Udacity SDCND | Vehicle Detection and Tracking | Project V

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

Writeup / README

Provide a Writeup / README that includes all the rubric points and how you addressed each one. You can submit your writeup as markdown or pdf. Here is a template writeup for this project you can use as a guide and a starting point.

This is the writeup for Project 5: Vehicle Detection and Tracking

Please also notice other **submitted files** (extract P5.zip):

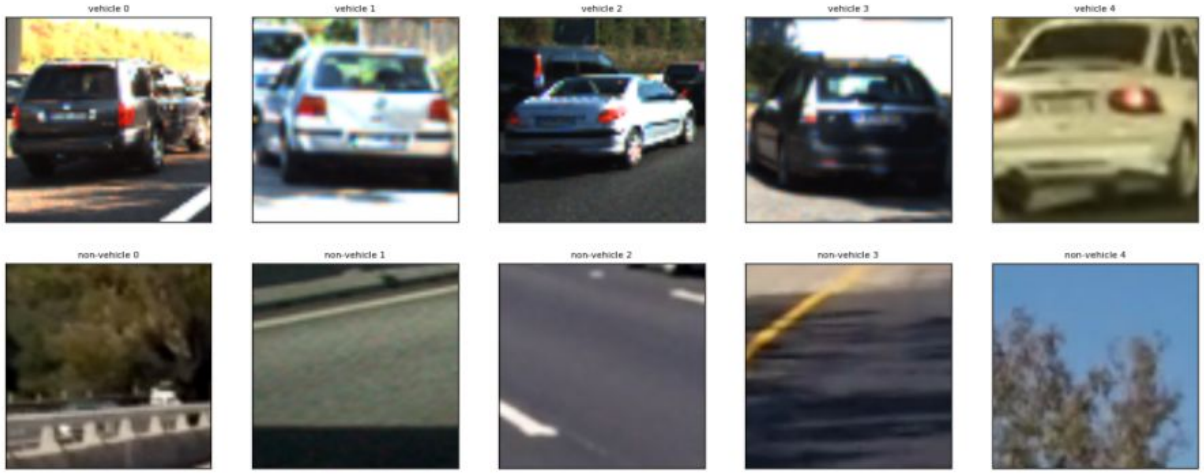
- project5.ipynb — project ipython notebook file (executable)
- project5.html — exported HTML file from notebook execute
- final_model.sav — pickle(v3) dump of trained LinearSVC model
- output_images/[Non-]Car-*-HOG.png — HOG feature image of a sample's 1 channel
- output_images/roi.mp4 — debug video to gauge ROIs for best catches
- output_images/SlidingWindowsGauge.png — example image for ROIs tuning (video cap)
- output_images/swin_catch*.png — examples showing object catch in diff ROIs
- output_images/test1-random-sliding*.jpg — example showing random sliding windows
- output_images/sliding*search-predict.jpg — example showing clf.predict() search
- output_images/sliding*decision_function.jpg — visualize of clf.decision_function() search
- output_images/test*-threshed-search.jpg — threshed decision_function search example
- output_images/test1-optimal-search.jpg — speedup search from hog() call optimization
- output_images/test1-heatmap-*-thresh.jpg — examples showing heatmap threshing
- output_images/test1-labels.jpg — example showing detected objects label pct.
- output_images/test1-bboxes.jpg — example showing bounding box of detection
- output_images/project5.mp4 — processed video from 'project_video.mp4'

