Behavioral Cloning Project

Done by: Wael Farag

1. Training Data Collection

Two main types of training data are used through-out the project:

1. **Type 1 - Udacity Supplied Data**: These collections with an unzipped size of 365MB consists of 24,108 images equally divided between center, left and right camera shots. Each image is 160x320 pixels size with 3 channels for RGB colors. The index of the data is stored in a CSV file which contains 8,036 line of records.
2. **Type 2 - Simulator Generated Data**: collected using the Udacity supplied simulator. A data set with an unzipped size of 808MB consists of 49,851 images equally divided between center, left and right camera shots. Each image is 160x320 pixels size with 3 channels for RGB colors. The index of the data is stored in a CSV file which contains 16,617 line of records. The data has been generated by driving the car manually around Track 1 several times (~ 10 times) with as good as possible driving behavior.

Several subroutines has been written for data visualization and analysis as shown in “model.py” lines 45 => 91. An Example of the output of these subroutines are presented in Figure 1 which displays a sample of the generated training data (type 2), and Figure 2 which presents the histogram of the steering angle values collected during driving (type 2).

The data (both type 1 and 2) is divided into 2 separate parts: training data which represents 80% of the chunk and validation data which represents 20% of the chunk.

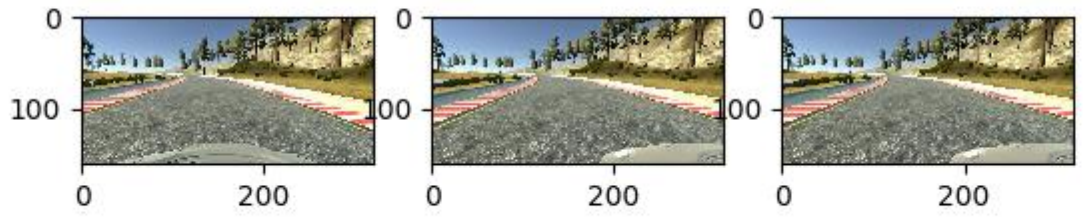

Figure 1. Sample of the images collected: center, left and right respectively.

