**Akshay Khole**
**SDCND Project 5 Writeup**

**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.**

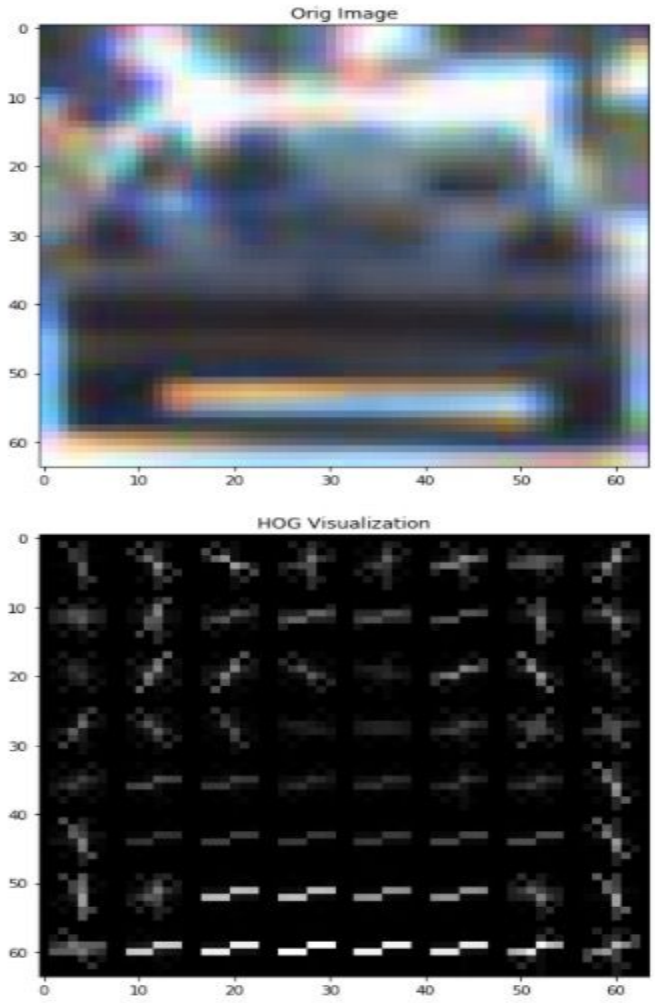
The HOG feature detection is a critical step towards identifying where the car(s) are in the image. The method ***get_hog_features*** does this job. To detect HOG features, we define a virtual grid over the image and extract features from each block of that grid. The way we do that is we measure orientations of gradients in each *cell* and combine individual orientations with surrounding cells in the block defined by us. The measurement is done using histograms of values from individual cells. We pass in the number of bins, cell size, block size and orientation values. We then feed these into a classifier trainer that relates orientations of the block with presence or absence of a car in the image.

I played with a number of values for the HOG parameters and settled with

```
colorspace = 'YUV' # Can be RGB, HSV, LUV, HLS, YUV, YCrCb
orient = 9
pix_per_cell = 8
cell_per_block = 2
hog_channel = "ALL" # Can be 0, 1, 2, or "ALL"
spatial = 32
histbin = 16
```

I tried tuning colorspace and ran the pipeline with HLS, YCrCb but finally YUV seemed to give me best results. The luminescence (Y) component helps us extract distinct shape of the ‘shiny’ parts of the car and UV components help identify color. I changed values of pix_per_cell, cell_per_block but these worked fine with speed and desired accuracy. Spatial and histbin params are both set to 32 and 16 since I am using window sizes in multiples of 32 and these seemed to work fine.

An example of this is:




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

I am training my classifier using HOG features, features extracted from Color Histogram technique and Spatial Histogram. The ***extract_features*** method in my code is responsible for extracting all these features, given an image and hyper parameters.

My car and non car training data run the extract_features method to fetch features of given image. These features are then labeled 0 / 1 based on whether they represent car or non car images. These are then fed into a linear support vector machine to train a linear model.

The training information is displayed in the notebook after training is done. I generally got an accuracy of 98%.

Example output of training:

```
print(round(t2-t, 5), 'Seconds to predict', n_predict,'labels with SVC')
```

```
14.3 Seconds to extract HOG features...
Using: 9 orientations 8 pixels per cell and 2 cells per block
Feature vector length: 8412
0.67 Seconds to train SVC...
Test Accuracy of SVC =  0.9892
My SVC predicts:  [ 1.  1.  0.  0.  0.  1.  1.  0.  1.  0.]
For these 10 labels:  [ 1.  1.  0.  0.  0.  1.  1.  0.  1.  0.]
0.00318 Seconds to predict 10 labels with SVC
```

**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 ***slide_window*** and ***search_windows*** methods perform the window generation and searching within the window. slide_window creates a number of windows out of the image (or it's subset) based on the parameters input. These windows are then used as guides by the search_windows method to look for matches against features within those windows. The windows where a car is detected, is then passed through a heat map generation function that gives the ultimate coordinates for drawing the windows that identify positions of the cars in the image.

