Training and Testing an Autonomous Driving System

Michael Martinez

ADS Background and Project Overview

Autonomous Driving Systems (ADS) have often been a phenomenon confined to science fiction media. However, recent advancements in computer vision, computation power, and big data technologies have allowed this phenomenon to become a reality. At the heart of every ADS or self-driving car is the ability to sense its environment. In fact, this ability is the most important feature in driving an automobile. We humans almost take this ability for granted when we get behind the wheel, but for an ADS the process of learning this task and implementing it takes years of optimization and incorporation of several technologies.

This project aims to optimize a very important variable that an ADS must evaluate after analyzing its environment: the steering angle. The steering angle is incredibly important to a self-driving car to ensure that it handles turns well, stays in its lane, avoids objects and collisions, and performs correctly in several other scenarios.

Udacity has open sourced software that puts a user behind the wheel of a car. The user has the choice of two tracks to drive around: a lake or a mountain. This simulator, built on Unity, has a couple of features key to this project - the ability to *collect* various driving data, and the ability to *test* a trained model on a specified track. The simulated car is equipped with 3 cameras on its windshield that capture a center, left, and right view of the car’s frontal surroundings. Furthermore, the car is equipped with sensors that measure throttle, speed, reverse, and steering angles. Thus, the data collected consists of 3 images and 4 measured variables per recorded observation. To generate this data, a user drives around the track with the *record* option toggled on. Therefore, my project will be an exercise in *behavioral cloning*- where I generate my own data and train a model based upon it.

Behavioral cloning is a specification of artificial intelligence. In this regime, an AI learns from a *human's* interpretation of a certain task. In other words, an AI seeks to *clone* a human's *behavior* as they accomplish a specific task. In my instance, the model I train will attempt to clone the behavior I exhibit as I drive around the track. This type of AI specification is very popular in today's leading real self-driving cars. And this makes sense because after all, driving was first made to accommodate humans and therefore the infrastructure, laws, and decisions have all been installed with humans in mind. It is advantageous then, for an ADS to clone a human's driving behavior.

Once sufficient data is collected, it will be imported into a Jupyter notebook to be analyzed and processed. For this project, I will be using a leading convolutional neural network (CNN) design developed by NVIDIA and implementing it using Keras. As with any CNN, images usually must be preprocessed to ensure optimal model performance. Therefore, I will preprocess all images according to industry-standard ADS techniques before training a model on them.

After training a model, the final step is to put it to the test on the track. The car will be trained on the lake track *only*, which means it will have never have "seen" the mountain track. Due to this training constraint, I fully expect the trained model to be able to complete several laps of the lake track without going off track or colliding into anything. However, the lake and mountain tracks differ quite dramatically in several aspects such as surrounding environment, turn angles, inclines, brightness, and several more variables. Therefore, the mountain track will serve as a nice benchmark to see how well the model can generalize to new environments.

This project aims to serve as a foundational platform for an autonomous driving system by addressing a fundamental variable- steering angle. Ultimately the results of this project seek to benefit the self-driving car industry and contribute to ADS technologies.

Gathering, Analyzing and Processing Data

As described in the previous section, I was responsible to generate and gather my own data in the simulation. I gathered data exclusively on the lake track to train my model and saved the additional track as a validation track. The two tracks as seen in the simulation are shown below in Fig. 1.

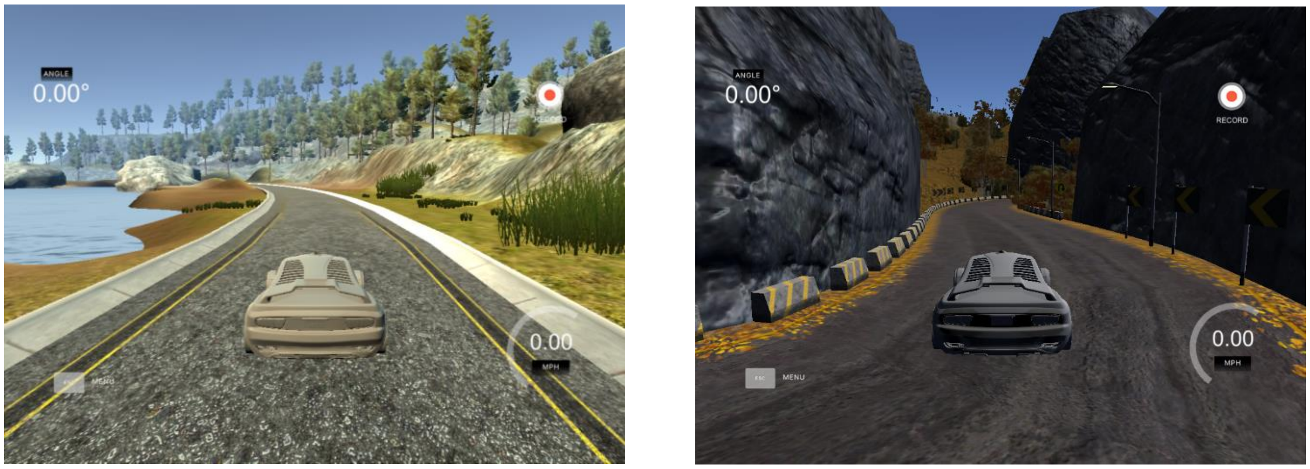

Figure 1. Screen Captures of the simulated car on the lake (shown on the left) and mountain (shown on the right) tracks. The lake track serves as the training track, and the mountain track serves as the validation track.