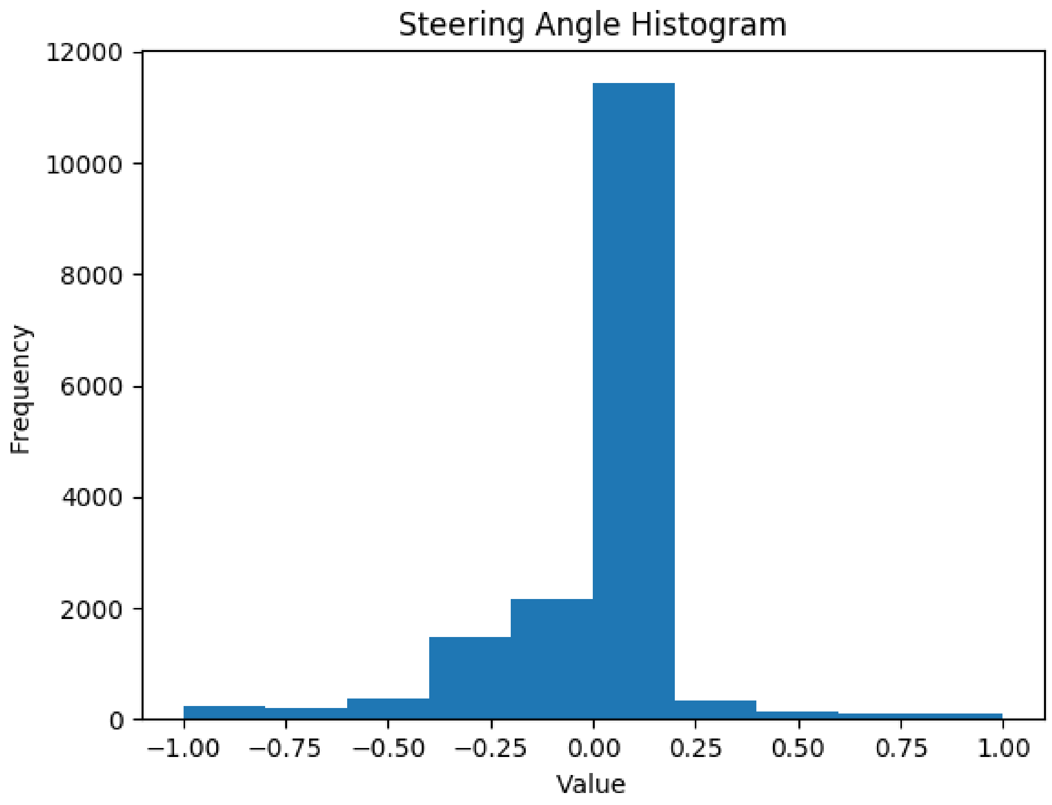

Figure 2. The histogram of the steering angle values collected during driving.

2. Training Data Pre-Processing

The following steps describes the implemented pre-processing steps in order of execution:

1. **Normalization (color)**: this is done for color images using the "Lambda function" in Keras. It is coded in line 158 in "model.py". The values of the pixels are scaled between -1 & 1 and centered on zero.
2. **Cropping images**: The images have been cropped from the top by 70 pixels and from the bottom by 25 pixels, in order to focus on the region of interest (ROI) and to reduce the number of inputs (faster learning process). The cropped images have the size of 65x320x3.
3. **Flipping images**: The data has been doubled (augmented) by flipping all the images (around the y axis) and reversing the sign of the corresponding steering angle. Accordingly, the type-1 data becomes 48,216 samples, and type-2 data becomes 99,702 samples. In other words, each CSV line record can generate 6 training samples (center, left, right, flipped-center, flipped-left, and flipped-right).
4. **Shuffling the training data**: this is done once each training BATCH as in "model.py line 98".

5. **<u>Using generator function to load data in memory</u>**: this step helps a lot to smooth out the training process, as it is actually mandatory. Loading the whole data in the computer memory was actually not possible (or at least not practical). Each patch (size = 128 images and angles) is generated and loaded in memory as shown in “model.py: lines 92=>142”. The fit_generator() function by Keras is used to manage the whole process.

Finally, the actually used pre-processing pipeline is: Color-Image → Normalization → Cropping→ Flipping→ Shuffling→ Batch Memory Loading.

We found out that the above pipeline is fair enough and produced the required results. However, other techniques were put into consideration; it may be needed or for future endeavors; as follows:

1. Converting color images to grey: reduces complexity a lot by reducing the number or inputs to 1/3 rd.
2. Edge detection and lane finding using Cunny algorithm and Hough transform.
3. Filter Blurring using Gaussian filtering to remove noise.
4. Data augmentation using other data processing techniques like random distortion, brightness manipulation, jitter and rotation … etc.

3. The Model Architecture

The implemented neural network model is a modified version of NVIDIA architecture [1] and is given the name WAF-NVIDIA. The model is coded using Keras in “model.py: lines 148=>183”. Table 1 below describes the architecture in details:

Table 1: WAF-NVIDIA Architecture.

#	Layer	Size/Output	Parameters	Comment
1	Input	160x320x3	-----------------	Color – 3 Channels RGB
2	Normalization	160x320x3	lambda x: x/127.5 - 1	Scaling the inputs => -1 & 1, using Keras Lambda function
3	Cropping	65x320x3 = 62,400	The new height will be (160-70) - 25 = 65	Cropping the images, cut 70 pixels from the top and 25 pixels from the bottom.
4	Convolutional #1	31x159x24	No. Filters =24, Kernel =5x5, Strides = 2x2, Padding: Valid, Activation = RELU	No Pooling [(W−F+2P)/S]+1
5	Convolutional #2	14x78x36	No. Filters =36, Kernel =5x5,	No Pooling [(W−F+2P)/S]+1

