**Vehicle Detection Project**

---

# Writeup / README

## Required Files

My project includes the following files / relevant folders:

- pipeline.ipynb is the primary file of this project and contains the full pipeline to generate the final output video
- output_images/ contains the individual images for each step of the pipeline shown in this report
- train_classifier.py holds the code for training the SVM classifier
- helpers.py holds all supporting functions needed to execute the pipeline
- output_video.mp4 is the final video output

## 1. 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.

You're reading it!

# Histogram of Oriented Gradients (HOG)

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

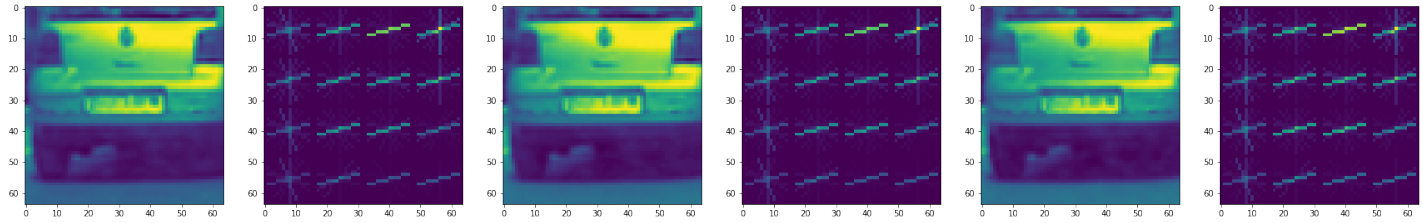
The code for this step is contained in the file `helpers.py` (in lines 6 through 23).

I started by reading in all the `vehicle` and `non-vehicle` images. Here is an example of one of each of the `vehicle` and `non-vehicle` classes:

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.

Here is an example using the `BGR` color space and HOG parameters of `orientations=9` ,

`pixels_per_cell=(16, 16)` and `cells_per_block=(2, 2)`:



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

I tried various combinations of parameters and iterated through the process of training the SVM classifier while inspecting the validation accuracy. Initially, I ended up working with the following parameters:

```
color_space = 'BGR'
orient = 9
pix_per_cell = 8
cell_per_block = 2
hog_channel = 'ALL'
```

While applying the full transformation pipeline on the video stream, I noticed that the initial setting with 8 pixels per cell increased the processing time dramatically while having little to no improvement on the accuracy. . Hence, I decided to inspect the pipeline again and chose to increase the number of cells per block from 8 to 16. Since working with the B-channel (from the BGR image) worked reasonably well to calculate the gradients, I did not see any further necessity to use more channels than that. Thus, I ended up using the following parameter settings:

```
color_space = 'BGR'
orient = 9
pix_per_cell = 16
cell_per_block = 2
hog_channel = 0
```

## 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 the SVM classifier can be found in the file `train_classifier.py`. In addition to the HOG features that I extracted, I also used spatial binning as well as well as color histograms. I applied a RBF kernel and kept the values for C and gamma to their defaults (C=1.0, gamma=1/n_features).

After some experimentation I used the following sizes / number of bins for binning and color histograms:

```
spatial_size = (16, 16)
hist_bins = 16
```

In the file, you will find that I first extract all the features for both vehicle and non-vehicle images (code lines 30 through 48). For the vehicle images, this looks for instance like this:

```
X_vehicle = []

for vehicle_image in vehicles:
    veh_img = cv2.imread(vehicle_image)
    features = single_img_features(veh_img, color_space=color_space,
                            spatial_size=spatial_size, hist_bins=hist_bins,
                            orient=orient, pix_per_cell=pix_per_cell,
                            cell_per_block=cell_per_block,
                            hog_channel=hog_channel, spatial_feat=spatial_feat,
                            hist_feat=hist_feat, hog_feat=hog_feat)
    X_vehicle.append(np.ravel(features))
```

The function `single_img_features()` ( `helpers.py` , lines 58-102) returns the flattened feature vector for each image and is then appended to the X_vehicle list. Afterwards, the y vectors are generated for vehicles and non-vehicles (lines 50-51):

```
vehicles_y = np.ones(len(vehicles))
non_vehicles_y = np.zeros(len(non_vehicles))
```

Afterwards, the X and y values get concatenated for both classes and a scaler is fit to the X values to standardize the features column-wise (zero-mean-centred, variance of 1; lines 53-58)

```
X = X_vehicle + X_non_vehicle
X = np.array(X)
y = np.append(vehicles_y, non_vehicles_y)

X_scaler = StandardScaler().fit(X)
scaled_X = X_scaler.transform(X)
```

