Behavrioal 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.pdf summarizing the results
- run1.mp4 for the recorded video of the car driving in autonomous mode.

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 network consists of 9 layer including a normalization later, 5 convolutional layers and 4 fully connected layers. (model.py lines 73-95)

As well, the data is normalized in the model using a Keras lambda layer (code line 76) and data cropping is also performed on the model (code line 78).

2. Attempts to reduce overfitting in the model

The model contains dropout layers at the end of the first two fully connected layers order to reduce overfitting (model.py lines 86 & 88).

The model was trained and validated on different data sets to ensure that the model was not overfitting (code line 94). The model was tested by running it through the simulator and ensuring that the vehicle could stay on the track.

3. Model parameter tuning

The model used an adam optimizer, so the learning rate was not tuned manually (model.py line 92).

4. Appropriate training data

Training data was chosen to keep the vehicle driving on the road. I used a combination of center lane driving, left and right sides of the road and as well as further data augmentation steps like flipping the images.

For details about how I created the training data, see the next section.

Model Architecture and Training Strategy

1. Solution Design Approach

I took hints of a model architecture from several famous network models like comma.ai and Nvidia's End to End Learning model.

My first step was to use a convolution neural network model similar to the Nvidia End to End Learning For Self Driving Cars Architecture. I thought this model might be appropriate because it was a proven model on a much tougher track than the track in the Udacity simulator so I figured it is a good starting point.

In order to gauge how well the model was working, I split my image and steering angle data into a training and validation set. After training the model on AWS and testing it on the simulator, I noticed the car had trouble navigating past the bridge. So, I added a dropout layer at the end of the 1st two fully connected layers as it might be a case of the model overfitting at certain corners.

The final step was to run the simulator to see how well the car was driving around track one. There were few spots where the vehicle fell off the track especially after coming off the bridge and turning left. To improve the driving behavior in these cases, I tweaked the value of the correction factor in the data augmentation steps.

At the end of the process, the vehicle is able to drive autonomously around the track without leaving the road.

2. Final Model Architecture

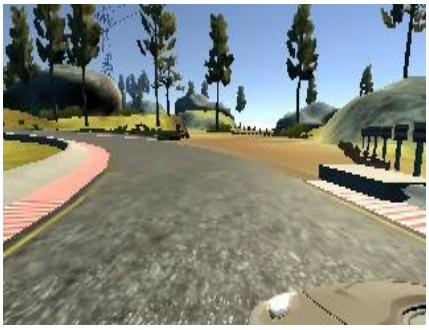
The final model architecture (model.py lines 73-92) is visualized as below:

Input (160x320x3) -> Normalization Layer -> Cropping Layer -> Convolutional -> ReLu -> MaxPool -> Convolutional -> ReLu -> MaxPool -> Convolutional -> ReLu -> MaxPool -> Convolutional -> ReLu -> MaxPool -> Convolutional -> ReLu -> MaxPool -> Flatten -> Fully Connected (100 units) -> Regularization (Dropout)-> Fully Connected (50 units) -> Regularization (Dropout)-> Fully Connected (10 units) -> Fully Connected (1 unit, output).

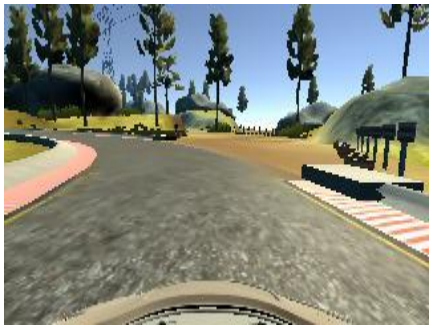
3. Creation of the Training Set & Training Process

Initially, I was intending to record my own driving data but after checking around the forum, I found that Udacity's dataset was good enough to run the project. Below is a sample image of the centre,left and right sides camera :

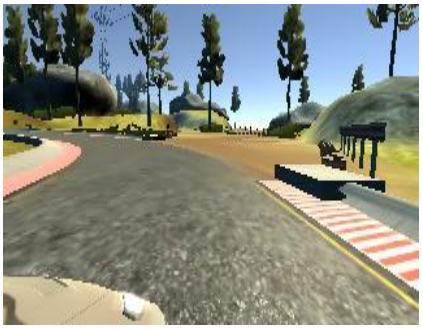
Left Image:



Centre image:



Right Image:



After running the model using only centre image, the car only managed to drive for a short while before crashing to the side. So, I added the left and right image as well to the dataset and compliment it with the correction factor of the steering angle. The correct correction factor was a tricky problem but after reading Vivek Yadav's blog on data augmentation, I proceed to use the correction factor of 0.25.

To augment the data further, I also created a condition where the car would be driving clockwise instead of the default counter clockwise. I did this by flipping the images and angles. As well, as the inclusion of flipped images doubled the size of images from the previous step, it may cause the model to read all data in memory thus will cause error of "Resource Exhaustion Error". To overcome this, all the training data are fed through a python generator which then feeds data to the model in pre-defined batch sizes. The code line 94 shows the usage of keras's "model.fit_generator" to take in batches of images which was set in the python generator.

In the model, I finally randomly shuffled the data set and put 20% 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 is 25 as the validation loss and loss plateaued around 20 to 25 epochs in training. I used an adam optimizer so that manually training the learning rate wasn't necessary.