			Strides = 2x2, Padding: Valid, Activation = RELU	
6	Convolutional #3	5x38x48	No. Filters =48, Kernel =5x5, Strides = 2x2, Padding: Valid, Activation = RELU	No Pooling [(W−F+2P)/S]+1
7	Convolutional #4	3x36x64	No. Filters =64, Kernel =3x3, Strides = 1x1, Padding: Valid, Activation = RELU	No Pooling [(W−F+2P)/S]+1
8	Convolutional #5	1x34x64	No. Filters =64, Kernel =3x3, Strides = 1x1, Padding: Valid, Activation = RELU	No Pooling [(W−F+2P)/S]+1
9	Flatten	1x34x64 = 2,176	Keras Flatten Function	
10	Drop-out	2,176	Keep Probability: 0.5 => 0.7	
11	Fully Connected #1	200	Keras Dense Layer	With biases
12	Drop-out	200	Keep Probability: 0.5 => 0.7	
13	Fully Connected #2	100	Keras Dense Layer	With biases
14	Drop-out	100	Keep Probability: 0.5 => 0.7	
15	Fully Connected #3	20	Keras Dense Layer	With biases
16	Drop-out	20	Keep Probability: 0.5 => 0.7	
17	Fully Connected #4	1	Keras Dense Layer	Output layer

4 Drop-out layers are added as to prevent over-fitting during training, and the fully connected layers are widened (actually doubled in size). Also, No Pooling layers are used here, as it is a regression problem not a classification. Also, all the convolutional layers are adapted according to the input image sizes after normalization and cropping.

4. The Model Training

- The WAF-NVIDIA model is trained using the parameters listed in Tables 2, 3 and 4.

Table 2: WAF-NVIDIA training parameters (Learning Rate).

Algorithm	Parameter	Value	Comment
ADAM Optimization [2]	Learning Rate	0.001	For epochs: 0 – 5 for data type 1 For epochs: 0 – 5 for data type 2
		0.0005	For epochs: 6 – 8 for data type 2
		0.0002	For epochs: 8 – 9 for data type 2

Table 3: WAF-NVIDIA training parameters.

Parameter	Value	Comment
Left Angle Correction	0.25	Radians
Right Angle Correction	-0.25	Radians

Table 4: WAF-NVIDIA training parameters (others).

Parameter	Value	Comment
Batch Size	120	For epochs: 0 - 9
Epochs	15	The whole training
Keep Probability	0.5 => 0.7	For Drop-out Layers

- The training results are listed in Table 5 below:-

Table 5: WAF-NVIDIA training results.

Phase	Data Type	Loss Value	Parameters	Comment
Phase 1 Coarse Tuning	Type-1 Data	Training : 0.0235 Validation: 0.0205	5 Epochs Learning Rate = 0.001 Keep Prob. = 0.5 Figure 3	Coarse Tuning with Udacity Data. Not enough for full learning.
Phase 2 Fine Tuning	Type-2 Data	Training : 0.0455 Validation: 0.0411	5 Epochs Learning Rate = 0.001 Keep Prob. = 0.5 Figure 4	Fine Tuning with self-collected data. Proved enough for full learning with acceptable Performance.
Phase 3 Fine Tuning	Type-2 Data	Training : 0.0417 Validation: 0.0377	3 Epochs Learning Rate = 0.0005 Keep Prob. = 0.6 Figure 5	More fine tuning with self-collected data. Caused over-fitting with a kind of inferior performance.
Phase 4 Fine Tuning	Type-2 Data	Training : 0.0348 Validation: 0.0295	2 Epochs Learning Rate = 0.0002 Keep Prob. = 0.7 Figure 6	More fine tuning with self-collected data. Full learning with very good performance.