Subsequently, a train-test split is performed with 20% testing data and SVC is trained on the training data (lines 59-62).

```
X_train, X_test, y_train, y_test = train_test_split(scaled_X, y, test_size=0.2, ra
ndom_state=42)

clf = svm.SVC()
clf.fit(X_train, y_train)
```

After calculating the accuracy, the final classifier is then pickled to be used afterwards for prediction (lines 64-70):

```
pred = clf.predict(X_test)
acc = accuracy_score(y_test, pred)

out = {'clf': clf, 'scaler': X_scaler, 'accuracy': acc}

with open('classifier.pickle', 'wb') as handle:
    pickle.dump(out, handle, protocol=pickle.HIGHEST_PROTOCOL)
```

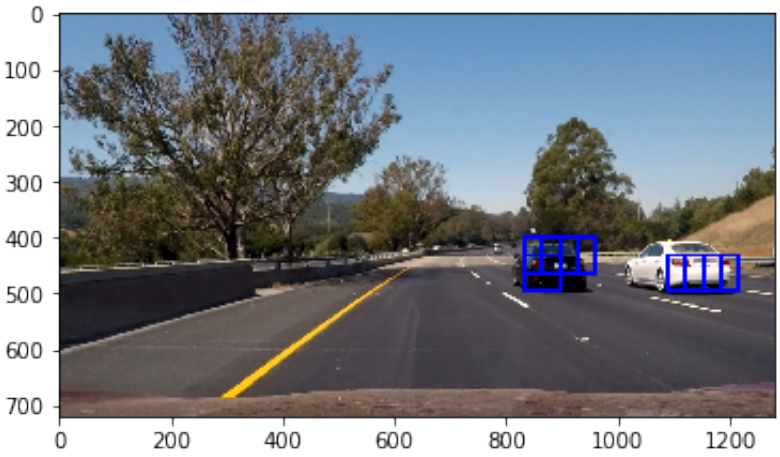
## Sliding Window Search

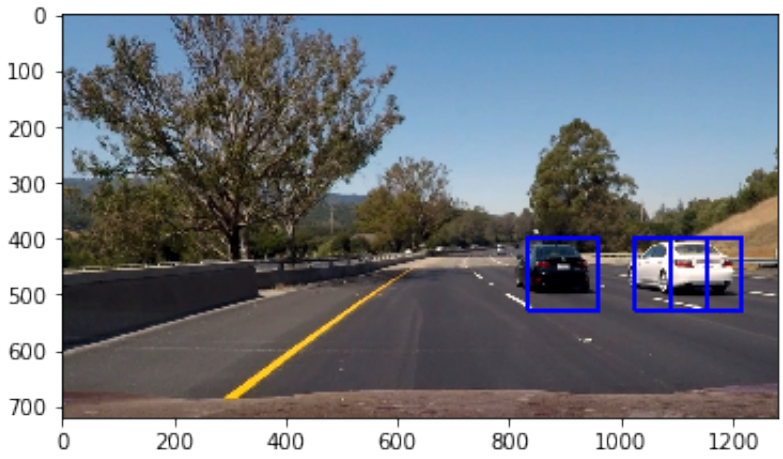
### 1. Describe how (and identify where in your code) you implemented a sliding window search. How did you decide what scales to search and how much to overlap windows?

After initially using the vanilla sliding window implementation (which extracted all HOG features for every window), I eventually switched to the sub-sampling version which only extracts features once. This is implemented in the function `find_cars()` in `helpers.py` (code lines 104-168).

To decide on the scales and the overlap I experimented on the test images and found that a combination of scales 1 and 2 (with a stride of 2 pixels) gave me reasonable results. A stride > 2 (especially in light of the larger HOG cell-sizes of 16x16 pixels) oftentimes resulted in an omission of valuable detected frames.

Two examples for the dections (for both scale 1 and scale 2) are given below:





## 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?

Ultimately I searched on two scales using BGR 1-channel HOG features plus spatially binned color and histograms of color in the feature vector, which provided a nice result. In order to optimize the performance of the classifier I attempted different combinations of features (initially e.g. HOG only; and only on Grayscale images). With accuracy values of 99.3% for the current classifier I decided to leave the classifier like this.

