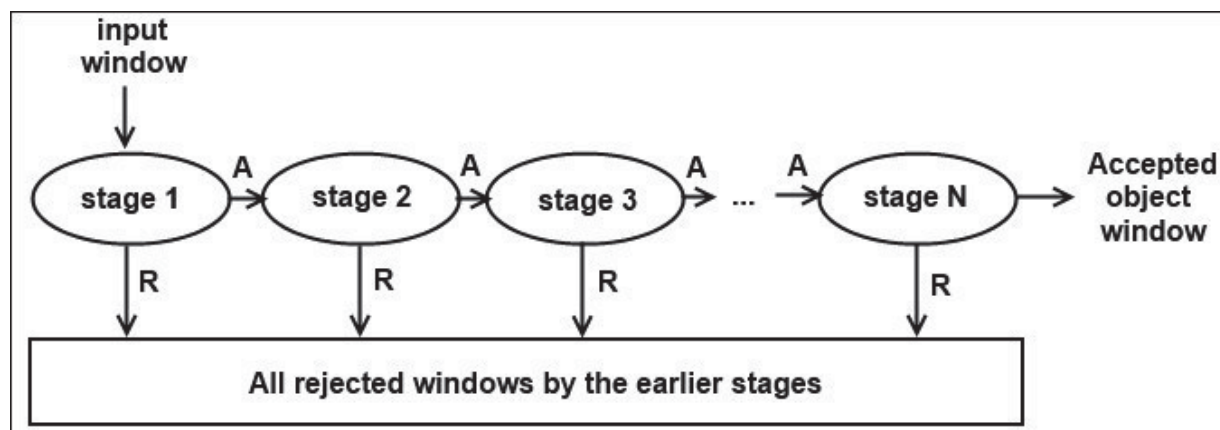
Before we do this, we will have a quick look at the outline of the boosting principle in general.

## Note

More information on the boosting principle can be found in Freund Y., Schapire R., and Abe N (1999). A short introduction to boosting. Journal-Japanese Society For Artificial Intelligence, 14(771-780), 1612

The idea behind boosting is that you have a very large pool of features that can be shaped into classifiers. Using all those features for a single classifier would mean that every single window in your test image will need to be processed for all these features, which will take a very long time and make your detection slow, especially if you consider how many negative windows are available in a test image. To avoid this, and to reject as many negative windows as fast as possible, boosting selects the features that are best at separating the positive and negative data and combines them into classifiers, until the classifier does a bit better than random guessing on the negative samples. This first step is called a weak classifier. Boosting repeats this process until the combination of all these weak classifiers reach the desired accuracy of the algorithm. The combination is called the strong classifier. The main advantage of this process is that tons of negative samples will already be discarded by the few early stages, with only evaluating a small set of features, thus decreasing detection time a lot.

We will now try to explain the complete process using the output generated by the cascade training software embedded in OpenCV 3. The following figure illustrates how a strong cascade classifier is built from a set of stages of weak classifiers.



*A combination of weak classifier stages and early rejection of misclassified windows*

*resulting in the famous cascade structure*

The cascade classifier training process follows an iterative process to train subsequent stages of weak classifiers (1...N). Each stage consists of a set of weak classifiers, until the criteria for that specific stage have been reached. The following steps are an overview of what is happening at training each stage in OpenCV 3, according to the input parameters given and the training data provided. If you are interested in more specific details of each subsequent step, then do read the research paper of Viola and Jones (you can have a look at the citation on the first page of this chapter) on cascade classifiers. All steps described here are subsequently repeated for each stage until the desired accuracy for the strong classifier is reached. The following figure shows how such a stage output looks like:

```
===== TRAINING 0-stage =====
<BEGIN
POS count : consumed    100 : 100
NEG count : acceptanceRatio    1000 : 1
Precalculation time: 1
+-----+-----+-----+
|  N  |      HR      |      FA      |
+-----+-----+-----+
|  1  |          1  |          1  |
+-----+-----+-----+
|  2  |          1  |          1  |
+-----+-----+-----+
|  3  |          1  |    0.221  |
+-----+-----+-----+
END>
Training until now has taken 0 days 0 hours 14 minutes 46 seconds.

===== TRAINING 1-stage =====
<BEGIN
POS count : consumed    100 : 100
NEG count : acceptanceRatio    1000 : 0.28393
Precalculation time: 2
+-----+-----+-----+
|  N  |      HR      |      FA      |
+-----+-----+-----+
|  1  |          1  |          1  |
+-----+-----+-----+
|  2  |          1  |          1  |
+-----+-----+-----+
|  3  |          1  |          1  |
+-----+-----+-----+
|  4  |          1  |    0.388  |
+-----+-----+-----+
END>
Training until now has taken 0 days 0 hours 34 minutes 19 seconds.
```

*An example output of a classifier stage training*

## Step 1 – grabbing positive and negative samples

You will notice that the first thing the training does is grabbing training samples for the current stage—first the positive samples from the data vector you supplied, and then the random negative window samples from the negative images that you supplied. This will be outputted for both steps as:

**POS: number\_pos\_samples\_grabbed: total\_number\_pos\_samples\_needed**  
**NEG: number\_neg\_samples\_grabbed: acceptanceRatioAchieved**

If no positive samples can be found anymore, an error will be generated and training will be stopped. The total number of samples needed will increase once you start discarding positives that are no longer useful. The grabbing of the negatives for the current stage can take much longer than the positive sample grabbing since all windows that are correctly classified by the previous stages are discarded and new ones are searched. The deeper you go into the amount of stages, the harder this gets. As long as the number of samples grabbed keeps increasing (and yes, this can be very slow, so be patient), your application is still running. If no more negatives are found, the application will end training and you will need to lower the amount of negatives for each stage or add extra negative images.

The acceptance ratio that is achieved by the previous stage is reported after the grabbing of the negative windows. This value indicates whether the model trained until now is strong enough for your detection purposes or not!

## **Step 2 – precalculation of integral image and all possible features from the training data**

Once we have both positive and negative window-sized samples, the precalculation will calculate every single feature that is possible within the window size and apply it for each training sample. This can take some time according to the size of your model and according to the amount of training samples, especially when knowing that a model of 24x24 pixels can yield more than 16,000 features. As suggested earlier, assigning more memory can help out here or you could decide on selecting LBP features, of which the calculation is rather fast compared to HAAR features.

All features are calculated on the integral image representation of the original input window. This is done in order to speed up the calculation of the features. The paper by Viola and Jones explains in detail why this integral image representation is used.

The features calculated are dumped into a large feature pool from which the boosting process can select the features needed to train the weak classifiers that will be used within each stage.

## **Step 3 – firing up the boosting process**

Now, the cascade classifier training is ready for the actual boosting process. This happens in several small steps:

- Every possible weak classifier inside the feature pool is being calculated. Since we use stumps, which are basically weak classifiers based on single feature to create a decision tree, there are as many weak classifiers as features. If you prefer, you can decide to train actual decision trees with a predefined maximum depth, but this goes

outside of the scope of this chapter.

- Each weak classifier is trained in order to minimize the misclassification rate on the training samples. For example, when using Real AdaBoost as a boosting technique, the Gini index is minimized.

## Note

More information on the Gini index, used for the misclassification rate on the training samples can be found in:

Gastwirth, J. L. (1972). The estimation of the Lorenz curve and Gini index. The Review of Economics and Statistics, 306-316.

- The weak classifier with the lowest misclassification rate is added as the next weak classifier to the current stage.
- Based on the weak classifiers that are already added to the stage, the algorithm calculates the overall stage threshold, which is set so that the desired hit rate is guaranteed.
- Now, the weights of the samples are adapted based on their classification in the last iteration, which will yield a new set of weak classifiers in the next iteration, and thus the whole process can start again.
- During the combination of the weak classifiers inside a single stage, which is visualized in the training output, the boosting process makes sure that:
  - The overall stage threshold does not drop below the minimum hit rate that was selected by the training parameters.
  - The false alarm rate on the negative samples decreases compared to the previous stage.
- This process continues until:
  - The false acceptance ratio on the negative samples is lower than the maximum false alarm rate set. The process then simply starts training a new stage of weak classifiers for the detection model.
  - The required stage false alarm rate is reached, which is  $\text{maxFalseAlarmRate}^{\text{\#stages}}$ . This will yield an end to the model training since the model satisfies our requirements and better results cannot be achieved anymore. This will not happen often, since this value drops rather quickly, and after several stages, this would mean that you correctly classify more than 99% of your positive and negative samples all together.
  - The hit rate drops below the stage specific minimal hit rate, which is the  $\text{minHitRate}^{\text{\#stages}}$ . At this stage, too many positives get wrongly classified and the maximum performance for your model is reached.

## Step 4 – saving the temporary result to a stage file

After training each stage, the stage specific details about the weak classifiers and the thresholds are stored in the data folder, in a separate XML file. If the desired number of stages has been reached, then these subfiles are combined into a single cascade XML file.

However, the fact that every stage is stored separately means that you can stop the training at any time and create an in-between object detection model, by simply restarting the training command, but changing the `-numStages` parameter to the stage value on which you want to check the model's performance. This is ideal when you want to perform an evaluation on a validation set to ensure that your model does not start overfitting on the training data!



