Autonomous Driving - Behavioral Cloning

Files Submitted & Code Quality

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

My project includes the following files:

- **model.py** containing the script to create and train the model
- **drive.py** for driving the car in autonomous mode
- **model.h5** containing a trained convolution neural network – Trained with Track 1 data only. Works best for Track 1 if "set_speed = 18" or below in drive.py [already set to 18]
- **writeup_report.pdf** summarizing the results
- **model_Track2_Track1.h5** containing a trained convolution neural network – Trained with both Track 2 and Track 1 data. Works best for Track 2 if "set_speed" in drive.py is 12. Also works for Track 1 if "set_speed" in drive.py is 15 or below.
- **Track1_18mph.mp4** Track-1 video at 18 mph
- **Track2_12mph.mp4** Track-2 video at 12 mph

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-

(project requirement, to test Track 1): `python drive.py model.h5`

This model works for Track-1 only since training data used to get 'model.h5' was taken from only Track-1. To test Track-1, "set_speed" in drive.py should be 18 [already set].

(Or, optionally to test Track 2): `python drive.py model_Track2_Track1.h5`

This model works for both Track-1 and Track-2 as the training data used to get 'model_Track2_Track1.h5' was a mixture of both Track-1 and Track-2. To test Track-2, "set_speed" in drive.py should be 12. Track-1 works if "set_speed" is 15 mph or below. Reason will be stated later below.

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.

Model Architecture and Training Strategy

1. An appropriate model architecture has been employed

First the data is normalized in the model using a Keras lambda layer (code line 139). Then top 70 pixels (which mostly include sky and trees) and bottom 25 pixels (which mostly include car hood) are cropped off so that misleading information doesn't penetrate into the network (code line 140).

My **FINAL** model (lines 143 to 155 in code) is based on NVIDIA autonomous driving architecture (https://devblogs.nvidia.com/parallelforall/deep-learning-self-driving-cars/) and it consists of a 5 convolution neural network layers and four fully connected layers and a dropout layer in between. The model includes RELU layers after each convolution and fully connected layers to introduce nonlinearity.

Table 1: Final Model

Layer	**Function**	**Description**
	Input	160x320x3 raw RGB image
Lambda	Normalizing	Input normalizing: zero mean, unit variance
Cropping	Cropping2D	Crop-off top 70 and bottom 25 pixels. Output dimension: 65x320x3 RGB
Layer 1	Convolution 5x5x24	1x1 stride, 2x2 subsampling, outputs 31x158x24
	RELU	Rectified linear unit, outputs 31x158x24
Layer 2	Convolution 5x5x36	1x1 stride, 2x2 subsampling, outputs 14x77x36
	RELU	Rectified linear unit, outputs 14x77x36
Layer 3	Convolution 5x5x48	1x1 stride, 2x2 subsampling, outputs 5x37x48
	RELU	Rectified linear unit, outputs 5x37x48
Layer 4	Convolution 3x3x64	1x1 stride, outputs 3x35x64
	RELU	Rectified linear unit, outputs 3x35x64
Layer 5	Convolution 3x3x64	1x1 stride, outputs 1x33x64
	RELU	Rectified linear unit, outputs 1x33x64
	Flattening	outputs 2112 units
	Dropout	To reduce overfitting
Layer 6	Full connection	Input 2112, output 100
	RELU	Rectified linear unit, output 100
Layer 7	Full connection	Input 100, output 50
	RELU	Rectified linear unit, output 50
Layer 8	Full connection	Input 50, output 10
	RELU	Rectified linear unit, output 10
Layer 9	Full connection	Input 10, output 1
	Final Output	Final Output, 1 unit – The predicted steering angle

2. Attempts to reduce overfitting in the model

The model contains dropout layer in order to reduce overfitting (model.py line 150).

The model was trained and validated on different data sets to ensure that the model was not overfitting. The model was tested by running it through the simulator and ensuring that the vehicle could stay on the track.

3. Appropriate training data – for the final model

Training data was chosen to keep the vehicle driving on the road. I used a combination of center lane driving both in clockwise (2 loops) and anti-clockwise (2 loops) directions, recovering from the left and right sides of the road (1 clockwise loop), and some more data with smooth transitions at turns and along the bridge. I have also included the sample training data given by Udacity.

For details about how I arrived at the 'final' training data, see the next section.

