Project Report
Lane Lines Detection
Self Driving Car Nano degree
Udacity

Ayush Jain
Submission Date: 19/05/2020

Introduction

Lane line detection is important to control a self driving vehicle. Since the lane lines can be of different colour's (white, yellow) or forms (solid, dashed) this seemingly trivial task becomes increasingly difficult. Moreover, the situation is further exacerbated with variations in lighting conditions. Thankfully, there are a number of mathematical tools and approaches available nowadays to effectively extract lane lines from an image or dashcam video. In this project, a program is written in python to identify lane lines on the road, first in an image, and later in a video stream. After a thorough discussion of the methodology, potential shortcomings and future improvements are suggested.

Methodology

Lane Finding Pipeline

Before attempting to detect lane lines in a video, a software pipeline is developed for lane detection in a series of images. Only after ensuring that it works satisfactorily for test images, the pipeline is employed for lane detection in a video. The pipeline consisted of 5 major steps excluding reading and writing the image. Consider the test image given below:

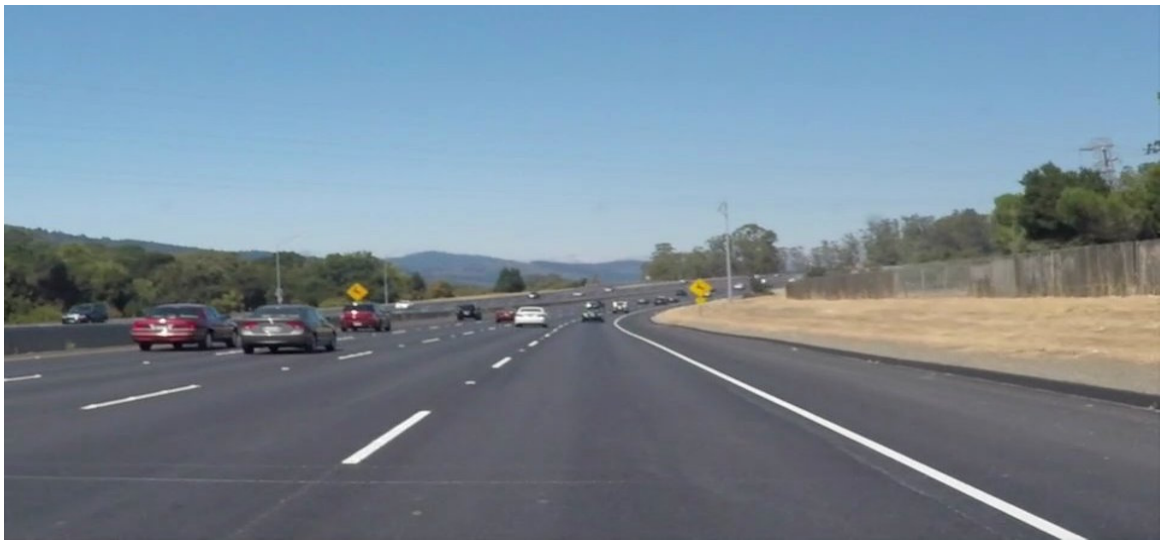

Figure 1: Original Image

The test image is first converted to grayscale from RGB using the helper function **grayscale()**. The gaussian blurred function **gaussian_blur(gray, kernel_size)** took a **kernel_size** of 5, to remove noise or spurious gradients. The blurred image is given below.