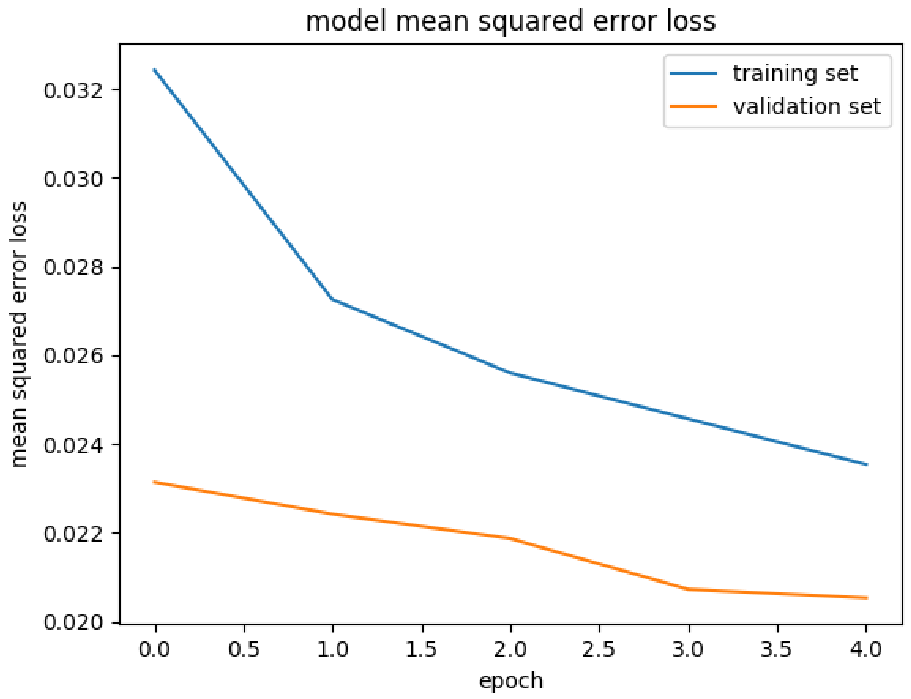

Figure 3. Learning progress for WAF-NVIDIA for data type 1 (Udacity Data).

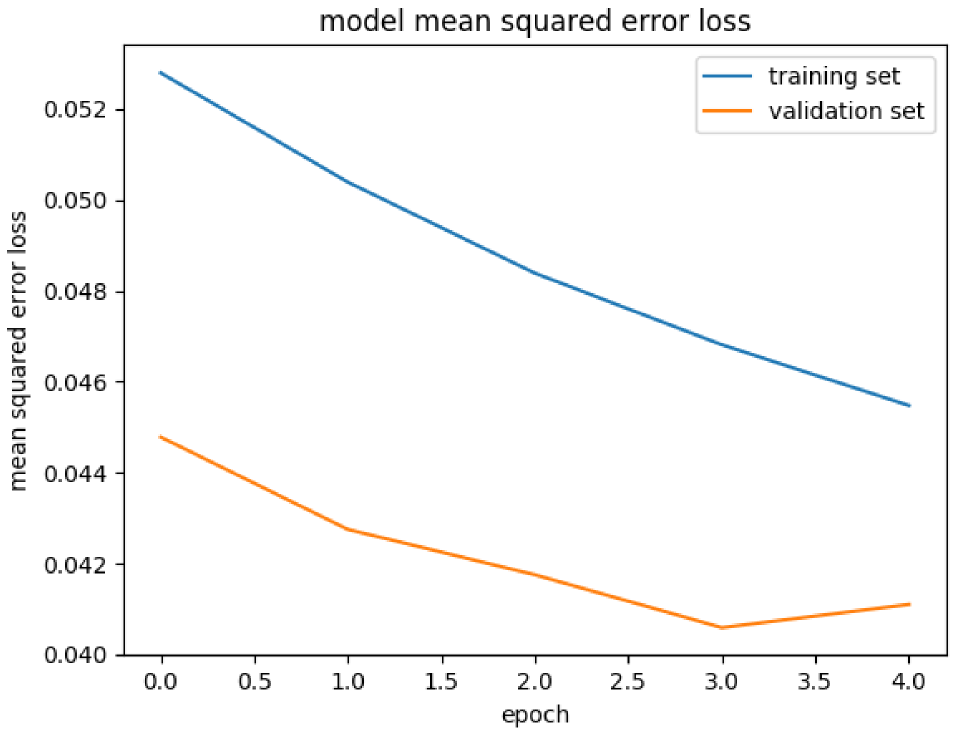

Figure 4. Learning progress for WAF-NVIDIA for data type 2 (Self-Collected).

