##Writeup Template ###You can use this file as a template for your writeup if you want to submit it as a markdown file, but feel free to use some other method and submit a pdf if you prefer.

**Vehicle Detection Project**

The goals / steps of this project are the following:

* Perform a Histogram of Oriented Gradients (HOG) feature extraction on a labeled training set of images and train a classifier Linear SVM classifier
* Optionally, you can also apply a color transform and append binned color features, as well as histograms of color, to your HOG feature vector.
* Note: for those first two steps don't forget to normalize your features and randomize a selection for training and testing.
* Implement a sliding-window technique and use your trained classifier to search for vehicles in images.
* Run your pipeline on a video stream (start with the test\_video.mp4 and later implement on full project\_video.mp4) and create a heat map of recurring detections frame by frame to reject outliers and follow detected vehicles.
* Estimate a bounding box for vehicles detected.

[**Rubric**](https://review.udacity.com/#!/rubrics/513/view)**Points**

###Here I will consider the rubric points individually and describe how I addressed each point in my implementation.

###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. [Here](https://github.com/udacity/CarND-Vehicle-Detection/blob/master/writeup_template.md) is a template writeup for this project you can use as a guide and a starting point.

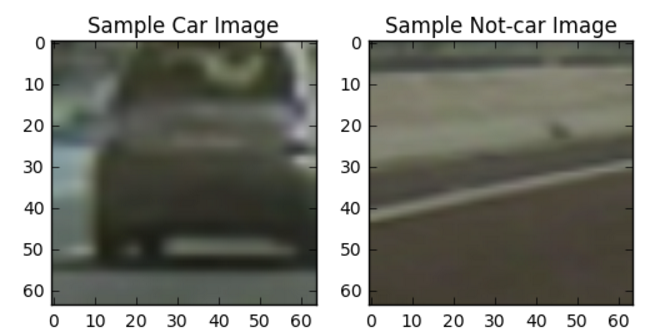
You're reading it!

**###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 in the cells # 3 and #4 of the IPython notebook.

I started by reading in all the vehicle and non-vehicle images. Here is an example of one of each of the vehicle and non-vehicle images:



The dataset contained 8792 cars and 8968 non-cars images. The code for counting car and not-car images is in code cell 2 in the ipython notebook.

For HOG features, I then explored different skimage.hog () parameters (orientations, pixels\_per\_cell, and cells\_per\_block).

The scikit-image hog () function takes in a single color channel or grayscaled image as input, as well as various parameters. The number of orientations is specified as an integer, and represents the number of orientation bins for the histogram. The pixels\_per\_cell parameter specifies the cell size over which each gradient histogram is computed. The cells\_per\_block parameter specifies the local area over which the histogram counts in a given cell will be normalized.

Here are the HOG parameters I used and I also converted the image to grayscale first –

