Behavioral Cloning

Udacity Project #4

By Devin Datt

Table of Contents

PROJECT 4: BEHAVIORAL CLONING

Project 4: Behavioral Cloning

Objectives

The goals / steps of this project are the following:

- Use the simulator to collect data of good driving behavior
- Build, a convolution neural network in Keras that predicts steering angles from images
- Train and validate the model with a training and validation set
- Test that the model successfully drives around track one without leaving the road

The deliverables of this project includes the following files:

- model.py - the script to create and train the model
- drive.py - file for driving the car in autonomous mode
- model.h5 - containing a trained convolution neural network
- run4-hd – HD video of front view drive of track 1

Quality of Code

Using the Udacity provided workspace simulator and my drive.py file, the car can be driven autonomously around the track by executing the command:

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

Since I was able to train my model with under 50,000 images I did not need to implement the use of a Python generator to generate data for training. I was able to store the full training data in memory on the GPU virtual server in the Udacity workspace.

Model Architecture and Training Strategy

Model Selection

I decided to use a network based on a Convolution neural network (CNN) layout with several layers of convolution followed by a series of flatten fully connected layers. CNN's are good for these types of tasks (image processing) since we want to feed in multiple somewhat similar images and thru 3 or more convolution layers we can exploit its unique ability to pick out distinct simple patterns at the input layers and consistently scale up its complexity in recognizing patterns. Then once a well convolved 'signature' is developed it passes thru a series of fully connected neurons and using gradient descent and back propagation to learn how that signature maps to the ground truth (ie. the actual steering angle).

I will attempt to use some base version of Nvidia CNN architecture that has the benefits previously described with a simple optimizer for gradient descent and default epoch settings. For the loss function I will use Mean Squared Error (MSE) as this a good loss measurement to minimize for regression problems like this one.

Once the base model was confirm to produce adequate loss results on validation and training I would work on adding normalization and regularization layers along with cropping techniques to ensure the data images are the easiest for the network to ingest.

Further details of the layers in the final convolutional network that I ended up will discussed in detail later on.

Overfitting

Initially, when I started to train the small dataset with a basic flatten layer I naturally saw the mean squared error was high on both a training and validation set. This was good indication that the model is underfitting (run 1). The easiest way I overcame this was to collect more data. When I started to increase the complexity of my model architecture I immediately noticed my validation loss was fluctuating up and down, which was a sign the model, might be trying to 'overfit' the data.

I found it very hard to prevent overfitting in this project, as almost every run of my model seemed to produce lower training losses then validation losses. When I started to input higher amounts of data and started to increase the complexity of my models I found that overfitting was natural outcome state of my models as my

mean squared error tended to be lower on my training set and higher on a validation set. To combat this overfitting I employed the following techniques:

- Train/validation splits have been used
- The model uses dropout layers to introduce randomness
- The model was trained and validated on many data sets (code line 37-41)
- The model was tested by running it through the simulator and ensuring that the vehicle could stay on the track.

Whenever I found my model had low mean squared error on the training and validation sets but the car was still driving off the track, I realized this could be because of the data collection process I was using. I found it extremely important to feed the network good examples of driving behaviour in order for the vehicle stayed in the center and recovered when getting too close to the sides of the road.

One thing I noticed was the 'quality' of the validation loss was not necessarily a great indication of the behavior of my car driving error free around the track. The loss significance seemed to only be contained relative to the other epochs within the same run. And one validation loss had no relative meaning to another run using another dataset, ie. run 2 might have higher validation loss but kept on the road as compared to run 1 which might have a lower validation loss. Doing some research (see Q&A link in resources) it seems that in order for the loss to become a significant measure across datasets and account for the vase complexity contented within the Udacity driving simulator we would need to drive hundreds or thousands of runs of various driving styles just on one track to collect all the 'meaningful' data. This was clearly not possible to complete in the time allotted for this project and would probably need more modeling and compute resources to compile a robust for all tracks.

Model Parameters

The model used an Adam optimizer which is a method for Stochastic optimization as a result the learning rate was not tuned manually (model.py line 118). A couple of reasons why this optimizer was a good choice are it is computationally efficient, has little memory requirements, is invariant to diagonal rescaling of the gradients, and is well suited for problems that are large in terms of data and/or parameters. Even though our model doesn't use high degree of parameters in its current state (ie. only a couple laps around the track) we can imagine this optimizer would play a greater role if we were to run the hundreds or thousands of laps that would be needed to fully capture the complexity of the simulator.

