SDCND Vehicle Detection

How To Run

Run Training

To run training on the dataset, just run train.py

Run Classification on Images

Ensure that test_image is set to True in line 25 in classify.py. Then just run classify.py from the command line.

Run Classification on Video

Ensure that test_image is set to False in line 25 in classify.py. Then just run classify.py from the command line.

Debug Mode

In order to see debug text and image annotations, ensure that debug_mode is set to True in line 27 in classify.pl. Then just run as before.

Rubric Points

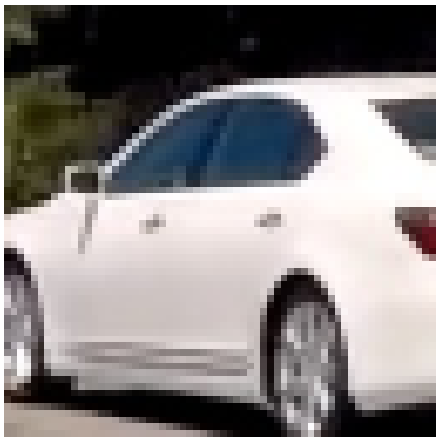
Here I will consider the rubric points individually and describe how I addressed each point in my implementation.

Histogram of Oriented Gradients (HOG)

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

I created a Features class (Features.py) where I handle all my feature extraction. The get_hog_features() function is where I generate the hog features.

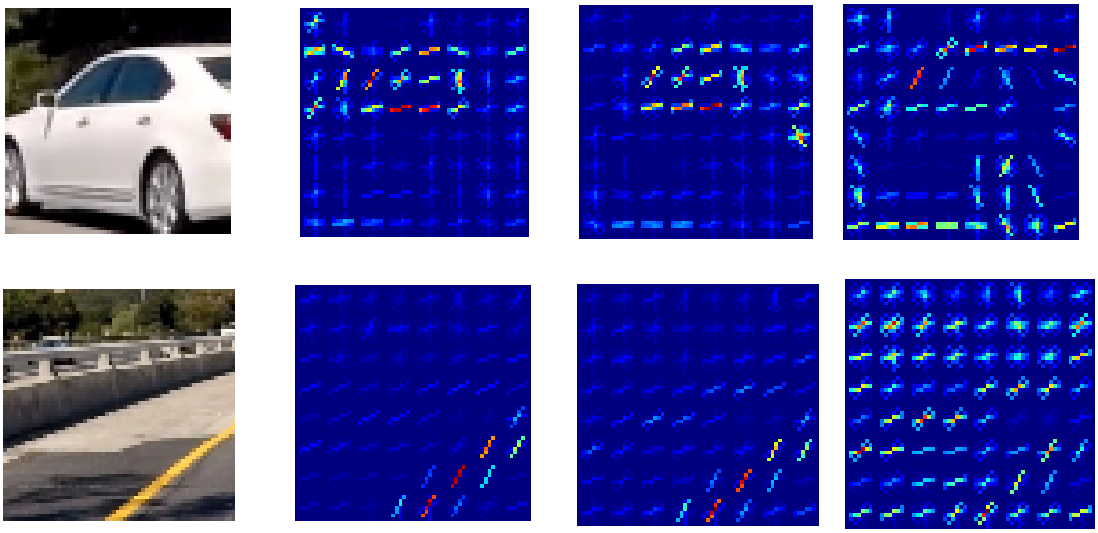
When I am running training, I read in all the vehicle and non-vehicle images. Here are an example of each:




Once the images are read in, I create training and test sets as usual. The features that I decided to use were HOG features from the YcrCb color space and also a histogram of each color channel.

To extract the HOG features, the extract_features() function first converts to YcrCb color space, then extracts the histogram features for each channel with the color_hist() function, and then extracts the hog features for each channel using the get_hog_features() function. It then concatenates these two features lists into a single feature list that is used for training.

The HOG features for the Y, Cr and Cb channels for the images above look like the following:



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

I spent a lot of time up front exploring different color spaces to determine which would give the best results in terms of prediction accuracy. I finally settled on the YcrCb color space, as this gave high test set accuracy scores with my Support Vector Classifier and also gave the fewest false positives in the test videos.

I also tried various combinations for the hog() parameters and found that the following worked well:

orientations = 9

pixels_per_cell = (8, 8)

cells_per_block = (2, 2)

For the histogram features, I found that a it worked best with 32 bins and a hist_range of (0,256).

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).

