Machine Learning Engineer Nanodegree

Capstone Project

Michael Virgo
May 4, 2017

Lane Detection with Deep Learning

Project Overview

One of the most common tasks while driving, although likely overlooked due to its constant use when a human drives a car, is keeping the car in its lane. As long as a person is not distracted, inebriated, or otherwise incapacitated, most people can do this after basic training. However, what comes very simply to a person – keeping the car between its lane's lines – is a much harder problem for a computer to solve.

Why is this complicated for a computer? To begin with, a computer does not inherently understand what the yellow and white streaks on a road are, the shifts in pixel values between those and the pixels representing the road in a video feed. One way to help a computer learn to at least detect these lines or the lanes itself is through various computer vision techniques, including camera calibration (removing the distortion inherent to the camera used), color and gradient thresholds (areas of the image where certain colors or changes in color are concentrated), perspective transformation (similar to obtaining a bird's-eye view of the road), and more. As part of the first term of the separate Self-Driving Car Nanodegree program, I was tasked with using some of these different computer vision techniques that require a decent amount of manual input and selection to arrive at the end result (see my Advanced Lane Lines project here).

With the knowledge gained from the Machine Learning Nanodegree the Deep Learning course on Udacity's website, I wondered if there might be a better approach to this problem - one directly involving deep learning. Deep learning involves utilizing multiple-layered neural networks, which use mathematical properties to minimize losses from predictions vs. actuals to converge toward a final model, effectively learning as they train on data.

Why this matters

You may say that a fully autonomous vehicle might not necessarily need to directly identify the lane lines - it might otherwise just learn that there are boundaries it is not meant to cross, but not see them as much different from other types of boundaries. If we're skipping straight to a fully autonomous vehicle, this may be true. However, for many consumers, they will likely see more step-by-step changes, and showing them (potentially on an in-vehicle screen) that the car can always sense the lanes will go a long way in getting them comfortable with a computer doing the driving for them. Even short of this, enhanced lane detection could alert an inattentive driver when they drift from their lane.

Problem Statement

Human beings, when fully attentive, do quite well at identifying lane line markings under most driving conditions. Computers are not inherently good at doing the same. However, humans have a disadvantage of not always being attentive (whether it be because of changing the radio, talking to another passenger, being tired, under the influence, etc.), while a computer is not subject to this downfall. As such, if we can train a computer to get as good as a human at detecting lane lines, since it is already significantly better at paying attention full-time, the computer can take over this job from the human driver. Using deep learning, I will train a model that can that is more robust, and faster, than the original computer vision-based model. The model will be based off a neural network architecture called a “convolutional” neural network, or “CNN” for short, which are known to perform well on image data. This is a great architecture candidate since I will feed the model frames from driving videos in order to train it. CNNs work well with images as they look first for patterns at the pixel level (groups of pixels around each other), progressing to larger and larger patterns in more expanded areas of the image.

Evaluation Metrics

As will be explained further in the Analysis section, my initial approach in the project was to teach a CNN to calculate the polynomial coefficients of the lane lines, and then draw the lane area based off of those lines. This approach, similar to a regression-type problem, made sense to use mean-squared error to minimize loss, meaning the difference between the actual coefficients and the model’s prediction (MSE uses the mean of all the squared differences to calculate loss). The final approach I used also utilized MSE, as I used a fully convolutional neural network (i.e. one that lacks any fully connected layers) to generate the green lane to be drawn onto the original image. Using MSE here meant minimizing the loss between the predicted pixel values of the output lane image and what the lane image label was. I will also evaluate it directly against my original pure computer vision-based model in both accuracy and speed, as well as on even more challenging videos than my CV-based model was capable of doing.