# The resulting object model explained in detail

It has been observed that many users of the cascade classifier algorithm embedded in OpenCV 3 do not know the meaning of the inner construction of the object model which is stored in the XML files, which sometimes leads to wrong perceptions of the algorithm. This subsection will explain each internal part of the trained object models. We will discuss a model based on stump-typed weak classifiers, but the idea is practically the same for any other type of weak classifiers inside a stage, such as decision trees. The biggest difference is that the weight calculation inside the model gets more complex as compared to when using stump features. As to the weak classifiers structure inside each stage, this will be discussed for both HAAR- and LBP-based features since these are the two most used features inside OpenCV for training cascade classifiers.

## Note

The two models that will be used for explaining everything can be found at

- [OpenCVsource/data/haarcascades/haarcascade\\_frontalface\\_default.xml](http://opencvsource/data/haarcascades/haarcascade_frontalface_default.xml)
- [OpenCVsource/data/lbpcascades/lbpcascade\\_frontalface.xml](http://opencvsource/data/lbpcascades/lbpcascade_frontalface.xml)

The first part of each XML stored model describes the parameters that specify the characteristics of the model itself and some of the important training parameters. Subsequently, we can find the type of training that is used, which is limited to boosting for now, and the type of features used for building the weak classifiers. We also have the width and height of the object model that will be trained, the parameters of the boosting process, which include the type of boosting used, the selected minimum hit ratio, and the selected maximum false acceptance rate. It also contains information about how the weak classifier stages are built, in our case as a combination of one feature deep trees, called stumps, with a maximum of 100 weak classifiers on a single stage. For the HAAR wavelet based model, we can then see which features are used, being only the basic upright features or the combined rotated 45-degree set.

After the training-specific parameters, it starts to get interesting. Here, we find more information about the actual structure of the cascade classifier object model. The amount of stages is described, and then iteratively the model sums up the training results and thresholds for each separate stage which were generated by the boosting process. The basic structure of an object model can be seen here:

```
<stages>
  <_>
    <maxWeakCount></maxWeakCount>
    <stageThreshold></stageThreshold>
    <weakClassifiers>
      <!-- tree 0 -->
      <_>
        <internalNodes></internalNodes>
        <leafValues></leafValues></_>
      <!-- tree 1 -->
      <_>
        <internalNodes></internalNodes>
```

```

        <leafValues></leafValues></_>
    <!-- tree 2 -->
    ... ..
    <!-- stage 1 -->
    ... ..
</stages>
<features>
    ... ..
</features>

```

We start with an empty iteration tag for each stage. At each stage the number of weak classifiers that were used are defined, which in our case shows how many single layer decision trees (stumps) were used inside the stage. The stage threshold defines the threshold on the final stage score for a window. This is generated by scoring the window with each weak classifier and then summing and weighing the results for the complete stage. For each single weak classifier, we collect the internal structure, based on the decision nodes and layers used. The values present are the boosting values used for creating the decision tree and the leaf values, which are used to score a window that is evaluated by the weak classifier.

The specifics for the internal node structure are different for HAAR wavelets and -based features. The storage of the leaf scores is equal. The values of the internal nodes, however, specify the relation to the bottom part of the code, which contains the actual features area, and which are also different for both the HAAR and the LBP approach. The difference between both techniques can be seen in the following sections, grabbing for both models the first tree of the first stage and a part of the feature set.

## HAAR-like wavelet feature models

The following are two code snippets from the HAAR wavelet feature-based model, containing the internal node structure and the features structure:

```

<internalNodes>
0 -1 445 -1.4772760681807995e-02
</internalNodes>
... ..
<_>
    <rects>
        <_>23 10 1 3 -1.</_>
        <_>23 11 1 1 3.</_>
    </rects>
    <tilted>0</tilted>
</_>

```

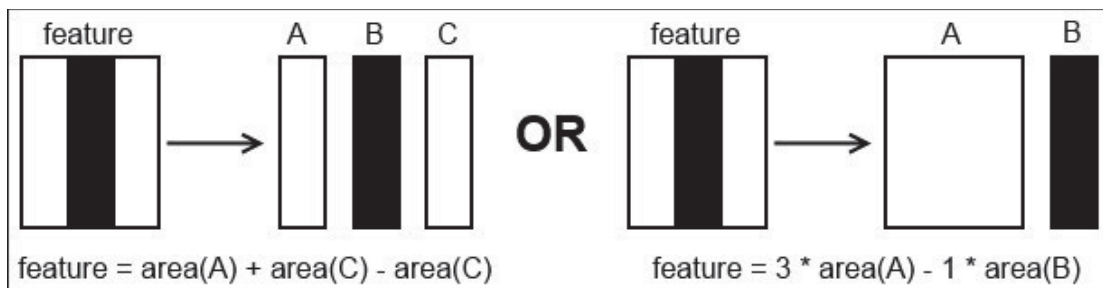
For the internal nodes, there are four values present at each node:

- **Node left and node right:** These values indicate that we have a stump with two leafs.
- **The node feature index:** This points the index of the feature used at this node according to its position inside the features list of that model.
- **The node threshold:** This is the threshold that is set on the feature value for this weak classifier, which is learned from all the positive and negative samples in this

stage of training. Since we are looking at models with stump based weak classifiers, this is also the stage threshold, which is set in the boosting process.

The features inside the HAAR-based model are described by a set of rectangles, which can be up to three rectangles, so as to calculate every possible feature from a window. Then, there is a value indicating if the feature itself is tilted over 45 degrees or not. For each rectangle, which is a partial feature value, we have:

- The location of the rectangle, which is defined by upper-left corner x and y coordinates and the width and height of the rectangle.
- The weight for that specific partial feature. These weights are used to combine both partial feature rectangles into a predefined feature. These weights allow us to represent each feature with less rectangles than is actually necessary. An example of this can be seen in the following figure:



*A three rectangle feature can be represented by a two rectangle weighted combination reducing the need of an extra area calculation*

The feature sum is finally calculated by first summing all values of the pixels inside the rectangle and then multiplying it with the weight factor. Finally, those weighted sums are combined together to yield as a final feature value. Keep in mind that all the coordinates retrieved for a single feature are in relation to the window/model size and not the complete image which is processed.

## Local binary pattern models

The following are two code snippets from the LBP feature-based model, containing the internal node structure and the features structure:

```
<internalNodes>
0 -1 46 -67130709 -21569 -1426120013 -1275125205 -21585
-16385 587145899 -24005
</internalNodes>
... ..
<_>
    <rect>0 0 3 5</rect>
</_>
```

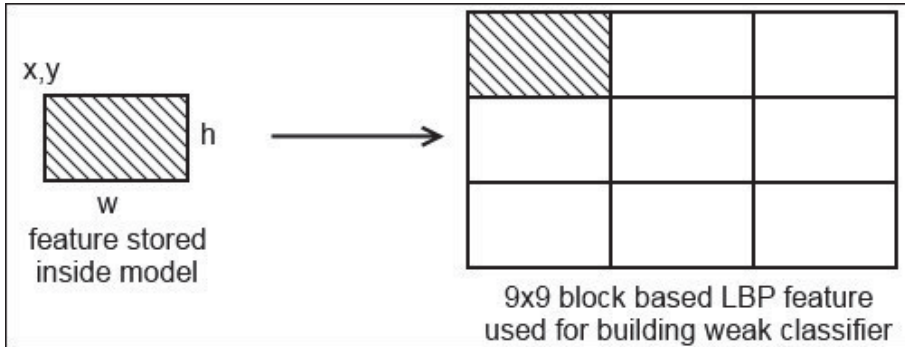
For the internal nodes, there are 11 values present at each node:

- **Node left and node right:** These values indicate that we have a stump with two leaves, which is identical to the HAAR-based model. If a more complex tree structure

is used, these values will expand.

- **The node feature index:** This points to the index of the feature used at this node according to its position inside the features list of that model.
- **Eight 32-bit values:** These values together in the combined form of a 256-bit LUT are calculated by comparing all subrectangle regions to the center subrectangle, as illustrated in the following figure, and which is used as a threshold to yield 1 or 0 as an outcome for the descriptor of the feature.

For the features inside the LBP based model, we have the dimensions (x, y, w, h) of a single subrectangle region (the top-left subrectangle) out of the nine that are needed for the LBP feature to be evaluated, as seen here:



*An example of a LBP feature-based on the single stored rectangle*

## Visualization tool for object models

However, there is nothing better than seeing your trained object model visualized on the object you have been training for. Therefore, I programmed a code snippet that takes a model and a base image and visualizes the complete model detection process on top of it.