The code for training my classifier is all in train.py. I use a Linear Support Vector Machine for classification and with the feature set mentioned above, it achieves 99.1% test accuracy:

Training Linear SVC Classifier...
22.919726848602295 Seconds to train SVC...
Train Accuracy of SVC = 1.0
Test Accuracy of SVC = 0.991598915989

Once the features have been extracted, I shuffle it and split the dataset into training and test sets with a 80/20% split. I regularize the data using a StandardScalar. I then train a LinearSVC to create a predition model.

In addition to the LinearSVC, and I use a CalibratedClassifierCV in order to be able to extract classification probabilities from the predictions. For example, when I run one of the data items through the model, I can now both predict the class (car or not-car) and also get an associated probability for the prediction:

0.0008509159088134766 Seconds to predict with SVC
Prediction [0.]
Prob [[9.99978197e-01 2.18032708e-05]]

Here you can see that the LinearSVC predicted that this item was not a car, and it is 99.99% certain of that prediction. I use this probability later on to help exclude false positives by rejecting predictions below a probability threshold.

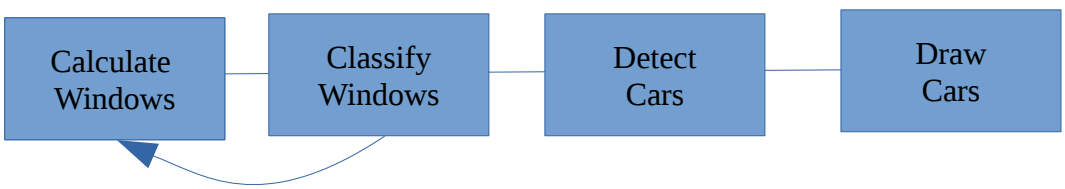
Finally, I save the StandardScalar, the LinearSVC and then CalibratedClassifierCV for re-use in the classifier code.

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?

My sliding window code is implemented in classify.py.

My main pipeline, process_image() is very straight-forward. It looks like the following:



The first two steps are run three times, for three different window sizes. I decided to check boxes of 256x256. 128X128 and 64x64 pixels from the source frame to see if they were cars or not.

Calculate Windows

The slide_window() function returns a list of windows to check later with classification. It takes in a source region in the input image, a window size, and an overlap amount and it returns a list of windows that match the inputs parameters. As mentioned above, I chose three scales to seach at, all multiples of the original dataset images of 64x64 pixels. As the input video images are 1280x720, I looked at what pixel sizes vehicles were likely to be in the images, and these sizes of 356x256, 128x128 and 64x64 seemed appropriate to capture vehicles in a reasonable locality to our car.

I also narrow the search space to regions where cars are likely to appear in the video frame. For example, this means that I never search in the sky for a car. For the boxes of 256x256, I pretty much search in the botto half of the frame. For boxes of 64x64, I narrow the search to a region like (300,380) to (1000,480), as cars this small will not appear outside this region. This helps to minimize the number of windows we classify later.

There is obviously a performance/accuracy trade-off to choosing more window sizes. The results I got from the choices above appeared reasonable for this excercise.

I also chose a 90% overlap. Again there is a performance/accuracy trade-off to be made here. If the overlap is too little, then cars are not detected in many frames as they do not ‘fit’ the windows that we are checking. If you choose to high of an overlap, then the amount of sliding windows that we ultimately classify increases and so the time to process the video increases. I found that 90% gace a good balance of processing time versus accuracy of result.

