Behavioral Cloning

My experiments in the Udacity Self-Driving Car Engineer Nanodegree Behavioral Cloning project.

Writeup

Behavioral Cloning Project

The goals / steps of this project are the following:

- Use the simulator to collect data of good driving behavior
- Build, a convolution neural network in Keras that predicts steering angles from images
- Train and validate the model with a training and validation set
- Test that the model successfully drives around track one without leaving the road
- Summarize the results with a written report

Rubric Points.Here I will consider the rubric points individually and describe how I addressed each point in my implementation.

Files Submitted & Code Quality

1. Submission includes all required files and can be used to run the simulator in autonomous mode. My project includes the following files:

- model.py containing the script to create and train the model
- drive.py for driving the car in autonomous mode
- model.h5 containing a trained convolution neural network
- writeup_report.md or writeup_report.pdf summarizing the results.

2. Submission includes functional code

Using the Udacity provided simulator and my drive.py file, the car can be driven autonomously around the track by executing

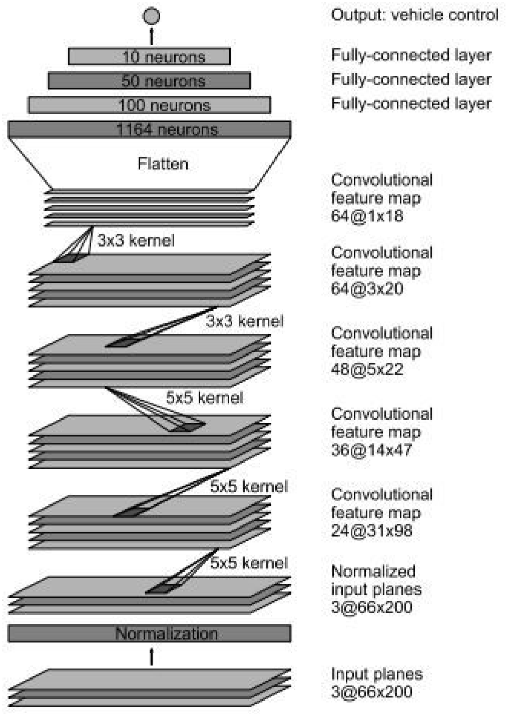
```
python drive.py model.h5
```

3. Submission code is usable and readable

The model.py file contains the code for training and saving the convolution neural network. The file shows the pipeline I used for training and validating the model, and it contains comments to explain how the code works.

Model Architecture and Training Strategy

1. An appropriate model architecture has been employed



The model was completely trained on a AWS instance (g-2.2x large)

I started out with a simple convolutional network. The performance was just ok, except the car drifted a hell lot of times and ended up inside the lake, and was not able to stay in the lane. Next, I modified the model to the be based on the Nvidia Model (https://devblogs.nvidia.com/parallelforall/deep-learning-self-driving-cars/)

Cropping the image to had significant performance difference in how the model performed. I had to ensure that sufficient information about the surroundings was captured while leaving out unnecessary details about the environment. Cropping out the top 60 pixels and the bottom 25 pixels seemed to capture the amount of information worth for the CovNet to learn the road orientation and the respective steering angle, and the model performance improved as a result.

I tried logging the loss function using the Keras Callbacks. I also attempted to log the model data using TensorBoard Callback function.I implemented Early Stopping , but still did not implement it in the callback as the model would gradually settle in a minimal loss function. I also used the Batch_Generator() which seemed very useful in creating a batch to train from the humongous data set.

I split my image and steering angle data into a training and validation set. The final step was to run the simulator to see how well the car was driving around track one.I used the train_test_split() function to split the data. . Any improvements that were made to the model resulted in a better performance around the track.Initially the Ensuring the car stayed on the road in places with no lane markers was especially challenging.

The model includes ELU to introduce nonlinearity, and the data is normalized in the model using a Keras lambda layer.

At the very end of the process, the vehicle was driving autonomously around the track without leaving the road. The attached video contains a successful run.

