Self-Driving Car Engineer NanoDegree
Project Report

Nyaribo Maseru
April 3rd, 2020

Finding Lanes On The Road

I. **Definition**

Project Overview

This project is a partial fulfilment for Udacity's Self-Driving Car Engineer NanoDegree program. In this project, I shall use the tools I learned about identifying lane lines on the road. I shall then develop a pipeline on a series of individual images, and later apply the result to a video stream.

The mission and the steps of this project are the following:

1. Make a pipeline that finds lanes on the road.
2. Reflect on my work in a written report.

II. Reflection

Describe the current pipeline process

My pipeline consisted of the 5 steps below:

1. Color selection
2. Gaussian smoothing
3. Canny Edge Detection
4. Region of interest selection
5. Hough Transform line detection

Color selection

As it happens, lane lines are not always the same color, and even lines of the same color under different lighting conditions (day, night) may fail to be detected by our simple color selection. What we need is to take our image and convert it to grayscale, in order to detect lines of color identified as edges. Below is an image of the original image and the grayscale image with dimensions.

```
This image is: <class 'numpy.ndarray'> with dimensions: (540, 960, 3)
This grayscale is: <class 'numpy.ndarray'> with dimensions: (540, 960)
```

Gaussian smoothing

After converting the original image to grayscale, the next step is to apply Gaussian smoothing, which essentially suppress noise and spurious gradients by averaging.

Canny edge detection

After applying the gaussian smoothing I then used Canny algorithm from open CV. What the algorithm does, is first detect strong edge (strong gradient) pixels above the high_threshold, and reject pixels below the low_threshold.

Next, pixels with values between the low_threshold and high_threshold will be included as long as they are connected to strong edges. The output edges is a binary image with white pixels tracing out the detected edges and black everywhere else

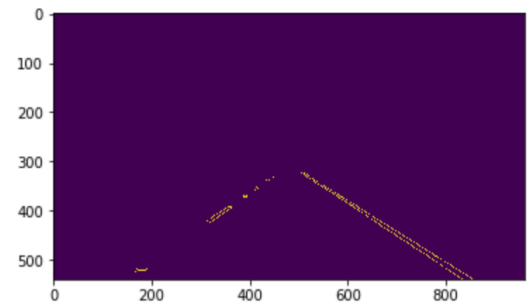
Region of Interest Selection

At this point, however, it would still be tricky to extract the exact lines automatically, because we still have some other objects detected around the periphery that aren't lane lines.

Next we'll create a masked edges image using cv2.fillPoly() with a four sided polygon to mask image.

1.3 Region of interest selection

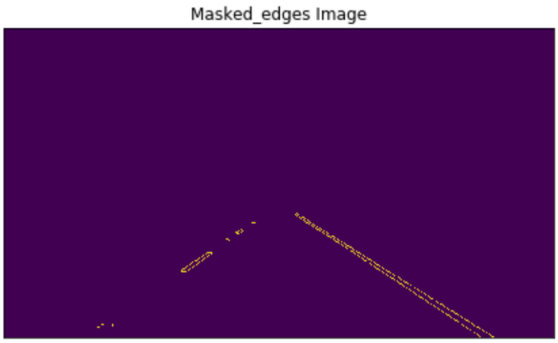
```
# Next we'll create a masked edges image using cv2.fillPoly()
# This time we are defining a four sided polygon to mask

imshape = img.shape
vertices = np.array([[(0,imshape[0]),(465, 324), (530, 324), (imshape[1],imshape[0])]], dtype=np.int32)
masked_edges = region_of_interest(edges,vertices)

# plt.figure(figsize=(12,8))
plt.imshow(masked_edges);
```

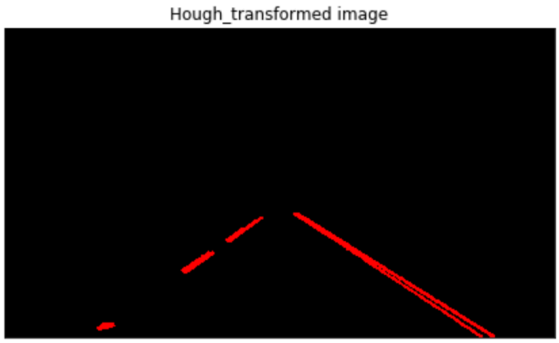


Hough transform line detection

To accomplish the task of finding lane lines, we need to specify some parameters to say what kind of lines we want to detect (i.e., long lines, short lines, bendy lines, dashed lines, etc.). To do this, we'll be using an OpenCV function called HoughLinesP that takes several parameters.

I shall crate a function that takes thresholds parameters for HoughLinesP to output a line image . See below image





In order to draw a single line on the left and right lanes, I modified the draw_lines() function by the writing the following code below;

1.5 Creating a Single Left and Right Lane Line

```
def getLeft_and_rightLane(img, lines):

    # Copy image
    image = np.copy(img)

    # get left and right lines
    left_line_x = []
    left_line_y = []
    right_line_x = []
    right_line_y = []


    # Calculating the slope.
    for line in lines:
        for x1,y1,x2,y2 in line:
            slope = ((y2-y1)/(x2-x1))

            if math.fabs(slope) < 0.5:# <-- Only consider extreme slope
                continue
            if slope <= 0: # <-- If the slope is negative, left group.
                left_line_x.extend([x1, x2])
                left_line_y.extend([y1, y2])
            else: # <-- Otherwise, right group.
                right_line_x.extend([x1, x2])
                right_line_y.extend([y1, y2])


    min_y = int(image.shape[0] * (3 / 5))
    max_y = int(image.shape[0])
```

```
    poly_left = np.poly1d(np.polyfit(
        left_line_y,
        left_line_x,
        deg=1
    ))
    left_x_start = int(poly_left(max_y))
    left_x_end = int(poly_left(min_y))
    poly_right = np.poly1d(np.polyfit(
        right_line_y,
        right_line_x,
        deg=1
    ))
    right_x_start = int(poly_right(max_y))
    right_x_end = int(poly_right(min_y))

    new_line = [[[left_x_start, max_y, left_x_end, min_y],[right_x_start, max_y, right_x_end, min_y],]]

    return new_line
```

The final out of the pipeline

```
continous_lines = getLeft_and_rightLane(img, lines)

final_image = drawSingleft_and_rightLane(image,continous_lines,thickness=6,)



plt.figure(figsize=(12,8))
plt.imshow(final_image)
plt.show()
```

Potential shortcomings with the current pipeline

One potential shortcoming would be what would happen when defining a four sided polygon to mask a region of interest selection. Images have different dimensions which make it difficult to choose the right polygon figures for the images.

Another shortcoming could be Hough transform parameters for different images. Like earlier stated, images have different dimensions and a scaler as to be applied to smoothen the image.

Possible improvements to the current pipeline

A possible improvement is able to find a function that finds the region of interests for image based on its dimensions without using trial and error.

Another potential area of improvement is how to stable erratic lines on videos after running through the pipeline.

References

In this project i leveraged most of the materials from the Self-Driving Car Engineer NanoDegree program classroom and student community at Udacity to complete the project.

1. https://medium.com/@mrhwick/simple-lane-detection-with-opencv-bfeb6ae54ec0