Classify Windows

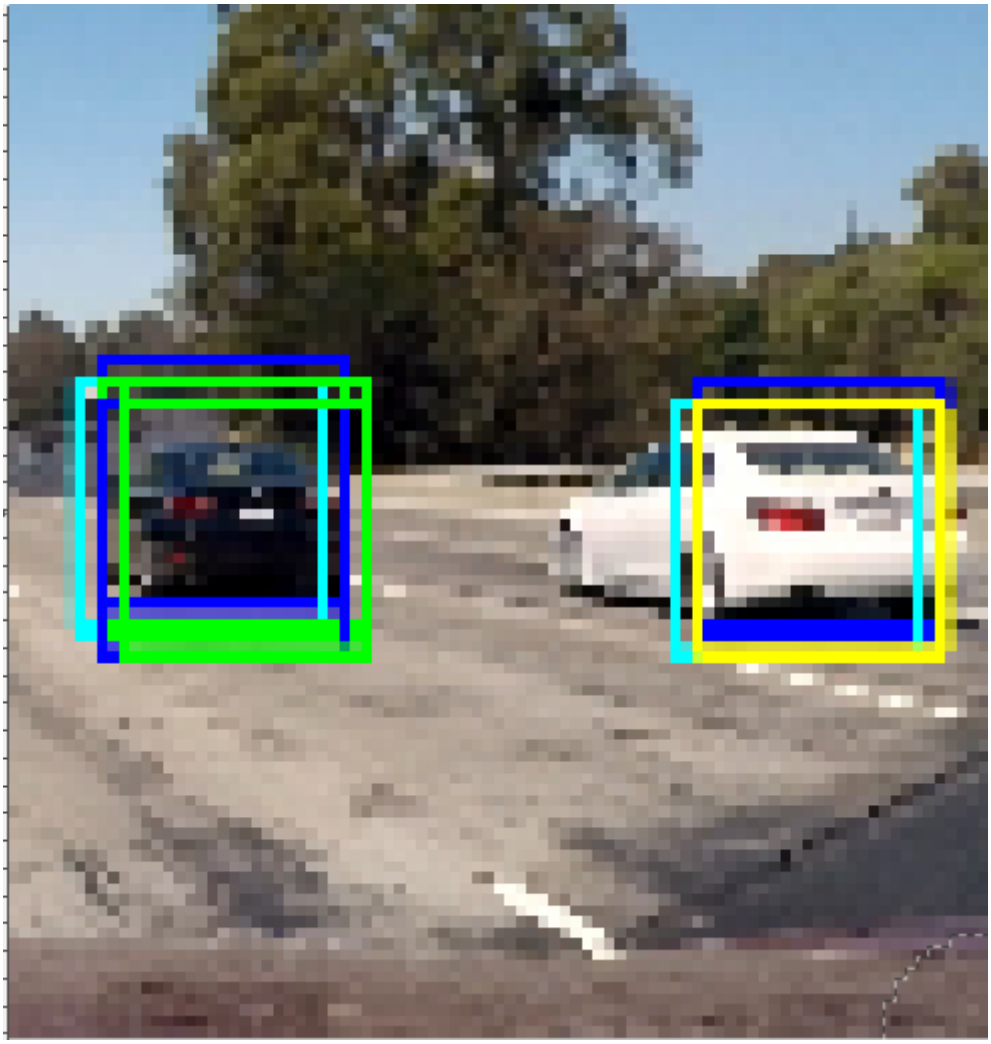
The classify_boxes() function is used to cycles through the window calculated above, and to run classification on them. Each box represents an area of the input frame. The first thing I do is to resize each box to 64x64 to match the feature input size of the classifier. I then use the Features class to extract the features from this image, which are the YcrCb HOG features and the YcrCb histogram as mentioned earlier. These features are then normalized using the StandardScalar, which is loaded from the training stage.

Now we run prediction on this feature list using the model that we trained earlier. If the model predicts this to be a car, we then look at the prediction probability using the CalibratedClassifierCV predict_proba() function. I added the box coordinates and the prediction probabilty to a list, which is the list of possible car locations. This list will be processed later by the Cars class.

Detect Cars

I created a Cars class to assist with determining where cars are and tracking positions from frame to frame. The first function, detect_from_possible_windows(), iterates through the list of possible car locations from the previous step. It calls resolve_bbox_overlaps() to reduce the set of possible car locations by ‘shrink-wrapping’ any windows that overlap. This function uses a very simple overlap check between each bbox in the possible car locations list, and if two boxes overlap, it replace them

with a larger box than encompasses both. For example, the images below show the overlapping bboxes before shrink wrap:



The output from this function is a list of car locations of the format

((car_bbox), bbox_center, confidence_score, frames_tracked, frames_lost), ...)

In this list, you can see I track the bbox and bbox center for each car, as well as the maximum confidence score for all of the original potential car locations that are encompassed by this bbox. It also has two other member which will be used later, the number of successive frames tracked and the number of successive frames lost.

Once the car locations list has been determined, I pass it on to the track_cars() function. This function has the role of tracking car locations from frame to frame. It compares car locations in the current frame to those that have been tracked, and uses this to filter out false positives.

First, it checks the list of current frame car locations against those that have been tracked. It checks to see if the center of the current car is in the same locality as any of the cars tracked from previous frames. If it is, then it considers the current car a match, and updates the tracked car bbox and confidence scores to match the current car. It also increments the successive frames tracked for this

car. If the new car does not match the tracked cars, it adds it as a new tracked car and sets the successive tracked frames to 1.

It then check to see if any of the previously tracked cars have not been matched with the car lcoations from the current frame. If they are not matched, it increments the successive frames lost value for this tracked car. If the successive frames lost value if 5 or greater, it then considers this car lost and removes if from the tracked car list.

At this stage, we now have a list of updated tracked cars which is the same format as the car locations list:

((car_bbox), bbox_center, confidence_score, frames_tracked, frames_lost), ...)