Analysis

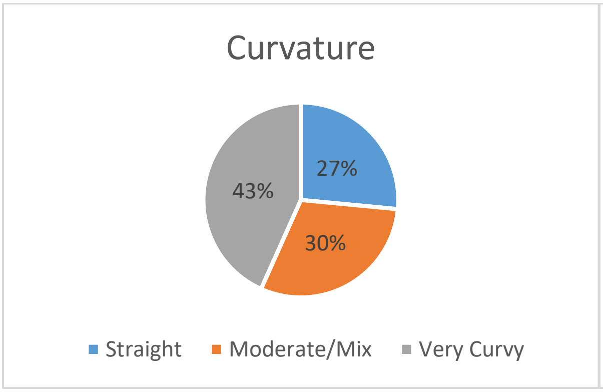
Datasets and Inputs

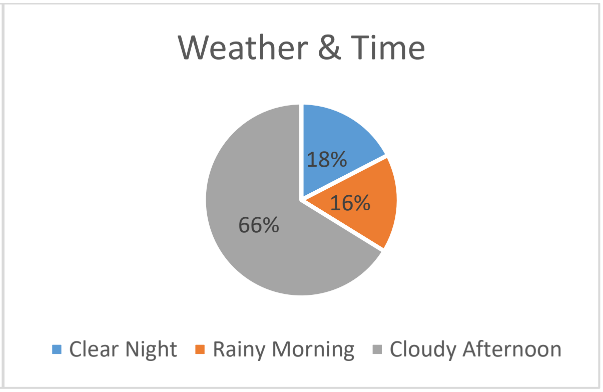
The datasets I used for the project are image frames from driving video I took from my smartphone. The videos were filmed in 720p in horizontal/landscape mode, with 720 pixels on the y-axis and 1280 pixels on the x-axis, at 30 fps. In order to cut down on training time, the training images were scaled down to 80 by 160 pixels (a slightly different aspect ratio than the beginning, primarily as it made it easier for appropriate calculations when going deeper in the final CNN architecture). In order to calculate the original labels, which were six coefficients (three for each lane line, with each of the three being a coefficient for a polynomial-fit lane line), I also had to do a few basic computer vision techniques first. I had to perform image calibration with OpenCV to correct for my camera's inherent distortion, and then use perspective transformation to put the road lines on a flat plane.

Initially, I wanted to make the model more robust my original model by drawing over the lane lines in the image, which can be blurry or fade away into the rest of the image the further to the back of the image it is. I drew over 1,420 perspective transformed road images in red, and ran a binary color threshold for red whereby the output image would show white wherever there had been sufficient red values and no activation (black) where the red values were too low. With this, I re-ran my original model, modified to output only the six coefficients instead of the lane drawing, so that I could train the network based on those coefficients as labels.

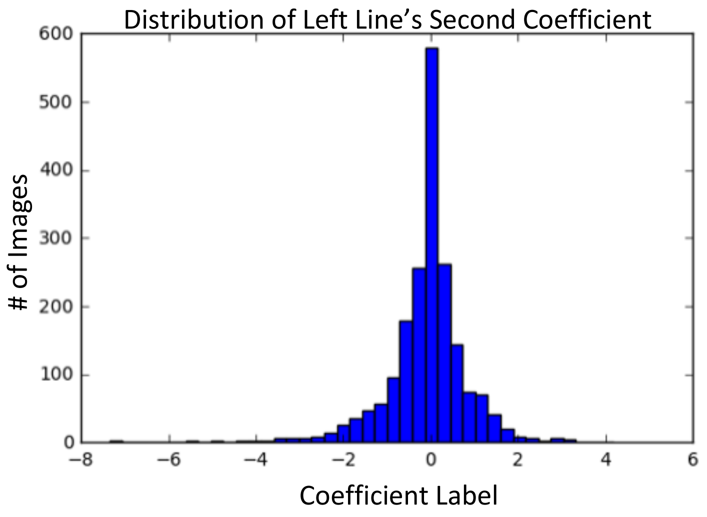
However, I soon found that the amount of data I had was not sufficient, especially because most of the images were of fairly straight lanes. So, I went back and ran the original CV model over all my videos from roads that were very curvy. I also added in a limited amount of images from the regular project video (I wanted to save the challenge video for a true test after finalizing my model) from Udacity's SDC Nanodegree Advanced Lane Lines project I previously completed, so that the model could learn some of the distortion from a different camera. However, this introduced a complication to my dataset – Udacity's video needed a different perspective transformation, which has a massive effect on the re-drawn lane. I will come back to this issue later.

I ended up obtaining a mix of both straight lines and various curved lines, as well as various conditions (night vs. day, shadows, rain vs. sunshine) in order to help with the model's overall generalization. These will help to cover more of the real conditions that drivers see every day. I will discuss more of the image statistics later regarding total training images used, but have provided two charts below regarding the percentage breakouts of road conditions for those obtained from my own driving video collected.





I noted before that one issue with the original videos collected was that too much of the data came from straight lanes, which is not apparent from the above charts – although "Very Curvy" made up 43% of the original dataset, which I initially believed would be sufficient, I soon found out the breakout was terribly centered around straight, as can be seen in the below chart from one of the coefficient labels' distributions. This is a definite problem to be solved.

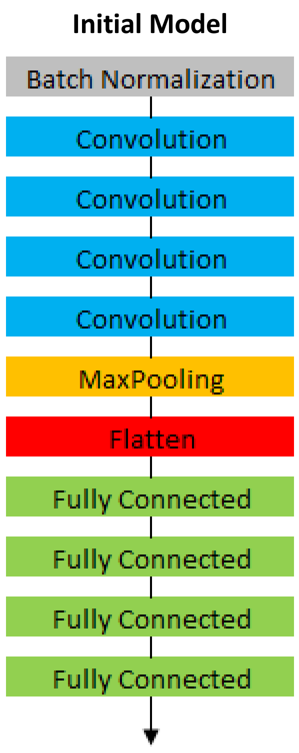


Algorithms and Techniques

First, I must extract the image frames from videos I obtain. Next, I must process the images for use in making labels, for which I will use the same techniques I previously used in my computer vision-based model. Next, I will calibrate for my camera's distortion by taking pictures of chessboard images with the same camera the video was obtained with, and undistort the image using "cv2.findChessboardCorners" and "cv2.calibrateCamera". With the image now undistorted, I will find good source points (corners of the lane lines at the bottom and near the horizon line) and determine good destination points (where the image gets transformed out to) to perspective transform the image (mostly by making educated guesses at what would work best. When I have these points, I can use "cv2.getPerspectiveTransform" to get a transformation matrix and "cv2.warpPerspective" to warp the image to a bird's eye-like view of the road.

