Behavioural Cloning using Neural Network

By Jonas Chan

This document is written as part of the submission of the Udacity Self-Driving Car Course and contains detailed explanations of the approaches taken to obtain the final working weights.

Code Organisation

The code was initially written and tested using the iPython notebook. This submission include the following files:

1. Behavioral Cloning HTML.htm
   - As per the rubric from the previous project, I am including this as part of my submission as a proof that this work is original and has been done solely by myself.
2. Drive.py
   - This file contains the script to allow the vehicle to drive autonomously in the simulator. No changes were made to the file.
3. Model.h5
   - This file contains a trained convolutional network.
4. Model.ipynb
   - As described earlier, the code was initially written and tested using the iPython notebook. I used this method since I was familiar using the iPython notebook rather than any other methods.
5. Model.py
   - After ensuring that the code works in the iPython notebook, model.ipynb was exported as model.py as per the requirements of the project.
6. Video.mp4
   - This file contains the video of a successful autonomous driving around the track for 1 lap. To further ensure that the video is proof of original work, I have also uploaded another video on Youtube which captures the screen containing the car running in autonomous mode alongside the terminal. You may find the video by clicking on the following link:

     https://www.youtube.com/watch?v=rRiNt93x-aE

Using the submitted model

In order to test the functionality of the working convolutional neural network generated on my end using the simulator, execute the following syntax:

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

Machine specifications

The following lists the specification of the machine used to generate, train and test the submitted neural network:

Specification	Description
Machine	Macbook Pro
Processor	2.7 GHz Intel Core i5
Memory	8 GB 1867 MHz DDR3
Graphics	Intel Iris Graphic 6100 1536 MB

Code Architecture and Readability

The code written in model.py are refactored such that each specific methods are separated individually into specific functions. Each functions are commented to describe their purpose and their role in training the neural network. Since I exported the code from the notebook, I have made sure that the exported model (model.py) works by executing the following syntax:

```
python model.py
```

Since debug mode is set to true, running the model returns the printed messages in the terminal as well as some graphical representations of the data. To disable this feature, set the debug to False in line 24 in the model.

Model Architecture and Training Strategy

This section of the report describes the approaches taken in order to achieve the final model architecture. Please note that I have written this section in reverse chronology which simply means that the final architecture is described first and the process is described later.

Training Strategy & Final Model Architecture

The following describes the final model architecture pipeline used to train the neural network:

```
model = Sequential()
model.add(Lambda(lambda x: x / 255.0 - 0.5, input_shape = (160, 320, 3)))
model.add(Cropping2D(cropping=((50,20), (0,0)), input_shape=(160,320,3)))

model.add(Convolution2D(32,3,3, activation = "relu"))
model.add(MaxPooling2D(pool_size = (2,2)))

model.add(Convolution2D(32,3,3, activation = "relu"))
model.add(MaxPooling2D(pool_size = (2,2)))

model.add(Convolution2D(64,3,3, activation = "relu"))
model.add(MaxPooling2D(pool_size = (2,2)))

model.add(Flatten())
model.add(Dense(64))
model.add(Activation('relu'))
model.add(Dropout(0.5))

model.add(Dense(1))
```

Figure 1: Model architecture (Line 183 - 201)

