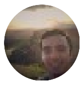
Lane Detection with Deep Learning (Part 2)


Michael Virgo Follow
May 11, 2017 · 12 min read

This is part two of my deep learning solution for lane detection, which covers the actual models I created in finding my final approach to the problem, as well as some potential improvements. Be sure to read Part One for the limitations of my previous approaches as well as the preliminary data used prior to the changes I made below. The code and data mentioned here and in the earlier post can be found in my Github repo.

With a decent dataset created, I was ready to make my first model for using deep learning to detect lane lines.

The Perspective Transformed Model

You may be asking, "Wait, I thought you were trying to get rid of perspective transformation?" And that's true. However, to create an initial model architecture, I wanted to check if it even looked like deep learning could learn to do the same thing as the CV-based model, given the somewhat limited dataset. So, the inputs here were perspective transformed road images, whereby it seemed logical that it might be easier for the neural network to learn the coefficients.

I used a similar model architecture here to what I used previously in the Behavioral Cloning project, including a batch normalization layer, followed by a number of convolutional layers, a pooling layer, flatten layer, and a number of fully-connected layers. The final fully-connected layer had an output size of six, for the number of lane line coefficients it was to be predicting. I also used Keras's ImageDataGenerator to try to make the model robust (mainly additional rotations, height shifts, and vertical flips—horizontal flips worried me as the labels would really need to be changed to accurately portray the line information). After a bit of fine-tuning (both for the model architecture, parameters, and the input image size), the model produced an okay result, but one that was very strictly reliant on the perspective transformation of the input and output—it could not generalize well at all.

The Road Image Model

After finding that deep learning could work with my dataset, I then went toward creating a model that could take in a road image without any perspective transformation. I utilized the exact same architecture as before, except for adding in a cropping layer to cut the top third of the image. My expectation was that the top third of any given road image would rarely, if ever, contain information necessary to detect the lane. The model easily converged to a similar result as the perspective-transformed model, and so I knew I could toss out the need for perspective transformation on the original image being fed to the model.

At this point, I wanted to give my model some additional data using a different camera than my own phone's (to account for different camera distortions), so I gave it some of the frames from Udacity's regular project video from my previous project. However, the problem here was in creating new labels for this data—the perspective transformation I used for *labeling* my own data was different than this separate video's. This also made me realize a huge problem with how I was approaching the issue in the first place—the labels themselves were polynomial coefficients in the bird's eye point of view, meaning that after prediction and drawing of the lane, the lane still needed to be inverse transformed back into the original image's point of view.

This was leading to a model that seemed fairly unable to generalize, but by looking at the produced result on only one of my own videos, I realized that perhaps the model was still correctly detecting the lines, just unable to reproduce a good visual result due to the perspective transformation still being used post-prediction. The model *seemed* like it was having trouble where the perspective would be different (in the case of both the separate Udacity video and in the more hilly spots of my own video). But what if it had actually learned to detect the lines—what if I could look directly at the activation of the layers itself? I could then potentially show the activation of the layers right on top of the original video.

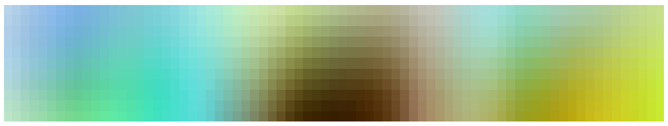
Activation Maps with keras-vis

I soon stumbled upon the great repository for keras-vis, which was exactly what I needed. I found a few research papers before this on the same concept, but I could not find the actual code necessary to reproduce the results. The keras-vis library was great because it allowed you to feed the trained model into a function, and return the activation maps for desired layers, technically made for each "class" in a typical classification neural network, but in this case for each of my polynomial coefficients.

Activation maps of the first few layers (note that the layers have been cropped by the top third)

This was a very interesting way to see exactly what my convolutional neural networks were looking at in my road images. Although the above image looks pretty good on the first layer, there were, of course, more problems with this approach.

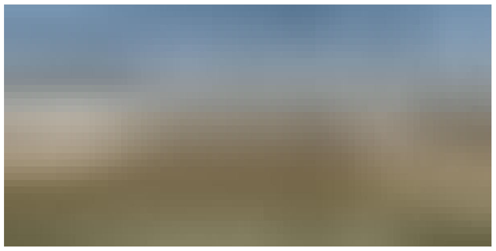
First off, in many of my curved road images, the model was actually only look at *one* of the lines. My model had learned that there was a relationship between the lines, which made since—for the most part, lane lines are going to be parallel, so if you only look at one line (assuming it is always the same of the two lines), you could get a general idea for where the other line was located. This did not help me though, as I wanted to be able to show where both lines or the lane itself was. Also, if you notice in the above, likely due to my image flips in the model's generator, the activation also occurs heavily in the sky—the model was learning to orient itself based off of where the sky was in the image, so I would have to somehow remove this activation area if I wanted to show it over my original video.


