Udacity Self Driving Car Nanodegree –

Project: Vehicle Detection and Tracking

By:

Saurabh Sohoni

Udacity profile: <https://profiles.udacity.com/p/u183059>

Contents

1. Introduction 3

2. Project Goals 3

3. Implementation of Project Rubric Points 3

3.1 Write up / README 3

3.2 Dataset Exploration 4

3.3 Histogram of Gradients (HOG) and Color Features 4

3.4 Sliding Window Search 7

3.5 Pipeline on Test Images 9

3.6 Video Implementation 10

3.7 Discussion 11

4. References 12

# Introduction

This document acts in support with code and output files submitted in order to implement ‘Vehicle Detection and Tracking’ project. In this project, a pipeline is created to detect cars on a road and track their path. This is achieved with the help of Support Linear Vector (SVC) classifier, one of the widely used classifiers in machine learning for real-time applications. A camera mounted on the hood of the vehicle in center records path ahead of a car. Project pipeline created in this project is then applied to this video recording to detect other cars in ahead and beside and track their path.

# Project Goals

Following were the goals of this project:

1. Extract images containing cars and not containing cars and label the data in order to train a classifier. The cars and not cars images were provided by Udacity. These example images come from a combination of the [GTI vehicle image database](http://www.gti.ssr.upm.es/data/Vehicle_database.html) [1], the [KITTI vision benchmark suite](http://www.cvlibs.net/datasets/kitti/) [2], and few examples extracted from the video recording done by a camera mounted on the hood of the car.
2. Extract raw color features and histogram of color features from each training image and form a vector of features.
3. Extract Histogram of Gradients (HOG) features from each training image and form a vector of features.
4. Normalize and randomize the feature vectors prepared in above step and feed them to a classifier.
5. Train a classifier to detect distinguish between images of cars and not cars and validate the accuracy on a test set.
6. Implement a sliding window technique to divide an image frame from the video recording into small sections. Apply the classifier created above on sections of image to search for cars and draw predictions.
7. Draw a bounding box in the corresponding section of image where cars were detected.
8. Apply car detection algorithm mentioned above to a video stream and maintain a track of cars. Reject spurious false positives by comparing their detections with track of cars.
9. Summarize the approach and results in a report.

# Implementation of Project Rubric Points

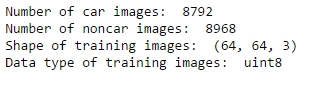
Following section lists down various rubric points for this project and also details out the implementation strategy followed:

## Write up / README

This report lists methodology used in creation of vehicle detection and tracking pipeline and supports the code written for various python methods used in this project.

## Dataset Exploration

For this project, vehicle and non-vehicle images were taken from a combination of labelled dataset from GTI vehicle image database [1] and KITTI vision benchmark suite [2]. A total of roughly 9000 images for each vehicle and non-vehicle class were present in the combination. This is shown below along with some other features:



## Histogram of Gradients (HOG) and Color Features

#### **3.2.1 Selection of hyper parameters for HOG**

Selection of right hyper parameters is important as to ensure the algorithm results in high accuracy and the processing time is lesser. For this project, following hyper parameters were selected:

1. Color space (YCbCr) – In general, cars are more saturated in color than other objects in the background. This feature is prominent in YCbCr color space and hence, original RGB images were converted to YCbCr color space before extracting HOG features.
2. Number of HOG channels (All channels) – HOG features were calculated for each channel in YCbCr color space of images in order to ensure color, saturation and illumination features are retained.
3. Number of orientations (11) – Number of orientations corresponds to the number of directions in which gradients in an image are oriented. For this project, 11 orientations were chosen. As a result, gradient features were separated by roughly 33 degrees (360 degrees for complete circle. divided by 11 orientations).
4. Pixels per cell (32) – While calculating HOG features for an image, the image is further sub-sampled into small boxes and oriented gradients are calculated for each box. This feature of HOG provides information on the shape of objects and edges around the corner. Images in training set were of the size 64 x 64 (width x height). Pixels per cell value of 32 suited the best to achieve high accuracy and faster computation speeds. Also, 32 pixels per cell divided the entire image into four quadrants where directional features for edges on cars are distinctively seen.
5. Cells per block (2) - Features in neighboring cells in an image may be similar (in case of no edge) or may be very different (in case of edges). Hence the magnitudes of features may be small or large but all features are equally important. In order to reduce the effect of magnitude on features, local features in a given cell are normalized. In this project, two neighboring cells were used to create a block of 4 cells and then normalization was implemented.
6. Square root of transforms – This is another form of normalization which uses power law of ‘gamma’ normalization scheme. As it is known to reduce the effects of dark shadows and bright illuminations in the image, this technique was also applied.