2. Final Model Architecture

Checklist for creating the pipeline of the NVIDIA model : https://tinyurl.com/y88xelpb¶

1. Preprocess incoming data, centered around zero with small standard deviation
2. Trim image the to train the network on the basis of the road and the steering angle only.
3. Layer 1 : Convolution Layer, Number of Filters : 24, Filter size= 5x5, stride= 2x2.

4. Layer 2 : Convolution Layer, Number of Filters : 36, Filter Size = 5x5, stride = 2x2.
5. Layer 3 : Convolution Layer, Number of Filters : 48, Filter Size = 5x5, stride = 2x2.
6. Layer 4 : Convolution Layer, Number of Filters : 64, Filter Size = 3x3, stride = 1x1.
7. Layer 5 : Convolution Layer, Number of Filters : 64, Filter Size = 3x3, stride = 1x1.
8. Flatten the Image from a 2D format to a row based array.
9. Layer 6 : Fully Connected Layer
10. A dropout layer to avoid overfitting.
11. Layer 7 : Fully Connected Layer
12. Layer 8 : Fully Connected Layer

3. Attempts to reduce overfitting in the model

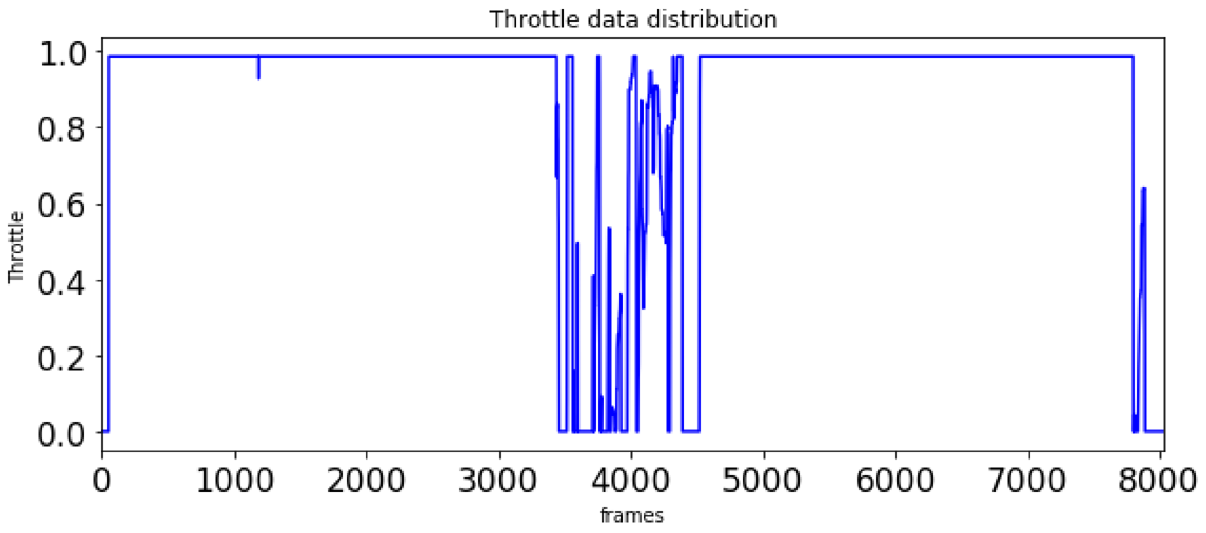
The model was trained and validated on the Udacity provided data sets to ensure that the model was not overfitting . The model was tested by running it through the simulator by using the Drive.py and ensuring that the vehicle could stay on the track.