Histogram of Oriented Gradients

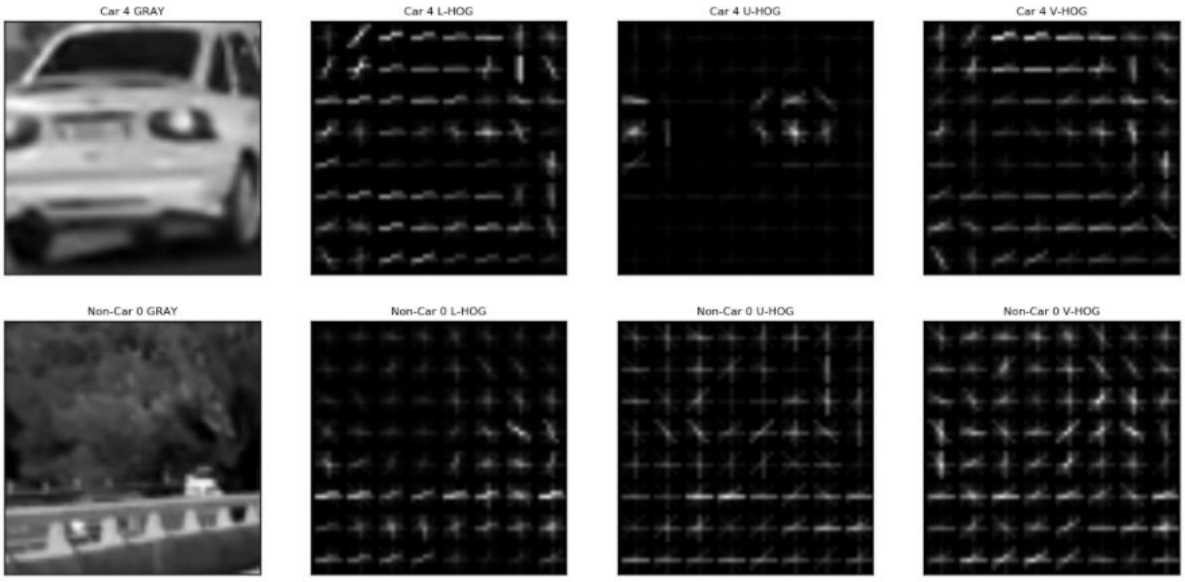
1. Explain how (where in your code) you extracted HOG features from the training images

The code for extracting HOG features to train my model is located in code cell **In [6]:** of **3.3.2 Model Training and Validation** of IPython notebook (project5.ipynb), where **extract_features()** in code **cell In [2]:** did the real job given my final HOG parameters. I explored HOG parameters and color channels selection before all these.

In code cell In [4]: of **2.2 Data Set Visualization** I first started by picking 5 random samples from each of vehicles and non-vehicles classes:



Then in code cell In [5]: of **3.1 HOG Feature Extraction**, I started use LUV color space, since it provides a best empirical classification accuracy in my lecture, and it works well to keep invariance of changes in lighting condition. I then plotted HOG images of each **L / U / V** channels side-by-side for these 5 random vehicles and non-vehicles, visualization tells that using HOG features from **L** channel itself is sufficient, with an motivation not to increase feature vector too big (for the same reason & no obvious benefit I didn't include extra bin_spatial and color_histogram in my final feature selection). Here is a part of that plot:



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

I further explored different skimage.hog() parameters (orientations, pixels_per_cell and cells_per_block) by using both HOG image visualization and classification testing score from course lecture (22.**27 HOG Classify**), and conclude following parameters as final in section **3.2 HOG Parameters**:

`LUV space: L channel, orientations=12, pixels_per_cell=(8, 8), cells_per_block=(2, 2)`

*N.B. **`orient=12`** provides more detailed spatial properties; **`pixels_per_cell`** and **`cells_per_block`** values fit sliding windows search optimization in section 4.4.2 later.*

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

First I explored dataset properties in section **2. Data Set & Exploratory Data Analysis,** there I sensed that we are at risk of overfitting with not bigger enough dataset (~9k samples to each class, which minimum features are at ~2k range), so the best solution would be get in more data. But under time constraint I have to stick with project dataset Udacity provided, this force me keep close eye on the feature counts:

`Current features vector length = 7 x 7 x 2 x 2 x 12 = 2352`

Given features count of 2352, and project dataset size (< ~18k), I'm reluctant to add any more features!

