Project ND Self-Driving Car -Advanced Lane Finding-

Introduction

In this report we will go through the designed pipeline which detects the lane lines of a road in a changing environment due to lines of different colors, shadows, curves and slopes. The input data would be the video from the dash camera of a car. We will go through every step, analyzing the operations and changes made to the input data until we obtain the output data. This will be the same video as the input data with the detected lines added to it.

Part 1: Camera Calibration

We need to calibrate our camera in order to correct the aberrations and distortions of itself. This step needs to be made only once and not for every frame. Once we obtain the calibration parameters of our camera, we can use for every frame received with the same camera.

For that purpose we created the function *Camera_Calibration()*, which takes the chessboard images from “camera_cal/” .

We will detect the corners of every chessboard image and then use the function *cv2.calibrateCamera* in order to obtain the calibration parameters of the camera: “mtx and dist”. We can apply then the function *cv2.undistort* to every frame of the dashboard in order to correct the distortion of the camera. As an example, we can check the function with one of the chessboard images:

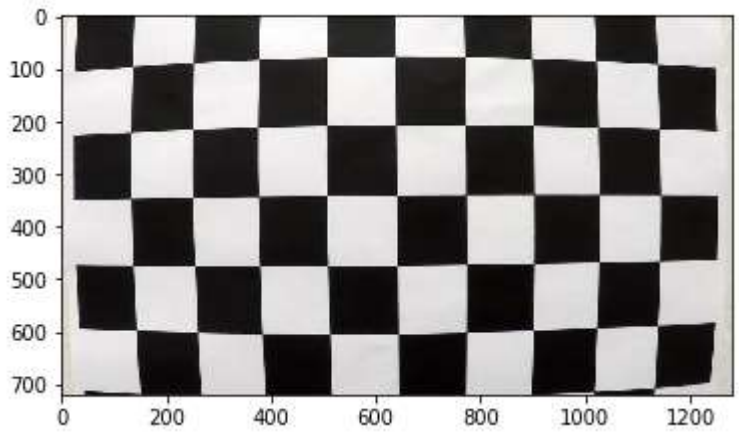

Figure 1: Original chessboard image

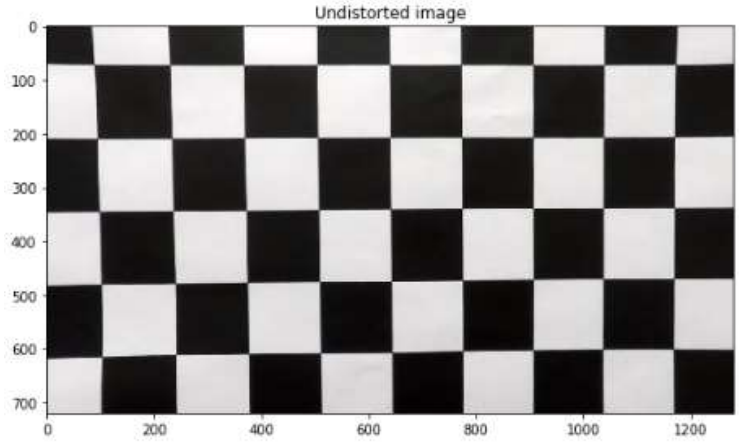

Figure 2: Undistorted chessboard image

As we can see, the image was undistorted properly and then the calibration process is finish. We will save the values of mtx and dist to be able to use them throughout the pipeline.

Here is a link to [the corresponding module in Github.]

Part 2: Applying thresholds

One basic step to detect the lane lines are the different filters that we can apply to our image. For that, there are different technics that we learned during our lesson. We have tried the following when handling this problem:

- Applying Sobel to x
- HLS transformation and filtering the Channel S (Saturation)
- HSV transformation and filtering all three channels to distinguish white and yellow

As it was illustrated during the lesson. We started applying Sobel x to the images and then adding another filter for the channel S after transformation from RBG to HLS. The detection of the lines worked decently but we the detection of the lines under the shadows was problematic.

When the image does not have any shadow, the filtering using the Channel S works fine:

Figure 3: Threshold for S-Channel

But after in the case of shadows we get the following result:

Figure 4: Threshold for S-Channel with shadows

Adding the filtering with Sobel X, some detection is improved but the shadows will still be detected as lines.

Figure 5: Threshold Sobel x

To avoid the problem with shadows, we can try to detect the yellow and white color after transforming the image from RGB to HSV. For doing that, we need to create thresholds for all three channels of an HSV-Image. For both white and yellow:

```
hue_thresh_yellow_lower = np.array([10, 80, 50])
hue_thresh_yellow_upper = np.array([60,255,255])
hue_thresh_white_lower = np.array([0, 0, 200])
hue_thresh_white_upper = np.array([150,25,255])
```

Since I did not know how the range of white and yellow in HSV, I look for help online. An example of valid HSV-values for white and yellow can be found here.

