Vehicle Detection

In this project, the goal is to write a software pipeline to detect vehicles in a video, but the main output or product is a detailed writeup of the project.

Writeup / README

1. Provide a Writeup / README that includes all the rubric points and how you addressed each one. You can submit your writeup as markdown or pdf.

You're reading it! All references to the code assume file “pipeline.py” located within the submission.

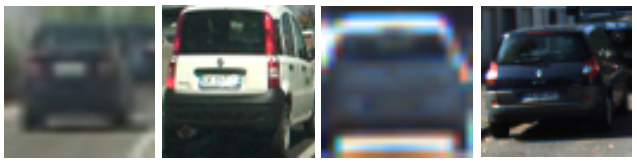
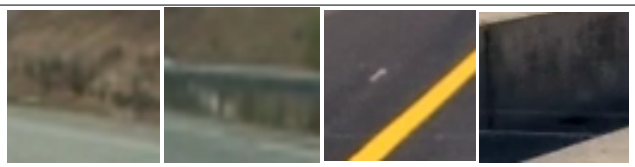
Histogram of Oriented Gradients (HOG)

1. Explain how (and identify where in your code) you extracted HOG features from the training images.

The code for this step is contained function “`get_hog_features()`” on lines [63]. This function has an image as one of its arguments. Its other arguments are parameters of the hog feature extractor. The function returns the feature vector for the image. This snippet of code is different from obtaining HOG features on test images (method “`find_cars()`”, where the step is optimized to share computations between adjacent windows).

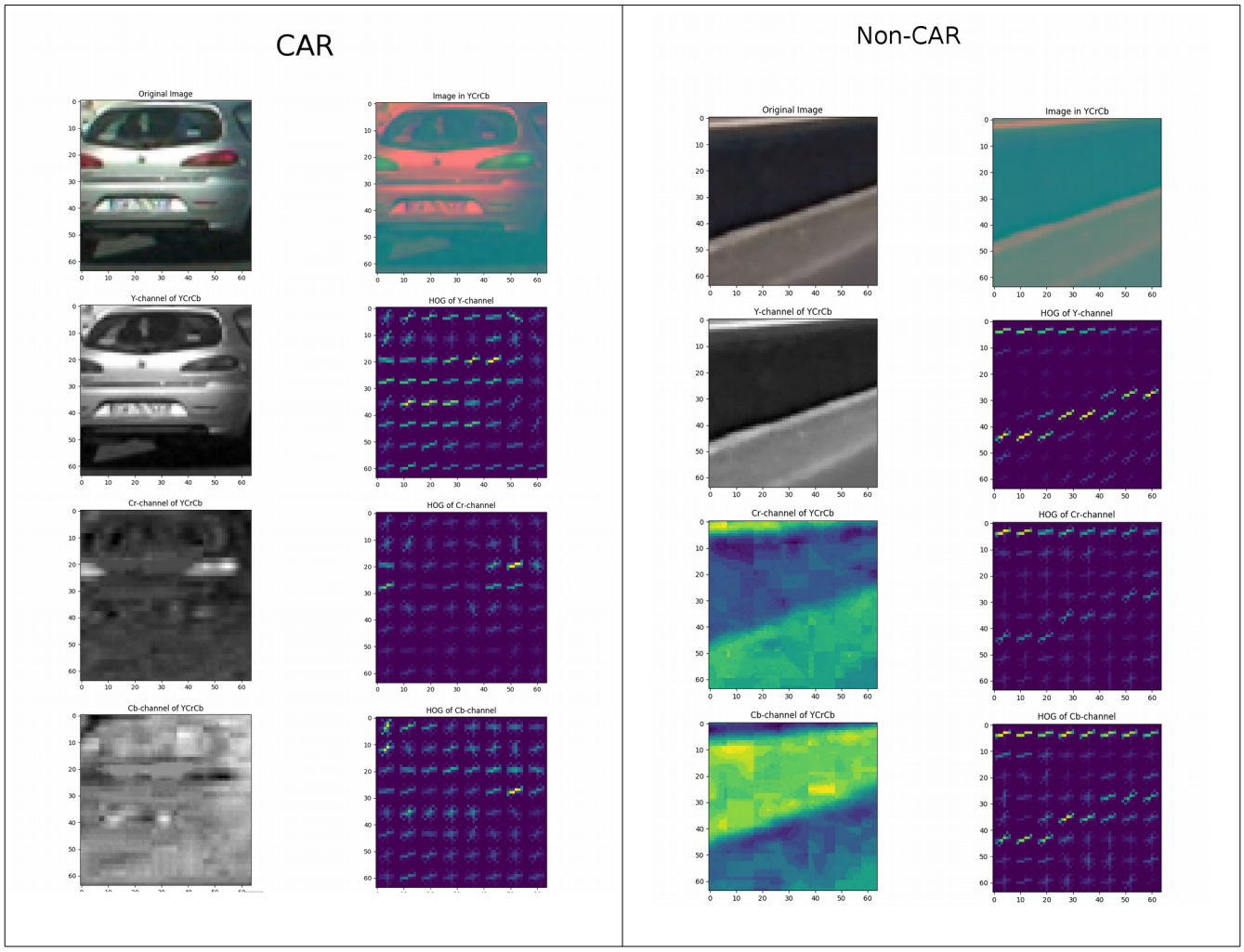
More high-level code specific to training that uses “`get_hog_features()`” function is located in “`extract_features()`” and “`train_classifier()`”. These two functions are constructing the list of images for “car” and “non-car” classes. The images for these classes are located in “train_set” under subdirectories with corresponding names. Before training the list of images are shuffled randomly and split into train and test sets. Training set gets 80% of the data whereas the test set gets 20%.