Next I chose to use **sklearn.svm.LinearSVC** model to build vehicle classifier, for a quick prototype I used all parameters by default value. I also used **sklearn.model_selection.train_test_split()** to split dataset to training set (*80%*) and validation set (*20%*) to train model, I could run cross validation or GridCV.Search to a better result (Accuracy, Precision or F1 score), but I left this to future practice.

Data normalization (0 mean, and unit variance) is also done using **sklearn StandardScaler()** prior to training the model. All these code is located in code cell **In [6]:** of **3.3.2 Model Training and Validation.** With simple settings, we got a simple evaluation matrix of: **Validation Accuracy of SVC = 0.951**

Which is yet a super high accuracy, but good enough to proceed.

***N.B.** Pursuing bigger dataset must be precedent over higher accuracy at this stage!*

After training, I stored model as **'final_model.sav'** in current path for future usage, e.g. in video pipeline to predict a sliding window for vehicle detection.

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?

A Region of Interest (ROI) is first to consider to lower false positive (noise) and increase search performance. Instead of coming up ROI(s) hypothetically, I did experimental gauging using project video (please refer to ***`output_images/roi.mp4`***) , there I used 3 pairs of color lines to mark ROIs selection:

Red-pair stripe *(128 pixels height)* represents ROI for sliding search windows of size 64x64 or 128x128
Green-pair stripe *(192 pixels height)* represents ROI for sliding search windows of size 192x192
Blue-pair stripe *(256 pixels height)* represents ROI for sliding search windows of size 256x256
All three stripes start at the same ***`y_min=384,`*** *`going downward.`*

`roi.mp4` *`proves this scheme yields best catch rate without much extra complexities, e.g. adding restriction to X-Axis as well.`* Section **4.1 Sliding Windows Gauge and ROI** and **Figure 4.1 ROIs Planning** provide all the detail, and section **4.2 Sliding Window ROIs Demonstration** showed good catching probability from these ROIs considering the sliding window sizes each of them serves.

Code cell of **[12]**: in **4.3.1 Sliding Window Selections** defines sliding windows specs. As following::

sliding window sizeof 64x64: overlap_rate = 0.75, or stride = 16 pixels

sliding window sizeof 128x128: overlap_rate = 0.75, or stride = 32 pixels

sliding window sizeof 192x192: overlap_rate = 0.5, or stride = 96 pixels

sliding window sizeof 256x256: overlap_rate = 0.5, or stride = 128 pixels

smaller window with smaller stride is generally suited to track farther away object, likewise

bigger window with larger stride is generally suited to track closer object but it is tunable

***N.B.** increase overlap_rate may help with final video output, but I haven't seen that obvious change by setting overlap_rate to 0.75 for windows_192 and windows_256 for example. Overlap_rate can be any percentage, but it is more convenient to keep its corresponding stride value a number of **block W/H (16, from HOG setting)** for an **optimal search (4.4.2 Optimal solution** Extract HOG features just once for the entire ROIs in each image)*.

Figure 4.3.2 demonstrates some random sliding windows according to their ROIs searching scheme:

Figure 4.3.2 Some random sliding windows on each ROI

Sliding windows at largest scale of 256x256 are used to capture nearest vehicles, while 64x64 are used to capture farthest objects, with every other scales in between. Their upper boundaries line up pretty well (all ***Y_min~=384***) from perspective view, so it is **incorrect** simply put smaller sliding windows to search upper part and larger windows to search bottom part of the image! This indicates the importance of ROIs gauging using real video clips (***roi.mp4***).

Section 4.4 Window Search/Classification deals with sliding window search, using two methods with examples.