From here, in order to enhance the model's robustness in areas with less than clear lane lines, I drew red lines over the lane lines in each image used. On these images, I will use a binary thresholding on areas of the image with high red values so that the returned image only has values where the red lane line was drawn. Then, histograms will be computed using where the highest amount of pixels fall vertically (since the image has been perspective transformed, straight lane lines will appear essentially perfectly vertical) that split out from the middle of the image so that the program will look for a high point on the left and a separate high point on the right. Sliding windows that search for more binary activation going up the image will then be used to attempt to follow the line.

Based off the detected pixels from the sliding windows, "numpy.polyfit" will be used to return polynomial functions that are most closely fit to the lane line as possible (using a polynomial allows for it to track curved lines as well as straight). This function actually returns the three coefficients of the "ax^2+bx+c" equation, where "a", "b" and "c" are the coefficients. I will append the total of six coefficients (three for each of the two lane lines) to a list to use as labels for training.



However, prior to training, I will want to check whether the labels are even accurate at all. Using the labels from above, I can feed an image through the original undistortion and perspective transformation, create an image "blank" with "numpy.zeros_like", make lane points from the polynomial coefficients by calculating the full polynomial fit equation from above for each line, and then use "cv2.fillPoly" with those to create a lane drawing. Using "cv2.warpPerspective" with the inverse of my perspective transformation matrix calculated before, I can revert this lane drawing back to the space of the original image, and then use "cv2.addWeighted" to merge the lane drawing with the original image. This way, I can make sure I feed accurate labels to the model.

Lastly, my project will use Keras with TensorFlow backend in order to create a convolutional neural network. Using "keras.models.Sequential", I can create the neural network with convolutional layers (keras.layers.Convolution2D) and fully-connected layers (keras.layers.Dense). I will first try a model architecture similar to the one at left, which I used successfully in a previous project for Behavioral Cloning.

Benchmark

I plan to compare the results of the CNN versus the output of the computer vision-based model I used in my SDC Nanodegree project (linked to above). Note that because there is not a "ground-truth" for my data, I cannot directly compare to that model from a loss/accuracy perspective, but as the end result is very visual, I will see which model produces the better result. Part of this comes down to robustness – my pure CV model failed to produce lane lines past the first few seconds of a Challenge video in my previous project. If this model can mostly succeed on the Challenge video (i.e. no more than a few seconds without the lane shown) without having been specifically trained on images from that video, it will have exceeded this benchmark. A second benchmark will be the speed of the model – the CV-based model can only generate roughly 4.5 frames per second, which compared to 30 fps video incoming is much slower than real-time. The model will exceed this benchmark if the writing of the video exceeds 4.5 fps.

Methodology

Data Preprocessing

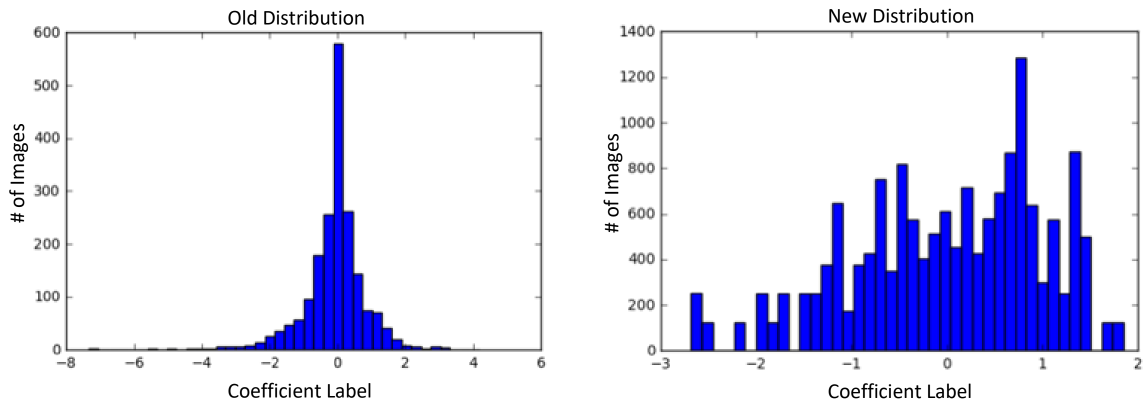
Some of the general techniques I used to preprocess my data to create labels are discussed above in the “Algorithms and Techniques” section. However, there was a lot more to making sure my model got sufficient quality data for training. First, after loading all the images from each frame of video I took, an immediate problem popped up. Where I had purposefully gathered night video and rainy video, both of these severely cut down on the quality of images. In the night video, where my smartphone camera already was of slightly lesser quality, I was also driving on the highway, meaning a much bumpier video, leading to blurry images. I sorted through each and every single gathered image to check for quality, and ended up removing roughly one-third of my initial image data from further usage.

