Project Report

Udacity

Self-Driving Car Nanodegree

Project 1: Finding Lane

Author:
Thomas Teh

Date: January 13, 2019

1 Introduction

This project is part of the Self-Driving Car Nanodegree by Udacity and its objective is to produce a pipeline that finds lane lines on images and videos of a car driving in lanes. The project uses OpenCV on Python.

2 Pipeline Description

The original Jupyter Notebook included some helper functions which uses the OpenCV functions to help draw lines on images, convert images to grayscale, run the Canny edge detector and etc. The major part of this project is to assemble all those helper functions into a pipeline that takes in still images as input and produces images annotated with the lane lines as output. Then, we apply this pipeline on videos and modify the draw line function to provide smooth, continuous, and extrapolated lane lines.

My pipeline consists of the following steps, as shown in Figure 1:

1. **Converting the original image to grayscale**: the original image is converted into grayscale as it is more suitable for edge detection. The Canny edge detector will look at how the gradient of the image changes to determine whether a pixel belongs to an edge.

2. **Gaussian blurring / smoothing**: the Gaussian filter will smooth the image and retain the most distinguishable parts. The size of the kernel will determine the size of the local region on the images that will be smoothed. In my implementation, I set the kernel size to be 5.

3. **Canny edge detection**: the Canny algorithm detects edges by look at how the pixel intensities of the image changes. This translates to computing the gradient of the pixel values. Pixels who have gradient values below the minimum threshold will be discarded and pixels who have gradients greater than the maximum threshold will be considered to be an edge.

   For pixels who lie between these two thresholds are classified edges or non-edges based on their connectivity. If they are connected to ”sure-edge” pixels, they are considered to be edges. Otherwise, they are also discarded. In my implementation, I found the minimum and maximum thresholds of 10 and 150 respectively to work rather well.

4. **Masking irrelevant regions**: we mask the irrelevant areas by constructing a region of interest, which is a trapezoidal region on the lower part of the image. In short, we are cropping the image and retaining the following area:

$$\begin{aligned}\text{bottom left} &= (0, \text{image height})\\ \text{bottom right} &= (\text{image width}, \text{image height})\\ \text{top left} &= (440, 320)\end{aligned}$$

$$\text{top right} = (540, 320)$$

5. **Hough lines detection**: probabilistic Hough line detection is being used (cv2.HoughLineP) with the following parameters:

$$\begin{aligned} \rho &= 1 \\ \theta &= \frac{\pi}{180} \\ \text{threshold} &= 70 \\ \text{min\_line\_len} &= 5 \\ \text{max\_line\_gap} &= 150 \end{aligned}$$

   The Hough line transform is a voting system. Both the rho and $theta$ define the granularity of our image space in terms of polar coordinates. Then we are counting the elements in each grid. For a line to be detected, votes must be above the specified threshold, which is the minimum number of intersections in a given grid cell that are required to choose a line.

   min_line_len is the minimum length of a line (in pixels) that we will accept in the output and max_line_gap: is the maximum distance(in pixels) between segments that will be allowed to connect into a single line. Decreasing min_line_len will lower the minimum of pixels to make a line while increasing max_line_gap will allow points that are farther away from each other to be connected.

   The output of this function is a blank image with the relevant lines tracing the lanes.

6. **Left and Right line detection (draw lines)**: the output from the Hough line transform may not form nice continuous lines to depict the lanes. Instead, we will need to determine the best fit of lines for the left and right lanes respectively. The following steps are taken in order to construct the lanes (as well as to solve the challenge problem):

   - We compute the slope, m of each line by $m = \frac{y_2 - y_1}{x_2 - x_1}$. If slope, $m > 0$, then the line belongs to the left lane, otherwise, the line belongs to the right lane.
   - For each lane, we only retain lines that are greater than a certain threshold. In my implementation, this threshold is set at 0.50. This is to filter out any horizontal lines that may have been detected. In the challenge video, these horizontal lines are detected when the road surface changes.
   - For all the points belong to the left lane, we fit a straight line and we used the estimate slope and intercept to compute the lines at different locations of the image. For example, to extrapolate, we can compute the point where the line would be at the edge of the image, which will be

given by:

$$\left(\frac{\text{image height} - \text{intercept}}{\text{slope}}, \text{image height}\right)$$

7. **Produce weighted image**: Once we have the smooth and extrapolated line, we overlay them onto the original image.

