Project 5 Writeup - Martin Hintz

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 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.

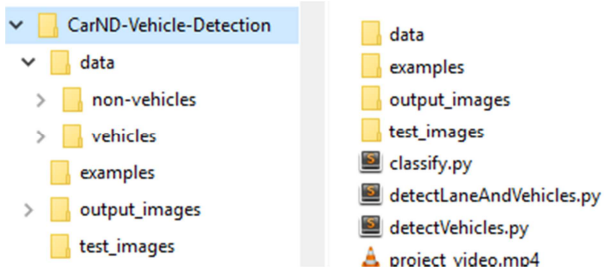
Project Code Usage

The code is available on Github: https://github.com/marthtz/CarND-P5-Vehicle-Detection

To run the classifier the datasets:

- https://s3.amazonaws.com/udacity-sdc/Vehicle_Tracking/vehicles.zip
- https://s3.amazonaws.com/udacity-sdc/Vehicle_Tracking/non-vehicles.zip

have to be downloaded and extracted into a common folder 'data', where the Python scripts are located.
The final structure should look like:



The project was cloned from the official P4 repository. I added the following files:

- In main folder (Python scripts, calibration data and this writeup)
  - vehicleSVCClassifier.py (script to train SVC classifier)
  - tools.py(script with common functions)
  - detectVehicles.py (script with main and processing pipeline for vehicle detection)
  - detectLaneAndVehicles.py (script that combines lane and vehicle detection, P4 + P5)
  - svc_YUV_32_indv.pkl (pickle saved classifier)
  - cam_cal_pickle.p (pickle saved camera calibration from P4)
  - Writeup.pdf

- In output_images folder (processed test images and videos)
  - project_video_detect.mp4 (main video)
  - project_video_detect_debug.mp4 (main video + debug info)
  - project_video_lane_vehicle.mp4 (lane and vehicle detection combined)
  - test1_detect.jpg to test6_detect.jpg
  - test1_detect_debug.jpg to test6_detect_debug.jpg

- In output_images/writeup_images folder(images mentioned in this writeup)
  - car.png
  - notcar.png
  - carhog.png
  - carhogfeatures.png
  - roihog.png
  - windows96.jpg
  - windows128.jpg

Project Code Usage

Usage of the Python scripts:

- vehicleSVCClassifier.py
  This script opens the supplied data set and trains a linear SVC classifier to detect vehicles. The supplied data must be extracted to a "data" folder.
  Execution: "python vehicleSVCClassifier.py"
  The script saves the classifier in a pickle file in the same folder as the script.

- detectVehicles.py
  This script runs vehicle detection on the provided images or videos. In addition, debug information can be created. Executing is done by calling the script with input images/videos as parameter. In addition, if the first argument is 'debug', additional output is generated. A saved classifier has to be specified as an argument before the input files (1st argument if debug is not enabled, 2nd argument if debug is enabled)
  Examples:
  - `python detectVehicles.py debug svc_YUV_32.pkl test_images/test1.jpg test_images/test2.jpg`
  - `python detectVehicles.py debug svc_YUV_32_indv.pkl project_video.mp4`
  - `python detectVehicles.py svc_YUV_32_indv.pkl project_video.mp4`

- detectLaneAndVehicles.py
  This script runs full lane line detection and vehicle detection on provided videos (images and debug not tested). A saved classifier has to be specified as an argument before the input files.
  This script is basically a combination of P4 and P5.
  Examples:
  - `python detectLaneAndVehicles.py svc_YUV_32_indv.pkl project_video.mp4`

Rubric Points

1. Histogram of Oriented Gradients (HOG)

1.1. Explain how you extracted HOG features from the training images

This code is found in "vehicleSVCClassifier.py"", line 38.

First step was to read all training data and into the two separate lists for "car" and "not car" images. As all images in the data set are PNG files, I had to adjust the pixel range to 0 to 255 and casting the array as uint8.

Examples from the dataset:

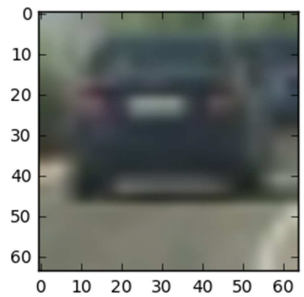
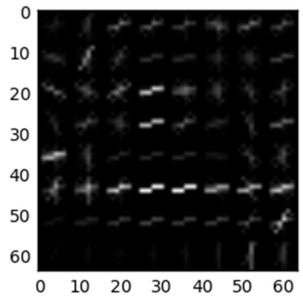
Car (car.png)	Not Car (notcar.png)
	

This code is found in "tools.py"", line 106, 88 and 61 with wrapper function line 196 and 428 and used in "detectVehicles.py", line 67/73 and "vehicleSVCClassifier.py"", line 74/86.

