Self-Driving Car Engineer Nanodegree

Project 3: Behavioral Cloning

Overview

The goal of this project is to build a behavioral cloning network. In order to do that I used a simulator provided by Udacity to drive around the track and collect data of my driving behavior. The data consists of – amongst others - images taken by the simulator and the respective steering angles. Then I wrote a deep neural network and I used the recorded data and the respective steering angles to train my model. To prove my network learned how to drive on the track I recorded multiple videos of the vehicle driving around autonomously. The autonomous driving comes down to a regression problem where the car needs to predict the steering angle based on its position on the track.

Files submitted and Code quality

1. My project includes the following files:
    a. model.ipynb containing the script to create and train the model
    b. drive.py for driving the car in autonomous mode
    c. model.h5 containing a trained convolutional neural network
    d. writeup_report.pdf summarizing the results
2. 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. The model.ipynb file contains the code for training and saving the convolutional 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

My model is based on the NVIDIA network which has been used for an end-to-end self-driving test by NVIDIA. One of my takeaways from the project though is that the success or failure of the autonomous driving – at least on this entry level – doesn’t depend too much on the network itself, much more on the quality training data.

As a preprocessing step, I used a lambda layer to normalize the input images. I also cropped the input images, that’s because the upper zone of the images – mostly the sky and meaningless image background - doesn’t contain useful information for the network to train.

In the model, I used a ReLU activation function after each layer as well as 2x2 max pooling following the convolutional layers.

The model architecture looks as follows:

Layer (type)	Output Shape	Param #	Connected to
lambda_1 (Lambda)	(None, 160, 320, 3)	0	lambda_input_1[0][0]
cropping2d_1 (Cropping2D)	(None, 65, 318, 3)	0	lambda_1[0][0]
convolution2d_1 (Convolution2D)	(None, 33, 159, 24)	1824	cropping2d_1[0][0]
activation_1 (Activation)	(None, 33, 159, 24)	0	convolution2d_1[0][0]
maxpooling2d_1 (MaxPooling2D)	(None, 33, 159, 24)	0	activation_1[0][0]
convolution2d_2 (Convolution2D)	(None, 17, 80, 36)	21636	maxpooling2d_1[0][0]
activation_2 (Activation)	(None, 17, 80, 36)	0	convolution2d_2[0][0]
maxpooling2d_2 (MaxPooling2D)	(None, 17, 80, 36)	0	activation_2[0][0]
convolution2d_3 (Convolution2D)	(None, 9, 40, 48)	43248	maxpooling2d_2[0][0]
activation_3 (Activation)	(None, 9, 40, 48)	0	convolution2d_3[0][0]
maxpooling2d_3 (MaxPooling2D)	(None, 9, 40, 48)	0	activation_3[0][0]
convolution2d_4 (Convolution2D)	(None, 9, 40, 64)	27712	maxpooling2d_3[0][0]
activation_4 (Activation)	(None, 9, 40, 64)	0	convolution2d_4[0][0]
maxpooling2d_4 (MaxPooling2D)	(None, 9, 40, 64)	0	activation_4[0][0]
convolution2d_5 (Convolution2D)	(None, 9, 40, 64)	36928	maxpooling2d_4[0][0]
activation_5 (Activation)	(None, 9, 40, 64)	0	convolution2d_5[0][0]
maxpooling2d_5 (MaxPooling2D)	(None, 9, 40, 64)	0	activation_5[0][0]
flatten_1 (Flatten)	(None, 23040)	0	maxpooling2d_5[0][0]
dense_1 (Dense)	(None, 100)	2304100	flatten_1[0][0]
activation_6 (Activation)	(None, 100)	0	dense_1[0][0]
dense_2 (Dense)	(None, 50)	5050	activation_6[0][0]
activation_7 (Activation)	(None, 50)	0	dense_2[0][0]
dense_3 (Dense)	(None, 10)	510	activation_7[0][0]
activation_8 (Activation)	(None, 10)	0	dense_3[0][0]
dense_4 (Dense)	(None, 1)	11	activation_8[0][0]

Total params: 2,441,019
Trainable params: 2,441,019

Network architecture

Model Training

To train the model I used the data captured by the simulator. The simulator is “equipped” with 3 cameras, each mounted on the windscreen, one at the center and two others are in the top corners. For driving the vehicle, one can use the mouse or the keyboard. Better training data leads to a better training. Also, more, highly variant training data leads to an even better training as the network can generalize better.

