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
  - because of github.com file limitation (100MB), I split model.h5 into two parts model.h5.tar.001 and model.h5.tar.002
- 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

I used NVIDIA ene-to-end model as base model with a dropout layer before the last dense layer. (File:model.py, Method:Model.build_model, Lines:153-169)

Data normalization is done via Keras Lambda layer. NVIDIA model includes RELU layers to introduce nonlinearity.

2. Attempts to reduce overfitting in the model

I added a dropout layer with 0.2 drop rate before the last dense layer in order to reduce overfitting. NVIDIA vanila version did not produce good results without dropout layer.

I generated almost 10 different datasets with simulator. Then first train the model with the dataset provided by Udacity and each of of generated datasets in order to find bad datasets. Then finally I chose 4 dataset for final train. 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 (File:model.py, Method:Model.train_save, Lines:191). I used saving and loading methods of Keras model in orther to use transfer learning from previous trained models (File:model.py, Method:Model.load_model, Lines:171-181).

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, driving in reverse direction.

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 train and test different known models and modify them inorder to get better results if it was necessary.

My first step was to use a convolution neural network model similar to the LeNet but it didn't produce good results. The I use the NVIDIA ene-to-end model with out modification.

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 NVIDIA model had a low mean squared error on the training set but a bit high mean squared error on the validation set. This implied that the model was overfitting.

To combat the overfitting, I modified the NVIDIA model and add a dropout layer with 0.2 droprate before the last dense layer.

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:

1. On the bridge, to imporove the driving behavior in this case I generate more driving dataset
2. Shadows on the road, this case solved after I dropout layer inclusion

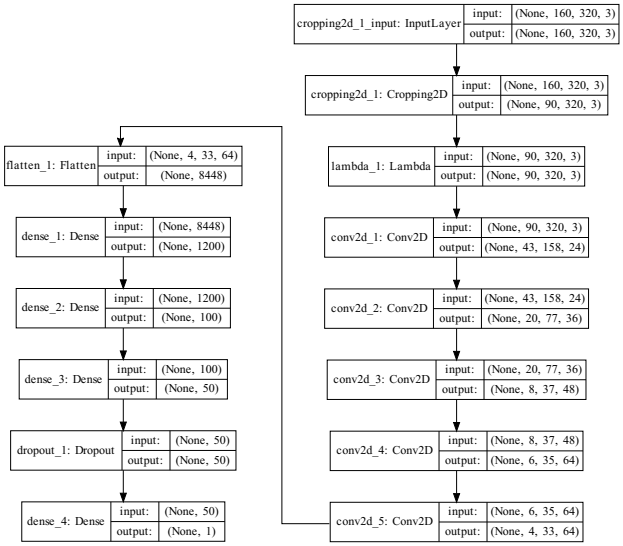
At the end of the process, the vehicle is able to drive autonomously around the track without leaving the road.

2. Final Model Architecture

The final model architecture (File:model.py, Method:Model.build_model, Lines:153-169) consisted of a convolution neural network with the following layers:

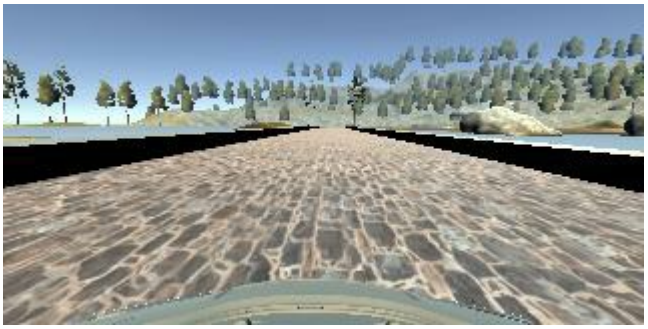
1. Cropping2D layer in order to crop top of the input image
2. Lambda layer in order to normalize data
3. 5 Convolution layers with 24, 36, 48, 64 and 64 depth sizes, the first 3 convolution layers have kernel size 5 and the last 2 layers have kernel size 3. Each convolution layer has a RELU activation.
4. Flatten layer in order to flat output of last convlution layer
5. 4 dense layer with 1200, 100, 50 and 1 sizes.
6. dropout layer before last dense layer with 0.2 droprate

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

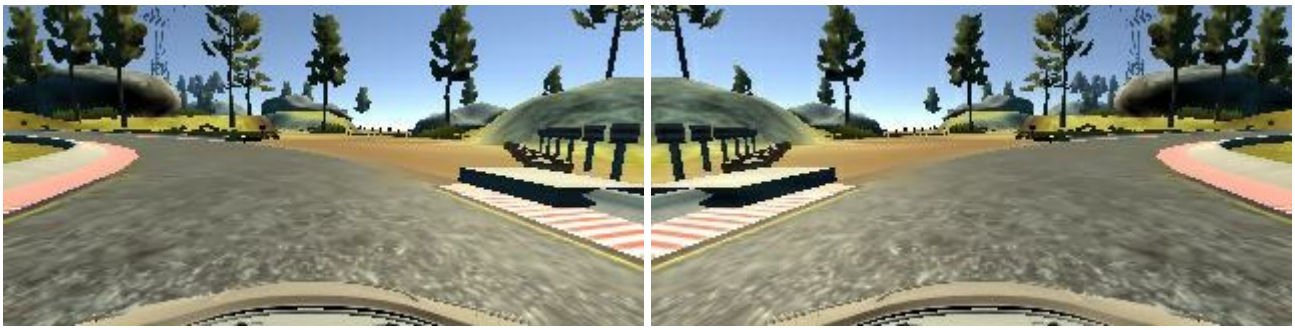
To capture good driving behavior, I first recorded 5 laps on track one using center lane driving. Here is an example image of center lane driving:



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 from road sides. These images show what a recovery looks like:

Then I repeated this process on track two in order to get more data points.

To augment the dataset, I also flipped images and angles thinking that this would the image when I drive in reverse direction. For example, here is an image that has then been flipped:



Further more I used left and right camera images with steering angle corrected by increasing and decreesing steering angle of center image respectively (File:model.py, Variable:DataGenerator.steering_correction, Line:36). The correction value defined in (File:model.py, Method:DataGenerator._read_image_list, Lines:66,69,75,78).

After the collection process, I had 26800 number of data points. I then preprocessed this data by converting image color space from RGB to YUV (File:model.py, Method:DataGenerator._get_data, Line:100). Because of this color space conversion I modified driver.py line 66 to apply this preprocessing to images received from simulator autonomous mode.

The model.py code organized into two classes DataGenerator and Model. DataGenerator class is responsible for reading csv files and making a list of files to be read later by get_train_data and get_valid_data generators. Model class is responsible for building the model or loading it from saved h5 file, training the model using fit_generator and a DataGenerator instanse and finally visualize model architecture into an image file.

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 (File:model.py, Method:Model.train_save, Line:185) I used an adam optimizer so that manually training the learning rate wasn't necessary.