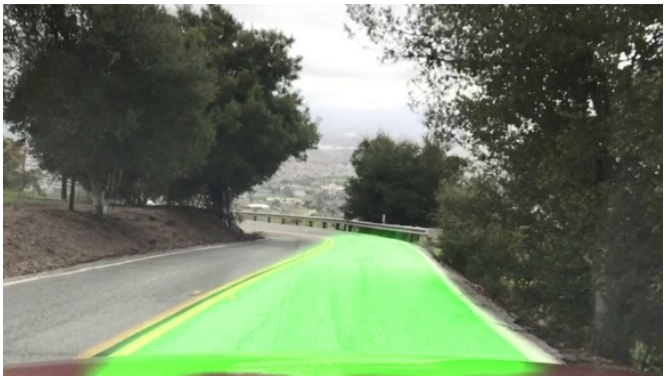
Lane Detection with Deep Learning (Part 1)

Michael Virgo Follow

May 10, 2017 · 9 min read

This is part one of my deep learning solution for lane detection, which covers the limitations of my previous approaches as well as the preliminary data used. Part two can be found here! It discusses the various models I created and my final approach. The code and data mentioned here and in the following post can be found in my Github repo.



People can find lane lines on the road fairly easily, even in a wide variety of conditions. Unless there is snow covering the ground, extremely heavy rainfall, the road is very dirty or in disrepair, we can mostly tell where we are supposed to go, assuming the lines are actually marked. While some of you reading this may already want to challenge whether other drivers actually succeed at staying within the lines (especially when you want to pass them), even without any driving experience, you know what those yellow and white lines are.

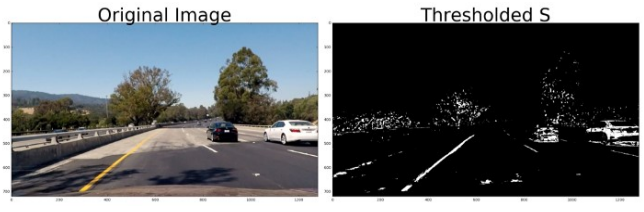
Computers, on the other hand, do not find this easy. Shadows, glare, small changes in the color of the road, slight obstruction of the line… all things that people can generally still handle, but a computer may struggle mightily with. It's certainly an interesting issue, so much that in Term 1 of Udacity's Self-Driving Car Nanodegree, they focused two of the five total projects to the problem. The first one was a nice introduction, serving to introduce students into some of the basic computer vision techniques, like Canny Edge detection.

Canny Edge Detection

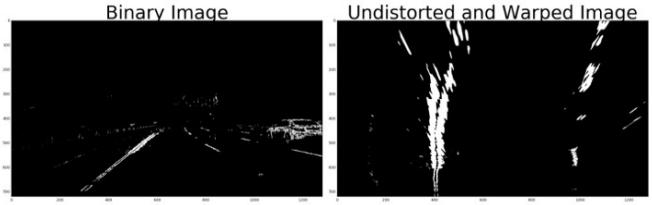
The second time around, in the overall fourth project of the term, we went a little deeper. This time, we used a concept called perspective transformation, which stretches out certain points in an image (in this case, the “corners” of the lane lines, from the bottom of the image where the lanes run beneath the car to somewhere near the horizon line where the lines converge in the distance) to destination points, which in the case of a road makes it look like you have become a bird flying overhead.

Perspective Transformation of an Image

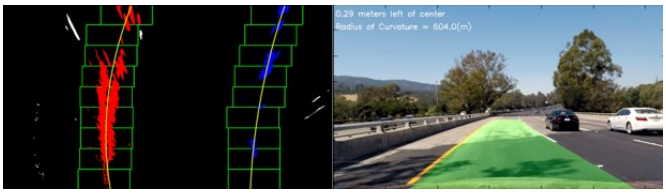
Before the perspective transformation, gradient (the change in pixel values as you cross an image, like where a dark road changes to bright lines) and color thresholds can be used to return a binary image, which is activated only where the values are above your given threshold. Once perspective transformed, a sliding window can then be run over the line to calculate the polynomial fit line for the curve of the lane line.



The ‘S’ channel, or Saturation, with binary activation



A few more thresholds (left) for activation, with the resulting perspective transformation



Sliding windows and a decent-looking result

This seems to work okay first, but there are some big constraints. First off, the perspective transformation is fairly specific to the camera (which also needs to individually be undistorted prior to the transformation), the mounting of that camera, and even the tilt of the road the car is on. Secondly, the various gradient and color thresholds only work in a small set of conditions—all those issues that I mentioned at the outset that computers have in seeing lane lines become very apparent here. Lastly, this technique was slow—the technique I used to generate predicted lane detections back on to video only ran at roughly 4.5 frames per second (fps), while video from a car would likely be coming in around 30 fps or more.

