Vehicle Detection Project Writeup

Charlie Wartnaby

Applus IDIADA

charlie.wartnaby@idiada.com

Udacity Self-Driving Car Engineer Nanodegree

Term 1 Computer Vision and Deep Learning

17 Sep 2017

Abstract

A Python program was successfully written to process single images or a complete video file to identify cars.

This built on the previous project (Advanced Lane Finding), so it was also able to highlight lane lines in the same output, and show numerical data relating to those lines.

The vehicle detection featured the following:

- A 'sliding window' approach to analyse overlapping squares in translation across and down each video frame.
- Use of a linear Support Vector Machine (SVM) classifier to decide whether each window contained a car or not.
- The SVM used normalised HOG (Histogram of Oriented Gradients) features in the HLS colour space.
- Colour transformation and HOG analysis was performed just once for each image at each required scale and then the resulting features extracted for each required window, reducing computational effort.
- The SVM also used normalised RGB channel colour histogram features which improved performance.
- The HOG parameters were explored to find an optimal set for best performance.
- A heatmap was built up from regions identified as 'car' by the sliding windows.
- The heatmap was filtered from frame to frame to reject spurious false positives and accumulate "strength" on genuine hits, effectively tracking genuine detections between successive frames.
- The heatmap was then thresholded to reject weak signals before being analysed for unique areas.
- Visualisation of the fitting process and extracted data by augmentation of the original video with a "picture in picture" view of the window search and heatmap.

Some limitations of the methods used are finally discussed, and possible enhanced approaches discussed.

Write-Up

This document is the project write-up, meeting the rubric points specified. The headings correspond to the structure of the rubric. The output images can be found in **`output_images`**, including the processed video **`output_images/project_video.mp4`**.

Accompanying Code

All of the code can be found in the accompanying file **`annotate.py`**, which is commented extensively; fragments are reproduced in this write-up, but please see the source file for complete implementation.

The entry point is, being Python, at the bottom of the file:

```
if __name__ == "__main__":
    # Validate command-line arguments and run appropriate action(s)
...
```

Legacy functions from Advanced Lane-Finding Project

Camera calibration, birdseye distortion, lane-fitting and real-world data extraction are retained from the previous project, but not discussed again in this write-up. The camera calibration capability was retained for lane-finding purposes but is not required for vehicle identification.

Command-line options

Command-line options have been included to produce output to meet the various rubric points as follows. Validation was included to ensure that required options were all provided together where appropriate. The program could be called to perform camera calibration, train the vehicle classifier (or load a previously trained one from disk), to work on single images, or to process a video as required. Options relating to the previous lane-finding project are greyed:

Command-line option	Description
--cam_cal_path_pattern	Path pattern to match camera calibration images
--cam_cal_nx	Expected number of calibration image x-direction corners
--cam_cal_ny	Expected number of calibration image y-direction corners
--cam_cal_eg_src	Example image to process to demonstrate camera calibration
--cam_cal_eg_dst	Output image to demonstrate camera calibration
--img_path_pattern	Process single JPEG images matching this path pattern
--img_out_dir	Output directory for processed single binary images
--video_in_file	File path for input video to process
--video_out_dir	Output directory for processed video
--find_lanes	Whether to identify and visualise lane lines and related extracted values
--find_cars	Whether to identify vehicles
--car_training_zip	File of images containing cars, if classifier is to be trained
--noncar_training_zip	File of images not containing cars, if classifier is to be trained
--save_classifier	Pickle file to save built classifier to
--load_classifier	Pickle file to retrieve built classifier from

Histogram of Oriented Gradients

Implementation

To avoid unnecessary computational overhead, the HOG feature matrix was extracted just once for the whole image slice of interest at each scale it was analysed at. The required subset from that matrix corresponding to each detection window was then extracted and fed, along with colour features, to the classifier for vehicle detection.

HOG features were computed for the whole image slice in **`get_whole_image_hog_features()`**. This function computed HOG features over each of the H, L and S colour channels and stacked them as an array.

The function **extract_features()** then took those HOG features for the whole image slice, extracted the required subset corresponding to the window of interest, and normalised them:

```
hog_matrix_subset = hog_features[y_block_offset:y_block_offset+num_blocks_per_dimension,
                                 x_block_offset:x_block_offset+num_blocks_per_dimension]
hog_vector_raw = hog_matrix_subset.flatten()
hog_features = normalise_vector(hog_vector_raw)
```

The HOG features were then concatenated with normalised colour histogram features to feed to the classifier.