Deeper layers visualized

The second problem was even more confounding. Where the curved roads tended to activate over a single line and over the sky, straight roads instead activated over the car itself at the bottom of the image, as well as a space directly in front of the car itself, and were not activated on the lines themselves close by. Further up the image, the lines would then sometimes be activated. Without any consistency in activation between curves and straights, this approach could not work.

Transfer Learning

One of the last ditch attempts I made to save the approach with keras-vis was in using transfer learning. I had completely forgotten about my Behavioral Cloning project, in which a simulated car had learned to steer itself based on images presented to its own trained neural network. Wait a second—I trained a model using over 20,000 images to stay on the road—what was it looking at to do so?


One of the simulated car's input images

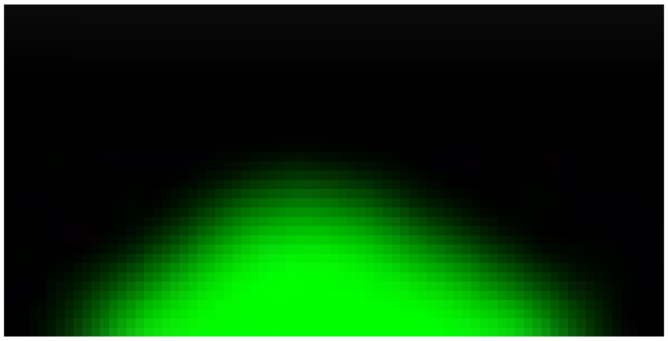
The answer was actually the whole road itself (there were not separate lanes in Udacity's simulator), but I wondered if I could use transfer learning to better focus the model's attention right off the bat. With a much larger dataset to begin the training with, I could then just add in a little extra training with the new data specific to my lane detection, and hopefully have better activation layers. Using model.pop() after loading the trained model from that project, I removed the final output layer from it(which was a single output, for the steering angle), and replaced it with an output of six for my coefficient labels. Please note if you are doing this yourself with a model to match up the input image sizes, or it isn't going to work.

It worked a little better, as the model, after some additional training with my own data, began to look at the lane lines instead of the entire road for activation. However, there were still big consistency issues with which parts of the image were getting activated, how highly activated they were, and more.

A Fully Convolutional Approach

Feeling like I was in the jaws of defeat, I began a search for a new approach. One of the recent areas I had seen a few different groups approach with respect to images from car cameras was image segmentation—dividing up a given image into classes like the road, cars, sidewalk, buildings, etc. SegNet was one of the image segmentation techniques I found interesting, and they had even posted about their general model architecture, which used convolutional layers (with batch normalization and ReLU activation) combined with pooling layers on the downswing, and upsampling (essentially unpooling) and deconvolutional layers from the midpoint out to the segmented image. This approach skips any fully-connected layers, leading to a fully convolutional neural network.

This seemed like a potential solution to the problem where the lane might get re-drawn in the wrong perspective—why not just have the neural network re-draw the predicted lane itself? I could still feed in the road images, while using the drawing of the lane that the original coefficient labels created as the new label (i.e. an output image). Given that I had been drawing the lanes in green already, I decided to make the output "filter" of my model be only the 'G' channel of RGB (and just stack two blank filters with it for the corresponding image to be blended with the original road image). This also helped allow me to *double the size my dataset*—remember how I was worried before about the potential ill-effects of horizontally flipping the images before, given that the coefficient labels would not match up well with what the neural network was seeing? By switching to this approach, in pre-processing the data I could flip the road image and the lane image label at the same time.

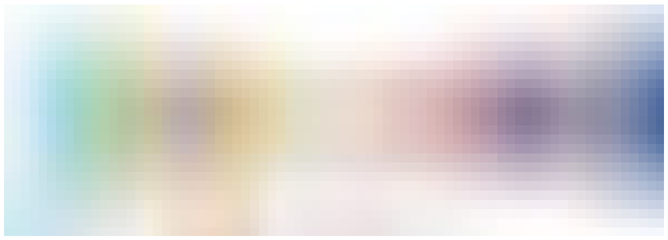

One of the new labels—a lane image

I'll take a quick step back to look at the dataset at this point in time:

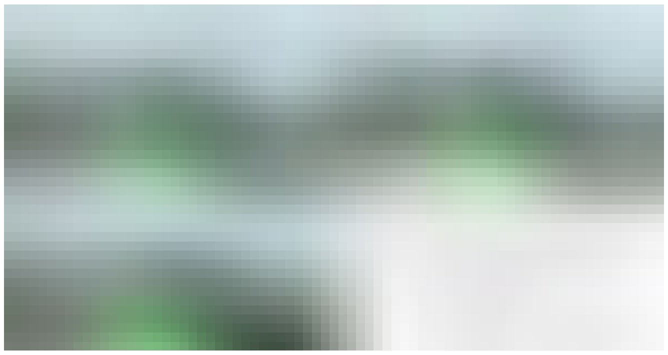
- My original data pull had 1,420 images (1 in 10 frames of the "good" frames from my video). I removed 227 of these that could not be appropriately labeled.
- I added 568 more images from the curved road videos only (from 1,636 I tried to use from those—a lot still failed due to the CV model I was using for labeling not being sufficient)
- Another 217 images from Udacity's regular project video
- In total, 1,978 images
- This was low, and not well-distributed, so with various small image rotations on the original coefficients not well-represented, I ended up with 6,382 images
- Horizontal flips doubled it to 12,764 images

Note that I was still able to use the image rotations for these new labels by just making sure to keep the road images, coefficient labels, and lane image labels linked up when I ran through which coefficients were outside the main distribution.

I wanted to follow SegNet's architecture fairly closely, given that I had never before used a fully convolutional neural network. Luckily, using Keras to do so was not too complicated—the one difference with the deconvolutional layers was that you had to make sure to actually match up the math correctly with what you were wanting to do. I made this easy by exactly mirroring my model after my final pooling layer, as well as using a slightly different aspect ratio with my input images of 80 x 160 x 3 (whereas keeping the original aspect ratio would have had 90 x 160 x 3). This was also done to keep the math easier—since I was using 2 x 2 for my pooling layers, starting at 90 would have quickly arrived at layers that could not be exactly divided by 2, leading to issues on the flip side of the model in trying to mirror the first half.

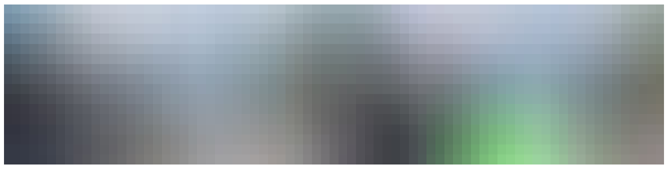

A quick mock-up of my new structure (left to right)

There were a few minor issues in the model I created, as I found I was running out of memory in trying to add batch normalization and drop-out to each of my convolutional & deconvolutional layers as I desired (in the above you'll note my widest layers at the start and end do not have drop-out, when those would really be the most effective spots for it). I settled for just batch normalization at the beginning and drop-out among the inner layers. The final deconvolutional layer outputs one filter (which I used as the 'G' color channel) with 80 x 160 dimensions to more easily match the original road image (but that could likely be smaller without any big issues). I also normalized the lane image labels by dividing by 255 prior to beginning training (meaning the output needs to be multiplied by 255 subsequent to prediction), which improved both convergence time as well as the final result.



Comparing results from different models

The end result was much better, as can be seen at the video here. This video, of course, has seen a chunk of the images already (probably somewhere between 10–20%), so it'd be cheating to stop there, although it was much more effective in areas where the old CV-based model had failed to produce sufficient labels originally. The true test was on Udacity's challenge video from the Advanced Lane Lines project—my model had never seen a single frame of it. But it did well on this video too! It had a bit of an issue with the shadows under the highway overpass, but had otherwise learned to generalize to a whole new video. Also, it was way faster than the old model—usually able to handle 25–29 frames per second with a GPU, just behind real-time of 30 fps. Even without GPU acceleration, it was still marginally faster than the CV model, coming in at 5.5 fps compared to only 4.5 fps before.



The old CV model vs. the new. You can't really see it, but the CV-based model actually thinks both lines are on the right.

. . .

Potential Improvements

That's my approach for lane detection with deep learning. It's not perfect of course. It is way more robust than the CV-based model, but in the Harder Challenge Video posted by Udacity, while making an admirable attempt, still loses the lane in the transition between light and shadow, or when bits of very high glare hit the window. Here are some of my ideas on improving my model in the future:

- More data. This is always the case in deep learning of course, but I think with an even wider array of conditions (such as those pesky transitions between light and shadow) and more different cameras, the model can get even better.
- Use of a recurrent neural network. I have yet to learn the technique behind actually creating the network architecture for one of these, but given its ability to use past information for the next prediction, I think this would be a very useful path to investigate. Interestingly enough, my current Term 2 studies with the SDCND are focusing on recursive methods (although not with deep learning) for localization, and I have a hunch there is a potential deep learning application there as well.
- Use of data without lane lines or only one line. This could of course be used in a lot more situations—many neighborhoods or areas outside of cities do not mark everything
- Expanding the model to detect even more items—similar to image segmentation, why not add in vehicle & pedestrian detection? I could do these on separate filters at the end, past the one single filter I used for the 'G' channel for the lane (and not necessarily just be limited to an 'R' and 'B' channel either). This may or may not be better than regular image segmentation.

I'd love to hear more thoughts on improvements to my approach as well. I really enjoyed figuring this approach out and I hope you enjoyed checking it out too!