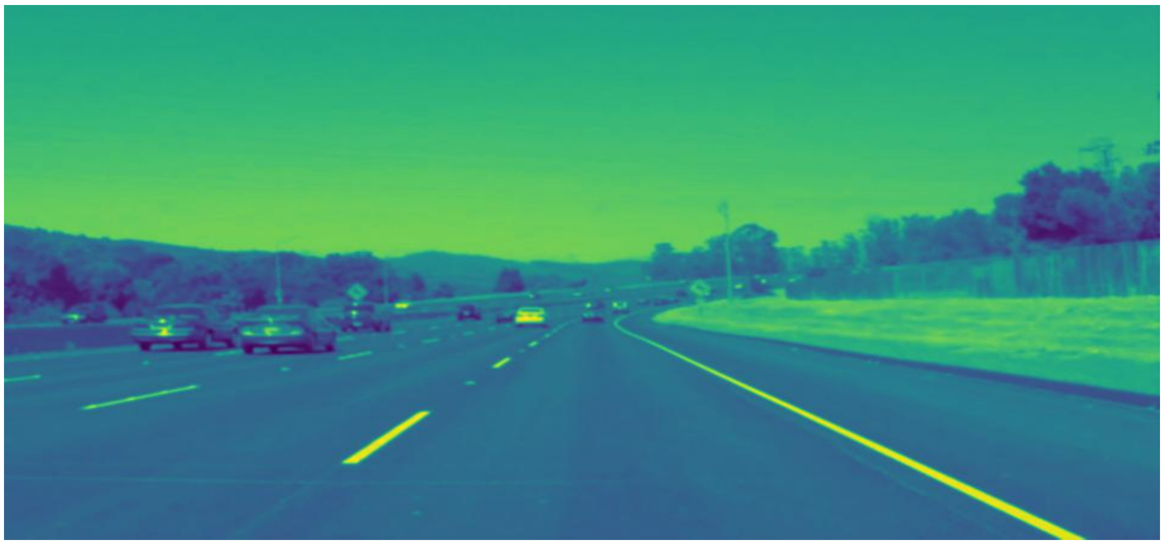

Figure 2: Gaussian blurred Image

Then, canny edge detection is applied, using **canny(gray_blur, low_threshold, high_threshold)** function on this blurred image and a binary image shown below is produced.

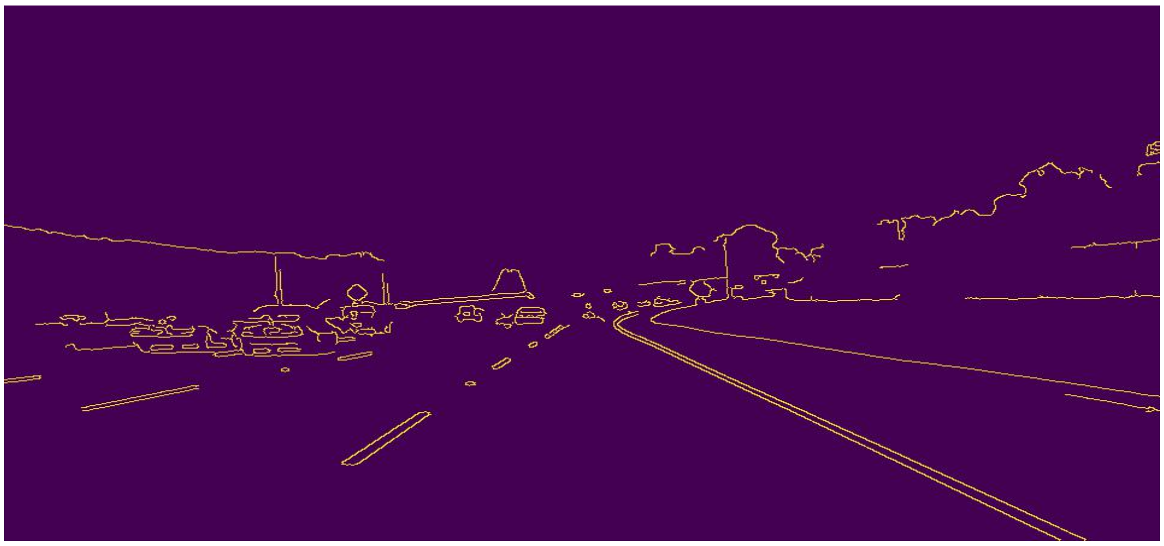

Figure 3: Canny edge detected Image

This image contains edges that are not relevant for lane finding problem. A region of interest is defined to separate the lanes from surrounding environment and a masked image containing only the lanes is extracted using **cv2.fillPoly()** and **cv2.bitwise_and()** function from opencv library. This can be seen below.