Solution Design Approach

1. Evolving the training data towards the final training data

If the task was only to build the network model, this project would have been easy. Since the task involved getting an 'appropriate' training data myself along with a proper network model, this was the most challenging project.

First I got one loop of clockwise data and one loop of anti-clockwise data. I have used images from all three cameras in my training data. I have also included the sample training data given by Udacity. A 'Correction' parameter is added to center steering angle measurement to imitate that for the left camera and 'Correction' is subtracted from center steering angle to imitate that for the 'right' camera. I have also included data augmentation step by appending 'flipped' images.

To decide what training data is okay, I first started with a simple network model, based on the LeNet-5 architecture. I trained the LeNet network with a basic training data to predict steering angle with some accuracy. The car was able to move a little bit and weaved off later. Later I got few more training data with continuous recovery from the edges so that the network can 'clone' the recovery behaviour. This time the performance was a bit better but not at all to the expectation. Therefore I decided to change the network architecture itself – by going for the 'exact' NVIDIA architecture (https://devblogs.nvidia.com/parallelforall/deep-learning-self-driving-cars/). This time I also added plots to visualize training and validation losses.

When trained with NVIDIA network, the model was performing better, but was failing at the turns. Therefore I added a few more data with smooth transitions at the turns. Still the data could not train the model fully and car was failing while going along the bridge since the road boundaries here are different that the remaining sections of the track.

Next, I added more data of the car moving along the bridge, plus some more 'smooth' transitions along the turns, plus more recovery data to get the car back from weaving.

This time, the NVIDIA model was able to drive the car along the track successfully at 12 mph. However, if I increase the speed further, the car would weave off a bit. For more than 15 mph, it would fail at turns. I also found out that the training data need some form of transformation. The 'cv2.imread' function reads data in BGR format. But all image data should be in RGB format because this is what the network needs to be trained for since the drive.py works on RGB images. Therefore all training images are converted from BGR to RGB by:

```
image = cv2.imread(current_path)

#BGR to RGB conversion

image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
```

Then I decided to add dropout layers to regulate overfitting of the model. This layer is added between the convolution layers and the fully connected layers.

With this the car was able to drive along Track-1 without leaving any portion of the road. However there were still some spots where the car was touching the road boundary. This was taken care by properly tuning some of the hyper-parameters of the model (explained later).

So my final data for Track-1 included the following:

1. Two rounds of clockwise loop
2. Two rounds of anti-clockwise loop
3. One round of continuous recovery loop (an example time series is shown in Figure 2)
4. 3 smooth transitions at every turns
5. Get 1-4 for all 3 cameras (an example snapshot is shown in Figure 1)
6. Convert output of Step 5 from BGR to RGB
7. Augment output of Step 6 by flipping each image vertically
8. Randomly shuffle all images after Step 7, with 80% for training and 20% for validation. Histogram of training and validation data is shown in Figure 3.

At Step 5, I had 16000 (x3 from 3 cameras) training samples and 4000 (x3 from 3 cameras) validations samples.

