

Vehicle Detection Project

Histogram of Oriented Gradients (HOG)

1. HOG features extraction

Code : line 21 - 33 in vehicle_classifier.py

To extract hog features of training samples, the idea is:

Firstly, convert original image to a color space

Secondly, call `skimage.hog()` to extract hog features in the image

Here is an example using the `yCrCb` color space and HOG parameters of `orientations=8`, `pixels_per_cell=(8, 8)` and `cells_per_block=(2, 2)`:

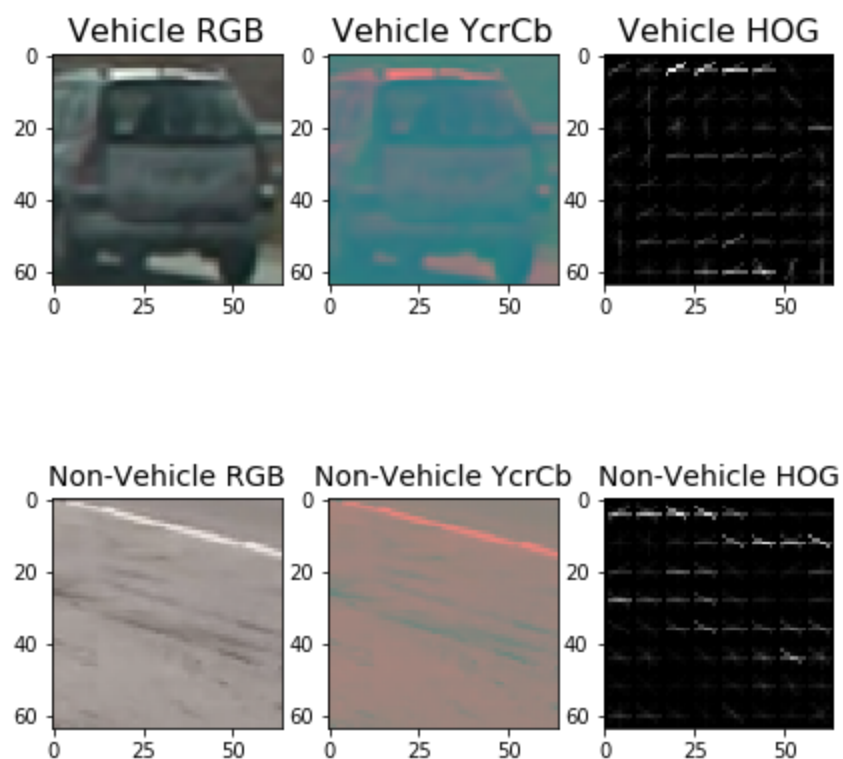


Figure 1. Hog features

2. Color space of HOG parameters choice

In order to settle the final choice for color space and HOG parameters, I tried various combinations of parameters and color space.

I use a linear SVM classifier to help me make the decision, the one which enable the classifier to achieve the highest accuracy on test dataset is chosen.

To note that, the test dataset I use is some screenshot images which I collected manually and resized to 64 by 64 pixels.

As shown below, the provided dataset is not well representative of real data in the video. We could clearly see that, the svm performs perfectly on whatever color space on the provided set. Its accuracy is consistently close to 100% on the provided dataset. However, there is a great distinction between its performances on different color space over the screenshot dataset.

Given that orientations=8, pixels_per_cell=(8, 8) and cells_per_block=(2, 2), YCrCb stands out.

accuracy	RGB	YUV	YCrCb
Training dataset	0.973	0.989	0.987
Screenshot images test	0.661	0.789	0.792

Table 1

Given that orientations=8 and cells_per_block=(2, 2), I also explored the pix_per_cell parameter on different color space. As shown below, by assigning pix_per_cell to 16, the classifier accuracy is dramatically improved to 86.64% on screenshot dataset.

pix_per_cell	RGB	YUV	YCrCb
8	0.661	0.789	0.792
16	0.6399	0.8614	0.8664

Table 2

The final choice for color space and hog parameter is:

Parameter	Value
orientations	8

pixels_per_cell	(16, 16)
cells_per_block	(2, 2)
Color space	YCrCb

Table 3

3. Classifier training

Code: line 41 - 63 in `vehicle_classifier.py`

3.1 Data augmentation:

In order to improve the the classification accuracy on real data. I augmented the training dataset by adding more vehicle examples and non-vehicles examples from the screenshots and internet.