There were several iterations of the generating process that were performed based upon the data analysis and model performance. The final iteration of the gathering process included 15759 observations, each of which including the center, left, and right frontal images and the 4 additional sensory variables. This final iteration included several laps around the lake track in *both* directions. The lake track is heavily left-turn biased and therefore not an ideal model to train on. Therefore, I included data driven in the reverse direction of the track, thus countering the left-turn bias with right-turn data. Furthermore, I padded the data with supplemental *recovery* data to optimize the cars ability to drive in the center of the lane. I define a recovery as the process of the car steering back into the center of the lane after it veers onto the edge of the lane. I load and format the gathered data into a pandas dataframe to begin the analysis. The dataframe consists of file paths to the images (center, left, and right) on my local disk as well as the 4 additional metrics- speed, throttle, reverse, and steering angle. The camera views are shown below in Fig. 2.

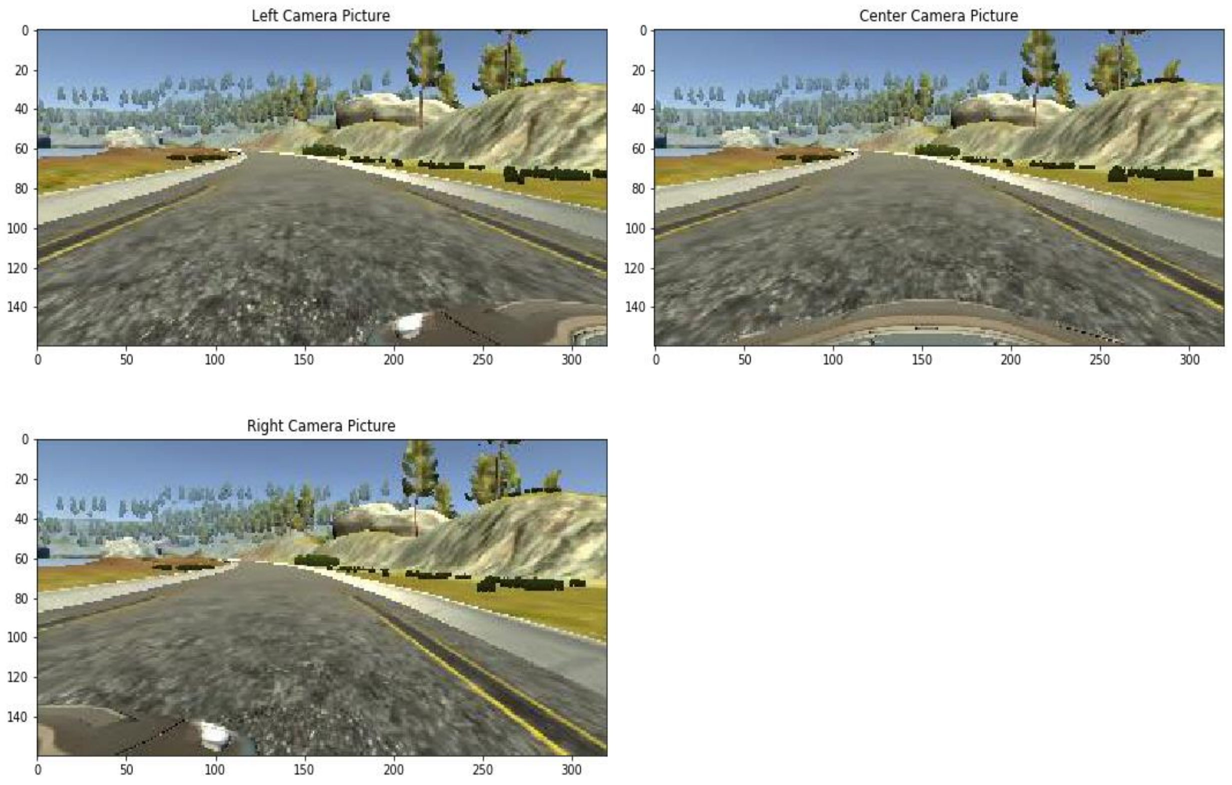

Figure 2. Image captures from the car's cameras. The car has a left, center, and right camera installed on its windshield.

Again, I am only concerned with the steering angle so I perform an initial visual analysis to understand the data distribution shown in Fig. 3.