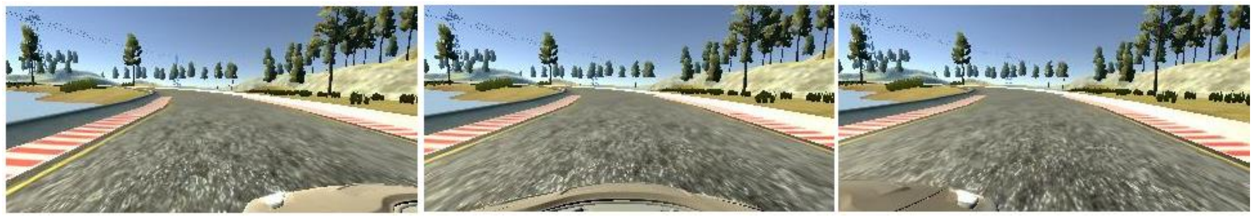

Figure 1: Left, center and right camera images

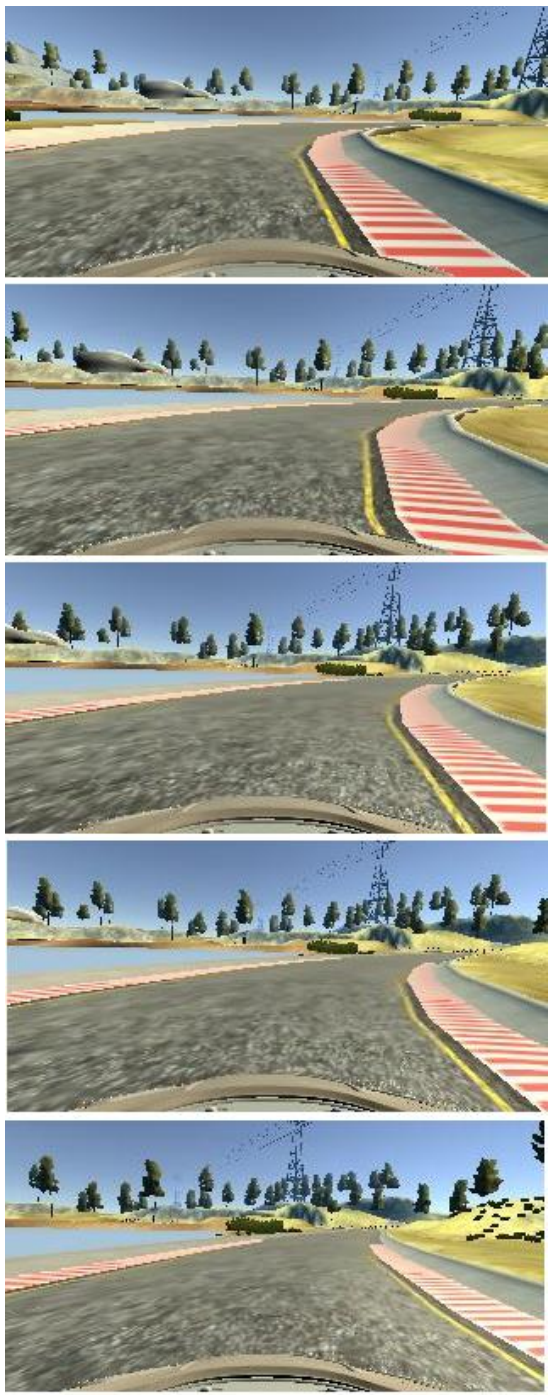

Figure 2: An example of recovery data - the car is purposefully wheeled towards the edge of the road and weaved back towards the center

Finally the training and validation data are given to the final model in **Table 1**.
A **generator** is used to get training and validation data to save computer memory (lines 134 and 135 in code)

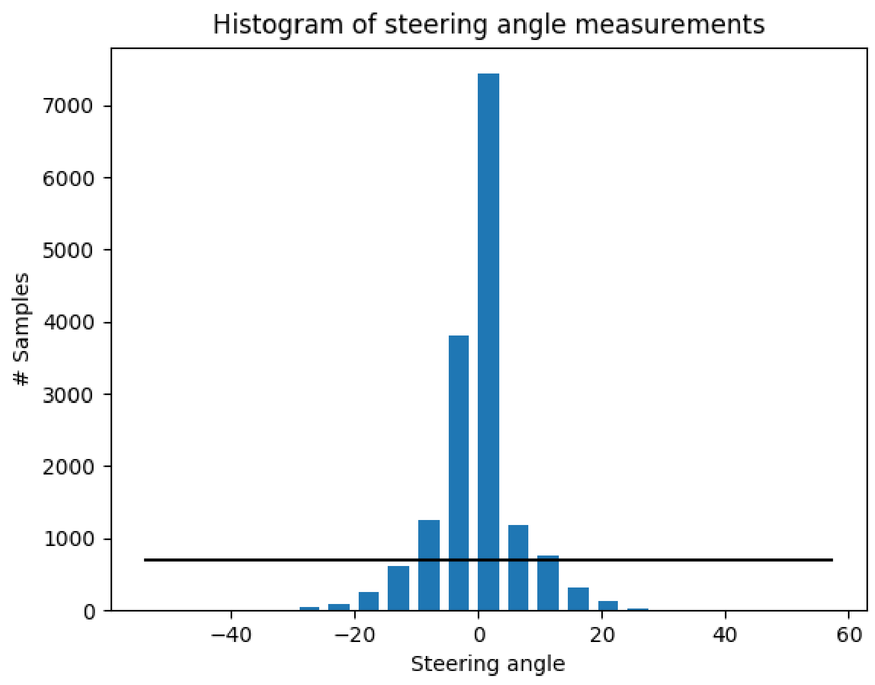

Figure 3: Histogram of Track-1 data only. This data is used to get model.h5

2. Model parameter tuning

Since there is still some turbulence, next logical step is to tune the parameters of the model to get the best drive possible on the tracks.