Using the entire configuration parameters described above, the HOG image obtained for a car from training set is shown below:

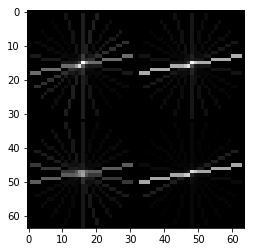


Figure: Image (64 x 64) of car on left, HOG features in 11 orientations, 32 pixels per cell for image on right

HOG features were calculated using hog() method from scikit image (skimage) python library. Code for calculating HOG features can be found in hog\_features() method in declared in the pipeline.

#### **3.2.2 Selection of hyper parameters for Color features – Spatial Binning**

It was observed that significant color features were retained even after the dimensions of training images were reduced. This insight was used to implement spatial binning on training images. Following hyper parameters were chosen:

1. Color space – To ensure a balance between bright and dark pixels features, both BGR and YUV color space were chosen in pre-processing of images.
2. Spatial dimensions – All images were down sized to 16 x 16. This step improved the processing time of the pipeline while retaining important color features at the same time.

#### **3.2.3 Selection of hyper parameters for Color features – Color Histogram**

Vehicles are more saturated in color than most other elements in the background. Also, most commonly, vehicles are of standard colors viz. white, black, red, maroon, violet, etc. In detection of vehicles, density of color and saturation prove to be important features. This insight was used to calculate histogram of pixel values in BGR and YUV space.

Following were the tuned hyper parameters:

1. Color range – Entire range of 8 bit pixel values from 0 to 255 were used.
2. Number of histogram bins – The above chosen color range was divided into 64 bins and pixel densities for each bin were calculated.

Color features for an image from the training dataset were calculated for RGB color space and are shown below:

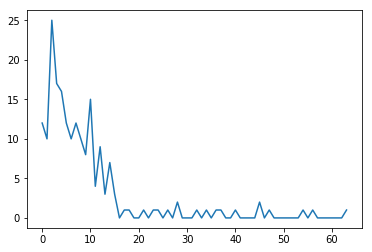
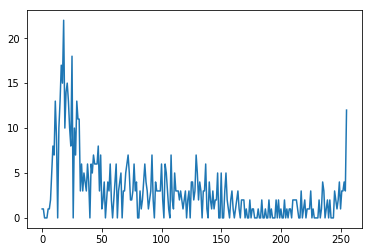


Figure : From left: Vehicle Image, Histogram of color features over entire color range, Histogram of color features when entire color range is divided into 64 bins, respectively.

Color features were calculated using histogram() method from numpy python library. Code for calculating color features can be found in bin\_spatial() and color\_hist() methods in declared in the pipeline.

#### **3.2.4 Feature Vector normalization**

An important step to be carried out before training is normalization of feature vectors. As stated above, once all features for an image are combined into a single row vector, all features are nothing but raw data values. The variable scales of these features results in classifier to train incorrectly. For e.g.: For color features, the data points are in the range (0, 255) while for HOG features, the data points are between 0 and 1. This inconsistency of values is resolved by normalizing each feature vector before combining.

In project pipeline, this step is implemented by using StandardScaler() method from scikit learn preprocessing library. Code can be found in normalize\_features() method declared in the pipeline.

#### **3.2.5 Combined Feature Vector**

HOG and Color features for every image in the training dataset were calculated and a combined feature row vector was generated after normalization. This was then fed as an input to the training classifier.

#### **3.2.6 Training on Linear SVM classifier**

Linear SVM are robust whenever there is clear decision boundary in the dataset for given output classes. In this project, since there were only two output classes (vehicles and non-vehicles) and also the dataset was limited (around 9000 images for each class), linear SVM classifier was the best guess. For validation purposes, 20% of the training dataset was kept aside. This was then used to predict the accuracy of the trained model.

Training was carried out on three combinations, viz. with HOG features alone, with color features alone and with HOG and color features combined. The results are shown below:



It is evident from the results that the model performed better when both HOG and color features were used. This trained model was then used to detect vehicles in each frame of the project video.

## Sliding Window Search

For detecting vehicles in test images and project video provided by Udacity, a sliding window search approach was used. In this approach, selected section of image/image frame was chosen. This is the section where the probability of vehicles is high. This section was then divided into small rectangular windows with overlap between windows to search for features present prominently in vehicles. For each positive detection, the window section was selected and all such windows were combined in a heat map to get overlapping detections in high confidence region. Details on each step are given below:

#### **3.4.1 Selection of region of interest**

Vehicles are usually seen in bottom half of the image, hence, upper half of the image was cropped. Also, in this project, only vehicles moving in the same direction as that of vehicle were detected and tracked. So some portion on left side of the image was also cropped as the vehicle is in the leftmost lane on the road. Exact details are given below:

1. In X axis (horizontal axis), search is limited to 85% of image pixels from right, i.e. 15% section from left is cropped. This was done as no vehicles in moving in the direction of vehicle can be expected on the left as the vehicle is moving in the left most lane. Also, search limitation speeds up the pipeline.
2. In Y axis (vertical axis), search is limited to 37% of image pixels in between, i.e. 55% section from top and 8% section from bottom is cropped. This was done to as no cars can be seen in top portion of the image. Also, search limitation speeds up the pipeline.

