Vehicle Detection Project

Udacity: Self-driving car engineer nanodegree

Student: Florian Wolf

About the Project

The goal of this project is to build a working pipeline, which classifies vehicles in a video stream. Therefore a classifier must be trained, image features extracted and be feed into the classifier. The output video will draw a rectangle around the detected vehicle.

Histogram of Oriented Gradients (HOG)

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

Gradients of an image give a more robust information about the shape of an object in an image. by dividing the image in various cells the Orientation of the Gradient can be calculated for each cell. Flattening this information into an array will give a specific signature to different shapes. This approach is used for the HOG feature extraction. Before computing the gradient the colorspace was transformed from RGB to HLS. To further extract the HOG features the scikit function hog() was used:

``` python from skimage.feature import hog

features, hog*image = hog(img, orientations=orient, pixels*per*cell=(pix*per*cell, pix*per*cell), cells*per*block=(cell*per*block, cell*per*block), transform*sqrt=False, visualise=vis, feature*vector=feature*vec)

```

To create a training set the `vehicle` and `non-vehicle` images from the Udacity dataset have been used. Here is an example of one of each of the `vehicle` and `non-vehicle` classes:

Experimenting different colorspaces led to the empirical decision to use the HLS colorspace for HOG transformation. The HOG parameters orientations, `pixels_per_cell`, and `cells_per_block` had to be tuned. The result of the tuning is explained in the next section

This is defined in lines 22-99 of pipeline.py.

2. HOG parameters.

The parameters were found imperially by testing with a linear classifier and comparing the accuracy results. The following parameters were chosen: ``` orientations = 9 pixels*per*cell = (8,8) cells*per*vlock = (2,2) channels = All

``` These parameters also match with the ones used in the Udacity lesson.

**3. Building and training the classifier**

The skilearn library offers several options when it comes to classifiers. To keep the computation time low a simple linear SVM as been used in this project. This classifier draws a linear decision line between vehicle and non vehicle objects. The classifier was trained with images of cars and images showing parts of the road. To differ between vehicle and non vehicle the images were labelled with ones and zeros. To train the classifier the images were pre processed. An array was created, containing the flattened HOG features as well as the HLS colorspace histogram and the spatial features of the image. Then the vehicle and non vehicle arrays were stacked and normalized with the scikit StandardScaler(). The data was then shuffled and split into a training and a testing set. With the training set the classifier has been trained. The trained classifier scored the following accuracy when used upon the test set:

```
clf.score(X_test, y_test) 0.9918355855855856
```

This is defined in lines 252-284 of pipeline.py.

## 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?**

To search the images for vehicles the approach from the Udacity lessons were used. First the image is scaled. A scaling value of 2 showed the best results. Then the image is converted to HLS colorspace. To save computational time, the next step is to HOG transform the three image channels. The image is divided in several windows of 64x64. The features are extracted for every window and passed on to the classifier. If a vehicle is detected, the corners of the vehicle's bounding box is appended to a list. with this list a heatmap is created. Every pixel within a box gets a value of +1 or higher for multiple detections. Then a threshold is applied to the heatmap to exclude false positives. The thresholded image is labelled with the label() function and rectangles are drawn around each label. The original image with the rectangles drawn onto is the final output of the pipeline.
```

This is defined in lines 125-196 of pipeline.py.

Video Implementation

1. Video implementation

The pipeline was applied to the video of the previous project. cars on the road are identified correctly with a minimal amount of false positives. Here is the link to the output video:

~~link to my video result~~

2. Filtering false positives and bounding multiple detections

The actual cars in the image get multiple detections when running through this code, therefore the heamap is created. by thresholding the heatmap most of the false positives are excluded, a threshold of 3 has been applied. To avoid sudden changes per frame a class was implemented, which saves previous heatmaps and averages them with every new one. Since cars move slowly through the picture this is a good way to eliminate errors from single frames In order to avoid false positives, parts of the image in which certainly no vehicle can be found were excluded from the search. Only the lower part of the image is searched for vehicles. This also speeds up the processing time.

Here's an example result showing the heatmap

Discussion

The code identifies cars on in an image but also a lot of false positives. Filtering out the false positives by applying a heatmap and averaging over the last frames has showed to be a solution for this video. In real life this might be dangerous since an actual vehicle might be filtered out. A better solution would be to use more features and a better classifier to get better results without filtering. Also the classifier has got to be trained on more then just vehicles. It has got to be able to identify trucks, motorbikes, bicycles, pedestrians and even animals.