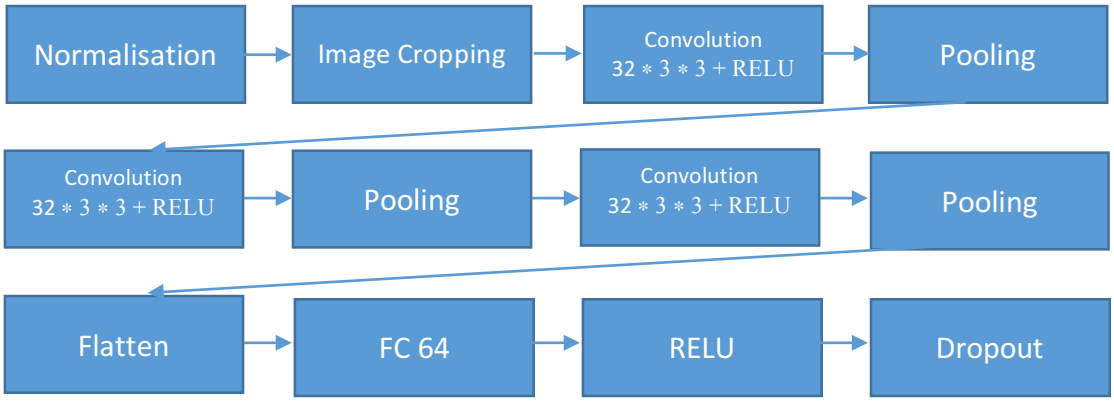

Figure 2: Final Model Architecture

Figure 2 illustrates a graphical description of the final model architecture for training the data. The architecture begins by implementing a lambda layer purposed to normalise the data. Then the data passes through a cropping layer which crops out portions of the images which are not helpful for training the data which can be illustrated in figure 3 and 4 below.

Figure 3: Before cropping

Figure 4: After cropping

This is followed by 3 convolutional blocks consisting of two 32 and one 64 filters of size 3 * 3. Each convolutional blocks had RELU activation function followed by a pooling layer of filter size 2 * 2. The RELU activation functions were implemented to introduce non-linearity in the architecture. Then, the model continues to a flattening layer, one fully connected layer, a RELU activation function and finally followed by a dropout layer in the attempt to reduce overfitting by progressively reducing the spatial size of the representation to reduce the amount of parameters and computation in the network. And finally, the architecture ends by going through a fully connected layer of size 1 since this is a regression. This architecture implements the adam optimiser hence the learning rate was not tuned manually.

Figure 5 illustrates the summary of the model I have implemented to train the data. This summary was obtained using the following syntax:

```
model.summary()
```

Layer (type)	Output Shape	Param #	Connected to
lambda_1 (Lambda)	(None, 160, 320, 3)	0	lambda_input_1[0][0]
cropping2d_1 (Cropping2D)	(None, 90, 320, 3)	0	lambda_1[0][0]
convolution2d_1 (Convolution2D)	(None, 88, 318, 32)	896	cropping2d_1[0][0]
maxpooling2d_1 (MaxPooling2D)	(None, 44, 159, 32)	0	convolution2d_1[0][0]
convolution2d_2 (Convolution2D)	(None, 42, 157, 32)	9248	maxpooling2d_1[0][0]
maxpooling2d_2 (MaxPooling2D)	(None, 21, 78, 32)	0	convolution2d_2[0][0]
convolution2d_3 (Convolution2D)	(None, 19, 76, 64)	18496	maxpooling2d_2[0][0]
maxpooling2d_3 (MaxPooling2D)	(None, 9, 38, 64)	0	convolution2d_3[0][0]
flatten_1 (Flatten)	(None, 21888)	0	maxpooling2d_3[0][0]
dense_1 (Dense)	(None, 64)	1400896	flatten_1[0][0]
activation_1 (Activation)	(None, 64)	0	dense_1[0][0]
dropout_1 (Dropout)	(None, 64)	0	activation_1[0][0]
dense_2 (Dense)	(None, 1)	65	dropout_1[0][0]

Total params: 1,429,601
Trainable params: 1,429,601
Non-trainable params: 0

Figure 5: Final model summary

Solution Design Approach for Deriving the Architecture

The ultimate strategy for deriving the model architecture had three main particular points which are as follows:

1. Low training time
2. Decreasing training loss values
3. Decreasing validation loss values

Initial attempts

My first attempt was of course to use the similar to the one in the tutorial as shown in figure 6 and the results of training the network is illustrated in figure 6 in which I thought was appropriate since I thought would help get me started which was rather disappointing because unlike the video, my simulation did not have issues with “glitching tyres” but rather it veered off straight into the lake!

```
model = Sequential()
model.add(Lambda(lambda x: x / 255.0 - 0.5, input_shape = (160, 320, 3)))
model.add(Convolution2D(6,5,5, activation = "elu"))
model.add(MaxPooling2D())
model.add(Convolution2D(6,5,5, activation = "elu"))
model.add(MaxPooling2D())

model.add(Flatten())
model.add(Dense(120))
model.add(Dense(84))
model.add(Dense(1))

model.compile(loss = 'mse', optimizer = 'adam')
model.fit(x_train, y_train, validation_split = 0.2, shuffle = True, nb_epoch = 5)

model.save('model.h5')
```

