Vehicle Detection and Tracking

By Erwan Suteau, Aug 27, 2017

# Histogram of Oriented Gradients (HOG)

## Parameter Selection

*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.*

*Explanation given for methods used to extract HOG features, including which color space was chosen, which HOG parameters (orientations, pixels\_per\_cell, cells\_per\_block), and why.*

The calculation of the histogram of oriented gradients is defined in the function get\_hog\_features() (line 28 of main.py).

The selection of parameters is based on trial and error. I started with the default parameters defined in the lesson, and then played around those values to see if the classifier would train better. I prioritized accuracy over speed.

I ended up with:

* Color Space: ‘YUV
* Orientations: 11
* Pixels per cell:32x32
* Cells per Block: 2x2
* Transform Sqrt: False
* Block Norm: ‘L2-Hys’

‘YUV’ seemed to perform better or equal to YCrCb. The classifier accuracy was significantly higher with one of those 2 color maps than RGB.

I selected a different norm function that the default one (L1) because the skimage library will use this normalization method in the future. I did not seem to make a huge difference from L1.

I also disabled the square root transform because it was causing problems with some images, returning nan values for small sections.

## Feature Selection

In addition to the HOG features, I also used spatial features and color histograms. Combined together with HOG, they seem to produce an excellent set of inputs for the classifier to train on. The classifier performed much better with all 3 combined.

## Classifier

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

The HOG features extracted from the training data have been used to train a classifier, could be SVM, Decision Tree or other. Features should be scaled to zero mean and unit variance before training the classifier.

Classifier training can be found in function run\_classifier(), line 322 of main.py

I chose to use Linear Support Vector Classification for this project, as it gave me around 99% of accuracy on the test set.

I used almost the entirety of the smaller training set provided by Udacity: 8792 images for each dataset (car and non-cars).

I made sure to normalize the dataset to zero mean and unit variance using the StandardScaler() class. (See function prepare\_feature(), line 306 of main.py)

The dataset was separated into a training set (80%) and a test set (20%). I used the built in score function on the latest to check the accuracy of my classifier.

# Sliding Window Search

## Implementation

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?

A sliding window approach has been implemented, where overlapping tiles in each test image are classified as vehicle or non-vehicle. Some justification has been given for the particular implementation chosen.

My initial approach was the one introduced in the lesson, which for a given set of window size and stride, extracts all the window pixels from the original image and then calculate the feature maps (HOG, spatial and color histogram features). This was implemented in the find\_windows\_slow() function.

I realized quickly that this approach was not very efficient, because with the window overlap we calculate multiple times the gradients at the same location. I kept it in the code as it gave me a reference.

To speed up computation, I implemented find\_windows\_fast(), which calculates the HOG features only once, and then extract the pixels corresponding to each sliding window.

Finding the right parameters for car detection was tricky. First of all, you need to restrict the sliding window boundaries to avoid spending time outside of the road. I limited the search window to the lower portion of the image, with Y in [400, 656]

Then you want a window size that is not too big so that you can detect cars at the end of the road, but you also want to able to detect cars when they are close to the camera.

I played a lot with those parameters, using multiple window sizes (256x256, 192x192, 128x128, 96x96, 64x64). In the end, a single window size of 64x64 gave me a good enough level of detection and helped speed up my calculations.

For overlap, I chose a stride of 1 cell, which corresponds to 8 pixels. This is a small stride, but it gave me good results with the window of 64x64.

## Examples

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

Some discussion is given around how you improved the reliability of the classifier i.e., fewer false positives and more reliable car detections (this could be things like choice of feature vector, thresholding the decision function, hard negative mining etc.)

After the window detection, we create a heat map by setting the pixels located inside each window to a value of 1 and summing over every window that was classified as a car.

Then we apply a threshold to remove false positive and keep the image patches with the most overlap. My final threshold was set to 10.

You can find below examples of my pipeline. This was produced on frame 935 of the test video. You can clearly see that the classifier detects false positives which are then discarded by the heat map thresholding, giving a pretty good detection of the car boundaries.

