Vehicle Detection and Tracking

Udacity Student

Udacity
Self-driving car Nanodegree program
Project 5

Contents

1 Introduction

This document serves as the project writeup for the Udacity Self-driving car Nanodegree Program, term 1, project 5, vehicle detection and tracking. The project code, as well as the present document can be retrieved at https://github.com/kacl780tr/carnd_project_05.

The goal of this project is to develop a processing pipeline to identify and visually indicate other vehicles in the immediate vicinity from on-board camera images.

2 Project structure

This project was built using the infrastructure created for the previous project, project 4 (https://github.com/kacl780tr/carnd_project_04), and therefore also uses the object-oriented design. The pipeline detailed here retains the functionality for lane lines, with the addition of the vehicle detection capability. We will not further describe the lane lines processing here, since it is adequately explained elsewhere, although we have renamed some of the files to better reflect the overall design, which has been refactored slightly to more easily allow the vehicle detection capability to be added.

The project code is organized into several files, we note which files are specific to the retained lane lines functionality:

detection.py the main file including processing pipeline setup, input and output of image and video files

calibration.py code related to camera calibration

detection_feature.py components for classifier feature extraction

display.py support functions for image display

pipeline.py the main processing pipeline objects

stage_detect.py pipeline component for vehicle detection and tracking

stage_lane.py pipeline component for lane line identification

stage_path.py components for path construction (lane lines)

stage_region.py components for perspective region identification (lane lines)

tools_data.py functions related to data compilation and handling

tools_image.py functions related to image manipulation

tools_net.py helper function related to neural network training, testing, and deployment

tools_perspective.py functions and objects related to perspective transformations (lane lines)

tools_region.py functions related to identification and manipulation of image regions (lane lines)

tools_svm.py functions related to support vector machine training and testing

tools_tflow.py functions related to Tensorflow neural network library

tools_train.py functions related to neural network training

tools_window.py objects for sliding window and detection heatmap handling

3 Classifier design and training

In order to detect vehicles in the video stream, we first must be able to differentiate them from the background. To this end, a classifier was designed and trained. The project materials suggest the use of a support vector machine (SVM), in conjunction with a hand-designed feature set. Another approach might be the use of a convolutional neural network (CNN), which effectively allows the classifier to detect its own features during the training process.

3.1 Training data

A dataset consisting of images of "cars", and "non-cars" was provided by Udacity. This consisted of

- 17,760 total images
- 8792 car images
- 8968 non-car images
- image shape (64, 64, 3) (*height, width, channel)*

Two samples of each of "non-vehicle" and "vehicle" categories for this dataset are shown in figure 1.

Figure 1: Samples from base training data

3.2 Support vector machine classifier

For the SVM classifier, one must define the features manually, and to this end, a number of different feature combinations were elaborated. Since the extraction of features must take place both during training, and during the functioning of the pipeline, feature extraction was implemented in terms of objects (see file **detection_feature.py**), with the extraction object stored on disk in the same file as the trained SVM, for convenience. The SVM training process consisted of using the scikit-learn grid search facility with 5-fold cross-validation, across a range of regularization values for both linear and radial basis function kernels.

The feature components in table 1 are described as

Histogram *color channel*/*bin count*

Spatial binning (*horizontal size*, *vertical size*)

HOG *color channel*/(*angle size* x *pixels per cell* x *cells per block*)

A series of SVM fits with different feature sets is shown in table 1. The accuracy appears to be very good. In fact, the accuracy for the smallest feature sizes remains competitive with that of the largest sizes, which is a somewhat counter-intuitive result. This could indicate that our training and test data was not sufficiently randomized, despite our efforts to randomize the manner in which the data set was compiled. To check this possibility, we set out to obtain further training data.