To get a comprehensive view of the parameter resources my model consumed to produce my results I used Keras model.summary function (line 118) to obtain the following output. The total number of parameters my model ended up using was nearly ~ 350,000.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lambda_1 (Lambda)            (None, 160, 320, 3)       0
cropping2d_1 (Cropping2D)    (None, 65, 320, 3)        0
conv2d_1 (Conv2D)            (None, 31, 158, 24)       1824
activation_1 (Activation)    (None, 31, 158, 24)       0
conv2d_2 (Conv2D)            (None, 14, 77, 36)        21636
conv2d_3 (Conv2D)            (None, 5, 37, 48)         43248
dropout_1 (Dropout)          (None, 5, 37, 48)         0
activation_2 (Activation)    (None, 5, 37, 48)         0
conv2d_4 (Conv2D)            (None, 3, 35, 64)         27712
conv2d_5 (Conv2D)            (None, 1, 33, 64)         36928
dropout_2 (Dropout)          (None, 1, 33, 64)         0
activation_3 (Activation)    (None, 1, 33, 64)         0
flatten_1 (Flatten)          (None, 2112)              0
dense_1 (Dense)              (None, 100)               211300
dense_2 (Dense)              (None, 50)                5050
dense_3 (Dense)              (None, 10)                510
dense_4 (Dense)              (None, 1)                 11
=================================================================
Total params: 348,219
Trainable params: 348,219
Non-trainable params: 0
_________________________________________________________________
```

Appropriate training data

There were a couple of ways to go about acquire data from Track 1 in order to use for training purposes of the model. We could use full dataset provided by Udacity, gather our own driving data or do both. I decided to train my network on as many datasets I could gather. I considered were the following ideas to ensure I got the highest quality of data from my simulator. I implemented the following steps:

Datasets:

- Use Udacity provided Track 1 images (~23K images)
- Drive three to four laps of center lane driving, as near center road as possible
- Drove one lap of recovery driving from the sides
- Drove one lap focusing on driving smoothly around curves
- Drove two laps in opposite direction on Track 1

Methods Used:

- Kept the car in the center of the road as much as possible
- Recovered back on track to center whenever the car veered off to the side
- Drove car counter-clockwise to generalize the model (mostly left turns here)
- flipping the images is a quick way to augment the data
- Collecting data from the second track can also help generalize the model
- Avoid over fitting or under fitting when training the model
- Knowing when to stop collecting more data

- Training data has been chosen to induce the desired behavior in the simulation (i.e. keeping the car on the track).
Training data was chosen to keep the vehicle driving on the road. I used a combination of center lane driving, recovering from the left and right sides of the road ...

Additional steps to 'cleanse' the data I choose to crop a portion from the top (ie. 75H x 320W) for the sky, mountains, and trees… and a portion at the bottom (ie. 25H x 320W) which roughly represents the hood of the car.

For details about how I created the training data, see the next section.

Architecture and Training Documentation

Solution Design Approach

Here we will discuss the approach I took to design my model architecture to predict steering angles that would allow the car to drive autonomously around track 1 in the simulator.

The approach I decided on would be to follow some suggestions and first start with small datasets and fit a basic model (see resources for reference cheat sheet). Once model was working then incrementally add some elements to my model network architecture and then change up by training with different types of datasets (track 1 one-way, track 1 opposite way,…etc.). Each time adding steps to process a neural network incrementally in an effort to reduce the training and validation loss with the goal not to overfit our model too much.

Next task was to shore up the robustness of a base Nvidia convolution neural network model that I chose to start with. These would be implemented with starting with kernel size of 5x5 and dropping them down to a 3x3 in the lower convolutions. The amount of filters used in these convolutions ranges from 36 in the first layer to 64 in the 5^{th} layer (model.py code line 101-106)

I introduced nonlinearity into my models with basic 'RELU' activation function multiple times thru out my model (line 100-109). I introduced some randomness in between my model using Dropout layers. This allowed my model to truly validate the aspects of the images that mattered (lines 99, 103,108)

A few other tricks I used to help the model 'learn' better was improve the procurement of the incoming data so it was normalized and centered. I did this with using a Keras Lambda layer to allow gradient descent functions to find the easiest way to minimize the loss function. The other way to help ease the job of the model to better learn was only let the convolution layer 'see' the parts of the image that mattered by using Keras Cropping2D (line 94).

I randomized and then split the data after collecting and mutating all incoming images into X_train and X_validation datasets. This was done by splitting data by a 80/20 ratio to validate and model that was trained. I used MSE (Mean Squared Error) as my error loss measurement that I tried to minimize with gradient descent Adam optimizer.

The final step was to run the simulator to see how well the car was driving around track one. There were a few spots where the vehicle fell off the track or clearly jump the bank into the hill or water. To improve this I run the model on recovery data with special attention to those specific spots on the road the car had trouble handing in autonomous mode.

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

Final Model Architecture

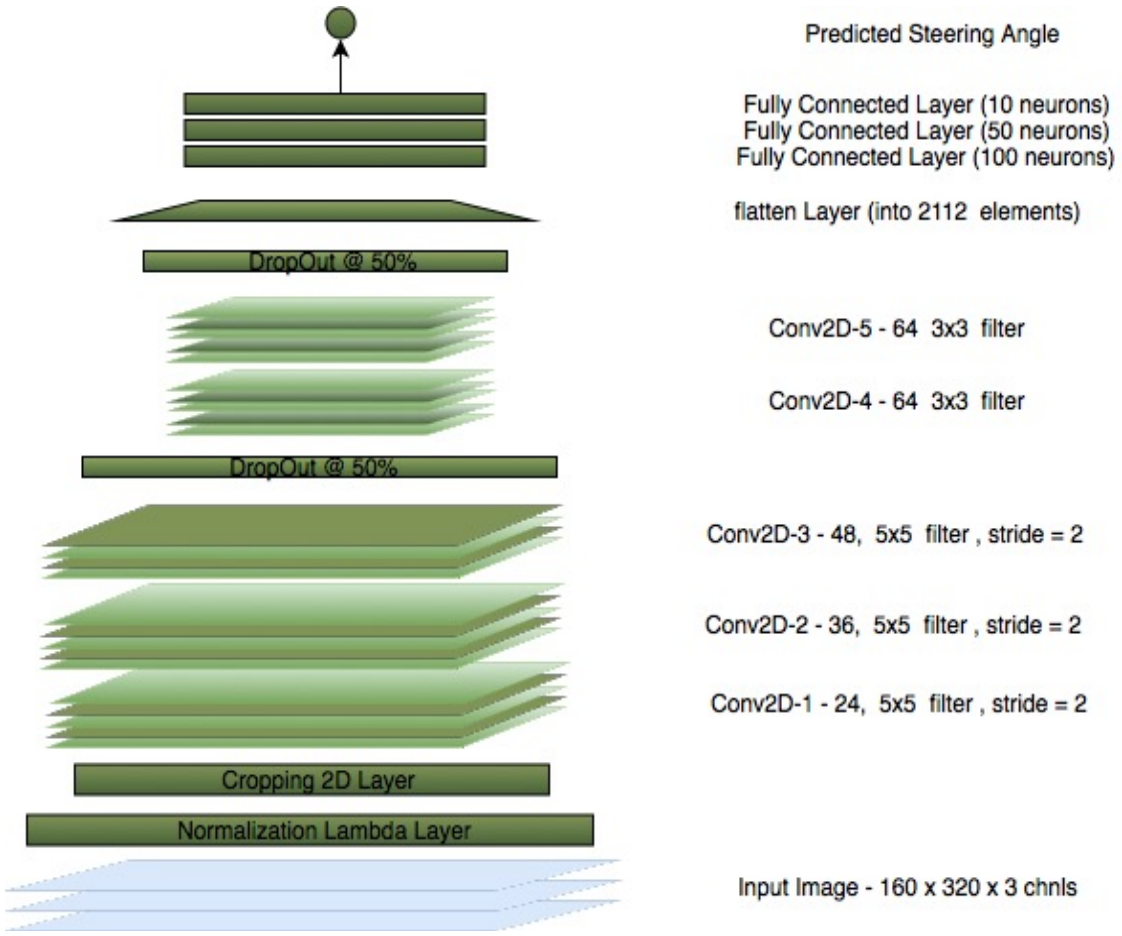
The details of the characteristics and qualities of my CNN architecture, such as the type of model used, the number of layers, the size of each layer are shown below in the following code and visualization.

The final model architecture (model.py lines 89-114) consisted of a convolution neural network with the following layers and layer sizes

