Behavioral Cloning

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 (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
- 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

Following along with the lessons, three model architectures were made; a basic (what I called 'placeholder') model architecture (method name model_architecture_placeholder() line 467 in model.py), based on LeNet5 (method name model_architecture_lenet() line 491 in model.py), and based on NVIDIA End to End Learning for Self-Driving Cars (method name model_architecture_nvidia() line 516 in model.py).

The final model used was based on NVIDIA's End to End Learning for Self-Driving Cars. The paper can be found here http://images.nvidia.com/content/tegra/automotive/images/2016/solutions/pdf/end-to-end-dl-using-px.pdf (http://images.nvidia.com/content/tegra/automotive/images/2016/solutions/pdf/end-to-end-dl-using-px.pdf).

The model consists of 5 convolutional layers of:

- filter 24, kernel_size 5, strides 5
- filter 36, kernel_size 5, strides 5
- filter 48, kernel_size 5, strides 5
- filter 64, kernel_size 3, strides 3
- filter 64, kernel_size 3, strides 3

Followed by 4 fully connected layers of

- 100 nodes
- 50 nodes
- 10 nodes

and final output

- 1 node

The final node is a float, so it is a regression model and not a classification model. It represents the angle to steer the car. 0 is center, negative and positive turns the car left and right.

The model includes RELU layers to introduce nonlinearity.

2. Attempts to reduce overfitting in the model

The model was trained and validated on different data sets to ensure that the model was not overfitting (code line 581-592). The model was tested by running it through the simulator and ensuring that the vehicle could stay on the track.

I attempted dropout in some model architectures (you can view past model achitecture attempts in the 'my_code' folder with timestamps.) In the end, I did not use dropout. I did pre-process the data with normalization, mean center around 0, and crop out irrelevant pixels.

3. Model parameter tuning

The model used an adam optimizer, so the learning rate was not tuned manually (model.py line 604-605).

4. Appropriate training data

I actually collected the data suggested from Project: Behavioral Cloning - 16. More Data Collection

- two or three laps of center lane driving
- one lap of recovery driving from the sides
- one lap focusing on driving smoothly around curves

I also drove the car the other direction on the first track and collected data from driving in the center of the road in the second track.

Unfortunately, I got terrible results. In the end, I used the data provided. Without changing the model architecture, I got satisfactory results on autonomous mode on the first track. I did try the model on track 2 'video_track2.mp4'. It stayed on the road until it met a really sharp turn, then it failed. Also, the car crossed the center line. The track 1 data did not contain examples of how to deal with these cases so it is not surprising that it failed. More data from track 2 would have to be collected and the network would have to be retrained. Also, braking may be needed as and output parameter to be controlled to get around sharp turns.

Training data was chosen to keep the vehicle driving on the road. I used the center, left, and right camera. I used the suggested hyper-parameter tuning of the manual correction of 0.2 for the left and right camera images.

Architecture and Training Documentation

1. Solution Design Approach

The overall strategy for deriving a model architecture was to ...

My first step was to use a convolution neural network model similar to the NVIDIA's End to End Learning for Self-Driving Cars. The paper can be found here http://images.nvidia.com/content/tegra/automotive/images/2016/solutions/pdf/end-to-end-dl-using-px.pdf (http://images.nvidia.com/content/tegra/automotive/images/2016/solutions/pdf/end-to-end-dl-using-px.pdf). I thought this model might be appropriate because it was designed and implemented to work on a real self driving car.

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 architectural model (placeholder basic neural network) had a high mean squared error on the training set but a high mean squared error on the validation set. This implied that the model was underfitting. The second architectural model was based on LeNet. It seemed more promising but again it had the same issue; a high mean squared error on the training set but a high mean squared error on the validation set. This implied that the model was underfitting.

Then I used the NVIDIA model with the hyper-parameters suggested by the lecture and it worked. Took 3 epochs to train and about 2 hours on GPU.