Color space	Histogram	Spatial binning	HOG	Feature size	Test accuracy
HSV	2/24	(32,32)	1/(9 x 8 x 2)	4860	0.9913
HSV	2/24	(32,32)	0/(9 x 8 x 2)	4860	0.9932
YUV	2/24	(32,32)	0/(9 x 8 x 2)	4860	0.9923
YCrCb	2/24	(32,32)	0/(9 x 8 x 2)	4860	0.9927
LUV	2/16	(16,16)	0/(7 x 8 x 2)	2156	0.9941
YUV	2/16	(16,16)	0/(7 x 8 x 2)	2156	0.9941
YCrCb	2/16	(16,16)	0/(7 x 8 x 2)	2156	0.9935
LUV	-	-	0/(9 x 8 x 2)	1764	0.9903
YCrCb	2/16	(16,16)	0/(5 x 16 x 2)	1036	0.9935
LUV	-	-	0/(5 x 8 x 2)	980	0.9918
LUV	2/8	(8,8)	0/(7 x 16 x 2)	452	0.9966
YCrCb	2/8	(8,8)	0/(5 x 16 x 2)	380	0.9952
LUV	2/6	(6,6)	0/(5 x 16 x 2)	294	0.9947
YUV	2/6	(6,6)	0/(5 x 16 x 2)	294	0.9947

Table 1: Selected SVM feature sets and test accuracies for training with base dataset

3.3 Additional data

Fortunately, Udacity also provided additional data. These consist of images obtained from a vehicle camera, along with an accompanying data file which defines bounding boxes for various objects in the images. For the **object-detection-crowdai** dataset, there are bounding boxes for cars, trucks and pedestrians.

We then devised an algorithm to compile additional data in the same format. The algorithm is implemented as the routine **compile_extra_data()** in the file *tools_data.py*. The basic steps of the algorithm are as follows:

- an image from the set is selected at random (without replacement)
- the image is passed through a histogram equalization process
- a list of the "car" and "truck" bounding boxes for that image is compiled from the *labels.csv* file
- the "car" bounding boxes are sorted in order of descending size
- if the largest car box has both width and height $\geq$ 128 pixels then
  - the car box is resized to 64 x 64 and added to the "car" image list
  - the box is segmented into sliding windows of size 64 x 64 with 50% overlap
  - one of these boxes is selected at random and added to the "car" image list
- otherwise we process the image for "non-car" image candidates as
  - a random bounding box of size 128 x 128 is generated
  - if this bounding box intersects with any of the bounding boxes for the image then generate a new box
  - otherwise extract and resize the box contents and add to the "non-car" image list
  - repeat until the number of "non-car" images matches the number of "car" images, or until we have failed to generate a non-overlapping box 10 times

The histogram equalization is applied in an attempt to correct what seem to be very dark images. The reason for selecting a sub-image from each of the suitable car images was to try to improve the ability of the classifier to detect cars when the sliding window partially overlaps the image of a car. The routine **compile_extra_data()** continues until it has generated the requested number of car and non-car images, or it runs out of candidate images. We used the routine to generate a further 5000 car images, along with same number of non-car images. Samples of the additional data are shown in figure 2.

Figure 2: Samples from additional training data

An number of the more efficient SVM candidate classifiers from table 1 were then re-trained on the augmented dataset, made up of the original data plus the 10,000 images extracted from **object-detection-crowdai**. The training results are detailed in table 2.

Color space	Histogram	Spatial binning	HOG	Feature size	Test accuracy
LUV	2/8	(8,8)	0/(7 x 16 x 2)	452	0.9795
LUV	2/6	(6,6)	0/(5 x 16 x 2)	294	0.9784
LUV	-	-	0/(5 x 16 x 2)	180	0.9624
YCrCb	2/8	(8,8)	0/(7 x 16 x 2)	452	0.9777
YCrCb	2/6	(6,6)	0/(5 x 16 x 2)	294	0.9791
YUV	2/8	(8,8)	0/(7 x 16 x 2)	452	0.9769
YUV	2/6	(6,6)	0/(5 x 16 x 2)	294	0.9782

Table 2: Selected SVM feature sets and test accuracies for training with augmented dataset

It can be seen that the test accuracy is somewhat reduced when the augmented dataset is used, which is to be expected, given a fixed feature size. Due to lack of time, training of the larger feature sets was not attempted. The F-score was also examined (not shown) in order to determine if there was a disproportionate number of false positives versus false negatives. The F-score was essentially equal to the test accuracy in all cases.