```
model = Sequential()

model.add(Lambda(lambda x: x / 255.0 - 0.5,
     input_shape=(160,320,3)))

model.add(Cropping2D(cropping=((70,25),(0,0))))

model.add(Conv2D(24, kernel_size=(5,5), strides=(2,2),
     input_shape=(160,320,3)))

model.add(Activation('relu'))

model.add(Conv2D(36, kernel_size=(5,5), strides=(2,2),
     activation='relu'))
model.add(Conv2D(48, kernel_size=(5,5), strides=(2,2),
     activation='relu'))
model.add(Dropout(0.5))

model.add(Activation('relu'))

model.add(Conv2D(64, kernel_size=(3,3), activation='relu'))

model.add(Conv2D(64, kernel_size=(3,3)))
```

```
model.add(Dropout(0.5))

model.add(Activation('relu'))

model.add(Flatten())

model.add(Dense(100))

model.add(Dense(50))

model.add(Dense(10))

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

Here is a better visualization of the architecture I described previously with the filter size and



Creation of the Training Set & Training Process

Below describes the approach I followed when developing my training data and the subsequent process I used to train my neural network. The pipeline code and the results are recorded in Appendix A:

0) **Initial Run:** My initial run was to first collect data by driving as smoothly as possible down the middle on Track 1 once. I took corners as slow as needed to ensure the car didn't go off the road. Going slow also had a benefit of capturing more images per second that showed angles I wanted the model to 'see' and learn while going around difficult curves.

