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.

Catalog

- Part 1: Files Submitted & Code Quality
- Part 2: Data Collection and Preprocessing
- Part 3: Model Architecture and Training Strategy
- Part 4: Quetions & My Solutions for Project 3

Part 1: 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.ipynb** -- containing the script to create and train the model as well as to plot those images display in this readme
- **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
- **README.md** -- summarizing the results
- **run1.mp4** -- recording video in autonomous mode
- **examples** -- folder contains all the images displayed in the README

- **READ.pdf** -- the pdf format of this REAMDME

2. Submission includes functional code

I defined a function `generator(samples, batch_size=32)` (model.ipynb in cell 9 or model.py in line 118-150)

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.

Part 2: Data Collection and Preprocessing

1. Data Recording

Only track one was used to collect and record data. Training data was chosen to keep the vehicle driving on the road. I used a combination of:

- one lap of clockwise center lane driving
- one lap of clockwise recovering from the left and right sides of the road
- counter-clockwise driving

Only track one was used to collect and record data.

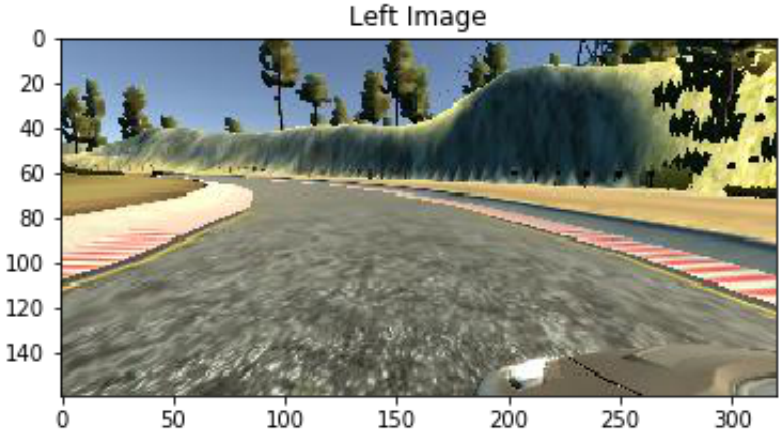
2. Appropriate Training & Validation Data

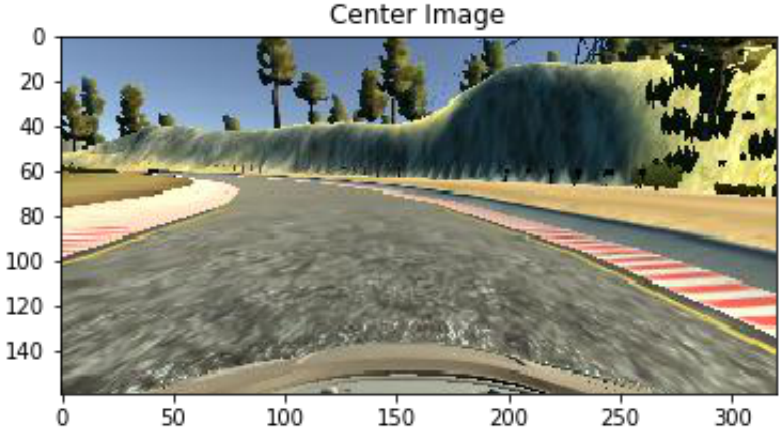
2.1 Using Multiple Cameras

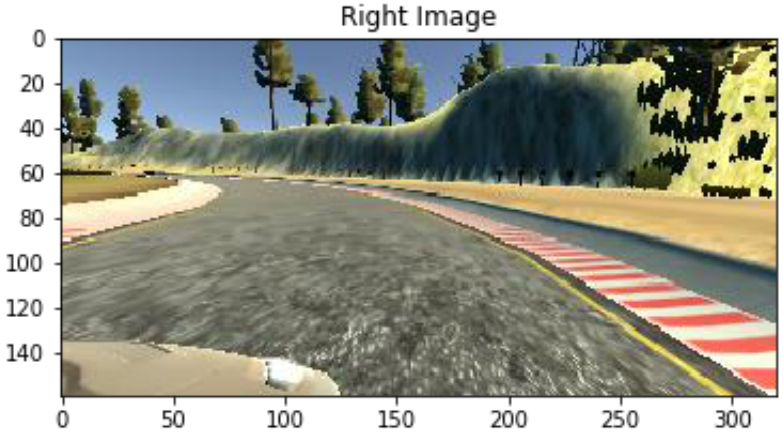
Training data was chosen to capture good driving behavior and keep the vehicle driving on the road. I used a combination of center lane driving, recovering from the left and right sides of the road.