orient = 9

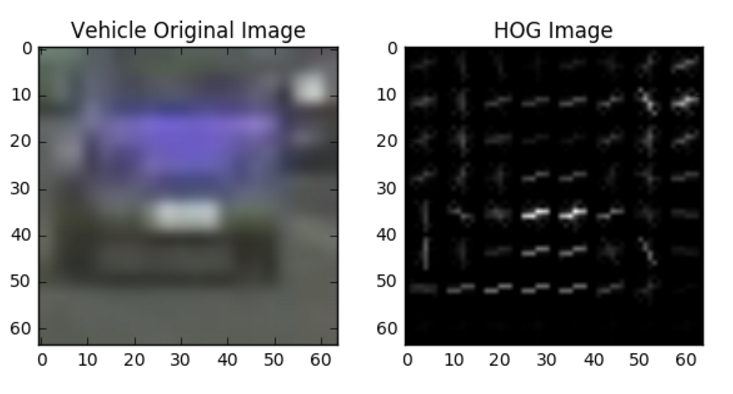
pix\_per\_cell = 8

cell\_per\_block = 2

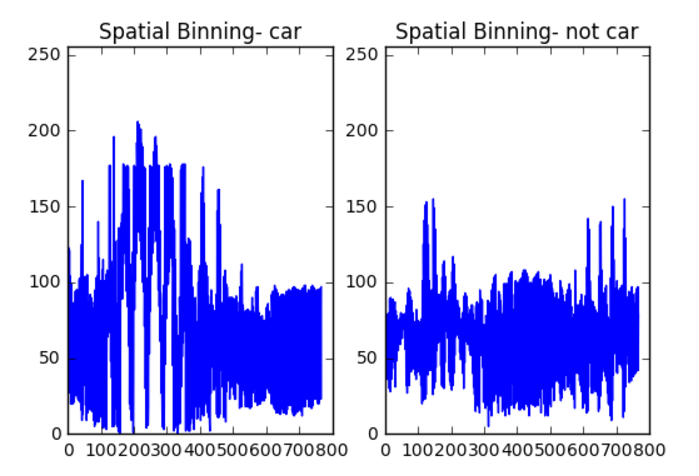
hog\_channel = 0

The code for extracting HOG features is in the cells 3 and 4 of the iPython notebook.

Here is an example of one image HOG features:



I also explored spatial binning (I reduced image size to 16,16) and used color\_space='HSV'. Here is a plot showing car and not car spatially binned features. The code for this step is contained in cell # 5 of the iPython notebook.



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

I tried various combinations of parameters and finally settled on these parameters:

color\_space = 'YCrCb'

orient = 9

pix\_per\_cell = 16

cell\_per\_block = 1

hog\_channel = 'ALL'

I started by using some of the same parameters that were used in the lectures and spent some time narrowing down the

parameters. For example I tried pix\_per\_cell 8 and 16 and settled for 16 since it decreased computation time. Color\_space = 'YCrCb’ resulted in better performance than RGB.

The code for this step is in cells 8 and 9 in the iPython notebook.

**####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).**

I followed these steps in training the classifier –

1. Extract features from the list of images
2. Stack and scale the feature vectors.

Create array stack of feature vectors

Fit per column scalar

Apply scalar to X. Now, scaled\_X contains the normalized feature vectors.

1. Shuffle and split the data into training and testing sets.
2. Use a linear SVM to define and train a classifier.
3. Check the accuracy of the classifier on the test data set.

The code for these steps is contained in cell 9 of the iPython notebook.

**###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?**

Define windows to search using a helper function slide\_window.

🡺Restricted search space to lower half of the image (altered value of variable y\_start\_stop).

Implement sliding window search using function search\_windows.

For each window,

🡺extract features for that window,

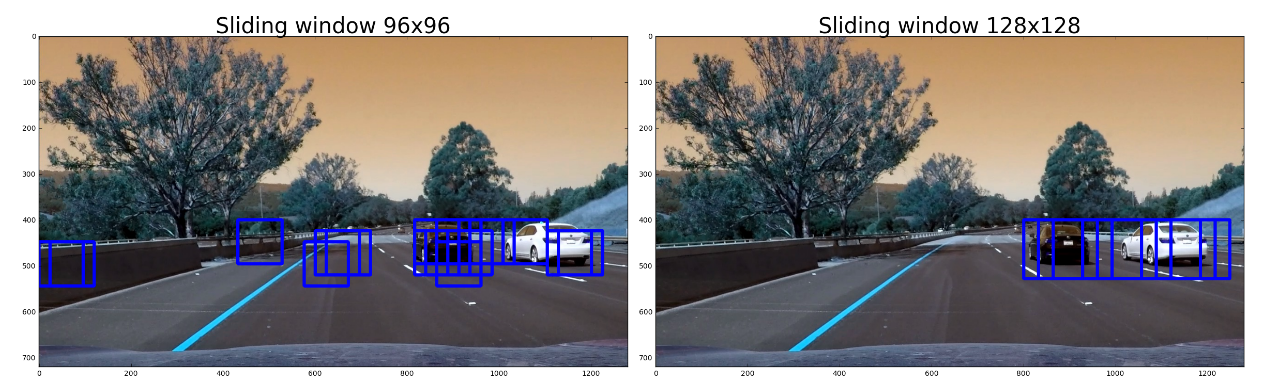
🡺scale extracted features to be fed to the classifier,

🡺predict whether the window contains a car using our trained Linear SVM classifier,

🡺and save the window if the classifier predicts there is a car in that window.