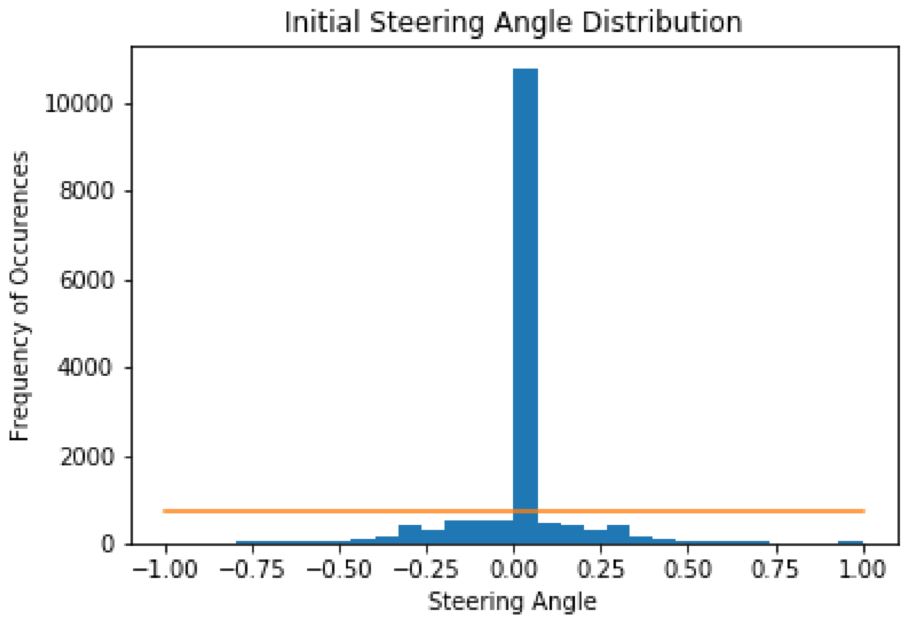

Figure 3. Distribution of the steering angles as recorded by the training laps.

From this steering angle histogram, we can clearly see an overwhelming small angle bias. This immediately raises a red flag. While the lake track can be inferred to be heavily straight from the data and therefore only need small steering angles, there are a few turns which necessitate steering angles of magnitude .25 and greater. This can also be seen from the distribution. If I let the model train on this distribution alone, it will most certainly overfit on small angles and not be able to handle the sharper turns. I therefore define a threshold defined by the orange bar. I define a function to iterate through the bins of the data and keep a *maximum* of 750 data points *per* bin. Importantly, I shuffle the data in each bin such that the whole track still remains represented and described by the steering angle data. This result of this function then cuts the small angle bin to a more reasonable proportion of data, while still representing the whole track. The effect of this function is shown below in Fig. 4.

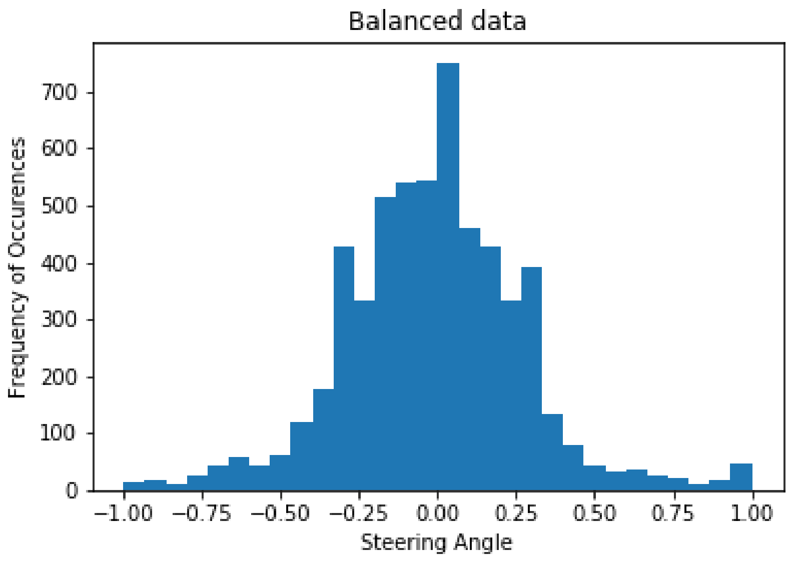

Figure 4. The distribution of steering angles after applying a cutoff function to achieve a more balanced distribution.

After cutting the data by the defined threshold, the resulting size is 5745 observations. The final step in this process then, is to split the data into training and validation data. This is done using the Sci-kit learn library. I initialize the train-test split as 80-20 (Fig. 5), meaning I allow my model to train on 80% of the data, and then validate on the remaining 20%. In plain numbers, this means 4596 observations to train on, and 1149 observations to test on.

