Udacity Self-driving car P5 Write-Up

This write-up is for P5: Vehicle Detection and Tracking in Udacity self-driving car nano-degree. Codes related to feature extraction and classification are packaged into object Extractor() in extractor.py. Codes related to video processing pipeline are packaged in object Filter() in filters.py.

To train the car/non-car classifier, run the following:
%python test_train.py

The training procedures are all in function train(self, model_file, scaler_file) in Extractor()

To process a video using the pipeline provided in Filter(), run the following:
%python main.py

The pipeline procedures are all in function pipepine(self,image) in Filter().

Rubric point#1:
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.

Answer:
The function related to feature extraction is extract_features_batch(self, image_paths) in Filter(), which will extract features for all images in the labeled data. Parameters related to hog feature extractions are:

```
self.hog_cspace = 'HLS'
self.hog  orient = 9
self.hog_pix_per_cell = 8
self.hog_cell_per_block = 3
self.hog_hog_channel = 1
```

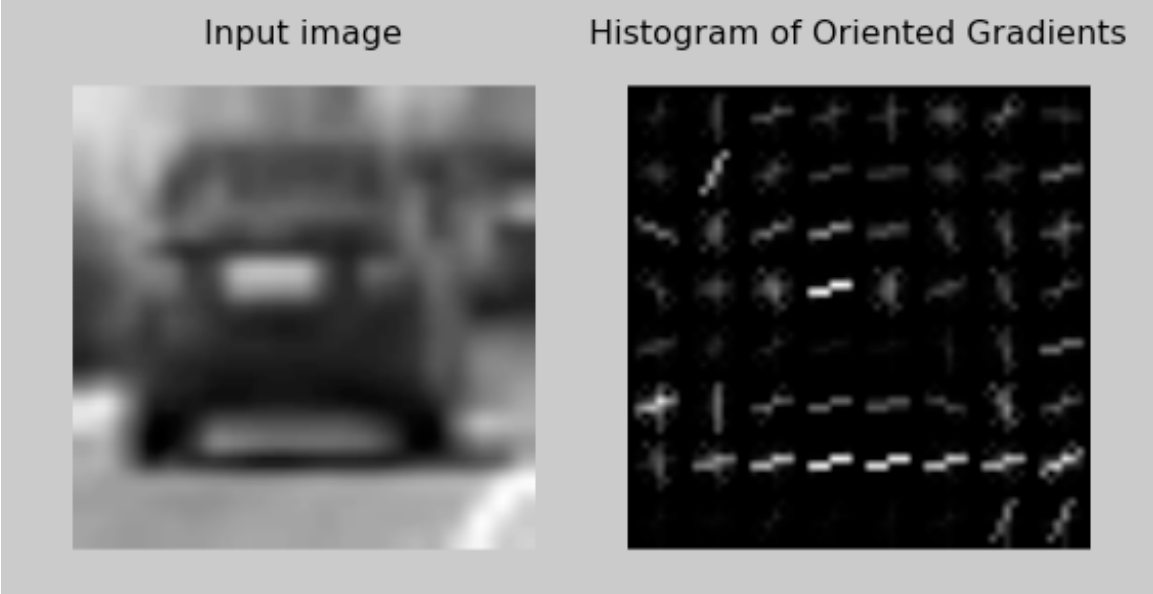
Intuitively, the contours and shapes are kept in the L channel of HLS image, whereas the other two channels will have somewhat blurred. Since HOG is interested in gradients, I choose the HLS space and L channel as the input to HOG feature extractor. The other parameters are chosen so that the number of features are not too small (concerning high bias) and too big (concerning training time).

The function to extract features from a single image is extract_features_one_image(self, image) in Filter(), in which there are following steps:

(1) Convert the image to specified color space (self.hog_cspace)
(2) Extract hog features for the specified color channel (self.hog_hog_channel), using specified hog extraction parameter (self.hog_orient, self.hog_pix_per_cell, self.hog_cell_per_block).

(3) Use the original image (in RBG space), resize it to (16*16), and extract special binning features.
(4) Use original image and extract color histogram features, with number of bins set to 32.
(5) Concatenate all above feature and return the concatenated features.

