Vehicle Detection Project

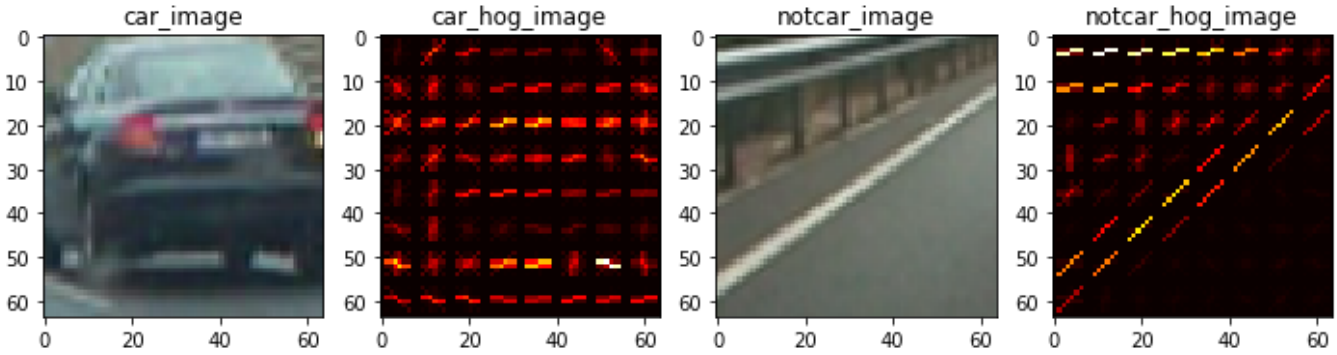
Histogram of Oriented Gradients (HOG)

1. HOG features from the training images.

The code for this step starts in the second cell of the IPython notebook “VehicleDetection.ipynb” with the call of the function ‘*single_img_features’*. This is located in in lines 16 through 33 of the file called *`lesson_functions.py`*.

The first code cell of the IPython notebook read in the training images: 8792 cars and 8968 non-cars.

I then explored different color spaces and different `skimage.hog()` parameters (`orientations`, `pixels_per_cell`, and `cells_per_block`). I grabbed random images from each of the two classes and displayed them to get a feel for what the `skimage.hog()` output looks like.



Beside each images is an example using the `YCrCb` color space and HOG parameters of `orientations=6`, `pixels_per_cell=(8, 8)` and `cells_per_block=(2, 2)`.

2. The final choice of HOG parameters.

I tried various combinations of parameters and choose that ones with the best score after the training.
color_space = 'YCrCb' orient = 9 pix_per_cell = 8 cell_per_block = 2 hog_channel = 'ALL'
spatial_size = (32, 32) hist_bins = 32 spatial_feat = True hist_feat = True hog_feat = True
block_norm = 'L2-Hys'

With the last one I got serious problem with block_norm = 'L2-Hys' in scikit-image version v0.13. The recommended 'L2-Hys' lose about 1% score in accuracy after the training. Therefore I update the libraries to scikit-learn version 0.19.1 and the scikit-image version is 0.14.0.After that the warnings disappeared and the results are on the same level or slightly better than with L1.

3. Training a classifier with my selected HOG features and color features

I trained a SVM using a GridSearchCV function to get the best combination of parameters.
The GridSearchCV are allowed to change parameters of the kernel from either 'linear' or 'rbf' and the parameter 'C' from 1 to 10 (). You can find this in the third cell of my IPython notebook in line 60-64.

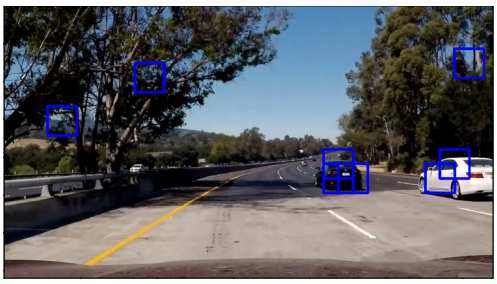
I used the hog and color features for the training . Therefor all features are set as True in line 12 - 14. You can find the code in the *“lesson_functions.py” under “def get_hog_features”, “def color_hist” and “bin_spatial.py”*. This takes a lot of time to train, but after one hour the best results are really fine:

Test Accuracy of clf = 0.0.9994399886 (min 0.9965)

The reason for this manual and automatic choice of parameter is only maximize the this accuracy.

