Vehicle Detection Project

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 classifier 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.

Histogram of Oriented Gradients (HOG)

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

Some of the code for this project has been taken from the project lessons and refactored modified. Especially, the code has been made more modular by refactoring into different Python modules so that they can be more easily tested and experimented with through Python unit tests.

The code for the feature extraction is in *lesson_functions.py*
It is invoked both in the learning/training phase as well as during the detection phase.

The training is invoked through ztest_classtrain.py It invokes class_train.train, which sets the following sequence in action:

class_train.train:
Invoke class_load.load which simply reads the vehicle and non-vehicle images from two different sub-directories and returns a tuple of the lists

1. Feature extraction is performed on both lists
2. Data is split into training and test data (20% test data)
3. A Scaler and a LinearSVC is instantiated and being trained
4. The resulting scaler and svc configurations are saved through pickle on disc.

Note that the last step allows us to separate the training phase and simple reload the SVC in detection phase, reducing turnaround times in development.

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

Some experimentation with the parameters of the feature extraction were done, but it turned out that the original parameters were quite reasonable.

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 trained a linear SVM using many features.
To be able to reuse the configuration both in training and in detection, the parameters were factored to a python file cnst.py. The relevant configurations are:

```
y_start_stop = [400, 656] # Min and max in y to search in slide_window()
color_space = 'RGB' # Can be RGB, HSV, LUV, HLS, YUV, YCrCb
orient = 9  # HOG orientations
pix_per_cell = 8 # HOG pixels per cell
cell_per_block = 2 # HOG cells per block
hog_channel = 'ALL' #0 # Can be 0, 1, 2, or "ALL"
spatial_size = (16, 16) # Spatial binning dimensions
hist_bins = 16    # Number of histogram bins
spatial_feat = True # Spatial features on or off
hist_feat = True # Histogram features on or off
hog_feat = True # HOG features on or off
```

Surprisingly, the RGB color space werked reasonably well. It turned out,
That using „hog_channel“ ALL obviously is a necessary setting, because other
Parameters would be ignored otherwise.
But note that this is probably an effect of the given video, since we do not have fancy other colorful cars in the video, which might be more sensitive to this approach.

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?

Sliding windows is invoked from hot_windows.multi_hot_wins. The actual definition of the sizes is again in cnst.py and is like this.

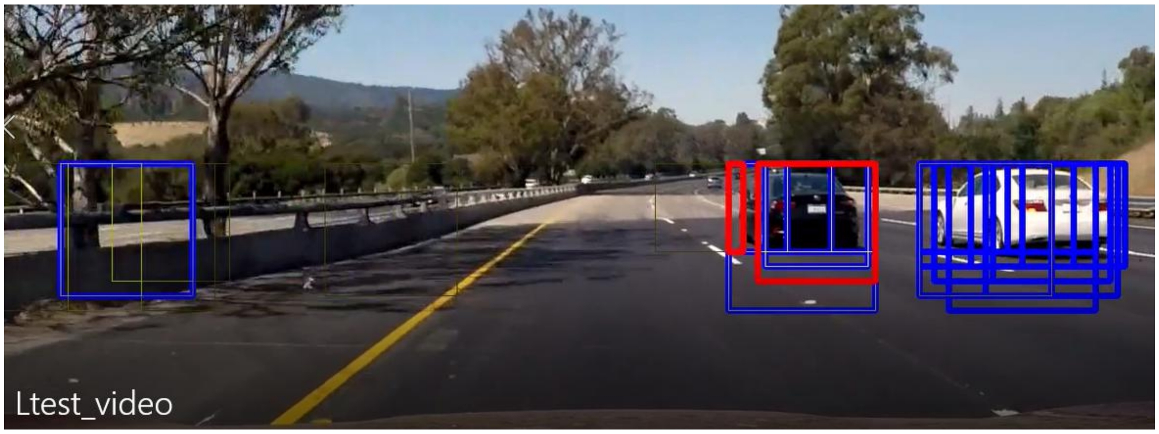
```
sizes = [160,144,128,112,96]
```

Note that:

- Windows are from biggest to smallest. That could be used for an optional optimization, where we ignore areas that have been positively identified as a car for a large window.
- Biggest window size has been estimated by looking at the size of the cars in the video as they come into sight.

- Smallest window size is size of the training windows. That seems to cover a reasonable are of distance before the vehicle. Smaller sizes would need upscaling and these small images might not provide additional data).

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?



The test image above shows the sliding windows, the filter for false positive (heatmapping) and the final image. It shows:

- Thick blue rectangles: Areas where the sliding window had a hit:
- Small wight/greyish lines: These are from my custom heatmapping routine (see below)
- Thick fat red rectangles: Final detections.

As an optimization, only the relevant area, where cars could be occur (lower half of image, but above the car hood).

Video Implementation

1. Provide a link to your final video output.

The final video is in unit_test/Lproject_video.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.

After reading the lesson’s explanations, I thought that rendering the heatmaps and then again deducting rectangles from that was quite an effort, since all the way we are dealing with rectangles anyway so I decided to try an approach, that works solely with rectangles as data structures. This is implemented in hotspots.Hotspots.

The basic idea is:

- Provide a „ring buffer“ for the last n frames. Each buffer element contains the list of positive rectangles from the window search of the last n frames.
- In addition, we have a list of lists, containing „heat“ lists.
- For each frame, do the following
  - Perform the window search
  - Push the list of rectangles to the ring buffer
  - Now process the ring buffer. For each list, check all rectangles. Check if any rectangle intersects with any in heat[0]. If so, calculate the list of intersections. After that, add all rectangles to heat[0].
  - Now, for all intersections, check if any of those intersects with rectangles in heat[1]. If so, calculate the list of intersections, add all rectangles to heat[2]
  - Repeat for heat[2] to heat[x], where heat x is the level of heat that we want for a „hit“

Add the end, we want the largest rectangles from heat[x], for that: Find all groups of intersecting rectangles in heat[x] and join them to individual rectangles.
Finally draw them as red :)

Discussion

- Initially, I encountered problems, because the color spaces of training and detection were different, which is a subtle error.
- Performance is still slow, current implementation could not reasonably run on an embedded system.
- The approach is obviously only sufficient for cars driving in roughly the same direction as the ego car, cars crossing the road could not be detected.
- Training data does not seem to contain commercial vehicles, military vehicles, offroad-vehicles etc. So that any of those might not be reasonably detected.
- Much of further optimization etc. Would be dependent on the use case / function associated with the vehicle detection and might include some other technology, e.g.
  - The image recognition could be combine with LIDAR/radar to
    - Focus only on the areas, where an obstacle is detected (increasing performance)
    - Validate the results of the image recognition

-