### Note

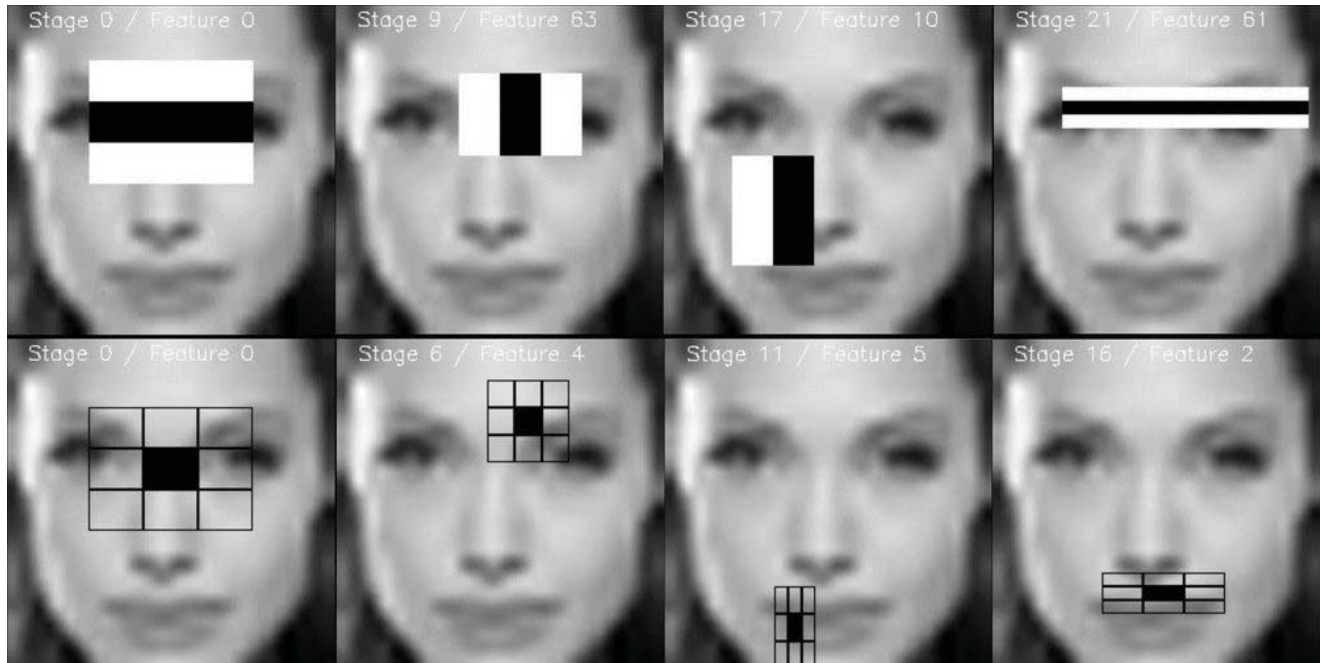
The software for visualizing Haar wavelet or LBP models can be found at [https://github.com/OpenCVBlueprints/OpenCVBlueprints/tree/master/chapter\\_5/source\\_cc](https://github.com/OpenCVBlueprints/OpenCVBlueprints/tree/master/chapter_5/source_cc)

The software takes in several input arguments, such as the model location, the image where the visualization needs to happen, and the output folder where the results need to be stored. However, in order to use the software correctly, there are some points of attention:

- The model needs to be HAAR wavelet or LBP feature based. Deleted because this functionality is no longer supported in OpenCV 3.
- You need to supply an image that is an actual model detection for visualization purposes and resize it to the model scale or a positive training sample from the training data. This is to ensure that a feature of your model is placed at the correct location.
- Inside the code, you can adapt the visualization scales, one being for the video output of your model and one for the images that represent the stages.

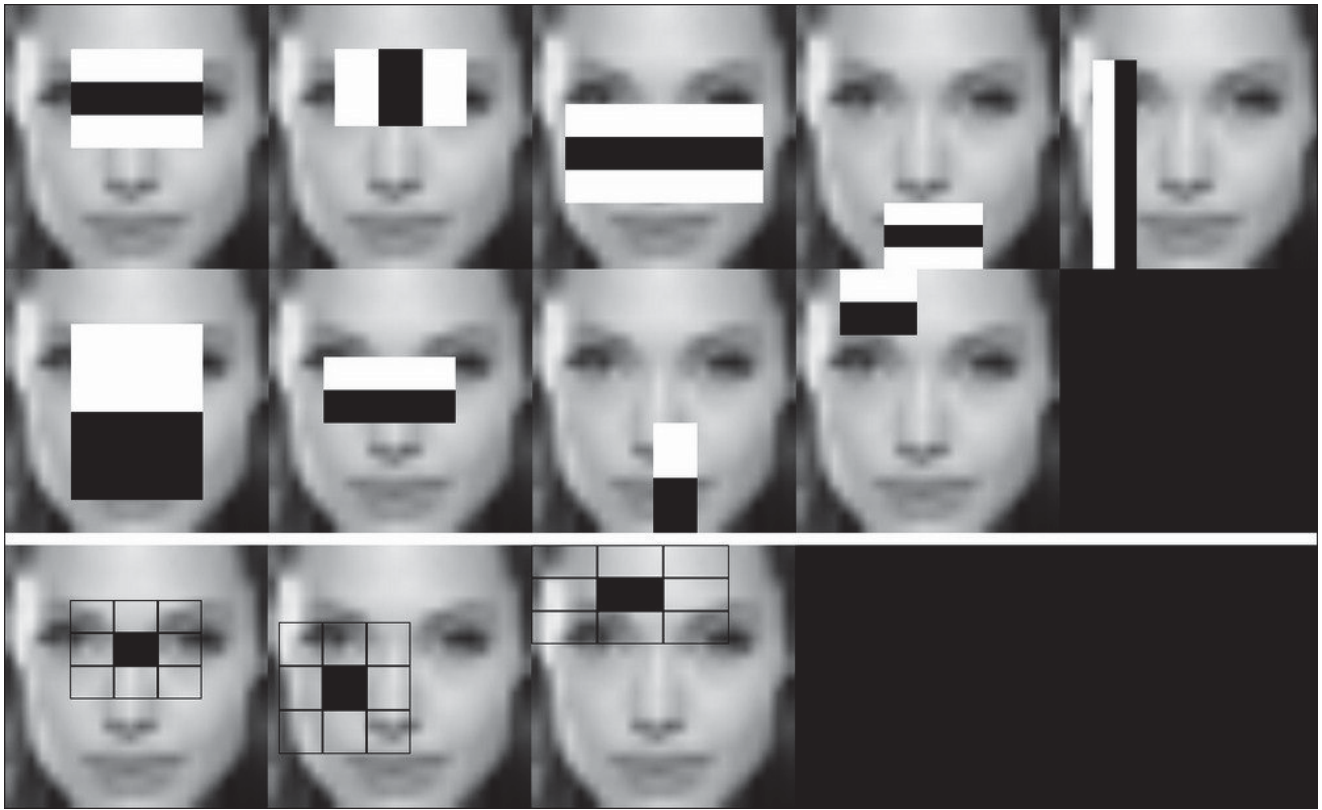
The following two figures illustrate the visualization result of the Haar wavelet and LBP

feature based frontal face model respectively, both incorporated into the OpenCV 3 repository under the data folder. The reason for the low image resolution of the visualization is quite obvious. The training process happens on a model scale; therefore, I wanted to start from an image of that size to illustrate that specific details of an object get removed, while general specifics of the object class still occur to be able to differentiate classes.



*A set of frames from the video visualization of the frontal face model for both Haar wavelet and Local Binary Pattern features*

The visualizations for example also clearly show that an LBP model needs less features and thus less weak classifiers to separate the training data successfully, which yields a faster model at detection time.



*A visualization of the first stage of the frontal face model for both Haar wavelet and Local Binary Pattern features*



# Using cross-validation to achieve the best model possible

Making sure that you get the absolute best model given your training, testing the data can be done by applying a cross validation approach, such as the leave-one-out approach. The idea behind this is that you combine both training and test set and vary the test set that you use from the larger set. With each random test set and training set, you build a separate model and you perform the evaluation using precision-recall, which is discussed further in this chapter. Finally, the model that provides the best result could be adopted as a final solution. Thus, it could mitigate the impact of an error due to a new instance that is not represented in the training set.

## Note

More information on the topic of cross validation can be found in Kohavi R. (1995, August), a study of cross-validation and bootstrap for accuracy estimation and model selection in *Ijcai* (Vol. 14, No. 2, pp. 1137-1145).



# Using scene specific knowledge and constraints to optimize the detection result

Once your cascade classifier object model is trained, you can use it to detect instances of the same object class in new input images, which are supplied to the system. However, once you apply your object model, you will notice that there are still false positive detections and objects that are not found. This section will cover techniques to improve your detection results, by removing, for example, most of the false positive detections with scene-specific knowledge.

# Using the parameters of the detection command to influence your detection result

If you apply an object model to a given input image, you must consider several things. Let's first take a look at the detection function and some of the parameters that can be used to filter out your detection output. OpenCV 3 supplies three possible interfaces. We will discuss the benefits of using each one of them.

## Interface 1:

```
void CascadeClassifier::detectMultiScale(InputArray image, vector<Rect>&
objects, double scaleFactor=1.1, int minNeighbors=3, int flags=0, Size
minSize=Size(), Size maxSize=Size())
```

The first interface is the most basic one. It allows you to fast evaluate your trained model on a given test image. There are several elements on this basic interface that will allow you to manipulate the detection output. We will discuss these parameters in some more detail and highlight some points of attention when selecting the correct value.

`scaleFactor` is the scale step used to downscale the original image in order to create the image pyramid, which allows us to perform multiscale detections using only a single scale model. One downside is that this doesn't allow you to detect objects that are smaller than the object size. Using a value of 1.1 means that in each step the dimensions are reduced by 10% compared to the previous step.