NOTE: This assignment reports results on a split is used for all further work – 80% of the data is used for training classifier and such classifier is serialized and saved for all further work. I would argue though, that once the performance of the classifier has been estimated and approved it is possible to use all the data, not just 80%, to re-train the classifier and, thus, make it more effective as more data is always better, especially for such high-dimensional vectors.

Car Images	Non-Car Images
	

I explored several color spaces and various parameters of “`skimage.hog()`”. When all color channels as used, the RGB color space was noted to deliver the worst performance between all color spaces. HSV and YCrCb was delivering best results. Between HSV ad YCrCb the latter was showing better visual results on test images and test video sequence. I used all color channels from YCrCb for reasons described in the next section. The table below shows the output of several channels from YCrCb color

shapes as well as its HOG feature visualization for both, “car” and “non-car” classes for sample images.



The example shows images processed with the parameters of HOG corresponding to the final parameter selection for the `YCrCb` : `orientations=9`, `pixels_per_cell=(8, 8)` and `cells_per_block=(2, 2)`. The parameters are set in the beginning of “`pipeline.py`” [28-47].

2. Explain how you settled on your final choice of HOG parameters.

I tried a number of combinations. I found that putting cells in blocks significantly improves classification performance, however, it also reduces the speed of processing as well as increases the feature’s dimensionality. Parameter `cells_per_block=(2, 2)` was a reasonable compromise, that was giving excellent descriptor and was not too slow. I selected “`pixels_per_cell=(8,8)`” as the most common parameter for this type of descriptor motivated by the original paper as well as sklearn recommendation.

3. 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 code for training classifier is located in functions "`get_classifier_scaler()`"[452], "`train_classifier()`**`"[386]`**.

During feature selection one dilemma was observed – when all features are used(including HOG for all channels) the dimensionality of a feature vector become large. Also, the processing time for an image becomes prohibitively high. I had a to make a choice which features to remove from the vector. One thing I observed was that the high-confidence classification requires color information. Either all channels of YCrCb HOG or a combination of Y-channel HOG + color histogram provide enough color information. However, all-channels HOG are faster to process because of the efficient shared computation - when the whole section of the image is processed to get histogram of oriented gradients and then only sub-window is used to combined HOGs for classifier. This shared computation, is significantly faster than trying to extract a color-histogram for each sub-window independently. So I ended up with all color channels of HOG and no color histogram. Neither I used spatial features as extracting those for sub-window is also prohibitively time-expensive and spatial features are unreasonably high-dimensional. The final number dimensions in the vector is 5292.

I used a "`StandardScaler`" from `sklearn.preprocessing` to scale data - make it have zero mean and unit variance.