We are using 3 sliding window sizes to account for apparent size of detected vehicle based on its position in the image.

The 3 sizes:

1. 64 x 64
2. 80 x 80
3. 96 x 96

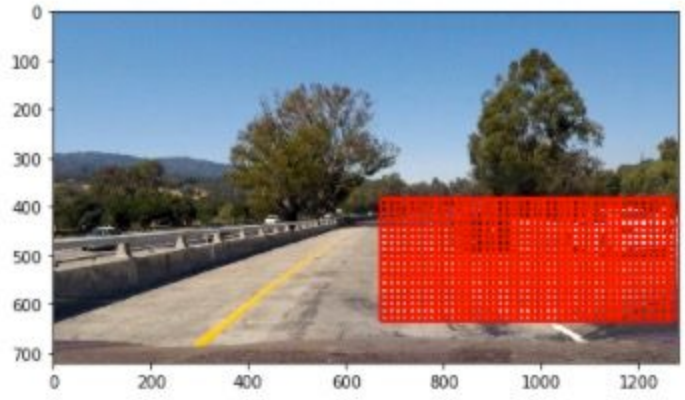
These window sizes are slid over the image with an overlap of 85%. I've tried different sizes and different number of sizes but this one worked best and fastest for me. I am also restricting the sliding windows over these X and Y ranges:

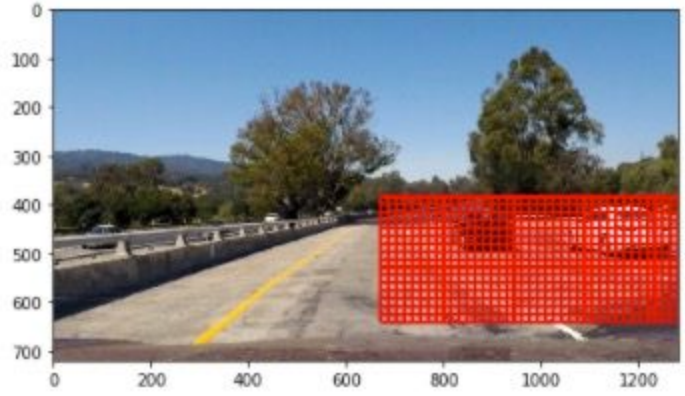
y_start_stop = [384, 640]
x_start_stop = [672, None]

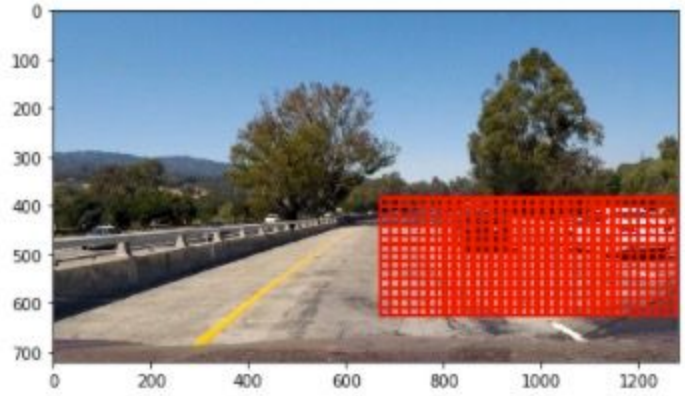
This helps speed up the process a bit.

The working of sliding window is visualized as follows:

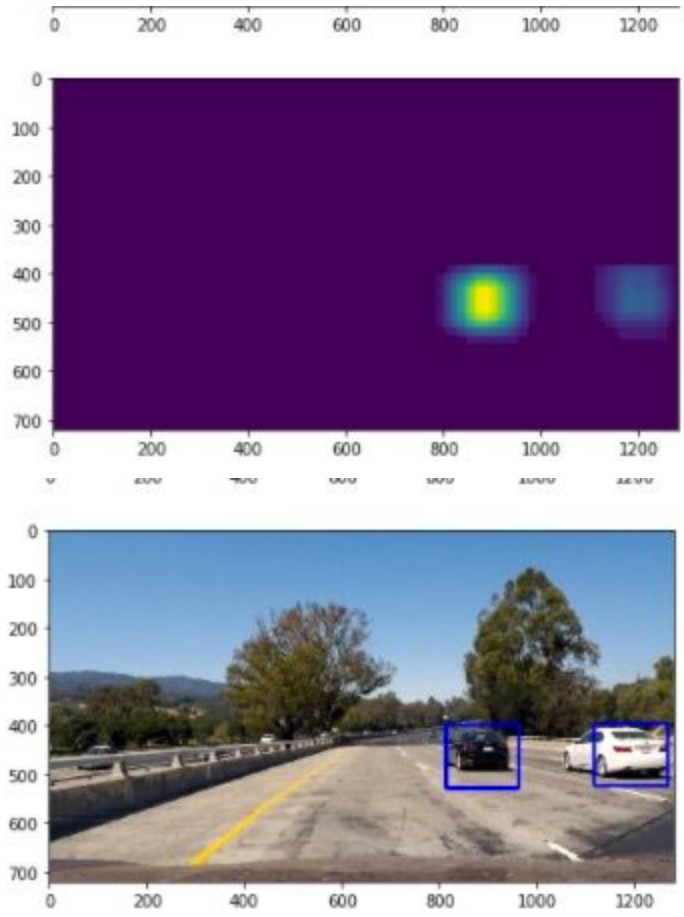
Running pipeline on image
Running pipeline on image



Example Heatmap and detection of vehicle:




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

The pipeline has following steps:

1. Read in car and non car images
2. Extract features of car and non car images
3. To extract features we use the combination of HOG sampling, spatial and color histograms.
4. We label these extracted features for car and non car images.
5. We normalize the extracted features.
6. We define a Linear classifier that uses Support Vectors for classifications
7. We train the classifier on our normalized feature data and label set
8. After training, we start the process of predicting and drawing bounding boxes around detected cars
9. To detect cars, for every input image, we extract hog, color histogram and spatial features.

10. We then run a ‘Sliding window’ over the image. The sliding window takes different sizes over a few iterations. The different sizes are needed since the same cars may vary in apparent size as the car moves.
11. Using the classifier over each window, we predict whether the area has a car or not
12. We may get multiple predictions for the same area so we apply a ‘Heat Map’ to find the general ‘hot’ area where a vehicle is detected
13. We draw bounding box around the ‘hot’ area.

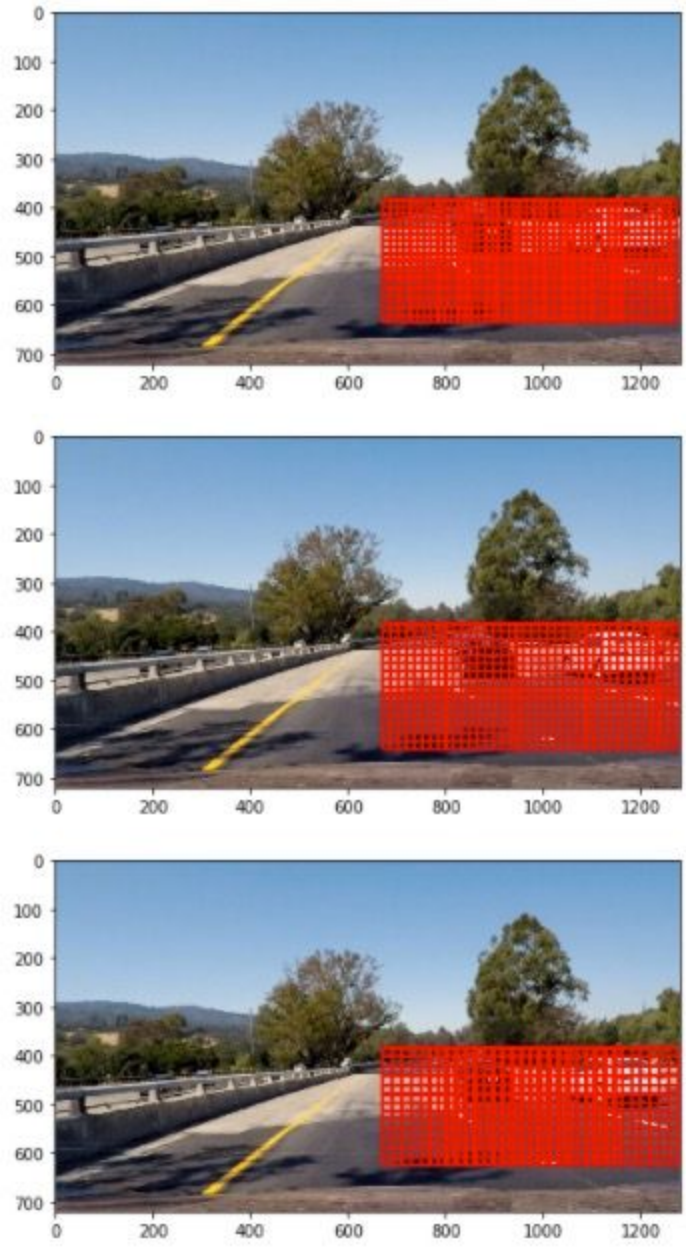
The demonstration of pipeline is visualized in the ‘notebook_outputs’ directory of the project. The notebook shows how we extract features, details of classifier training and visualizing of the sliding window and vehicle detection process.