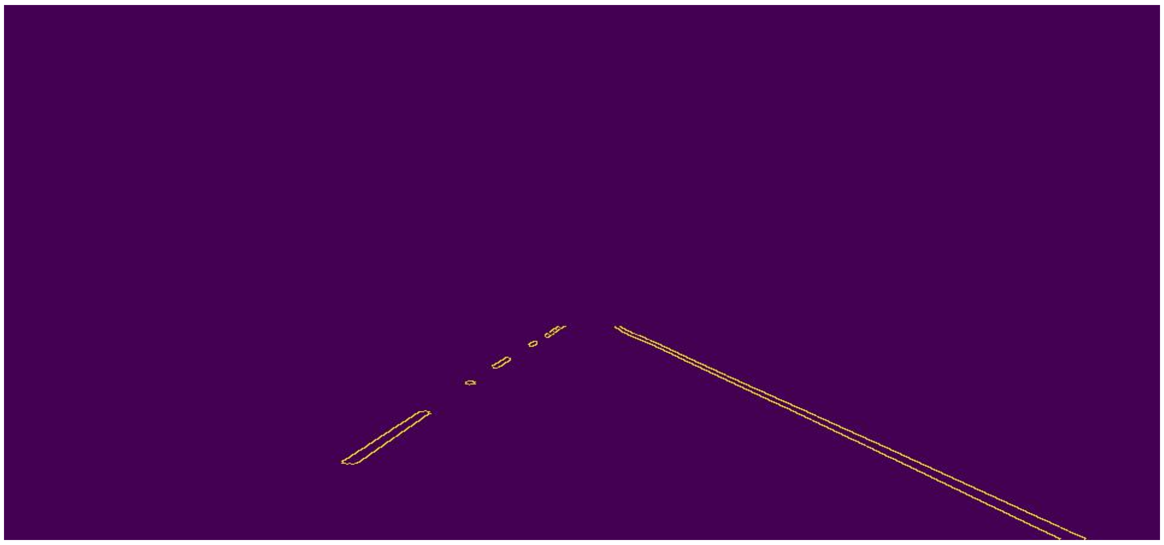

Figure 4: Masked Image

This binary image of identified lane lines is then used to find lines with minimum line length and maximum line gap using **hough_lines()** as shown below:

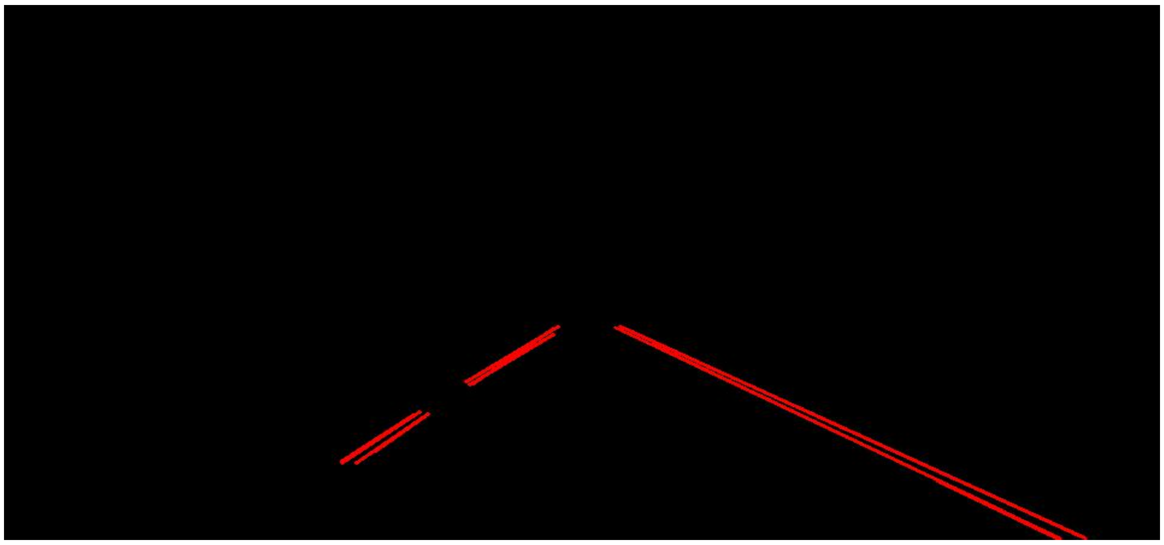

Figure 5: Image with lines detected through hough transform before modifying draw_lines()