Number of features for a single image are 3780. Here is an example of hog feature extractions:



Rubric point#2:
Describe how (and identify where in your code) you trained a classifier using your selected HOG features (and color features if you used them).

Answer:
The first step is to normalize all the features extracted in previous step. The function that does this job is normalize(self, car_features, noncar_features) in Extractor().

Secondly, all the normalized data are splite into training set and validation set:

```
X_train, X_test, y_train, y_test = train_test_split(scaled_X, y, test_size=0.2, random_state=rand_state)
```

And thirdly, the classifier is LinearSVC(C=2) wrapped by CalibratedClassifierCV(), which enable us to estimate a confidence for a particular prediction that is used later to reject false positives. The C parameter is set empirically so that we don't see severe overfitting that degrades validation accuracy.

```
svc = LinearSVC(C=2)
svc = CalibratedClassifierCV(svc)
```

Fourth, hard negative mining is used to further reject false positive. The procedure is to train the classifier with the above classifier. Then run the video processing

pipeline and manually identify the false positives. The identified false positives are then added to the training data. This will have non-trivial improvement in false-positive rejection as I have seen from experiments.

After the hard negative samples are added to the training set, we have 10640 car samples, and 10677 non-car samples.

Finally we train the classifier. On my 2009 macbook pro it takes 180secs to train the classifier. Final test accuracy is 97.51%; and final precision(true positive/(true positive + false postive)) is 97.52%.

The following plots show 12 classification examples, concerning true positive, true negative, false positive and false negative.

Rubric point#3:
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?

Answer:
Three levels of sliding windows are implemented. They are defined in Filter():

```
self.sliding_box_param_level = [
    {'size': 100, 'x_step': 0.3 * 100, 'y_step': 0.3 * 100, 'portion': 0.5},
```

```
    {'size': 140, 'x_step': 0.3*140, 'y_step': 0.3*140, 'portion': 1},
    {'size': 180, 'x_step': 0.3*180, 'y_step': 0.3*180, 'portion': 1},
]
```

Only the lower half of the video image is scanned.

The "size" is the sliding boxes' size. The "x_step" and "y_step" controls how fine the sliding process is. Obviously, the smaller the step the finer the process. However, box size is not set too small, otherwise the box lost information regarding the shape of the car.

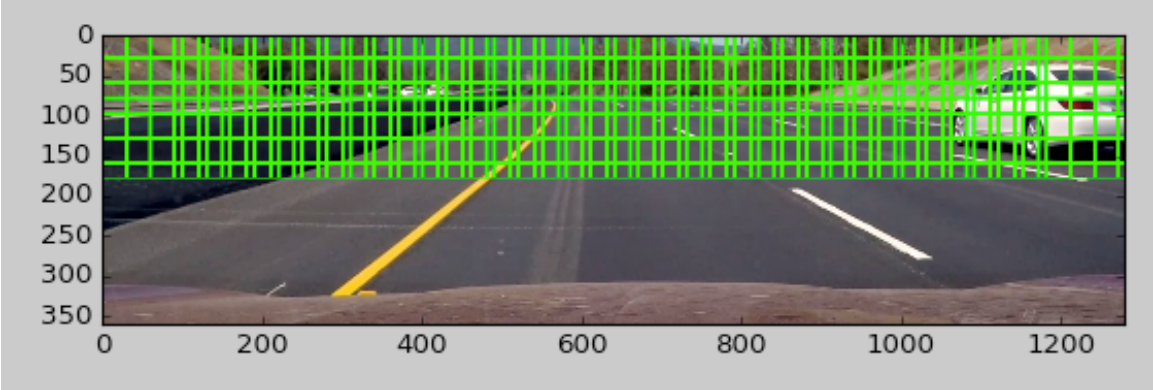
The "portion" parameter in the above definition means how much in the lower half of the video image is scanned. For example, "portion"=0.5 means the upper 50% of the lower half of the image is scanned. This help reduce the number of iterations in scanning process.

All these parameter are found empirically, so that the processing pipeline is not horribly slow but maintain enough detection accuracy.

Here are all the bounding boxes' positions. The example only shows the lower half of the input image from video.