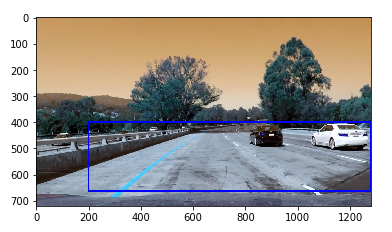
The region of interest is marked in the example shown below:

Figure : Region of interest for sliding window search

#### **3.4.2 Windows with overlap**

Vehicle detection is tricky as the shape of vehicle varies in size for a nearer and farther away vehicle. To achieve efficient detection, sliding windows of multiple shapes (scales) were used. Details are as follows:

1. Window dimensions for 55% to 92% in Y axis from top (Span the entire region of interest) - 96 x 64 with overlap of 50% and 75% in x and y axis respectively. Vehicles not very far and not very close to the vehicle were detected using this window and overlapping setting.
2. Window dimensions for 55% to 66% in Y axis from top (Span few cells in top region of interest) - 64 x 32 with overlap of 80% both in x and y axis. Vehicles almost near the horizon were detected using this window and overlapping setting
3. Window dimensions for 55% to 92% in Y axis from top (Span the entire region of interest) - 192 x 128 with overlap of 50% and 70% in x and y axis respectively. Vehicles very close to the vehicle were detected using this window and overlapping setting.

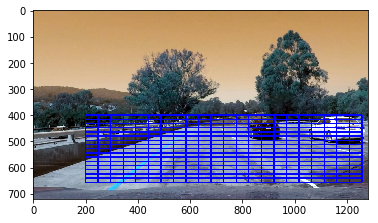
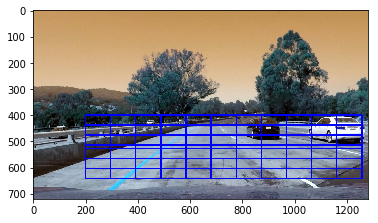
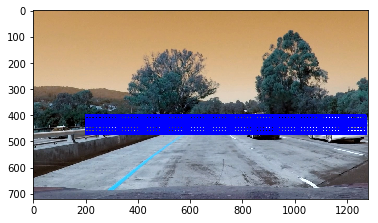
Windows for all of the above settings were drawn in a sample image which is shown below:

Figure : Window sizes and overlap

#### **3.4.3 Thresholding using heat maps**

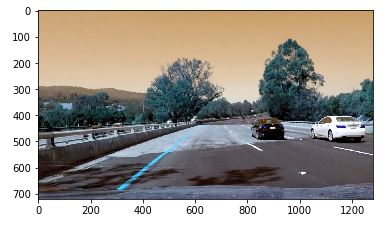
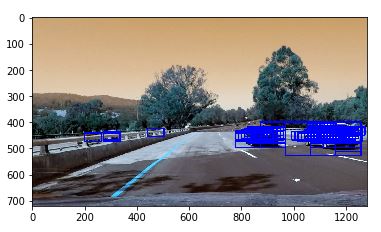
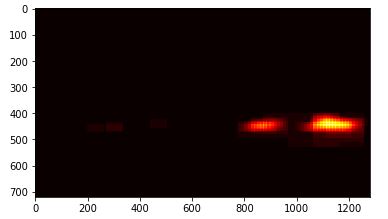
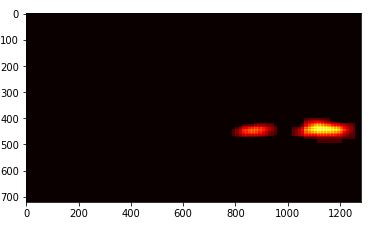
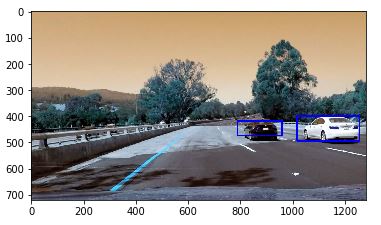
For each test image, probable vehicle locations were obtained by running the model on all image sections derived from windows. Detections of overlapping windows strengthened the probability of a vehicle in the region. Also, non-overlapping detections were mostly false positives and can be thresholded from the heat map. In this project pipeline, heat maps were thresholded to select detections with minimal overlap of 3. Example is shown below:

Figure : From top left, in clockwise direction, Original image, probable vehicle detections, heat map, thresholded heat map, final detection

False positives were removed once a threshold of minimum 3 overlap was applied on heat map. Sliding window implementation can be found in slide\_window() declared in the pipeline. All code in this method is attributed to code shared in one of the class videos of Udacity [3]. Window sizes and overlap chosen for this project can be found draw\_predicted\_cars() method declared in the pipeline. Code for thresholding heat maps and labelling them is also included in this method.