You can see my other model architectural attempts in the 'my_code' folder. I tried many datasets that I generated. My data was probably not the most ideal, it is hard to control the car, especially through VNC in a webbrowser where there would be some frame drops and freeze frames.

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_architecture_nvidia() method model.py line 516) consisted of a convolution neural network with the following layers and layer sizes

The final model used was based on NVIDIA's End to End Learning for Self-Driving Cars. The paper can be found here http://images.nvidia.com/content/tegra/automotive/images/2016/solutions/pdf/end-to-end-dl-using-px.pdf (http://images.nvidia.com/content/tegra/automotive/images/2016/solutions/pdf/end-to-end-dl-using-px.pdf).

The model consists of 5 convolutional layers of:

- filter 24, kernel_size 5, strides 5
- filter 36, kernel_size 5, strides 5
- filter 48, kernel_size 5, strides 5
- filter 64, kernel_size 3, strides 3
- filter 64, kernel_size 3, strides 3

Followed by 4 fully connected layers of

- 100 nodes
- 50 nodes
- 10 nodes

and final output

- 1 node

The final node is a float, so it is a regression model and not a classification model. It represents the angle to steer the car. 0 is center, negative and positive turns the car left and right.

The model includes RELU layers to introduce nonlinearity.

Here is a visualization of the architecture from the NVIDIA paper

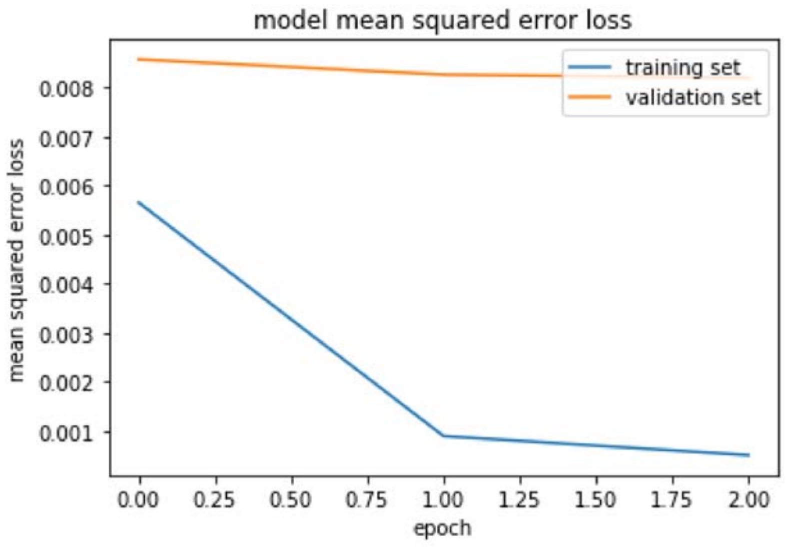
<img src = "NVIDIA_Model_Architecture.PNG">

```
In [1]: %matplotlib inline
        import matplotlib.pyplot as plt
```

Example code from history_object returned from model.fit_generator()

In [ ]:
```
# from keras.models import Model
# import matplotlib.pyplot as plt

# history_object = model.fit_generator(train_generator, samples_per_epoch =
#     len(train_samples), validation_data =
#     validation_generator,
#     nb_val_samples = len(validation_samples),
#     nb_epoch=5, verbose=1)

# ### print the keys contained in the history object
# print(history_object.history.keys())

# ### plot the training and validation loss for each epoch
# plt.plot(history_object.history['loss'])
# plt.plot(history_object.history['val_loss'])
# plt.title('model mean squared error loss')
# plt.ylabel('mean squared error loss')
# plt.xlabel('epoch')
# plt.legend(['training set', 'validation set'], loc='upper right')
# plt.show()
```