The **draw_lines()** function is then modified to extend the detected lines. The lines are now continuous as required. The result is as shown below:

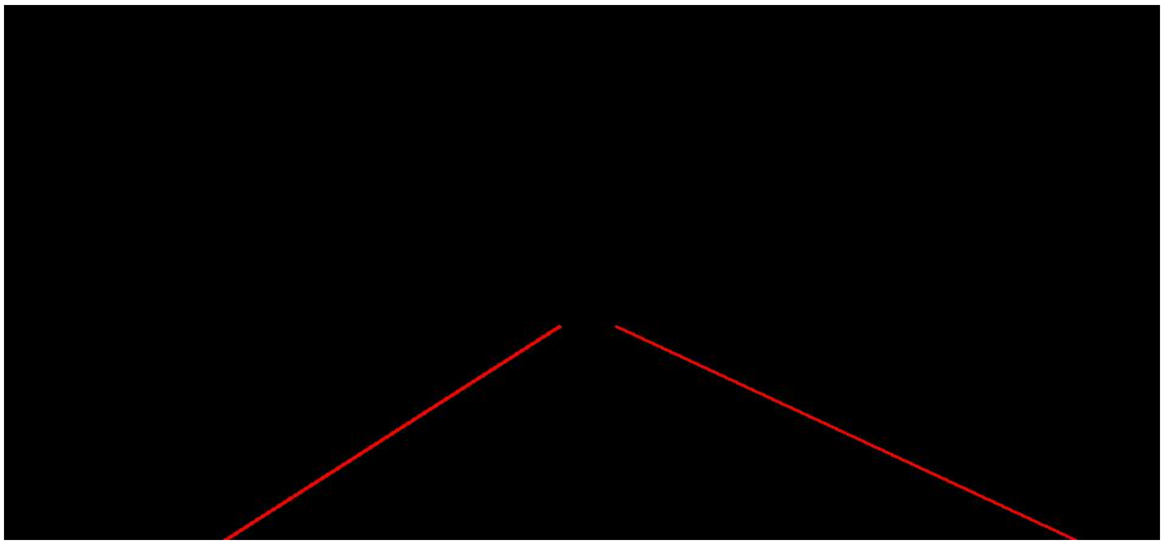

Figure 6: Image with lines detected through hough transform after modifying draw_lines()

Finally the images with hough lines is merged with the original image using **cv2.addweighted()** function from opencv library.