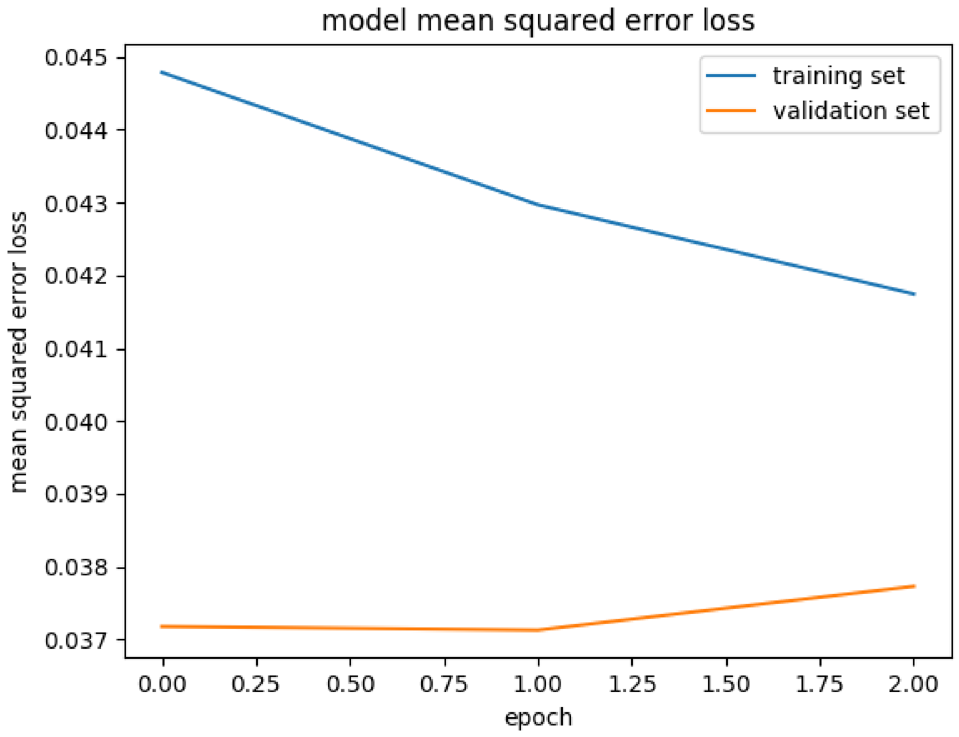

Figure 5. Learning progress for WAF-NVIDIA for data type 2 (Self-Collected).