The feature extractor is able to do all 3 kinds of feature extraction, spatial binning, color histogram and HOG features. The HOG feature extractor is the standard function "hog" from the Python module SKIMAGE.

The HOG feature extractor is capable of working on a single or multiple channels of an image as well as on a single image as well as on a list of images. This allows me to use a common function for training (3 channels, multiple images) and for processing (individual channels, single image). The "hog" function from SKIMAGE supports debug output to visualize the HOG features of a given image by setting the parameter "visualise" to true.

Here's an example of extracted HOG features:

Car Image (carhog.png)	HOG features Y channel (carhogfeatures.png)
	

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

There are quite a few parameters that can be tuned for the HOG feature extractor:

- color_space
- orient
- pix_per_cell
- cell_per_block
- hog_channel
- transform_sqrt

After watching the lesson videos on HOG features, the values for parameter orient, pixel and cells made a lot of sense and seemed to be a reasonable choice. They are well balanced between number of features and processing speed given that the training data is 64 x 64 pixel images. Therefore, I concentrated on tuning the color space and the number of HOG channels.

I evaluated different combinations by training a classifier and checking the its accuracy on a test set. I quickly found out that using all HOG channels gives a much better result than using a partial channel set. Modifying the color space didn't make a huge difference in the test set accuracy, which was always > 99%. However, a difference could be seen in performance on the videos. In the end, "YCrCb" and "YUV" performed very similar. I finally selected "YUV".

I also played with the parameter "transform_sqrt" as it was said it improves the performance if shadows are involved. However, my model experienced more false positive detections when this option was enabled. Hence, I disabled this parameter.

1.3. Describe how you trained a classifier using your selected HOG features and color features

This code is found in "vehicleSVCClassifier.py"", line 59 to 96.

In addition to HOG features I also used features from spatial binning and color histogram. For my project I always used the combination of these 3 feature sets. The lesson videos explain very god that a single method is just too susceptible for false positives. In addition, spatial binning and color histogram don't require lots of processing power, so it's relative cheap and easy to get more information and improve the prediction. I experimented with the parameters and came to the conclusion that a spatial size of 32 x 32 and a color histogram of 32 bins performs best on my model.

This code is found in "vehicleSVCClassifier.py"", line 101 to 144.

After the feature extraction I normalize them "StandardScaler().fit()" (both categories combined).

For evaluation of my classifier I split my dataset into a training (80%) and test set (20%). I did this by splitting each category ("car" and "not car") and then combining both to be sure both categories are represented in the same way.

This code is found in "vehicleSVCClassifier.py"", line 154/155.

The classifier was then set up as "LinearSVC()" and trained using its "fit()" function. I then verified the classifier by getting the predictions on the test set using the classifier's "predict()" function.

As a final step I save the classifier and all relevant information using the "pickle" module:

- svc (classifier)
- scaler
- orient
- pix_per_cell
- cell_per_block
- spatial_size
- hist_bins
- color_space

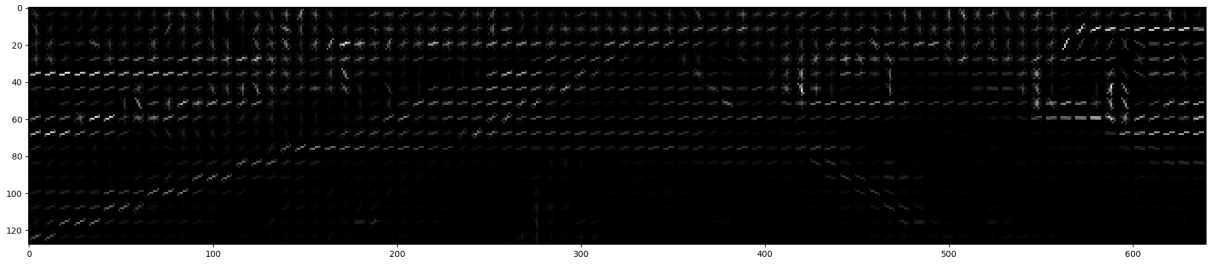
2. **Sliding Window Search**

2.1. Describe how you implemented a sliding window search

This code is found in "tools.py"", line 428 and used in "detectVehicles.py"" line 67/73.