My choice for the classifier classifier – RBF-kernel SVM. I performed a few experiments with a linear classifier and RBF. Algthough a linear SVM is much faster to train, the inference time of both RBF-SVM and LinearSVM is insignificant compared to the time of feature extraction. The RBF-based SVM, however, is significantly more accurate. My final classification accuracy on the test set of 3552 instances of independent car/non-car images was 99.27%. The training set consisted of 14208 images approximately equally split between car and non-car classes. The classifier, along with scaler is stored in a file classifier.pickle.

Sliding Window Search

1. Describe how (and identify where in your code) you implemented a sliding window search. How did you decide what scales to search and how much to overlap windows?

The search space for windows is limited in the image between height positions of 370 and 656. The code for sliding window search was implemented in a function "`find_cars()`"[290]. The implementation, following Udacity instructions, followed the method of sharing computation between adjacent windows. In this method, the oriented gradients are computed for all 8x8 cells within the search area first, and then combined into individual sub-windows corresponding to region of interest. I used a dense 7/8 overlap for windows to make sure there is always a good overlap between the window and car-targets. I experimented with various settings for overlap, scales, and heat-map threshold and found that the best results for detection accuracy was observed for 3-scale solution [1.2, 1.6, 2.2] with overlap of 56 pixes for 64-pixel windows with a heat-threshold of 2. However, for the purpose of

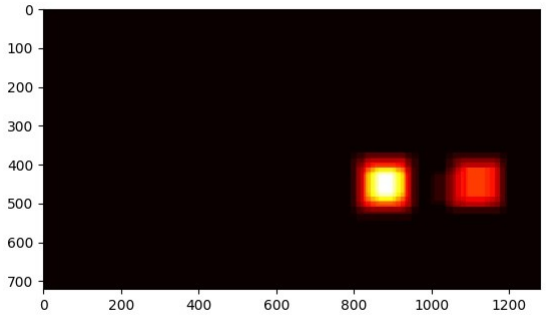
keeping a reasonable running time I ended up in using just the last two scales [1.6, 2.2] as the first scale takes the most time to compute.

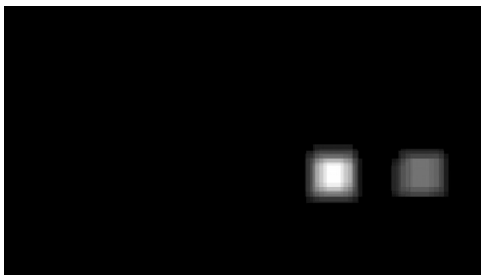
2. Show some examples of test images to demonstrate how your pipeline is working. What did you do to optimize the performance of your classifier?

As mentioned above, I searched on two scales [1.6, 2.2] using YCrCb 3-channel HOG features in the feature vector, which provided a nice result. Here are some example images:

Individual Detections

Thresholded heatmap





Heatmap of detections	
 Labeled heatmap	Final 

More examples from the pipeline are available in the “pipeline_images” directory.

To optimize the speed of processing my implementation used multi-threading implementation (see classes “`ImageJob`”[526], “`ImageDirJobsProducerThread`”[538], “`ImageDirJobsConsumerThread`”[569], “`VideoJobsMoviePyProducerThread`”[668], “`VideoJobsConsumerThread`”[709], “`VideoJobsOpenCVRecorderThread”[733]`, and functions “`process_image_dir()”[593],”process_video()”[844], “process_image()”[863].`

As mentioned above I chose the ‘heavy’ version of SVM - RBF-kernel SVM as the classifier prediction was not taking as much time as the feature extraction itself and I could afford a better classifier.

Video Implementation

1. Provide a link to your final video output. Your pipeline should perform reasonably well on the entire project video (somewhat wobbly or unstable bounding boxes are ok as long as you are identifying the vehicles most of the time with minimal false positives.)

Please see the project_video_tracking.mp4 for the output of the project_video.mp4. Also, see test_video_tracking.mp4 and images in output_images.

