Vehicle Detection Project

The goals / steps of this project are the following:

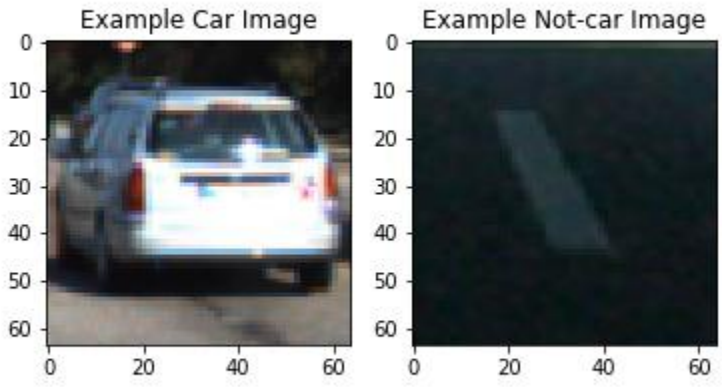
- Perform a Histogram of Oriented Gradients (HOG) feature extraction on a labeled training set of images and train a classifier Linear SVM classifier
- Optionally, you can also apply a color transform and append binned color features, as well as histograms of color, to your HOG feature vector.
- Note: 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.

Histogram of Oriented Gradients (HOG)

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

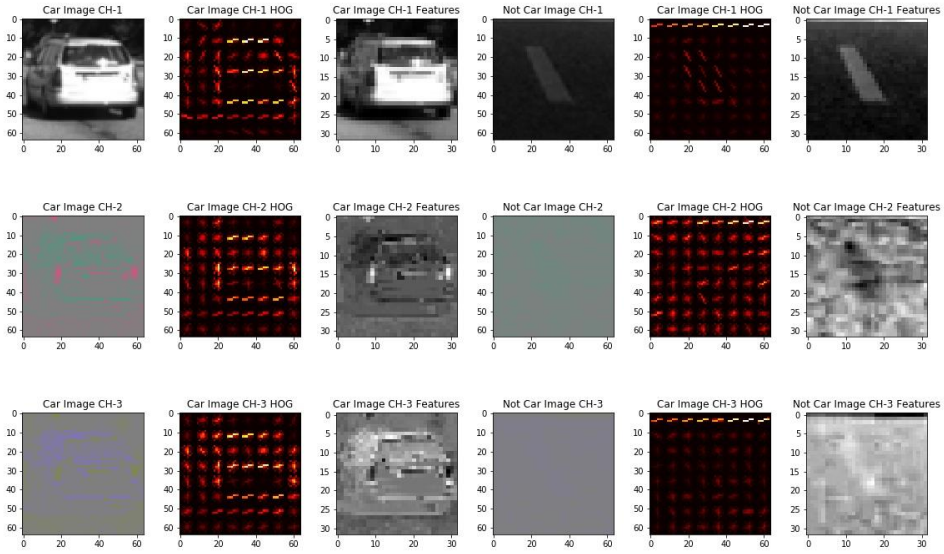
The code for this step is contained in the first and third code cells of the IPython notebook (or in lines 41 through 56 and 186 through 199 of the file called `train_svc.py` and lines 52 through 103 of the file called `lesson_functions.py`).

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` classes:



I then explored different color spaces and different `skimage.hog()` parameters (`orientations`, `pixels_per_cell`, and `cells_per_block`). I grabbed random images from each of the two classes and displayed them to get a feel for what the `skimage.hog()` and spatial feature output looks like.

Here is an example using the `YCrCb` color space and HOG parameters of `orientations=9`, `pixels_per_cell=(8, 8)` and `cells_per_block=(2, 2)` along with spatial features of size (32,32):



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

I tried various combinations of parameters and color spaces and finally settled for following parameters.

HOG parameters

- `orientations=9`
- `pixels_per_cell=(8, 8)`
- `cells_per_block=(2, 2)`
- `hog_channel='ALL'`

Spatial features

- `spatial_size=(32, 32)`

I did not used color histogram features at the end since it only adds a slight increase in accuracy at the expense of some computation cost in feature calculations. In contrast HOG + spatial features were enough to achieve decent accuracy (~99%)

Log:

2018-04-11

- Color spatial and histogram classification
  YCrCb, Using spatial binning of: (32, 32) and 32 histogram bins
  4000 samples
  LinearSVC with default parameters.
  Validation accuracy: 0.9462

2018-04-12

- HOG Classify
  YCrCb, ALL channels Using: 9 orientations 8 pixels per cell and 2 cells per block
  4000 samples
  LinearSVC with default parameters.
  Validation accuracy: 0.9812
- HOG + spatial features
  Color (histogram) features omitted
  Parameter tuning using GridSearchCV (best_params_:C=1.0)
  Model trained and saved using split from entire data set. (Accuracy = 0.9904)
  --
- DecisionTreeClassifier
  HOG + spatial + color histogram features
  Parameter tuning using GridSearchCV ('min_samples_split': 5, 'criterion': 'entropy')
  4000 samples
  Validation accuracy: 0.9275

2018-05-27

- Parameter fine tuning for LinearSVC (C=0.1)
  Validation accuracy: 0.9907

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).