I have implemented ***TWO*** majorly different sliding window search algorithms, folded in the common function called **`search_window2()`** in code cell **`In[2]:`** of **Section 1. Helpers, Routines.** While user code is split into *TWO* part:

4.4.1 Non-optimal solution **(window-by-window HOG** extraction and classification**)** in code cells **In[14], [15], [16]**

This method **Iterate through all candidate windows (e.g. 442), do following for each:**

- **Resize (to model training image size 64x64), normalize each window (data preprocessing)**
- **Extract HOG Feature Vector from it, with the same HOG parameters used for training data**
- **Scale window image's Feature Vector (0 mean, unit variance), use same StandardScaler from training**
- **Send finalized Feature Vector to pre-trained Model to predict label: vehicle or non-vehicle**

This method is **SLOW** but **reliable** and acting as sanity and performance **baseline.**

4.4.2 Optimal solution (**Extract HOG** features just **once** for the **entire ROIs**) in code cells **In[17], [18]**

This method first extract **entire HOG features** of full **ROI once** an image into **`ROI feature arrays`**, then:

- **For each 64x64 searching window, subsample the ROI feature arrays to get its feature vector directly**
- **For each 128x128 searching window, same technique applies, but using a `secondary ROI feature array`**, check out function **`multi_win_features()` in code cell `In[17]`.**
- 192x192, 256x256 searching windows is handled the old way, without speedup provided their low quantities
- **Speedup 64x64 and 128x128 windows along spared 420 `skimage.hog()` calls per image!**
- Classification / prediction logic is no change, so I reused **`search_window2()`** to wrap up new optimization, with an **optional parameter** `winfeatures` returned by **`multi_win_features()`** to enable optimization.

This method is ***Super FAST!***

- **A** mix of fast 64x64/128x128 windows search and legacy 192x192/256x256 windows search still show ~16 times faster than legacy search in **4.4.1** approach, from my testing. So well done!

`search_window2()` is the common API to either optimal (64x64, 128x128 windows) or legacy (192x192, 256x256 windows) searches, refer to final video pipeline (**6.1 Image Pipeline** in code cell **`In[23]`**) for much a clearer view:

```
def process_image(image):
    #1)
    f64_128 = multi_wind_features(image, roix_start_stop=[0, 1280], roiy_start_stop=[384, 512],
                                  win64=windows_64, win128=windows_128, color_space='LUV', orient=12,
                                  pix_per_cell=8, cell_per_block=2, hog_channel=0)
    #2)
    hot_fast = search_window2(image, win64_128, model, X_scaler, color_space='LUV', orient=12,
                              pix_per_cell=8, cell_per_block=2, hog_channel=0, confidence=3, winfeatures=f64_128)
    #3)
    hot_slow = search_window2(image, win192_256, model, X_scaler, color_space='LUV', orient=12,
                              pix_per_cell=8, cell_per_block=2, hog_channel=0, confidence=3)
    #4)
    heat_map = heatmap(image, hot_fast+hot_slow)

    # continue ….
```

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?

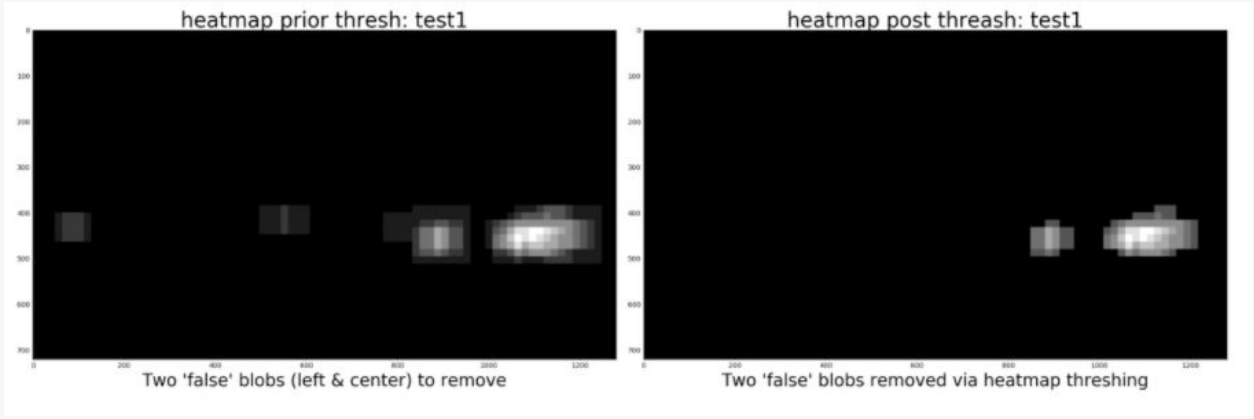
Finally I searched on all 4 scales (64/128/192/256) using LUV L-channel HOG features without spatially binned color nor histograms of color in the feature vector, which provided a nice result.
Here are some examples from STEP1). threshed single image sliding windows search prior to heatmap processing:

Example images above demonstrate results from all scale windows searching using classifier's decision_function() to thresh positive predictions based on confidence score (signed confidence score represents the sample's distance to the classifying hyperplane or class boundaries). confidence score threshold is provided as `confidence` parameter in the search_window2() common API, as in following use case:

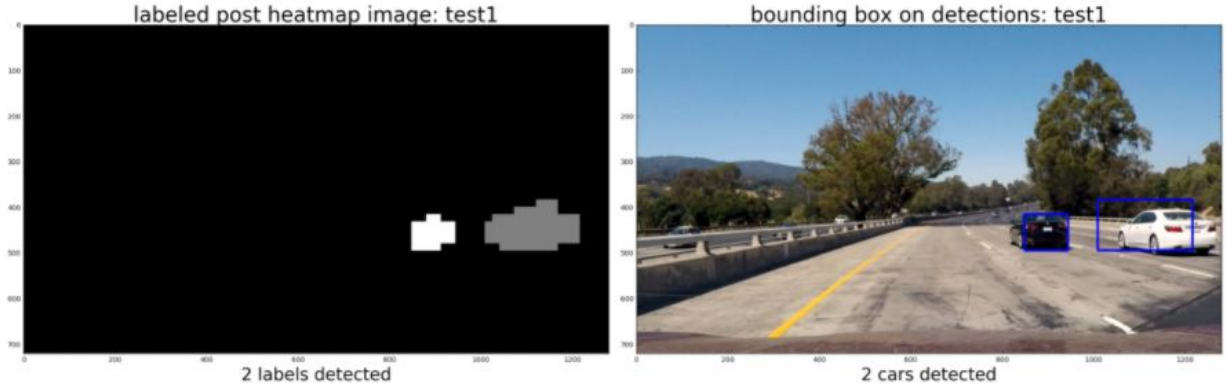
```
    hot_fast = search_window2(image, win64_128, model, X_scaler, color_space='LUV', orient=12,
            pix_per_cell=8, cell_per_block=2, hog_channel=0, confidence=3, winfeatures=f64_128)
    Key parameters:
       model: the pretrained classifier
       X_scaler: the same StandardScaler() from model training
       confidence: threshold of confidence score used to filter model.decision_function()
return
                 OPTIONAL, when provided, fall back to use model.predict() without
threshing.
       winfeatures: prepared HOG feature list matching with win64_128 window list
                 OPTIONAL, when provided, fall back to extract HOG features for every
window.
```

So far following mechanisms have been put in place to less false positive & reliably detect cars at single image level:

- **Deploy reasonable ROIs as spatial filter**
- **Tuning good feature vectors - I have chosen LUV channel L, and orient=12 with more orientation details**
- **Apply tuned confidence score threshold to model to filter 'marginal' positive samples**
- **Lastly applies threshold to single image heatmap to remove lower density detections, see following example pictures:**



As we see, thresh heatmap on a single image basis can help to remove non-confident detections or false positives



Final detections in bounding boxes are much cleaner or reliable!