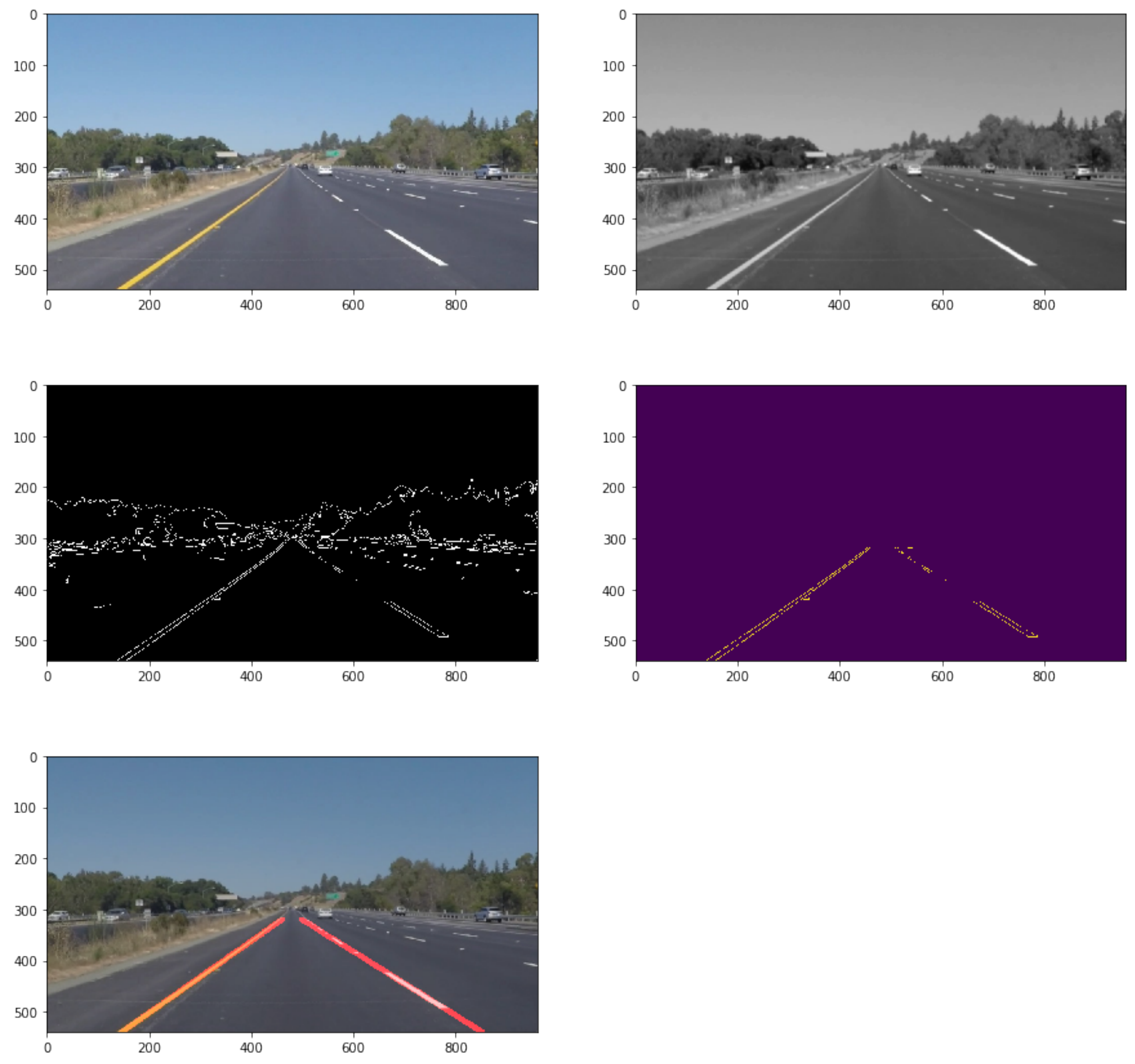

Figure 1: First row: original image (left); image after converted to gray scale and applying Gaussian blurring (right). Second row: Image after applying Canny edge detection (left); image after masking the irrelevant regions (right). Third row: applying Hough transform and overlaying on original image.

3 Potential Shortcomings

Here are some of the major shortcomings of the pipeline:

1. It is strictly dealing with straight lines. There is no estimation of curvature. Hence it will not be able to find the lane if the car is moving along a steep corner or turn. Hence, its use for all circumstances will be limited.

2. The region of interest is fixed. However, realistically, the region of interest ought to change whether the car is moving up or down or turning. Hence, we may need to make the region of interest dynamic.

3. The road conditions in the test images / videos are mostly the same, i.e. during the day on a highway. This limits the usefulness of the pipeline under different weather conditions, time of day, and also surrounding terrain. For example, in the city, the edges of the building may be misconstrued to be part of the driving lane.

4. The current data we have tested the algorithm with is basically from a car that is driving in lane. It is vital to be able to estimate the lane the car is driving on when it is moving across two different lanes.

4 Ideas for Improvements

In order to improve the lane finding algorithm, we can implement the following:

- Take into account of the lane curvature (which will be dealt with by Project 2 Advanced Lane Finding)

- For videos, it is possible to make use of the previous frame and estimate the lanes, instead of detecting the lane from scratch. A Kalman filter should be able to make use of that and produce a much more stable lines for the lanes.

- Need to adapt the region of interest such that it is dynamic and make full use of the information that is being captured by the camera.

- Need to investigate real time curve detection. Currently, we are working on images and videos that are offline. However, for autonomous vehicles, the algorithm needs to be optimized such that it can recognise lanes quickly and robustly.