- Increasing this value will make your detector run faster since it has less scale levels to evaluate, but it will yield the risk of losing detections that are in between scale steps.
- Decreasing the value will make your detector run slower, since more scale levels need to be evaluated, but it will increase the chance of detecting objects that were missed before. Also, it will yield more detections on an actual object, resulting in a higher certainty.
- Keep in mind that adding scale levels also gives rise to more false positive detections, since those are bound to each layer of the image pyramid.

A second interesting parameter to adapt for your needs is the `minNeighbors` parameter. It describes how many overlapping detections occur due to the sliding window approach. Each detection overlapping by more than 50% with another will be merged together as a sort of nonmaxima suppression.

- Putting this value on 0 means that you will get all detections generated by the windows that get through the complete cascade. However, due to the sliding window approach (with steps of 8 pixels) many detections will happen for a single window, due to the nature of cascade classifiers, which train in some variance on object parameters in order to better generalize over an object class.
- Adding a value means that you want to count how many windows there should be, at least those combined by the nonmaxima suppression in order to keep the detection. This is interesting since an actual object should yield far more detections than a false

positive. So, increasing this value will reduce the number of false positive detections (which have a low amount of overlapping detections) and keep the true detections (which have a large amount of overlapping detections).

- A downside is that on a certain point, actual objects with a lower certainty of detections and thus less overlapping windows will disappear while some false positive detections might still stand.

Use the `minSize` and `maxSize` parameters to effectively reduce the scale space pyramid. In an industrial setup with, for example, a fixed camera position, such as a conveyor belt setup, you can in most cases guarantee that objects will have certain dimensions. Adding scale values in this case and thus defining a scale range will decrease processing time for a single image a lot, by removing undesired scale levels. As an extra advantage, all false positive detections on those undesired scales will also disappear. If you leave these values blank, the algorithm will start building the image pyramid at input image dimensions, in a bottom-up manner, downscale in steps equaling the scale percentage, until one of the dimensions is smaller than the largest object dimension. This will be the top of the image pyramid, which is also the place where later, at the detection time, the detection algorithm will start running its object detector.

### Interface 2:

```
void CascadeClassifier::detectMultiScale(InputArray image, vector<Rect>&
objects, vector<int>& numDetections, double scaleFactor=1.1, int
minNeighbors=3, int flags=0, Size minSize=Size(), Size maxSize=Size())
```

The second interface brings a small addition, by adding the `numDetections` parameter. This allows you to put the `minNeighbors` value on 1, applying the merging of overlapping windows as nonmaxima suppression, but at the same time returning you a value of the overlapping windows, which were merged. This value can be seen as a certainty score of your detection. The higher the value, the better or the more certain the detection.

### Interface 3:

```
void CascadeClassifier::detectMultiScale(InputArray image,
std::vector<Rect>& objects, std::vector<int>& rejectLevels,
std::vector<double>& levelWeights, double scaleFactor=1.1, int
minNeighbors=3, int flags=0, Size minSize=Size(), Size maxSize=Size(), bool
outputRejectLevels=false )
```

A downside of this interface is that 100 windows with a very small certainty of detection on an individual basis can simply out rule a single detection with a very high individual certainty of detection. This is where the third interface can bring us the solution. It allows us to look at the individual scores of each detection window (described by the threshold value of the last stage of the classifier). You can then grab all those values and threshold the certainty score of those individual windows. When applying nonmaxima suppression in this case, the threshold values of all overlapping windows are combined.

### Tip

Keep in mind that if you want to try out the third interface in OpenCV 3.0, you have to put

the parameter `outputRejectLevels` on `true`. If you do not do this, then the level weights matrix, which has the threshold scores, will not be filled.

## Note

Software illustrating the two most used interfaces for object detection can be found at [https://github.com/OpenCVBlueprints/OpenCVBlueprints/tree/master/chapter\\_5/source\\_cc](https://github.com/OpenCVBlueprints/OpenCVBlueprints/tree/master/chapter_5/source_cc) and [https://github.com/OpenCVBlueprints/OpenCVBlueprints/tree/master/chapter\\_5/source\\_cc](https://github.com/OpenCVBlueprints/OpenCVBlueprints/tree/master/chapter_5/source_cc). OpenCV detection interfaces change frequently and that it is possible that new interfaces are already available which are not discussed here.



# Increasing object instance detection and reducing false positive detections

Once you have chosen the most appropriate way of retrieving the object detections for your application, you can evaluate the proper output of your algorithm. Two of the most common problems found after training an object detector are:

- Object instances that are not being detected.
- Too much false positive detections.

The reason for the first problem can be explained by looking at the generic object model that we trained for the object class based on positive training samples of that object class. This lets us conclude that the training either:

- Did not contain enough positive training samples, making it impossible to generalize well over new object samples. In this case, it is important to add those false negative detections as positive samples to the training set and retrain your model with the extra data. This principle is called “reinforced learning”.
- We overtrained our model to the training set, again reducing the generalization of the model. To avoid this, reduce the model in stages and thus in complexity.

The second problem is quite normal and happens more than often. It is impossible to supply enough negative samples and at the same time ensure that there will not be a single negative window that could still yield a positive detection at a first run. This is mainly due to the fact that it is very hard for us humans to understand how the computer sees an object based on features. On the other hand, it is impossible to grasp every possible scenario (lighting conditions, interactions during the production process, filth on the camera, and so on) at the very start when training an object detector. You should see the creation of a good and stable model as an iterative process.

## Note

An approach to avoid the influence of lighting conditions can be to triplicate the training set by generating artificial dark and artificial bright images for each sample. However, keep in mind the disadvantages of artificial data as discussed in the beginning of this chapter.

In order to reduce the amount of false positive detections, we generally need to add more negative samples. However, it is important not to add randomly generated negative windows, since the extra knowledge that they would bring to the model would, in most cases, simply be minimal. It is better to add meaningful negative windows that can increase the quality of the detector. This is known as **hard negative mining** using a **bootstrapping** process. The principle is rather simple:

1. Start by training a first object model based on your initial training set of positive and negative window samples.
2. Now, collect a set of negative images, which are either specific to your application (if you want to train an object detector specific to your setup) or which are more general

(if you want your object detector to work in versatile conditions).

3. Run your detector on that set of negative images, with a low certainty threshold and save all found detections. Cut them out of the supplied negative images and rescale them towards the object model size dimensions.
4. Now, retrain your object model, but add all the found windows to your negative training set in order to ensure that your model will now be trained with this extra knowledge.

This will ensure that the accuracy of your model goes up by a fair and decent amount depending on the quality of your negative images.

## Tip

When adding the found extra and useful negative samples, add them to the top of your `background.txt` file! This forces the OpenCV training interface to first grab these more important negative samples before sampling all the standard negative training images provided! Be sure that they have exactly the required model size so that they can only be used once as a negative training sample.



# Obtaining rotation invariance object detection

A large downside to the current OpenCV cascade classifier implementation is that it only supports multiscale single rotation object detection. Many industrial applications that could actually use object detection do not know the orientation of the object beforehand and thus rotation invariant multiscale object detection would be much more interesting. Therefore, I will guide you through some techniques for applying multiscale rotation invariant object detection, by simply using the provided functionality in OpenCV.

## Note

OpenCV 3 also provides other techniques that are able to perform multiscale rotation invariant object categorization like the Bag of Visual Words approach. A good tutorial on this technique can be found at <https://gilscvblog.wordpress.com/2013/08/23/bag-of-words-models-for-visual-categorization/>.

There are three main ideas when trying to achieve rotation invariant object detection:

- Train a single object model with all possible orientations of the object as training data.
- Train multiple detectors for multiple orientations and then combine the outputs of all detectors together into a single output for a frame.
- Use a single rotation detector on a rotated object space and then warp back each found detection to the original frame.

The first approach is rather risky, especially if you do not know exactly what your algorithm will do with the provided training data. If you have an object that is of nature rotation invariant on its contours, such as a circle or ball-shaped object, then this could be a good approach. It will simply reject or ignore orientation specific information on its inner structure since not all samples will have the same feature value ranges inside the object, and will in turn focus only on object edge information. The figure *Example of the object annotation tool* illustrates a case of detecting cookies, of which the outer contour is 360 degrees rotation invariant, and thus, in this case, a single model can be used for all possible orientations.

However, this approach has several downsides:

- If the contour doesn't have enough object specific details, then it could trigger a lot of false positive detections on background structures with the same contour properties.
- If you want to differ between two similar object classes, then this will not work if the outer contours are rather similar since many of the other object specific features will simply be lost.
- If your object is not rotation invariant on its contour, then your features will have to be too generic to trigger a high hit rate on all your positive samples, and thus your model will generalize too much. This will result in a huge amount of false positive detections and a model that detects a lot of items that are actually no object instances.