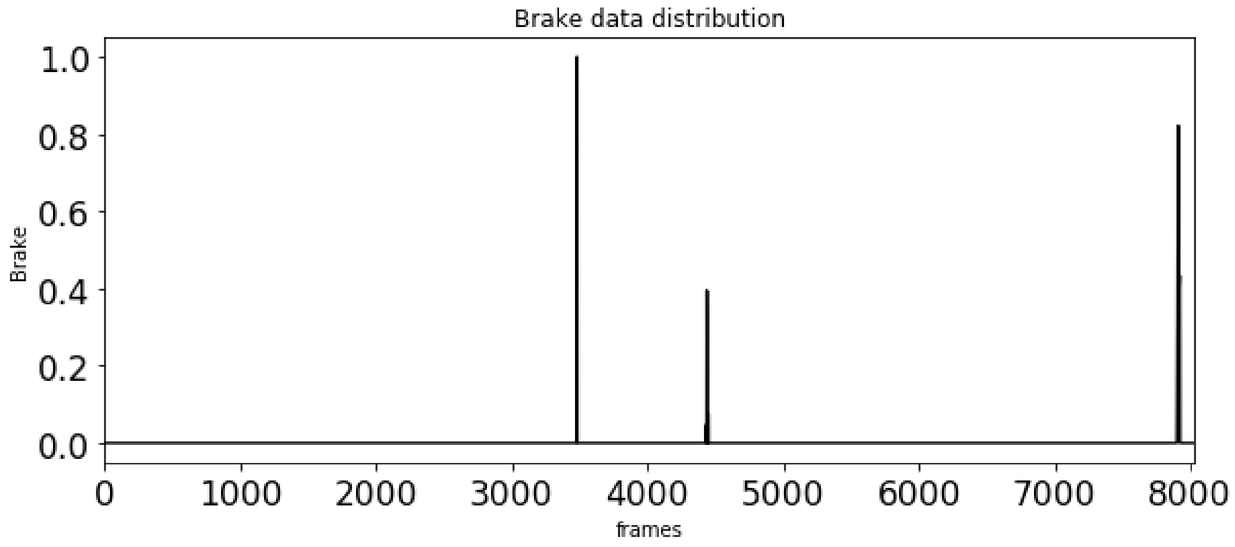
4. Model parameter tuning

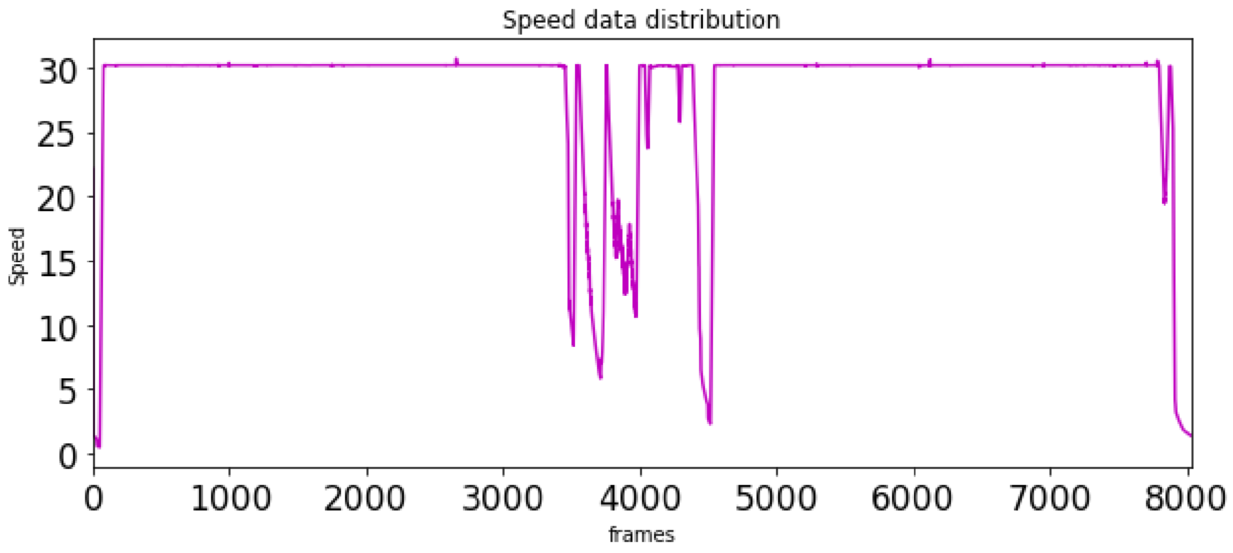
The model used an adam optimizer.