If you run in debug mode, you can clearly see this process happening, e.g.:

Adding 3 cars to empty tracked list
[(((1032, 380), (1268, 532)), (1150, 456), 0.99798673725855835, 1, 0), (((420, 428), (560, 556)), (490, 492), 0.6143180784323872 8, 1, 0), (((804, 380), (980, 556)), (892, 468), 0.99925371299047916, 1, 0)]

Draw Cars

The final stage of the pipeline draws the bboxes around the cars that are tracked. This iterates through the tracked car list output from the Detect Cars stage. If we are not running in debug mode, it only draws a car bbox if the following criteria are met

1/ Successive frames tracked is greater than a threshold (5)

2/ Confidence score is greater than a threshold (80%)

This helps eliminate false positivies but ignoring any car that has not been tracked for at least 5 frames and cars that do not have prediction confidence scores >= 80%.

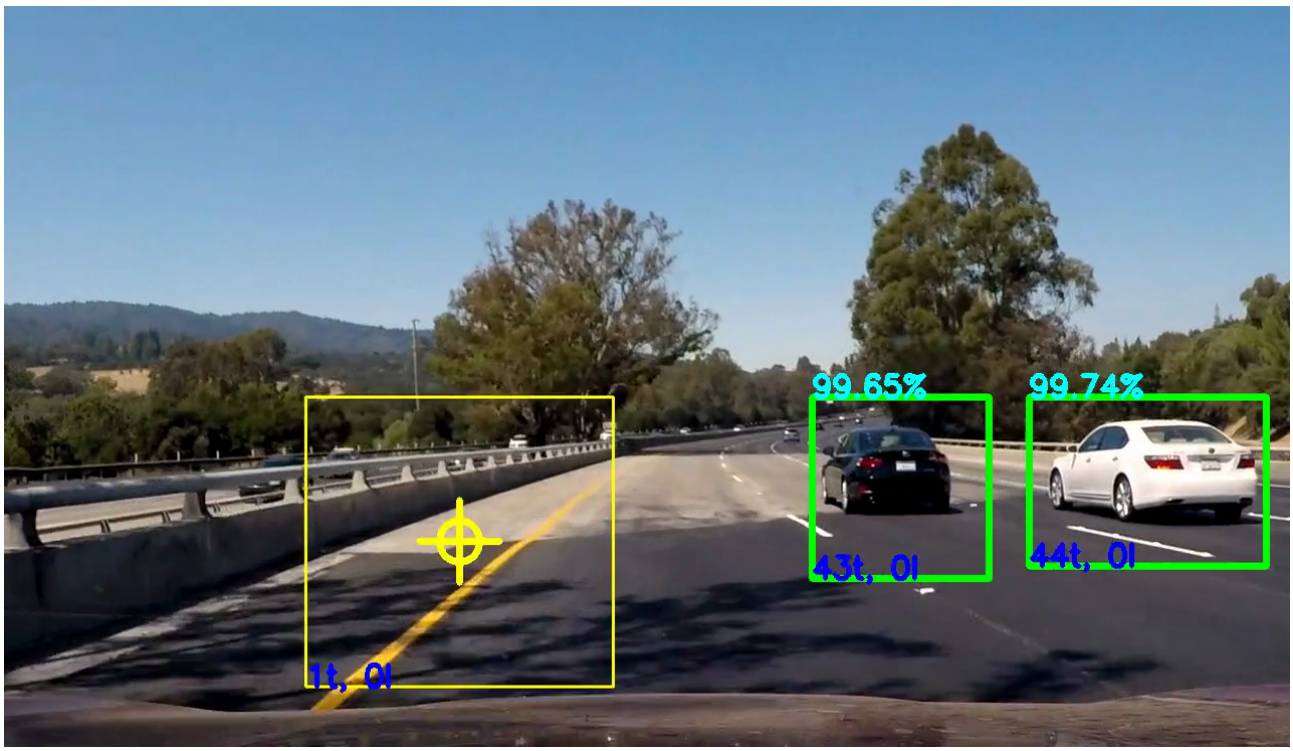
In debug mode, a lot more is output to the screen. It draws a bbox for every car tracked, regardless of confidence score or how many successive frames are tracked.

You can see examples of both below.

2. 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?

I have discussed the steps I go through to minimize false positives and reliably detect cars above.

