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 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
- video.mp4 containing a video recording of my vehicle driving autonomously one lap around the track

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

My model consists of a convolution neural network with 3 convolutional layers, a flatten layer and 5 full connected layers, 7 RELU layers, 3 dropout layers, 1 lambda layer and a cropping layer (model.py lines 70-106).

The model includes RELU layers to introduce nonlinearity (code line 81, 87, 93, 99, 101, 103, 105), and the data is normalized in the model using a Keras lambda layer (code line 76). Images are cropped with a Keras Cropping2D layer, from size (16, 32, 3) to (9, 32, 3).

2. Attempts to reduce overfitting in the model

The model contains dropout layers in order to reduce overfitting (model.py lines 80, 86, 92).

The model was trained and validated on different data sets to ensure that the model was not overfitting (code line 110). 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 109).

4. Appropriate training data

Training data was chosen to keep the vehicle driving on the road. I used a combination of center lane driving, recovering from the left and right sides of the road.

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 start from the model mentioned in paper “End to End Learning for self-driving cars”, adjust the model, and fine tune it.

My first step was to use a convolution neural network model similar to the model mentioned in the NVIDIA paper. I thought this model might be appropriate because the problem solved is very similar to the project, and the model in the paper is simple and powerful. Before feeding the data into training, validation, and test sets, I resized the image from (160, 320, 3) to (16, 32, 3). The resizing

also made the model perform well without using generator. Since the size of the images are very small, the downsizing layer Maxpooling2D is not very appropriate after the convolutional layers(I still keep it by setting the pool size as (1,1)). And I also downsized the 5 convolutional layers in the paper to 3 in my model. I added one more fully connected layer in my model to gain more information.

In order to gauge how well the model was working, I split my image and steering angle data into a training and validation set. Firstly I used mouse and keyboard to collect data, which turned out to a really bad idea, the accuracy was very low -- around 1%. I figured out the low accuracy was because of data quality by firstly training with example data given by Udacity as a benchmark. The data I collected should build a model that having not dramatical lower performance (accuracy around 56%). Then I switched to the xbox 360 controller, which is a life saver. After the data quality problem was solved, I tested the model and the accuracy and loss of the test and validation data looked good. The training set mse loss and validation set mse loss are pretty similar. It seems my model is not overfitting or underfitting.

I read some discussions in the forum that suggested that while building the model you should check the performance frequently with automation driving test (simulator), which is a great suggestion. My first model only contains 3 loops of regular driving, and when I tested the model on the road, it run into the river, or got caught and stopped at side of the road, which are signs of lacking recovery data. Thus I spend more time on collecting recovery data by memorizing where the car failed, and how it failed. For example, when entering the first turn after passing the bridge, the car leaned to the right on/passing the lane. In the recovery data, I only recorded how the car successfully recovered from those previously failed situations. My normal driving data points number is 16072, and my recovery driving data points number is 12409.

After several iterations about running simulator collecting recovery data and rerun the model. The car can finally smoothly drive around track one without leaving the road.

2. Final Model Architecture

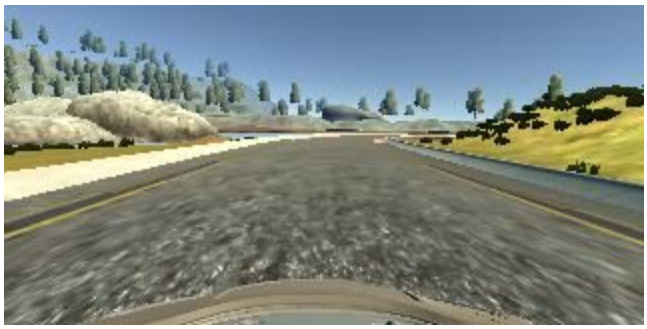
The final model architecture (model.py lines 70-106) consisted of a convolution neural network with the following layers and layer sizes .

- Cropping2D layer
- Normalization layer
- 1st convolutional layer, with 5x5 filter size and depth 24
- Dropout layer
- RELU layer
- 2nd convolutional layer, with 5x5 filter size and depth 36
- Dropout layer
- RELU layer
- 3rd convolutional layer, with 3x3 filter size and depth 48
- Dropout layer
- RELU layer
- a flatten layer
- 1st FC layer, with output 100

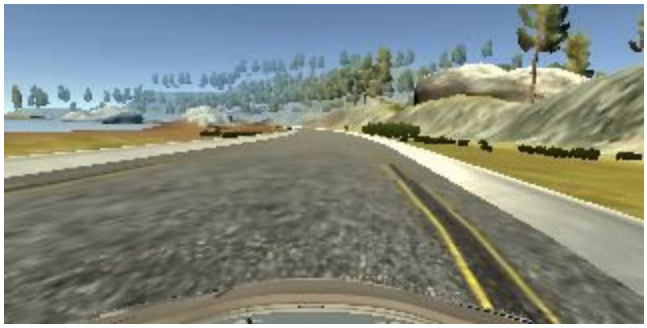
- RELU layer
- 2nd FC layer, with output 80
- RELU layer
- 3rd FC layer, with output 60
- RELU layer
- 4th FC layer, with output 40
- RELU layer
- 5th FC layer, with output 1 as the output of the model