To generate more examples, I use `crop_image` function (line 11 - 19 in `vehicle_classifier.py`) to crop the top-right, top-left, bottom-right and bottom-left parts of a car image and then resize them all to 64*64 png images.



Figure 2. screenshot car image



Figure 3. augmented dataset

3.2 Train and Test classifier:

The classifier I use is linear SVM, which is fast to train and performs well. I trained the classifier in the following steps:

- Shuffle augmented dataset
- Split into training and testing dataset
- Fit training dataset in linear SVM model and persist to a file (svc_model.sav)
- Test the classifier on test dataset

In the end, the classifier achieves 98.8% accuracy on test dataset.

Sliding Window Search

1. Sliding window search.

Code : line 8 - 46 in vehicle_detection.py

As sliding window search is a very expensive operation, I restrict the sliding window search region to the bottom left part of the image.

As a result of perspective, the cars further away are smaller and those closerby are bigger, it doesn't make sense to slide small windows across the bottom of the image since the cars that appear there are too big to fit into the window. The same reasoning goes for the image area in the middle, big window won't help there as cars are too small to be recognized in a big window.

Five different sized windows are used in my sliding window algorithm. As shown in the image, window sizes varies along with the perspective.

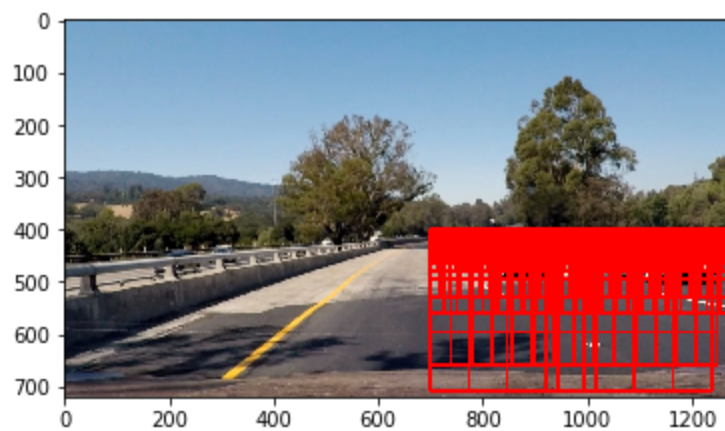


Figure 4. All sliding windows

x_start_stop	y_start_stop	window_size	xy_step
(400, 464)	(700, 1280)	(64, 64)	(0.1, 0.1)
(416, 480)	(700, 1280)	(64, 64)	(0.2, 0.2)
(400, 496)	(700, 1280)	(96, 96)	(0.2, 0.2)
(432, 528)	(700, 1280)	(96, 96)	(0.2, 0.2)
(400, 528)	(700, 1280)	(128, 128)	(0.2, 0.2)
(432, 560)	(700, 1280)	(128, 128)	(0.2, 0.2)
(400, 596)	(700, 1280)	(196, 196)	(0.2, 0.2)
(400, 596)	(700, 1280)	(196, 196)	(0.2, 0.2)
(464, 660)	(700, 1280)	(196, 196)	(0.2, 0.2)

(464, 720)	(700, 1280)	(244, 244)	(0.3, 0.3)
(464, 720)	(700, 1280)	(244, 244)	(0.3, 0.3)

Table 4. sliding window region

2. Pipeline

Code : line 54 - 141 in vehicle_detection.py

Ultimately I used YCrCb 3-channel HOG features as feature vector, which provided a nice result. The pipeline is:

- Detect cars with sliding window
- Locate heat areas with heatmap
- Fit heat areas with bounding box

Here are some example images:

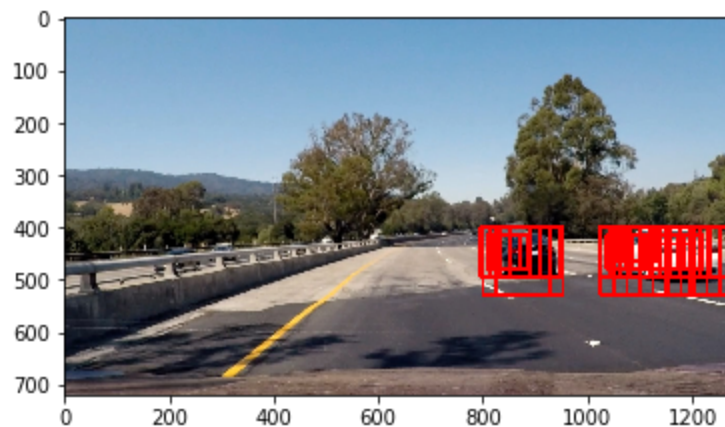


Figure 5. Active sliding windows

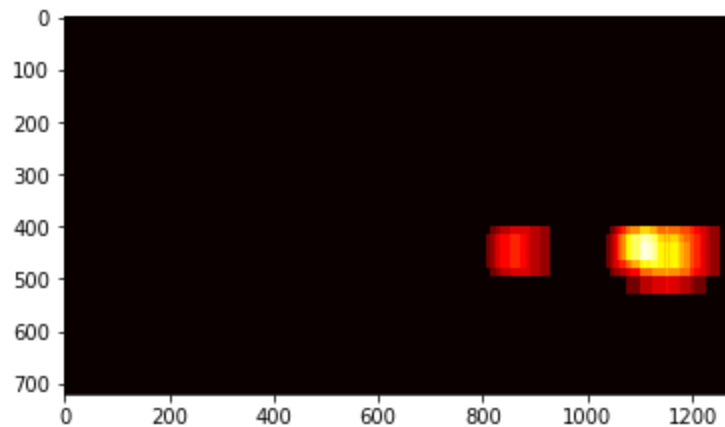
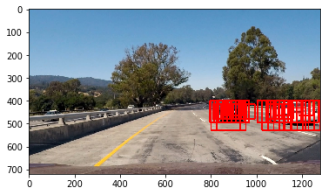
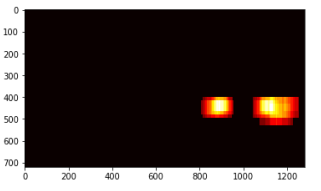


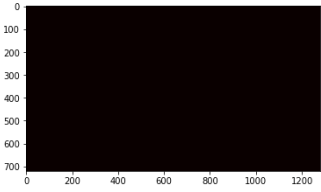


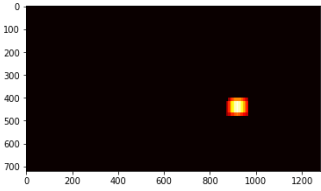
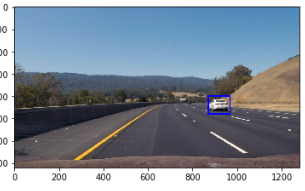


Figure 6. Corresponding heatmap



Figure 7. Resulting bounding box

	Sliding window	Heatmap	Bounding box
Test 1			
Test 2			
Test 3			

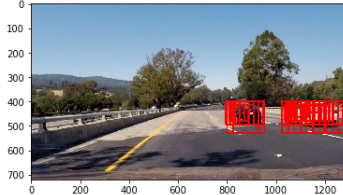
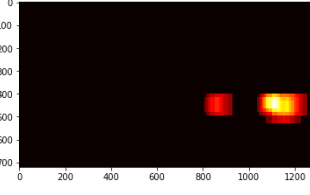
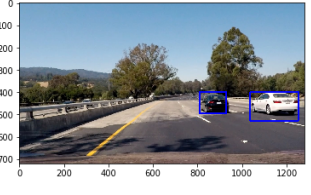

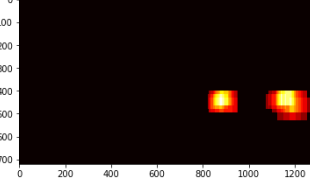

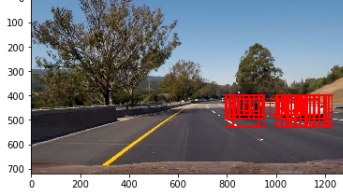
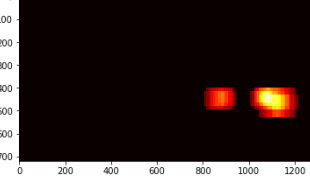
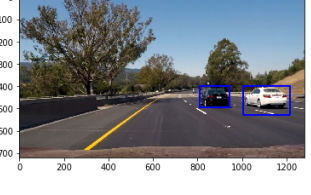
Test 4			
Test 5			
Test 6			

Table 8. Six frames pipeline example

Video Implementation

1. Final video output.

My pipeline turns out to perform reasonably well on the entire project video, although there are somewhat wobbly or unstable bounding boxes, the vehicles are well identified most of the time with no false positives.

