The first project of the Self driving car engineer Nano degree program started with

Finding Lanes on the road

Objective:

1) make a pipeline that finds lane lines on the road using road images obtained from the camera mounted on the car
2) Reflection on the work performed during this project.

Pipeline:

The pipeline describes the process of manipulating images obtained from camera mounted on car. The manipulation is done using some standard libraries in python like matplotlib, numpy and opencv().

1) **Importing** : The initial part of the project included importing images into the code matplolib.image function in python. *Image 1* shown below shows the original image

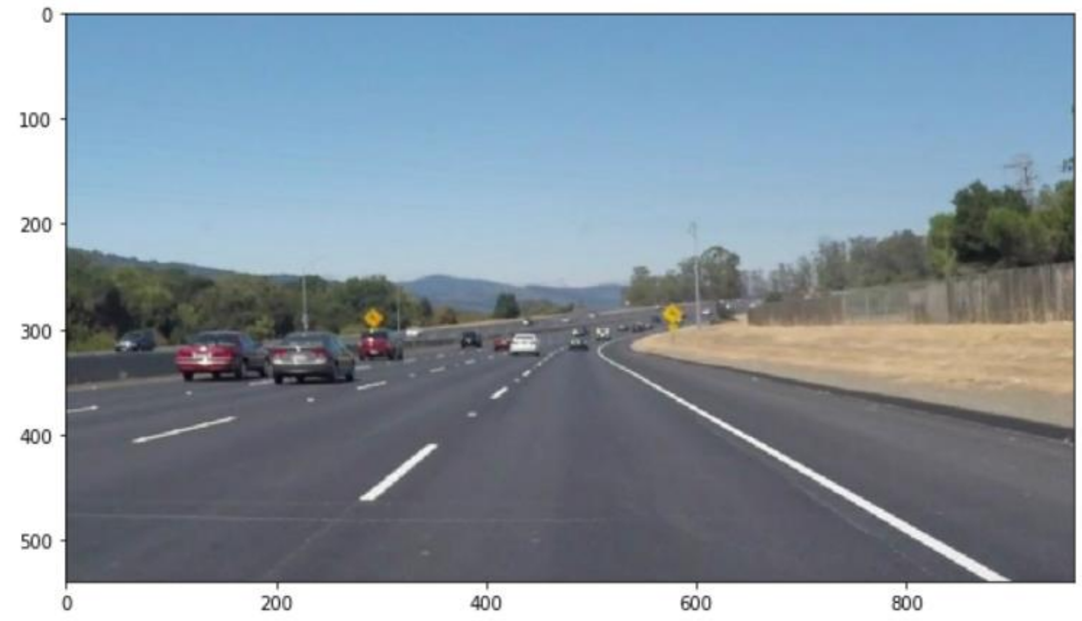

Image 1 : Original Image

2) **Grayscale conversion** : The imported image is then converted to grayscale image to detect edges of the object in the image. This is done using a predefined grayscale function which uses cv2.cvtcolor() function from opencv() library. The *Image 2* shown below shows the grayscale converted image of *Image 1*.

3) **Image Smoothing** : The grayscale image is then passed through gaussian_blur() function to smoothen out any noise or abrupt changes in the image. This step is also known as averaging or smoothing of the image. However it decreases efficiency of the edge detection algorithm.

   The image 3 shown below shows image after smoothing (gaussian blur) of Image 2 with a smoothing factor (kernel size) of 7. However, the differences may not be perceivable in the original image but edge detection is affected as shown in Image 4 and Image 5 which are obtained from image 2 and 3 after final hough lines detection (explained in point 8). The lanes detected in far left part of image 5 are not clear due to high kernel size used for gaussian blur.