When Lane Detection Goes Wrong

. . .

Two of the other projects in Term 1 of the SDCND instead focused on deep learning, in these cases for traffic sign classification and behavioral cloning (getting a virtual car to steer at certain angles based on images it is fed, copying your behavior when you trained it). The last of the five projects also could potentially be approached with deep learning, although the primary method used a different machine learning technique. I actually had been putting off my Capstone project for the separate Machine Learning Nanodegree specifically in order to try to use a deep learning approach on Lane Detection, and now that I had completed all these projects, both from lane detection and from deep learning, it seemed like a perfect time to take a crack at my Capstone.

The Data

My first decision, perhaps the most crucial (and unfortunately, perhaps also the most time-consuming), was in deciding to create my own dataset. Although a lot of datasets are getting out there for use with training potential self-driving cars (increasing almost daily), they were not mostly labeled with regard to the car's own lane. Also, I thought it would be an interesting and rewarding challenge to create a sufficiently curated dataset to train a deep neural network.

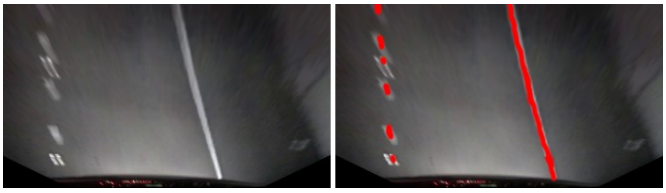
Collecting the data was easy. I live in San Jose, CA, and there are a wide variety of places I could drive. I knew from my previous deep learning projects how important a large dataset is, but perhaps even more so, how important having a *balanced* dataset is. I drove on the highway, side roads, up to a favorite hiking spot along very curvy roads up the mountainside, at night, in the rain (luckily the California drought completely flipped right as I went to collect this data). Although it might not sound like a lot, I ended up with nearly 12 minutes of driving time, which translated to over 21,000 individual image frames, all taken from my smartphone.

After finding a good method to extract the image frames, I almost immediately noticed an issue. While the videos where I drove slower in brighter conditions were comprised mostly of quality images, the night highway driving and the driving in the rain had a TON of blurry images (I blame the night highway driving issues on the bumpiness of San Jose highways as much as on my phone's issues in the dark). I had to go through each of the images individually, or else my neural network might never even be capable of learning anything. I was suddenly down to 14,000 images, still purposefully leaving in some that were slightly blurry to hopefully lead to more robust detection down the road.

An example of poor data obtained and removed. The trained model, that never saw this image, actually did okay in later predicting the lane for this frame!

In order to actually label my datasets, I planned to still use my old computer vision-based techniques. This meant running the images through my old algorithm, while instead of outputting the original image with the predicted lane drawn on top, would instead output six polynomial coefficients, or three for each lane line (i.e. the a, b and c in

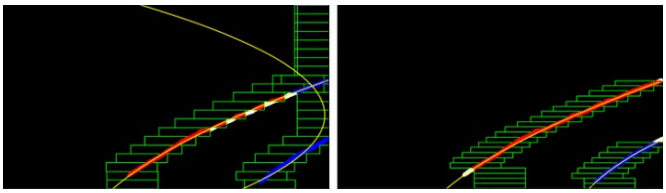
the equation ax^2+bx+c). Of course, if I was just going to purely use my old algorithm in labeling, I was only going to be training my neural network to have the same issues as the old model, and I wanted it to be more robust. I decided I was going to manually re-draw *over* the real lane lines in a single color (I picked red) so that I could use a red color channel threshold for better detection of the lane lines. In a brief moment of madness I originally thought I was going to do this for 14,000 images, but soon realized that would take way too long (unless I was as lucky/well-connected as comma.ai and could crowd-source my labeling). Instead, taking into consideration the impact of time on the data (the model could potentially "peek" into its own validation set if I fed it images within a few frames of each other at low speeds, given little change would be expected), I decided to only do this process for 1 in 10 images, thereby creating an early set of 1,400 training images.



Drawing over some blurry lines. Given the final model's ability to still detect lines even in blurry night images, it appears this method did assist in robustness.