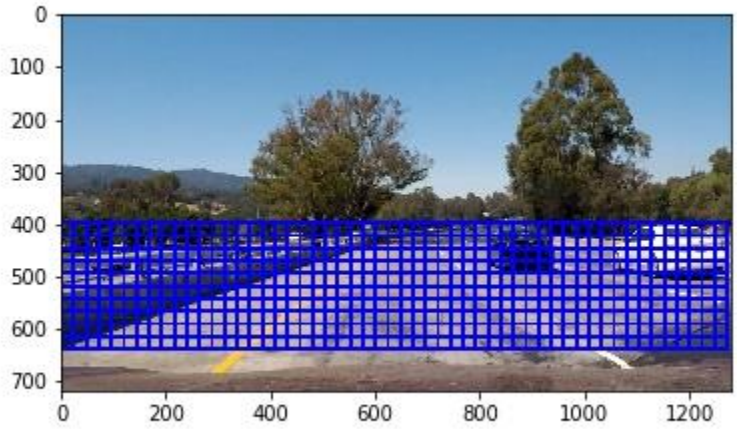
I trained a linear SVM using HOG features and spatial features as selected above. I used GridSearchCV to fine tune parameters of the classifier (`C=0.1`) The code for this step is contained in the sixth and seventh code cells of the IPython notebook (or in lines 134 through 235 of the file called `train_svc.py`).

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 code for this step is contained in the eighth code cell of the IPython notebook (or in lines 12 through 83 of the file called `hog_subsample.py`).

I decided to search evenly distributed window positions at selected scales over selected regions of the image using HOG sub sampling and came up with this:



Here are the selected regions:

- `x_start_stop=[None, None] #entire width of the image`
- `y_start_stop=[400,656]`
- `scale=1.5 # (96,96) window size`
- `cells_per_step=2 #Overlap = (0.75,0.75)`

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?

Ultimately, I searched on a single scale using YCrCb 3-channel HOG features plus spatially binned color features in the feature vector, which provided a nice result. Here are some example images:

To optimize the performance of classifier I used a scale of 1.5 which results search windows of size (96,96) which happened to be roughly the average size of vehicles at medium distance. Also I restricted the search area to `y_start_stop=[400,656]` region and used HOG sub-sampling to speed up the processing.

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.)
Here's a link to my video result

I have also combined the vehicle detection pipeline with the lane finding implementation from the last project. Here's a link to my video result with lane tracking

Furthermore, I have implemented this on my own video I've recorded myself. Here's a link to my own video result

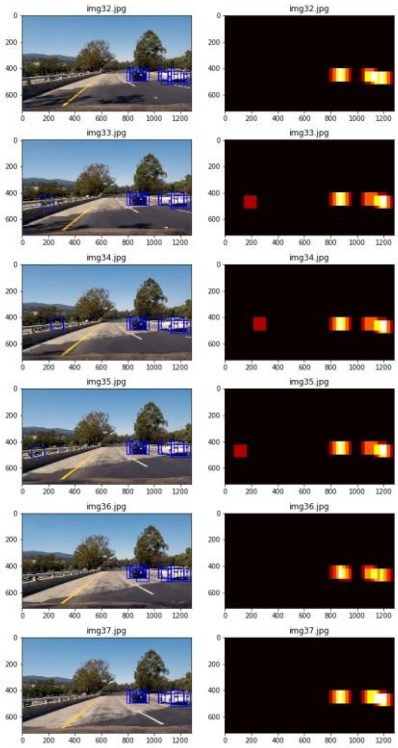
2. Describe how (and identify where in your code) you implemented some kind of filter for false positives and some method for combining overlapping bounding boxes.

I recorded the positions of positive detections in each frame of the video. From the positive detections of recent frames, I created a heatmap and then applied threshold to that map to identify vehicle positions. 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.

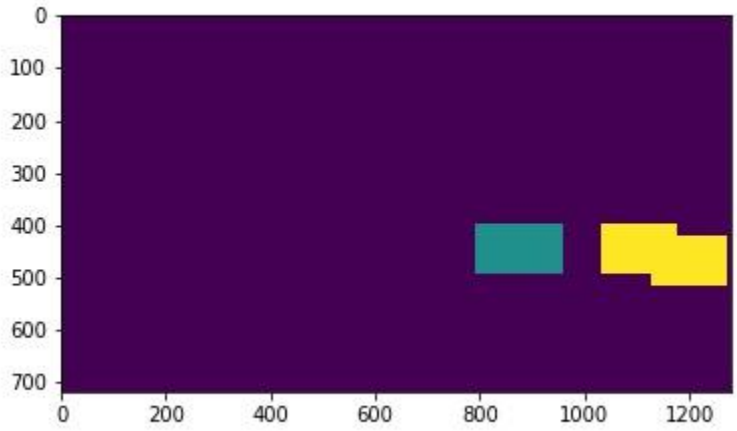
The code for this step is contained in the `find_vehicles(self, img)` method in twelfth code cell of the IPython notebook (or in lines 88 through 96 of the file called `vehicle_tracker.py`).

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

Here are six frames and their corresponding heatmaps:



Here is the output of **`scipy.ndimage.measurements.label()`** on the integrated heatmap from all six 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?

Here I have used HOG and spatial features from all three channels of YCrCb color space. YCrCb color space seems to discern vehicles vs non-vehicles better compared to other color spaces. I also used LinearSVC for the speed and accuracy performance. (While a rbf kernel would provide

better accuracy, it comes with a cost of speed of processing. It's also hard to train against large dataset of over 10000 samples as it's fit time complexity is more than quadratic)

It is observed that while parameters and training set used works fairly well for the unmodified project video, accuracy of the pipeline is greatly reduced for different videos. It's interesting that even a simple operation like distortion correction itself resulted in large number of false positives and required higher thresholding level for project video. To overcome these situation it may require more hard negative mining (Training classifier using more false positives)

It's also required careful tuning of the pipeline where it requires a very delicate balance between accuracy versus speed. It would be interesting to explore deep learning approaches and compare the accuracy vs speed against the computer vision-based detection algorithms.