3.4 Convolutional neural network classifier

Unfortunately, despite the effort expended on the SVM classifiers, satisfactory results were not obtained. It was therefore decided to design and train a convolutional neural network classifier in an attempt to improve the outcome.

The network configuration for training is described in table 3.

Layer	In	Out	Notes
Input	-	$64 \cdot 64 \cdot 3$	Training input
Convolution $(5 \cdot 5)$	$64 \cdot 64 \cdot 3$	$64 \cdot 64 \cdot 64$	Stride $= 1$, padding $=$ same
Batch normalization	$64 \cdot 64 \cdot 64$	$64 \cdot 64 \cdot 64$	-
Leaky ReLU activation	$64 \cdot 64 \cdot 64$	$64 \cdot 64 \cdot 64$	Common leak rate
Convolution $(5 \cdot 5)$	$64 \cdot 64 \cdot 64$	$64 \cdot 64 \cdot 128$	Stride $= 1$, padding $=$ same
Batch normalization	$64 \cdot 64 \cdot 128$	$64 \cdot 64 \cdot 128$	-
Leaky ReLU activation	$64 \cdot 64 \cdot 128$	$64 \cdot 64 \cdot 128$	-
Dropout	$64 \cdot 64 \cdot 128$	$64 \cdot 64 \cdot 128$	Common retain rate
Convolution $(5 \cdot 5)$	$64 \cdot 64 \cdot 128$	$64 \cdot 64 \cdot 64$	Stride $= 1$, padding $=$ same
Batch normalization	$64 \cdot 64 \cdot 64$	$64 \cdot 64 \cdot 64$	-
Leaky ReLU activation	$64 \cdot 64 \cdot 64$	$64 \cdot 64 \cdot 64$	-
Dropout	$64 \cdot 64 \cdot 64$	$64 \cdot 64 \cdot 64$	-
Convolution $(3 \cdot 3)$	$64 \cdot 64 \cdot 64$	$64 \cdot 64 \cdot 32$	Stride $= 1$, padding $=$ same
Batch normalization	$64 \cdot 64 \cdot 32$	$64 \cdot 64 \cdot 32$	-
Leaky ReLU activation	$64 \cdot 64 \cdot 32$	$64 \cdot 64 \cdot 32$	-
Dropout	$64 \cdot 64 \cdot 32$	$64 \cdot 64 \cdot 32$	-
Convolution $(7 \cdot 7)$	$64 \cdot 64 \cdot 32$	$8 \cdot 8 \cdot 8$	Stride $= 8$, padding $=$ same
Leaky ReLU activation	$8 \cdot 8 \cdot 8$	$8 \cdot 8 \cdot 8$	-
Convolution $(8 \cdot 8)$	$8 \cdot 8 \cdot 8$	$1 \cdot 1 \cdot 1$	Training output (scalar)

Table 3: CNN architecture as configured for training

This network was implemented in Tensorflow[1], and trained on the augmented data, as described in section 3.3, with the best test accuracy of 98.3%.

The network is fully convolutional, with no fully connected layers. This means that the parameterizations of the various layers do not change when the input size is changed. We are thus free to reconfigure the network to directly accept a large slice of the frame to be processed. The network will then output a prediction signal for a large number of (64 x 64) input patches.

4 Processing pipeline

The processing pipeline devised here is composed of the following stages:

1. correction for camera distortion
2. identification and painting of lane lines as per project 4
3. identification and tracking of proximal vehicles and painting of their location boxes

While the pipeline is based heavily on that from project 4, the code has been refactored somewhat to allow a cleaner integration of the detection code. It also allows the lane line detection and drawing to be switched on and off with a configuration switch.

We now discuss each step in further detail.