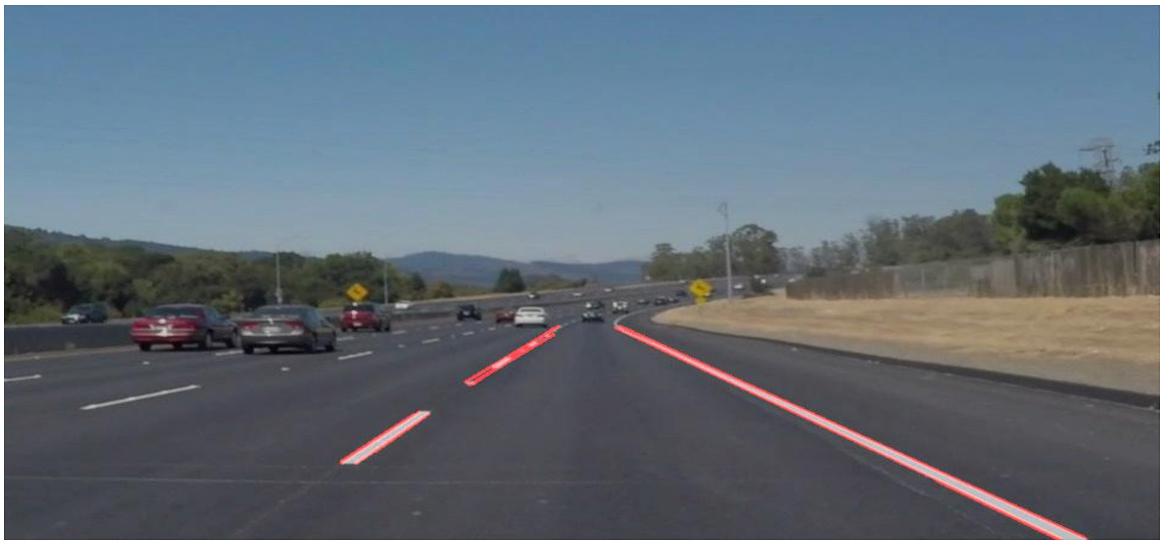

Figure 7:

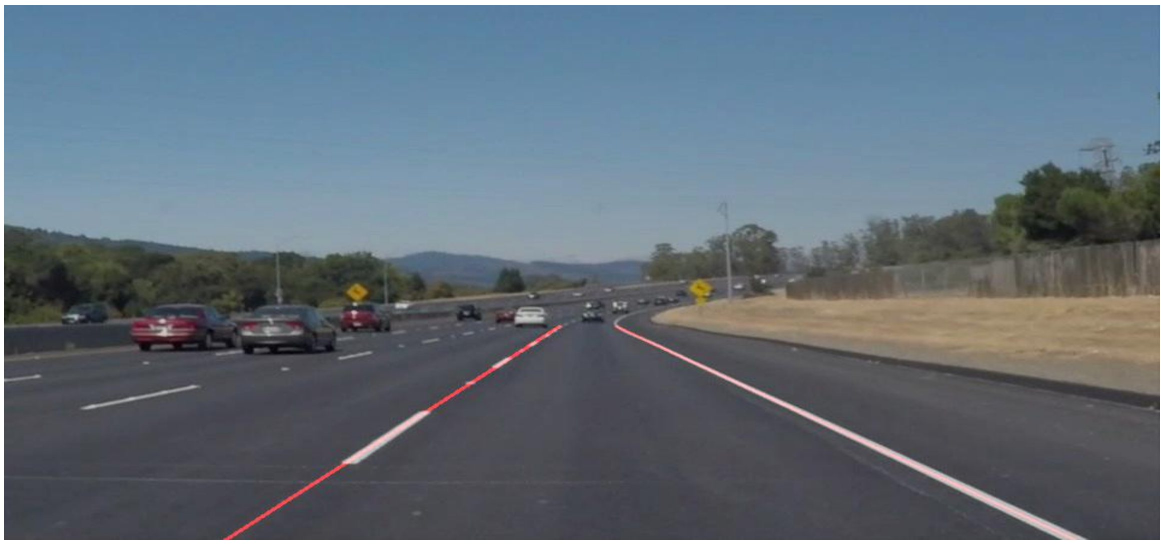

Figure 8:

Implementing the pipeline on test videos

The pipeline developed in the project is implemented on 2 different test videos. It was observed that the pipeline produces acceptable results for both the test videos, solid yellow left - dashed white right lane and solid white right - dashed white left lane. The youtube video can be viewed here, **Link**.