I used several techniques to make my network train better. I recorded 4 laps of driving clockwise as well as counter-clockwise. I used the recorded data from all 3 cameras. I also flipped the images as a further data augmentation step. All these increase the training data and variance the model can learn from and therefore help the model to generalize better.

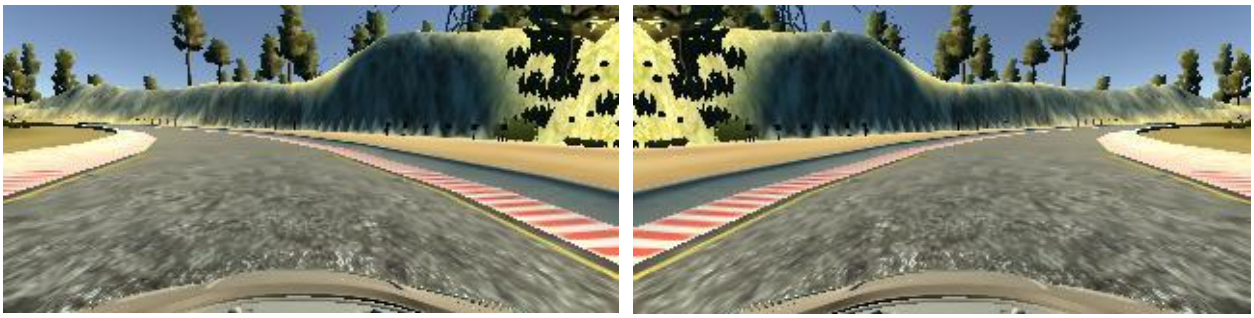
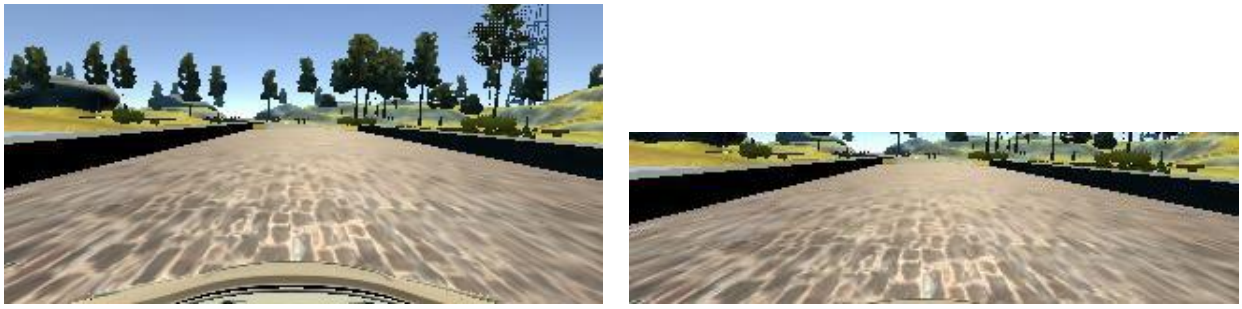


Image captured by the center camera (left) and its flipped version (right)



Original image taken by the simulator (left) and its cropped version

I recorded 4-4 laps of training data in each direction, that means 12428 images taken by each camera. Even after cropping the images, the amount of the training data is way too much to fit it into the memory of my computer at once. That's where generators come into play. Instead of storing the data in memory all at once, a generator takes pieces of the whole data and process them on the fly only when it needs them. A generator is like a coroutine, a process that can run separately from another main routine.

I chose the batch size being the default 32. The number of epochs doesn't have to be large to achieve a good model. As another pre-processing step, I shuffle the samples to do a random permutation of the collected, sequential data. I used Adam optimizer with the learning rate of 0.0001 to optimize my network's gradient descent.

Conclusion

I have a few important takeaways from this project. First, it's worth to point out how much more important the good quality/amount data than what network is being used or how well-tuned are the hyperparameters. I've used a few pre-processing and data augmentation steps but one might use many more other techniques to help the network generalize better. I might have collected data by driving on the other track. The dominant direction of driving the car is straight ahead, I might have thrown some of the 0-degree steering angle data points away. I could have used Dropout layers to help my network avoiding overfitting. I might have used more image-pre-processing steps, like shearing the images.
That's being said my network doesn't reach its full capabilities, but good enough to keep the car on track even with the highest speed. It's also important to note how poorly my network performs once I change to the other track and try to let the car to drive itself. The first step to improve the performance on the jungle track would be to train my network included images from that track too.