Udacity CarND Vehicle Detection and Tracking Project

Aleksander Czechowski

September 3, 2017

The purpose of this project was to implement a vehicle detection and tracking algorithm. The main goal of the project was to reliably detect the vehicles in the video project_video.mp4. The objective was achieved, as documented in the video output.mp4. All the relevant code is contained in the notebook VD.ipynb.

1 Rubric points

1.1 Histogram of oriented gradients

1.1.1 Explain how (and identify where in your code) you extracted HOG features from the training images. Explain how you settled on your final choice of HOG parameters.

The features for the classifier are extracted by the *extract_features* function, and are formed by a vector that consists of three parts:

- spatial features,
- a color feature histogram,
- histogram of oriented gradients (*hog*).

The code for each extraction is contained in functions *bin_spatial*, *color_hist*, and *get_hog_features*, respectively.

Many adjustments have been made to find a combination of parameters that give a satisfactory accuracy for the classifier. As a rule of thumb, any combination that would yield accuracy below 99% was discarded. Once I found a combination, that gives accuracy of above 99%, I decided to stick with it, and try to improve the performance of the detector on the video during the video processing part.

After manual adjustment of I chose the following parameters for the histogram of gradients (the abbrieviations are the same as in the online course):

$$
\begin{aligned}
color\_space &= ' \, LUV' \\
orient &= 8 \\
pix\_per\_cell &= 8 \\
cell\_per\_block &= 2 \\
hog\_channel &= 0 \\
spatial\_size &= (32, 32) \\
hist\_bins &= 32
\end{aligned}
$$

The spatial binning features were set to $(32, 32)$ and the number of color histogram feature bins to 32.

I was experimenting with $YCrCb$ color space that was reported to give good performance on the forums, however I could not fine tune it to obtain satisfiable results.

1.1.2 Describe how (and identify where in your code) you trained a classifier using your selected HOG features (and color features if you used them).

The classifier was trained in the 6th block of code of the jupyter notebook, the one starting with the comment *HERE WE TRAIN THE CLASSIFIER*.

An instance of *StandardScaler* was used to scale the features before training. The same scaler was later used in the video pipeline in the *find_cars* function.

The classifier was trained by use of the Linear Support Vector Classification method with default configuration. The images, where a car is present were labelled with 1, and the ones without a car with 0. The classifier based on Linear SVC method took about 13 seconds to complete the computations and achieved a test accuracy of 0.9903 (20% of images were used to form the test set).

1.2 Pipeline

Below, I summarize my pipeline, which is executed in the function *process_image*.

1.2.1 Provide an example of a distortion-corrected image.

A distortion-free image from the road is provided below:

a

Figure 1: A road photo, before undistortion (left) and after (right).

1.2.2 Describe how (and identify where in your code) you used color transforms, gradients or other methods to create a thresholded binary image. Provide an example of a binary image result.

After undistorting, I used a combination of color and gradient thresholds. The thresholding is performed in the master function *combine_threshold*, which calls

each of the subfunctions that perform the thresholding, and overlays the resulting binary output on a single image.

Namely, I used the following thresholds:

- Canny edge detection (function *Canny_threshold*) with $minVal = 50$ and $maxVal = 150$;
- a color threshold based on the S channel in the HLS color space (function *hls_threshold*), activation for $S \in [70, 255]$;
- a Sobel threshold, both in x and y directions (functions *abs_sobel_thresh* and *combine_Sobel*), with range $[10, 255]$ and $kernel = 3$ for both. The Sobel thresholds are taken in conjunction, i.e. only the pixels selected in both directions contribute to the final binary output.

I also implemented magnitude and direction thresholds (functions *mag_thresh*, *dir_threshold* and *combine_MagDir*), but ultimately they were not included in the pipeline, as they did not bring any significant improvement. Regardless, I will keep the implementation for future.

1.2.3 Describe how (and identify where in your code) you performed a perspective transform and provide an example of a transformed image.

The perspective transform was performed in two steps. First, one of the straight lines images was used to identify four corner points to define the transform, which were in turn used to compute the perspective transform matrix (the function *get_perspective_transform*, using *cv2.getPerspectiveTransform*).

The second step was to apply the matrix M obtained from *cv2.getPerspectiveTransform* to the processed image.

1.2.4 Describe how (and identify where in your code) you identified lane-line pixels and fit their positions with a polynomial?

In order to identify lane-line pixels, I performed the procedure of finding window centroids, as described in details in the course. To give a rough overview, the image is divided into 9 horizontal layers. Within each layer, a window search is performed, by convolving the binary image from the previous step, with a sliding window. The position, where the window yielded highest activation is selected, and the center of each such window is later used to fit the polynomial (function *find_window_centroids*).

Each (left/right) set of window centroids has been interpolated as a function of Y (so rotated 90 degrees), with a polynomial of the form:

$$X = AY^2 + BY + C$$

Simple reality checks were performed to stabilize the interpolation for the video (function *reality_checks*). It is expected, that the polynomial curves will vary continuously from frame to frame, so if the difference between the current and the previous curve (in L^2 norm) would exceed a given threshold, then a replacement curve (based on previous frames) was substituted.

The thresholds were set to 600 for the left lane line and 200 for the right lane line. The x-values for the replacement curve were computed with a power series:

$$x_{saved}(t) = (1/3)\sum_i (2/3)^{i-1} x(t-i)$$

and stored in the *saved_line* class instance. The coefficients in this series were chosen quite arbitrarily, but seem to have a nice smoothing effect on the video.

1.2.5 Describe how (and identify where in your code) you calculated the radius of curvature of the lane and the position of the vehicle with respect to center.

The curvature and the respective position was calculated in the function *curvature_and_offset*.

The curves are second order polynomials, so the radius of curvature can be computed from the formula

$$R_{curve} = \frac{(1 + (2AY + B)^2)^{3/2}}{|2A|}.$$

Then, I averaged from the values of the left and the right lane line.

To compute the displacement, I assumed that the camera is mounted in the middle of the car. Then, the formula the displacement is simple, since in a centered position the left and the right line needs to be equidistant from the center, and sum up to the width of the image:

$$\text{displacement} = (\text{image\_width} - C_{left} - C_{right})/2.$$

In these computations I assume I am at the bottom of the image, which in my pipeline corresponds to $Y = 0$ (as contrary to $Y = Y_{max}$ in the example pipeline). Values above zero indicate that the car is on the right side of the lane, below zero, that it is left of the center.

The calculations were performed in meters, and I assumed that each pixel consists of 60/720 meters in the y direction and 3.7/700 meters in the x direction. In the y direction I have about twice as many meters per pixel, as recommended in the course, but this is because I choose my perspective transform in such a way, that the lines are more squeezed in the vertical direction. The curvature and displacement values were displayed in the upper left corner of the processed images and of the video.

1.2.6 Provide an example image of your result plotted back down onto the road such that the lane area is identified clearly.

Outputs of all provided test images are displayed below:

1.3 Pipeline (video)

1.3.1 Provide a link to your final video output. Your pipeline should perform reasonably well on the entire project video (wobbly lines are ok but no catastrophic failures that would cause the car to drive off the road!)

Here is a link to the **output video**.

1.4 Discussion

1.4.1 Briefly discuss any problems / issues you faced in your implementation of this project. Where will your pipeline likely fail? What could you do to make it more robust?

The pipeline does not really perform well on the challenge videos. A more structured approach to backtracing history and reality checks could have been taken (which would incorporate e.g. confidence in the result), but given time constraints only a limited version of these was implemented.

Also, some variables could have been precomputed outside of the pipeline, which would have accelerated video processing.