Firstly I recorded laps on track one using center lane driving.

Secondly I recorded the vehicle recovering from the left side and right sides of the road back to center so that the vehicle would learn how to steer if it drifts off to the left or the right. These images show what a recovery looks like starting from left to right :


Left Image
0
20
40
60
80
100
120
140
0
50
100
150
200
250
300

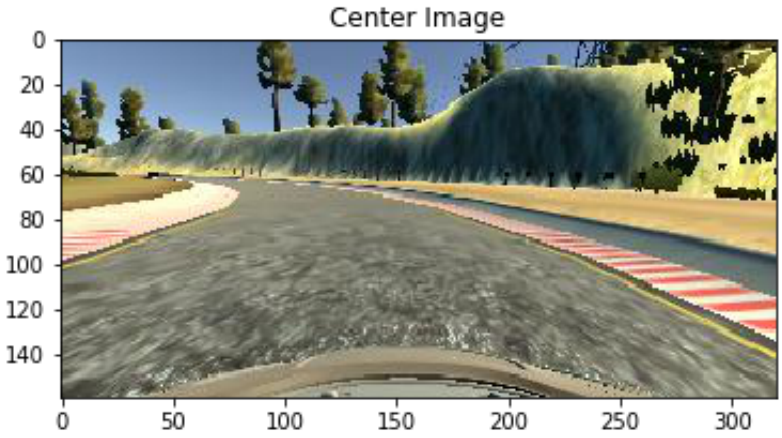

Center Image
0
20
40
60
80
100
120
140
0
50
100
150
200
250
300

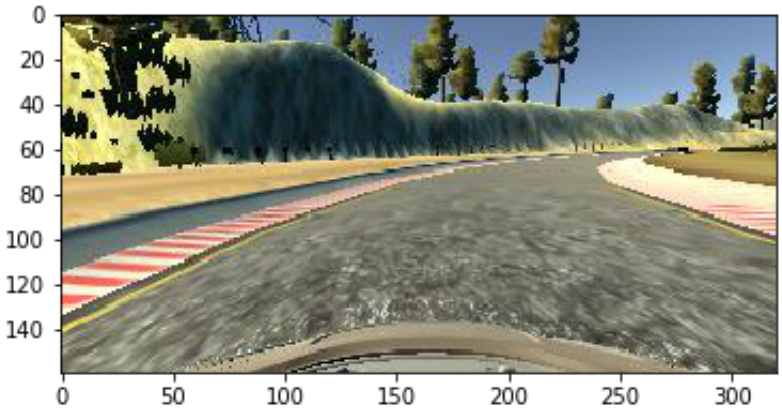

Right Image
0
20
40
60
80
100
120
140
0
50
100
150
200
250
300

2.2 Data Augmentation

Although I recorded both clockwise and conter-clockwise laps, the left turns in my data are still much more than the right turns, which could contribute tp a left turn bias.

Flipping images is an effective technique for helping with the left turn bias. For example, here is an image that has then been flipped:





2.3 Data Preprocessing

The data processing is actually carried out at the beginning of the model training.

- **Nomalization** -- A Lambda layer is used to normalize the data, converting its value range from (0, 255.0) to (-0.5, 0.5).
  `model.add(Lambda(lambda x: x/255.0 - 0.5, input_shape=(160, 320, 3)))`

- **Cripping Images** -- A Cropping layer is used to crip 70 and 25 rows pixels from the top and the bottom of the images respectively. `model.add(Cropping2D(cropping=((70, 25),(0,0))))`

2.4 Data Split

After the collection process, I had 51264 number of data points.

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.

Part 3: Model Architecture and Training Strategy

1. Solution Design Approach

The overall strategy for deriving a model architecture was to imply a well built-up CNN and then adapt it to my dataset.

My first step was to use a convolution neural network model similar to the NVIDIA architecture. I thought this model might be appropriate because it is more powerful than LeNet.

Avoid Underfitting

My strategy to avoid underfitting is to apply powerful CNN, which is NVIDIA architecture in this project, and collect enough data (more details in Part2: Data Collection and Preprocessing).