And then we will create both masks that filter only the pixels of the image which fall into the presented thresholds. That means, we will only keep the pixels that are white or yellow. By doing this we solved the problem with the shadows.

Figure 6: Threshold detecting white and yellow

Now, if we add Sobel X to our detection algorithm, we will get the desired results:

Figure 7: Threshold with color and Sobel X

Part 3: Undistort and warp

After having successfully detected the lane lines, we will use the previous obtained calibration values of the camera to undistort the image and then proceed to warp the obtained image to build a bird-eye view of the road. This will help us when determining the polynomial that will correspond to the lane lines.

For doing this, we used the function *cv2.undistort* with the obtained values of the calibration of the camera. After that, we warped the image using the following source and destination points obtained by trial and error a comparison with different images. The best way to get a correct image consisted on using an image of a car driving straight forward and transforming it to a rectangle.

Source	Destination
200, 720	300, 720
1080, 720	980, 720
584, 455	300, 0
696, 455	980, 0

After that we use the following functions and will obtain the corresponding warped binary image:

```
M = cv2.getPerspectiveTransform(src, dst)
warped = cv2.warpPerspective(distort, M, image_size, flags=cv2.INTER_LINEAR)
```

Figure 8: Image undistorted and warped

Part 4: Obtain histogram to detect base point of the lines

We need now to obtain the polynomial fit of both lines as a function f(y) = x. These lines will depart from a point that intersect the x axis at its base. We can find that point by creating the histogram obtained by the sum of all the white pixels through the y axis. We will take only into account the bottom half of the image:

```
bottom_half = warped[warped.shape[0]//2:,:]
histogram = np.sum(bottom_half, axis = 0)
```

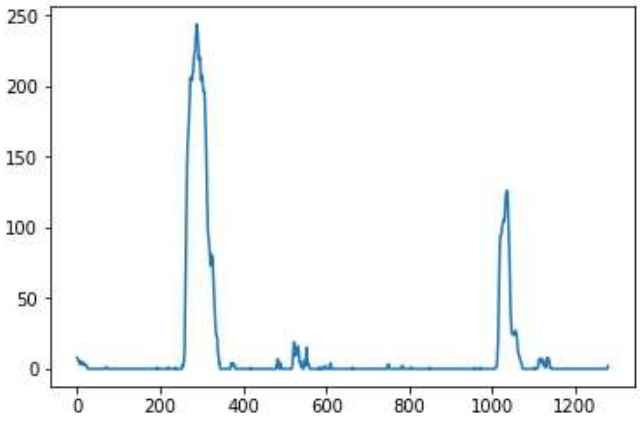

Figure 9: Histogram

Part 5: Finding the lines

As we did during the lesson, now it's time to find the polynomial that follows both lines. We can do that by iterating through n windows. We will start from the obtained base point in the Part 4 and then we will find the pixels in a rectangle which is center in the previous top point and a margin left and right which is adjustable as an hyperparameter. I used a small margin to try to make the pipeline more robust, since the lines should not change very abruptly.

The iteration through the windows will be made only for the first frame or every time the lines are lost (See Part 8: Sanity checks). This operation is highly resource-consuming, and it is very probable that the lines of one frame are similar to the lines of the previous frame. If no new lines are found, a reset will be made in order to iterate through the n windows again in order to find them. To know if a line has been found or not, we used the Class Line () in which we save the current parameter which defines if a lines was founded: *self.detected_lines*.

Every frame, we will look into that parameter and depending of its value, we the program iterates through the n windows or not.

The implementation of this part of the code can be found at the parts #5.1 and #5.2 of the python code.

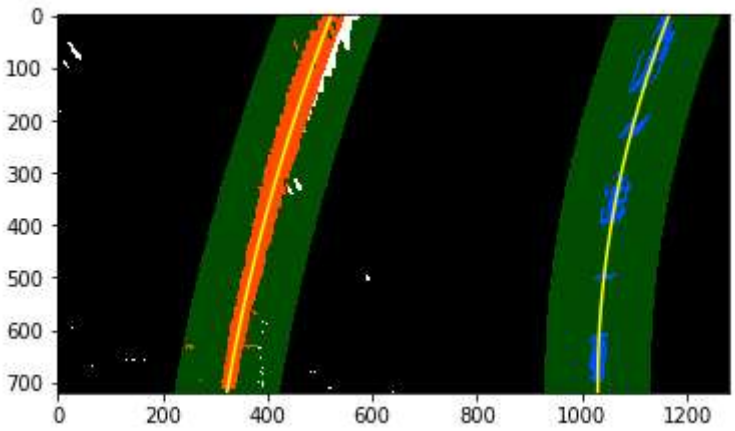

Figure 10: Detected lines

Part 6: Curvature and distance to the center

The right and left curvatures were calculated for every frame but only the average of the last 25 frames are shown in the image. They range normally around 1000m except for when the way is straight. In this case the curvature tends to the maximum.