HOG Parameter Optimisation

To tune the parameters, the classifier was trained on the dataset supplied (**`vehicles.zip`** and **`non-vehicles.zip`**) with different permutations of parameters until the best performance was identified. This was done using just the S channel in the HLS colour space; later, HOG analysis was extended to cover H and L channels also.

One experimental dimension was to explore whether good performance could still be obtained using a smaller image size (32x32 instead of 64x64 pixels), to reduce run-time computational load when analysing video.

Pixel size to match	HOG orientation bins	HOG pixels per cell	HOG cells per block	Test accuracy
32	8	8	1	0.8834
32	8	8	2	0.8925
32	8	8	3	0.8846
32	8	8	4	0.8606
32	8	4	2	0.9062
32	8	2	2	0.891
64	8	2	2	0.9001
64	8	4	2	0.9186
64	**8**	**8**	**2**	**0.9291**
64	8	16	2	0.9102
64	4	8	2	0.9169
64	16	8	2	0.9229
64	**12**	**8**	**2**	**0.9268**
64	**6**	**8**	**2**	**0.9268**

In conclusion it was better to use the training images at full 64x64 resolution, with around 8 orientation bins (but the accuracy was not too sensitive to this), using 8 pixel square cells within the image, and with two cells per HOG normalisation block. These were in fact the defaults in the tutorials, which must have already been chosen carefully!

Choice of Colour Space

In initial experiments just the S channel was used, as cars tend to have strong (saturated) colours.

However, the classifier test accuracy increased considerably by using the H and L channels also, so these were used too. But the separation into H, L and S channels at least gave the classifier to make good use of the S channel if, as hoped, it gave stronger signals for vehicles.

Classifier and Training

As suggested a linear State Vector Machine (SVM) classifier was used, which obtained excellent test accuracy.

The dataset supplied for the project (**`vehicles.zip`** and **`non-vehicles.zip`**) was used for training.

As explained above, the original 64x64 pixel size was used as this produced better results than reducing the size (in attempt to improve speed).

The classifier construction and training was implemented in **`construct_train_classifier_from_zips()`**.

`sklearn.cross_validation.train_test_split()` was used to shuffle the dataset to avoid bias, and to reserve 20% of the data as an independent test set for validation.

The following features were fed to the classifier for each 64x64 image analysed:

- HOG features for each of the H, L and S channels, concatenated and normalised.
- Colour histogram for the whole image patch in the R, G and B channels, again normalised.

Normalisation was implemented in the utility function **`normalise_vector()`**, which rescaled a feature vector to the range [-1, 1]. (This was not quite normalising to zero mean and unit variance, but was faster to compute.)

Options were provided to save the trained classifier to disk as a Pickle file to retrieve for subsequent reuse, to avoid the overhead of retraining each time the program was run on new input or with different processing options.

Sliding Window Search

Square windows were translated across and down the image, and the classifier run on each patch to make a car/non-car decision. The top-level loop over all search windows was implemented in **`find_cars_at_all_scales()`**.

A first-order filtered heatmap followed by thresholding was used to effectively track genuine detections between frames while rejecting spurious false positives.

These aspects are discussed in more detail in the following subsections.

Region of Interest and Scaling

To reduce computational load and avoid false positives in "impossible" image regions, the area covered by the window search was limited to where cars might reasonably be found. The upper and lower limits of the region of interest were defined in terms of proportions of the frame height (so that video of different resolution could be processed without changes), but were "calibrated" on the 1280x720 test image. The definitions are in **`FrameProcessor.__init__()`**:

```
self.useful_top_y_range_propn = 390/720 # in test images no chance of cars above about y=400
self.useful_bottom_y_range_propn = 640/720 # in frames where car very close, useful bottom of
car never beneath about y=640
```

Further parameters defined the "smallest" (i.e. most distant) and "biggest" (nearest) matching regions that were used to match cars:

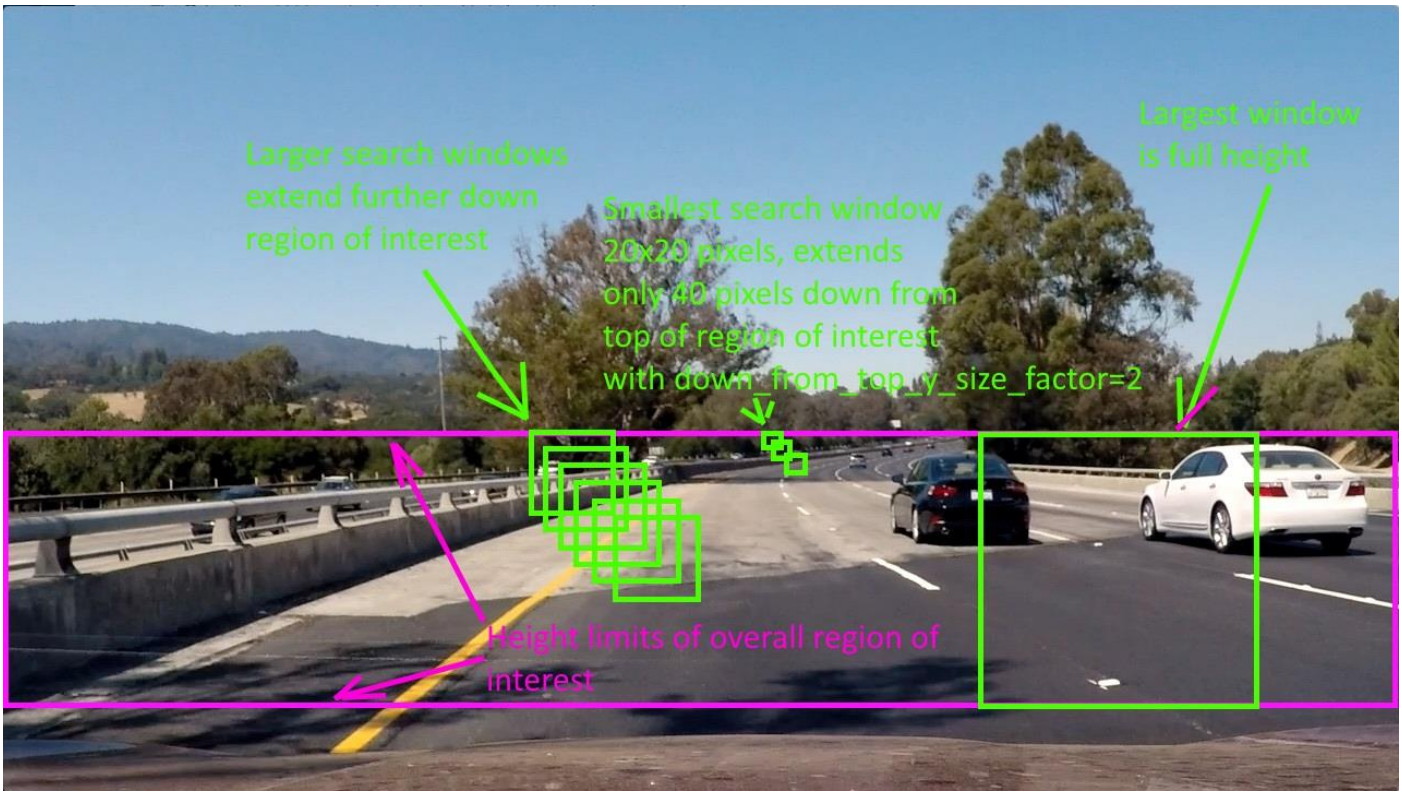
```
self.smallest_car_height_range_propn = 20/720 # in test images smallest useful (distant) cars
about 20 pixels high
self.biggest_car_height_range_propn = self.useful_bottom_y_range_propn -
self.useful_top_y_range_propn # nearest car fills useful y range
```

It was noted that while "small" cars could only realistically appear higher up in the frame – a "small" car low in the frame could only be a toy! – "large" cars could appear high or low in the frame, depending on their position relative to the ego car. So a simple rule was used that the larger the box, the further down the image it would be tested. This

saved computational load by avoiding searching many small windows low in the frame where they could not be real cars anyway, as defined by this parameter:

```
self.down_from_top_y_size_factor = 2 # e.g. if 1.5, a 20-pixel high search box could go as low as (1.5*20=30) to 49 pixels below topmost useful y
```

The limits of the region of interest, and where windows of different sizes would be tested vertically, are shown approximately by the boxes here on **`test4.jpg`**; note that the smallest search boxes are sized to be similar to the most distant reasonable cars in this test image, while the largest search box occupies the full possible height where a car may appear:



Finally a number of intermediate window sizes were tried between the smallest and largest. The number of 'steps' was defined by this parameter:

```
self.num_scale_steps = 5
```

Given that the classifier tended to find fragments of cars (i.e. it was not very sensitive to scale), that seemed an adequate number of steps.