5. Appropriate training data

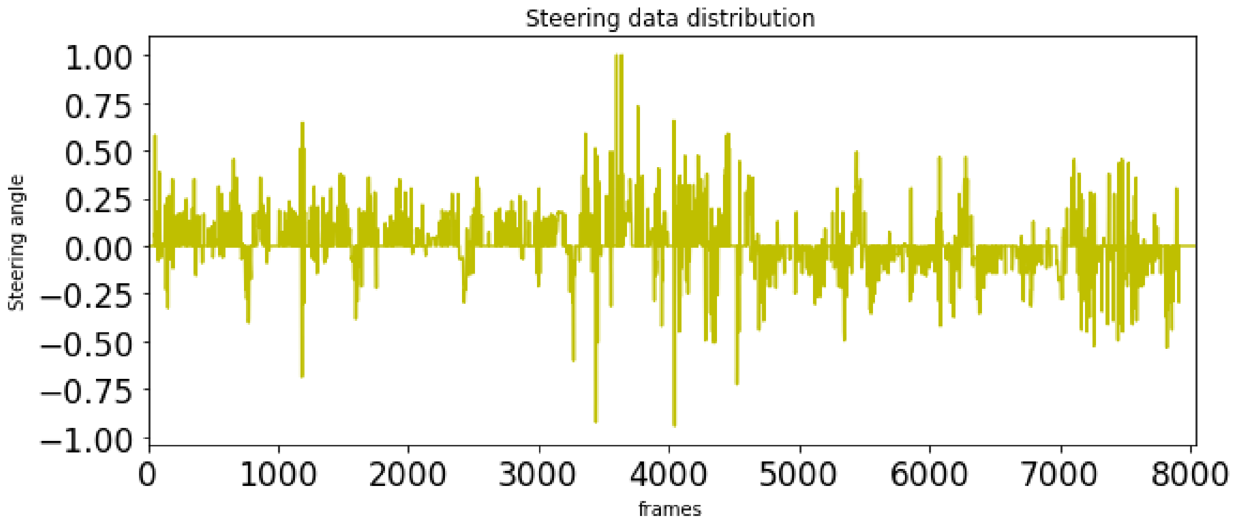
Training data was chosen to keep the vehicle driving on the road. My training set consisted of variations in throttle at different timestep. The Importance of capturing good data , preprocessing the data in this project taught me how important Data is in Machine Learning and Deep learning. The evident fact was that the Convolution Layer could only predict to a certain level at which it was trained. Though this is a inefficiency, Convolution Networks are really helpful in classification and regression problems.