Sliding Window Search

1. Implemented of the sliding window search

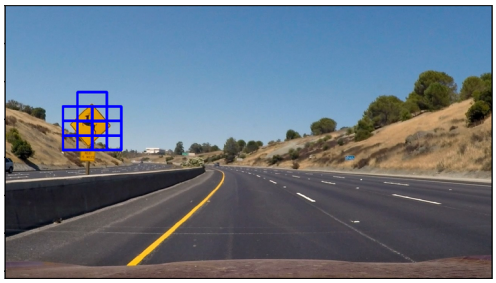


In the next cell of my IPython notebook I figure out the window size and position. Big windows didn't find small cars. Small windows find a lot of car in trees and else where, but not big cars.

In the images beside I search with 544 windows with a size of 77x 77 pixel and 50% overlapping all over the image.

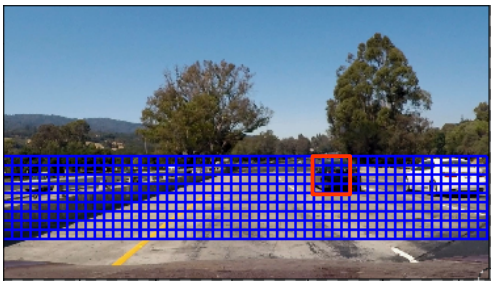
In line 10 you find the normalization of the image.
All color information must be divided by 255 for the hog process. This is mandatory for the process.

Here you can see how wrong the 99,9% classifier can go.
A yellow traffic sign with a deer is clearly no car for me, but all the technology above predict that. Well there are still no features to detect tires or some think like that.
Well, remarkable what hog() can assume to be a car.



In the next cell you find my finale sliding window search code. I choose a scale factor of of 1.4 which are 96x96 pixels in the original image. In the code the window size is always 64 pixel, but the scale factor change the resolution of the image, which makes a 64x64 pixel window bigger (line 24).
Also I change area of search, assuming that there are not really cars in the trees above the horizon (line 6). And the deer – well, traffic signs was classified in an other lesson.

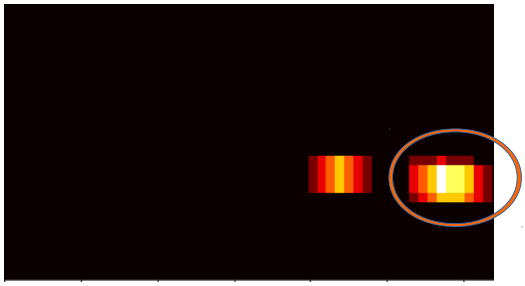
After I set all predictions as true (line 72), it looks like this.==>



Otherwise it looks like this image beside, only the cars are classified with multiply windows of the same size. ==>

In this window you can see also the overlapping. The window with the red frame has 64x64 pixels. It's divided by 8x8 cells with 8x8 pixels. The overlapping of the search window is two cells wide (*def find_cars* line 26). This will result in four rows which are marked with the red Window again. ==>