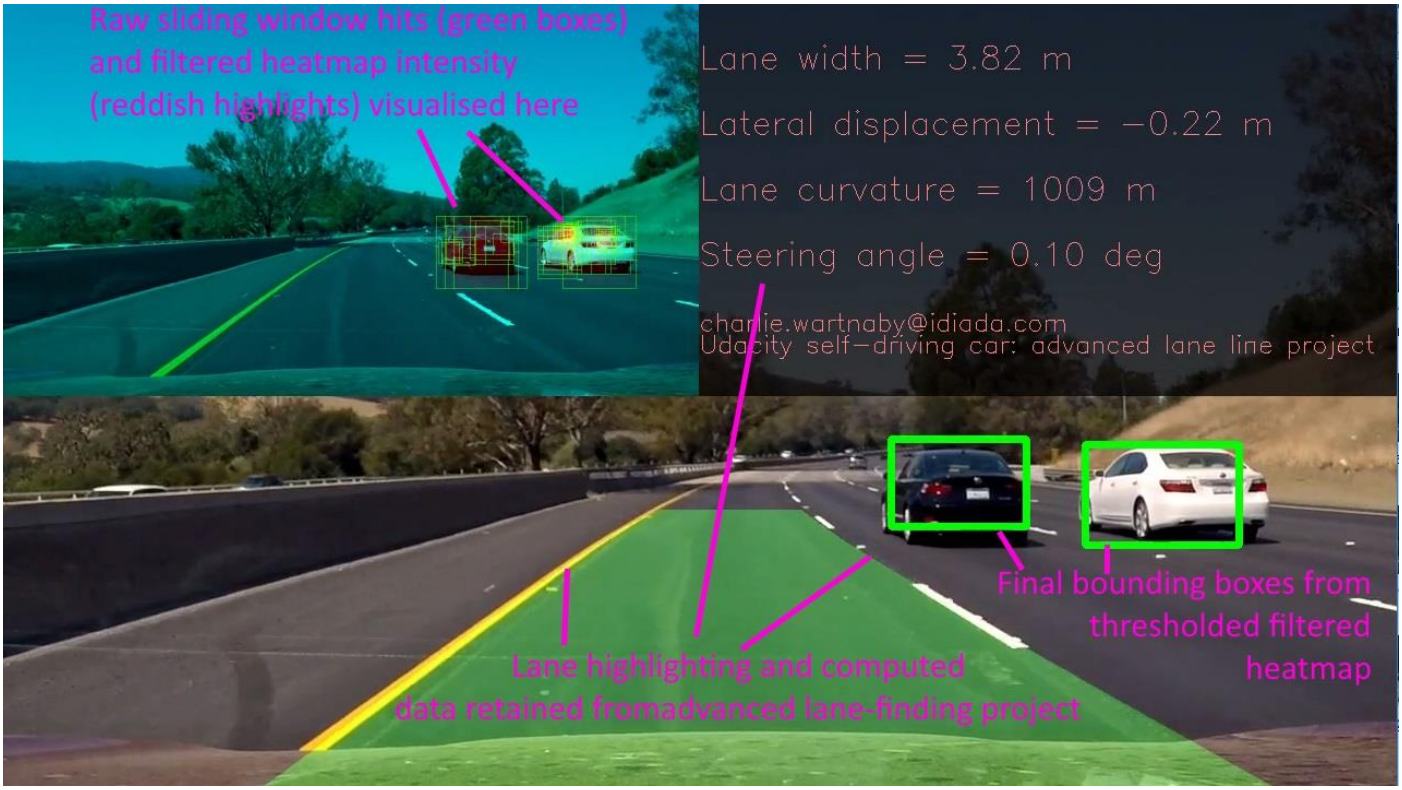
A geometric progression was used to iterate over the different sizes (i.e. taking linear steps in the logarithm of the size). So the *relative* increase in size from one window set to the next was constant. By contrast if linear steps were made, the 'jump' would have been excessively large from the smallest window to the next-smallest, in terms of relative size, while the difference between the larger sizes would have been too small to be useful. This was controlled by these variables in **`find_cars_at_all_scales()`**:

```
        # If we iterated linearly from smallest size to biggest, that would be a
        # large relative step at the small end, but a small relative step at
        # the big end. So use a log scale to keep the interval sensible as a
        # geometric progression
        log_smallest_size = math.log(smallest_pixel_size)
        log_biggest_size  = math.log(biggest_pixel_size)
```

```
log_step          = (log_biggest_size - log_smallest_size) / (self.num_scale_steps - 1)
```