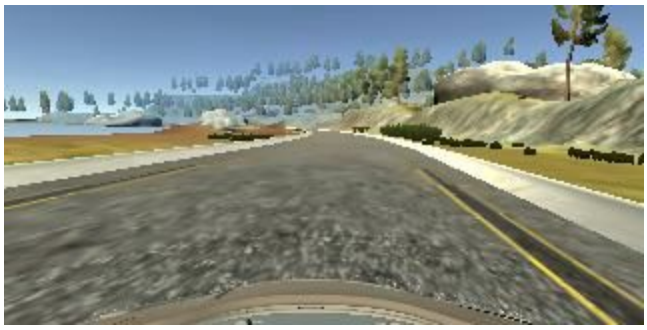
3. Creation of the Training Set & Training Process

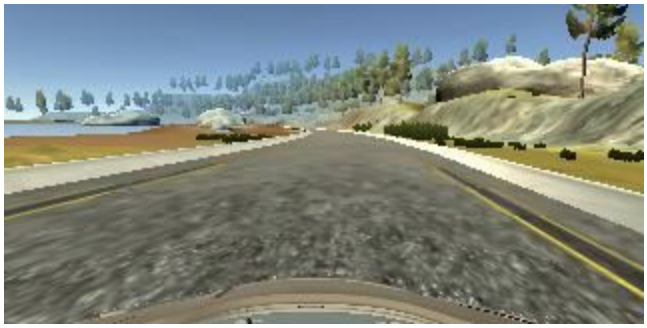
To capture good driving behavior, I first recorded three laps on track one using center lane driving. Here is an example image of center lane driving:



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 when encountering the similar situations. These images show what a recovery looks like when starting from tires on the right lane, turning to the center and finally recovering.

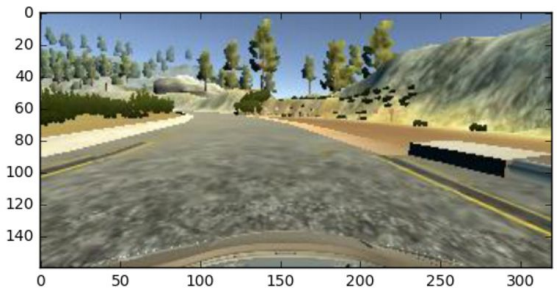






To augment the data sat, I also flipped images and angles only on the normal driving data, thinking that this would help with left turn bias. For example, here is an image before and after being flipped:

```
In [10]: plt.imshow(img)
Out[10]: <matplotlib.image.AxesImage at 0x7f5187c5d630>
```




```
In [13]: new_img = cv2.flip(img_cp,1)
In [14]: plt.imshow(new_img)
Out[14]: <matplotlib.image.AxesImage at 0x7f5187c1f1d0>
```

After the collection process, I had 28481 number of data points (normal: 16072, recovery:12409.). I then preprocessed this data by resizing from (160, 320, 3) to (16, 32, 3), cropping the image to (9, 32, 3), and normalization.

The training, validation, and test data are split as 64%, 16%, 20%.

I used this training data for training the model. The validation/test set helped determine if the model was good. I used an adam optimizer so that manually training the learning rate wasn't necessary.

Here is the output info of my final model below. The ideal number of epochs was 5. The difference of the mean square error loss between training set and validation set became smaller and smaller: in epoch 5 training loss is 0.0211, and validation loss is 0.0216. It takes 6+4+4+4+4 = 22s to run the model in Amazon EC2 g2.2xlarge instance.

Epoch 1/5

18227/18227 [==============================] - 6s - loss: 0.0294 - acc: 0.4673 - val_loss: 0.0253 - val_acc: 0.4714

Epoch 2/5

18227/18227 [==============================] - 4s - loss: 0.0240 - acc: 0.4707 - val_loss: 0.0231 - val_acc: 0.4727

Epoch 3/5

18227/18227 [==============================] - 4s - loss: 0.0225 - acc: 0.4718 - val_loss: 0.0237 - val_acc: 0.4700

Epoch 4/5

18227/18227 [==============================] - 4s - loss: 0.0213 - acc: 0.4734 - val_loss: 0.0222 - val_acc: 0.4738

Epoch 5/5

18227/18227 [==============================] - 4s - loss: 0.0211 - acc: 0.4727 - val_loss: 0.0216 - val_acc: 0.4740

Test score: 0.0204818469045

Test accuracy: 0.47112515359