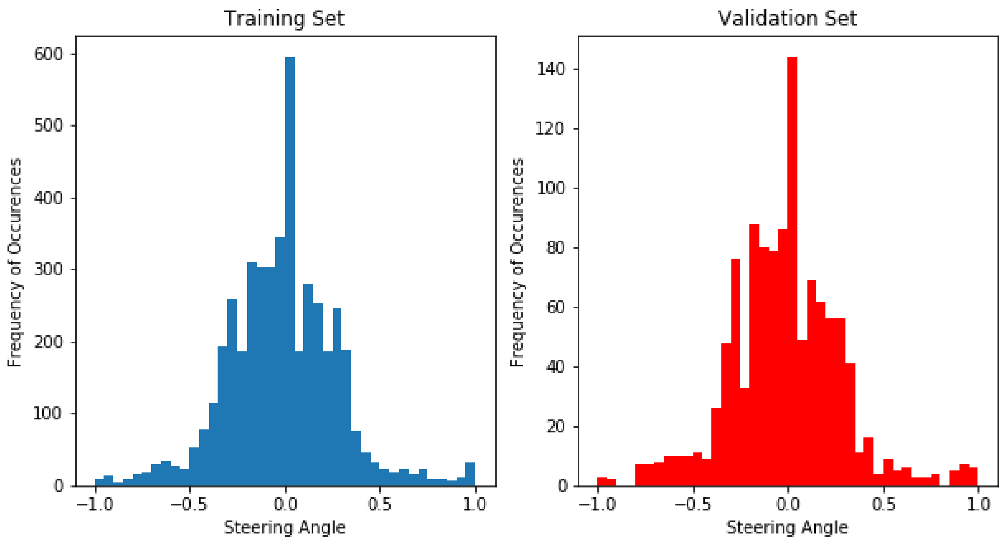

Figure 5. Distribution of steering angle data split into the training and validation sets

Image Processing

The next important step in building the model is preprocessing the images. This step is arguably the most important step in training an ADS convolutional neural network. As previously mentioned, I am using the NVIDIA CNN. This model is optimized under a couple of preprocessing parameters, that I do not stray from in this project. I convert the image to a YUV color scheme from the traditional RGB and resize the image to be in dimensions 200 pixels by 66 pixels. These are the only two parameters that I implement specific to this model. Additionally, I crop the image to include the most relevant data to an ADS and apply a gaussian blur across the image to smooth out some picture details. This processing is applied to all images and the result can be seen below in Fig. 6.

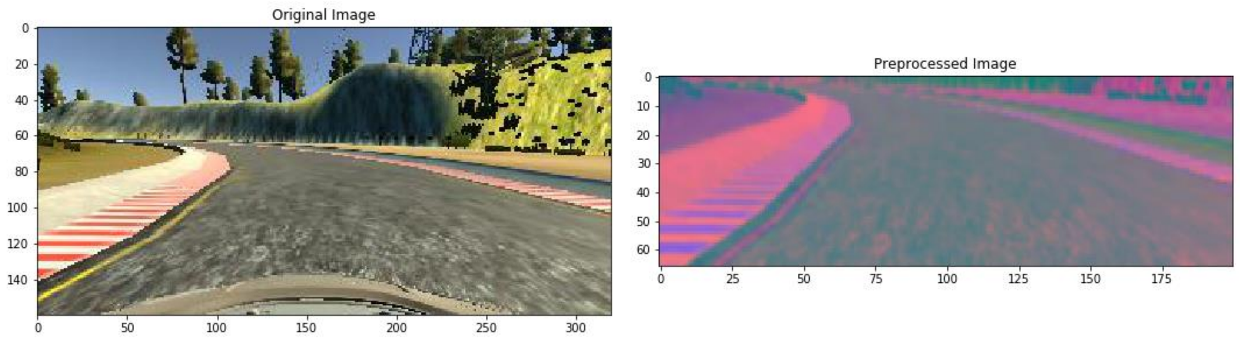

Figure 6. Example of a preprocessed image. Left color scheme is standard RGB, while the Right color scheme is YUV.

While this processing is highly impactful on model performance, it turns out to not be quite enough for a successful model. I trained an NVIDIA model using images processed only to this extent and it failed quite early on the lake track- a requirement I initially set for this model. Due to the importance of image processing on ADS performance, I decided to take this phase one step fruther. The remainder of the image processing phase consisted of certain augmentation techniques that would allow the model to learn from dynamic images rather than static ones. I used a library called imagaug to handle the augmentation process. The techniques I used were as follows: zoom, pan, sharpen, darken, brighten, and flip. I defined a separate function for each augmentaion that would appropriately augment each image. Furthermore, some augmentaions such as darken and brighten augment the image in a numeric interval, rather than a set value. Examples of each image augmentation are presented bellow in Figs. 7-12.