Image 2 : Grayscale Image

Image 3 : Grayscale image with smoothing factor = 7

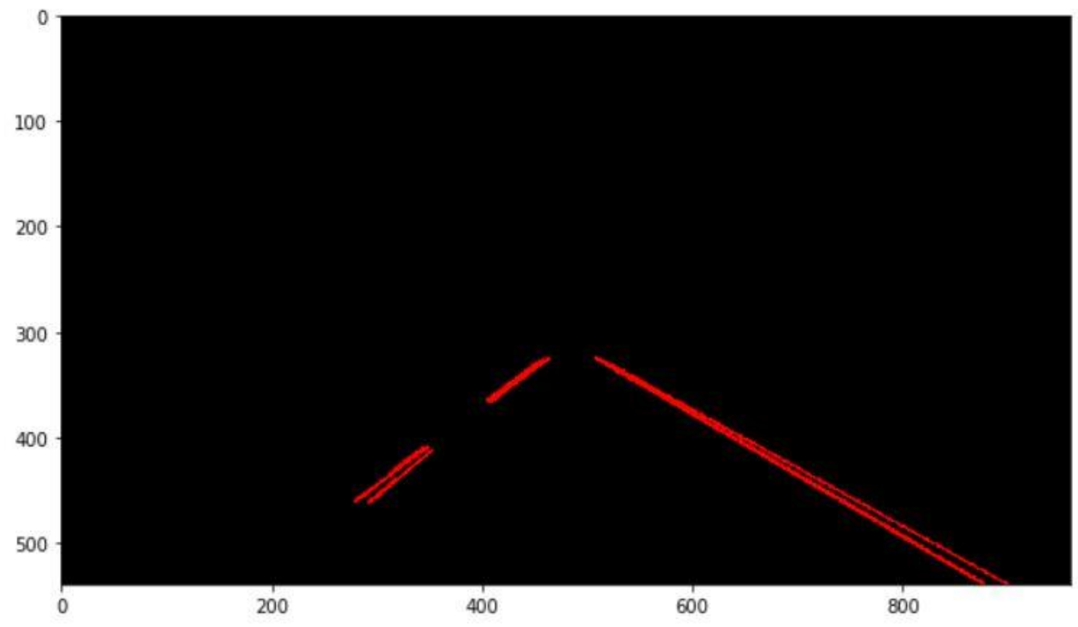

Image 4 : Lanes from image without smoothing

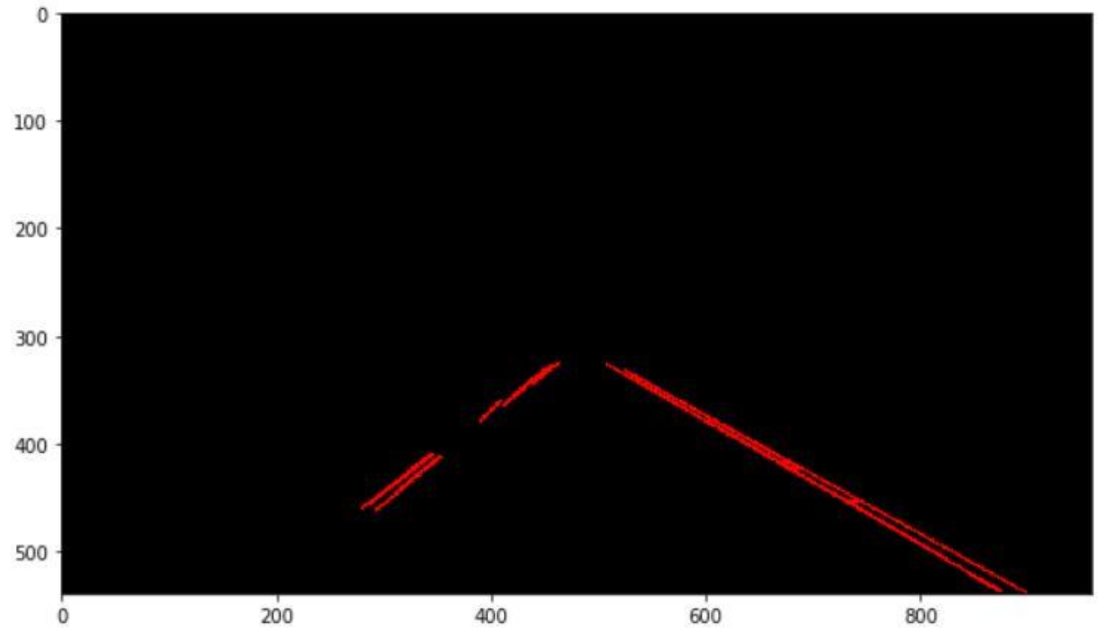

Image 5 : lanes from image with smoothing factor=7

4) **Object edge detection** : The smoothened image is then passed through canny edge detection algorithm which as the name suggests is used to find boundaries of the objects in the image using function canny. The arguments required for canny algorithm are low threshold and high threshold which define the pixels intensity below which no changes are detected and above which all changes are recorded as lines, respectively. The lines detected within the intensity

range are recorded based on their connectivity to lines detected above the threshold. The intensity values used for the code were

a. Low_threshold=50
b. High_threshold=150

The image 6 shown below is result obtained after passing the smoothened image with kernel size 5 into canny edge detection algorithm.