Figure 6: Initial model architecture

```
Starting convolutional processing...
Train on 12857 samples, validate on 3215 samples
Epoch 1/5
12857/12857 [==============================] - 756s - loss: 0.3017 - val_loss: 0.0126
Epoch 2/5
12857/12857 [==============================] - 681s - loss: 0.0101 - val_loss: 0.0123
Epoch 3/5
12857/12857 [==============================] - 666s - loss: 0.0087 - val_loss: 0.0124
Epoch 4/5
12857/12857 [==============================] - 671s - loss: 0.0078 - val_loss: 0.0126
Epoch 5/5
12857/12857 [==============================] - 677s - loss: 0.0071 - val_loss: 0.0126
```

Figure 7: Results from the initial model architecture

At this point, I made minor adjustments to it and tried out exponential RELU (ELU) since Vivek Yadav mentioned that it helps to make the transitions between the angles smoother. Based on the results above, it can be seen that the data is clearly highly overfitted since the training set had much lower mean squared error compared to the mean squared error on the validation set. To resolve this issue, I then implemented dropout to the architecture in the attempt to reduce overfitting which helped a little although the validation loss seemed to fluctuate as shown in figure 8. During this stage, the simulation managed to show some signs of progress where there were minor “glitches” in the tyres showing some sign of recovery to bring it back to the centre of the road.

[At this point, I also included side camera images, read more in the section below]

```
Starting convolutional processing...
Train on 14714 samples, validate on 3679 samples
Epoch 1/5
14714/14714 [==============================] - 953s - loss: 0.0222 - val_loss: 0.0152
Epoch 2/5
14714/14714 [==============================] - 844s - loss: 0.0162 - val_loss: 0.0143
Epoch 3/5
14714/14714 [==============================] - 836s - loss: 0.0151 - val_loss: 0.0142
Epoch 4/5
14714/14714 [==============================] - 847s - loss: 0.0143 - val_loss: 0.0145
Epoch 5/5
14714/14714 [==============================] - 851s - loss: 0.0138 - val_loss: 0.0126
```

Figure 8: Results from further attempts

I then proceeded to increase the depth of the filter to the one similar to that of the depth used by Vivek Yadav as well as to use RELU activation after one of the fully connected layers which then showed progress where overfitting was reduced. During this stage, the vehicle managed to drive through the entire track without leaving the road!

```
Train on 14714 samples, validate on 3679 samples
Epoch 1/5
14714/14714 [==============================] - 872s - loss: 0.0201 - val_loss: 0.0152
Epoch 2/5
14714/14714 [==============================] - 838s - loss: 0.0154 - val_loss: 0.0148
Epoch 3/5
14714/14714 [==============================] - 860s - loss: 0.0147 - val_loss: 0.0142
Epoch 4/5
14714/14714 [==============================] - 842s - loss: 0.0140 - val_loss: 0.0140
Epoch 5/5
14714/14714 [==============================] - 865s - loss: 0.0136 - val_loss: 0.0141
```

Figure 9: Results from final attempt

and the plot of the validation and training losses…

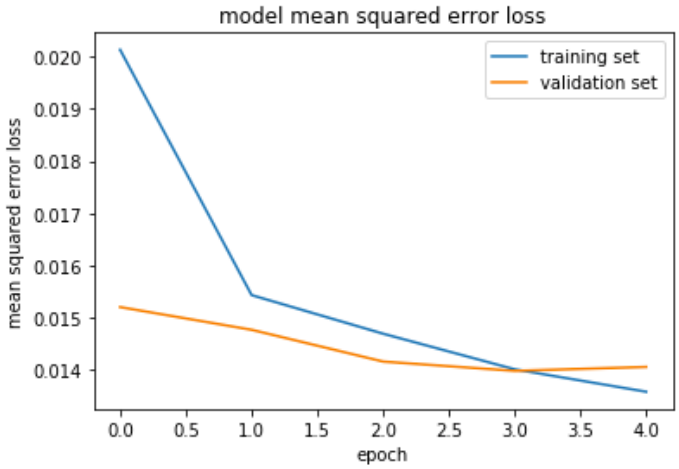

Figure 10: Plot of training and validation losses from the final attempt

Of course this step was paired with several other steps (described below) in order to achieve this.

Data Handling (How I obtained my training data) and Pre-Processing (How the training set was chosen and how did it affect the training process)

First and foremost, for each of the attempts, 80% of the total data were used for training and 20% of the total data used were used for validation.

The dataset used to train the neural network uses the images and steering angle measurements provided by Udacity. I have attempted many times to generate my own dataset and passing them through for training but the car seems to go off-course despite taking different driving methods; driving off-road and recover as well as driving from the other side of the road. Due to time constraints, I have used the provided data instead to train my network. Each tests were conducted by running them through the simulator in which the end goal is to drive the car through the track whilst keeping the vehicle on the road.

Initial Attempts

During my first attempt, I resorted to using just the data from the centre camera (12, 262 images) to train the network. However, unlike the tutorial, my simulation did not drive in a straight line but immediately drove off-course and straight into the lake (which is on the left side of the road)!

After watching the video multiple times, I realised that the data collected wasn't sufficient since the road is a left loop which meant that the vehicle only knows how to turn left! This took multiple tries by the way and also included using my own generated set of data which did not help!

Further attempts

In order to resolve this, I conducted only one pre-processing technique in order to increase the amount of data so the vehicle is able to generalise better. The only pre-processing technique I have used is flipping the images and adding them back to the original array which

contained the original set of images. In addition to that, I also used the side camera images in order to increase the amount of training images.

At this point, the vehicle was able to go through the road (somewhat smoothly with some minor off-road bumps with successful recoveries) until it reached a point in the road that had a junction without any road markings. At this point, the car did not turn left but went straight through it! What was surprising about this was that the vehicle continued driving along the dirt road and recovered back into the main road at the other end! (Sadly this wasn't part of the course and I would have stopped there if it was). After multiple tries with the same results, I was led to believe that the data now was slightly biased to turning to the right. So as part of the new attempt, I only took a certain percentage of the flipped images, randomised them and added them back into the original array. At this point, there was a small hint of success where the car started to turn left instead of right (into the dirt road).

After many attempts, I resorted to using 50% of the flipped images and including camera images with steering angle measurements of 10% which resulted in a total of 18, 393 images. The distribution of the final steering data used for training and validation is plotted into a histogram and is illustrated in Figure 11 and a random image taken from the dataset is illustrated in Figure 12. There is of course more to this but this is just a summary of my work that has been done.

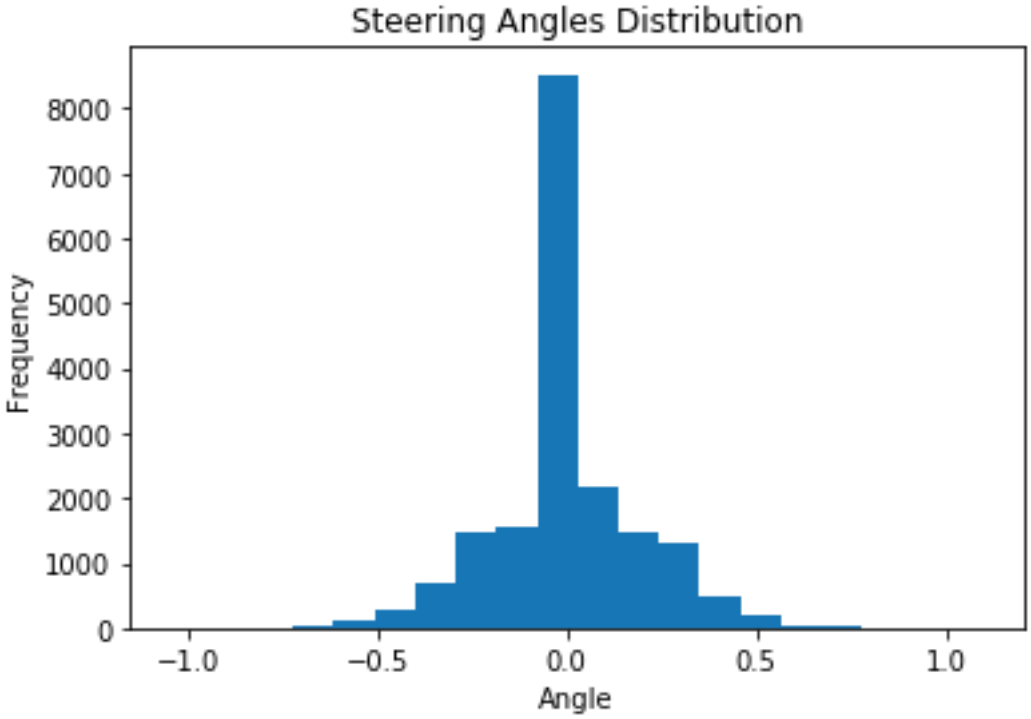

Figure 11: Final dataset distribution histogram plot

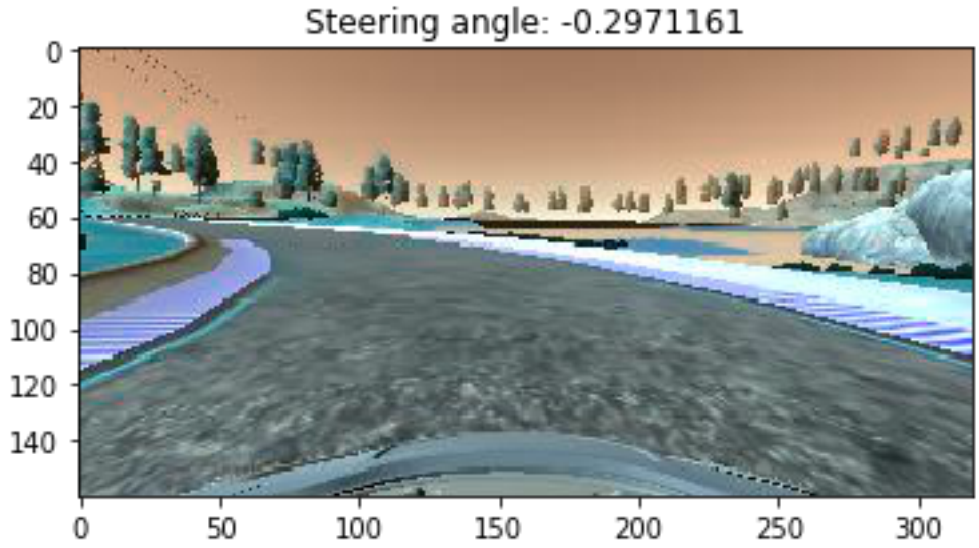

Figure 12: Random image and steering measurement obtained from the final dataset

Testing/ Autonomous Driving

Based on my testing, I realised that my final model performance differs every time it was retrained even with the same configuration which sometimes causes the vehicle to either drive off-course, drive up in the mountain or drive straight into the river. In addition to that, while running the saved trained neural network, I realise that it's performance is also affected by the quality of the simulator which meant that using different graphics settings could either lead to a success or a failure. To be fair as well as to eliminate additional factors caused by the simulator, I conducted the test with similar simulator settings; Resolution at $640 * 480$ and graphics quality set as "SIMPLE". In addition to that, I also conducted multiple tests and chose the model which performed well.

Further work

To be fair, I am kind of surprised that this simple model works even without any image augmentation techniques. My further work would include, using image augmentation techniques such as adding in shadows, brightness normalisation, etc (same techniques I used in the previous project) as well as trying out proper architectures and of course trying it out in track 2! I am stopping here for now due to time constraints but in the future, I will definitely do more of those now that I have the tools to work on it.