From here, I also wondered whether the model might overfit itself by getting a sneak peek at its own validation data if images were too similar to each other – at 30 frames per second, there is not a whole lot of change from one frame to the next. I decided to only use one out of every ten images for training. With these, I drew over the images in red as mentioned above, and then ran my programs for making labels and checking the labeled images. Here, I found that process for making the labels was flawed – the original code for my sliding windows failed completely on curves. This was because the initial code, when it hit the side of an image, would keep searching straight up the image – causing the polynomial line to think it should go up the image too. By fixing my code to end when it hit the side of the image, I vastly improved its usefulness on curves. After re-checking my labels and tossing out some bad ones, I moved on to check the distribution of my labels. It was bad – there were still hardly any big curves! Even the videos from curvy roads still had mostly straight lines. So, I re-ran my process over one in every five images, only from the four videos with mostly curved lines.

I also decided at this point to add a little bit of the Udacity basic project video from my CV-based project (not from the Challenge video though because I wanted that as the true test of the model’s ability to generalize) in order to train the model for different distortion (which I had already obtained previously). However, this caused a new issue – these images needed a different perspective as well. This was not so hard for creating the labels, but I knew it could cause an issue on the tail end, as checking the labels would require the specific inverse perspective transformation to re-draw the lines.

As shown previously “Datasets and Inputs” section, after adding in this additional data, my distribution was still fairly unequal. However, I found by using the histograms of the distributions of each label, I could find where the exact values were where only a limited amount of images fell. By iterating through each of the labels, and finding which training images were on the fringes of the data, I could then come back and generate “new” data; this data was just rotations of these images outside the main distribution, but my CNN would likely become much more likely to not overfit to straight lines.

Improving the Distribution of Lane Labels



The changes in the distribution of image labels for the second coefficients are shown above. I originally normalized the labels with sklearn's "StandardScaler" (causing the differences in values above), which improved training results but also did need to be reversed after training to return the correct label.

At this point, the approach depending on the model diverge. In my initial models, I took in either a perspective transformed image or regular road image, downscaled it from 720x1280x3 to 45x80x3 (scaling down 16X), gray-scaled the image, added back a third dimension (cv2.cvtColor removes the dimension when gray-scaling but Keras wants it to be able to run properly), and then normalized the image (new_image = (new_image / 255) * .8 - 1) to be closer to a mean of zero and standard deviation of one, which is key in machine learning (the algorithms tend to converge better).

On the flip side, once I changed to a fully convolutional model, I instead was only down-sizing the road images to 80x160x3 (a slightly different aspect ratio than the original but roughly 8X scaled down), without any further gray-scaling or normalization (I instead on my Batch Normalization layer in Keras to help there). Additionally, since a fully convolutional model essentially returns another image as output, instead of saving down my lane labels as numbers only, I also saved down the generated lane drawings prior to merging them with the road image. These new lane image labels were to be the true labels of my data. I still used my image rotations to generate additional data, but also rotated the lane image labels along with the training image (based on the distributions of the original label coefficients still). I also added in a horizontal flip of each image and corresponding lane image label to double my dataset for training. For these new lane image labels, I dropped off the 'R' and 'B' color channels as the lane was being drawn in green, hoping to make training more efficient with less of an output to generate.

Image Statistics

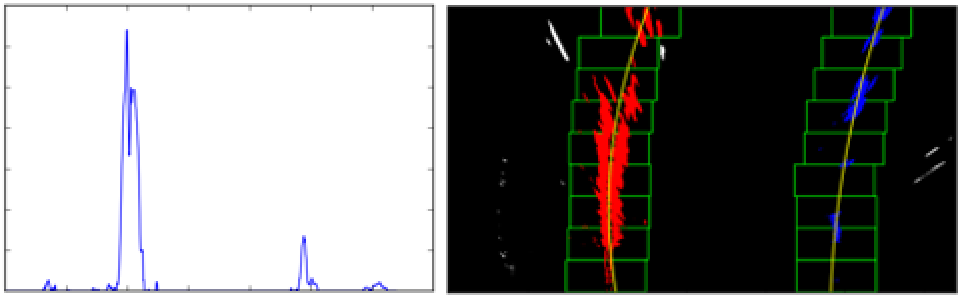
Here are some statistics from my data pre-processing:

- 21,054 total images gathered from 12 videos (a mix of different times of day, weather, traffic, and road curvatures – see previous pie chart breakout)
- The roads also contain difficult areas such as construction and intersections
- 14,235 of the total that were usable of those gathered (due to blurriness, hidden lines, etc.)
- 1,420 total images originally extracted from those to account for time series (1 in 10)
- 227 of the 1,420 unusable due to the limits of the CV-based model used to label (down from 446 due to various improvements made to the original model) for a total of 1,193 images
- Another 568 images (of 1,636 pulled in) gathered from more curvy lines to assist in gaining a wider distribution of labels (1 in every 5 from the more curved-lane videos; from 8,187 frames)
- In total, 1,761 original images
- I pulled in the easier project video from Udacity's Advanced Lane Lines project (to help the model learn an additional camera's distortion) - of 1,252 frames, I used 1 in 5 for 250 total, 217 of which were usable for training
- A total of 1,978 actual images used between my collections and the one Udacity video
- After checking histograms for each coefficient of each label for distribution, I created an additional 4,404 images using small rotations of the images outside the very center of the original distribution of images. This was done in three rounds of slowly moving outward from the center of the data (so those further out from the center of the distribution were rotated multiple times). 6,382 images existed at this point.
- Finally, I added horizontal flips of each and every road image and its corresponding label, which doubled the total images. All in all, there were a total of 12,764 images for training.

Implementation

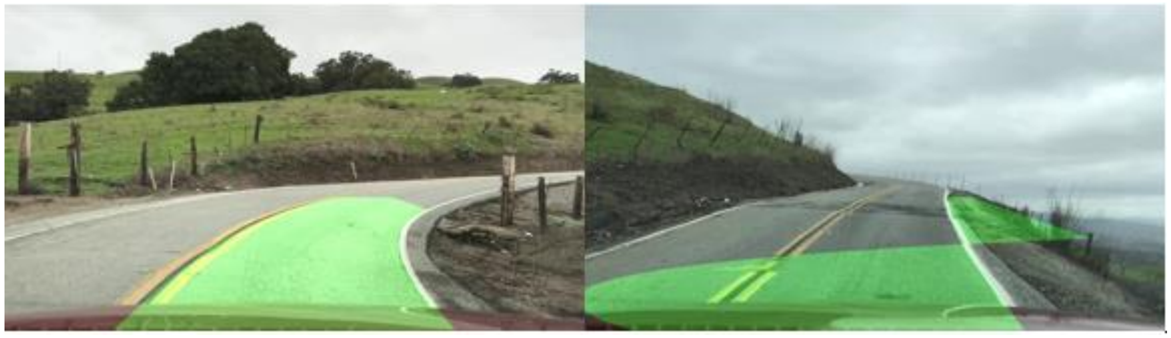
My first CNN built used perspective transformed images as input, and can be seen in the "perspect_NN.py" file. It used batch normalization, four convolutional layers with a shrinking number of filters, followed by a pooling layer, flatten layer, and four fully-connected layers, with the final fully-connected layer having six outputs – the six coefficient labels of the lane lines. Each layer used RELU activation, or rectified linear units, as this activation has been found to be faster and more effective than other activations. I tried some of the other activations as well just in case, but found RELU to be the most effective, as expected. Also, in order to help prevent overfitting and increase robustness, I added in strategic dropout to layers with the most connections, and also used Keras's ImageDataGenerator to add in image augmentation like more rotations, vertical flips, and horizontal shifts. I originally used mean-squared-error for loss, but found that mean-absolute error actually produced a model that had could handle more variety in curves. Note that I also made the training data and labels into arrays before feeding the model as it works with Keras. Also, I shuffled the data to make sure that the different videos were better represented and the model would not just overfit on certain videos. Last up was splitting into training and validation sets so I could check how the model was performing.

Perspective Image's Histogram and Sliding Windows



After training this first model and creating a function to actually see the re-drawn lanes, I found this first model to be moderately effective, given that you were using the same perspective transformation from the original model (see video here). However, my end goal was to ignore the need to perspective transform an image for the CNN altogether, so after finding that the first model was moderately effective at producing a re-drawn lane, I shifted course. In "road_NN.py", this second model is included. Other than feeding in a regular road image, the only change I made to this model was adding a Crop layer, whereby the top third of the image was removed (I played around with one half or one third without much difference). I found quickly that the CNN could, in fact, learn the lane coefficients without perspective transformation, and the resulting model was actually a little bit more effective even (see video here).

Good Prediction vs. Poor Prediction



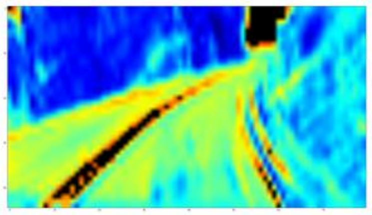
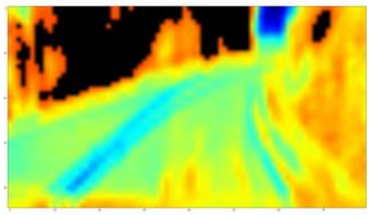
There was still one big problem – if my model was predicting the lane label coefficients, this meant that the lines still need to be drawn in a perspective-transformed space, and reverted to the road space, even if the original image was not perspective-transformed. This caused massive issues in generalizing to new data – I would need the transformation matrix of any new data (even slight changes in camera mounting would cause a need for a new matrix).