The video lessons presented two approaches for the sliding window search. In the first one, HOG features extraction is done for each window individually, in the second one the HOG feature extraction is done only once for the entire image (or basically just the region of interest) and then just subsamples for each window. I implemented both versions to see the impacts on the performance. I found out that the subsampling method was almost 10x faster than the individual approach. As speed was also a factor mentioned in the project specification I kept the subsampling method as default. This method also uses a

very clever way to implement different window sizes. Actually, the window size is kept constant and similar to the size the classifier was trained on (64x64) and only the entire image is scaled with respect to the chosen window size. To speed up the process and to focus only on relevant data a region of interest is created that basically crops the sky and the engine hood of the vehicle.
Here's an example of the HOG features for a region of interest:

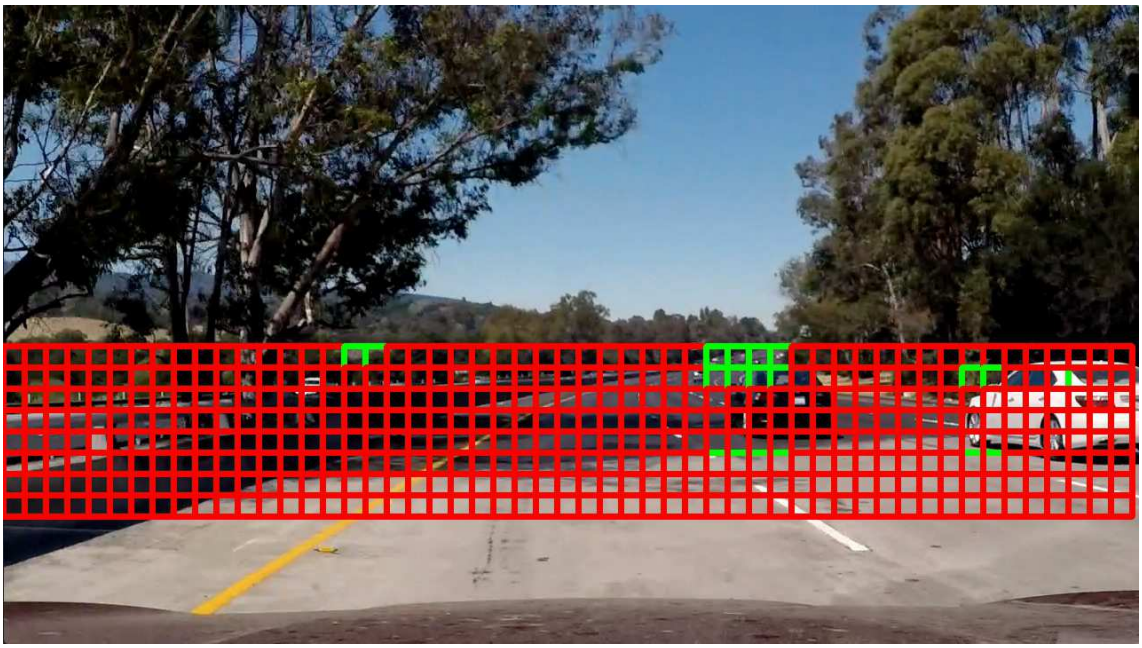


Due to perspective variation it was clear that a single window size couldn't be enough for good performance. Close vehicles are quite big on an image while vehicles farther away are much smaller. I always kept the processing speed in my mind so I wanted to use just as many windows as required but not too many. Regarding the window size I simply tested a couple of sizes on single images where vehicles are far away, near the horizon and close to our vehicle. I decided that 2 images sizes were sufficient to cover cases, 96x96 (250 windows) and 128x128 (75 windows).

For tuning the overlap between windows I chose a smaller overlap of 50% for the larger window size and a larger one (66%) for the smaller window. I basically come up with this by first minimizing the overlap for speed purposes but then figured out that the performance of the smaller windows was worse than expected. Looking at single frames of the videos then revealed that the overlap wasn't big enough for the smaller window size.

I also thought about restricting the region of interest on the x-axis as the video only shows the car on the left most lane in a right curve. This might reduce the processing time and probably improve the false detection rat. However, this model would immediately underperform when driving in another lane.