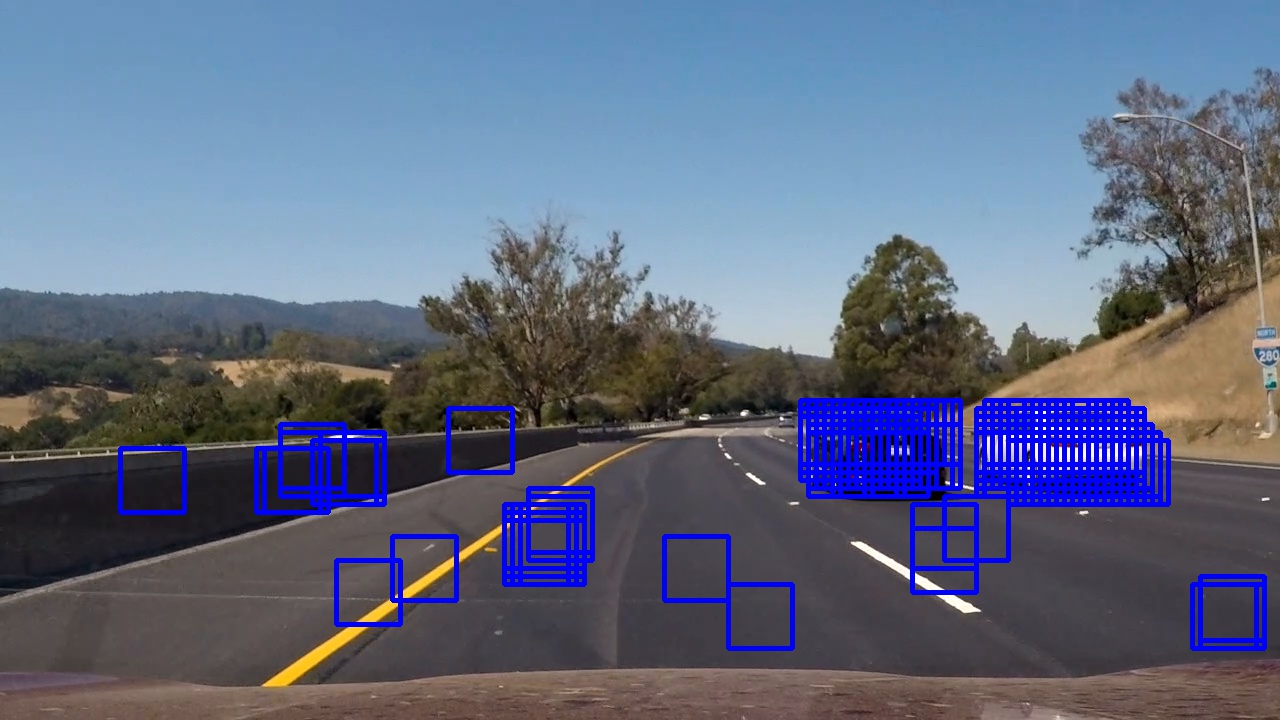


Figure : 64x64 Sliding Window Detection



Figure : Heat Map Thresholding



Figure : Final Boundaries drawn back on the original picture

# Video Implementation

## Results

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 sliding-window search plus classifier has been used to search for and identify vehicles in the videos provided. Video output has been generated with detected vehicle positions drawn (bounding boxes, circles, cubes, etc.) on each frame of video.

The output video can be found in my github project, at the following link:

<https://github.com/esuteau/car_vehicle_detection/blob/master/output_video.mp4>

## False Positive Filtering

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.

A method, such as requiring that a detection be found at or near the same position in several subsequent frames, (could be a heat map showing the location of repeat detections) is implemented as a means of rejecting false positives, and this demonstrably reduces the number of false positives. Same or similar method used to draw bounding boxes (or circles, cubes, etc.) around high-confidence detections where multiple overlapping detections occur.

With the video input feed, we know that there is a very high correlation between consecutive frames, meaning that the position of the vehicle will vary very slightly between them. The video input is at 30 frames/sec, and if you look at 1s of the video, you will see very small difference between the initial and final positions of the car.

We can use this at our advantage to filter false positives better. What I did was a simple averaging of the heat map over 1s (30 frames). False positives, which could be considering as noise, would average out to a value close to zero, whereas the car positions would stay at a high value in the heat map.

If this average is not done efficiently computation wise, it could slow down significantly the pipeline. To do it properly, you need a few things:

* Store the 30 consecutive images: we use collections.deque container to do that efficiently, which if a FIFO of limited size.
* We do not calculate the average heat map by using the numpy.mean function over 30 images at each frame. This is not very efficient. Instead, we calculate the cumulative sum of the heat map, which requires one single addition at every frame. When the FIFO is full, just like deque, we subtract the first image of the queue before discarding it. Doing this scales really well with any memory length,

# Discussion

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?

Discussion includes some consideration of problems/issues faced, what could be improved about their algorithm/pipeline, and what hypothetical cases would cause their pipeline to fail.

The biggest hurdle I encountered during this project was just to wait for the results. Running this on a 6 year-old laptop did not help and I had to use a bunch of tricks and really think about computational efficiency to get the most of it.

My best friend in this process was the pickle module. I used it to store anything I did not have to recalculate, like for example the SVC classifier, but even the detected windows. Storing the latest could let me iterate quickly with different memory length and heat map thresholds without having to go through the entire pipeline again.

The final video is satisfying, but it could still be enhanced.

I think we can do a much better job with the classifier. Even though I have a high accuracy on the training set, I realized that many false positive are still detected.

Here I see different options:  
1. Use a Convolutional Neural net to detect cars. This should improve the accuracy of the classifier and thus help with detection

2. Use the additional datasets available to train of more samples. Hopefully this could help generalize a lot better and help detection.

Also some work could be done to find the right tradeoff between computation speed and accuracy. This pipeline should run in real time on a car, so optimizing this algorithm for speed would make a lot of sense. (Also using better hardware and doing operations in parallel with graphics cards or FPGA could help)