Refinement

My first thought was whether or not I could actually look directly at the activation of the convolutional layers to see what the layer was looking at. I assumed that if the CNN was able to determine the appropriate line coefficients, it was probably activating over the actual lines of lane, or at least some similar area in the image that would teach it the values to predict.

After some research, I found the keras-vis[1] library to be great for looking at the activation of each layer. This library can actually look at the class activation maps (in my case the "classes" are actually each of the coefficient labels since this is not a classification problem) in each layer. I thought I had found my solution, until I looked at the activation maps themselves.

Class Activation Maps

While the above activation maps of the first few layers look okay, these were actually some of the clearest I could find. Interestingly enough, the CNN actually often learned by looking at *only one lane line* – it was calculating the position of the other line based off of the one it looked at. But that was only the case for curves – for straight lines, it was not activating on the lane lines at all! It was actually activating directly on the road in front of the car itself, and deactivating over the lane lines. As a result, I realized the model was activating in different ways for different situations, which would make using the activation maps directly almost impossible. Also, notice in the above second image that the non-cropped part of the sky is also being activated (the dark portion) – due to the various rotations and flips, the model was also activating in areas that was telling it top from bottom. Other activation maps also activated over the car at the bottom of the image for the same purpose.

I also briefly tinkered with trying to improve the activation maps above by using transfer learning. Given that in my Behavioral Cloning project, the car needed to stay on the road, I figured it had potentially learned a similar, but perhaps more effective, activation. Also, I had tens of thousands of images to train on for that project, so the model was already more robust. After using "model.pop" on that model to remove the final fully-connected layer (which had only one output for that project), I added a new fully-connected layer with six outputs. Then, I trained the already-established model further on my real road images (the old model was trained on simulated images), and actually found that it did a better job on looking at both lines, but still failed to have a consistent activation I could potentially use to re-draw lines more accurately.

At this point, I began to consider what I had read on image segmentation, especially SegNet[2], which was specifically designed to separate different components of a road out in an output image by using a fully convolutional neural network. This approach was different from mine in that a *fully* convolutional neural network does not have any fully-connected layers (with many more connections between them), but only uses convolutional layers followed by deconvolutional layers to essentially make a whole new image. I realized I could skip the

[1] https://github.com/raghakot/keras-vis
[2] http://mi.eng.cam.ac.uk/projects/segnet/#research

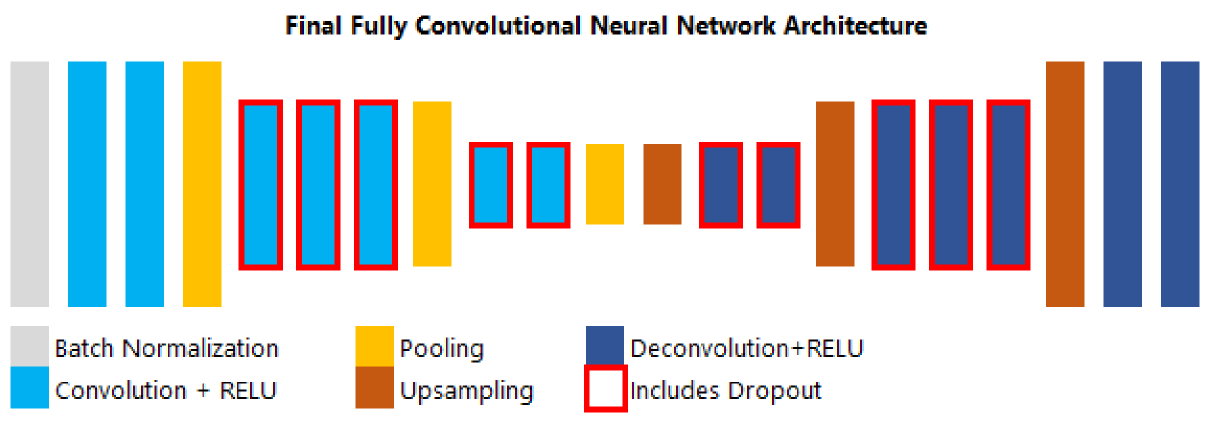
undoing of the perspective transformation for the lane label entirely but actually training directly to the lane drawing as the output. By doing so, it meant that even if the camera was mounted differently, the lanes were spaced differently, etc., the model would still be able to return an accurate drawing of the predicted lane.