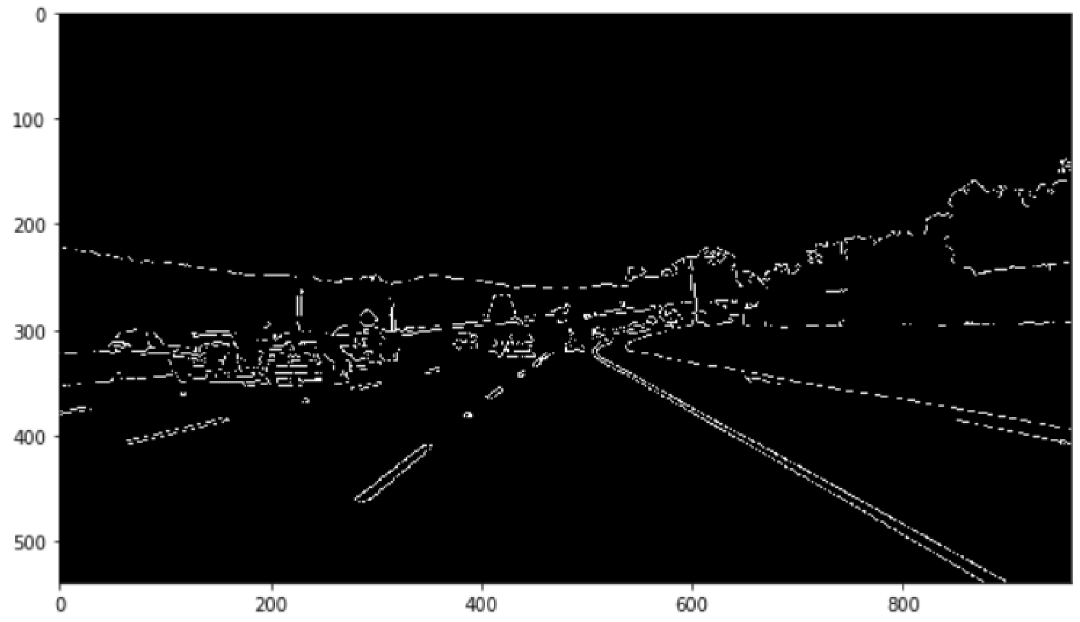

Image 6 : Image after Canny edge detection

5) **Region of interest** : A region of interest is defined to let algorithm know to pick lines from within a defined region. This is to make the lane detection easier and more robust. The region of interest function is used to manipulate the original image supplied with all areas outside of the region of interest masked (intensity level set to zero). Image 7 shown below is the original image 1 with region of interest cropped. The vertices used for this project are shown below

    Vertices= [100, 540], [460, 325], [510, 325],[960,540]

6) **Mask matrix** : The result from the region of interest function is used to create a binary matrix with 1's within the region of interest and 0's everywhere else.

7) **Filtering edge Image** : This mask image (matrix) is then multiplied with image (matrix) obtained from canny edge detection algorithm to remove lines which are outside region of interest. Image 8 shown above is obtained after applying region of interest on canny edge image as shown in Image 6

8) **Straight line detection** : The filtered image is then passed through Hough transform to find straight lines in the image. The lines are detected based on specific criterion (which is defined using input arguments to function hough). The lines are then drawn on the image using draw_lines() function. This was also modified to draw_lines_extend() to extrapolate edges to straight solid lines (which has been explained later). The Image 8 shown is obtained after this step is applied on Image 6.