Visualisation of window 'hits'

A small picture-in-picture view highlighted the windows of different scales and at different, overlapping positions where raw car detections were made, so this can be seen in every frame of the output video as highlighted here:



Rescaling to feed classifier

The smallest search window (20x20 pixels) was significantly smaller than the training image size used (64x64 pixels), while the largest search window (250x250 pixels) was significantly larger. Therefore the classifier could not be used directly on the square windows.

Instead, at each new window size, the applicable region of interest was rescaled such that the search window would be of the same size as the training patch size (64x64); that is, for small windows the useful image slice was expanded, while for large windows it was shrunk, so that the search window became 64x64 pixels after scaling. This resizing was done just once at each scale required to avoid computational effort, and the HOG analysis run once on that resized image.

The HOG and colour analysis and classifier were then run on that scaled image in **`find_cars_in_windows()`**, with the window sliding across and down in steps defined by this parameter at that new scale:

```
self.car_detect_patch_shift_px = 8 # pixels to shift along when sliding window, in x and y direction
```

So again, the degree of overlap was constant in terms of proportion of the window size, being just a few pixels in the *original* image for small windows but a much bigger *absolute* jump for large windows. The aim of this was to make it reasonably likely that any car shape would be detected by the classifier, without the degree of *relative* overlap becoming excessive (and so computationally expensive) for large windows. This degree of overlap (an eighth of a "whole car") was enough to make detections, but computationally much cheaper than (say) iterating only one pixel at a time.

The corner coordinates defining windows in which 'car' detections were made were rescaled back to the original image coordinate system for further processing here in **`find_cars_in_windows()`**:

```
features = self.extract_features(slice_colour_image, hog_feature_matrix, x_offset, y_offset)
predict_result = self.classifier.predict(features)
if (predict_result > 0.5):
    # Convert coordinates of window back to those of original image;
    # left edge is always same, top edge as supplied
    x1 = int(round(x_offset / to_scaled_factor))
    x2 = int(round((x_offset + self.car_detect_patch_size_px) / to_scaled_factor))
    y1 = int(round(unscaled_top_y + (y_offset / to_scaled_factor)))
    y2 = int(round(unscaled_top_y + ((y_offset + self.car_detect_patch_size_px) /
to_scaled_factor)))
    # Record nested tuple of opposite window corners in original image coordinates
    window_corners_with_car.append(((x1,y1),(x2,y2)))
```

Tracking vehicles through time and rejecting false positives

A unified approach was taken to building confidence in vehicle detections which were consistent between frames, tracking genuine "hits", while rejecting transient false positives.

Weight was added to a heatmap for every pixel in a window in which a "car" detection was made in the classifier. This meant that multiple detections in neighbouring or overlapping detection windows were effectively summed to reinforce a genuine detection, at one instant in time.

However, this weight was filtered in to the "previous" heatmap using a first-order time constant, so that intensity would build in regions where cars were detected consistently between frames, while transient intensity would decay away to zero through time. This was the means by which genuine detections were tracked through time. This was implemented here in **`identify_visualise_cars()`**:

```
# Gradually filter out previous heatmap results across the board, 1st order decay
self.filtered_heatmap *= (1.0 - self.heatmap_filter_in_const)
...

# Filter in new hits to heatmap, so repeated detections in both space (different
# windows) and time (between frames) accumulate
self.filtered_heatmap[y1:y2,x1:x2] += heat_increment
```

It was not found necessary to extrapolate the trajectory of 'hot' regions between frames, because the genuine movement of cars moving in the same direction[1] on the highway was slow compared to the filter time. However, this will have introduced a slight lag into the position identified for each genuine target, of the order of a few frames.

The filtered heatmap was shown in red in the small visualisation image in the top-left corner of every video frame.