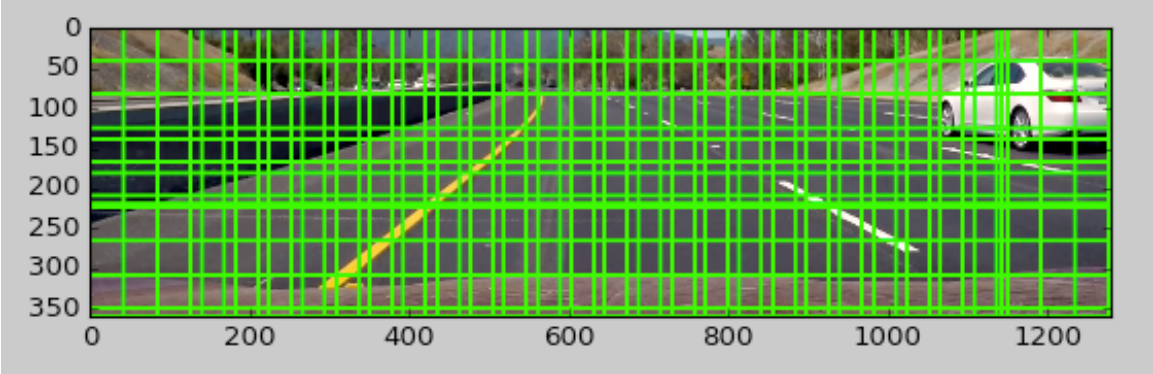
Level 1:

```
{'size': 100, 'x_step': 0.3 * 100, 'y_step': 0.3 * 100, 'portion': 0.5}
```



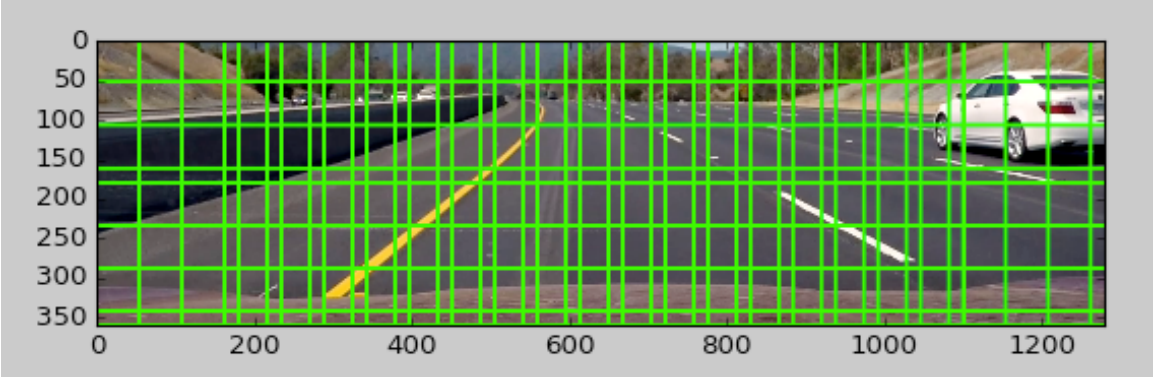
Level 2:

```
{'size': 140, 'x_step': 0.3*140, 'y_step': 0.3*140, 'portion': 1}
```



Level 3:

{'size': 180, 'x_step': 0.3*180, 'y_step': 0.3*180, 'portion': 1}



Rubric Point#4
Show some examples of test images to demonstrate how your pipeline is working. How did you optimize the performance of your classifier?

Answer:
To reduce the time to process a single frame, currently I have managed to reduce the time to about 2.5sec/frame from 9.5sec/frame by switching from SVC() classifier using rbf kernel to LinearSVC() classifier. Further possible explorations will be discussed in the end of this writeup.

