Finding Lane Lines on the Road

The project was intensive, challenging, and also a great learning experience. The goal of the project was to build a pipeline that can be used for determining the lane line of a given road image. The robustness of the code was later tested by running the code on 6 different test images. Apart from that, towards the end of the project, we had an opportunity to test our code on a few sample videos. Even though there were moments of confusion, the overall experience of implementing the code was very positive. When we could see the code fared well in many images and videos, the confidence that it gives is immense. However, there were few shortcomings which I would discuss along with the write up on project discussions.

Project Pipeline:

The project pipeline includes the following steps:

1. **Read image**
2. **Convert the image to grayscale()**
3. **Use appropriate kernel and apply gaussian_blur()**
4. **Define hyper-parameters and find edges using Canny()**
5. **Find the region of interest**
6. **Define parameters and execute hough transform to find lines**
7. **Average and Extrapolate to make the lines smooth**
8. **Draw lines cv2.lines function**

Details of the pipeline are explained below:

The image is first converted to grayscale to reduce the image to one channel having values from 0-255. The grayscale is smoothened using a Gaussian filter and a canny filter is applied over it to sharpen its edges. Now, we have an image with all the edges in the images visible clearly as shown in figure 1.

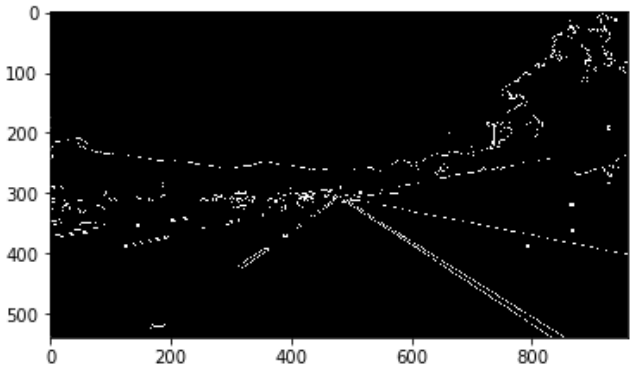

Figure 1: Canny filtered Image

However, we are only interested in the region where the line marking is present, that is the region containing the road. For this purpose, the vertices of the area covered by the road is chosen and feed it to the fillpoly() command to extract the region of interest. Figure 2 shows the masked image

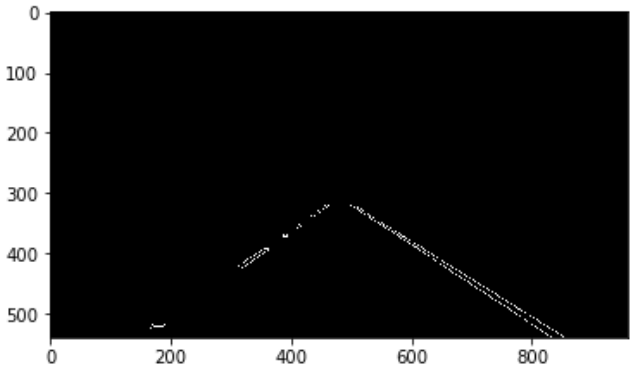

Figure 2: Masked image

Now, lines in the image are extracted using Hough transform by carefully tuning the input parameters. From the obtained lines, we now unfold the coordinates of each line. The cv2.line function (inside the draw_line function) is used to draw the line using these coordinates. By obtaining the line image and combining this with the original image, we can see the drawn lines over the lane line of the road (Figure 3).

Figure 3: Lines over lane

But as we can see the lines we drew are not smooth. Two functions, namely draw_coordinates and average_slope, are defined to address this issue. The draw_coordinate function is called inside the average_slope function.

average_slope function: The average_slope fucntion takes image and the lines as input and returns the averaged line coordinates, which is inturn used to draw the line in draw_line function.In this function, the line coordinates are extracted for each line, found using hough transform, and given to polyfit function which returns slope and intercept of the lines. Then we make use of the fact that the slope of the left lines will be negative and those of right line are positive. Using this fact, an array of slope and intercept of lines having negative slope (left lane lines) and positive slope (right lane lines) are constructed. Next, the average of slopes and intercepts of both left lines and right lines are found using the np.array function. Utilizing these parameters (slope average and intercept average), the coordinates need to be obtained to draw the image.

make_coordinates: Make coordinate function computes the coordinates for the input slope and intercept values from the average_slope function. y1 coordinate will be botton row value and y2 is computed as 3/5th of y1. x1 and x2 is found using line equation.

Finally, these coordinates are used to draw the new lines, which are combined with the original image to get the final image as shown in figure 4.

Figure 4: Averaged lines over lane

Conclusion and discussion

All the functions defined in the above section are fed into a pipeline function which takes the image as input to execute other examples and videos in the following exercise. While writing the code, at first the major thing that caused confusion was the write-ups between the functions. Even though it was crucially helpful at some points, towards the end as the code became lengthy the write-ups was making thing little messy for me. Hence, took the decision to clear everything other than code.

In the beginning, the topics taught in the classes were more than enough to fare forward. But, the averaging of the slopes and intercepts to make the line look smoother was a new concept to learn. But the end of the day, these challenges is what makes us confident and happy.

I was excited to see the code working well with many of the images and the videos. Even though for few images the line was not detected accurately. The reason might the possible change in hyper-parameter values chosen for different functions. The most pleasant surprise was to see the output of the first video, which worked amazingly well. For the second video, even though there was some flickering and intersection of lines, the code executed pretty well. The reason for intersection might be the length of y2 coordinates being outsized at the curves. The lane line detection of the 3rd optional video went terribly wrong, for which the reason might be the change in angle of the camera in the video. Coordinates for which the program was written is not matched with those in the video.

Having seen outputs of code executed on various images and videos, it is understood that a more generalized method is required for working in a real environment. The results obtained by executing code over different test images are given below.

Figure 5: Output of solidWhiteRight

Figure 6: Output of solidWhiteCurve

Figure 7: Output of solidYellowCurve

Figure 8: Output of solidYellowCurve2

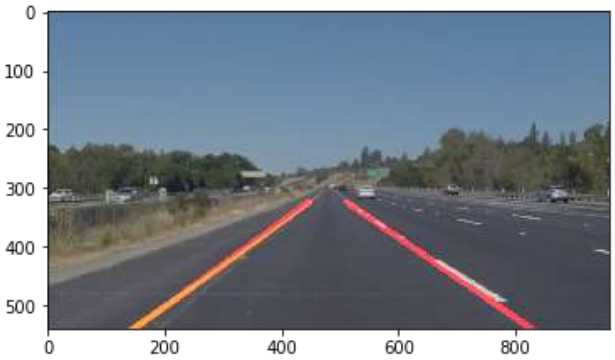

Figure 9: Output of solidYellowLeft

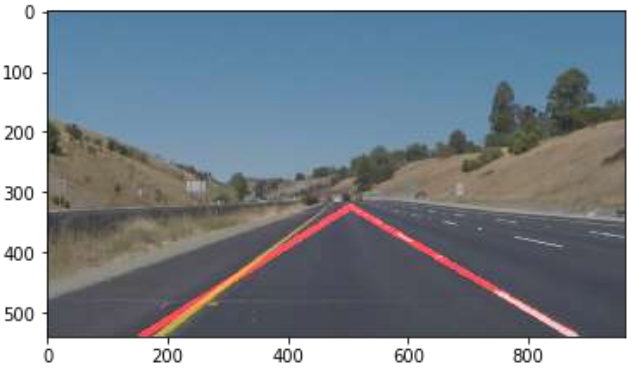

Figure 10: Output of whiteCarLaneSwitch