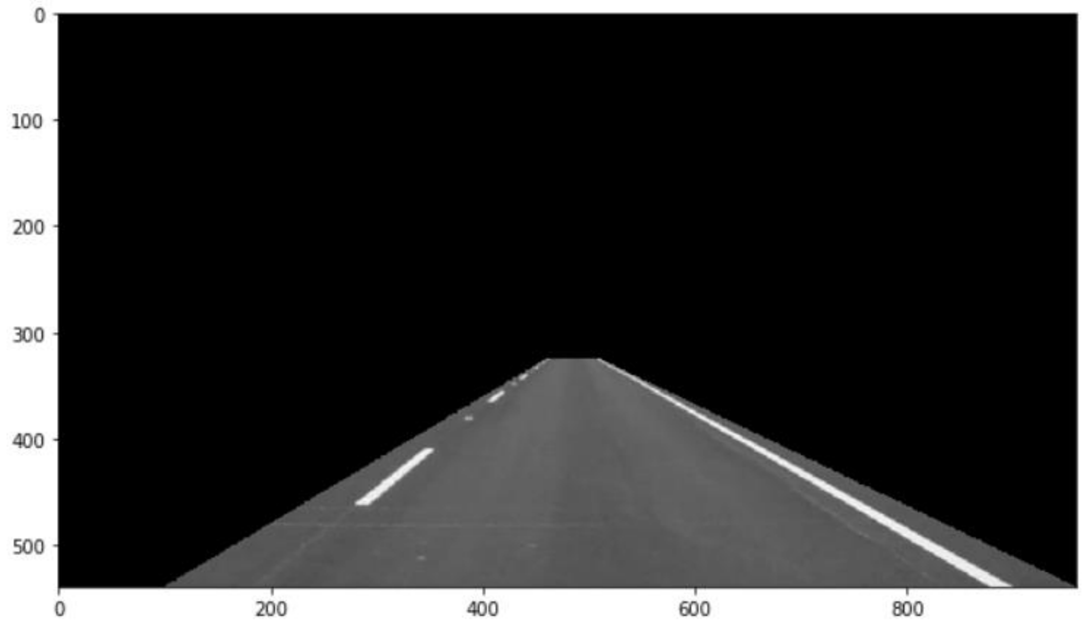

Image 7 : Image with only region of interest

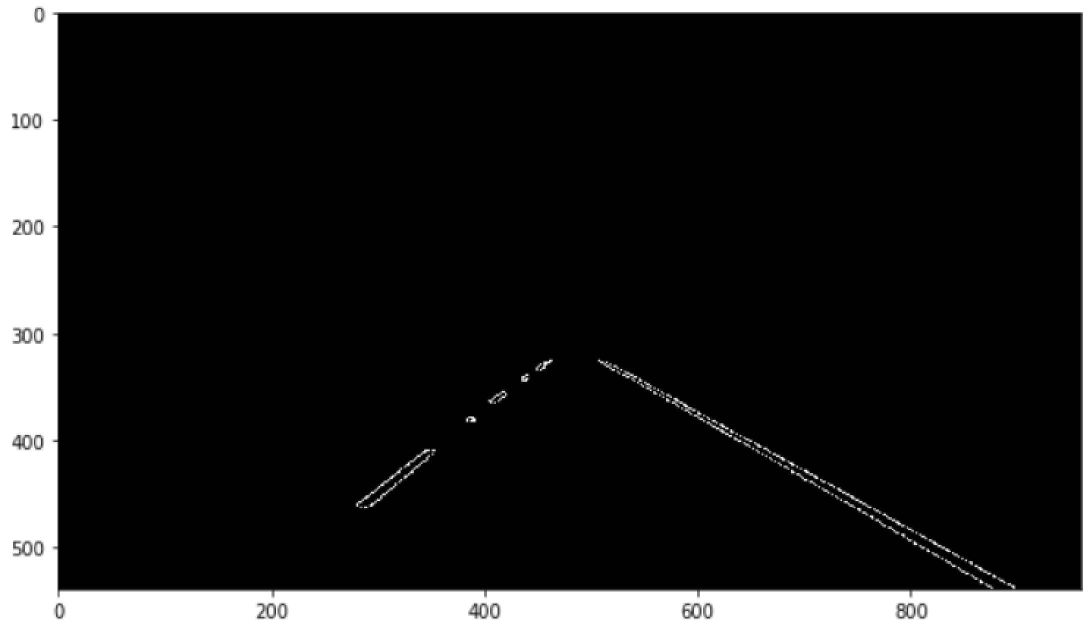

Image 8 : Canny image with region of interest highlighted

9) **Lane superposition** : The image obtained from hough_lines only contain detected lanes which are then superimposed on the original color image to create a image with driving lanes detected. The image 9 shown below is final image obtained after superposition of original image and with image 5.