Results

The Final Model

Although I had made a CNN previously that ended in fully-connected layers, I had never before made a fully convolutional neural network, and there were some challenges in getting the underlying math to work for my layers. Unlike in the forward pass in normal Convolution layers, Keras's Deconvolution layers flip around the backpropagation of the neural network to face the opposite way, and therefore need to be more carefully curated to arrive at the correct size (including the need to specify the output size). I chose to make my new model a mirror of itself, with Convolutional layers and Pooling in slowly decreasing in size layers, with the midpoint switching to Upsampling (reverse-pooling) and Deconvolution layers of the same dimensions. The final deconvolution layer ends with one filter, which is because I only wanted a returned image in the 'G' color channel, as I was drawing my predicted lanes in green (it later is stacked up with zeroed-out 'R' and 'B' channels to merge with the original road image). Choosing to input 80x160x3 images (smaller images were substantially less accurate in their output, likely due to the model being unable to identify the lane off in the distance very well) without gray-scaling (which tended to hide yellow lines on light pavement), I also normalized the incoming labels by just dividing by 255 (such that the labels were from 0 to 1 for 'G' pixel values).

The final model is within the "fully_conv_NN.py" file. I stuck with RELU activation and some of the other convolution parameters (strides of (1,1) and 'valid' padding had performed the best) from my prior models, but also added more extensive dropout. I had wanted to use dropout on every Convolutional and Deconvolutional layer, but found it used up more memory than I had. I also tried to use Batch Normalization prior to each layer but found it also used up too much memory, and instead I settled for just using it at the beginning. A more interesting discovery, given that using MSE for loss had previously failed, was that it performed much better than any other loss function with this new model. Even more intriguing was that adding *any* type of image augmentation with ImageDataGenerator, whether it be rotations, flips, channel shifts, shifts along either the horizontal or vertical axes, etc., actually caused the model to have substantially less convergence regardless of how many epochs the model ran on, as well as a worse result on any test images I looked at. Typically, I expect the image augmentation to improve the final model, but in this case, skipping any augmentation (although I kept the generator in anyway without it, as it is good practice) lead to a substantially better model. This is fed into the "draw_detected_lanes.py" file, in which the model predicts the lane, it is averaged over five frames (to account for any odd predictions), and then merges with the original road image from a video frame.



Evaluation and Validation

After 20 epochs, my model finished with MSE for training of 0.0046 and validation of 0.0048, which was significantly lower than any previous model's I had tried (although a bit of apples and oranges against the models using six polynomial coefficients as labels). I first tried the trained model against one of my own videos, one of the hilly and curved roads for which the model had potentially seen up to 20% of the images for, although likely much less – from the image statistics earlier, I had to throw out a large portion of the images from these videos, so even though I ran it on one in five images, the model probably only saw 5-10% of them. Fascinatingly, the model performed great across the entire image, only losing the right side of the lane at one point when the line became completely obscured by leaves. The model actually performed near perfectly even on a lot of the areas I knew I had previously had to throw out, because my CV-based model could not appropriately make labels for them. The output video can be seen here.

Justification

However, the fact remained that the model had in fact seen some of those images. What about trying it on the Challenge video[3] created by Udacity for the Advanced Lane Lines project? It had never been trained on a single frame of that video. Outside of a small hiccup going under the overpass in the video, the model performed great, with a little bit of noise on the right side where the separated lane lines were. It had passed my first benchmark – outperforming my CV-based model, which had failed on this video. This video can be seen here.

My second benchmark was with regards to speed, and especially when including GPU acceleration, the deep learning model crushed the earlier model – it generated lane line videos at between 25-29 fps, far greater than the 4.5 fps for the CV model. Even without GPU acceleration, it still averaged 5.5 fps, still beating out the CV model. Clearly, GPU acceleration is key in unlocking the potential of this model, running almost real-time with 30 fps video. With regards to both robustness and speed, the deep learning-based model is a definite improvement on the usual CV-based techniques.

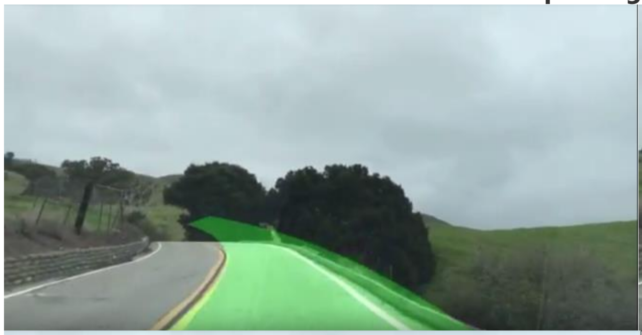
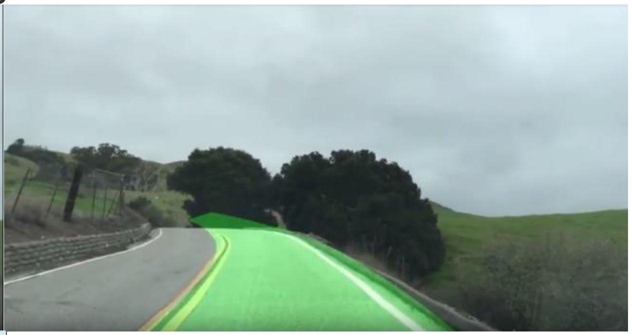
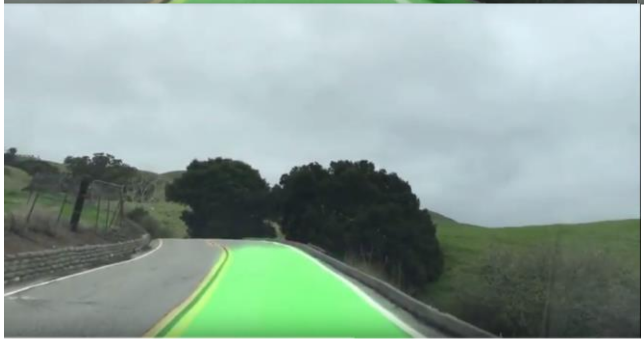
[3] https://github.com/udacity/CarND-Advanced-Lane-Lines/blob/master/challenge_video.mp4