Example of all windows, size 96 x 96 (66% overlap), green windows are positive detections:



Example of all windows, size 128 x 128 (50% overlap) , green windows are positive detections:

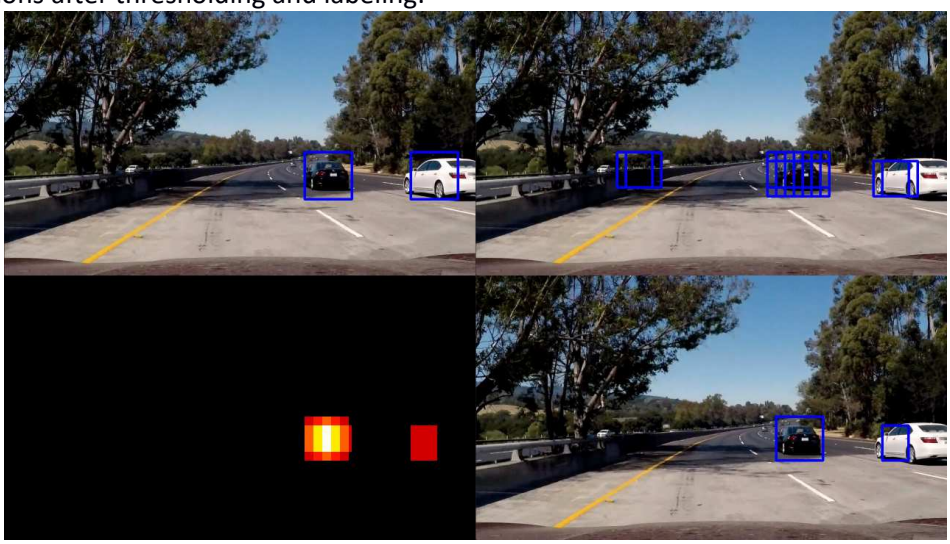
2.2. Show some examples of test images to demonstrate how your pipeline is working. What did you do to optimize the performance of your classifier?

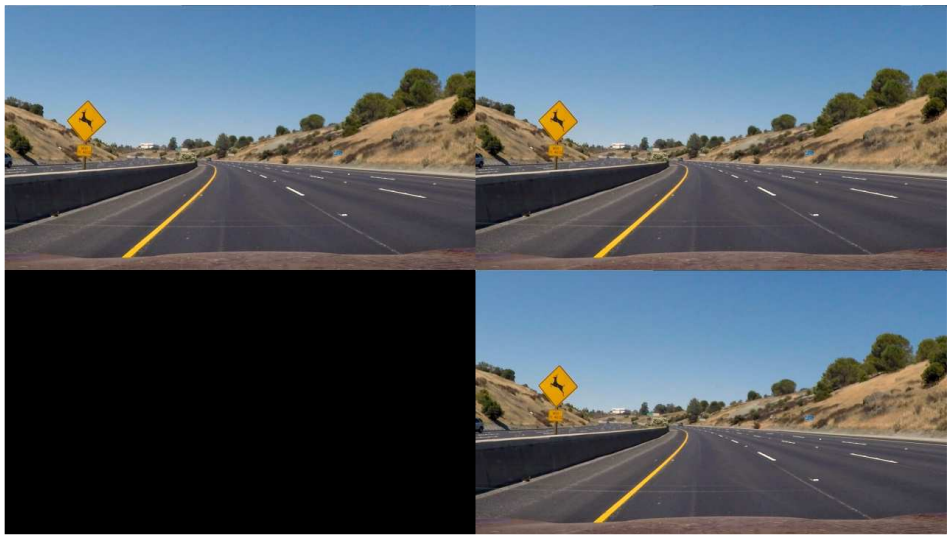
This code is found in “detectVehicles.py” line 84 to 106.

