Andriy Kumanovskyy – Vehicle Detection Project Project 5 (Resubmission)

Histogram of Oriented Gradients (HOG)

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

The extraction of HOG features was a three step process involving inspiration from the classroom code in order to first define and test the feature extraction function "get_hog_features" (**section 3a**), then define an "extract_features" function (**section 3b**) in order to extract features (histogram, special and hog) from a list of images taken from the lesson which can then be fed into a Support Vector Machine (SVM) classifier. In the **section 3c**, the Hog Feature Extraction function was applied to all vehicle & non-vehicle images in order for the system to be able to detect, and with the right "boxing" function, track vehicles. The features were extracted utilizing *9 orientations* per cell as suggested in the lesson (to minimize my computation requirements, as well as perform well), as well as with the *YUV* color space, which seemed the most promising in grouping different colored cars together on a color space plot as seen in **section 0** – more on this later.

Here is an image of HOG feature extraction on an example image:

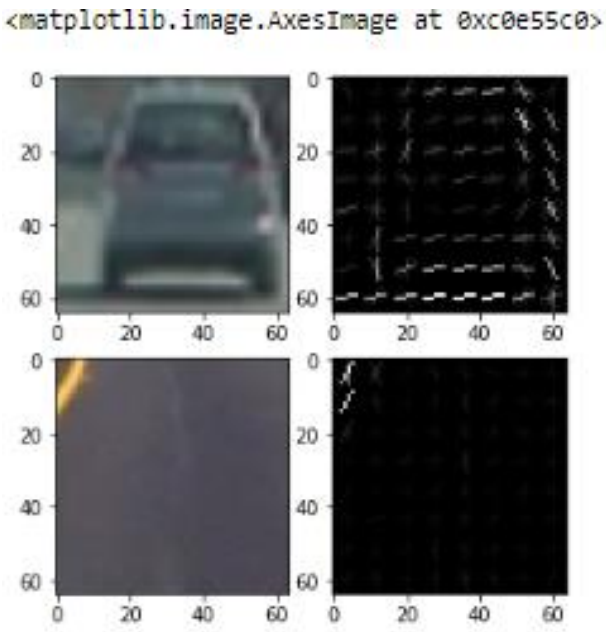

Figure 1: HOG feature extraction on an example image

1 b. Explain other features utilized

As mentioned before, aside from HOG features – 32 by 32 pixel spatial binning of color and histogram features were also extracted, giving extra information to the classifier in future steps. These features were defined in **section 3b** and inspired by the classroom lessons.

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

ORIENTATION AMOUNTS (this parameter was the 2nd last to be finalized):

While at first I employed a higher number of orientations in order to maximize how robust my algorithm was, I ran into a bottleneck with my computer. Since the computer I was running on was less than ideal (my main one is having issues with Jupyter as of late, which I will try and resolve soon), and often ran into a "memory problem" during feature extraction, I was forced to make the extraction of features more efficient and reduce the amount of memory/computation needed.

One such way was to reduce the number of orientation back a little bit from my initial attempts in the mid-teens to 9 (which was the number presented in the lesson). This reduces the size (len) of the feature vector, while still performing the required task of detecting vehicles when applied to the video. When I retrained with the orient setting set to 9 (as opposed to 14) – the detection algorithm performed with no visible difference when compared to the algorithm with 14 orientations, at the benefit of lower memory usage.

This also reduced the amount of time the computation took for both feature extraction and training, which allowed me to focus more time on employing code and testing various different cases and hyper-parameters empirically.

HOG COLOR SPACE (this parameter was the last to be finalized):

1.) Empirical Reasoning for choosing YUV:

The first attempt was to try the whole project with RBG as provided in the lesson, including the Hog color space section. However, through testing I came across a few red flags with the RGB:

RGB accuracy of SVC = 0.9848 (ok, but lower), while picking out useless objects (a lot of false positives [*with same relative heat map settings*]), and absolutely **no** cars were detected

After this I explored various different color spaces in **section 0** to see which one managed to group all of the car colors in a nearby location in the color space, with the most promising results (out of the ones that I tested) in YUV and HSV, as presented below as well as can be seen in the PDF file attached:

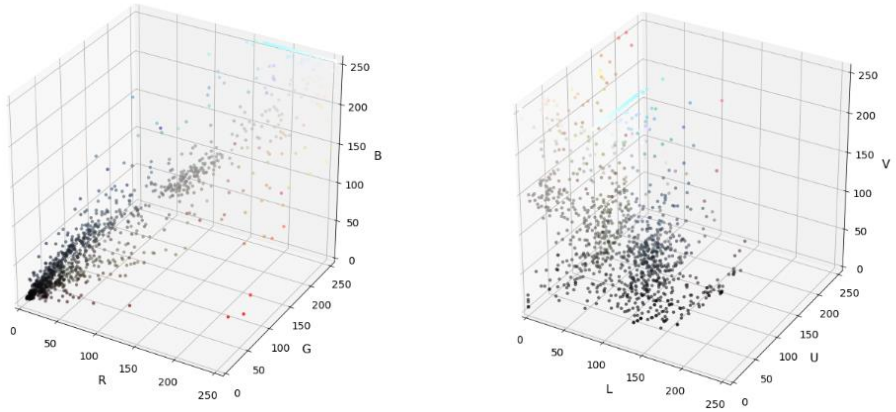

Figure 2: RGB color space on the left (colors scattered), LUV color space on the right (colors grouped☺)

Following this, I tested my project with the YUV color space (by retraining) which proved to be exceptionally promising:

YUV accuracy of SVC = 0.9921 (a bit better compared with RGB), while rarely picking out useless objects (very few false positives [*with same relative heat map settings*]) and **all** of the cars that were in the frames were detected.

After this it was clear that I needed to utilize YUV (perhaps HSV, as it seemed to yield the exact same color space graph)

2.) Analytical & Research Bases Reasoning for choosing YUV:

- Certain lesson images have suggested YUV, and this was verified when I tried testing the various images by myself – I can see the separation of the vehicles in a nearby space on the graph (**figure 2**).

- Cars tend to be rather saturated in color and the YUV space does a great job of grouping the various different colors of cars into the same group as well have capability for illumination invariance (necessary during night time). I have also noticed the exact same results on an image in HSV space, however opted to use LUV due to best practices in the industry (which brings me to my next point)

- My hunch was validated when I decided to dwell a little bit into literature, and I can see that Nvidia's self-driving car also utilized the "YUV" color space:

"The input image is split into YUV planes and passed to the network."

https://devblogs.nvidia.com/deep-learning-self-driving-cars/

PIX_PER_CELL, CELL_PER_BLOCK, HOG_CHANNEL (values taken from lesson)

While the original goal was to adjust all values to their optimal, the values given in the lesson proved to be effective with all of the other settings I have used for this project. The pix_per_cell, cell_per_block and Hog_channel: "All" values in the lesson proved to be able to detect an acceptable amount of HOG features from the images provided, without being too computationally stressful, or too topical in the execution.

SPATIAL BIN & COLOR HISTOGRAM FEATURES: Color Space for both, Spatial size & number of histogram bins (Inspired by the lesson)

While at first I had a hunch that my color special features and color histograms bins needed to be YUV as well, this proved to not be the case – and I was able to get successful results setting those to the lesson default values of RGB. It was only the HOG feature extraction color space that needed to be YUV or HSV (I used YUV).

Spatial size and number of histogram bins were taken from the lesson, and proved to not need alteration, once again, proving to be able to detect an acceptable amount of spatial and color histrogram features from the images provided, without being too computationally stressful, or too topical in the execution.

The following is a summary of all of feature extraction/training hyper-parameters chosen:

For HOG feature extraction, the final parameters are:

- Color_Space = 'YUV'
- Orient = 9
- Pix_per_cell = 8
- Cell_per_block = 2
- hog_channel = 'ALL'

For the spatial bin features, the final parameters are:

- Color_Space = 'RGB' (*I kept since making this into YUV didn't seem to yield any benefits, it was only crucial to use YUV for HOG feature extraction*)
- Spatial_size = (32,32)

For the color histogram features, the final parameters are:

- Color_Space = 'RGB' (*I kept since making this into YUV didn't seem to yield any benefits, it was only crucial to use YUV for HOG feature extraction*)
- Hist_bins = 32

In summary: After each test run (running the code at each step) I tested the accuracy and then made a list of performance in order to assess the best possible results at each iteration. The final parameters were first taken as best practice values from the lessons, and then tweaked and personalized for the individual scenario of my code and its unique training. This method applied to everything, including the sections after the feature extraction / training such as the sliding window approach.