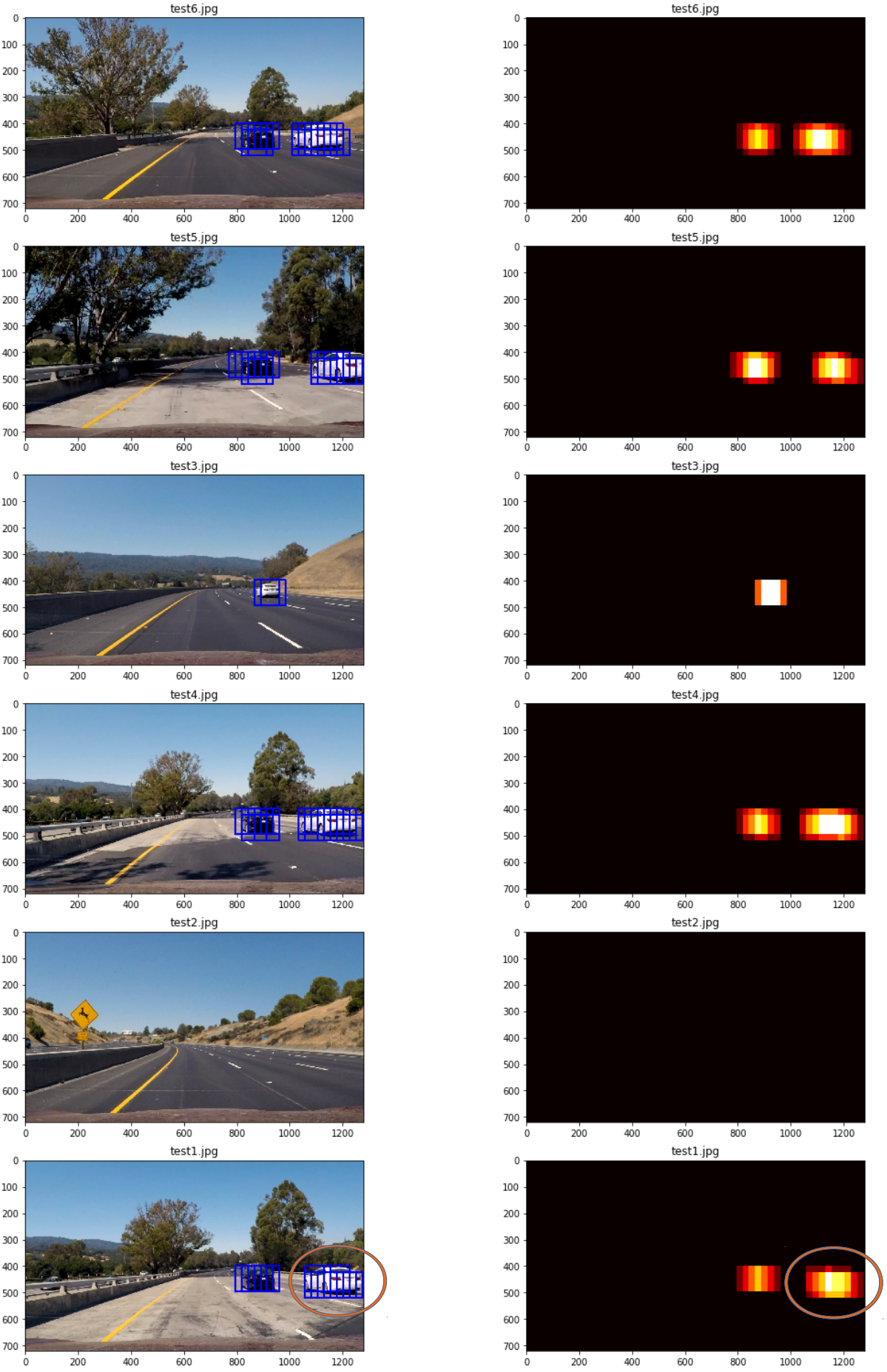
In this cell you can find the heard of the code. It applies the windows technology as described above get the hog features in line 42-56 and also the color features in line 64 -66. Both will concatenate in line 69 and finally in line 70 we predict with our trained classifier, if the pixel of this this window shows us a part of a car or not.



