Finding Lane Lines on the Road

Finding Lane Lines on the Road

The goals / steps of this project are the following:

- Make a pipeline that finds lane lines on the road
- Reflect on your work in a written report

Reflection

1. Pipeline Overview

The lane-detection pipeline has 5 steps:

- Change the image to gray.
- Find the edges in the image through edge detection algorithms.
- Select interested area in the image.
- Find lines in the interested area.
- Perform interpolation on the found lines.

Now Let's go through each steps in details.

2. Details of Each Step in the Pipeline

Change image to gray

This part is short and sweet. By calling cv2.cvtColor() function, we can change a color image to gray.

Edge Finding

Edge-finding is done by Canny Algorithm (cv2.Canny() function). The way the algorithms works is calculating the gradient of the image and find the where the local maximum locate. This is because the maximum gradient appears to be perpendicular to edges.

There are two parameters needs to be tuned in Canny function: low_threshold and high_threshold.

However, gradient is susceptible to noise. Thus before applying Canny function, the image should be denoised using a Gaussian Blur function.

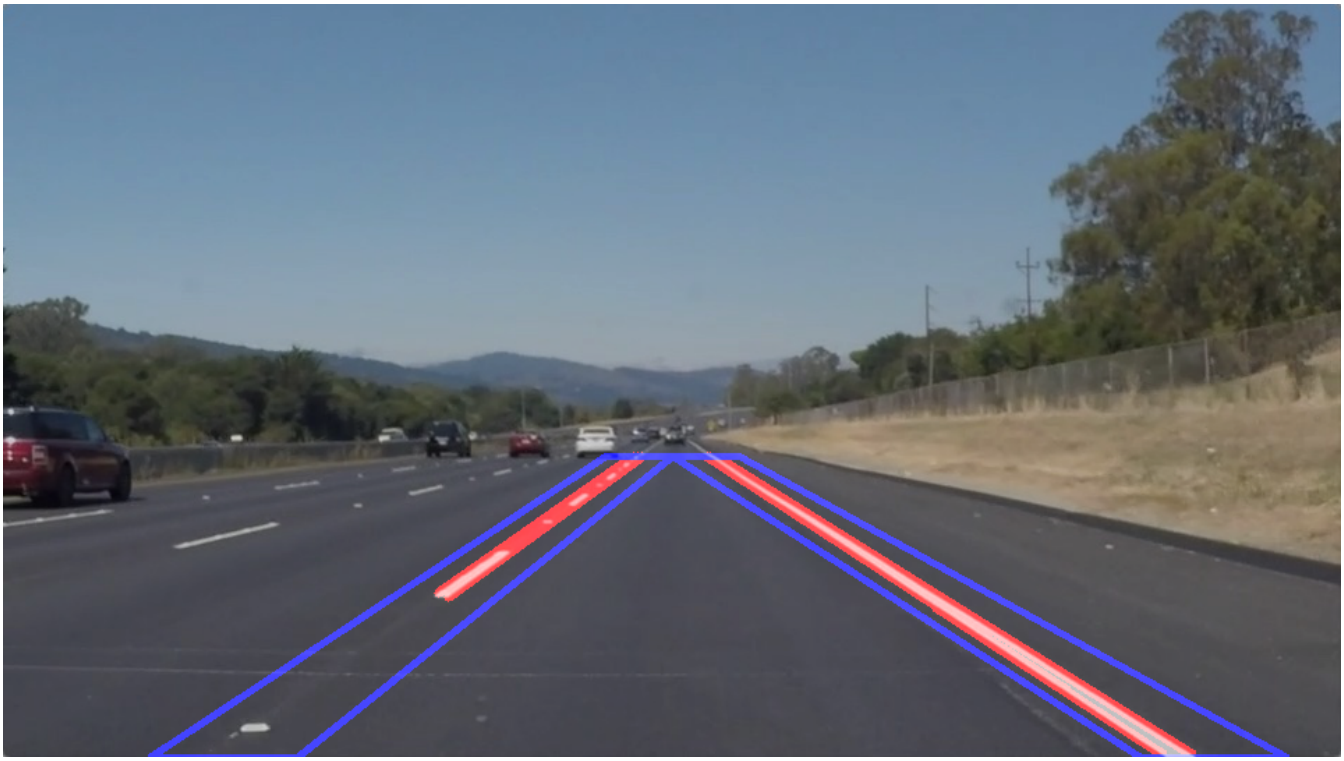
Interested Area Selection

After edge-finding, the next step is to find lines in the image. However, sometimes there are objects other than lane marks that appears to be lines in the image. We want to exclude these objects as much as possible before finding the lane boundaries in the image. In other words, the line-finding algorithm should be run on a selected area that does not contain too much unrelated objects.

In this pipeline, the shape of the area is determined by two trapezoids. And the position of the 8 vertices are selection such that it includes the lane marks and be as small as possible.