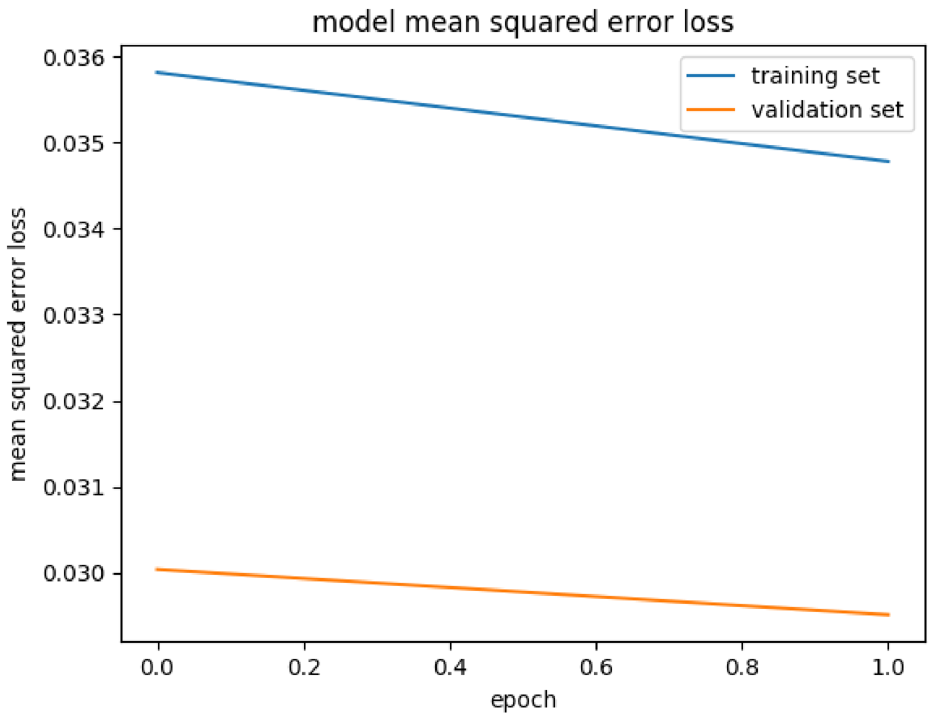

Figure 6. Learning progress for WAF-NVIDIA for data type 2 (Self-Collected).

- Attached with this report the model Keras file “model.h5” which represents the state of the model after the training in Figure 6.
- The state of the model is a bit over-fitting after the training represented by Figure 5. For this reason, the learning rate is reduced further and the keep probability increased.
- “run1.mp4” and “run2.mp4” represent the simulation result on Track 1 of the WAF-NVIDIA model after being fully trained. The results show several laps on the track without touching the lane lines and following the center of the lane with good precision.

5. Solution Approach

The training of the WAF-NVIDIA has been carried-out through several trials to the results shown in Table 5. The following observations has been collected during the training process:

1. I have tried to fully train the network using the Udacity supplied data only several times by augmenting the data as well, however, I didn’t achieve acceptable results, and the car always hit the boarders.
2. I have started collecting data using the simulation and maneuvering the car using the arrows on the keyboard. I was successfully able to collect good data for training by looping the car around the Track 1 several times (~ 10).
3. After coarse tuning the model using the type-1 data, the resultant model weights are then reused to training the data on type-2 data for fine tuning the model (with ADAM optimizer learning rate of 0.001). The matter is a kind of transfer learning approach. Note that the two types of data are never used together. After this fine tuning phase the resultant model is then tested on track 1 and produces acceptable results.
4. In order to improve the performance further, the learning rate of the ADAM optimizer has been halved to 0.0005 and the model got training for further 3 epochs. However, the resultant model is over-fitting as shown in Figure 5. Furthermore, the testing proved that after produced inferior performance even with both training and validation loss are lower than the previous case. Consequently, this model has been further fine-tuned.
5. The learning rate of the ADAM optimizer has been reduced further to 0.0002 and keep probability increased to 0.7 and the model got training for extra 2 epochs. The resultant model is then tested on track 1 and produces very good performance as shown by videos “run1.mp4” & “run2.mp4”. “run1.mp4” has been simulated on “Good” graphic quality while “run2.mp4” has been simulated on the “Fastest” Graphic quality.

6. Shortcoming of the implemented approach

The following list summaries the identified shortcomings:

1. The presented neural network model doesn’t not have a memory, it take momentarily decision and doesn’t build on previous states to make the current decision. However, I believe driving is a sequential process and the current approach doesn’t mimic that.

2. After training the network on Track 1 data, it has been tested on Track 2. The result was horribly bad (kind of expected) because it has never seen this track before. I am afraid this approach may require the network to be exposed to a massive number of tracks in order to generalize well for actual street deployment (testing).

7. Suggested Improvements

The following list summaries the suggested improvements:

1. Test other network topologies with memory like Long Short-Term Memory (LSTM) models for end-to-end learning. I think it is an interesting area of research.
2. Train the network for more Tracks and try to make it generalize as much as possible.
3. Generate more data from the current collected data by random distortion, brightness manipulation, jitter and rotation … etc.
4. Apply the concept of an FIR filtering for the steering angle estimation during simulation instead of using the raw estimated value directly. In such a case, the new estimated value will depend of previous history as well.

References

1. M Bojarski, D Del Testa, D Dworakowski, B Firner, B Flepp, P Goyal, ..., "End to End Learning for Self-Driving Cars", arXiv:1604.07316, 25 Apr 2016.
2. Diederik P. Kingma, Jimmy Lei Ba, "ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION", Published as a conference paper at ICLR 2015, arXiv: 1412.6980v9, 30 Jan 2017.