On a positive prediction we draw the frame of the sliding window into a new back image. Of course this can be overlapping. All the prediction will add to a feature map. Because of the color it is named heat map. White means a hight chance for a car and dark red is only represented by one frame and the prediction is low. Here you see the heat-map looks like from the image above: ==>

2. Demonstration of the pipeline

Ultimately I searched on two scales using YCrCb 3-channel HOG features plus spatially binned color and histograms of color in the feature vector, which provided a nice result. Here are some example images:



Video Implementation

1. Here is my final video output with the best scalefactor 1.4.

https://github.com/vanAken/CarND-Vehicle-Detection_VOLKER_VAN_AKEN/P5_video_ave_S14.mp4

2. My filter for false positives and combining overlapping bounding boxes

I recorded the positions of positive detections in each frame of the video (*def process_image_ave*).
From the positive detections I created a heatmap (line 11, line 65 find_cars) and then thresholded that map to identify vehicle positions (line 19).

Therefore used `*scipy.ndimage.measurements.label()*` to identify individual blobs in the heatmap.
I then assumed each blob corresponded to a vehicle. I constructed bounding boxes to cover the area of each blob detected (line 28).

The cell toolbox contains the threshold (line 5) and bounding box function (line 16).

3. averaging technology

To average the results I define the '*heat_map_recent*' as global and save the current result in it (line 18).
In the next loop every detection will reduced to value of the threshold (line15)) and add on the new heatmap (line 16).
My one layer heatmap que does an overlay before the threshold will be applied. This keep one single window below the threshold, reduce flicker on a detected blob and close missing frames, if only one detection is found on a car.
Let's say there in the area of a previous detect cars will no threshold apply. Simple but working well here.

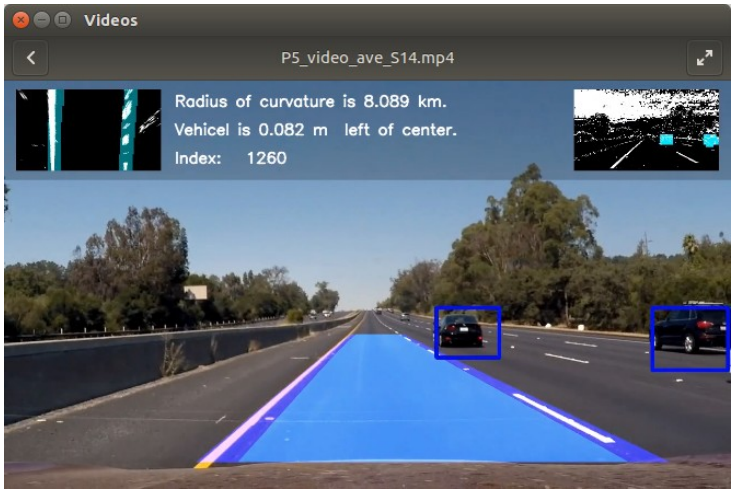
3. lane detection

I implement improved code from the last project, advanced lane detection. This includes:

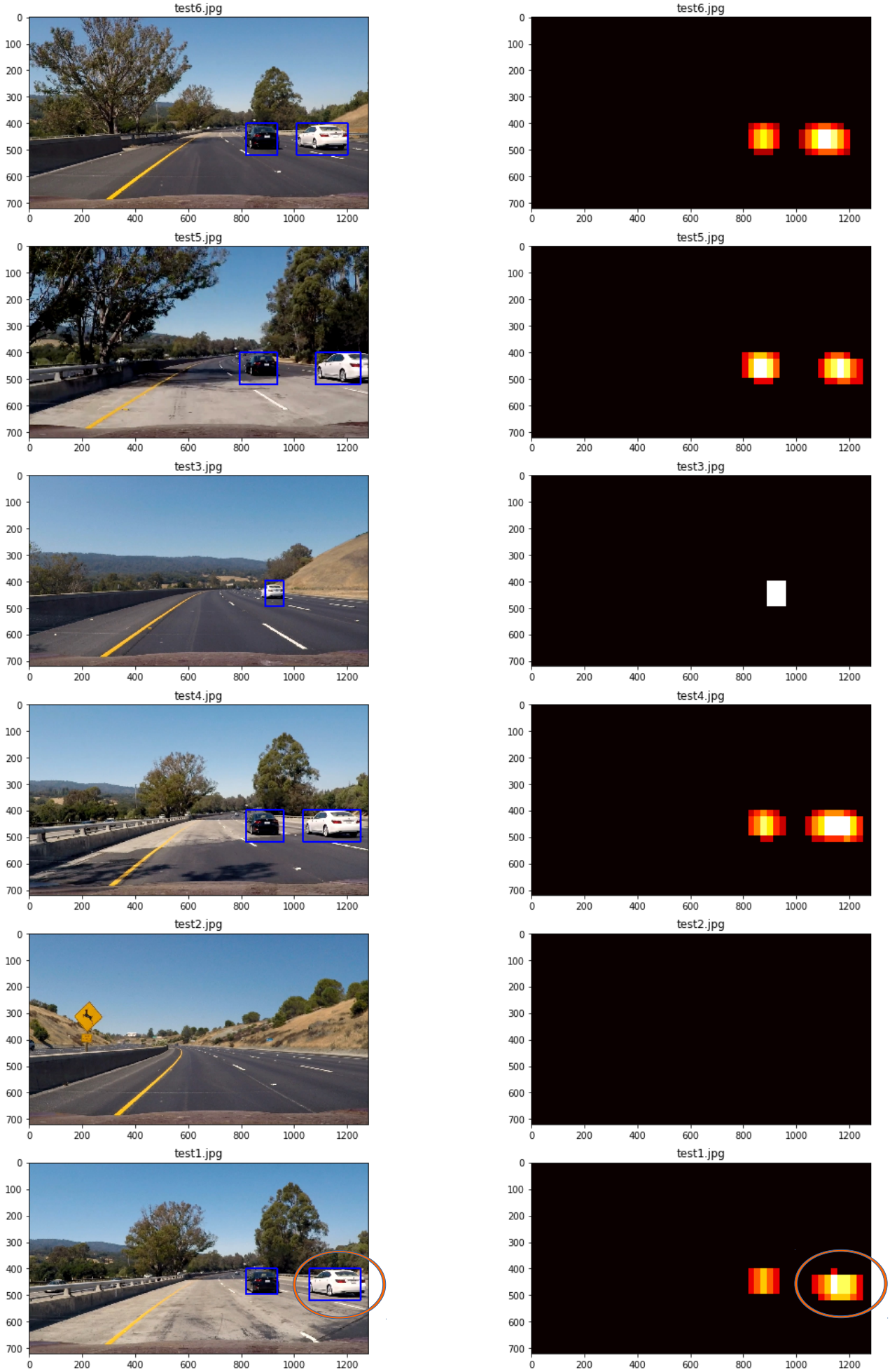
- undistorted the images from Barrel distortion of the lens
- plot the debugging image of the lanes into the left top of the video frame
- Status information like 'Radius of curvature', Vehicle position and a index counter.
- And new: add the heatmap image into the binary plot it into the right top of the video frame

The code can be found in *'def process_image_lanes'* and the file *helper_lanes.py* and *tracker.py*.

4. Here the resulting bounding boxes are drawn onto the last frame in the series:



5. Here are six frames and their corresponding heatmaps:



Discussion

1. Briefly discuss any problems and issues of this implementation

I'm really surprised how well a pipeline of tool come to results like that. I didn't expect that from the beginning. The open points are:

1) The carlist. When one car is behind an other it melts together to one. Here is are some improvements possible with a tracker object.

2) Small cars, that are cars with very less pixels in the image, like the cars on the other side of the road. This seems to be impossible to catch that cars with this kind architecture.

3) Misclassifying, here came the deer from the traffic sign in mind. This sign in just a car, because there are to less traffic sign in the nocar class. More and better data is needed here.

4) Speed - for real time I need process that 50 images in 5s – on my laptop its about 500s.