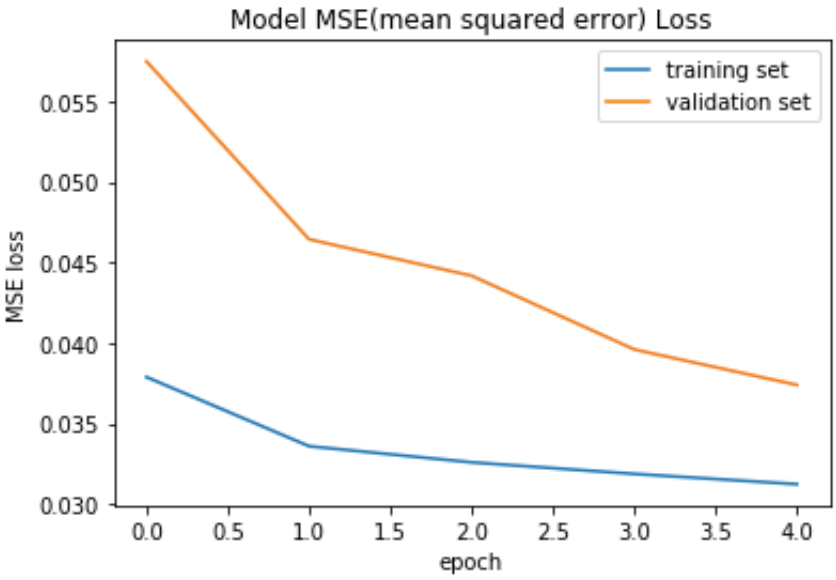
Avoid Overfitting

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, I two dropout layers in my model architecture.

Parameter Tune

- batch size = 32
- learning rate -- I used an adam optimizer so that manually training the learning rate wasn't necessary.
- epoch = 5 -- The ideal number of epochs was 5 as evidenced by the Model MSE line chart.

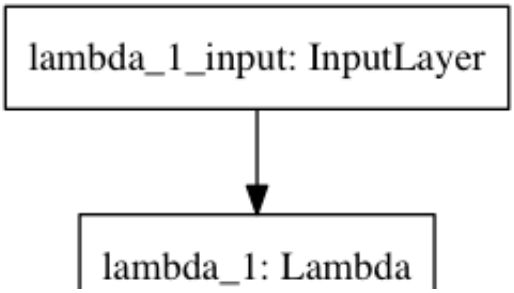


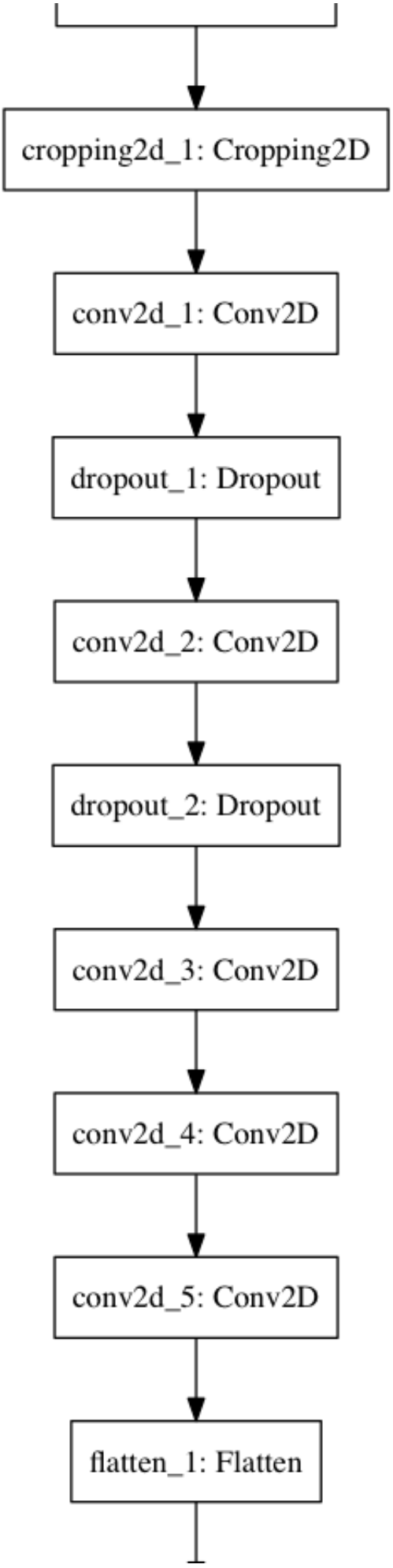
2. Final Model Architecture

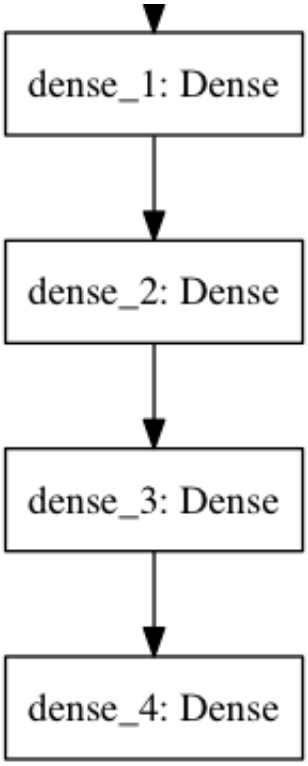
The final model architecture (model.py lines 18-24) consisted of a convolution neural network with the following layers and layer traits:

Layer	Description
Input	160x320x3 RGB image
Lambda Layer	Normalization
Cropping	Cropping Images 65x320x3
Convolution_1	output depth = 24, kernel size = 5x5, stride = 2x2
RELU_1	contained in Convolution_1
Dropout_1	drop probability 0.7
Convolution_2	output depth = 36, kernel size = 5x5, stride = 2x2
RELU_2	contained in Convolution_2
Dropout_2	drop probability 0.7
Convolution_3	output depth = 64, kernel size = 3x3, stride = 1x1
RELU_3	contained in Convolution_3
Convolution_4	output depth = 64, kernel size = 3x3, stride = 1x1
RELU_4	contained in Convolution_4
Convolution_5	outputs 84
RELU_5	contained in Convolution_5
Flatten	
Dense_1	outputs 100
Dense_2	outputs 50
Dense_3	outputs 10
Dense_4	outputs 1

Here is a visualization of the architecture:




cropping2d_1: Cropping2D
conv2d_1: Conv2D
dropout_1: Dropout
conv2d_2: Conv2D
dropout_2: Dropout
conv2d_3: Conv2D
conv2d_4: Conv2D
conv2d_5: Conv2D
flatten_1: Flatten



3. Train Model

I used **model.ipynb** to train my model and save the final output in **model.h5**. Then I download **model.ipynb** as **model.py** and only kept cell [1], [9], [10], [11] and [15] in **model.py**.

At the first time I the model.h5 trained by model.ipynb, but the outcome was not satisfactory. Therefore I collected more data and train my model via model.py, the outcome is as below:

```
(carnd-term1) Ryan (origin *) CarND-Behavioral-Cloning-P3 $ python model.py
Using TensorFlow backend.
Epoch 1/5
55914/55914 [==============================] - 2833s - loss: 0.0395 - acc: 0.227
0 - val_loss: 0.0376 - val_acc: 0.2290
Epoch 2/5
55914/55914 [==============================] - 2788s - loss: 0.0358 - acc: 0.227
1 - val_loss: 0.0340 - val_acc: 0.2290
Epoch 3/5
55914/55914 [==============================] - 2918s - loss: 0.0344 - acc: 0.227
1 - val_loss: 0.0339 - val_acc: 0.2290
Epoch 4/5
55914/55914 [==============================] - 2908s - loss: 0.0333 - acc: 0.227
1 - val_loss: 0.0330 - val_acc: 0.2290
Epoch 5/5
55914/55914 [==============================] - 2900s - loss: 0.0325 - acc: 0.227
2 - val_loss: 0.0334 - val_acc: 0.2287
The model has been saved successfully.
```

3. Run Autonomous Mode and Record Video

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` See my outcome in **run1.mp4**.

Part 4: Quetions About this Project

Q1: How to visualize the model architecture?

Solution1:

I use

```
from keras.utils import plot_model
from keras.models import load_model

plot_model(model, to_file='./examples/model.png')
```

to visualize my model architecture.

Q2: Q2: My car will lose control after passing the bridge, how can I improve it?

Solution2:

- *More Data*: I collected more turning driving examples.
- *Aligning the color spaces*: I order to align the color spaces the both **drive.py** and **model.py** I made changes in **model.py** as the tabel below

replace	with	to
`cv2.imread()`	`PIL.Image.open()`	open or read image
`cv2.save()`	`plt.savefig()`	save image
`images.append(cv2.flip(image, 1))`	`images.append(np.fliplr(image))`	flip image

- *Normalization*: I changed my normalization from `model.add(Lambda(lambda x: x/225.0 - 0.5, input_shape=(160, 320, 3)))` to `model.add(Lambda(lambda x: x/127.5 - 1.0, input_shape=(160, 320, 3)))`.

Thus the data range changed from (-0.5, 0.5) to (-1.0, 1.0) and the loss would double for the same sample and model weights. I wonder whether double loss help the network to learn quickly with bigger learning rate.