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).

As discussed previously, I trained a linear Support Vector Machine (SVM) using HOG, spatial bin and color histogram features. The final results yielded a classification accuracy / test accuracy of 99.21%, validated on 20% of the overall data at random. I extracted the features in **section 3c** and trained my classifier in **section 4.** This code can be found roughly halfway through the PDF file.

Sliding Window Search

1 a. Describe how (and identify where in your code) you implemented a sliding window search. How did you decide what scale to search and how much to overlap windows.

The sliding window search was applied in **section 6**, utilizing the preliminary code from **section 5** to define the "find_vehicles" and "convert_color" sections from the lesson. Initially my approach was the same of the lesson – an attempt to get a "for-loop" to work in order to have various different iterations of the sliding window search with different scales in order to be able to detect vehicles that are close and further away. However, I was not able to get the "for-loop" to work effectively (*perhaps a dictionary data structure, or a modified glob.glob function would have yielded the results, and the commented out for-loop in my code would have worked*) and I opted to use a method of repeated "bboxes_output1" layers appending the layered results with a "bbox_extend" function.

From the start, I had a hunch to repeat the search function multiple times for difference scale values to generate multiple-scaled search windows -- since we aren't sure what size the vehicles would be it would be best to run program in multiple different scales. The start values were chosen at the start close to the vertical centre of the video (720 pixel height / 2 roughly), as well as keeping in mind to not steer too far away from not looking above 400px in the y-axis and not looking below 500px (taken from studying the video space, and where cars typically appear). The values were then and adjusted accordingly for performance through empirical testing on an image.

Initially I spent some time with a higher number of repeated loops (alternate window overlap layers, shown on the next page) for the bbox extend function, but this yielded video output results that seemed to merge two neighboring cars with ONE box. This was less than desirable since in a real life situation you would not want the self-driving car to assume that two cars are one… and then make predictions on how to avoid this assumed block of two cars as if it was one. It is much better to be able to separate the two cars for as long as possible. Perhaps this was caused by:

i.*too many overlapping boxes (caused by a large amount of repeated ystart/ystop iterations)*
ii.*less than ideal heat map limits*

I chose to reduce the amount of repeated loops (i. solution, reducing the amount of overlapping boxes / alternate window overlap layers) This also seems to reduce the amount of computation, further improving my workflow with the project on my less than ideal laptop from 2011. While doing this I further improved my ystart, ystop (masking) and scale (predetermined window sizes) values by reducing their variance. A slight difference is effective in being able to detect cars that one layer of alternate window overlaps didn't, but when they are too far off it seemed to reduce the function of the repeated loop detection, hence there was a balance that needed to be maintained – which was found through empirical testing. In the end, while I haven't completely alleviated the problem (of two neighboring cars being detected by ONE box), I have reduced a noticeable amount using the method above (using a small number of repeated loops).

I created what might be called 3 different "epochs", or alternate window overlap layers in order increase the robustness of the detection algorithm, while preventing the "box flying over two cars issue to a large extent". Ystart/ystop parameters' starting points were chosen to vary just a little bit from one to the other in order to avoid creating redundant calculations, while also taking advantage of the function iterating 3 times. Additionally, from observations, I've noticed that vehicles within the proximity of the

car and those quite a bit further away can be effectively detected with scales that are different, but not drastically different. Perhaps this would differ with a different project video with cars very close and very far away. If I were to utilize an increased amount of sliding box iterations I would add one with a scale that is large and one that is small compared with the current 3 used, and keep roughly the same ystart and ystop values.