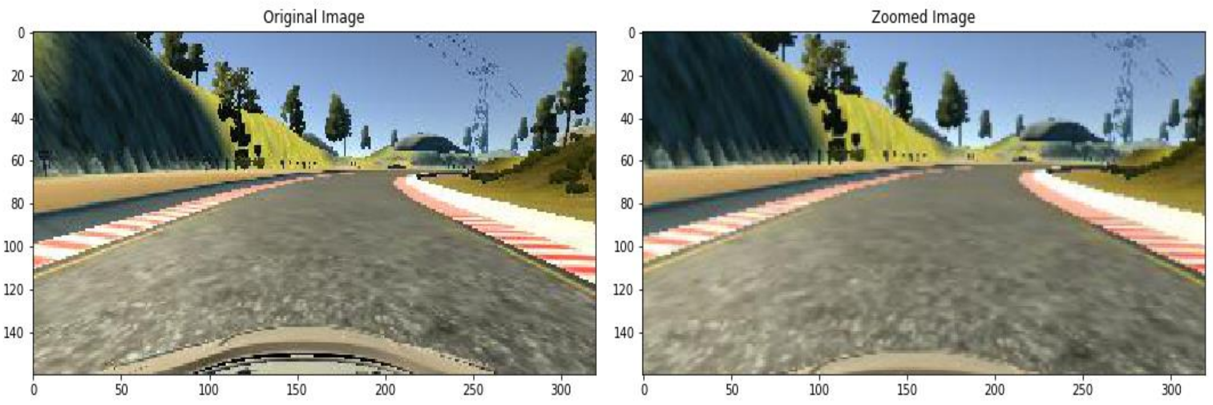

Figure 7. Effect of a Zoom augmentation

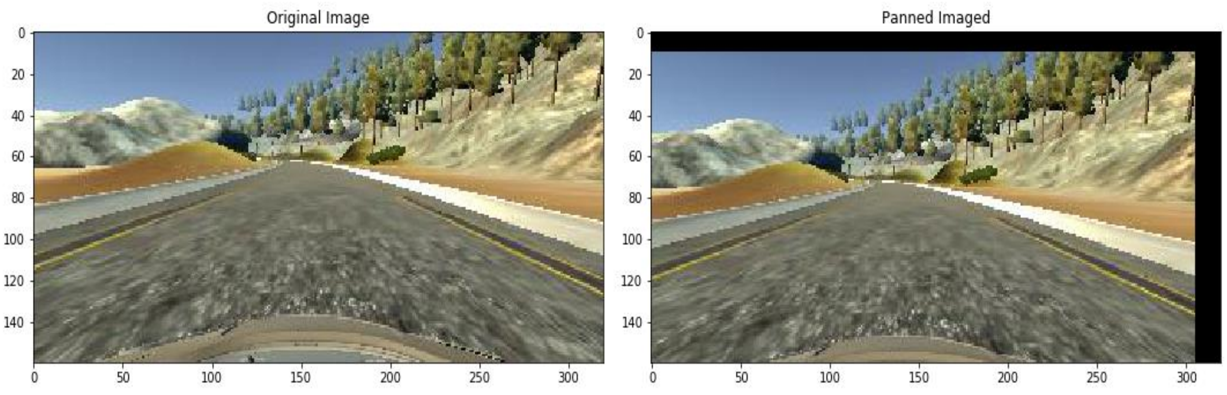

Figure 8. Effect of a Panned augmentation

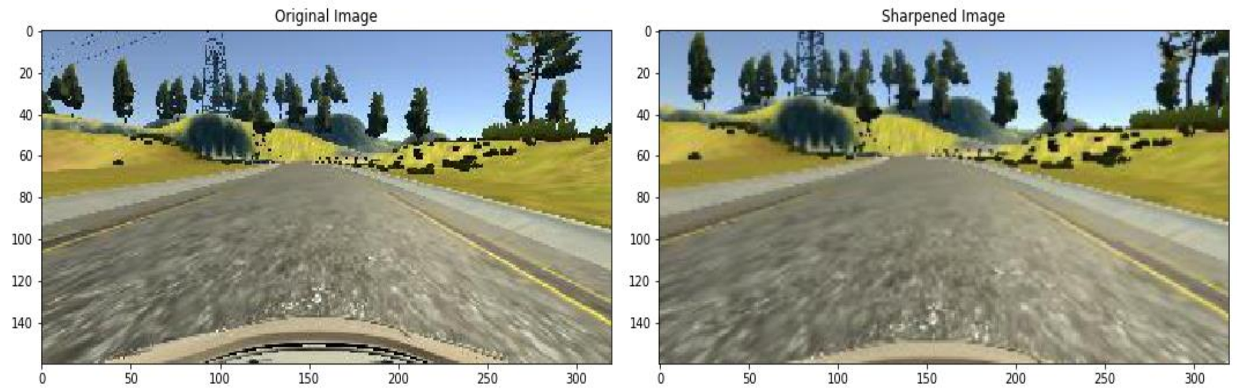

Figure 9 Effect of a Sharpened augmentation

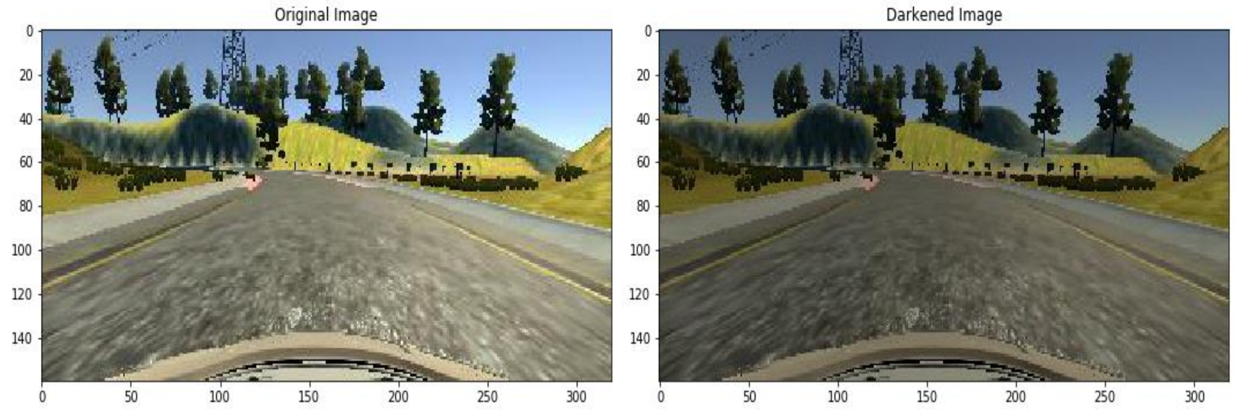

Figure 10. Effect of a Darkened augmentation

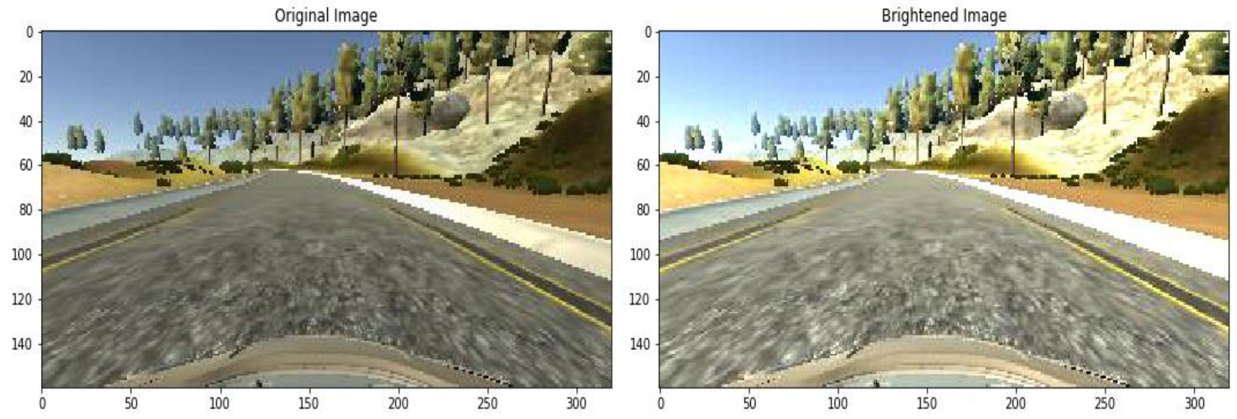

Figure 11. Effect of a Brightened augmentation

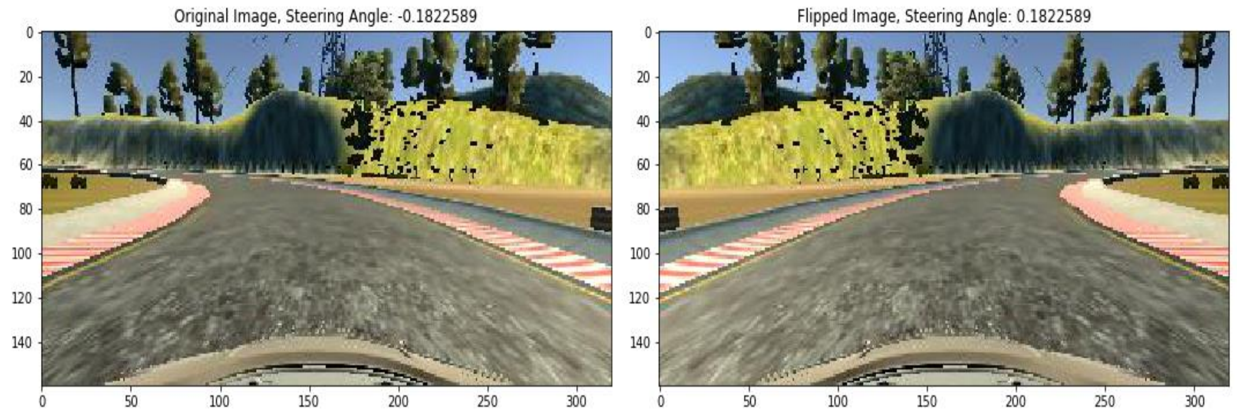
Original Image, Steering Angle: -0.1822589
Flipped Image, Steering Angle: 0.1822589

Figure 12. Effect of a Flipped augmentation

After defining the augmentation techniques, I constructed a function that would randomly apply each augmentation to an image 50% of the time. Therefore the probability that an image would have every augmentation applied to is is 1.5%. An example of an image run through this function is shown below in Fig. 13.

