Writeup for Vehicle detection project

Part of Udacity self driving car nanodegree

Sven Eriksson

May 21, 2017

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 classifier 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.

I have done all of these things and the purpose of this report is to describe that.

My code is organized in several files:

- main.py

- development.py
- classifier.py
- undistort.py
- pipeline.py
- udacityCode.py

To run my code from the beginning one begins with running 'classifier.py'. It will train and save a SVM classifier to a file that is used by later steps. I used 'development.py' to test my pipeline on the provided test images. Almost every step of the pipeline for every test image can be seen in the folder 'output_images'. Testing the entire pipeline on a video is done by running 'main.py'.

1 Histogram of Oriented Gradients (HOG)

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

The code used for extracting features from my images are located in several files. Line 32 - 38 in 'classifier.py' that calls the function featureExtractByName located at line 109 in 'udacityCode.py' which in turns calls the function 'featureExtractionTraining.py' located at line 206 in 'udacityCode.py'.

I converted the images to the colorspace YCrCb and extracted HOG feature from the Y color channel. I did also do histograms of pixel intensity for each of the 3 colors and downsampled image in each color channel as features.

The HOG parameters ended up being: Orientations = 8, pixels_per_cell = 8, cells_per_block = 1 (located at line 119 in 'udacityCode.py')

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

After noticing that processing a single frame took almost 20 seconds I had to reduce the number of features in order to speed up processing.

Part of this effort was to remove the HOG features for 2 of the 3 color channels and removing the block normalization. This is combination with other speedup efforts like reducing the number of searched windows gave me a processing time for each frame of approximatly 3 seconds.

So my final choice was driven by speed improvement while at the same time retraining and testing my classifier. I choose the parameters for my feature extraction that resulted in 1% false negatives and 1% false positives but that at the same time gave an accaptable video processing time.

1.3 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 entire classification training process is done in the 'classifier.py' file.

I used the provided images for this project to train. I flipped each image in order to double my dataset. This does not change the histogram of pixel intensity but it did change the rest of the features.

Features were scaled to zero mean and unit variance by the use of a StandardScaler from the sklearn.preprocessing libary. 10 % of the images were kept as a test set and the rest was shuffled and then used to train a SVM classifier with the parameters, kernel = rbf and C = 10. These parameters were found to work well by using the function GridSearchCV in the sklearn.model_selection libary.

2 Sliding Window Search

2.1 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?

The code for window search is found in 'pipeline.py' where I set the different scales and the area of the image that shall be searched with that window size. The actual window search for a particular window search is then done by calling the function find_cars located at line 129 in 'udacityCode.py'.

Not only did find_cars output a list of window coordinates for windows classified as cars but did also output an image for found cars with that windows size and an image with all windows that were searched. These 2 kind of images were saved in the output_images folder as foundCars* and allSquares* where the number following these names indicate the scale of the window size.

I used these created images to test different scales on the previously supplied test images and tuned both the overlap (which is 75%), scales and areas to be search by each scale to work well with the test images. In the end the used scales were 0.5, 0.75, 1, 1.5, 2, and 2.5.

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?

Some examples of processed test images can be found in output_images folder named allFoundCars*.

The pipeline detected the cars close to our vehicle and sometimes also the cars far away. It did only detect a few false positives and I deemed that I could probably filter that out by looking at a squence of frames in the video.

I tried to optimize my classifier by testing different HOG configurations in different color spaces, different histograms of pixel intensity, and also downsampled resolution of images in different color space. As previosuly mentioned I did also use GridSearchCV to find good parameters to my SVM classifier.

In addition to this I did also try a decision tree classifier and naive bayes classifier. I wasn't able to get there working very well so I couldn't use the 3 classifiers together.

3 Video Implementation

3.1 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.)

Final video can be found in the github repository as:
'processed_project_video_h4.00_t1.00.mp4'

It can also be found at `http://svene.se/carND/tracking_final.mp4`

The unfiltered version of the final video can be found in the github repository as:
'processed_project_video_noHistory.mp4'

It can also be found at `http://svene.se/carND/tracking_raw.mp4`

3.2 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.

I am using a heat map to combine overlapping bounding boxes. By calculating this over several frames and using a threshold I am also exluding quite many of the false positives if you compare the two previosly linked videos.

The code for this is located in 'pipeline.py at line 31 - 38 and in the functions add_heat, apply_threshold, and draw_labeled_bboxes in 'udacityCode.py'. These functions are at lines 68 - 100.

For each frame this is how I use create the heatmap from the previous heatmap. I have copied add_heat, apply_threshold, and draw_labeled_bboxes from one of the course lessons.

Extracted code from 'pipeline.py' related to this:

```
heatmap *= (historyFactor -1)
heatmap = add_heat(heatmap, listOfAllBoxes)
heatmap /= historyFactor

labels = label(apply_threshold(heatmap, threshold))

allFoundCars = draw_labeled_bboxes(image, labels)
```

The final video is created with a historyFactor of 4 and a threshold of 1.

4 Discussion

4.1 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?

One problem that I had was that I was forced to choose between extracting more features and processing time. I choose a set of feature so that processing the video took no more than an hour. There are a few options to avoid this choice. One way is to understand that the window searches for different scales are independent from each other and could be done in parallel. This should probably reduce the amount of time needed by a factor of 4-6.

Another option is to use a different classifier that isn't based on extracting this number of features. It should probably be possible to use a neural network like we did in the Traffic sign classifier project. If we base this on transfer learning of one of the existing networks being used to classify images already it shouldn't take too long to train.

It should also be possible to improve the training data by gathering more data. Experimenting a bit more with more color spaces and parameters might improve the classification as well.

If one is able to train a few classifier that fails on different images it should be possible to use them together and require that they all agree for car classifications. This should remove false positives and reduce the need for filtering based on a sequence of frames.

Even if my pipeline works reasonably well for the 'project_video.mp4' it won't work for any real time application as each frame takes a few seconds to process.