4 methods are implemented regarding false-positive rejections:

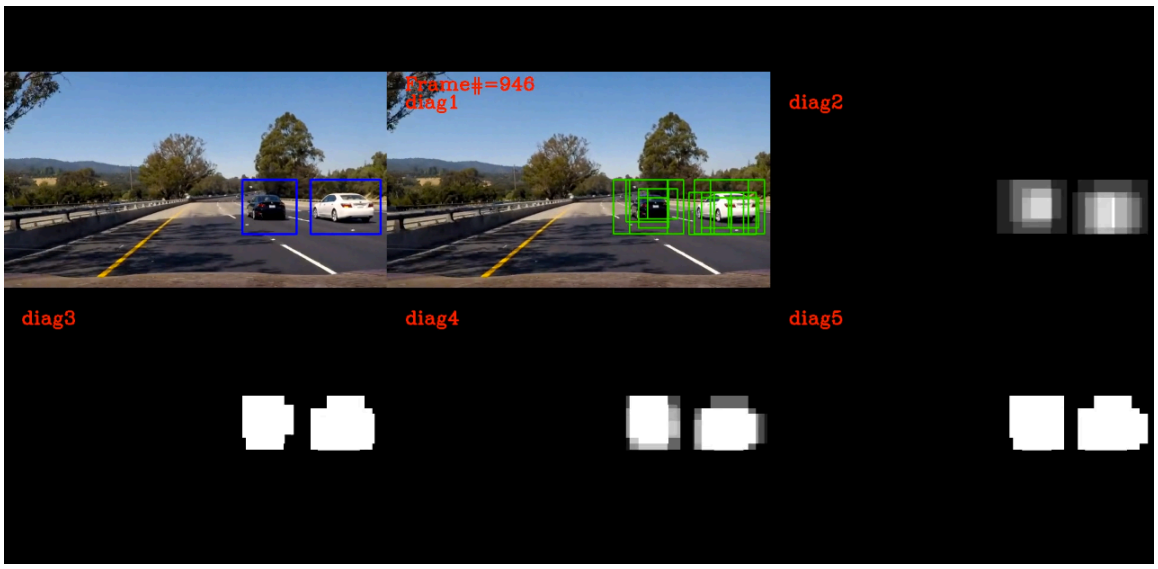
(1) Hard negative mining, which is discussed in Rubric point#2.

(2) LinearSVC wrapped by CalibratedClassifierCV() will assign a probability for a prediction, which can be used to reject false positive. The probability threshold is set to 0.7 currently.

(3) For a single frame, heat-map threshold is implemented. The threshold is controlled by parameter self.heatmap_threshold_single_frame (= 2) in Filter().

(4) For consecutive frames, a latency mechanism is implemented. The latency controls how many heatmaps are stored in an list. One heatmap corresponds to one frame. The processing pipeline takes the list and sum up all the heatmap (pixel-wise) in the list. Another threshold is applied to the resulting heatmap to further reject false positive. The latency is a parameter called self.heatmap_latency (= 5) in Filter(). The threshold is self.heatmap_threshold_multi_frame (= 3) in Filter().

Here is an example of the output of the video processing pipeline:



diag1: Bounding box found in a single-frame processing
diag2: Heatmap for a single-frame bounding boxes
diag3: Tresholded Heatmap for a single-frame bounding boxes
diag4: Multi-frame accumulative heatmap, latency=5
diag5: Thresholded multi-frame accumulative heatmap

Rubric point#5:
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.)

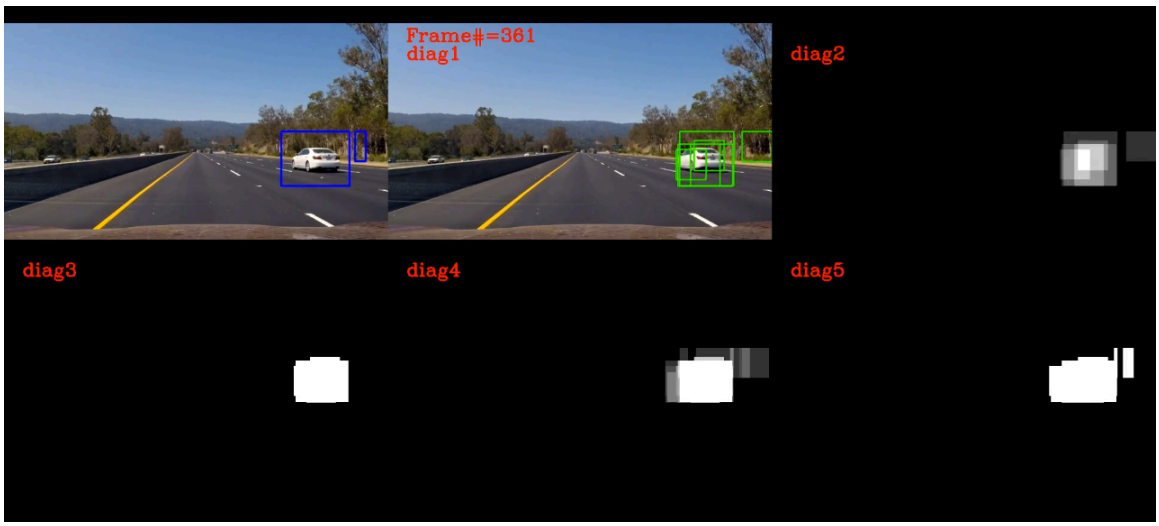
Answer: The video is called “out.mp4” in the github vault.