4.1 Main pipeline

As noted in section 2, the main processing pipeline is instantiated and configured in the file **detection.py** The objects which constitute the main processing pipeline are found in **pipeline.py**. Upon instantiation, the *Pipeline* object carries out the following operations:

1. loading of the required camera calibration *Camera* object
2. instantiating the *LaneBuilder* object, responsible for lane line detection
3. instantiating the *DetectionBuilder* object, responsible for detecting and tracking vehicles

The pipeline object can be configured to trace its activity, by writing images produced by the steps of the process into a list object. The pipeline will then call a *TraceHandler* object, if such is configured. Two *TraceHandler* subclasses are currently available: *TraceDisplay* will display the image trace sequence to the user, while *TraceDump* will dump the trace images to disk. This facility is useful for debugging.

4.2 Camera calibration

Code related to the camera calibration is found in **calibration.py** The main pipeline object retrieves the *Camera* object upon construction, and fails if it is unable to do so. The function **calibration_retrieve()** can either compute the calibration from a series of suitable images on the fly, or it can be loaded from disk, if already computed. Such is the case here.

An example of a raw camera image is shown in figure 3, while the same image, corrected by the camera object, is shown in figure 4.

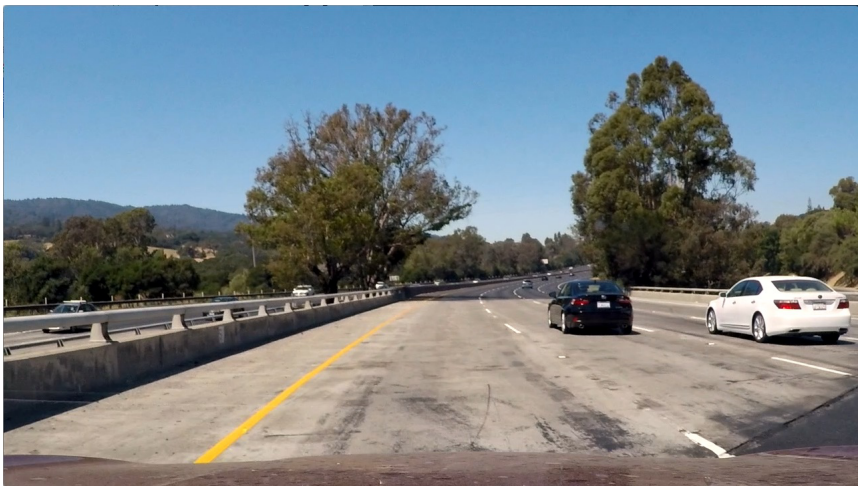

Figure 3: Example of raw camera image

Appendix A shows a further test of the camera calibration.

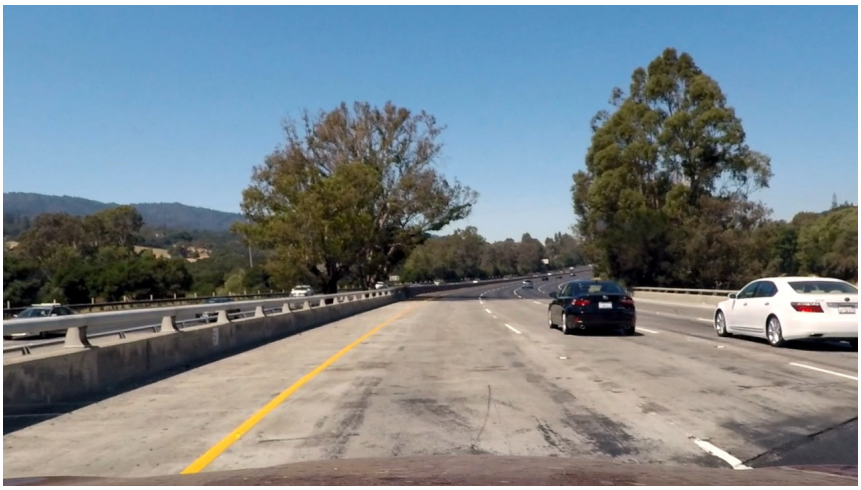

Figure 4: Corrected version of image in figure 3

4.3 Vehicle detection

As noted in the section on classifier training (3), both SVM- and CNN-based classifiers were trained and implemented. Here we will describe both approaches.