Reflection

Modification to drawlines() helper function

Since the resulting line segments after the processing the image through the pipeline were not continuous, a modification was made to the drawlines() helper function. Consider the code snippet below:

```


'''
def draw_lines_robust(img, lines, color=[200, 0, 0], thickness = 10):

    x_left = []
    y_left = []
    x_right = []
    y_right = []
    imshape = image.shape
    ysize = imshape[0]
    ytop = int(0.6*ysize) # need y coordinates of the top and bottom of left
   and right lane
    ybtm = int(ysize) #  to calculate x values once a line is found

    for line in lines:
        for x1,y1,x2,y2 in line:
            slope = float(((y2-y1)/(x2-x1)))
            if (slope > 0.5): # if the line slope is greater than tan(26.52 deg
   ), it is the left line
                    x_left.append(x1)
```

~~~
                    x_left.append(x2)
                    y_left.append(y1)
                    y_left.append(y2)
            if (slope < -0.5): # if the line slope is less than tan(153.48 deg), it is the right line
                    x_right.append(x1)
                    x_right.append(x2)
                    y_right.append(y1)
                    y_right.append(y2)
    # only execute if there are points found that meet criteria, this eliminates borderline cases i.e. rogue frames
    if (x_left!=[]) & (x_right!=[]) & (y_left!=[]) & (y_right!=[]):
        left_line_coeffs = np.polyfit(x_left, y_left, 1)
        left_xtop = int((ytop - left_line_coeffs[1])/left_line_coeffs[0])
        left_xbtm = int((ybtm - left_line_coeffs[1])/left_line_coeffs[0])
        right_line_coeffs = np.polyfit(x_right, y_right, 1)
        right_xtop = int((ytop - right_line_coeffs[1])/right_line_coeffs[0])
        right_xbtm = int((ybtm - right_line_coeffs[1])/right_line_coeffs[0])
        cv2.line(img, (left_xtop, ytop), (left_xbtm, ybtm), color, thickness)
        cv2.line(img, (right_xtop, ytop), (right_xbtm, ybtm), color, thickness)
```
~~~

Observe that a classification of lines identified through houghlines criteria is made based on their slope. Evidently, lines with positive slope are classified as being on the left lane and lines with negative slope are classified as being on the right lane. Flat lines having slope below absolute value of 0.5 are discarded. After storing points for respective left and right lanes, a linear curve fit (degree 1) using **polyfit()** function from numpy library is done to obtain the slope and intercept of left and right lanes. Following this, x coordinates are found for respective **ytop** and **ybtm** coordinates (user defined) using the lane equations for both lanes. This gives us starting and ending coordinates for both left and right lane. Finally, lines are drawn using **cv2.line()** function to connect these points and the image is merged with the original image as before to produce the below result.

Shortcomings observed in the current pipeline

1. Since the first step is converting the image to grayscale from RGB, shadows and light variations in the environment are difficult to capture. This can be gleaned from the fact that the current pipeline while working reasonably well for the first two test videos breaks down for the challenge video.

2. The lane lines detected in the resulting output are not as stable as the ones in "P1_example" video. This is not desirable since it is difficult to follow rapidly changing steering commands.

Possible improvements to the current pipeline

1. Instead of converting the image to grayscale directly, the test image can be preprocessed using RGB normalization. This can help in mitigating effect of shadows and lighting variations and make the current pipeline more robust. A function definition for producing a normalized RGB image is shown below. This approach did not produce an improvement when implemented in it's current form and can be further improved.

~~~
    ```
    # Reference: http://akash0x53.github.io/blog/2013/04/29/RGB-Normalization/
    def normalized_rgb(img):
~~~

```
        imshape = img.shape
        ysize = imshape[0]
        xsize = imshape[1]
        norm=np.zeros((ysize,xsize,3),np.float32)
        norm_rgb=np.zeros((ysize,xsize,3),np.uint8)
        b=img[:,:,0]
        g=img[:,:,1]
        r=img[:,:,2]
        sum = b + g + r
        norm[:,:,0]=b/sum*255.0
        norm[:,:,1]=g/sum*255.0
        norm[:,:,2]=r/sum*255.0
        norm_rgb=cv2.convertScaleAbs(norm)
        return norm_rgb
‘‘‘

```

2. Another way of dealing with shadows and variations in light is to use a different colorspace, for example, say HSV instead of RGB. A function definition for applying RGB to HSV transform is shown below. The test image before being fed to the pipeline is given an RGB to HSV transform. This approach while producing acceptable results for the first two test videos did not result in any significant improvements in the performance of identifying lane lines for challenge video.

```
‘‘‘
def rgbtohsv(img):
    "Applies rgb to hsv transform"
    return cv2.cvtColor(img, cv2.COLOR_RGB2HSV)
‘‘‘

```

3. Another improvement in the pipeline can be to include a running average of slopes for identified lane lines so that there is a smooth transition from one frame to the next. This avoids rapid changes in commands to the steering control system.

Further, machine learning approaches can be explored to make the lane finding pipeline more robust in future.