Project 5 - Vehicle Detection

Udacity Self-Driving Car Nanodegree

February 26, 2017

Robert Crowe

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 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.

Rubric Points

Here I will consider the rubric points individually and describe how I addressed each point in my implementation.

Writeup / README

1. **Provide a Writeup / README that includes all the rubric points and how you addressed each one.**

You're reading it!

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 file features.py – specifically in get_hog_features (lines 12-29)

I like to start by breaking down the different sets of parameters, rather than trying to explore the entire NxN parameter space. During this exploration I set each set of parameters to when seems to be a reasonable guess. I take each set one at a time, and try to find the optimal values in that space, with the hope that they are somewhat independent, and that when I combine them they will yield reasonable results.

Tuning the HOG came only after I had explored color spaces. I trained the model using each color space (RBG, HSV, HLS, LUV, and YCrCb), selecting all channels and each channel individually, training an SVC model and recording the test accuracy:

Color Space	Channel 0	Channel 1	Channel 2	All Channels
HSV	0.9099	0.9085	0.9496	0.9811
HLS	0.9015	0.9476	0.9082	0.9837
RGB	0.9448	0.9386	0.9445	0.9699
LUV	0.9507	0.9054	0.9293	0.9837
YUV	0.9451	0.9065	0.9313	0.9862
YCrCb	0.9485	0.8882	0.9333	0.9837

The YUV color space had the highest score when all channels are used, and channel 0 of the LUV color space had the highest single-channel score. Clearly each color space contains the most information when all channels are used, but the number of features is also much larger, and the Curse of Dimensionality may come into play. Also, the processing required for all channels is much larger. So after experimenting to see how much I actually needed the increased accuracy of using all channels, I settled on using LUV channel 0 only, primarily with the goal of trying to process the video in real time as would be necessary in an actual vehicle.

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

I then took each HOG parameter (orient, pix_per_cell, and cells _per_block) and performed similar testing. During this phase I also recorded the time required to train the model. The results are contained in config.py, in the env global dictionary, which is imported by main.py and other files:

'color_space'*: 'LUV', # Can be RGB, HSV, LUV, HLS, YUV, YCrCb*
'orient': 18, # HOG orientations
'pix_per_cell': 16, # HOG pixels per cell
'cell_per_block': 3, # HOG cells per block
'hog_channel': 0, # Can be 0, 1, 2, or "ALL"

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 train the model once and save the trained model and the scaler in a Pickle called SVCmodel.pkl. The trained model is then loaded from the pickle when processing the video.

Although I can achieve higher accuracy by using all the channels, I decided to reduce the processing overhead by using only LUV channel 0 in an effort to achieve real-time performance.

But first …

Training the model is done in train_model.py. Before training the model I also performed similar parameter space exploration for spatially binned features and histogram features. The Excel file that I used to track this process is results.xlsx

Augmenting the Dataset

In order to help the model generalize I augmented it by flipping each of the images horizontally, after normalizing them. A car photographed from the right looks just like a car photographed from the left when the image is flipped, so this effectively doubled my training data.

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 sliding window is implemented in windows.py. I realized that although the training set does have some variety in the orientation and distance of the car from the camera it is relatively limited. For example:

Far

Left

Middle Close

Right

So the apparent size of the car in a sliding window should match as closely as possible the apparent sizes of the cars in the training data. The cars in the training data mostly fill the image, and often the car extends beyond the edges of the image.

I also realized that depending on the pixel coordinates when the sliding window is placed on the image it will need to be different sizes. In the locations when a car is seen in the distance the window should be smaller, and where the car is close, especially off to one side, the window should be larger.