The second approach is a bit more interesting. Research has proven that a single cascade classifier can handle rotations from about -10 to +10 degrees towards the original orientation of the model; however, it does vary a lot over different object classes. This enables us to basically cover the whole 360 degrees that an object can be in, with the use of about 18 models. This approach would request you to warp all training samples before training each specific orientation assigned model. Again, this technique has several downsides:

- You have to find optimal training parameters for each model. It is not guaranteed that the training parameters applied for the standard orientation will yield the same results if applied for a rotation of 45 degrees.
- You need to train 18 models! Depending on the complexity of your objects and the amount of training data this can take a very long time, especially with HAAR wavelet features. This is the time that you cannot spend on optimizing a single orientation model.
- If you rotate an image, then you will have to fill up blank space around the image in order not to lose image information. However, this creates artificial borders, which will not occur in your application. If your model trains on those regions, the result will be very poor. This is one of the major reasons why I suggest most people should also discard this second approach.

In the third approach, we will simply create a 3D rotational representation of our input image over the different orientations. I will supply you with a code snippet that can do this for you and discuss it step by step.

## Tip

Software for performing rotation invariant object detection based on the described third approach can be found at

[https://github.com/OpenCVBlueprints/OpenCVBlueprints/tree/master/chapter\\_5/source\\_cc](https://github.com/OpenCVBlueprints/OpenCVBlueprints/tree/master/chapter_5/source_cc)

The biggest advantage of this approach is that you only need to train a single orientation model and can put your time in updating and tweaking that single model in order to make it as efficient as possible. Another advantage is that you can combine all the detections in different rotations, by providing some overlap, and then increase the certainty of a detection, by smartly removing false positives that do not get detections over multiple orientations. So basically, it is kind of a trade-off between benefits and downsides of the approach.

However, there are still some downsides to this approach:

- You will need to apply a multiscale detector to each layer of your 3D representation matrix. This will definitely increase the search time for object instances compared to single orientation object detection.
- You will create false positive detections on each orientation, which will also be warped back, thus increasing the total number of false positive detections.

Let's take a deeper look at parts of the source code used for performing this rotation invariance and explain what is actually happening. The first interesting part can be found

in the creation of the 3D matrix of rotated images:

```
// Create the 3D model matrix of the input image
Mat image = imread(input_image);
int steps = max_angle / step_angle;
vector<Mat> rotated_images;
cvtColor(rotated, rotated, COLOR_BGR2GRAY);
equalizeHist( rotated, rotated );
for (int i = 0; i < steps; i++){
    // Rotate the image
    Mat rotated = image.clone();
    rotate(image, (i+1)*step_angle, rotated);
    // Preprocess the images

    // Add to the collection of rotated and processed images
    rotated_images.push_back(rotated);
}
```

Basically, what we do is read the original image, create a vector of Mat objects that can contain each rotated input image, and apply the rotation function on top of it. As you will notice, we immediately apply all preprocessing which is needed for efficient object detection using the cascade classifier interface such as rendering the image to grayscale values and applying a histogram equalization in order to cope a bit with illumination changes.

The rotate function can be seen here:

```
void rotate(Mat& src, double angle, Mat& dst)
{
    Point2f pt(src.cols/2., src.rows/2.);
    Mat r = getRotationMatrix2D(pt, angle, 1.0);
    warpAffine(src, dst, r, cv::Size(src.cols, src.rows));
}
```

This code first calculates a rotation matrix based on the angle, which is expressed in degrees, that we want to be rotated and then applies an affine transformation based on this rotation matrix. Keep in mind that rotating an image like this can lead to an information loss of objects at the borders. This code example assumes your objects will occur at the center of the image and thus this does not influence the result. You can avoid this by enlarging the original image by adding black borders to that. The width and height of the image are equal so that the image information loss is minimal. This can be done by adding the following code right behind the reading of the original input image:

```
Size dimensions = image.size();
if(dimensions.rows > dimensions.cols){
    Mat temp = Mat::ones(dimensions.rows, dimensions.rows, image.type()) *
255;
    int extra_rows = dimensions.rows - dimensions.cols;
    image.copyTo(temp(0, extra_rows/2, image.rows, image.cols));
    image = temp.clone();
}
if(dimensions.cols > dimensions.rows){
    Mat temp = Mat::ones(dimensions.cols, dimensions.cols, image.type()) *
255;
```



```

int extra_cols = dimensions.cols - dimensions.rows;
image.copyTo(temp(extra_cols/2, 0, image.rows, image.cols));
image = temp.clone();
}

```

This code will simply expand the original image to match a square region depending on the largest dimension.

Finally, on each level of the 3D image representation, a detection is performed and the found detections are warped back to the original image using a similar approach as warping the original image:

1. Take the four corner points of the found detection in the rotated image and add them into a matrix for rotation warping (code line 95-103).
2. Apply the inverse transformation matrix based on the angle of the current rotated image (code line 106-108).
3. Finally, draw a rotated rectangle on the information of the rotated four matrix points (code line 111-128).

The following figure shows the exact result of applying a rotation invariant face detection to an image with faces in multiple orientations.



*Rotation invariant face detection starting with the following angle steps [1 degree, 10 degrees, 25 degrees, 45 degrees]*

We see that four times the suggested technique is applied to the same input image. We played around with the parameters in order to see the influence on detection time and the detections returned. In all cases, we applied a search from 0 to 360 degrees, but changed the angle step in between each stage of the 3D rotation matrix from 0 to 45 degrees.

Applied angle step	Total time for executing all detections
1 degree	220 seconds
10 degrees	22.5 seconds
25 degrees	8.6 seconds
45 degrees	5.1 seconds

As we can see, the detection time is reduced drastically when increasing the step of the angle. Knowing that an object model on itself could cover at least 20 degrees in total, we can easily reduce the step in order to significantly decrease the processing time.



## 2D scale space relation

Another large advantage that we can exploit in industrial cases is the fact that many of these setups have a fixed camera position. This is interesting when the objects that need to be detected follow a fixed ground plane, like in the case of pedestrians or objects passing by on a conveyor belt. If these conditions exist, then there is actually a possibility to model the scale of an object at each position in the image. This yields two possible advantages:

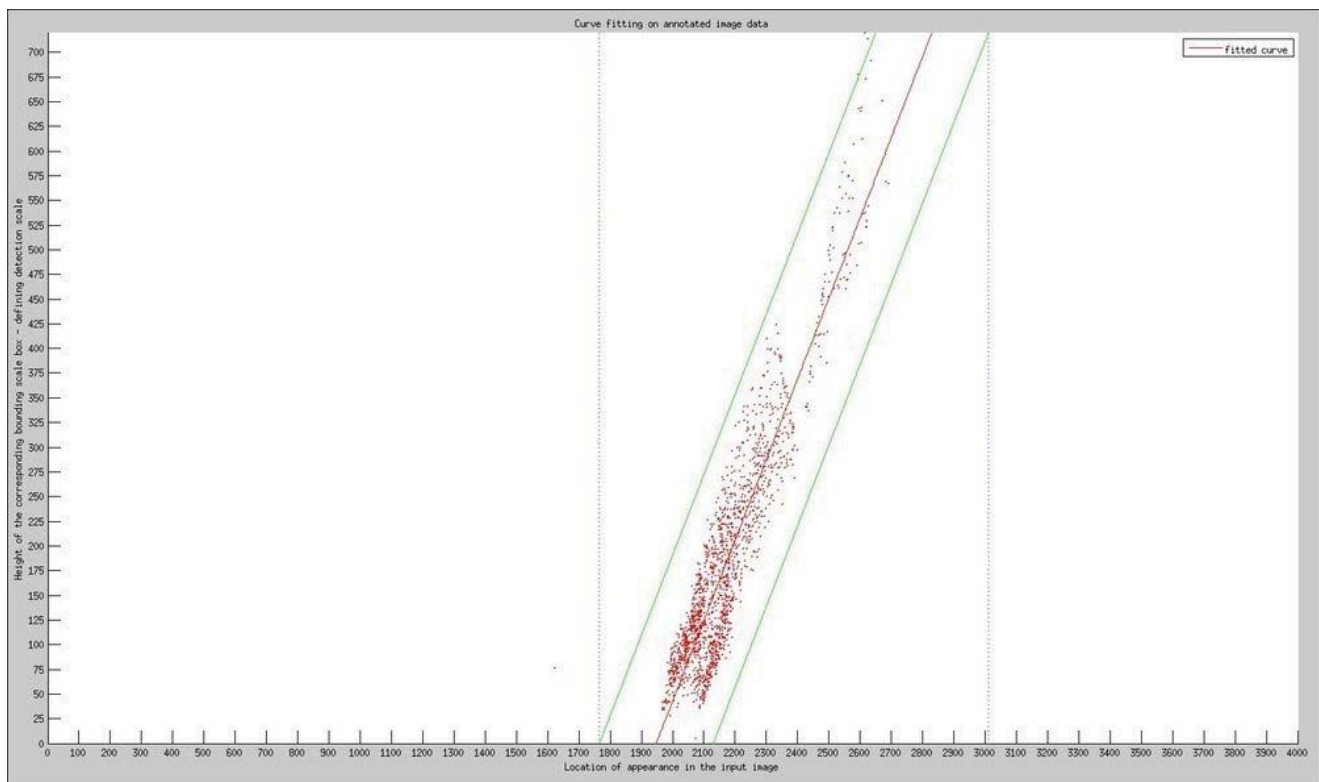
- First of all, you can use this knowledge to effectively reduce the number of false positive detections while still keeping your certainty threshold low enough so that low certainty and good detection still stick around. This can be done in some sort of post-processing step after the object detection phase.
- Secondly, this knowledge can be used to effectively reduce the detection time and search space for object candidates inside the image pyramid.