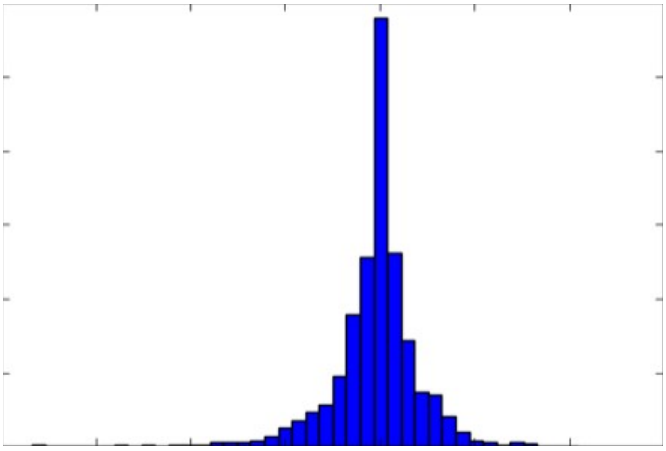
At this point, I created a program that could use my old CV-based model on a road image, detect the line polynomials, and use the polynomials to re-draw the lines, similar to what I had done in my previous project. This saved down each of the resulting images so I could check which were not sufficient labels. I noticed an immediate problem—although I had fed the old model a decent amount of curvy roads, it had failed to appropriately detect the lines on nearly all of those. Nearly 450 images of the 1,400 I was feeding in, mostly curves, could not be used.

However, I realized this was due to the nature of the original approach itself, due to how its sliding windows worked. If a lane line went off the side of the image, the original sliding window would continue *vertically* up the image, leading to the algorithm believing the line should be drawn toward that direction as well. This was solved by checking for whether the window box was touching the side of the image—if it was, and the windows had already progressed a little ways up the image (to prevent the model from completely failing at the start if the window was near the side to begin with), then the sliding windows would stop.



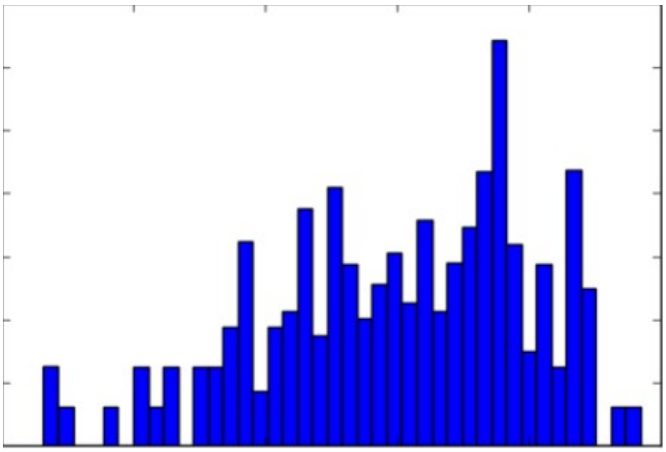
The original sliding windows on a curvy road, versus cutting the window search at the edge (also more windows)

It worked! I chopped the failure rate in half, down to ~225 failed images from the original ~450. They were still a lot of the extreme curves failing here, so I wanted to check what the actual distribution of the labels was. I did this by actually checking the distribution of each of the six coefficients through the use of histograms. Once again, I was disappointed. The data was still skewed too much toward straight lines. I even went back and added additional images only from the curvy road videos I took. When I looked at the images, the issue was still obvious—even on very curvy roads, the majority of the lanes were still fairly straight.



The original distribution of one of the lane line coefficients—too heavily centered around straight lines

One of the big difference-makers on my separate Traffic Sign Classification project had been creating “fake” data through adding in small rotations of the original image set for any traffic sign classes with few representations in the data. Using this approach again, for any of the coefficient labels outside a certain range of distribution (I iterated through roughly the outside 400, 100 and 30 images, meaning the furthest outside actually went through the process 3X), the image was rotated slightly while keeping the same label.



The more well-distributed data for a single line coefficient after adding image rotations for the curvier data

After this procedure, the data was finally a little more well-distributed for each coefficient. There were a few other quick pre-processing items I used for my images and labels as well. The training images were re-sized from the original 720 x 1280 (I ended up trying different versions shrunk by 4X, 8X and 16X) and normalized (which helps with eventual convergence of the model). I also normalized the labels using StandardScaler from sklearn (if you do this please make sure to **save the scaler** so that you can reverse it at the end of your model!). This normalizing of labels may trick you a little because the loss in training will automatically be lower, but I found the end results when drawn back onto images to be much more impressive as well.

I was finally ready to create and train a model.

Ready for Part 2? Read about my lane detection model here.