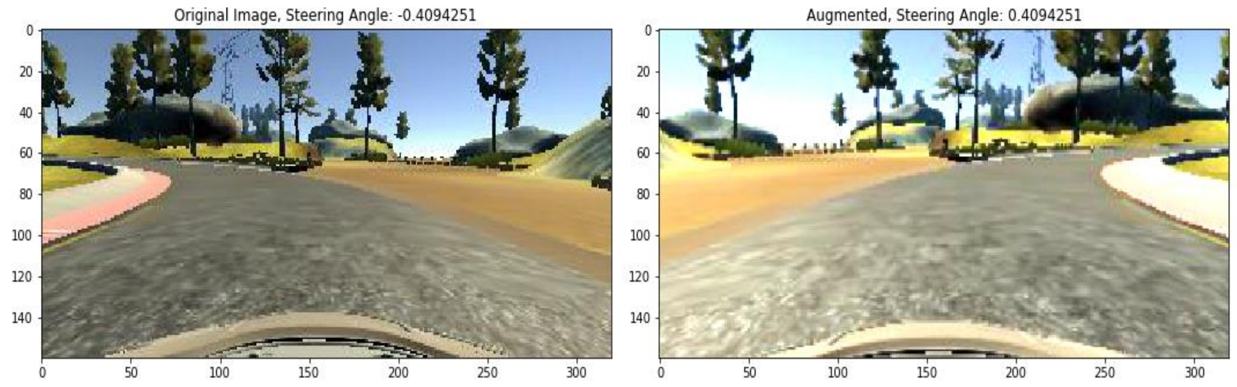
Original Image, Steering Angle: -0.4094251
Augmented, Steering Angle: 0.4094251

Figure 13. Effect of a random compilation of the above augmentations applied to an image.

Model Architecture

As mentioned before, I use the NVIDIA deep learning model to train the self-driving car. The NVIDIA model has been optimized to perform very well in training an ADS and its general architecture can be seen to the left. To tailor the model to my task, I specify the exponential linear unit activation function as well as the mean-squared-error loss function. I implement the construction of this model using the Keras library. The complete model architecture along with the model summary (which shows an overview of the data as its processed through the various layers) can be found in the jupyter notebook.

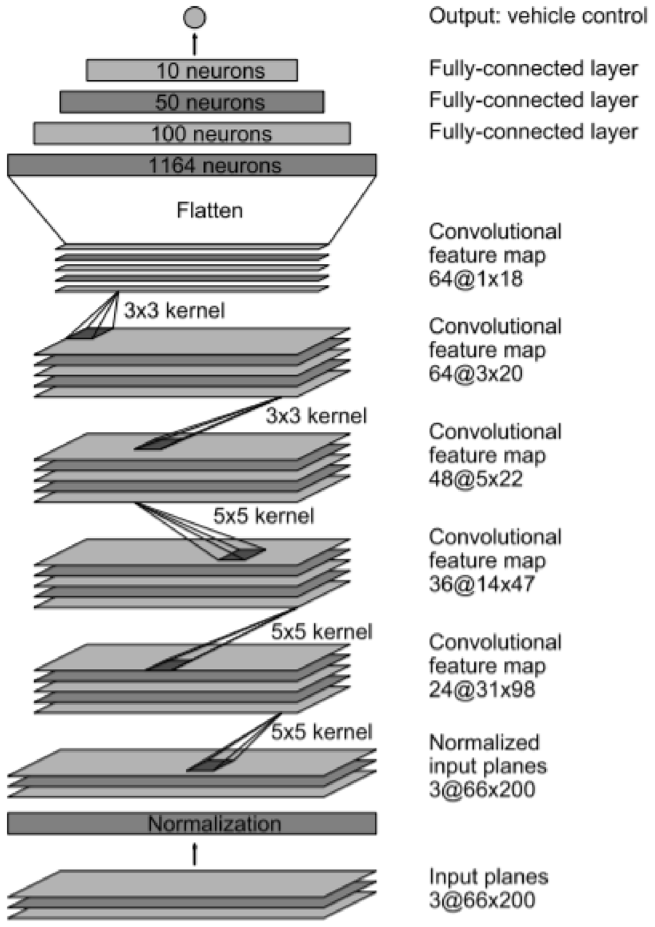

Figure 14. NVIDIA ADS architecture used in my Keras model

Because I am augmenting my images, I elect to use Keras' fit generator function to train my model. This function allows a batch generator (in my case an augmented image generator) to be implemented inside of a fitting function. The advantage of this method is to save on memory costs and computation time. The batch generator generates an augmented image for the fit generator to train on immediately. I initialize the final model to iterate through 10 epochs with a training batch size of 100 and number of training steps per epoch at 400 and a validation batch size of 100 and number of training steps per epoch at 200.

