Udacity Project 05 - Vehicle Detection and Tracking

README

The code I used for this project can be found in `p5.ipynb and functions_required.py (attached files).` The required import functions defined in both the files (codec cell 1 of p5.ipynb). The upcoming sections discusses further detail about the specific points described in the Project Rubric.

Udacity course codes were used for constructing the pipeline as required. A .pdf copy of the p5.ipynb is also attached for ready reference.

Attached File Set

1. Python File	p5.ipynb
2. Python File	functions_required.py
3. Html file	p5.ipynb in HTML format
4. Trained Classifier File	classifier.p
5. output _images Folder	Containing 6 test_output jpg files
6. Video File	project_video_output.mp4
7. Project ReadMe File	readme.pdf

Usage

```
IPython notebook p5.ipynb
(from Command prompt after due set up of conda and python environment variables)
```

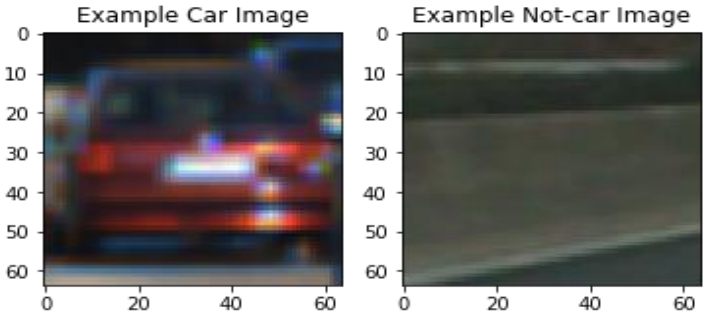
Histogram of Oriented Gradients (HOG)

1. Loading the training data

The code for this step is contained in the 2nd and 3rd code cells of the IPython notebook `P5.ipynb`. First, the `glob` package is used to read in all the files in the data-set. Length of car set is 8792 and notcar set 8968.

Here is an example of one of each randomly picked of the `vehicle` and `non-vehicle` classes:

```
Your function returned a count of 8792  cars and 8968  non-cars of size:  (64, 64, 3)
and data type: float32
Out[3]: <matplotlib.text.Text at 0x7f2f2664d128>
```

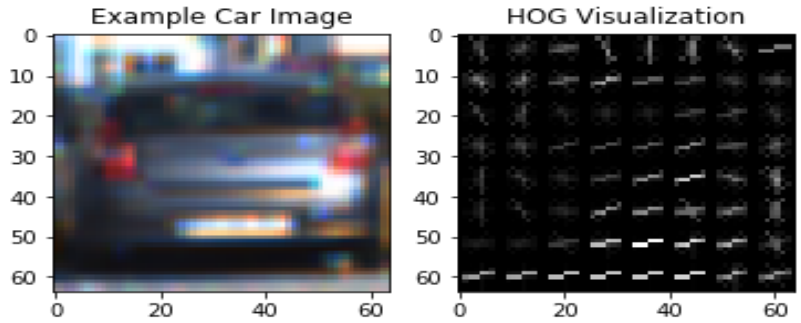


2. HOG Parameters

Different colorspaces and HoG parameters were explored. The final configuration was chosen according to the one which gave the best test-set accuracy from the classifier (described under Training the Classifier below).

Here is an example as coded in 4th coded cell of p5.ipynb with HOG parameters of of `orientations=9`, `pixels_per_cell=(8, 8)` and `cells_per_block=(2, 2)`:

```
Out[4]: <matplotlib.text.Text at 0x7f2f26518b70>
```



3. Training the classifier

The quality of the training and prediction directly varies with the details of HOG features. It can be tweaked with the following parameters to get an effective HOG feature.

Color Space: Using appropriate color channel shall help in getting the essential features to be used for HOG feature generated. I tried various color spaces and finally settled with **YCrCb**, which deals with separating the luminance and color difference that helps in segregating features effectively.

Number of Channels to use in an image: HOG features can be arrived at an individual color channel of the image or it can be arrived at all channels and combine them to a single HOG feature for the given image. Very often, using single channel to use for HOG feature generation would be faster to process, but many essential features may be left out. Hence I used **all** the channels in the chosen color space of the image.

Number of Orientations bin: In HOG, the binning happens based on orientation gradients. You can bin the orientation from 6 to 12 buckets. Having very less bins (ex: 6) will lead to broad binning and thereby losing some essential features. On the other hand, having too many bins (ex: 12) will lead to too much detail used and will not help in generalizing the feature and might take long time for processing. To have an effective bin size, I chose the orientation bin size as 9 which I consider may be optimum.

Pixels per Cell: When we try to arrive at a Histogram, the entire image is split into a bunch of smaller number of cells of size (rows x cols) pixels. Having too many pixels in one cell will lead to less number of cells and thereby more generalization might happen. Similarly having too less pixels in a cell will lead to too specific & would lead to over fitting as well. So, I chose an average of 8 pixels per cell on both rows and columns. For the given 64x64 pixel image, there were 8 cells on each direction.

Cells per Block:. If we have too many cells per block, then the generation is more leading to loss of essential features. So, I used 2x2 cells block for effective HOG features

A `LinearSVC` was used as the classifier for this project. The training process can be seen in code cells 5 of `p5.ipynb`. The features are extracted and concatenated using functions extract_features, bin_spatial, color_hist, get_hog_features in python file: `functions_required.py` which is imported into p5.ipynb. Color space used for the training is "YCrCb" and Hog Channel = "ALL". Training parameters are also stated in the code cell 5 of p5.ipynb file.