## Pipeline on Test Images

The trained model created above was then used to identify vehicles in static images. Out of the 6 test images, the model predicted vehicles accurately with only one false detection in one of the images. The results are shown below:

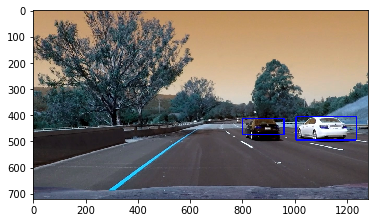
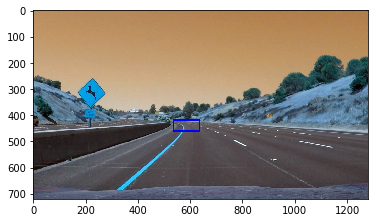
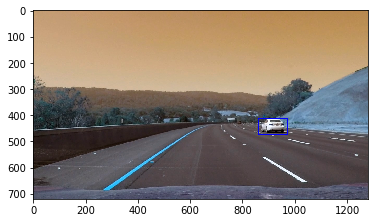
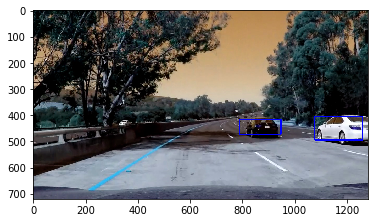


Figure : Vehicle detections on test images

## Video Implementation

Trained model and vehicle search using sliding windows was employed on subsequent frames of project video provided by Udacity. A bounding box of green color was drawn where the position of vehicle(s) was estimated. For extracting image frames from a video and writing back vehicle detected frames to a video, VideoFileClip() method from moviepy editor python library was used.

Detections in video frames were wobbly and there was a need to smooth the size of bounding boxes in subsequent image frames. This was implemented using heuristic and average of heat maps given below:

1. VehicleTracker class was created to track thresholded heat maps in generated in subsequent frames of video
2. In the beginning, no heuristic is available. Hence, the heat map for current frame is retained in ‘self.last\_heatmap’ property
3. For every subsequent frame, major portion from retained heuristic (weighted by 0.7) and few updates from heat map for current frame (weighted by 0.4) was merged and thresholded. Thresholding was done to retain detections with a minimum overlap of 3.
4. After every 20 frames, the heuristic was dropped and new heuristic was fetched. This is simply a go to step 2 and loop until all frames in the video are processed.

## Discussion

#### **3.7.1 Challenges faced**

One big challenge faced during implementation was choosing the right size of window and overlap. Incorporating more windows resulted in slowing the algorithm. Lesser windows could not guarantee high probable detection and introduced false positives.

#### **3.7.2 Scenarios resulting in failure of pipeline**

Following are a few of scenarios where the pipeline created currently may fail:

1. Currently, the pipeline takes about 2.5 seconds to process each frame in a video. Hence, this algorithm will fail in the real time. The algorithm has to be roughly 60 times faster consider the video is recorded at 25 frames per second.
2. Current pipeline only detects vehicles moving in the same direction as that of car. This was done to make algorithm run faster and that the car was moving in the left most lane, so no vehicle is expected on the left. If the car changes its lane to the right, then the pipeline will fail to detect vehicles in the left lane.

#### **3.7.3 Solution to challenges and carving a robust pipeline**

To avoid failure of pipeline in scenarios mentioned above, few algorithms given below should be used along with existing pipeline:

1. To truly get real time performance, a convolutional neural network like SSD or YOLO can be used. YOLOv2 gives about 50 FPS. By using a high end GPU like NVIDIA GeForce GTX Titan X, computation time of HOG features and color histogram features can be reduced to a great extent.

# References

1. [1] [GTI vehicle image database](http://www.gti.ssr.upm.es/data/Vehicle_database.html)
2. [2] [KITTI Vision benchmark suite](http://www.cvlibs.net/datasets/kitti/)
3. [3] [Sliding Window Search](https://classroom.udacity.com/nanodegrees/nd013/parts/fbf77062-5703-404e-b60c-95b78b2f3f9e/modules/2b62a1c3-e151-4a0e-b6b6-e424fa46ceab/lessons/fd66c083-4ccb-4fe3-bda1-c29db76f50a0/concepts/8e39c07e-afd5-4ba5-9204-8b44aa39285c)