Finding Lane Lines on the Road

James A. Brodovsky

Describe your pipeline

I developed a pipeline that built on the process laid out over the course of the prior lesson. This involved five steps. First the imagery was loaded and converted to grayscale. Then I preprocessed the image with a Gaussian blur to dull any exceedingly sharp edges. This was then given to the Canny edge detection function to identify the edges present. The grayscale conversion to Canny edge detection was handled within the same line.

At this point the image was prepped for line-finding. A trapezoidal mask corresponding to the approximate area where the lane was anticipated to be was created. This allowed the program to reject any excessive edges that didn't correspond to the lane lines we were interested in. This was then fed into a Hough transformation function to turn the Canny points into lane lines. The transformation required a minimum threshold of 5 and a minimum line length of 10. This allowed for some additional "noise" in the image to be filtered out.

The Hough transformation developed short line segments. Each of these line segments were checked for slope. The pointes defining segments with a positive slope were put in a list for the right lane marker, and the negative slopes were put in a list for the left. These points were then fit to a first order polynomial function using numpy.polyfit() to define the left and right lane lines. Then using the top and bottom y-axis values of the trapezoid defining our area of interest the x-values for the start and end points of each lane-line segment were solved.

Running this pipeline over the video files it was found that the lines overlaid on the video were quit jittery. Going back and reanalyzing the pipeline it was found that occasionally a line segment along one side of the image would have a differing slope. For example: a positive slope despite being on the left side. An additional filtering step was implemented that only retained points on the list that were within one standard deviation. These filtered points were then fed into the polyfit function and the resulting output was much more stable.

Potential shortcomings

This feels like somewhat of a brute-force approach. There are four FOR loops in the code, two of which make up a nested pair. While not terribly slow, there was a discernable lag in the time it took the video file to be processed. This lag would be problematic in a self-driving car.

Secondly, while noticeably better, the line markers are still slightly jittery and barely work on the challenge video. This might be due to the curve invalidating the assumption of a first order polynomial.

Possible improvements

Instead of reconstructing a polynomial to fit to the lane lines, use an estimation technique to filter out the outlier points and to estimate coefficients of the polynomial.