I used an **adam** optimizer so that manually training the learning rate wasn't necessary.

Next the '**Correction'** parameter for left and right steering angles were tuned between 0.15 to 0.25, and finally **0.22** was set.

The '**Keep Probability'** value for dropout layers was tuned between 0.5 to 0.8, and finally set as **0.6**.

The '**batch size'** was tuned between 32 to 256 and set as **128** finally.

The '**number of epochs'** was set as **20** in the final model.

The final training loss and validation loss are shown in Figure 4.

At the end of the process, the vehicle is able to drive autonomously around the track without leaving the road at 18 mph.

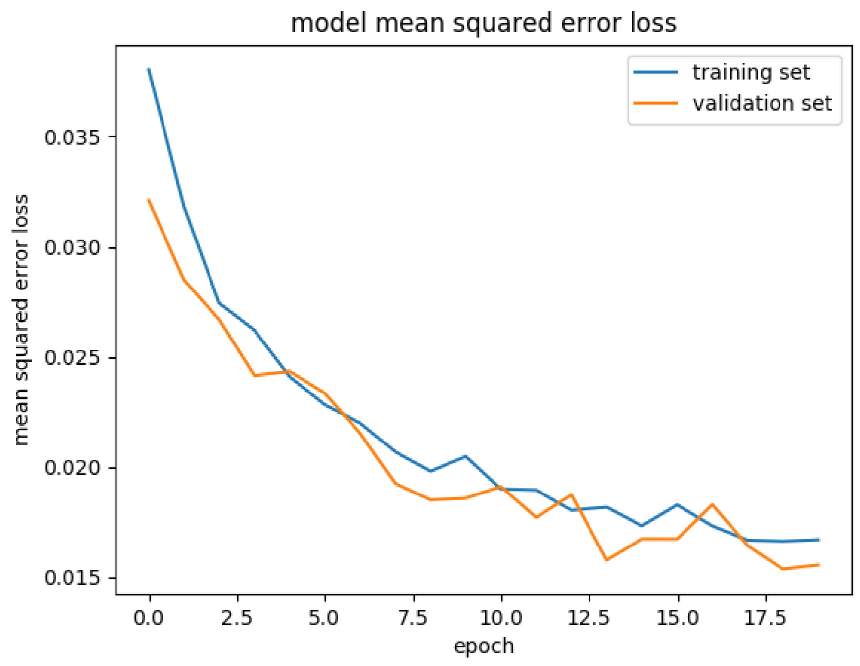

Figure 4: Training and validation loss for the final model ‘model.h5’ - shows no under or overfitting

Trying my model on Track-2

I collected Track 2 data same way as I did for track 1, that is, two rounds of clockwise data, 1 round of anti-clockwise data and 1 round of recovery data, and a few smooth transitions at sharp turns. Then this data was merged with Track-1 data, and then the whole chunk of data was used to train the model in Table-1. Parameters were tuned same as before (Overall, the same code is used, except that input data is different).

Histogram of training and validation data is shown in Figure 5. As can be seen here, the distribution is wide compared to Figure 3 as there are many sharp turns in Track-2. After 80% splitting, I had 32000 training data samples.

The final training loss and validation loss are shown in Figure 6.

With this data, **model_Track2_Track1.h5** was obtained. When I tested **Track-1** with this model, it worked only at 15 mph or below (that is, ‘set_speed’ in drive.py should be 15 or below). For any higher speed, it used to weave continuously (not jumping off the edges) along its trajectory. Weaving intensity increased with increase in speed. This could be explained as follows. The data also includes Track-2 data, which has many sharp turns and therefore diverse steering angles. Hence the model ‘model_Track2_Track1.h5’ would have learnt to change the angles more dynamically compared to the previous case (model.h5). Therefore, even though the Track-1 doesn’t have much sharp turns compared to Track-2, **model_Track2_Track1.h** was not good enough to drive on Track-1 beyond 15 mph. It was working fine for speeds below 15 mph without turbulence.