In summary, the following hyper-parameters were chosen, with hyper-parameters from training being consistent with the hyper-parameters for the sliding window function:

Color_space = 'YUV'
Orient = 9
Pix_per_cell = 8
Cell_per_block = 2
Hog_channel = 'ALL'

Alternate Window overlap layer 1:
ystart = 400
ystop = 512
scale = 1.05

Alternate Window overlap layer 2:
ystart = 400
ystop = 516
scale = 1.3

Alternate Window overlap layer 3:
ystart = 420
ystop = 516
scale = 1.5

1 b. Show some examples of test images to demonstrate how your pipeline is working. What did you do to try to minimize false positives and reliably detect cars?

As was stated in the previous sections of the code, I searched utilizing YUV 3-channel HOG features, and spatially binned color and histograms of color in the feature vector. This was fed into my sliding window algorithm and overall yielded some pretty good results as can be below in figure 3:

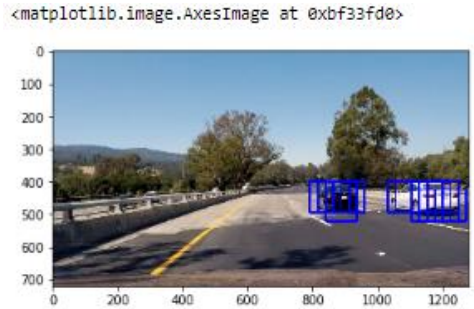

Figure 3: Pipeline post feature extraction, training and sliding window implementation

Regarding the rejection of false positives (as well as to avoid overlapping windows in the final results, both image and video output), a **heat-map** was implemented in **section 7** of my code. Its purpose is to detect which area are most likely to have the image that we are looking for (vehicles) by rejecting any number of windows over an object that is under the threshold. Windows that test positive in the classifier add 1 to the heat-map for each pixel in the window. While I am currently using a threshold of 0, if the results end up not being favorable and too many false positives occur, I will increase this number to the 5-10 range (The results were pretty good, I so decided to leave it for now).

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.)

The video is attached in the zip attachment as an .mp4 file. The video output performs reasonably well on the entire project video, while minimizing the amount of time two cars yield a single box (though I am sure that it can be improved). The number of false positives it kept to a minimum, and the algorithm is able to detect all vehicles in the proximity of the camera.

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.

The method of combining overlapping bounding boxes in images is implemented in the “draw_labeled bboxes” function in **section 7** of my code. While not entirely related to the **heat-map** section, I kept this function under the same heading. Perhaps it would have been slightly more clear to add a different heading ☺.

As discussed previously, the approach used to detect false positives was to use a **heat-map** in the pipeline. You can find the video implementation section of the code near the end of the PDF .html file in **section 9** of my code, which utilizes all of the previous functions defined as one pipeline, called “tracking pipeline” in **section 8**.

Discussion

Some of the problems I faced with this project:

A lot of the problems in this project were already mentioned in the previous section as I described them, however here they are in summary:

-computer issues which limited the amount of RAM memory I could use, hence constricted me to a smaller amount of orientations for feature extraction (HOG, spatial, color histogram, ect)

-I ran into some "for loop" implementation issues when I tried to implement it in in my sliding-window algorithm, then decided to apply an alternate solution of re-writing several lines of alternate window overlap layers, which are slightly different for improved robustness at different distances and locations. I have used this approach in project 3 (my favorite :D) – behavioral cloning. In the future I will come back to this and try to implement it using a for-loop method, perhaps it would work a little better, and at the very least I will learn something ☺

Some of the further things I can improve is to:

- increase the heat-map threshold, as the pipeline may to fail in an area with a lot of objects that may resemble vehicle, however it would be possible to reduce the issue here by increasing the threshold for the **heat-map**

- implement "scipy.ndimage.measurements" in order to have more information with each run to make better decisions

- Further enhance and refine my approach to remove large rectangle boxes from fitting over two cars as if they were one (utilizing car centroids)

- combine this method with the advanced lane lines project

- Perhaps utilize a wide-angle camera lens for the tracking video to be able to see more of the surrounding

I also had a thought to apply masking to remove cars from outside of the range of what is expected (since I was getting annoyed at the algorithm picking up cars from the opposite lane over the separation column) – but I realized that it might be dangerous as we never know from where a car could be coming from, and creating blind spots in the detection algorithm would not be wise.