Let's start by focusing on the following case, illustrated in the following figure. Consider the fact that we want to create a pedestrian detection and that we have an existing model for doing so. We have a 360-degree camera mounted on top of a car and are grabbing those cycloramic images at continuous intervals. The cycloramas are now passed on towards the computer vision component that needs to define if a pedestrian is actually occurring in the image. Due to the very large resolution of such a 360-degree cyclorama, the image pyramid will be huge, leading to a lot of false positive detections and a very long processing time.



*An example of the Viola and Jones cascade classifier pedestrian detection model in OpenCV 3 based on HAAR features*

The example clearly shows that when applying the detector it is very hard to find a decent score threshold to only find pedestrians and no longer have a bunch of false positive detections. Therefore, we took a base set of 40 cycloramic images and manually annotated each pedestrian inside, using our object annotation tool. If we then visualized the annotation heights of the bounding box in function of the x position location of appearance in the image, we could derive the following relation, as shown in the following graph:



*Scale space relation between the position of the annotation in the image and the scale of the found detection*

The red dots in this figure are all possible ground truth annotations that we retrieved from the test bench of 40 cycloramas. The red line is the linear relation that we fitted to the data and which describes more or less which scale should be detected on which location in the image. However, we do know that there could be a small variation on that specific scale as defined by the green borders, in order to contain as much of the annotations as possible. We used the rule of assigning a Gaussian distribution and thus agree that in the range  $[-3\sigma, +3\sigma]$  98% of all detections should fall. We then apply the minimal and maximal value according to our ranges and define a region where objects can occur naturally, assigned with the blue borders and visualized in the following picture:



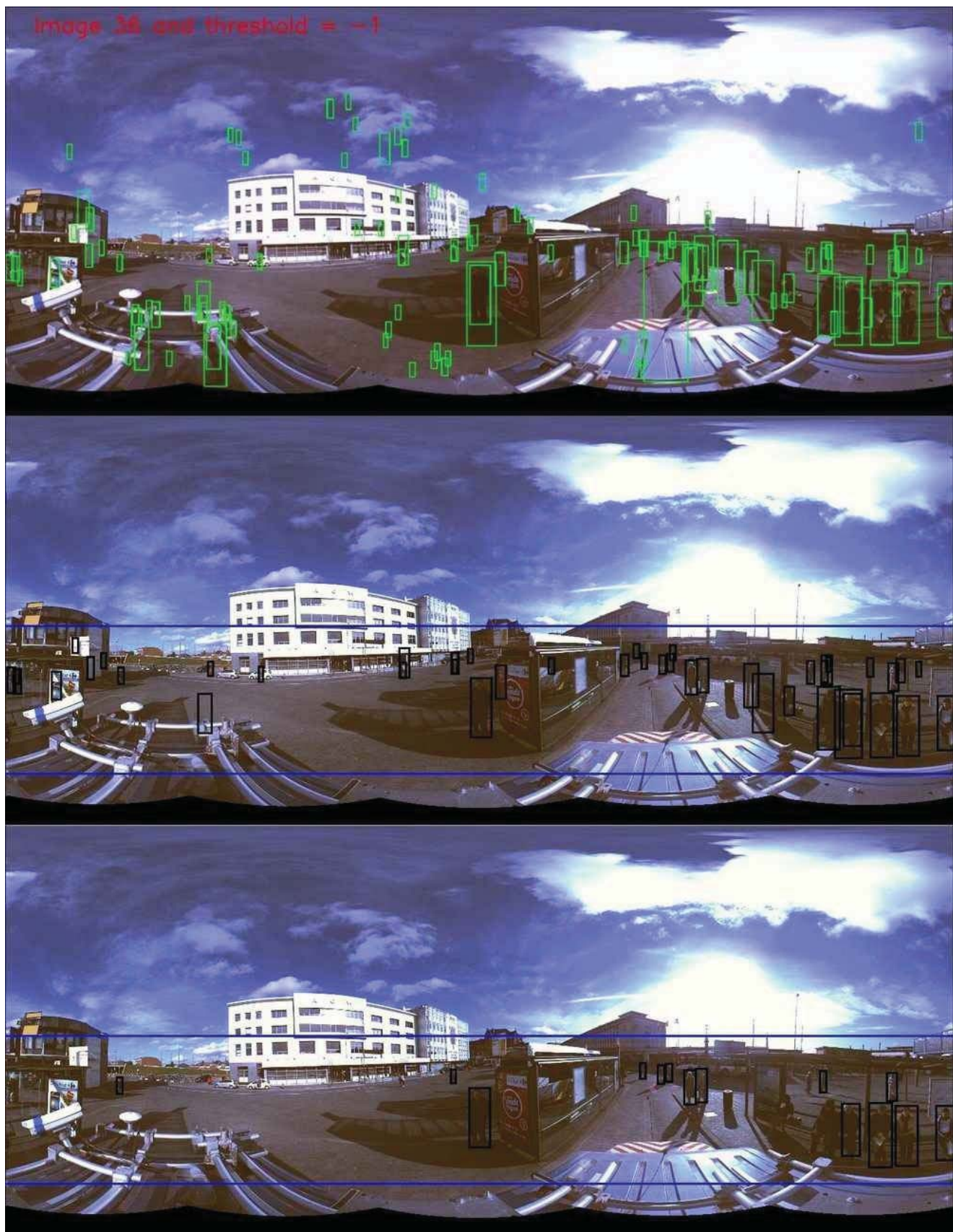


*Possible locations of pedestrians walking in the same ground plane and fully visible by the camera system*

This means that if we run a detector on this input image, we already can eliminate more than 50% of the image because training data clearly shows that a pedestrian cannot occur in that location. This reduces the search space quite a lot! The only downside to this approach of limiting the search space with an image mask is that people on, for example, balconies will simply be ignored. But again, in this application, it was not necessary to find these people since they are not in the same ground plane.

We then finally combined everything we know from this chapter together. We applied a scale space relation for all possible scales that can occur, already only inside the mask area because objects cannot exist outside of it in our application. We then lowered the score threshold to have more detections and to ensure that we have detected as many pedestrians as possible before applying our filtering based on the scale-space relation. The result can be shown here. It clearly shows that there are applications where the contextual information can increase your detection rates a lot!





*The complete pipeline: 1) detection with low threshold, 2) applying the mask and removing a lot of false positives, 3) enforcing the scale space location to remove extra false positive detections*





# Performance evaluation and GPU optimizations

We are heading towards the end of this chapter, but before we end, I would like to address two small but still important topics. Let's start by discussing the evaluation of the performance of cascade classifier object detection models by using not only a visual check but by actually looking at how good our model performs over a larger dataset.

# Object detection performance testing

We will do this by using the concept of precision recall curves. They differ a bit from the more common ROC curves from the statistics field, which have the downside that they depend on true negative values, and with sliding windows applications, this value becomes so high that the true positive, false positive, and false negative values will disappear in relation to the true negatives. Precision-recall curves avoid this measurement and thus are better for creating an evaluation of our cascade classifier model.

*Precision* =  $TP / (TP + FP)$  and *Recall* =  $TP / (TP + FN)$  with a true positive (TP) being an annotation that is also found as detection, a false positive (FP) being a detection for which no annotation exist, and a false negative (FN) being an annotation for which no detection exists.

These values describe how good your model works for a certain threshold value. We use the certainty score as a threshold value. The **precision** defines how much of the found detections are actual objects, while the **recall** defines how many of the objects that are in the image are actually found.

## Note

Software for creating PR curves over a varying threshold can be found at [https://github.com/OpenCVBlueprints/OpenCVBlueprints/tree/master/chapter\\_5/source\\_cc](https://github.com/OpenCVBlueprints/OpenCVBlueprints/tree/master/chapter_5/source_cc)

The software requires several input elements:

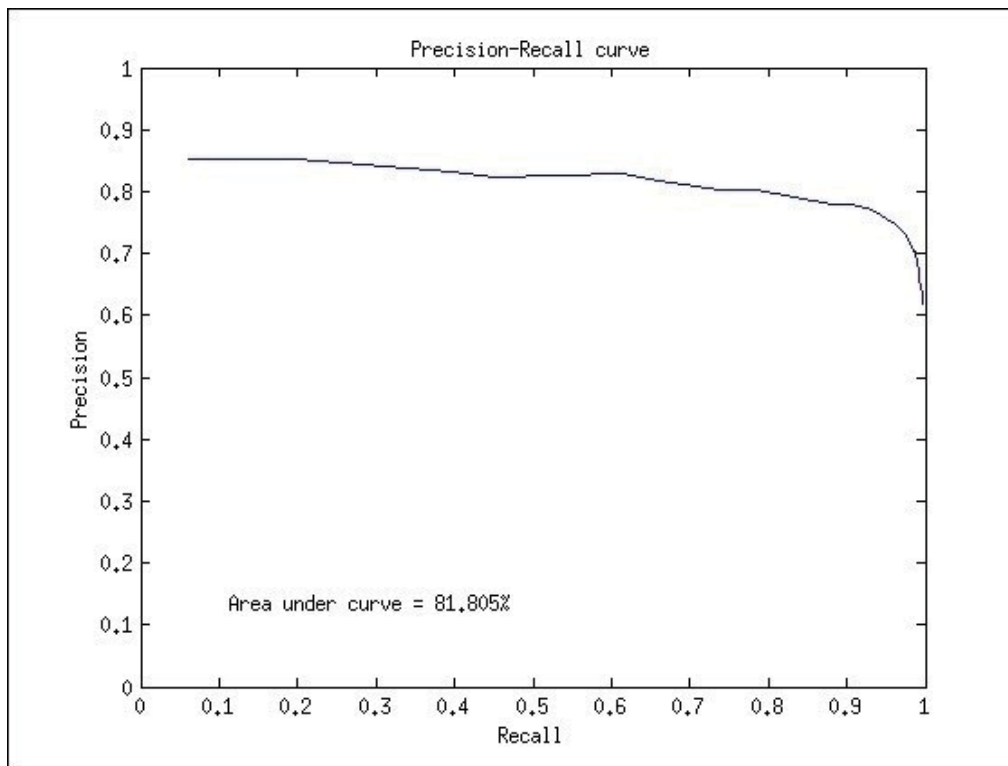
- First of all, you need to collect a validation/test set that is independent of the training set because otherwise you will never be able to decide if your model was overfitted for a set of training data and thus worse for generalizing over a set of class instances.
- Secondly, you need an annotation file of the validation set, which can be seen as a ground truth of the validation set. This can be made with the object annotation software that is supplied with this chapter.
- Third, you need a detection file created with the detection software that also outputs the score, in order to be able to vary over those retrieved scores. Also, ensure that the nonmaxima suppression is only set at 1 so that detections on the same location get merged but none of the detections get rejected.

When running the software on such a validation set, you will receive a precision recall result file as shown here. Combined with a precision recall coordinate for each threshold step, you will also receive the threshold itself, so that you could select the most ideal working point for your application in the precision recall curve and then find the threshold needed for that!

coordinates.txt	
1	0.617763 0.997093 -5
2	0.617763 0.997093 -4.9
3	0.617763 0.997093 -4.8
4	0.617763 0.997093 -4.7
5	0.617763 0.997093 -4.6
6	0.617763 0.997093 -4.5
7	0.617763 0.997093 -4.4
8	0.617763 0.997093 -4.3
9	0.617763 0.997093 -4.2
10	0.617763 0.997093 -4.1
11	0.617763 0.997093 -4
12	0.618008 0.997081 -3.9
13	0.618008 0.997081 -3.8
14	0.619882 0.996987 -3.7

### *Precision recall results for a self trained cascade classifier object model*

This output can then be visualized by software packages such as MATLAB (<http://nl.mathworks.com/products/matlab/>) or Octave (<http://www.gnu.org/software/octave/>), which have better support for graph generation than OpenCV. The result from the preceding file can be seen in the following figure. A MATLAB sample script for generating those visualizations is supplied together with the precision recall software.



### *Precision recall results on a graph*

Looking at the graph, we see that both precision and recall have a scale of [0 1]. The most

ideal point in the graph would be the upper-right corner (precision=1/recall=1), which would mean that all objects in the image are found and that no false positive detections are found. So basically, the closer the slope of your graph goes towards the upper right corner, the better your detector will be.

In order to add a value of accuracy to a certain curve of the precision recall graph (when comparing models with different parameters), the computer vision research community uses the principle of the area under the curve (AUC), expressed in a percentage, which can also be seen in the generated graph. Again, getting an AUC of 100% would mean that you have developed the ideal object detector.



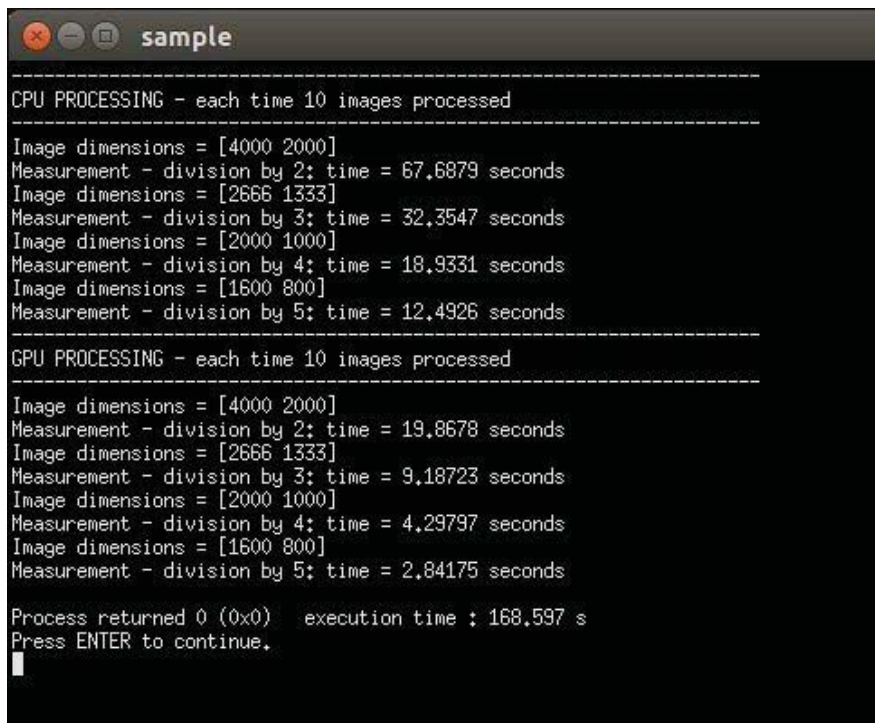
# Optimizations using GPU code

To be able to reconstruct the experiments done in the discussion about GPU usage, you will need to have an NVIDIA GPU, which is compatible with the OpenCV CUDA module. Furthermore, you will need to rebuild OpenCV with different configurations (which I will highlight later) to get the exact same output.

The tests from my end were done with a Dell Precision T7610 computer containing an Intel Xeon(R) CPU that has two processors, each supporting 12 cores and 32 GB of RAM memory. As GPU interface, I am using an NVIDIA Quadro K2000 with 1 GB of dedicated on-board memory.

Similar results can be achieved with a non-NVIDIA GPU through OpenCL and the newly introduced T-API in OpenCV 3. However, since this technique is fairly new and still not bug free, we will stick to the CUDA interface.

OpenCV 3 contains a GPU implementation of the cascade classifier detection system, which can be found under the CUDA module in. This interface could help to increase the performance when processing larger images. An example of that can be seen in the following figure:



```
-----
CPU PROCESSING - each time 10 images processed
-----
Image dimensions = [4000 2000]
Measurement - division by 2: time = 67.6879 seconds
Image dimensions = [2666 1333]
Measurement - division by 3: time = 32.3547 seconds
Image dimensions = [2000 1000]
Measurement - division by 4: time = 18.9331 seconds
Image dimensions = [1600 800]
Measurement - division by 5: time = 12.4926 seconds
-----

GPU PROCESSING - each time 10 images processed
-----
Image dimensions = [4000 2000]
Measurement - division by 2: time = 19.8678 seconds
Image dimensions = [2666 1333]
Measurement - division by 3: time = 9.18723 seconds
Image dimensions = [2000 1000]
Measurement - division by 4: time = 4.29797 seconds
Image dimensions = [1600 800]
Measurement - division by 5: time = 2.84175 seconds

Process returned 0 (0x0)   execution time : 168.597 s
Press ENTER to continue.
```

*CPU-GPU comparison without using any CPU optimizations*

## Note

These results were obtained by using the software that can be retrieved from [https://github.com/OpenCVBlueprints/OpenCVBlueprints/tree/master/chapter\\_5/source\\_cc](https://github.com/OpenCVBlueprints/OpenCVBlueprints/tree/master/chapter_5/source_cc)

For achieving this result, I built OpenCV without any CPU optimization and CUDA

support. For this, you will need to disable several CMAKE flags, thus disabling the following packages: IPP, TBB, SSE, SSE2, SSE3, SSE4, OPENCL, and PTHREAD. In order to avoid any bias from a single image being loaded at a moment that the CPU is doing something in the background, I processed the image 10 times in a row.

The original input image has a size of 8000x4000 pixels, but after some testing, it seemed that the `detectMultiScale` function on GPU would require memory larger than the dedicated 1 GB. Therefore, we only run tests starting from having the image size as 4000\*2000 pixels. It is clear that, when processing images on a single core CPU, the GPU interface is way more efficient, even if you take into account that at each run, it needs to push data from memory to the GPU and get it back. We still get a speedup of about 4-6 times.

However, the GPU implementation is not always the best way to go, as we will prove by a second test. Let's start by summing up some reasons why the GPU could be a bad idea:

- If your image resolution is small, then it is possible that the time needed to initialize the GPU, parse the data towards the GPU, process the data, and grab it back to memory will be the bottleneck in your application and will actually take longer than simply processing it on the CPU. In this case, it is better to use a CPU-based implementation of the detection software.
- The GPU implementation does not provide the ability to return the stage weights and thus creating a precision recall curve based on the GPU optimized function will be difficult.
- The preceding case was tested with a single core CPU without any optimizations, which is actually a bad reference nowadays. OpenCV has been putting huge efforts into making their algorithms run efficiently on CPU with tons of optimizations. In this case, it is not for granted that a GPU with the data transfer bottleneck will still run faster.