This step is pretty important. Instead of spending a lot of time to tune the hyper-parameters in edge-finding and line-finding algorithms, one can eliminate most of the hassles by finding the correct area. For a single frame, we want the area to be as small as possible. However, we do want to have some wiggle room for it to work well on a video.

One of the results is shown below:

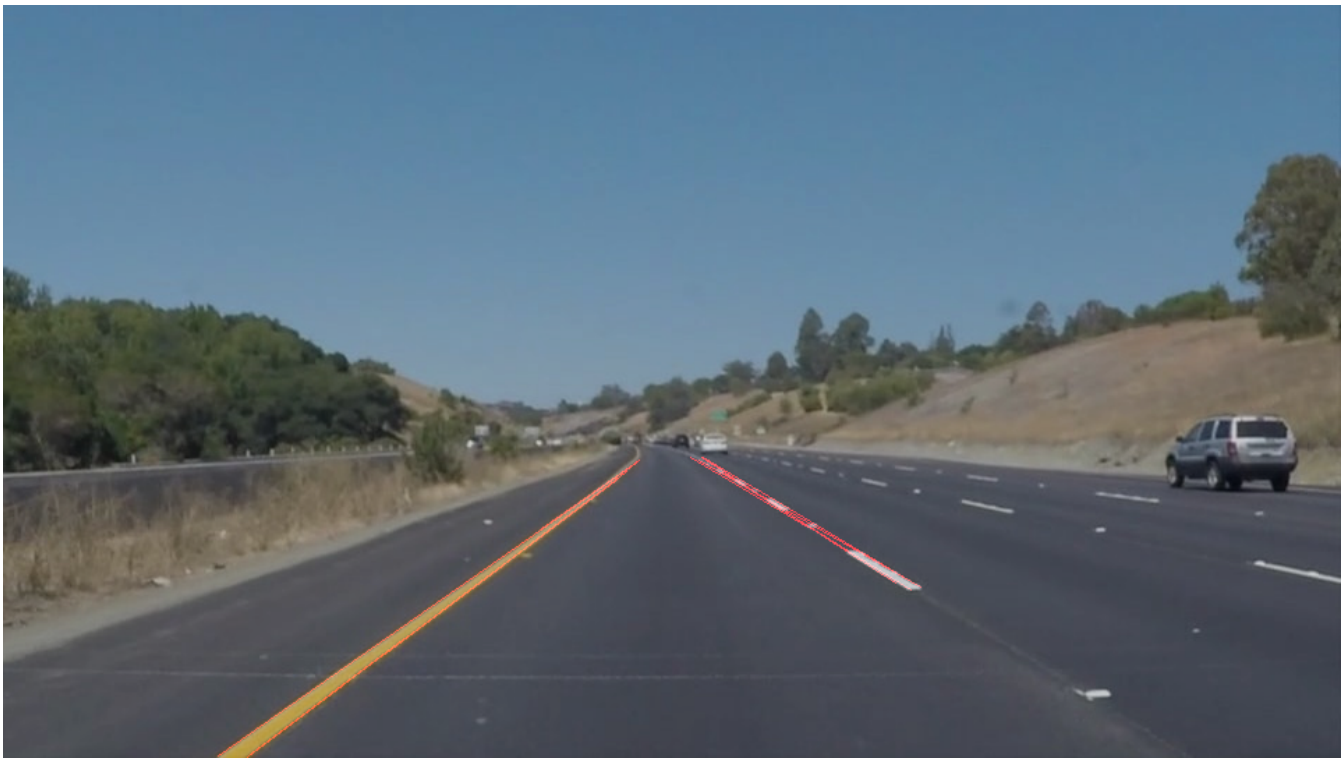


Line-Finding

Once the interested area is selected, line finding algorithm can be performed to find the lines in the area. The way the algorithm works is to transform the edges to Hough space. Those points on the same line will be transformed to the same point in Hough space. By setting up a threshold, we can filter out those lines that are too short.

Specifically, cv2.HoughLinesP() function is used to find the lines. And the hyper-parameters of interest are threshold, min_line_length and max_line_gap.

Here is one of the results after applying lane-finding algorithm on the image.



Note that what cv2.HoughLinesP() returns is an array of point pairs that represent lines. At the end of the day, what we want is only two lines that indicates the left and right of the current traveling lane. This leads to the find step of the pipeline: Lane-Interpolation.