And that's what I think from beginning: it's only digital technology. What does that mean?
"HOG and SVM is old-fashioned. No reason to learn it today.", was a comment of someone.

It's only digital - not neural.

We have the Lyft Challenge in the last month – is anybody taking about there over SVH and HOG? No, far to slow. Vehicle and lane detection and a lot more will do today with CNN like YoloV3 for example. And thats the solution of the some points. A general fast tool for that problem.

The car list has to be programmed, too – OK, but CNN's work pixelwise an that has more resolution than a 8x8 HOG cell. Misclassifying is also reduced by CNN, because it detects for a sign 95%, than is no longer a car with 75% for example. And the speed is faster, because the whole image will be processed at once no sliding windows. The winner of Lyft challenage Asad Zia got 16FPS with 99.29% prediction of the road and 87.15% of the cars. This is the count of pixel. And a lot of SDC-engeners are close to him.

But to be fair: After that project I've learn some handcraft programing skills, which are useful in general. And now I know this SVM and HOG and have build it once.

The time is fast today. How long was that ago, the breakthrough of SVM and HOG? The break trough was in the 2005. So lets say about 15 Years. This comparable with the time of LeNet (1998-2012). From here on the times runs much faster for the CNN. It seen like a critical mass of hardware and data is there and the speed of developing software increase. Everywhere we found new ideas and the speed of developing is still rising. No limits to see in the future …

And what is about SVM and HOG in the last years? Slow development compare with CNN since 2012. So finally I like to ask for what is SVM and HOG useful in the future? What is the difference?

SVM and HOG adjust the data flow by it's many parameters, while CNN outputs are only controlled by the input data. As an lazy engineer I think CNN are so much simpler in therms of complexity quality documentation and maybe judicial.

However the next step in classification is YoloV3, darknet and do some segmentation.

YOLOv3

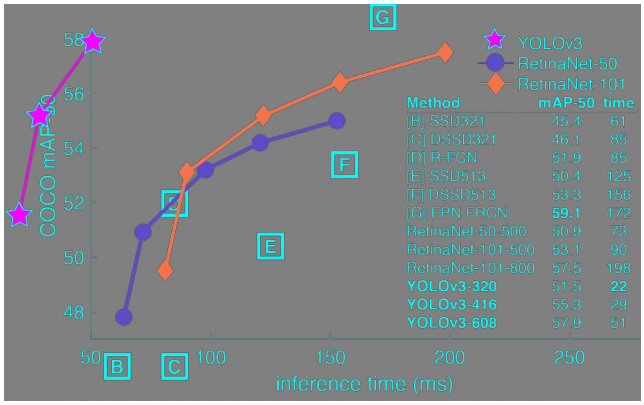
During the final run of my video with the scale factor 1.4. I down load YOLOv3 for the official site:

https://pjreddie.com/darknet/yolo/

It promise to be fast and good!

Y axis: The mean Average Precision
for the detection task.

X axis: process time per image



Follow the steps to the web cam and compile with GPU and openCV.

Than use,

```
./darknet detector demo cfg/coco.data cfg/yolov3.cfg yolov3.weights
data/project_video.mp4 -prefix results/P5 > out.txt
```

and the magic begun. The project video from data flow through yolo into the result folder as slides and the out.txt receives the information of detected objects, like this: Objects: Car: 99% car: 57% car: 59%

I use ffmpeg to stack the slide into a mp4 video with this command.

```
ffmpeg -framerate 25 -i P5%08d.jpg -c:v libx264 -profile:v high -crf 20 -pix_fmt
yuv420p P5_yolo.mp4
```

Believe me before the video of SVM and HOG was finished YOLOv3 version of P5 was already there.

And it separates the car very well and also detect randomly car on the other side of the road.
I will make some test with `-thresh .20` later. But my result show the best frames from the out of the box default settings.

Big cars sometimes small cars or only the corner of car can be detected. Here a fine pre last frame image: