Behavioral Cloning

Mohit Pandey

2018
January

Chapter 1

Introduction

Build a 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

Chapter 2

Writeup/Readme

You're reading it. Project code is at *https://github.com/diamondspark/Self-Driving-Car-Behavior-Cloning*

Chapter 3

Files Submitted & Code Quality

3.1 Submission includes all required files and can be used to run the simulator in autonomous mode

My project includes the following files:
1. DeepNet.ipynb containing the script to create and train the model
2. drive.py for driving the car in autonomous mode
3. model.h5 containing a trained convolution neural network
4. writeup.pdf summarizing the results

3.2 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

3.3 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.

Chapter 4

Model Architecture and Training Strategy

4.1 An appropriate model architecture has been employed

I started with LeNet architecture. The car would very quickly go off the road especially around the lake. So I moved on to try the nVidia Autonomous Car Group model. After few experiments and data augmentation the car drove the entire track seamlessly after training for 3 epochs. The model architecture itself can be located in cell#2 of DeepNet.ipynb in *nVidiaModel()*. Following are the details of model architecture.

Layer Type & Name	Output Shape	Parameters count	Layer connected to
lambda_1 (lambda)	(None, 160, 320, 3)	0	lambda_input_2[0][0]
cropping2d_1 (Cropping2D)	(None, 90, 320, 3)	0	lambda_1[0][0]
convolution2d_1 (Convolution2D)	(None, 43, 158, 24)	1824	cropping2d_1[0][0]
convolution2d_2 (Convolution2D)	(None, 20, 77, 36)	21636	convolution2d_1[0][0]
convolution2d_3 (Convolution2D)	(None, 8, 37, 48)	43248	convolution2d_2[0][0]
convolution2d_4 (Convolution2D)	(None, 6, 35, 64)	27712	convolution2d_3[0][0]
convolution2d_5 (Convolution2D)	(None, 4, 33, 64)	36928	convolution2d_4[0][0]
flatten_1 (Flatten)	(None, 8448)	0	convolution2d_5[0][0]
dense_1 (Dense)	(None, 100)	844900	flatten_1[0][0]
dense_2 (Dense)	(None, 50)	5050	dense_1[0][0]
dense_3 (Dense)	(None, 10)	510	dense_2[0][0]
dense_4 (Dense)	(None, 1)	11	dense_3[0][0]

4.2 Attempts to reduce overfitting in the model

I tried to keep overfitting to minimal to none by training the model for as few as 3 epochs. I would have tried other standard overfitting avoidance techniques like Max pooling, dropout etc. but by training the model for 3 epochs I got a model that performs perfectly fine on both Lake track as well as Jungle track.

4.3 Model parameter tuning

The model used an adam optimizer, so the learning rate was not tuned manually (DeepNet.ipynb Cell#2)

4.4 Appropriate training data

It was very important to have a good training data and also enough frames/pixels for a model such powerful as this to learn from. We tried a bunch of data permutations and eventually settled for using the driving data provided by Udacity. In addition to that we garnered 2 rounds of Lake track using Udacity simulator. We also got 1 round of lake track in opposite direction. For the purpose of generalization, we collected 1 round on Jungle track as well. While collecting the additonal driving data, we made sure to get on the road edges every once in a while so the model have training data to learn what to do when car tries to run off the street.
The simulator provides three different images: center, left and right cameras. Each image was used to train the model.

Chapter 5

Model Architecture and Training Strategy

5.1 Solution Design Approach

5.1.1 LenNet Architecture

I started the project by trying out the LeNet Architecture for regressing on Steering Angles. I used the data provided by the Udacity Team. The model trained pretty fast on my GPU but the results were poor after multiple experiments with epoch numbers and the car would very soon get off the road. The model in general looked pretty naive for this task and we needed a more powerful architecture.
Following this I decided to try a more powerful network by nVidia Team which was especially designed for self driving cars and hence was my motivation that the network would work if I do my experiments right.

5.1.2 nVidia Autonomous Car Group Architecture

The first step, I used the same dataset provided by Udacity and trained this nVidia Architecture. The results were straightaway better than the LeNet model. The car would make the first turn/curve but would drive into the lake. The next step was to normalize
I decided to normalize the images. I used (pixel)/255 -0.5 to approximately normalize the data. This was done because this helps to normalize each dimension so that the min and max along the dimension is -1 and 1 respectively. In our case of RGB images, this ensures that the contribution of each of the 3 channel is approximately the same and the set of weights learned affect each of the 3 channels in approximately same manner.
It was also intuitive that not the entire image was necessary for training. Approximately the upper 1/3$^{\text{rd}}$ of the image was sky and didn't contain meaningful

information for our cause. So it makes sense to crop out this region before training. We utilized a Cropping2d layer to achieve this.
The resulting model performs very well and the car drives the entire first track albeit it goes over the sidelines sometimes and during the second round, it bumps onto side wall of the bridge.
The next step was to get more data. I drove the car

- 2 rounds forward on lake track
- 1 round backward on lake track
- 1 round forward on Jungle track

Following this the model was trained again for 3 epochs and we had a model that drives the car perfectly on both lake and Jungle track.

Figure 5.1: Lake track backwards

Figure 5.2: Jungle track

5.2 Final Model Architecture

The final model architecture consists of following

- Input (160x320x3)
- Cropping2D ((50,20),(0,0))
- Convolution2D
- Flatten

- Dense
- Conv2D Kernel:5x5 RELU Strides: 2x2 Filters:24
- Conv2D Kernel:5x5 RELU Strides: 2x2 Filters:36
- Conv2D Kernel:5x5 RELU Strides: 2x2 Filters:48
- Conv2D Kernel:3x3 RELU Strides: 1x1 Filters:64
- Conv2D Kernel:3x3 RELU Strides: 1x1 Filters:64
- Dense Output :100 LINEAR Input:8448
- Dense Output :50 LINEAR Input:100
- Dense Output :10 LINEAR Input:50
- Dense Output :1 LINEAR Input:10

5.3 Creation of the Training Set & Training Process

Additional data was collected by driving the car in the simulator in following ways The next step was data-augmentation. This was done after I saw the distribution of classes in the dataset

- 2 rounds forward on lake track
- 1 round backward on lake track
- 1 round forward on Jungle track

The data was randomly shuffled and trained for 3 epochs.