The center is calculated in the same manner as the curve. For doing the calculations we defined two functions inside the Class Line ().

```
  def average_curve(self, curvature):
    self.radious_of_curvature.append(curvature)
    if len(self.radious_of_curvature)>=25:
      average_curvature = np.mean(self.radious_of_curvature)
      del self.radious_of_curvature[0]
    else:
      average_curvature = 0
    return (average_curvature)

def average_center(self, center):
    self.distance_center.append(center)
    if len(self.distance_center)>=25:
      distance_center_avg = np.mean(self.distance_center)
      del self.distance_center[0]
    else:
      distance_center_avg = 0
    return (distance_center_avg)
```

We keep showing the average of the last calculated values in order to minimize the effects of the fluctuations.

Part 7: Tracking

In order to do sanity checks we created a Class which can be used to track the most important indicators of the detected lines, such as the curvature of the lines, the detection of the lines

and the distance to the center. We will use that class to keep the data of the last 25 frames and diagnose our code. For doing that we used the Class Line() as suggested in the project.

Part 8: Sanity Checks

We will use the curvature to detect possible failures during the detection of the lines. The Class Lines () would calculate the mean value of the curvatures in the last 25 frames and will assure that the curvature as normal values. When this happen, the program is reset and the pipelines iterate again through the n windows.

Part 9: Unwarping and merging with the original image

Once the lines had been correctly detected in the corresponding frame. We must merge the detected lane lines with the original image. For doing that we only need to use again the function *cv2.warpPerspective* back to its original plane.

We also added the curvature values of the right and left lines. When the pipeline does not detect a curve, the corresponding text will be shown. In the same manner the distance from the car to the center of the image is calculated and presented.

By doing that we achieve the following result:

Figure 11: Final result

Part 10: Pipeline (video)

Here is a link to my video result.

Part 11: Discussion

1. *Briefly discuss any problems / issues you faced in your implementation of this project.*

I had some problems defining the general strategy for my project. I did start by writing independent functions and modules that communicate with each other. At the end when I was trying to sort it out together, I had some problems of communication between the different functions. To solve that, I merged everything into one only continuous pipeline.

Since I decided to tackle the problem of tracking and sanity checks at the end, I had to start all over again in some parts of the project. I created then the Class Line() and therefore had to delete some global variables and minor functions. I think I could have got a much cleaner result if I had established a clear tracking strategy from the begging.

Another problem I had was the detection of the lines in shadowy spots. I spent a long time comparing different thresholds and methods until I came up with the solution of filtering the colors. After I tried several thresholds with it, I had to research in internet in order to find the thresholds which could detect yellow and white. Once I had them, I could detect the lines better than with any other methodology.

2. *Where will your pipeline likely fail?*

I think my pipeline will fail when some other high contrast objects, for example, a white car, happen to pass near the main lines. It is possible that the histogram shows in that case two peaks or one of the lines merges with another line formed by the passing objects.

The farther spot could be a problem too, since there the lines are not so clear and big. They could get confused with other objects and thus change the polynomial fit of the whole line.

I run the challenge in order to find more weak spots of my pipeline. I saw that the lines are lost in the tunnel and no pixels are found. I think that could be improved by changing and adapting the chosen thresholds.

3. *What could you do to make it more robust?*

More sanity checks with deeper changes can be helpful. The only measure we take currently when detecting that something is going wrong is to reset the calculation of the lines and starting all over again iterating through the n windows. But what would happen if the detected pixels form a line clearer to the program than the true line? A reset would not work in that case.

I think that in that case we could give more priority to the pixels that are found where the line should be. To know where the line should be, we just have to use the average or ideal distance between the lines throughout the y – axis. Then we should for the line to be found in that area instead of nearer or further away. This could ensure that if one line is correctly found, the

other could be estimated properly. This should work by adding a parameter to the Class Line (), Line_surely_detected and another variable containing a list of every distance between the lines throughout the y – axis.

Part 12: Modules in Github

General Pipeline (Main)

Camera Calibration

Video Reproduction