I used MoviePy for working with video – see classes . I also have an OpenCV implementation for Video, but the videos present were generated with MoviePy framework. I used 8 threads in parallel to speed-up processing. The parallelization was implemented on a image-frame level (8 images a processed in parallel). I obtained a very good false-positive performance, although bounding boxes around cars were sometimes ‘wobly’. Also, the detection of the vehicle on image edge is not guaranteed.

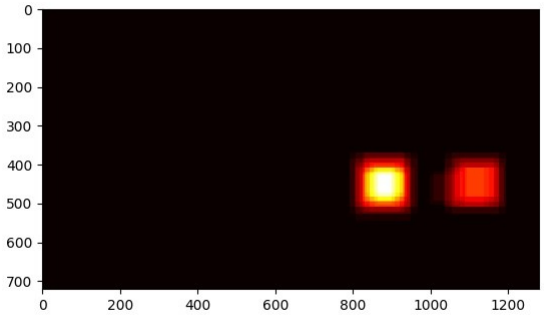
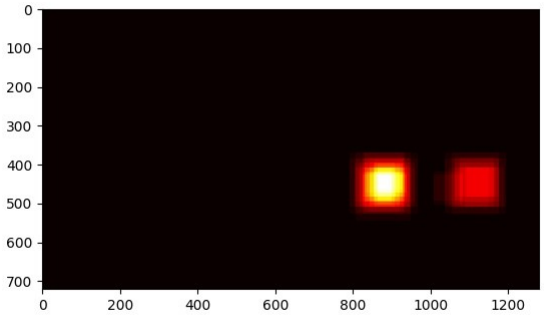
2. Describe how (and identify where in your code) you implemented some kind of filter for false positives and some method for combining overlapping bounding boxes.

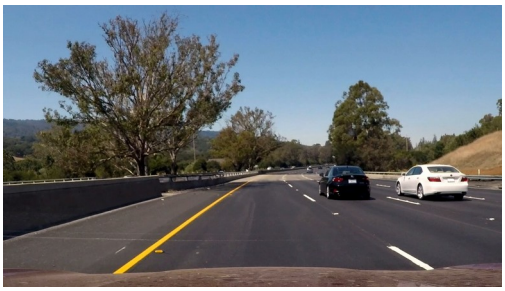
I recorded the positions of positive detections in each frame of the video. From the positive detections I created a heatmap and then thresholded that map to identify vehicle positions. I then used a “connected components” algorithm to combine positive outputs of several overlapping sub-windows. I used “`scipy.ndimage.measurements.label()`” for connected components’ implementation. I added the values of those components into a ‘heatmap’ of confidence and thresholded heatmap by value of 2.

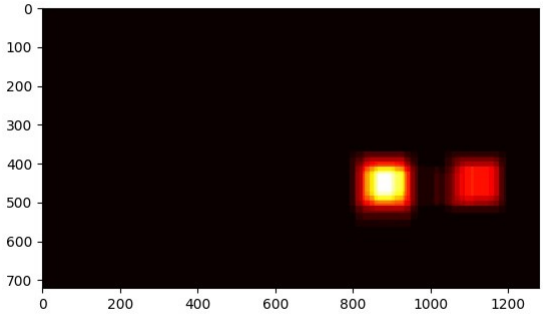
To make results more robust for videos I also included a small filter on spurious detections. This filter, implemented in function '`make_frame()`' of `VideoJobsMoviePyRecorderThread` class, compares detections in two consecutive frames and only "approves" a detection if there is a geometrical intersection between the bounding boxes of this detection and any detection from the previous frame.

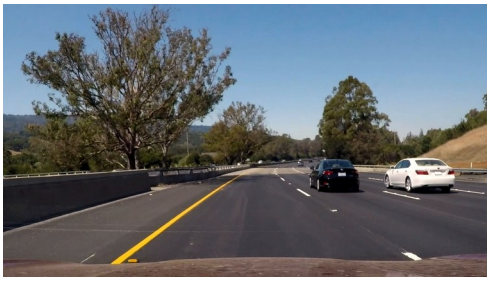
Here's an example result showing the heatmap from a series of frames of video, the result of `scipy.ndimage.measurements.label()` and the bounding boxes then overlaid on the last frame of video:

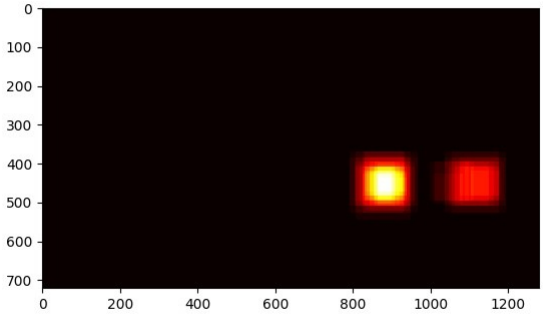
Here are six frames and their corresponding heatmaps:

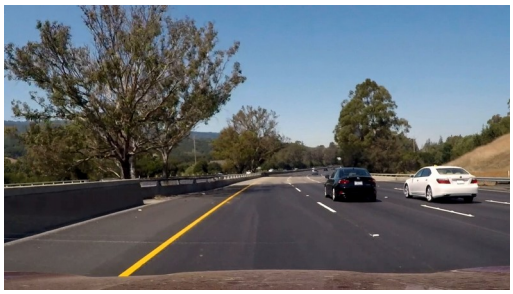
<table>
<tr>
<td>  </td>
<td> 
</td>
</tr>
<tr>
<td>  </td>
<td> 
</td>
</tr>
</table>

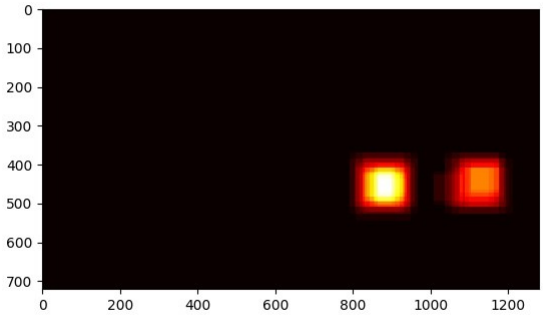


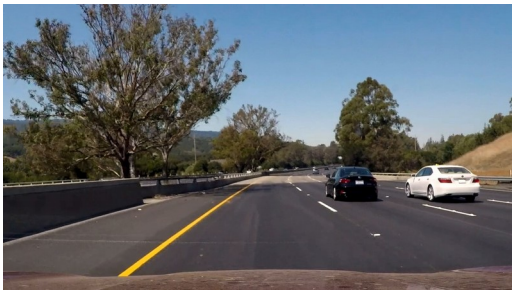

0
100
200
300
400
500
600
700
0
200
400
600
800
1000
1200

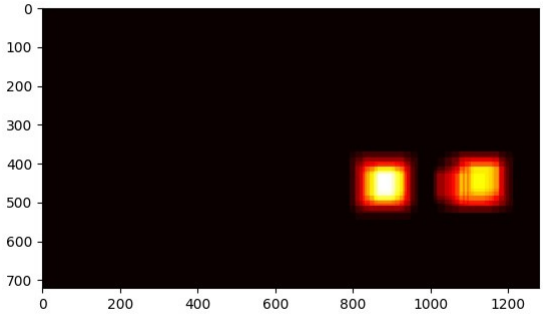



0
100
200
300
400
500
600
700
0
200
400
600
800
1000
1200




0
100
200
300
400
500
600
700
0
200
400
600
800
1000
1200





Here is the output of "`scipy.ndimage.measurements.label()`" on the integrated heatmap from the last two frames:

Here the resulting bounding boxes are drawn onto the last frame in the series:

Discussion

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?

My only significant problem was the speed of processing. My choice of dense overlapped windows and three-channel HOG on two scales was quite expensive to compute. To speed things up I created a parallel implementation to process several consecutive images at the same time . This addressed the problem – I was able to generate a video in several hours, but it is far from real time processing required for self driving cars. With the current choice of parameters – two scales, dense windows, three HOG channels and tracking between the adjacent frames the pipeline is very robust for detection of large vehicles going in the same direction as the ego-car. However, the tracking of the opposite traffic is difficult for several reasons. The opposite direction's car are usually smaller and should really be detected in a different scale. They also require a more sophisticated method to track. The pipeline will also fail on small cars going in ego-direction when the 'receptive field' of the window is much bigger than the cars themselves.