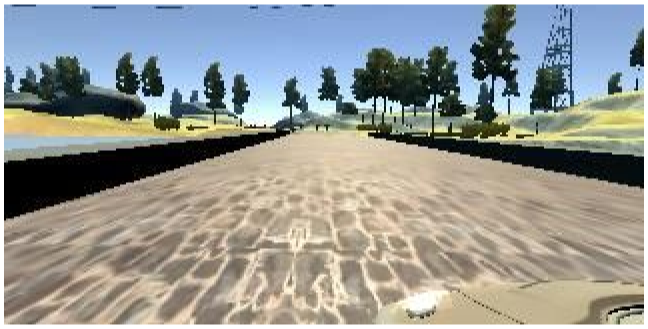
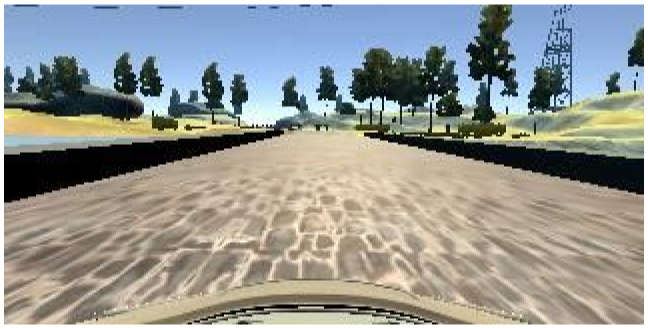
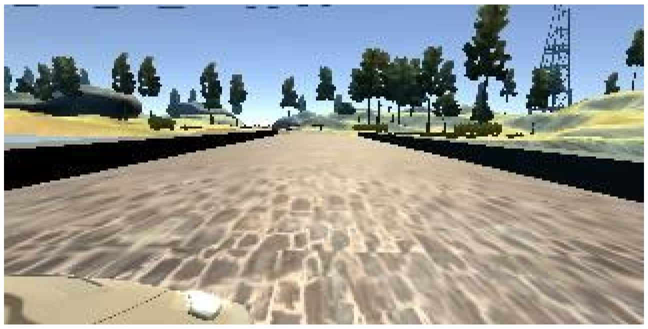

Throttle data distribution
Throttle
frames
0.0
0.2
0.4
0.6
0.8
1.0
0
1000
2000
3000
4000
5000
6000
7000
8000


Brake data distribution
Brake
frames
0.0
0.2
0.4
0.6
0.8
1.0
0
1000
2000
3000
4000
5000
6000
7000
8000

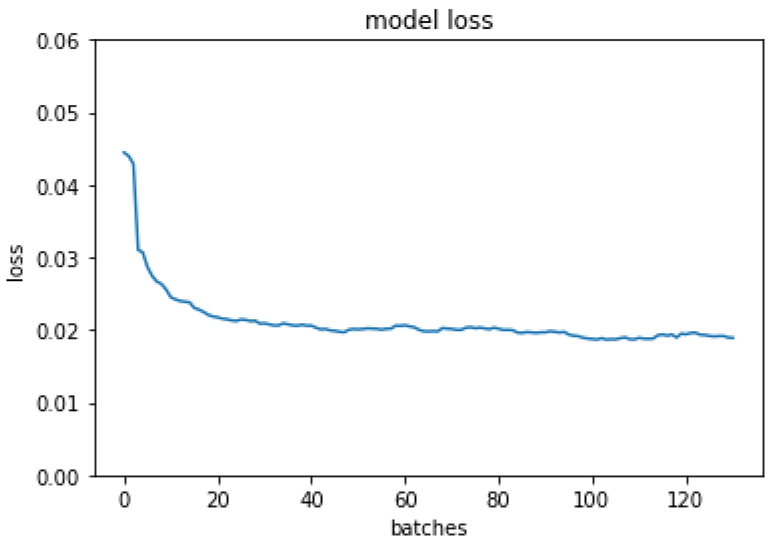


The Training data had a bias towards the left turns as the track had more number of left turns than the right turns. A compensation of 0.2 was added and subtracted to maintain a optimal level of steering angles for training the data.



The most challenging parts were ensuring that the car crashed on the edge of the bridge at different parts. I had added a random brightness augmentation to generalize the images, instead it had memorized the data and goes crazy on the bridge due to the lake.



.

I randomly shuffled the data set and put 2% of the data into a validation set.

I used this training data for training the model. The validation set helped determine if the model was over or under fitting. The ideal number of epochs was 5 based on the train and validation losses. I used an adam optimizer.

Places to improve:

- Use a pretrained model with minimal hyperparameters , to efficiently navigate the car autonomously. Some pretrained nets that could be helpful are :
    - SqueezeNet
    - AlexNet
    - VGGNet

- Image Augmentation could help the model learn what shadows and sunlight is as the model would also understand the dynamic changes in brightness while driving on the bridge and driving at the curves of the first track. This would help in more generalizing the Data.