Finally, a threshold was applied to the heatmap to make a binary car/no car map:

```
        # Apply thresholding to discard weak signals, likely false positives
        thresholded_heatmap = np.zeros(self.filtered_heatmap.shape)
        thresholded_heatmap[self.filtered_heatmap >= self.heatmap_threshold] = 1
```

The bounding boxes of the surviving 'confirmed' car regions were then identified using **`scipy.ndimage.measurements.label()`** and **`scipy.ndimage.measurements.find_objects()`**, for display on each frame in bright green:

[1] This would have been less satisfactory for tracking cars travelling in the oncoming direction on a two-way road without barriers.

```
        for bounding_box in found_objects:
            slice_y, slice_x = bounding_box
            x1,x2,x_stride = slice_x.indices(self.x_size)
            y1,y2,y_stride = slice_y.indices(self.y_size)
            cv2.rectangle(annotate_image, (x1,y1), (x2,y2), (0,255,0), 5)
```

Final Output

Please see **`output_images/project_video.mp4`** for the final output. This includes:

1. The final detections of other cars shown as bright green bounding boxes.
2. The raw window hits and time-filtered heatmap intensity shown in the 'picture in picture' version in the top left.
3. Lane line highlighting and extracted data parameters (e.g. road curvature) still present from the previous advanced lane-finding project.

The video was produced by invoking the program in the carnd-term1 Anaconda environment as follows:

```
(C:\udacity\Miniconda3\envs\carnd-term1) C:\svn\courses\udacity\self_driving_car\CarND-Vehicle-
Detection>python annotate.py --find_cars --find_lanes --load_classifier classifier.p --
video_in_file project_video.mp4 --video_out_dir output_images --cam_cal_path_pattern
camera_cal/calibration*.jpg --cam_cal_nx 9 --cam_cal_ny 6
Calibrating camera expecting 9 by 6 corners in images matching 'camera_cal/calibration*.jpg'
Warning: expected corners not found in 'camera_cal\calibration1.jpg', skipping that file
Warning: expected corners not found in 'camera_cal\calibration4.jpg', skipping that file
Warning: expected corners not found in 'camera_cal\calibration5.jpg', skipping that file
17 image files had usable chessboard corner points
Camera calibration complete
Loading saved classifier from classifier.p
Processing video file: project_video.mp4
[MoviePy] >>>> Building video output_images\project_video.mp4
[MoviePy] Writing video output_images\project_video.mp4
100%|#################################################################################9| 1260/1261
[3:42:32<00:08,  8.58s/it]
[MoviePy] Done.
[MoviePy] >>>> Video ready: output_images\project_video.mp4

Processed video written to: output_images\project_video.mp4
```

Discussion

Although the program as developed was reasonably successful for the required test video, it has its limitations.

The filtering of heatmap intensity between frames did not attempt to account for the *trajectory* of confirmed hits, because for vehicles travelling in the same direction, the movement across the video frame was slow compared to the filtering time. However, this would not have been the case for oncoming vehicles on a two-way road.

The algorithms used successfully combined vehicle "hits" in adjoining or overlapping search windows. However, this also meant that two cars which were genuinely overlapping in the image were treated as one. Distinguishing between different individual cars would take more subtle analysis of different colours, trajectory tracking starting from before the cars were visually overlapping (if available in the video clip at all), or detecting the slightly different trajectories of blocks of pixels representing the different cars. This would be very challenging however for two cars of similar colour travelling at the same speed throughout a video clip which overlapped from the point of view of the ego car. In a real self-driving car, LIDAR, radar or stereo imaging might distinguish the two using range information.

Also, no video taken at night, or in poor light, or in heavy rain or even snow was analysed. The training data would need augmenting for these different conditions for the classifier to be able to make detections given these different circumstances, and overall the difficulty would be increased.

The video processing of the 50 second clip was very slow, taking over three hours on a business laptop – two orders magnitude slower than real time. This would be no good for continuous operation in a self-driving car. For real-time deployment, significant effort would have to be put on run-time optimisations, such as:

- paring down the number of windows used in each search;
- working at lower resolution to reduce the size of matrix operations;
- skipping video frames, effectively reducing the time resolution of detections;
- using parallel processing across multiple CPU or GPU cores;
- removing the unnecessary visualisation graphics which demonstrate the detection process.