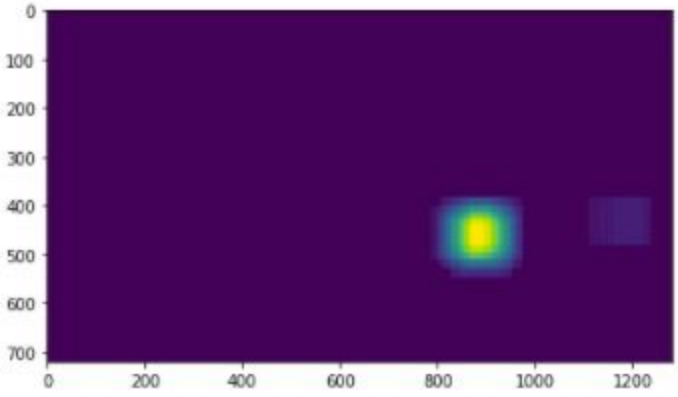
To tune the performance of the classifier we are applying a few steps.

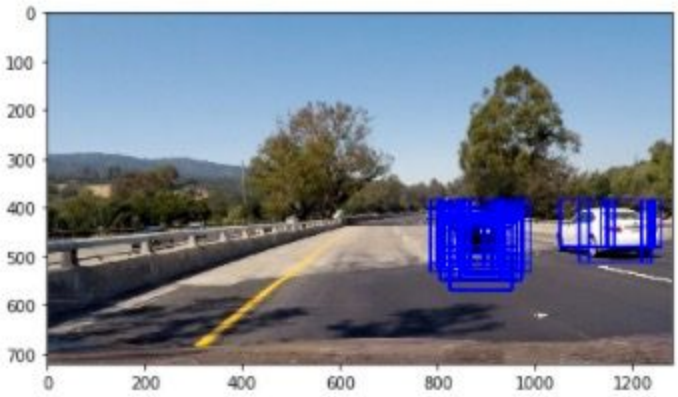
1. Search within a targeted area of the image. This means we will not look at the entire image, only the part where other vehicles can possibly be
2. We tune the sizes of the window so that we make minimum iterations and still are able to detect varying sizes of apparent vehicle.
3. We also tune the window overlap value so that we can maintain good accuracy of detection without seeing every pixel WRT to every other pixel!

There are several other ways we can enhance performance. I want to apply HOG subsampling, but tuning the subsampling to even detecting was taking too much time and I decided to tune on a slower, simpler pipeline to get the output quickly. I will extend the project to apply HOG subsampling for other challenge videos.

Visualization of working pipeline:




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

**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.)**

The project video is available in the github repository at
https://github.com/chmod600/carnd1-project5/blob/master/p5-output.mp4

The pipeline seems to work reasonably well. I have tried to remove maximum false positives and keep the detection steady.

**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.**

With my implementation, I have been able to generate a video that detects the vehicles 99% of the time. But with such an implementation, come false positives. To filter these false positives out, I have used an approach of eliminating false positive windows based on their centroid position and comparing their centroid position with the centroid of vehicles in the last N frames.

The method ***get_qualifying_bboxes*** performs this job along with its helper methods. This helps weed out most of the false positives.

Another approach to minimize false positives is using average of last ***n*** heat maps. This is done in the ***add_heat*** method***.*** This method reduces likelihood of no detections or very close duplicate detections. One parameter used here is last H heat maps used. I have tried with H = 30 and yields results that aren't too bad.

**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?**

One problem I faced was while implemented the sub-sampling approach for a faster pipeline. I tried to tune parameters but it wouldn't give me any detections. I wanted to progress with the pipeline at that time so I dropped and went with the slower, but working approach. But I plan to implement the sub-sampling after getting acceptable results on the project video because it takes too much time to generate an output to even debug.

The pipeline is a bit biased for the project video. To increase the speed, I am not applying sliding window to detect cars directly in front of our car. This will fail if a car comes directly in front. But I have kept it this way for purposes of this project and plan to remove this condition and add the sub-sampling approach for the challenge video. I plan to remove all these issues once the base case is complete. It's an incredibly amazing learning experience working on this project.