Project 03: Behavioral Cloning

Thomas Antony
December 20, 2016

1 RUNNING THE MODEL

The Keras model is stored in the file *model_5.json* with the weights being stored in *model_5.h5*. Run the model using the command:

```
$ python drive.py model_5.json
```

drive.py pre-processes the telemetry obtained from the simulator and passes it along to the Keras model for predicting the steering angle. It also has a rudimentary "cruise-control" system for setting the throttle to achieve a target speed. It was found that the model is able to drive on Track 1 at 30 mph and navigate Track 2 at speeds of up to 20 mph.

In order to obtain the live trainer code, be sure to run the following command after cloning the repository.

```
$ git submodule update --init --recursive --remote
```

2 MODEL ARCHITECTURE AND TRAINING STRATEGY

2.1 NEURAL NETWORK ARCHITECTURE

Multiple training architectures based on Convolutional Neural Networks were evaluated. Most of them failed. Transfer learning with VGG16 and Inception v3 was also attempted. These networks were used by removing the final layers and replacing them with my own fully connected and convolutional layers. However, none of them were able to drive properly on the tracks. This could

be because of bad training data, which was mitigated in my final model by using live training described in Section 2.2.2.

Finally, the architecture described in NVIDIA's End-To-End deep learning paper[1] was used. This architecture was chosen at the end after experimenting with many others, primarily due to the small size and hence low processing latency. A normalization layer was added on top of the network to normalize the input images. The use of the live training methodology described in Section 2.2.2 also helped getting this model to work. Additional models were not tested using the live training strategy due to lack of time. The model architecture is shown in Fig. 2.1. The model consists of five convolutional layers to extract features followed by four fully-connected layers to perform regression.

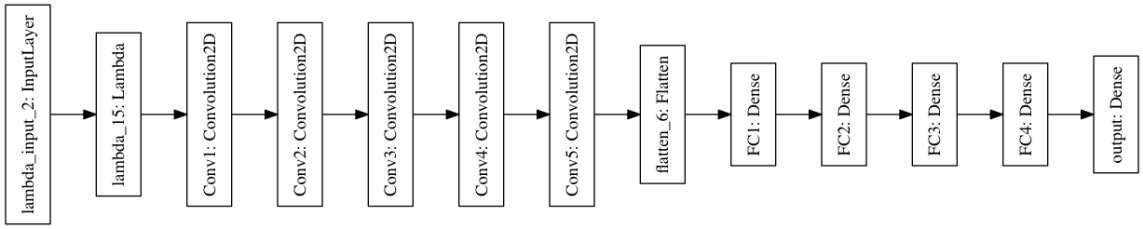

Figure 2.1: Model architecture

The architecture does not contain any dropout layers. The MSE of the model was always above 0.01 for working versions and so there doesn't seem to be much overfitting going on. However, to be sure of this, a model can be trained on just data from track 1 and tested on track 2. This wasn't done due to time constraints as the focus was on getting at least one model that worked on one or both tracks. To further test for "memorizing" the track, the car was also tested by deliberately perturbing it's path from time to time and it was found that it always returned to the center of the lane.

During the initial training step, a separate dataset was generated and used for testing. A validation dataset with 10% of the original dataset was also used. Further more, live testing + training was also performed as described in the upcoming sections. The resulting model is able to drive on track 1 at 30 mph and track 2 at 20 mph.

2.2 Training Strategies

Two separate strategies were used for training the model. The first was a batched training strategy using data collected from the "Training mode" of the simulator. This is done is *train_model.ipynb*. The model was then further fine tuned with a low learning rate using a live trainer that was developed specifically for this purpose. This is further discussed in Section 2.2.2

2.2.1 BATCHED TRAINING

The Jupyter Notebook *train_model.ipynb* contains the code used to train the model initially. The dataset obtained from the simulator consisted of image files and a CSV file. The CSV file was loaded into a pandas dataframe and randomized. Data was collected for both straight lane-keeping and recovery from perturbations. Both tracks were driven in both directions and data was collected.

An image generator that takes the pandas dataframe as input and generates batches of training data was developed. This can be seen in the `batch_generator` method in the notebook. The generator can run indefinitely and starts over the from the beginning when it runs out of data. The data set was augmented by also using the right and left camera images with a 1.5 degree shift to the original steering angle. This resulted in a total of around 24000 images (starting from 8000 original data points). The input images were converted to YUV color space and a 240×80 region of the image was cropped out. This region starts under the horizon, removes most of the exterior environment and focuses on the road. This is similar to what was done in the NVIDIA paper [1]. The image was then resized to 200×66 before being passed into the model.

The ADAM optimizer with a learning rate of 0.0001 was used. The data was trained over 6 epochs of 24000 samples each.

2.2.2 LIVE TRAINING

The live trainer allows manual override of the model while it drives the car in autonomous mode. The car can then be controlled using the keyboard and there is also the option to collect the data being generated at this time and use it to fine-tune the model. More details can be seen in the ReadMe file of the live trainer and this article. The live trainer can also be found as a sub-module in the project repository.

The live trainer was used to collect additional data in portions of the track where the model was failing to navigate. This was done by considerably slowing down the car when approaching these sections and then driving manually through the area while simultaneously collecting data and training the model. This way, the model was able to learn to get through these spots where the original dataset was probably lacking in data. During the live training, the learning rate was further reduced to $1e-6$ to prevent corruption of existing weights from any sudden mistakes during manual driving.

2.3 CRUISE CONTROL IMPLEMENTATION

Th cruise control was implemented as a simple proportional gain controller. The throttle was computed from the speed feedback available in the telemetry.

$$T = K_p \left(v_{target} - v \right)$$

where T is the throttle, K_p is the proportional gain and v_{target} and v are the targeted and current speeds respectively. The gain was picked to be 0.35. This was able to get the speed within ± 1

mph of the target speed.

3 SIMULATION

The model was tested in the simulator in autonomous mode on an Ubuntu Desktop with and AMD FX-8320 processor, 8GB RAM and a GTX 1070 GPU. The videos of running the Simulator in autonomous mode using both the live trainer and *drive.py* can be seen at the following links:

https://www.youtube.com/watch?v=oAujMdetS-0 - Track 1 using live trainer in autonomous mode

https://www.youtube.com/watch?v=5vQrzi_CVaM - Track 1 using *drive.py*

https://www.youtube.com/watch?v=SINNzj16PvA - Track 2 using *drive.py*

REFERENCES

[1] Bojarski, Mariusz and Del Testa, Davide and Dworakowski, Daniel and Firner, Bernhard and Flepp, Beat and Goyal, Prasoon and Jackel, Lawrence D and Monfort, Mathew and Muller, Urs and Zhang, Jiakai and others. End to End Learning for Self-Driving Cars. *arXiv preprint arXiv:1604.07316*, 2016.