4.3.1 CNN detection process

For the CNN classifier, the output shape is determined by the input shape. As shown in section 3.4, for a (64 x 64 x 3) training image, the output shape is (1 x 1 x 1). For the purposes of vehicle detection, we select the most relevant part of the input frame, which spans the entire width horizontally, and spans height $= 396 \cdots 662$. Thus the size of the image submitted to the classifier is (266 x 1280 x 3). This results in an output shape of (27 x 153 x 1). Thus we effectively sample 4131 windows of size (64 x 64 x 3).

These windows are mapped onto the image using a *WindowGroup* object (**tools_window.py**), which need only be constructed once at setup time. The search windows are shown drawn onto the corrected image in figure 5.

Figure 5: Search windows for CNN classifier

A detection probability threshold is then applied to the output of the classifier, which is then flattened and used as a window prediction indicator. An example of predicted windows drawn on a frame is shown in figure 6.

Figure 6: Example window prediction for CNN classifier

The predicted windows are then summed into a heatmap, with each window adding 1.0 to the value of the pixels that it covers. The set of detections across video frames are managed by two objects: the *HeatMap* object represents the heatmap from a single frame, as shown in figure 7, while the *HeatMapGroup* object is responsible for combining the *HeatMap* from several consecutive frames into a cumulative heatmap. (**tools_window.py**)

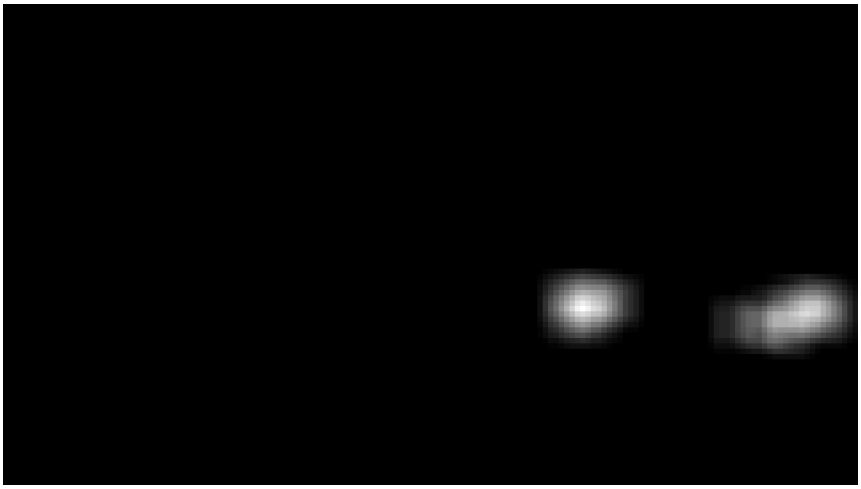

Figure 7: Example heatmap for CNN classifier

The *HeatMapGroup* maintains a queue of the N most recent frames, as well as the cumulative heatmap over those frames. When a new frame is processed, the oldest heatmap is subtracted from the cumulative heatmap, and that for the newest frame is added. The cumulative heatmap data is then thresholded to a value given by a threshold factor $\in (0, 1]$, multiplied by the heatmap level value. The level is maintained by the *HeatMapGroup* as the sum of the maximum values in all individual frame heatmaps.

In the current implementation, the *HeatMapGroup* converts each individual frame heatmap into a binary map, and the threshold effectively becomes a detection floor for the number of detections over N frames. An example of a cumulative heatmap is shown in figure 8. The process of summing and thresholding over multiple frames is key to the removal of false positive detections.

Figure 8: Cumulative for CNN classifier

The cumulative heatmap is then segmented using the SciPy[2] ndimage labeling tool and the predicted tracking windows are drawn, as shown in figure 9.

Figure 9: Cumulative for CNN classifier

The final output for the vehicle detection is then shown in figure 10.

4.3.2 SVM detection process

For the SVM classifier, the detection process differs only slightly from that of the CNN classifier, therefore we will not only the differences, and will not include illustrations in the interests of brevity. In this case, the fixed set of search windows are constructed when the first frame is submitted to the *DetectionBuilder* component of the pipeline. Until the first frame is submitted, we do not know the actual frame dimension. However, once this is known, all subsequent frames are assumed to have the same dimensions.