The code for this implementation is in cells 10, 11 and 12 in the iPython notebook.

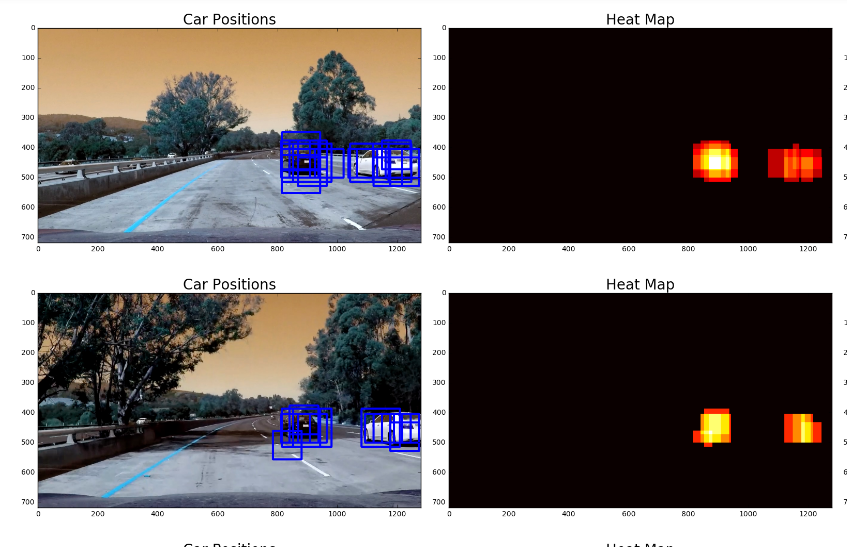
Some examples of sliding windows using window size (96,96) and (128,128) on test image test6.jpg:



For the final model I chose window sizes [(96,96),(128,128)] and y\_start\_stop of [[350, 600], [350, None]]. I also picked x\_start\_stop=[700, None] to start at the right lanes. I picked an overlap of 0.7. The code is in cells 10, 11, 14 and 15 of the iPython notebook.

**####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?**

Here are some examples of test images from my classifier. There are multiple detections in these cases. To smooth out multiple detections and false positives, I used the code for generating heatmaps using code similar to the lectures and using a threshold of 2. The code for this step is in cell 15 of the iPython notebook.



**Video Implementation**

**####1. Provide a link to your final video output. Your pipeline should perform reasonably well on the entire project video**

The video is included with the list of files submitted with this project - video\_v2.mp4

**####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. I combined detection for 20 frames or using the number of frames available if there have been fewer than 20 frames before the current frame. From the positive detections, I created a heatmap and then thresholded that map to identify vehicle positions. I found the best performance was with threshold parameter = 22. I then used scipy.ndimage.measurements.label() to identify individual blobs in the heatmap. I then assumed each blob corresponded to a vehicle. I constructed bounding boxes to cover the area of each blob detected.

The code for this is in cells 16 and 17 in the iPython notebook.

###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?**

One of my concerns with this project is that for this project, I relied heavily on the code provided in the lectures. I can only become an expert in this if I learn to generate my own code for any picture or video. Another concern is that this code relies heavily on hardcoding parameters for window size, HOG parameters, thresholds etc. This is a very good learning experience, however, this approach will not be able to generalize for a wider range of situations. Also, when I look at the video file, while it works and detects some cars, it does detect false positives in some frames and almost does not detect the car in some other frames. Some more work could be required in tuning the number of frames over which windows are added. Another issue is that this pipeline will probably not work in very heavy traffic situations when there are many cars.

This was another fun project and was amazed when the cars were detected by the pipeline. Thanks to the excellent code in the Udactity lectures!

**Resources used:**

1. Udacity lectures – extremely helpful!
2. <https://github.com/>
3. <http://stackoverflow.com/>
4. Udacity forums