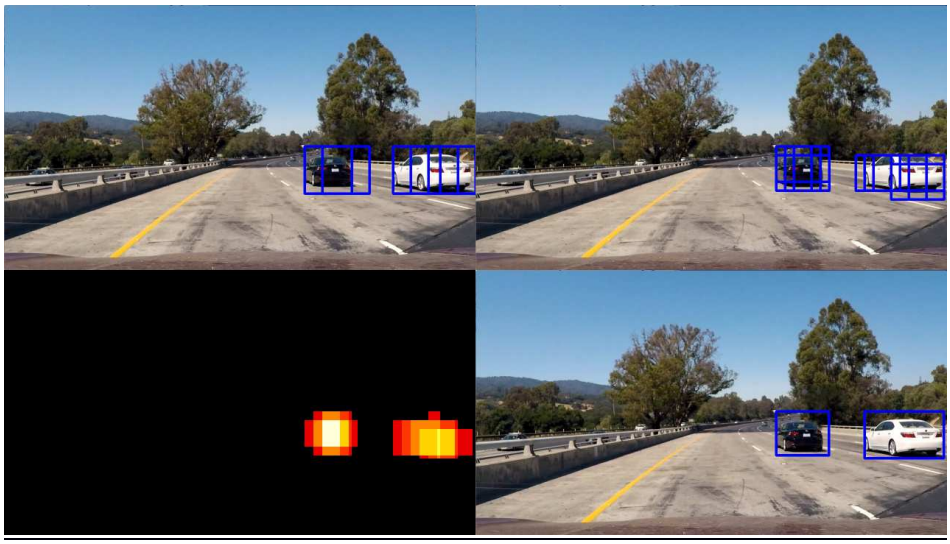
The result of the sliding window search is a list of windows where the classifier found a true prediction for a car. Therefore, the next step is to integrate all of these true predicted windows to generate a combined prediction. This is done by basically creating a heatmap from these windows. The heatmap function adds 1 for each pixel location of a positive detected window. As multiple detections may overlap a good prediction has higher values and gets ‘hotter’. Using the SCIPY function “label” we can assign labels for each heat area. A heat area is a combined area of non-zero values in the heatmap. Using the min and max x/y positions for each label enables us to draw a new single box for each heat area, which basically is a detected vehicle.

Working on several single images it could be seen that vehicles are usually detected by several search windows while false positive detections were usually detected by far less windows. This fact can be used to filter out false positives in our detections. For single images, the threshold must be quite low (1 detection). However, for videos we can average over several frames and therefore have a much higher threshold. This is addressed in a later section.

The following pictures show the test images including debug data to visualize processing steps. The image at the top left shows detections using 128 window size, top right shows detections using 96 window size. Bottom left shows the heatmap integrated over the FIFO. Bottom right shows the final detections after thresholding and labeling:







3. **Video Implementation**

3.1. Provide a link to your final video output

The video can be found here on my Github in folder output images.
https://github.com/marthtz/CarND-P5-Vehicle-Detection

3.2. Describe how you implemented some kind of filter for false positives and some method for combining overlapping bounding boxes

This code is found in "detectVehicles.py" line 84 to 106.
As mentioned earlier the video implementation can make use of several frames to increase the accuracy. To have a more reliable and robust pipeline I integrated heat maps of positive detections over 16 frames. Each heatmap is stored in a FIFO of max length 16. So, when a new heatmap is available the oldest is automatically dropped. Integration is done by simply adding up all heat maps in the FIFO. This improves the detection as some frames probably don't detect a vehicle (e.g. due to unlucky search window placement). In addition, the integrated heatmap can also be used to filter out false positive detections by applying a threshold on this heatmap. I chose a threshold value of 8, so basically a detection must be present in 12 windows over the last 14 frames to be true.
I also figured out that most false positive detections are quite small areas. This is due to the relative high speed of the car and just very few windows that detect these false positives. Hence, I added another filter to drop all predictions where either x or y length is less than 64 pixel. This code is found in "tools.py" line 343.

4. **Debug tools**

To be able do debug the videos I created a debug video that combines images of detection on large windows, small windows, the integrated heatmap and the final image with single bounding boxes. This allowed me to understand the decision making process for every single frame in the videos. This code can be found in '"tools.py", line 532. The debug output can be seen in the previous section.

5. Lane detection combined with vehicle detection

As suggestion in the project description I combined my code for lane detection and vehicle detection. The result is as good as expected and can be found in "output_images" as project_video_lane_vehicle.mp4

6. **Discussion**

The biggest problem was probably to reduce the false positive detections and keeping track of the vehicles close to the horizon. I improved it by slightly adjusting the region of interest to make sure the vehicles were definitely within a search window. Regarding the false positives, modifications to the HOG and binning parameters helped to reduce false positive detections.
Nevertheless, there's still one occasion where a large false positive is detected for 1 or 2 frames.

It would also be interesting to see the performance of a CNN compared to the used linear classifier. However, the training set is not very big, which might explain the amount of false positive detections.

Another interesting insight was the processing speed. HOG feature extraction and sliding window search take quite some time and are far from real-time, especially given that an autonomous car must detect much more than just vehicles.

Finally, it'd be interesting to see how the model performs in various environmental conditions (different weather, tunnel, etc.) but again, this basically comes down to an appropriate data set for training.