To prove the fact that a GPU implementation can be worse than a CPU implementation, we built OpenCV with the following freely available optimization parameters: IPP (free compact set provided by OpenCV), TBB, SSE2, SSE3, SSE4 (SSE instructions selected automatically by the CMAKE script for my system), pthread (for using parallel for loop structures), and of course, with the CUDA interface.

We will then run the same software test again, as shown here.

```
sample
-----
CPU PROCESSING - each time 10 images processed
-----
Image dimensions = [4000 2000]
Measurement - division by 2: time = 12.517 seconds
Image dimensions = [2666 1333]
Measurement - division by 3: time = 5.79939 seconds
Image dimensions = [2000 1000]
Measurement - division by 4: time = 3.47494 seconds
Image dimensions = [1600 800]
Measurement - division by 5: time = 2.26352 seconds
-----
GPU PROCESSING - each time 10 images processed
-----
Image dimensions = [4000 2000]
Measurement - division by 2: time = 19.9184 seconds
Image dimensions = [2666 1333]
Measurement - division by 3: time = 9.18187 seconds
Image dimensions = [2000 1000]
Measurement - division by 4: time = 4.31584 seconds
Image dimensions = [1600 800]
Measurement - division by 5: time = 2.87771 seconds
-----
Process returned 0 (0x0)   execution time : 61.211 s
Press ENTER to continue.
█
```

### *CPU-GPU comparison with basic CPU optimizations provided by OpenCV 3.0*

We will clearly see now that using the optimizations on my system yield a better result on CPU than on GPU. In this case, one would make a bad decision by only looking at the fact that he/she has a GPU available. Basically, this proves that you should always pay attention to how you will optimize your algorithm. Of course, this result is a bit biased, since a normal computer does not have 24 cores and 32 GB of RAM memory, but seeing that the performance of personal computers increase every day, it will not take long before everyone has access to these kind of setups.

I even took it one step further, by taking the original image of 8000\*4000 pixels, which has no memory limits on my system for the CPU due to the 32 GB or RAM, and performed the software again on that single size. For the GPU, this meant that I had to break down the image into two parts and process those. Again, we processed 10 images in a row. The result can be seen in the following image:

```
-----
CPU-GPU PROCESSING on the original, 10 times
-----
CPU Measurement - [8000x4000] pixels: time = 21.446 seconds
GPU Measurement - [8000x4000] pixels: time = 70.4371 seconds
```

### *Comparison of an 8000x4000 pixel processed on GPU versus multicore CPU*

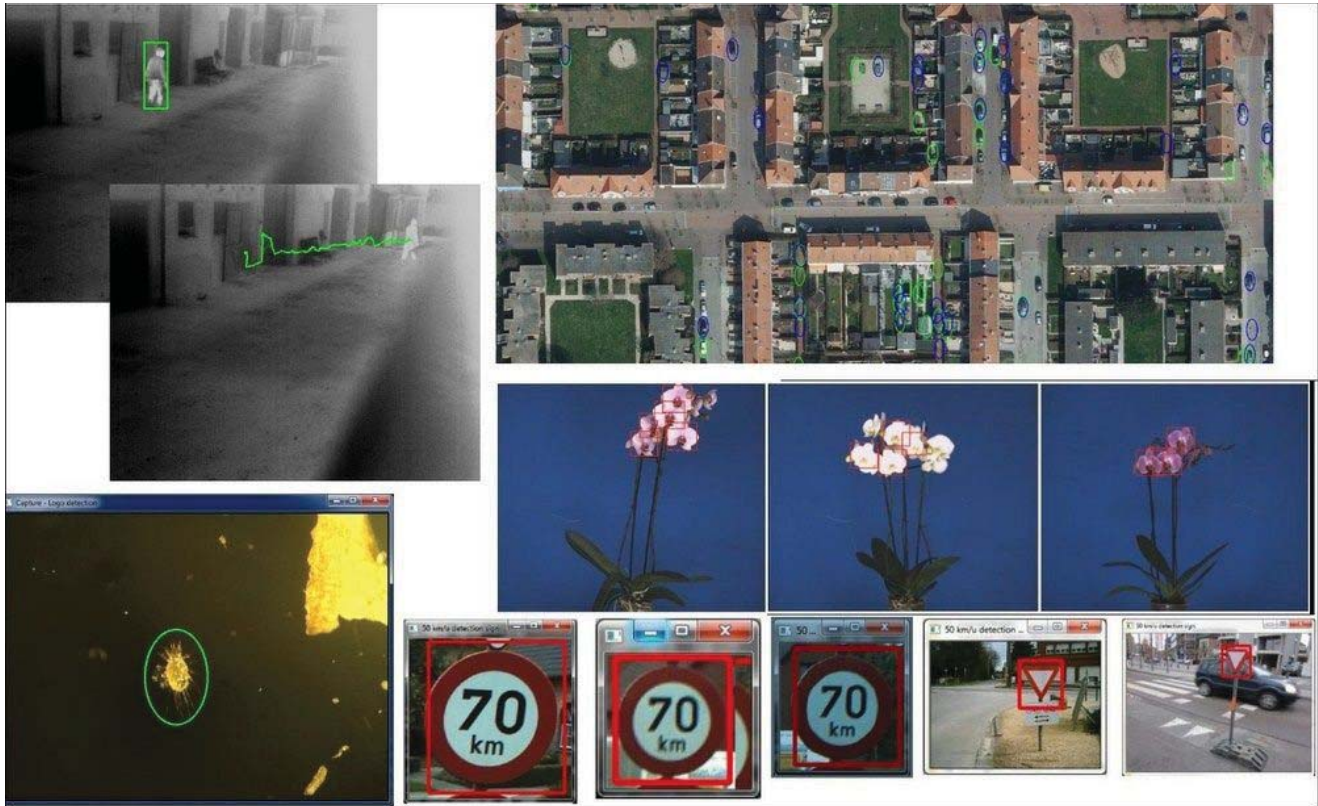
As you can see, there is still a difference of the GPU interface taking about four times as long as the CPU interface, and thus in this case, it would be a very bad decision to select a

GPU solution for the project, rather than a multicore CPU solution.



# Practical applications

If you are still wondering what the actual industrial applications for this object detection software could be, then take a look at the following:



*Examples of industrial object detection*

This is a quick overview of the applications that I used this software for to get accurate locations of detected objects:

- Dummy test cases containing rotation invariant detection of both cookies and candies on a set of different backgrounds.
- Automated detection and counting of microorganisms under a microscope instead of counting them yourself.
- Localization of strawberries for ripeness classification.
- Localization of road markings in an aerial imagery for automated creation of a GIS (Geographic Information System) based on the retrieved data.
- Rotation invariant detection of peppers (green, yellow, and red combined) on a conveyor belt combined with the detection of the stoke for effective robot gripping.
- Traffic sign detection for ADAS (Automated Driver Assist System) systems.
- Orchid detection for automated classification orchid species.
- Pedestrian detection and tracking in NIR images for security applications.

So, as you can see, the possibilities are endless! Now, try to come up with your own application and conquer the world with it.



Let's wrap up the chapter with a critical note on object detection using the Viola and Jones object categorization framework. As long as your application is focusing on detecting one or two object classes, then this approach works fairly well. However, once you want to tackle multiclass detection problems, it might be good to look for all the other object categorization techniques out there and find a more suitable one for your application, since running a ton of cascade classifiers on top of a single image will take forever.

## Note

Some very promising object categorization frameworks that are in research focus at the moment, or that are a solid base for newer techniques, can be found below. They might be an interesting starting point for people wanting to go further than the OpenCV possibilities.

- Dollár P., Tu Z., Perona P., and Belongie S (2009, September), Integral Channel Features. In BMVC (Vol. 2, No. 3, p. 5)
- Dollár P., Appel R., Belongie S., and Perona P (2014), Fast feature pyramids for object detection. Pattern Analysis and Machine Intelligence, IEEE transactions on, 36(8), 1532-1545.
- Krizhevsky A., Sutskever I., and Hinton G. E (2012), Imagenet classification with deep convolutional neural networks. In Advances in neural information processing systems (pp. 1097-1105).
- Felzenszwalb P. F., Girshick R. B., McAllester D., and Ramanan D (2010), Object detection with discriminatively trained part-based models. Pattern Analysis and Machine Intelligence, IEEE Transactions on, 32(9), 1627-1645.





# Summary

This chapter brought together a wide variety of tips and tricks concerning the cascade classifier object detection interface in OpenCV 3 based on the Viola and Jones framework for face detection. We went through each step of the object detection pipeline and raised attention to points where it can go wrong. This chapter supplied you with tools to optimize the result of your cascade classifier for any desired object model, while at the same time suggesting optimal parameters to choose from.

Finally, some scene specific examples were used to illustrate that a weaker trained object model can also perform well if you use the knowledge of the scene to remove false positive detections.

