Behavioral Cloning Project Writeup

Goals :

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](https://review.udacity.com/#!/rubrics/432/view) 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
    - DataProvider.py a module that providing formated, processed and augmented data to the model and drive programs
    - 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. The simulator request predictions from drive.py by prodiving it for each request an image and wait for a steering angle prediction from the model parameter build and computed before. The drive.py program submit to the simulator the model predictions using the model parameters.

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. The model.py Contains a BehavioralCloningModel class with four methods :

    a. Constructor method : Calls an interfnal method to buid an nvidia-like model, summarize it and compile it using Adam optimizer and a learning rate of .0001

    b. Execute method : Calls the DataProvider module's generator which yields a batch of data each time it is requested by the caller program. Note that generator is running in the background thread indefinitly thanks to the while(1) loop. This method is Model Trainer. It executes the model training each time the generator send it back a train and validation set.

c. Save method : Once the training is done, it saves the resulting weights to model.h5 format and to Json format

d. Data visualizer : this section of the program visualize all the train and validation loss. It also visualizes the history graphic of the train loss vs validation loss.

Model Architecture and Training Strategy

1. **An appropriate model architecture** has been employed. I use an nvidia-like architecture:

#Normalization(64,64,3)==>Conv(3,5,5,24)==>Relu==>Conv(24,5,5,36)=>relu==>Pool(2,2)==>Conv(36,5,5,48)==>Relu==>Pool(2,2)#==>Conv(48,3,3,64)==>Relu==>Pool(2,2)==>Conv(64,3,3,64)==>Relu==>Pool(2,2)==>Flatten==>Dense(1164)==>Relu==>Dense(100)==>relu==>Dense(50)==>Relu==>

Dense(1)

The Convolution network is tied for images recognition, the more convolution layers has the model and the more he is able to extract features accurately. the first layers extract basic features, edges and color regions and the next layers extracts shapes

2. Attempts to reduce overfitting in the model

Data selection and quantity : I used a dataset of 24 108 samples. The raw images and angles are randomly processed and augemented. To avoid overfitting and underfitting batches are generated randomly using python generator to avoid running the program out of memory. The model was trained and validated on different data sets to ensure that the model was not overfitting (). 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).

4. . Appropriate training data

Training data was chosen to keep the vehicle driving on the road. I used a combination of :

- three track with center lane driving,
- one track with recovering from the left and right sides of the road ...
- one track with smoothing steering angle during left and right turns

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

Model Architecture and Training Strategy

1. . Solution Design Approach

The overall strategy for deriving a model architecture was to starting from a LeNet like by adding more Conv layers and Dense layers. My first step was to use a convolution neural network model inspired by LeNet I thought this model might be appropriate because conv net are for images recognition. the more layers and the more successful is the model. In order to gauge how well the model was working, I split my image and steering angle data into a training and validation set. I found that my first model had a low mean squared error on the training set but a high mean squared error on the validation set. This implied that the model was overfitting. To combat the overfitting :

- Process each image of the batch by croping, resizing, applying a random (flip, gamma effect, rotation, shear)
- generate batches for epochs randomly
- shuffling samples

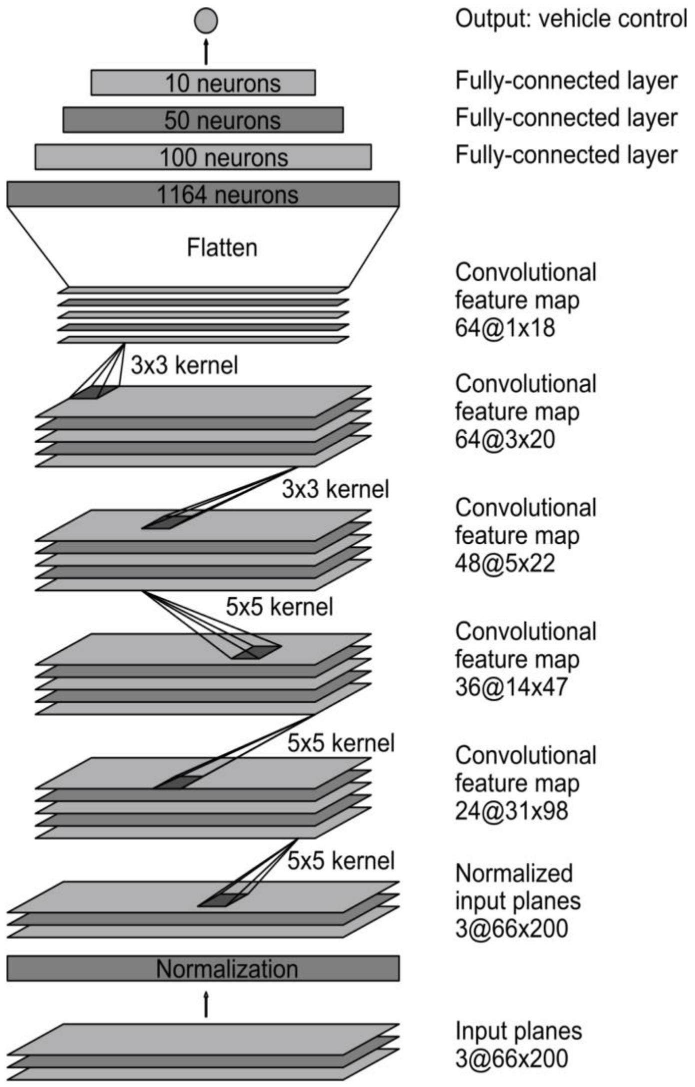
The final step was to run the simulator to see how well the car was driving around track one. There were a few spots where the vehicle fell off the track… to improve the driving behavior in these cases, I added more images to different situation such as recorvery from left to ight and vice-vers-ca. At the end of the process, the vehicle is able to drive autonomously around the track without leaving the road for several tracks