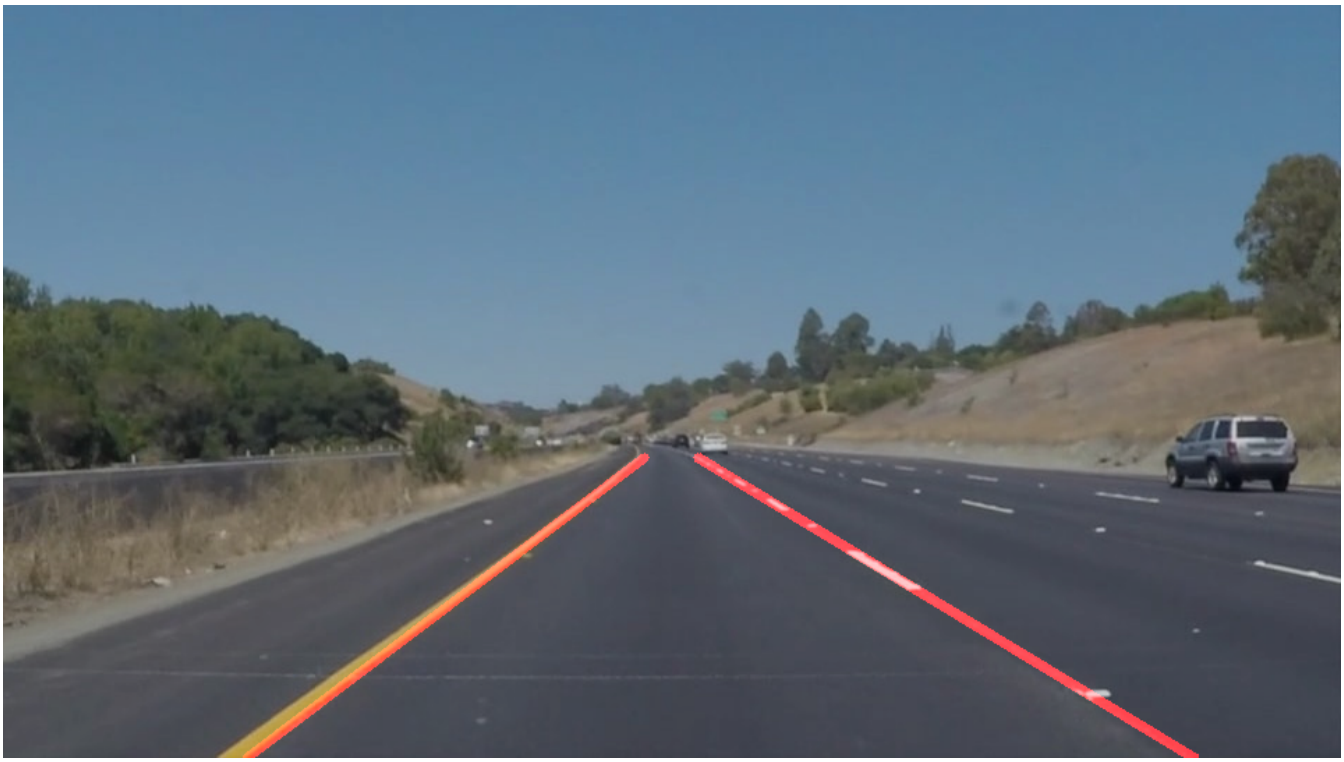
Lane-Interpolation

In the line-finding step, we got a few lines. Most of them are part of the lane boundaries. However, there could be a couple of lines that do not belong to lane marks. A threshold is set to filter out those lines with slope absolute value smaller than it. After the filtering, the lines are divides into two groups based on the sign of their slopes (From the camera's view angle, one lane boundary slope is negative and the other is positive).

After grouping the lines, we can interpolate a line in each group as the final lane boundary result. There are two things need to be done: 1. find the interpolated slope and intercept. 2. choose the start and end point to draw the interpolated line.

For the first task, I used the average slope and intercept in each group. For the second task, I chose the line start from y_full to 0.6 * y_full where y_full the full size of the image along y-axis. The x position can be calculated with the averaged slope and intercept from the first task.

For some frames, a vanilla average does not work well because of those lines from unrelated objects. What I did to alleviate this issue is to remove those lines that lie more than one standard deviation away before averaging(for details, please see interpolate_lane() function in the code). One of the results after interpolation is shown below:



3. Identify potential shortcomings with your current pipeline

One of the shortcomings of this pipeline is that it may not work well on unseen images. This is because, all the parameters are tunning on only a few experimental images. In this project, I found that my pipeline could work well on the 6 test images but still failed on several frames in the whole video. I have to extract those failed frames and retune the parameters on them to make sure the pipeline still work for them. One can imagine this kind of situation could happen again if we run this pipeline on new images that are different from tunning images(such as driving at night or rainy day).

4. Suggest possible improvements to your pipeline

One of the improvements we could do is to use the information from adjacent frames. Currently, although the pipeline could work on a video, it treats each frame individually. We didn't use any information from the previous frames. Since the lane cannot change abruptly, the lane in the current frame must be very close to those in previous frames. We can use this information to minimize the effect from unrelated lines when performing interpolation. For example, instead of doing a vanilla average, a weighted averaged could be used to calculate the slope and intercept. And the weights are inversely proportional to how different the slope is from the previous lane slope.