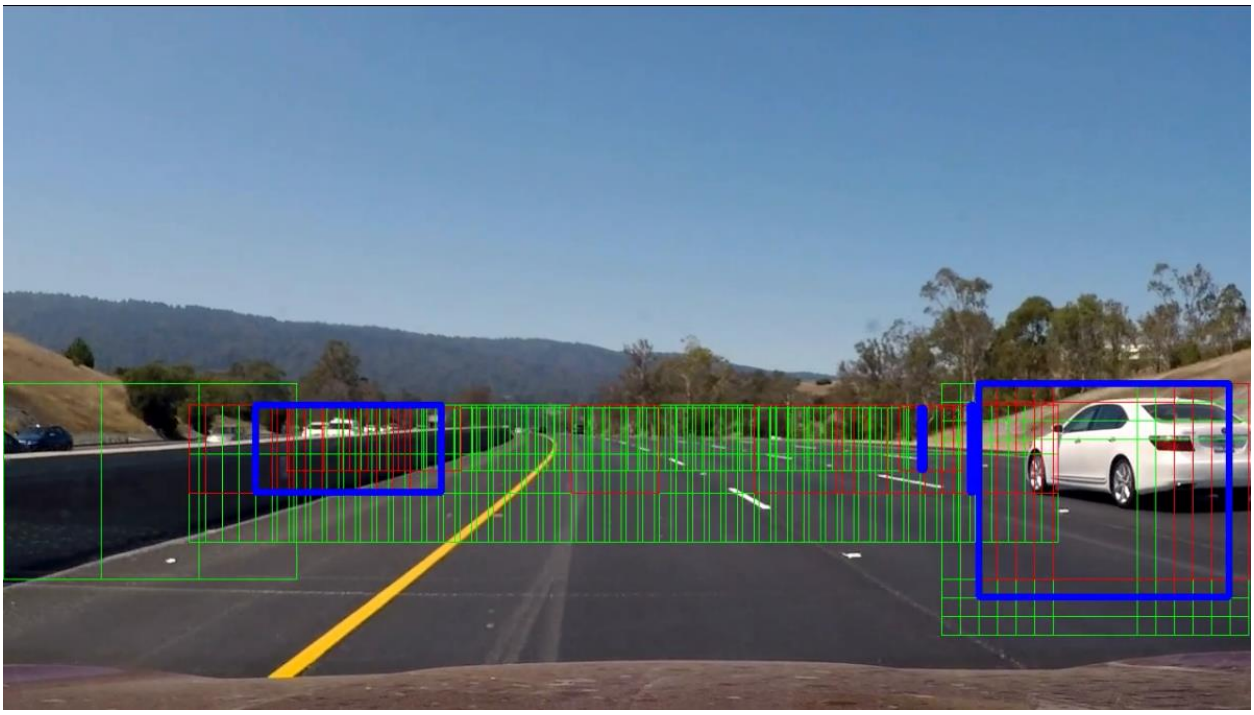
So I decided to create groups of windows of different sizes, tuning each group independently. For example, I needed a group of large windows off to the sides:

Similarly, while running tests with the video I displayed the sliding windows using different colors:

- All sliding windows (green)
- Windows that had some activation (red)
- Windows with a heat signature that exceeded my threshold (thick blue)

To see this technique in use, see project_video_result-tuning.mp4 in the repo, or at:

https://drive.google.com/open?id=0B_xVDdG99gPeS3dEbk5CQmRBN28



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?**

These examples are taken from the video. Notice that the cars in the oncoming traffic are also detected. At first I thought these were false positives, but I realized that oncoming traffic could be important in certain situations, like an undivided highway, so detecting them was a good thing.

Here's another example, showing the black car. The black car was more difficult to detect than the white car:

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 available in my repo, and at:

https://drive.google.com/open?id=0B_xVDdG99gPedDV0Ul9JTWN2NWM

Dropping Frames

The video is 25 fps, which I realized is more than I need. When processing all frames of the video I could not keep up with real time processing, so I decided to drop frames to reduce the amount of processing required. I drop 10 frames (less than half a second) for every frame that I process. The result is that I am able to keep up with the video in real time.

Heat Maps

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 add the activations for sliding windows to a heat map in add_heat(), lines 5-13 in heat.py. Areas on the heat map accumulate heat as different sliding windows add to the same pixels.

In temperature(), lines 36-45 of heat.py, I keep a memory of heat maps. New maps are added to the memory each time a frame is processed, and older maps are forgotten after a certain number of frames. The contents of the memory is then summed to create the current aggregated heat map.

The aggregated heat map is then thresholded and label() is used to group areas of heat in lines 98-99 of main.py

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?**

My pipeline gets more false positives than I would like, and sometimes loses the black car. This is partially due to my efforts to reduce the processing required so that it can keep up with the video in real time. Most of the false positives are near an actual car, so while they don't look clean I feel that they should be acceptable.

The bounding boxes are sometimes too large, and sometimes include both cars when the individual cars could be separated. I'm not sure that's too much of a problem though, since having the car look bigger than it is would be erring on the side of safety, and having two cars that are close appear to be one large car is probably nearly equivalent in terms of vehicle avoidance.

Real Time Processing

I am able to process this 50 second video in 50 seconds! I'm really jazzed about that.

Potential Improvements

I'd like to try lowering the heat threshold somewhat, and then using the activations as hypotheses. I would feed the bounding box of the activations into a second iteration of the model to confirm that they contain a vehicle, and reject more false positives. I think the same model could be used, but the sliding windows might be structured a bit differently since we could absorb more false positives on the initial pass.

It would be interesting to see that the processing overhead would be for a convolutional neural network approach, although the SVC gets 98% accuracy.