Vehicle Detection project write-up

The code I used for this project is in my CarND1_Project5 github repository as a jupyter notebook: Project5.ipynb. The code extensively used the functions we developed in the course, with modifications as needed to optimize performance.

Since I am running dangerously close to the deadline cutoff & I am aware that submission review could take longer near end-of-term, this write-up is shorter (but hopefully covers all rubric points), and should be viewed along with the project notebook. Images are in the notebook (and some are in this report too—I have not saved them separately in the repository as of this writing. I'm trying to save some time in case any changes are required after the review.

Training Dataset

In the beginning I planned to use only the KITTI dataset (for cars) and Extras (non-cars), as I recalled that SVMs are prone to overfitting, and also slow down on datasets that are too large. In addition my own experience with the traffic sign classifier project was that accuracy could be very high, but the classifier did a mixed job dealing with images I fed it from the internet. I also wanted to "save" the other datasets for a "truer" test accuracy metric (which I had hoped to implement).

In hindsight, I think I'm glad I did that. My classifier accuracy was almost always 98-99% or higher, but was not a good indicator of how the classifier would perform with the test images or the video file. At the projects conclusion, I added the other dataset for vehicles, but I'm not sure if it ultimately helped in the processing of the video file.

The KITTI dataset has 5966 car images in .png format, while the "Extras" set has 5068 non-car images; all training images are of 64x64 pixel resolution. These are the images I initially used for training my SVM classifier. I explored the color space to discern trends in car and non-car images. On the statistically insignificant number of images I evaluated, I could argue that car images in the YUV or YCrCb color space had a more "concentrated" color distribution but I am not sure if that is the case across the board. I have had reasonably favorable experience with luma based color spaces especially in regards to tolerance to shadowing effects. In either case I decided to go with YUV. Somewhat surprisingly, to me, early on in the project the classifier was trained on YUV images, while the test images had been converted to YCrCb color space, and I still got reasonably good detection of cars in the video files.

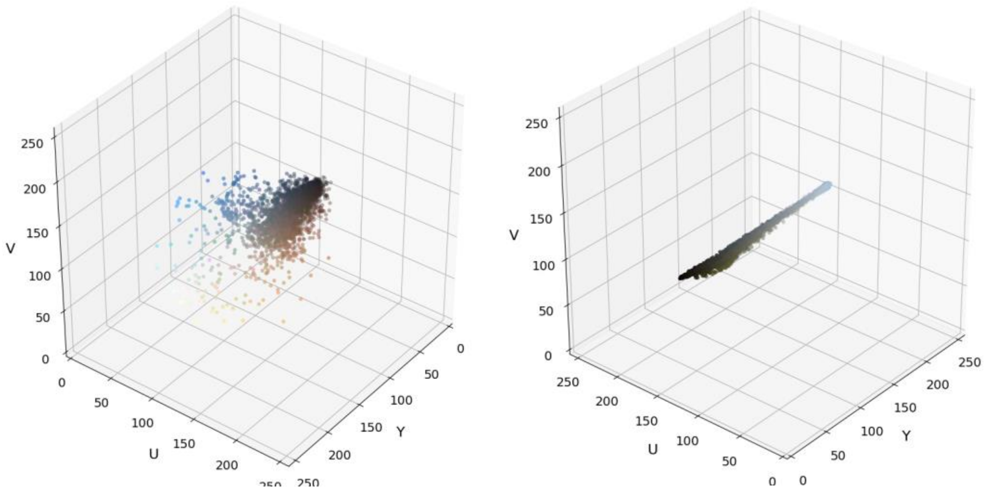

Figure 1. Non-car (left) and Car(right) images pixel binning in YUV space. The non-car image has a much wider distribution than the car image, which may help the classifier

Histogram of Oriented Gradients (HOG) parameters

The HOG function takes in a single color-channel to compute gradients. To test this out, I fed in some