The search window set is constructed by the *WindowBuilder* object, which is provided with a set of parameters when it is constructed. These parameters include:

rules a list of rules for window construction. These rules are specified as 3-tuples (*magnification*, *start height*, *end height*), where the start and end heights are specified as fractions of the vertical image

Figure 10: Final output for detection with CNN classifier

dimension. Magnification is specified relative to the baseline size, the image size expected by the classifier.

overlap a 2-tuple specifying the amount of (*horizontal*, *vertical*) window overlap

x_stretch the factor by which the window width should be magnified with respect to the height

The *WindowBuilder* object creates and returns a *WindowGroup* object, which contains a list of *Window* objects (**tools_window.py**). Each *Window* object is able to

- extract its coverage region from an image
- draw itself onto an image
- update its coverage region onto a heatmap array

The *WindowGroup* object makes working with sets of windows more convenient. In particular, it allows working with subsets of its windows, such as those identified by the classifier as containing vehicles.

5 Results

The results are shown in section 4.3.1 for the CNN classifier, as well as in the project video (**project_video.mp4**). Unfortunately, Tensorflow is somehow incompatible with the lane lines facility, and we were unable to compile a video with both lane lines and vehicle tracking, using the CNN classifier.

We were able to do this with the SVM classifier, and this video is included as **project_video_svm.mp4**, which was created using the SVM classifier with feature set as

Color space YUV

Histogram channel 2 with 6 bins

Spatial binning (6, 6)

HOG channel 0 with 5 angles, 16 pixels per cell, 2 cells per block.

The achieved results could be improved further. False positive detections proved the most bothersome problem for either of the classifiers. It was very difficult to eliminate false positives without also removing the detections.

Speed was also an issue with the SVM classifier. It was difficult to find a feature set small enough to be fast, which retained the power to differentiate vehicles from non-vehicles. The CNN classifier is

substantially faster, even with the large number of patches being considered, through the use of GPU computation. It could easily be made faster by restructuring the network to reduce the number of patches.

The most promising avenue of improvement is likely to be improvements to the CNN architecture, along with additional data collection. This would hopefully improve the precision and recall of the CNN, reducing the need for post-processing. There is also the possibility of augmenting the prediction network with deconvolution layers, and then applying the SciPy[2] segmentation directly to the CNN output.

A Camera calibration test

Figure 11 shows the calibration test image before correction, while figure 12 shows the same image after correction. This serves to demonstrate that the camera distortion correction is functioning as intended.

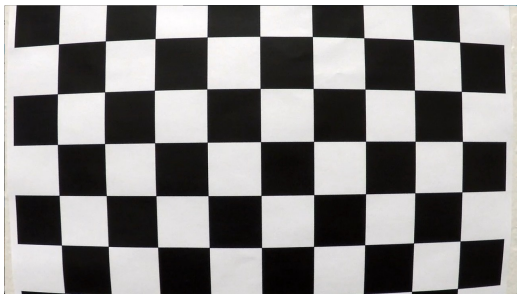

Figure 11: Camera calibration test input

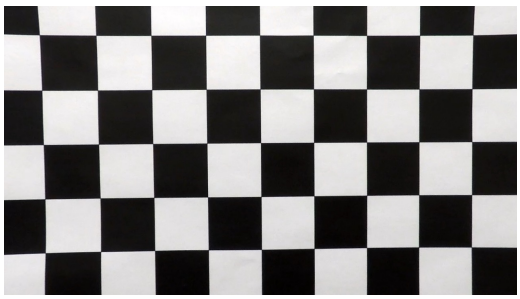

Figure 12: Camera calibration test output

References

[1] Google and other developers. TensorFlow: A machine learning software system, 2015. Available from tensorflow.org.

[2] E. Jones, T. Oliphant, P. Peterson, et al. SciPy: Open source scientific tools for Python, 2001–.