Conclusion

More Visualizations

Below I have included some additional visualizations, comparing the various stages of my own model as well as in comparison to my original model using typical computer vision techniques.

Improving Models

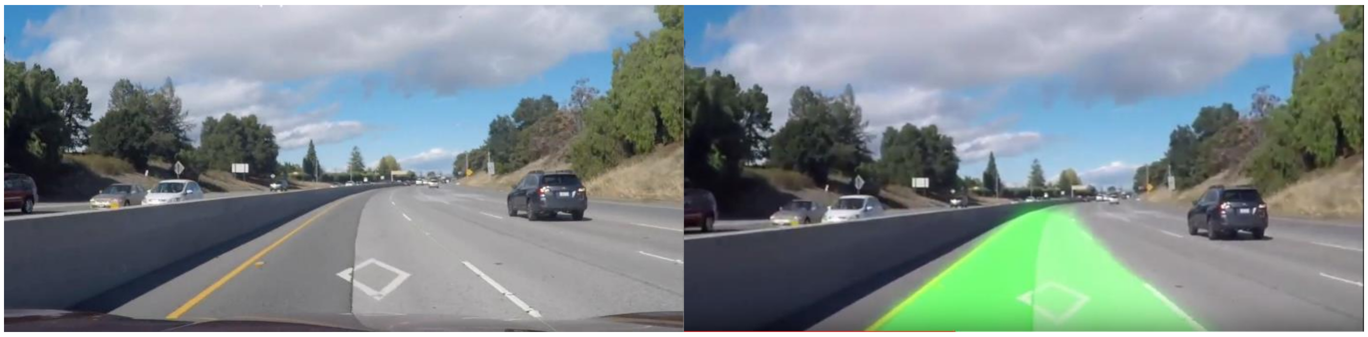
Top left: Input – Perspective Transformed Image
Output – Six polynomial coefficients

Top right: Input – Road Image
Output – Six polynomial coefficients

Bottom left: Input – Road Image
Output – Lane in 'G' color channel

The CV-based model believed both lines to be on the right side of the lane, hence only a faint line and not a full lane drawn. Some of this comes down to weaknesses in the algorithm there, which lacked checks to see whether the lanes were separate from each other.

Computer Vision Techniques Model vs. Deep Learning Model
Udacity Challenge Video Output



Reflection

My project began with collecting driving video, which I then extracted the individual frames from. After curating the data to get rid of various blurry or other potentially confusing images, I calculated the calibration needed to undistort my images, and perspective transformed them to be able to calculate the lines. After additional image processing to improve the dataset at hand, I then created six coefficient labels, three each for both lane lines. Next, I created a program to make those labels into re-drawn lanes, and then had to improve my original label checking algorithm to work better for curves. Following this, any still poorly labeled images were removed from the dataset.

After checking histograms of the coefficient labels, I realized I needed additional curved line images, and gathered additional data for curved lines, as well as from a different camera, in order to help even out the distribution. After finding they still needed a better distribution, I found ranges of the labels to iterate through and create additional training images through rotation of the originals.

The next step was to actually build and train a model. I built a somewhat successful model using perspective-transformed images, built a slightly improved model by feeding in regular road images, but still was not at a sufficient level of quality. After trying to use activation maps of the convolutional layers, I moved on to a fully convolutional model. After changing the training labels to be the 'G' color channel containing the detected lane drawing, a robust model was created that was faster and more accurate than my previous model based on typical computer vision techniques.

Two very interesting, but very challenging issues arose during this project. I had never before used my own dataset in training a model, and curating a good dataset was a massive time commitment, and especially due to the limits of the early models I used, often difficult to tell how sufficient of a dataset I had. The second challenge was in settling on a model – I originally worried I would have to also somehow train the neural network to detect perspective transformation points or similar. Instead, I learned for the first time how to use a fully convolutional neural network, and it solved the problem.

Improvement

One potential improvement to the model could be the use of a recurrent neural network (RNN). The current version of my model uses an averaging across five frames to smooth out any issues on a single frame detection, outside of the actual neural network itself. On the other hand, a RNN would be able to directly look at previous frames in order to learn that what was detected in a previous frame matters to the current frame. By doing so, it would potentially lose any of the more erratic predictions entirely. I have not yet used a RNN architecture, but I plan to do so eventually for future projects.