Rubric point#6:
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.

Answer: I believe this issue has been addressed in my answer given to Rubric point#4.

Rubric point#7:
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?

The primary problem of vehicle detection is false positive rejection. As you can see, I have implanted a few mechanisms to reject false positive. Yet in the final video there are some places where false positive show up. Below is an example:



Here the false positive are the trees. Perhaps a harder negative mining could alleviate the problem

The second challenge is the size of the bounding box. As we can see here, the box is somewhat bigger than the car. This is because the step that is used by the sliding box procedure. Perhaps a finer step would alleviate the problem. But that increases run time.

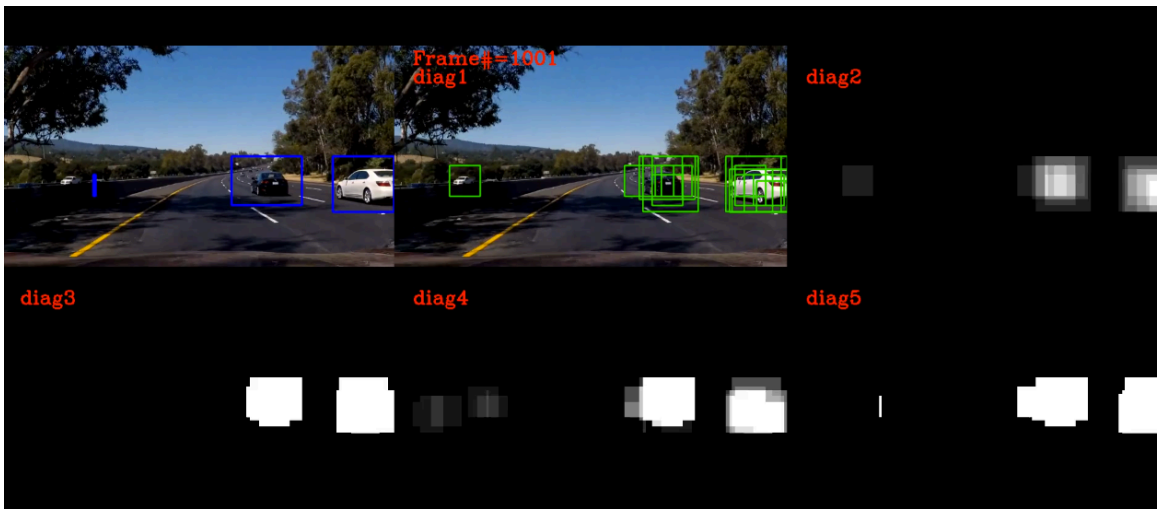
Which brings to the third point: processing time. Currently the processing time is 2.5sec/frame, which is far from satisfactory. A couple thing to try here, if given more time:

(1) Can we use the cv2 hog feature extractor instead (written in C++)?

(2) Implement batch prediction. Current implementation is loop through the bound box and run feature extraction and prediction one by one. If we run in batch, we might be able to use vectorized algorithm in LinearSVC() if there is any.

(3) Is the feature extraction, especially the HOG feature too slow? Again, I am currently extracting feature one box by another box while sliding through the image. In the lecture it is mentioned we can extract HOG for the whole image all in once. But I didn't figure out how we can do this while using sliding box of different size in later stage.

(4) The pipeline sometimes detects the car from the opposite direction, which might or might not be a problem. See an example below:



The detection of car from the opposite direction is not very reliable in this stage, which cause some jitter in the bounding box in the final result (blue image).