In [2]:
```
# I printed the output from the terminal to a text file called "model_data_out
put.txt"
# final loss
# [0.0056514676773990521, 0.00089578832602261224, 0.0005081381764944357]
# final validation loss
# [0.0085703516299530866, 0.0082603847246318762, 0.0081931548043767607]

### plot the training and validation loss for each epoch
list_loss = [0.0056514676773990521, 0.00089578832602261224, 0.0005081381764944
357]
list_val_loss = [0.0085703516299530866, 0.0082603847246318762, 0.0081931548043
767607]
plt.plot( list_loss )
plt.plot( list_val_loss )
plt.title('model mean squared error loss')
plt.ylabel('mean squared error loss')
plt.xlabel('epoch')
plt.legend(['training set', 'validation set'], loc='upper right')
plt.show()
```



3. Creation of the Training Set & Training Process

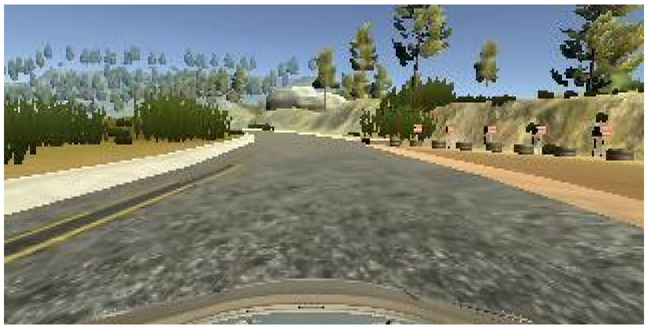
As mentioned earlier, collected the data suggested from Project: Behavioral Cloning - 16. More Data Collection

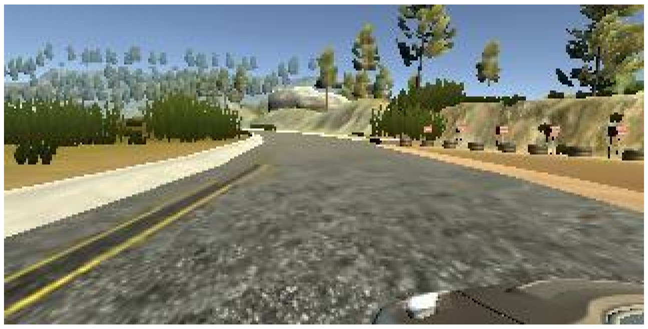
- two or three laps of center lane driving
- one lap of recovery driving from the sides
- one lap focusing on driving smoothly around curves

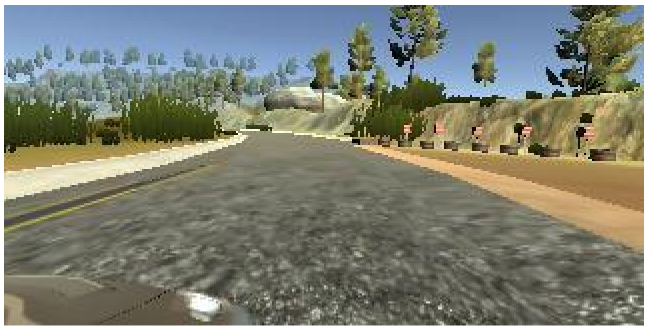
However, I did not get great results. Maybe it was the data that was produced. It is hard to control the car in the simulator with the mouse or keyboard at the speed and refresh rate of a VNC in a web browser with drop frames or freeze frames issues.

In the end, I just used the data that was provided which was found '/opt/carnd_p3/data'.

Here is an example of a center, left, and right camera images







To augment the data set, I also flipped images and angles thinking that this would help the model to turn right since the track is biased to turning left.

After the collection process, I had X number of data points. I then preprocessed this data by normalizing the image by dividing by 255, mean centering it by subtracting 0.5, and cropping out pixels not needed. I made a method create_model_and_preprocess_layers() line 444 in model.py to pre-process the data before adding more layers in the other methods for the model architecture.

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 3 as evidenced by graph shown earlier. I used an adam optimizer so that manually training the learning rate wasn't necessary.