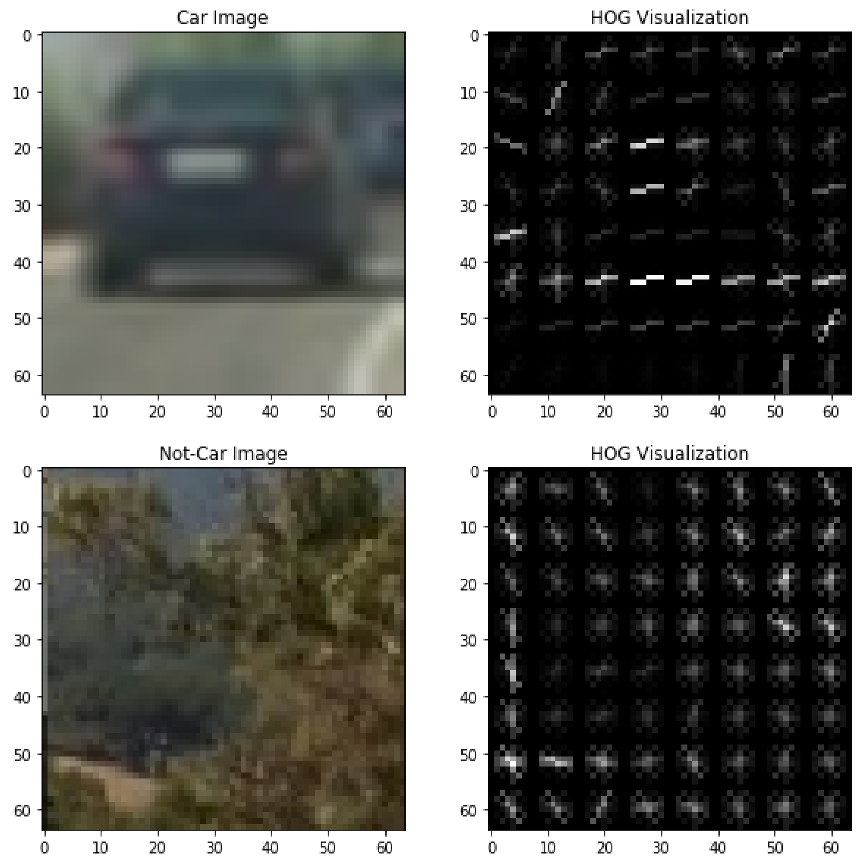

Figure 2. HOG transform of car and non-car image using the Y-channel of images on left

sample car and non-car images into the get_hog_features function to get the output shown in Figure 2. For starters I used the parameters used in class: orientation bins=9, pixels per cell: 8x8, and cells per block=2x2.

For training the classifier, and in order to speed up video image processing, I settled on 5 orientation bins, 16 pixels per cells, 2 cells per block and concatenating the HOG for all 3 color channels (YUV). The orientation bins, and pixels per cell had a significant impact on frame processing rate. These changes nearly doubled the processing speed compared to the parameters used in the previous paragraph, without adversely impacting the test accuracy of the classifier, and more importantly the correct identification of cars in the test images and video frames.

Additionally the length of the feature vector (from the extract_features function) dropped down from 8460 to 936! Again, I would like to emphasize that test accuracy remained roughly the same even with nearly a 90% reduction in the size of the feature vector. What this suggests is that just throwing a ton of data to the classifier does not always help.

For the color classification part, I also reduced (resized) resolution from 32x32 to all the way down to a spatial size of 8x8, but later nudged that back up to 16x16. Test-accuracy did not change meaningfully. I discovered that the test accuracy of the classifier was an extremely poor predictor of actual performance on the video pipeline. This was a disappointing discovery, and unfortunately meant that the only way to know how everything was working was to process a video file, which made things more time consuming. I would like to seriously hamper the classifier to get test accuracy down to 70% or lower to see 1. What is needed to hurt classifier performance, and 2. How does poor test-accuracy correlate with performance on the video file.

With the above parameter modifications, the final feature vector length was 1392, which should benefit speed of the classifier and also during the inference phase.

The 5966 car images (later increased to 8792 to include GTI set too), and the 5068 non-car images were fed into the extract_features function, and the output array (13860 x 1392) was normalized using sklearn’s StandardScaler() method. 10% of the data was saved as the test-set, while the randomly shuffled remaining training data was fed into a Support Vector Machine using a linear kernel. I had considered exploring the choice of an rbf kernel or even polynomial, but knowing that I was already in a high-variance/ over-fitting scenario I did not experiment further. I did reduce the C parameter all the way down to 0.0005 from the default value of 1.0 to ensure a smoother and less convoluted decision boundary and help temper the over-fitting.

With these parameters, test accuracy of the trained SVC is 0.9986.
Another important point to highlight is that the size of the model data binary is only 31kB! To me this highlighted the efficiency of traditional machine learning techniques like SVM vs. Neural Nets. Please note that the training code and parameters are in section 5 of the project notebook file.

Sliding-window search and evaluation on test images

I used the sliding window technique for a very short time just to confirm that things were working in the code, but then started using the find_cars function which is significantly more efficient, as it performs the HOG calculation only once on the image/frame to be evaluated. In both the sliding windows and find_cars approach, I started the search at a height value of 400 (in the 720x1280 image) and ended it 256 pixels lower at 656 (the entire 1280 pixels of width were scanned for the above height range). When using the slide_windows function I left the overlap at 50% for both x and y, but did not experiment at all with this.

In the find_cars function, I experimented with the scale function from 0.5 to 2.0, finally settling at 1.5. Lower values result in a lot more positive identifications where there are cars (and also some noise), but result in a significant slowdown for processing the video file. A higher magnitude for scale usually missed identifying cars. So I tried to maximize scale to the largest possible value while having an acceptable level of detection.

A significant challenge I encountered here while experimenting with the scale is that performance on the test images did not correlate well with performance on the video, which makes the process time consuming. The final step was to add the "heatmap" and apply a threshold to reduce false positives.