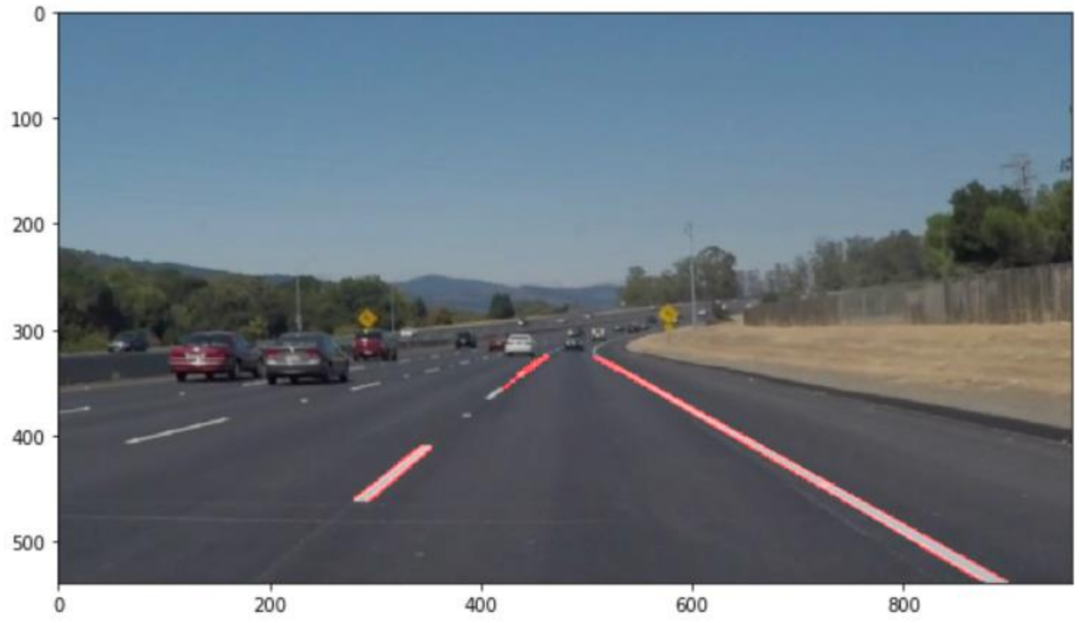

Image 9 : Final Image with lanes detected

Modified draw_lines function:

The original draw_lines() function is called within hough function which passes on lines detected to draw_lines() along with original image. The lines contain start and end x , y coordinates of line segments detected in step (as described above) which are drawn over empty image.

However, in order to extend the lanes from start to end require determining the slope and location of the solid lane within region of interest. Initially left and right lanes are separated using the center of the image as a separator. The lines to the left (smaller than (Max x_coordinate)/2) are labelled as left while other are labelled as right lanes.

Next problem was detecting parallel lines (or nearly parallel lines) as the solid line should pass through center of the edge lines detected earlier. Since we don't have any idea about the range within which slope of the line should lie. Thus, in order to have a reliable estimate of the line an extra parameter length of the line is calculated along with slope of the line. Hence, each of the left and right lane are 2d arrays which have x, y coordinates of the end of lines along with their slopes and length of the line in every row.

The array is then sorted along rows using distance as a parameter in descending order. Thus, longest line segment is at the top which is most reliable. Hence, is used to identify a 1st line using np.polyfit command (slope m and y intercept – y=mx+c form). Now to identify a parallel line, line equation is represented in y-(mx+c) form and other points in the same array are substituted in the equation. If both points are on the line they y-(mx+c) results to zero (or a very small value usually less than 2). If both points are results to a value greater than threshold (in this case I found around 5) then lines are assumed to be parallel and second line is found out. Averaging out the parameters from the two lines obtained are used to find out the average line.