And I have incorporated single image heatmap threshing in video pipeline as in step ***#5)***:

```
def process_image(image):
  #1)
        ... ...
```

```
#5)
htthresh = heathresh(heat_map, 1)
```

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.)

Implementation is in **Section 6.** Generated project video is located in submission as:

output_images/project5.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 created a **global FIFO** (collections.deque) in code cell `In[24]` of **Section 6.2 Load and Process Video** to queue up heatmap from each individual video frame, tuning has set this **FIFO** maxlen 25, so it can queue up to 25 consecutive frames' heatmap, with new frame queued in, oldest frame get pushed out. So at any given time this deque stored last (up to) 25 frames heatmaps.

At time I am about to construct bounding boxes to each image's augmentation, I just sum up the deque's all (up to 25) heatmaps, which represents an accumulative detections of past period of time, so it naturally weakens false positive (occasional) detections, but strengthens true positive detections (assuming no systematic errors, e.g. a vehicle is always hard to detect in certain area)

Then I apply an **running average threshold** to this summed heatmaps, just like did heatmap threshing on a single image before, but it is with grouping threshold this time. This is done in **`process_image()`** step #6) of code cell `In[23]` in **Section 6.1 Image Pipeline.**

Finally I apply scipy.ndimage.measurements.label() to the threshed heatmaps sum to yield statistic view of bounding boxes (note that label() has automatically dealt with **'overlapping bounding boxes**', it's returns don't have overlapping label areas). This is done in the same utility function **`bounding_boxes()`** in code cell `In[21]` of **Section 5. Heatmap / Bounding Box**

After acquired statistically threshed bounding boxes, I just draw them back to the image right before its return in the main video image handler **`process_image(image)`** in code cell `In[23]` of **Section 6.1 Image Pipeline.**

This running average threshing really helped in removing false positives that are sometime (when it is not a lower density detection at one moment) impossible to be filtered on a single image basis.

Most work are kept in cell `In[23]` of **Section 6.1 Image Pipeline** with a handful clean code!

Here is an example video frame showing on-vehicle bounding boxes from filtered accumulating heatmaps

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?

Primarily I took a straightforward approach to build image pipeline for this project, use HOG features to build supervised learning model to detect vehicles on relatively clean road, compare to complete autonomous driving system, this is already simplified a lot, but the project does provide me some true feeling how much more a real system could be!

In addition to HOG and Scikit-learn ML library modules, I mainly used heatmap threshing, and scipy.ndimage.measurements.label() to filter and retrieve detections, I also implemented FIFO queue as an average smoothing utility. But I haven't gone far ahead to build object tracking class to enhance further, mainly due to time constraint.

I have addressed the sliding windows search performance bottleneck from excessive skimage.hog() calls, and enjoyed the benefits from it - fast iteration, trial and error!

With all these technique: feature tuning, ROI selection, classifier confidence tuning (decision function), single image heatmap filtering, cross frame heatmaps filtering, etc. be deployed, I still see tiny rough spots, this motivates me to improve current design and in the meantime look for a more robust and effective object detection method.

Here are some concerns and thought that I have regarding current implementation:

- *Some false detections still exist, try following to improve*
    - *Evolve from current stateless (passive) tracking, add active object tracking class to find and remove harder false positives.*
    - *Revisit data itself, preferably always use more new data, secondarily use Cross Validation from existing available data*
    - *Revisit model itself for a better baseline, fine tune or GridSearchCV for a better performance. Or just find better learner to job, use R-CNN, DNN, etc.*
    - *Revisit Computer Vision for a better feature map, may combine multiple HOG channels from multiple color space.*

- *Some pure enhancement or nice improvement may be*
    - *Harness 'bird-eye' perspective view (see from top) to help with surrounding vehicles detection, localization and tracking.*
    - *Implement distance vector vs. bounding box to target object for real value.*
    - *Explore DL methods to further cut the complexity of hand crafting rules.*

Thanks for your review and comments!

Hai Xiao