To evaluate the model, I use two metrics. The first is by examining the loss function on both the training and validation data sets. The loss function represents the model's mean squared error in its predicted steering angles. A lower loss score corresponds to a better performing model. The main outcomes I look to avoid are overfitting and underfitting. Both of these challenges can be evaluated by inspecting loss plot of both data sets. A validation loss above the training loss indicates overfitting, while a validation loss significantly below the training loss indicates underfitting. An ideal comparison between training loss and validation loss has a validation loss consistently below (but not too far below) the training loss. The plot below shows the training and validation losses .

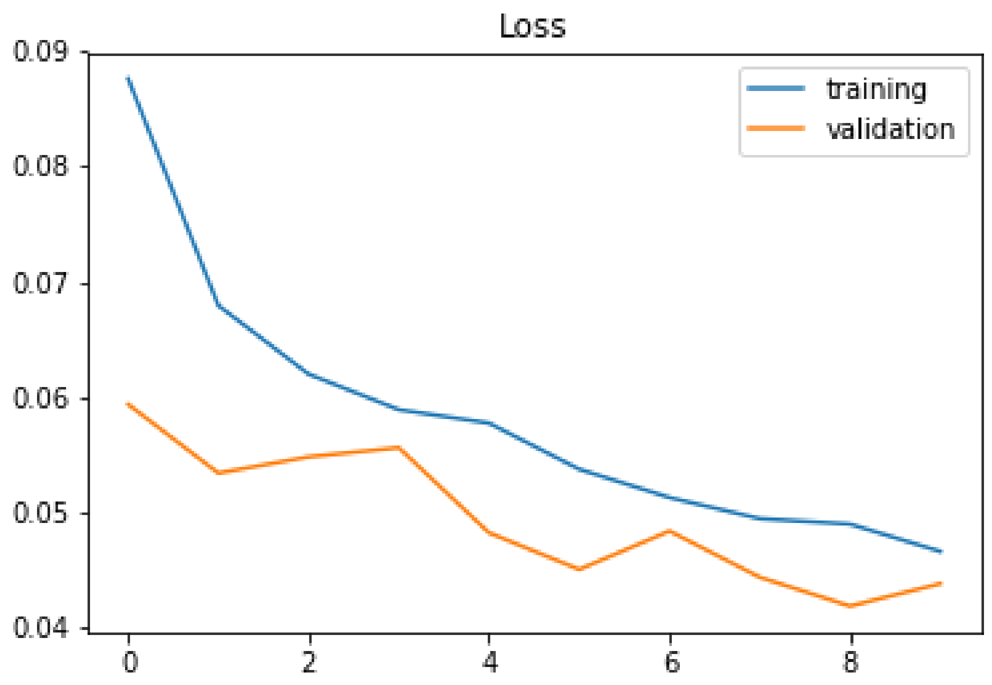

Figure 15. Loss function of training and validation set

Evaluating Model Performance

The final training loss is .0466 with a corresponding validation loss of .0438. Moreover The plot of training loss and validation loss certainly indicates that my model has not overfit or underfit, but instead learned to predict a steering angle based upon camera images very well and generally. However, my final metric of evaluating model performance is much more concrete. I put my model to the test on the simulated track to fully evaluate its capabilities. To connect my model to the simulator, I adapt a Flask and Socketio applet to my specifications. This applet interface allows the saved model as constructed in Keras to connect to the simulator and execute.

After tuning the NVIDIA model and otpimizing the loss plot, my finalmodel is able to complete lap after lap on the lake track. However, I had seen this result in previous builds of the model and the true test was how well it did on the mountain track- a track it had never seen before. Up until my final model, the self-driving car was at most able to navigate through 30% of the track. The final, optimized model was able to navigate through 80% of the track before failing on a rather steep turn. A video of the simulations on each track can be found in the github repository under *'lake_track_sim.mp4'* and *'mountain_track_sim.mp4'*.

To extend this project, further optimization would need to be performed on the model. As it stands, it is able to complete lap after lap on the training track. However, there is a bit of swaying and straying from the center of the lane. I believe that with more augmented training data, this can be corrected. Furthermore, more augmented training data will allow the model to generalize to more and more unique tracks. To completely round out this ADS model, I would take all of the data collected from the training- left camera, right camera, speed, throttle, and reverse- and build a model that learns from them. This would truly be the most robust model and certainly the most interesting extension to this project.