Here's the link to my video: <https://youtu.be/jKj8i51It5g>

2. False positive and overlapping bounding box filtering

Code : line 106 - 110 in vehicle_detection.py

The idea is very intuitive, cars are not likely to be tall and thin like a pin. It's often the case that such thin and tall windows are the overlapping area of two close cars.

The windows with very small height or width, or with large ratio of height over width are filtered out.

3. Bounding box stabilization

Code : line 113 - 123 in `vehicle_detection.py`

My bounding box stabilization technique doesn't work out well. :(

My idea is simple, average the box size with the last frame:

- Calculate the center point of the car. (Code : line 65 - 74 in `vehicle_detection.py`)
- Find the box of the same car in the last frame (the one with largest overlapping area with current bounding box). (Code : line 77 - 92 in `vehicle_detection.py`)
- Average current box with the one in the last frame (Code : line 115 - 122 in `vehicle_detection.py`)

Below is an illustration of how this algorithm works:

Given `boxes_in_last_frame = []`, the output is like the following figure:

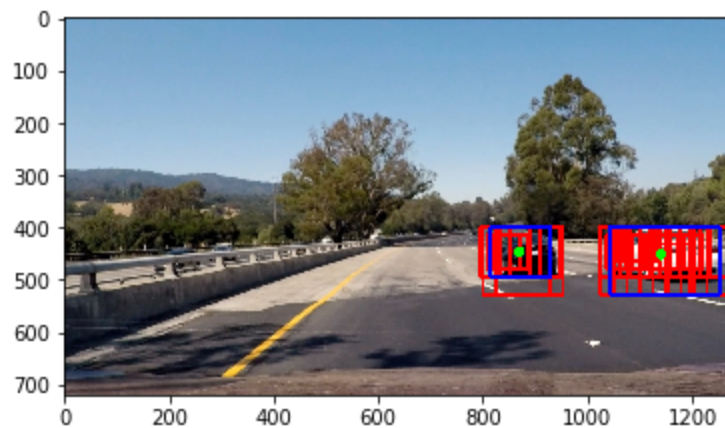


Figure 8. Bounding box with no averaging

Given `boxes_in_last_frame = [(1042, 400, 210, 90), (814, 400, 160, 96)]`, the bounding boxes are averaged with the ones in the last frame.

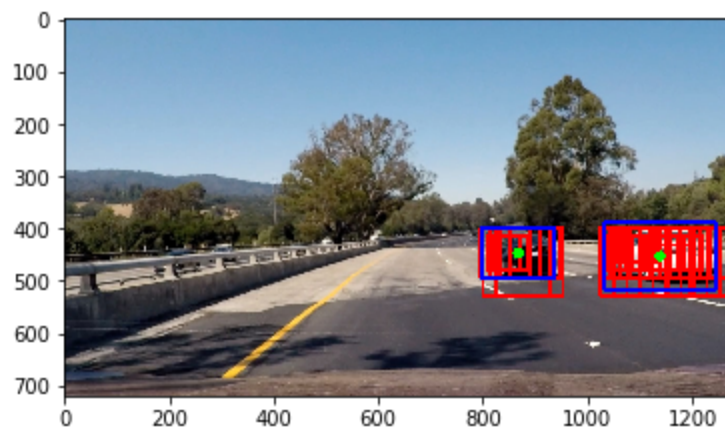


Figure 9. Bounding box with averaging

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?

Problem 1: sliding window approach is too slow.

It took my mac pro 20 mins to process the 50-seconds project video. Real autonomous driving car won't work out with my implementation.

Problem 2: too many hand-tuned parameters.

The hog feature parameters and sliding window algorithm are too hand-crafted and specially tuned for the project video only, which may not generalize well to other scenarios.

Problem 3: classifier leverages only HOG features.

If I were given more samples, I would include more features to feature vectors. The reason for which I didn't do so is that I fear the curse of high dimensionality if I use too many features with only less than 9000 samples provided.