The average line is then extended by using maximum y coordinate (bottom line) and the y coordinate of the top of the quadrilateral (far edge of the region of interest). Hence, lanes are drawn with the given end points. The result from one of the images are shown below (Image 10)

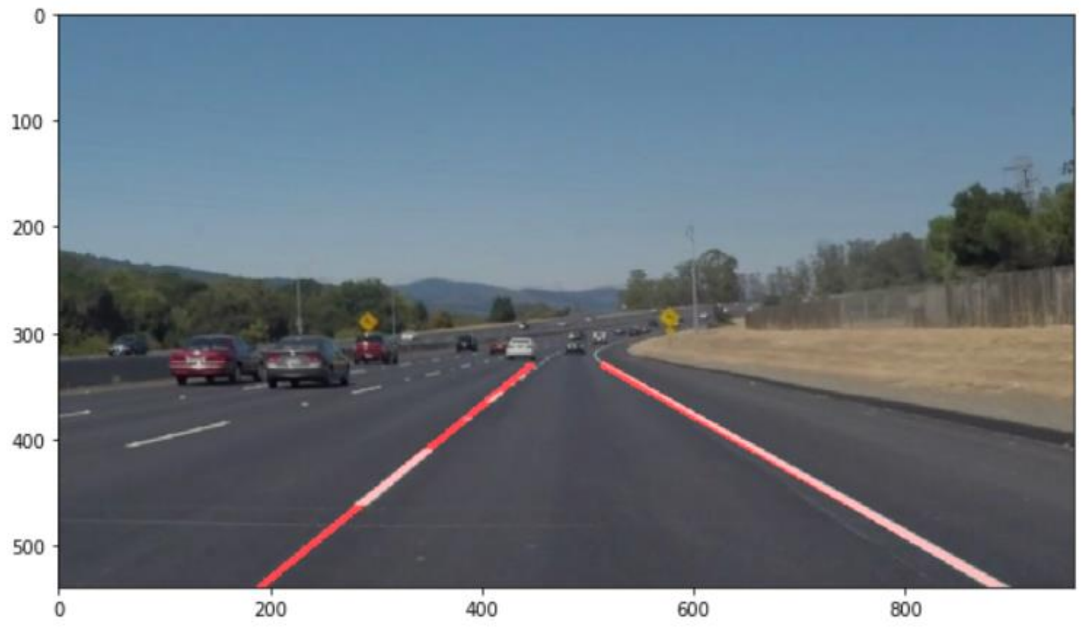

Image 10 : Images with extrapolated lanes

In order to avoid confusion, the modified draw_lines() is created as a new function draw_lines_extend(). And to call draw_lines_extend() function, an argument has been added at the end to the hough() function. Modified function is called by adding another argument with value equal to 1 at the end of the hough() function.

Potential shortcomings :

1) The pipeline is not able to detect lanes for some of the frames in the video. It can be difficult if these patches occur in succession.
2) The pipeline is also not able to predict a stable slope for the line solid line in the video which can give random inputs to the steering. However, this was solved to a large extent using high values for threshold(~50), min line length(>100) and max line gap(>100).
3) The pipeline is also not able to detect lanes under different lighting conditions as seen in the challenge video (after 4s).
4) It is also not able to detect turns (or detect curved lanes) which can make it hard to steer for the car.
5) The pipeline is also not able to differentiate between boundaries of other object in another lane and the lane boundaries which might led us to detect false lanes.

Possible improvements :

1) The pipeline can be improved to predict lanes in some frames where lanes are not detected by algorithm. This can be done by using lane parameters from the certain previous frames to predict a probable path in case of no detection.
2) The pipeline can be improved to predict turns by doing a curved line fit over a number of line segments (or points) and also region of interest can be shortened for a turn based on observing a observing a turn in the lanes near the far (top) edges of the region of interest. This can be done by keeping some portion of the top part of the region of interest as dynamic which can be accepted or rejected based on certain conditions
3) Pipeline lane detection can be improved under different lighting conditions by performing canny edges filtering in color scale rather than grayscale image
4) Pipeline detection can also be improved by defining tolerance regions near the detected lanes. Initially region of interest can be to identify two lanes and then shifting to two regions within which two lanes (left and right) must lie.