Since the training data consists of PNG files, and because of how `mpimg.imread` loads PNG files, the image data is scaled up to 0-255 before being passed into the feature extractor using following code lines in the process_image function of p5.ipynb file.

image = image.astype(np.float32)/255

The features include HOG features, spatial features and color histograms. The classifier is set up as a pipeline that includes a scaler as shown below:

```
clf = Pipeline([('scaling', StandardScaler()),
                ('classification', LinearSVC(loss='hinge')),
               ])
```

This keeps the scaling factors embedded in the model object when saved to a file. The model stored in the file classifier.p obtained a test accuracy of 99.07% where the test-set was 20% of the total data.

```
Using: 9 orientations 8 pixels per cell and 2 cells per block
Feature vector length: 8460
6.46 Seconds to train SVC...
Test Accuracy of SVC =  0.9907
```

Function visualize3Images indicated in p5.ipynb file is a Utility Function to visualize images (Called only When required Testing and Checking)

Sliding window Search

1. Scales of Windows and Overlap

The function `process_image (in p5.ipynb)` and `slide_window, search_windows (in functions_required.py)` were used to generate the list of windows to search. The input to the function specifies both the window size, along with the 'y' range of the image that the window is to be applied to. Different scales were explored and the following set was eventually selected based on classification performance:

Window size	Y-range	Overlap
(80, 80)	[400, 640]	(0.5,0.5)
(96, 96)	[400, 640]	(0.5,0.5)
(128, 128)	[450, 640]	(0.5,0.5)
(160, 160)	[450, 640]	(0.5,0.5)

The above windows sizes and overlaps are arrived at trying various options and optimized to arrive a model where it detects the cars with a reasonable time and with very less false positives..

2. Image Pipeline

In the `process_image` function, the `search_windows` function is called to search the windows and find out the hot windows**.** These windows were used to create a heatmap which was then thresholded to remove false positives. This strategy is of course more effective in a video stream where there the heatmap is stacked over multiple frames. `scipy.ndimage.measurements.label` was then used to define clusters on this heatmap to be labeled as possible vehicles. This is shown in lines of the `process_image` function in `p5.ipynb`. The heatmap is created using the `add_heat` function in `functions_required.py.`

3. Example Images Processing

In my code p5.ipynb, the function `process_test_images` iterates through the test images given in folder a folder/file name pattern as an input and processes those images using a function process_image and arrives at the output.

As explained above, I used the Spatial feature, Color histograms and HOG features of the given image to train and detect the cars in the sample images. I trained the model using the training data provided as part of project kit. And it ultimately gave a reasonably good output on the test images.

In order to further optimize, I applied the threshold on the identified hot boxes to remove any false positives. And combine the hot boxes to find out the min and max ranges and arrive at the bounding boxes. The output of the test images are shown below and the output test images are included in the output_images folder of this submission.

Video Implementation

1. Video Output: project_video_output.mp4 is included in the submission file-set. Function process_video as indicated in the p5.ipynb file did the job of video creation. As explained in this write-up already my program p5.ipynb along with functions_required.py which was used to test the output on the sample videos provided. It did reasonably well. In order to optimize the detections, I applied threshold to remove any false positives.

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 the `process_image` function, the window positions containing all positives are captured in the hot_windows variables. It would contain false positives as well. In order to remove the false positives, I am thresholding them to eliminate any detections that are less than 2. The function apply_threshold does the thresholding on the heat map image of the detections. scipy.ndimage.measurements.label() function is applied on that combined hot windows to arrive a smooth detected car positions in a given video using draw_labeled_bboxes function as included in the functions_required.py file.

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?

It is a nice project that gives us a good start on detecting vehicles in the road alongside the autonomous vehicle. The approach that I took is to use a trained LinearSVC classifier to detect vehicles. The model was trained with a reasonable size of training data (~8900 cars and not cars each).

In order to train the model, I used the spatial features, color histogram and HOG features of the given training images. The test accuracy of the trained model came to 99.07% which is a very good accuracy in this case.

Subsequently, I applied the sliding window techniques to arrive at the list of windows to be searched for cars, and with that list, I searched for the vehicles in each frame using the trained classifier.

After a detection is done, they are thresholded to remove false positives, then smoothened and arrived at the final rectangle sizes of the detected vehicles. The above images and video shows the output.

2. Challenges:

1. I had issues with initial detection with lots of false positives. After tweaking the threshold, heat parameters along with re-training the model tweaking the training parameters, I could achieve this result.The final output video took nearly 25 minutes to generate.
2. Being with a core Java programmer mind-set, I always find it is bit slower in dealing with python code which is sensitive to even indentation errors.. Of course no doubt that python has really lots of robust libraries especially for AI tasks. The learning journey is challenging and very interesting.
3. Tweaking the parameters for the feature extraction was quite challenging, multiple iterations of trial and error was needed with some logic behind each change to arrive at the final settings. However there is still lot of opportunities to improve.
4. For detecting just cars alone is this challenging, then detecting pedestrians and other objects are going to be even more challenging. And the worst part is to find the training data set for those other objects. It is better to use a Deep Learning Neural Network instead of a Machine Learning classifier. Even better option is to use any pre-trained models like LeNet or GoogLeNet CNN could be used to detect any objects that come around the autonomous vehicle. I plan to continue work on this further and try with some pre-trained CNN model to predict.