Behavioral Cloning (BC) Project Submission Writeup Report

For the Udacity Self-Driving Car Program – Term 1

Submitted by:

Stewart Teaze

June 25, 2017

Project Goals

The goals of this project are:

- Use a car driving Simulator to collect data (driver viewpoint images & steering angles) of good driving behavior
- Build a Convolution Neural Network(CNN) model in Keras that predicts steering angles, during autonomous operation of the car in the Simulator, based on runtime images received from the Simulator
- Train and validate the model with a training and validation set
- Test that the model successfully drives around Simulator track one without leaving the road
- Summarize the results with a written report

Rubric Points

The following sections of this document relate to each of the project's rubric points (project requirements), and provide details of how each requirement was addressed in the project implementation.

---

1. Submitted Files

1. model.py Contains the Python Script to create and train the model
2. drive.py Contains script for driving the car in autonomous mode
3. model.h5 contains the trained CNN
4. writeup_report.pdf - This file is the writeup itself.
5. video.mp4 – Video recording of vehicle driving autonomously for 1 lap around test track.

The project code and data files can be found at:
https://github.com/blunderbuss9/CarND-P3-BC-StewartTeaze

2. Functional Code

a. Car-Driving Simulator and Autonomous Driver Server

The Udacity-provided vehicle driving Simulator (based on the Unity3D visualization game engine API), can be operated in “Training Mode” or “Autonomous Mode”.

TRAINING MODE: When operated in Training Mode, the Simulator can be manually operated, to generate imagery and steering command data, which can be recorded and saved to disk. The recorded imagery and steering command data can then be used to train the CNN model implemented in file model.py.

AUTONOMOUS MODE: When the Simulator is operated in Autonomous Mode, and used simultaneously and in conjunction with the drive.py autonomous driver server (which utilizes the built/trained CNN model in model.h5, to derive steering commands that are sent back to the simulator program, corresponding to each frame of video received from the simulator), the simulated car will drive autonomously around the selected simulator track. To start the drive.py autonomous driver server, execute the following command, after the Simulator is started and placed in Autonomous Mode:

```
python drive.py model.h5
```

b. CNN Model and Training Implementation – model.py & model.h5

The model.py file contains the code for training and saving the convolution neural network into file model.h5; specifically, the Keras-based CNN pipeline training, validation, and model-saving implementation. Detailed comments are also provided to explain how each section of the code works.

3. CNN Model Architecture and Training Strategy

a. Keras CNN implementation

The Neural Network is implemented using the Keras API (described at https://keras.io/), and is based on the NVIDIA CNN architecture **[Bojarski16]**.

The 12 (deep) layers of this project's machine learning network model's architecture consist of:

1) A Lambda normalization layer effected on the 160x320 input image data [line 70 in model.py].
2) A 2D cropping layer to reduce the resulting size of the input data to 65x320 in size (to eliminate extraneous sky and hood/road imagery) [line 71].
3) 5 Convolutional layers with RELU (Rectified Linear Unit) activation functions applied to each. The first three convolutional layers utilize a filter (kernel) size of 5x5 with 2x2 strides (subsamples), with gradually increasing numbers of convolutional feature maps, of quantities of 24, 36, and 48 for the three layers. The last two convolutional layers utilize a filter size of 3x3 (with no subsampling), and each result in 64 increasingly more compacted feature maps [lines 72-76].
4) A Flattening fully-connected layer [line77].
5) 4 Densing layers, the first 3 with RELU activation functions applied, and the last with a TANH activation function applied, to arrive at the singular steering prediction value [lines 78-81].

b. Avoiding Training Overfitting

Regularization methods to address overfitting, that were evaluated, and used where beneficial, were Data Augmentation, Dropout, and Early Stopping **[Geron17]**.

DATA AUGMENTATION: Augmenting the large original set of "center lane driving" data, with sets of data of example situations of recovering from "general off-center" situations, and "wide turn recovery" were implemented – see section d. below, for more details on the implementation and benefits of the augmented data sets.