2. Final Model Architecture

I started by trying LeNet-like model with normalization, croping, multi convulotion, pool and dense layers taking a (160,320) input shape. The result was that the simulator fails after crossing the bridge. I tryed afterwards, the commaai-like model with fails from the starting. The final model architecture is a nvidia-like model(model.py lines 18-24) consisted of a convolution neural network with the following layers and layer sizes …

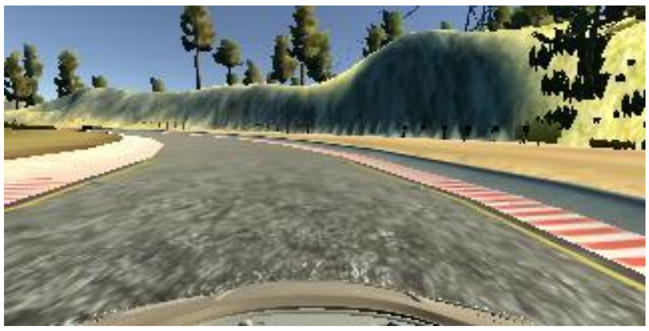
#lambda==>Cropping=>Conv(3,3,3,18)==>Relu==>Conv(3,3,18,36)=>relu==>Pool(3,3)==>Conv(3,3,36,72)==>Relu==>Pool(5,5)#==>Conv(3,3,72,144)==>Relu==>Flatten==>Dense(16)==>Relu==>Dense(16)==>relu==>Dense(16)==>Relu==>Dense(1)

Here is a visualization of the architecture (note: visualizing the architecture is optional according to the project rubric)

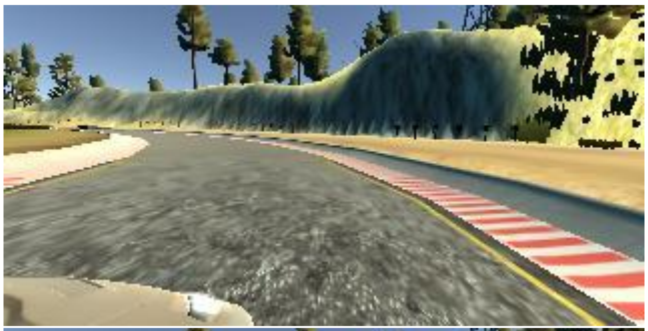
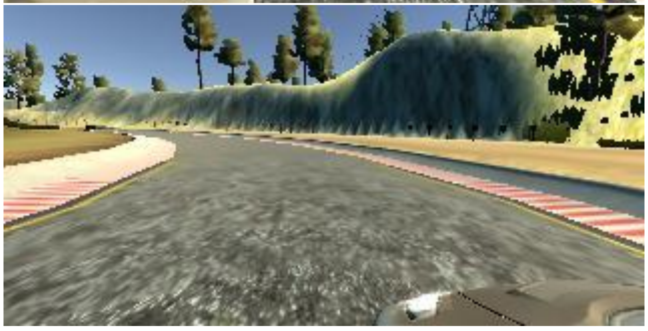


3. Creation of the Training Set & Training Process

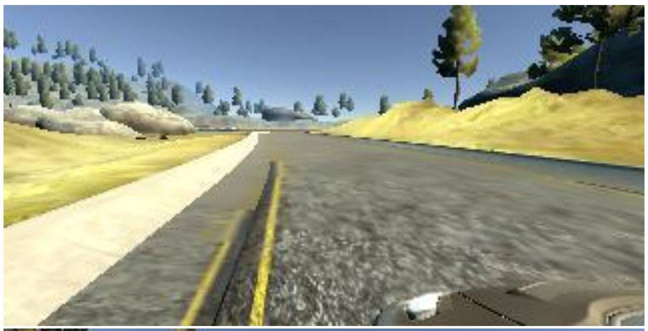
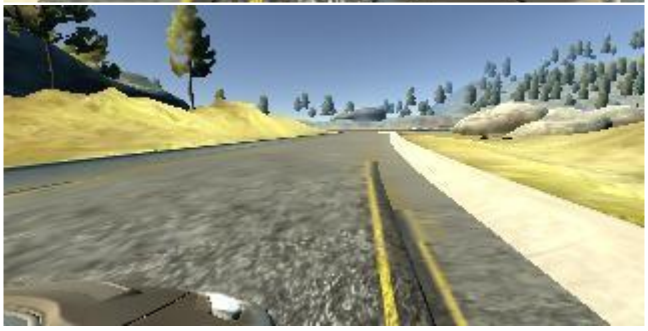
To capture good driving behavior, I first recorded two laps on track one using center lane driving. Here is an example image of center lane driving:



I then recorded the vehicle recovering from the left side and right sides of the road back to center so that the vehicle would learn to recover the steering angle to from left or right to center of the lane. These images show what a recovery looks like starting from ... :

Then I repeated this process on track two in order to get more data points. To augment the data sat, I also flipped images and angles thinking that this would learn the model the opposite situation of the one it occurs For example, here is an image that has then been flipped:

After the collection process, I had the double number of data points. I then preprocessed this data by normalizing each image as follow : That lambda layer could take each pixel in an image and run it through the formulas:

pixel_normalized = pixel / 127.5

pixel_mean_centered = pixel_normalized - 0.5

using a Lambda layer : Lambda(lambda x: (x / 127.5) - 1)

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 was 10 as evidenced by the number of trials. I used an adam optimizer so that manually training the learning rate wasn't necessary.