Given the test image above you can see how the individual bounding boxes get merged together through a heatmap detection:


The way this works is through a call to the `heat()` function in `helpers.py`. First a 2d-matrix filled with zeros of the same size as the original image is generated (line 206). Then a call to `add_heat()` is carried out which takes in the empty heatmap and a list of detected bounding boxes. This function loops through all boxes and for each pixel inside that box a "1" is added to the heatmap (lines 176-179):

```
for box in bbox_list:
    heatmap[box[0][1]:box[1][1], box[0][0]:box[1][0]] += 1
```

In order to remove false positives, a threshold value of 1 is applied to the heatmap, which means that for a

pixel to be counted as relevant, it must appear in 2 or more (overlapping) bounding boxes (code line 212).

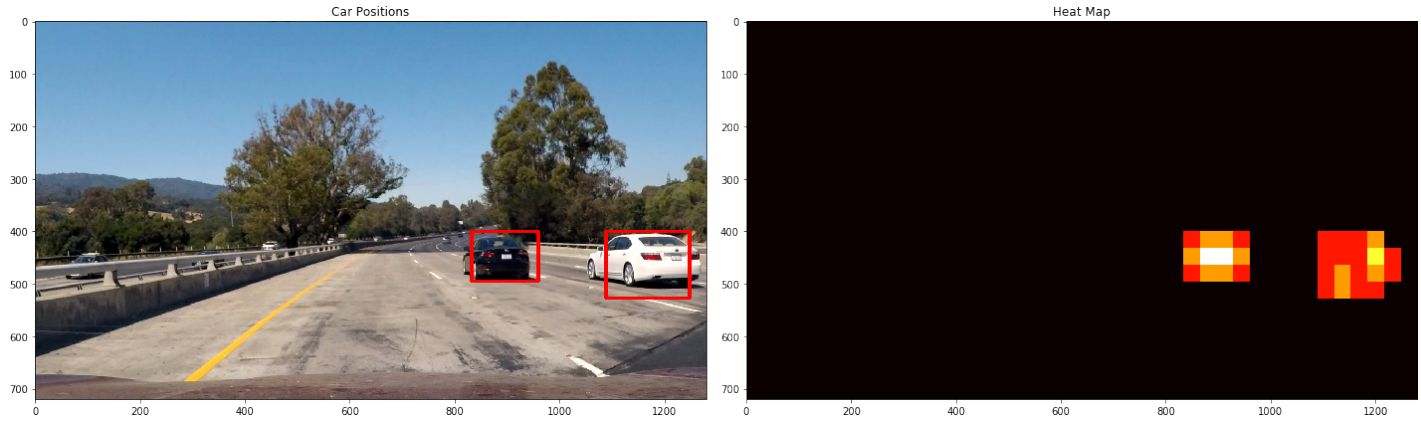
```
heat = apply_threshold(heat,1)
```

To subsequently draw one bounding box for all the overlapping detections in the heatmap, the SciPy function `label()` is applied to the heatmap (line 216). This function returns all adjacent pixels in an image, grouped by region. This allows afterwards to detect the min- and max- values on the x- and y-scale, which can be used to identify two opposite points that describe a rectangle (code lines 190-203).

The result of applying the same pipeline on two other images can be seen below:

**Window detection:**

**Combination:**




## 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.)

Here's a link to my video result

### 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.

In addition to the heatmap thresholding described above, I applied one simple additional technique: I threw out all detections where the outer right x-coordinate is <= 600 because I had to fight with false positives with traffic from the opposite direction. As described before, the overlapping bounding boxes are merged together by applying the labels function and then extracting the bottom-left and top-right corners.

```
pt_right = (xbox_left+win_draw)
if pt_right > 600:
    bboxes.append([(xbox_left, ytop_draw+ystart),(xbox_left+win_draw,ytop_draw+win
_draw+ystart)])
    cv2.rectangle(draw_img,(xbox_left, ytop_draw+ystart),(xbox_left+win_draw,ytop_
draw+win_draw+ystart),(0,0,255),6)
```

## 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?

As you can see in my project video, the detections are quite jumpy. This means that it would make sense to

try to "smoothen" the detections. This could be done by trying to follow a detection from frame to frame and only starting to count it as a viable detection after being detected in, say, 3 subsequent frames. This could be implemented similarly to the Advanced Lane Finding project with a "Vehicles" class that stores previous detections.

Apart from that I have the feeling that my classifier has problems with the white car in the video, which might be due to the fact of an underrepresentation of white cars in the training data. To make the pipeline more robust, it would make sense to collect more training data or augment the training set.

In order to accurately discard detections from the opposite direction of the road, it would be necessary to determine the outer-left lane line, e.g. through the techniques applied in the Advanced Lane Finding project. Currently I'm only discarding detections with outer-right x-value < 600 pixels.