DROPOUT: Dropout layers were added to the working model, between densing layers, using the Keras function model.add(Dropout(.2)); while some improvement in the validation accuracy figures was noted, when training past 5 epochs, the actual performance of the model when dropout was included, was unacceptable in practice (the vehicle would slowly veer off the track toward the left, almost immediately after being engaged with the model, autonomously in the Simulator). As such, dropout layers were not used to address overfitting in the final model implementation, and Early Stopping and Data Augmentation were utilized.

EARLY STOPPING: Numerous training passes were made, using various number of epochs, and overfitting would usually begin to occur after approximately 5 Epochs; as such, the final model was implemented using the training overfitting technique of Early Stopping, after training for 5 epochs.

c. Keras model.compile() & model.fit() Parameters

The loss parameter was initially set to 'mse' (mean square error). During early attempts to improve performance (after "recovery" samples had been added to the training data), the loss parameter was changed to 'mae' (mean absolute error); however, this change resulted in greatly-reduced performance of the model, so 'mse' was used for the remainder of the training-test efforts, and it could be seen that changing the loss parameter, in the middle of the training-test effort does not appear to be advisable [line 83].

The optimizer parameter was set to 'adam', avoiding the need to tune the learning rate manually [line 83].

In the Keras model.fit() call, the number of training epochs was set to 5, the training set/validation set split was set to 80%/20% and the training data was shuffled [line84].

d. Appropriate Training Data

Starting with a baseline of center lane driving recording only, it could be seen that it was important to also add recorded samples of "recovery" driving sequences, so that the autonomous model could have good examples of how to get back toward the middle of the road, when it inadvertently wandered too much towards the left and right sides of the road.

Many different training "runs" were recorded, but the best combination worked out to be the baseline of center lane driving of about 8000 sample images, and recovery "runs" (included in the submitted run7 and run12 image directories); one of 250 "general left and right correction recoveries", and the second of about 25 images of an "emergency recovery" in the "brown dirt turn", located about 70% of the way around the track (this turn seemed to most consistently give the model the most trouble).

Figure 1 below provides a visual representation of the Training Data steering command distribution, with the Green Bars representing the Base (original, center-lane driving) steering command data, the Brown Bars representing the "general left and right-side correction-to-center recoveries" made during "run7", and the Red Bars representing the "special-case critical recovery" from a "red hashes" leading into a sharp left-hand turn, which helped the models deal best with the "treacherous brown dirt turn" (which is preceded by a short sequence of "red hashes").

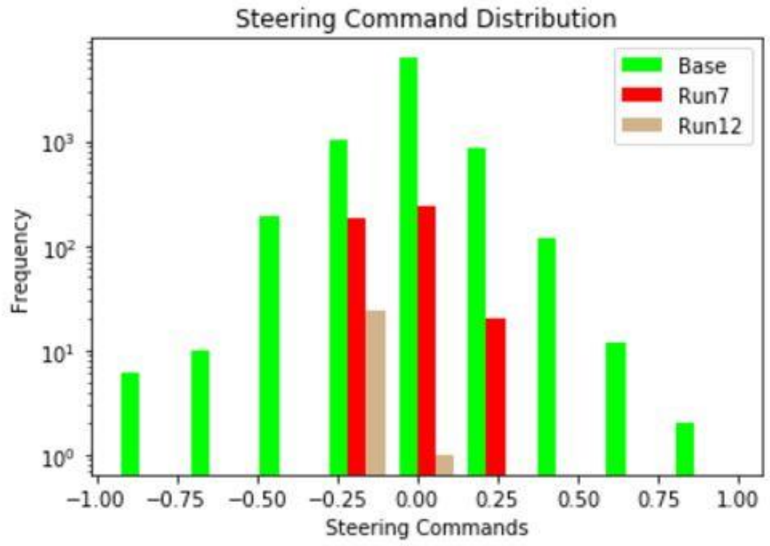

Figure 1

e. Recovery Image Recording Sequences

During numerous early training/test runs, with less well-performing models, and with numerous different recorded capture sequences (runs), it was seen that the data recorded during "general recovery to center" run7 and "emergency recovery" run12 performed the best when combined with the base training dataset, as far as influencing the image/steering command predication model to command the autonomous vehicle to stay on the track for the longest period of time. The three images provided in Figure 2 below, show the beginning, middle, and end image frames from the 25 sequential image frames selected (in run12.csv) from the run12 IMG directory of captured images from run12 ("safe/not-over-reacting" recovery from off-center in sharp left-hand turn). This sequence of images (and associated steering commands) helped achieve the best performance in the most-problematic "dead man's curve" (brown curve leading to "dirt road bypass") during early test runs.

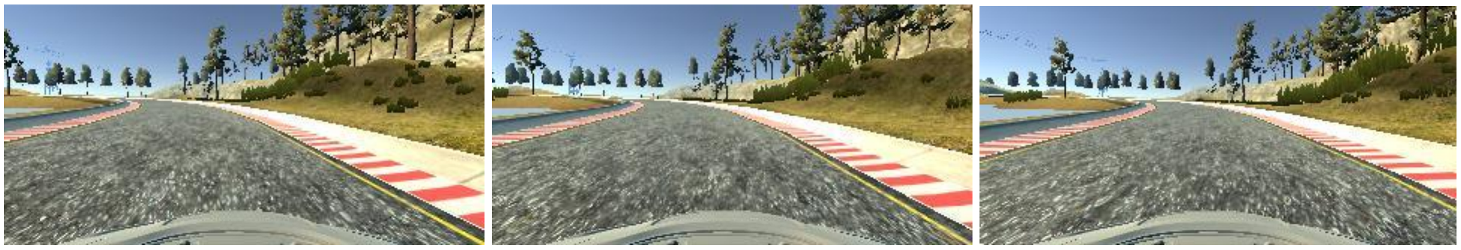

Figure 2

f. Solution Design Approach and Training/Test Implementation Process

After learning that a Microsoft multi-layer (>120 layer) Deep Learning implementation had performed well in recent deep learning competitions, I briefly researched trying to find information on using and implementing such a design for this self-driving car project; however, while Microsoft has recently released an open source beta version of an "advanced virtual world" (simulator) for training autonomous drones, which is said to be also usable to test autonomous driving systems (https://techcrunch.com/2017/02/15/microsoft-open-sources-a-simulator-for-training-drones-self-driving-cars-and-more/), I was not able to find much helpful information on easily using the Microsoft multi-layer approach on this particular project.

Conversely, the NVIDIA self-driving car CNN approach was very well-documented, and lots of helpful and applicable resources were available on The Internet.

Initially, I started with code excerpt shown in the video in Project: Behavioral Cloning Lesson 14. Even More Powerful Network; implemented a working solution, and began to record and test various combinations of centered-driving/recovery driving to get the autonomous vehicle to complete the project requirement of achieving 1 good lap around the track 1, without going off the road. However, after several days of implementing/training with/testing dozens of different combinations of centered+recovery recordings, and having various levels of success (the best getting the autonomously-operating vehicle to about 75% around the track), I finally concluded that I was not converging on a full solution to get all the way around the track, and so I did some more research on the actual NVIDIA architecture, and found that the base solution, that I had started with, was missing activation functions on the 4 fully-connected layers – so, I added RELU and TANH activation functions, re-trained and re-tested… and was amazed and extremely pleased that the vehicle immediately performed extremely well, using my "best combination" of training data, and easily made it fully all the way around track 1, without any significant anomalies (about the only anomaly I observed were some slight deviations from center, when passing through lots of moving shadows, but these deviations did not come close to causing the vehicle to leave the track, as can be observed in the video submitted in file video.mp4).

REFERENCES

[Bojarski16] Bojarkski, M., Yeres, P., Choromanska, A, Choromanski, K., et al (2016). Explaining How a Deep Neural Network Trained with End-to-End Learning Steers a Car; arXiv:1604.073116v1 [cs.CV] 25 Apr 2017 https://arxiv.org/pdf/1704.07911.pdf

[Geron17] Hands-On Machine Learning with Scikit-Learn & TensorFlow by Aurelien Geron (O'Reilly). Copyright 2017 Aurelien Geron, 978-1-491-96229-9

[Goodfellow16] Deep Learning; Goodfellow, I., Bengio, Y., and Courville, A. (2016). MIT Press

[CS231n17] Karpathy, A., Li, F. (2017) http://cs231n.stanford.edu/ Convolutional Neural Networks for Visual Recognition; Stanford University, Spring 2017.