However, this model can successfully be used to drive on Track-2 at 12 mph (set_speed in drive.py should 12).

Finally I am happy that my model could work even on Track-2!!!!!

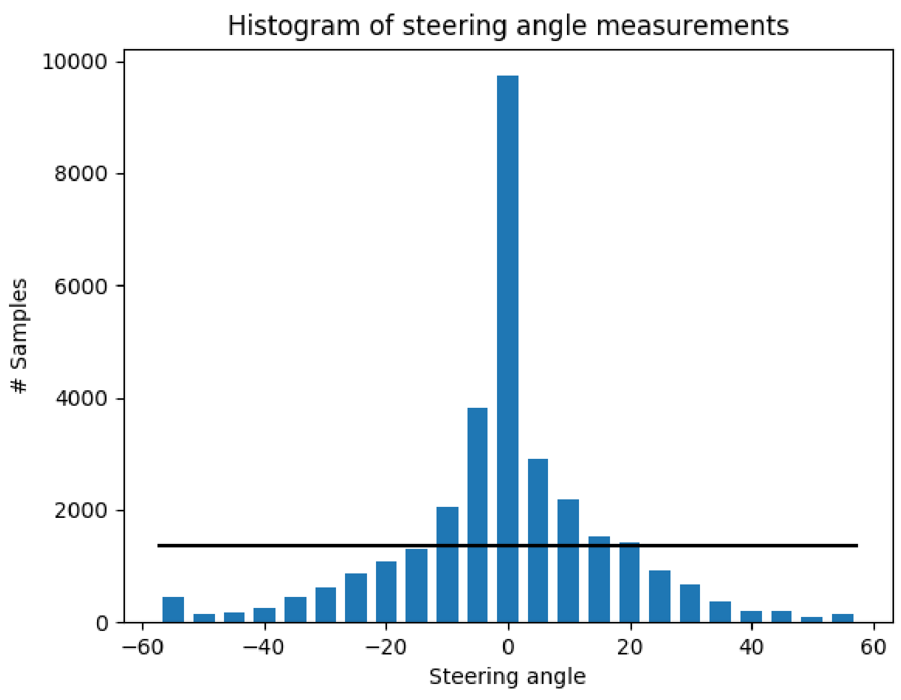

Figure 5: Histogram of Track1 and Track2 data. This data is used to get model_Track2_Track1.h5

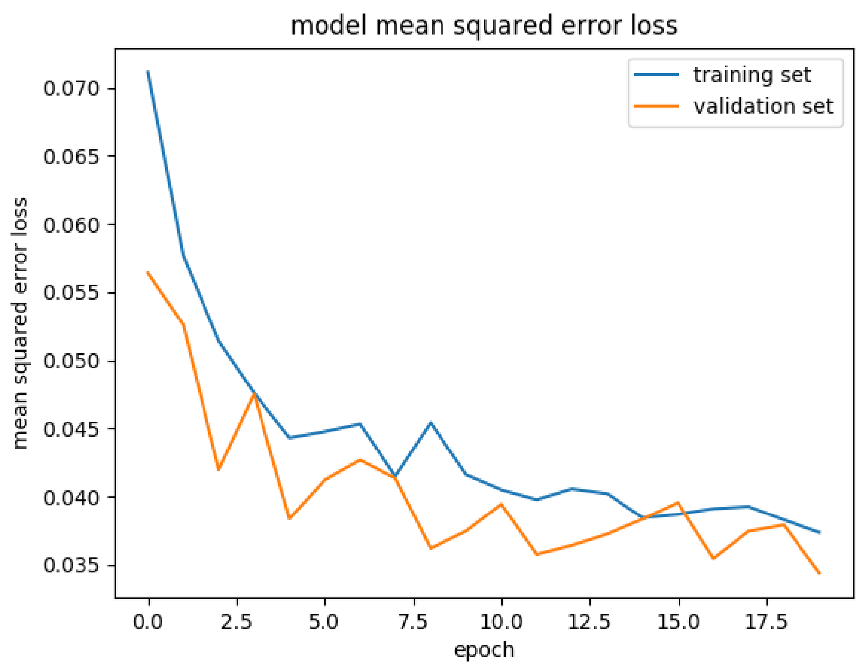

Figure 6: Training and validation loss for the final model 'model_Track2_Track1.h5' - shows no under or overfitting