This is an example image of the debug_mode output:

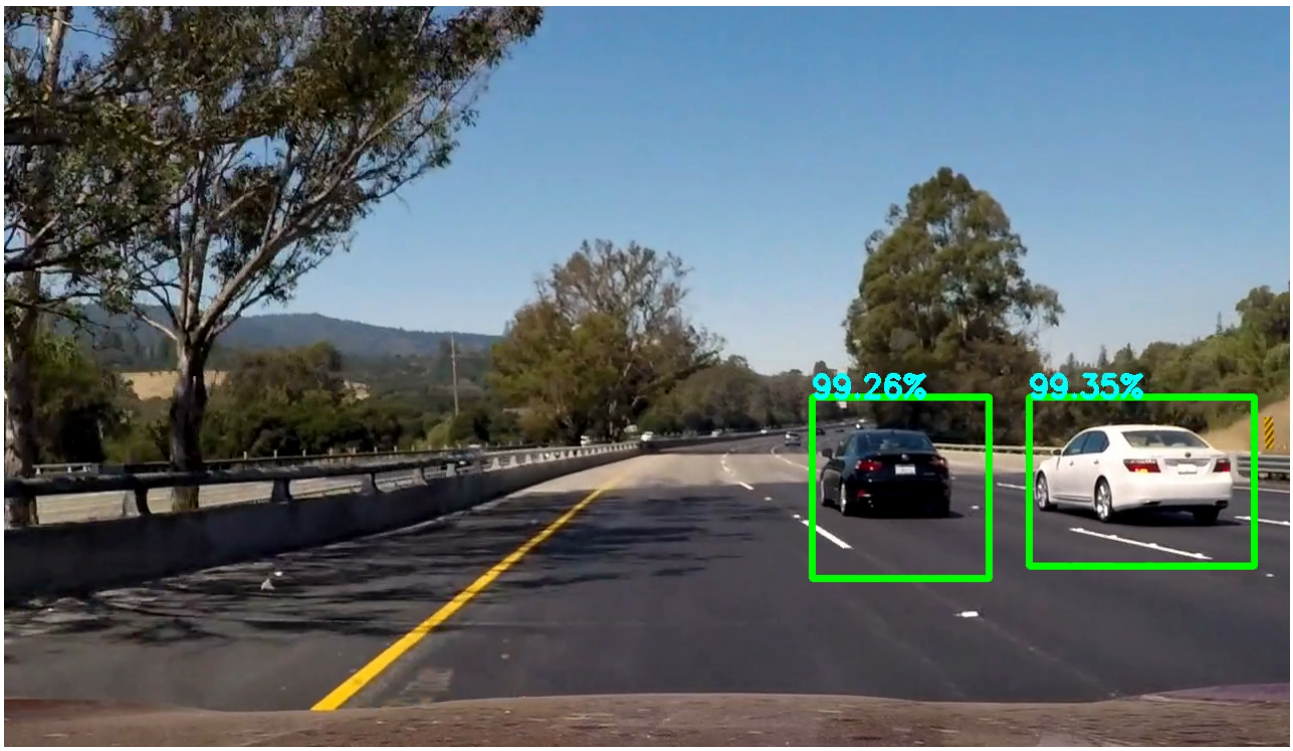


Here you can see a good example of everything working. Each box shows debug text on top which shows the prediction confidence, and on the bottom it shows the successive frames tracked ‘t’ and the successive frames lost ‘l’.

The green boxes are cars that meet the tracked criteria of >5 frames tracked and >80% confidence. You can see these ‘shrink-wrapped’ boxes do a reasonable job of encompassing the cars.

The yellow box is a box that does not meet the tracked critera. In the frame above, you can see that this box has only been tracked for one frame, and so in the non-debug output this would be discarded. If the model predicted cars in this locality for 5 or more successive frames with confidence >80%, then this would turn into a green box. However, this is unlikely to happen and chances are this box would never be drawn in the final output.

The same frame above without debug_mode on would look something like this:



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.)

I'm providing links to two videos, the debug version and the non-debug (final) version:

Debug Video

https://github.com/PaulHeraty/CarDetectionCV/blob/master/project_video_processed_debug.mp4

Final Video

https://github.com/PaulHeraty/CarDetectionCV/blob/master/project_video_processed.mp4

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 have described this already above.

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?

The main issues with the pipeline that I have is speed. It takes around 3 seconds to process each frame currently. The first thing I would try is to implement the same with a deep learning approach in order to increase the speed.

I would also look to see if any of the manual implementations I made for window merging or window tracking have OpenCV functions or similar. I wanted to have full control over how this worked, but perhaps there are faster implementations available that I could look to use.

The pipeline would probably fail in situations where our car was going up or down steep hills, as the regions where I look for cars are hard-coded and would ‘move’ if the road was not relatively flat. In these cases, I’d have to add code to determine where the horizon was, and then calculate the appropriate regions in which to conduct a sliding window search.