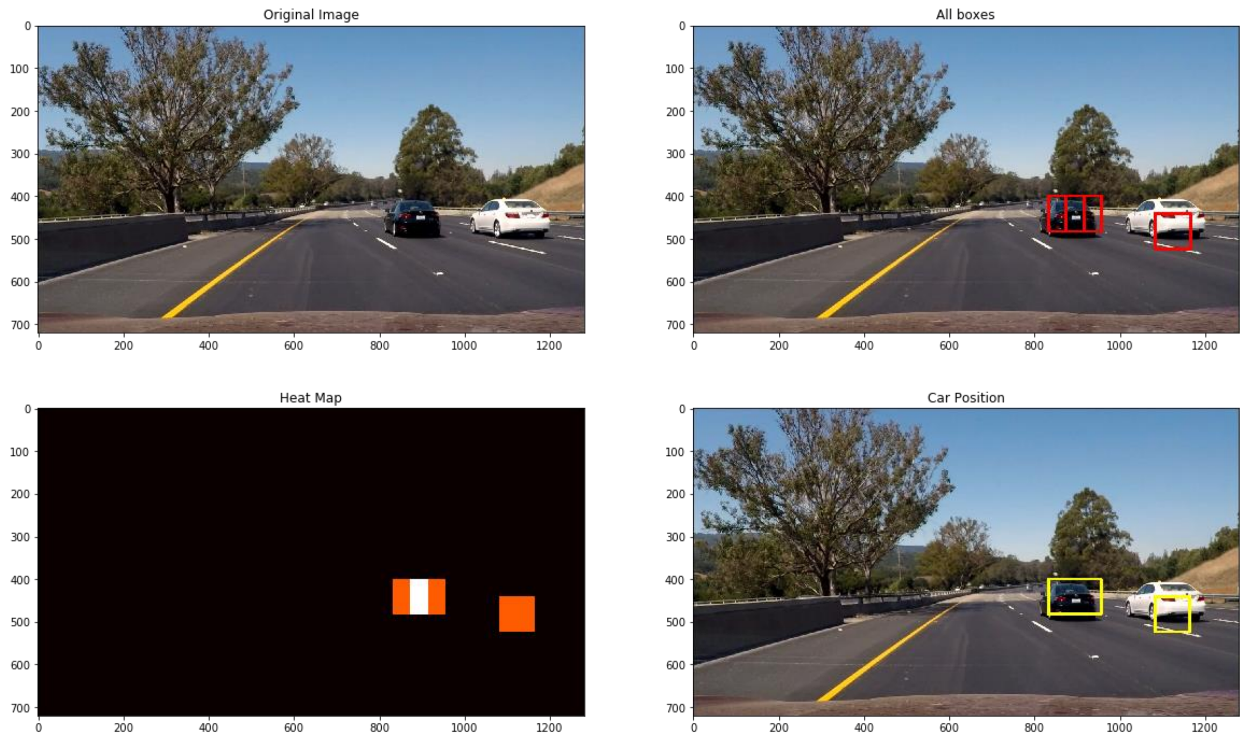

Figure 3. Original test image (top-left) and with the cars detected using the find_cars function; more examples in the project notebook file in section 7. Lower-left image shows the "heatmap", with the combined boxes on the lower right.

The code for this is in section 8 of the project notebook file. With the choice of scale parameter, I did not have any false positives on the test images, so I left the threshold at zero; in fact, I have missed a car in test image 7, so for static images I would reduce the scale parameter. As suggested earlier, this however, is a tradeoff: lower scale values = better/ more “heat” for car detections, but slower video processing.

Video Pipeline

The code for this is in section 9. To minimize jitter and smooth things out a bit, I averaged over the last 5 frames. Performance would be much more smoother if I averaged over more frames, but the speed of processing frames decreased if I averaged over too many more frames. As can be seen, video processing rate was nearly 9 frames/ second on my i7 based PC (if the code ran on the GPU, frame rate should be significantly faster, but I did not have time to explore PyCUDA, etc). The scale parameter and number of frames averaged over was a tradeoff for getting relatively faster processing video speeds.

I am continuing to search for other parameters to tweak to improve performance on the video without reducing processing speed.

Discussion

I think the key problem I faced was that the trained classifier did not generalize well enough for the test video. As a result, I consistently got very high test accuracy, but the only way to evaluate performance was to process the video file. This was a very time consuming way to optimize the pipeline.

Also my video output still has much to be desired—there are times when there is no bounding box on the cars, and there are occasional false positives. This could have been mitigated by using a lower value for scale. I have submitted 2 video files—one is “projectoutFinal.mp4” and has yellow boxes, the other “projectout1.mp4” is prior to smoothing (& using heatmap) and a lower scale value, shows that a car is indeed detected in every frame there is a car.

My other over-arching focus has been on having an efficient pipeline—that potentially could be used in a near-realtime mode; unfortunately I’m facing a tradeoff between processing speed and accuracy.

Finally I’ve seen other students use Neural Net techniques such as YOLO and SSD; I would like to explore that also, but I’m quite impressed with the ability and resource-efficiency of SVMs.