1) **Simple Network:** My next design was small dataset of 60 images divided evenly with negative, positive, and zero steering angles. I then pumped that data into the most basic network consisting of one Flatten layer followed by one Dense (1) layer to get one result. to test functionality with one Flatten layer with Adam optimizer and 10 epochs. As expected both training and validation loss was massive.

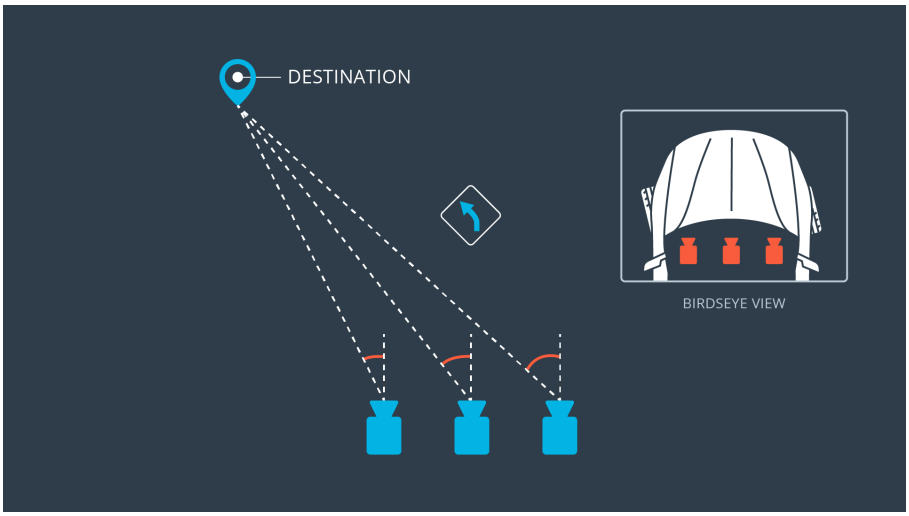
2) **More Dense Layers:** Since the loss was huge my next design consisted of the same 60 images and 1x Flattening but changed to 3x Dense layers (120, 84, 1 neurons). All this still using the Adam Optimizer but changed to five (5) epochs as I found the loss didn't seem to get better after five epochs, and in most cases got a lot worse. Still training and validation loss was massive.

3) **Lambda Layer:** Add a Lambda layer to normalize image pixels (by divide by 255) and Mean Centering the data (by subtracting by 0.5). This allowed us to concentrate and generalize the data around the center axis, which allows minimizing the loss gradient much easier.

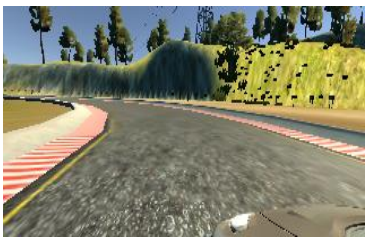
4) **More Convolutions:** Next added 2 convolutional layers one with 6 layers, 5x5 filter, with 'relu' activation. Another was 16 layers, 5x5 filter with 'relu'. These layers take advantage of finding patterns from input data. They stack them to make abstract concepts by these **convolution** layers. This reduced the MSE loss significantly.

5) **Flip Images:** Add 'flipped' images doubling amount of data points and artificially providing 'balanced' data to the network to learn from only making mostly left turns on Track 1 (ie. a counter-clockwise track).

6) **Left & Right Angles:** Add ‘left’ and ‘right’ camera images tripling the amount of data points. This provided more granular data on different position views on the track. Since we didn’t have exact left and right measurement data for these cameras we simply added or subtracted a small correction factor from the base center came angle.



Here is a sample of the three camera angles from the same position on track 1.



7) **Recovery Data:** Run new 42K images from Recovery data. This data is taken by the camera only recording when the car starts off or near off to the side of the road and corrects the car back to the middle with the corresponding compensated steering angle.

8) **Opposite Run:** Run new 55K images of opposite direction on Track1. This additional data provide somewhat new track data as it was

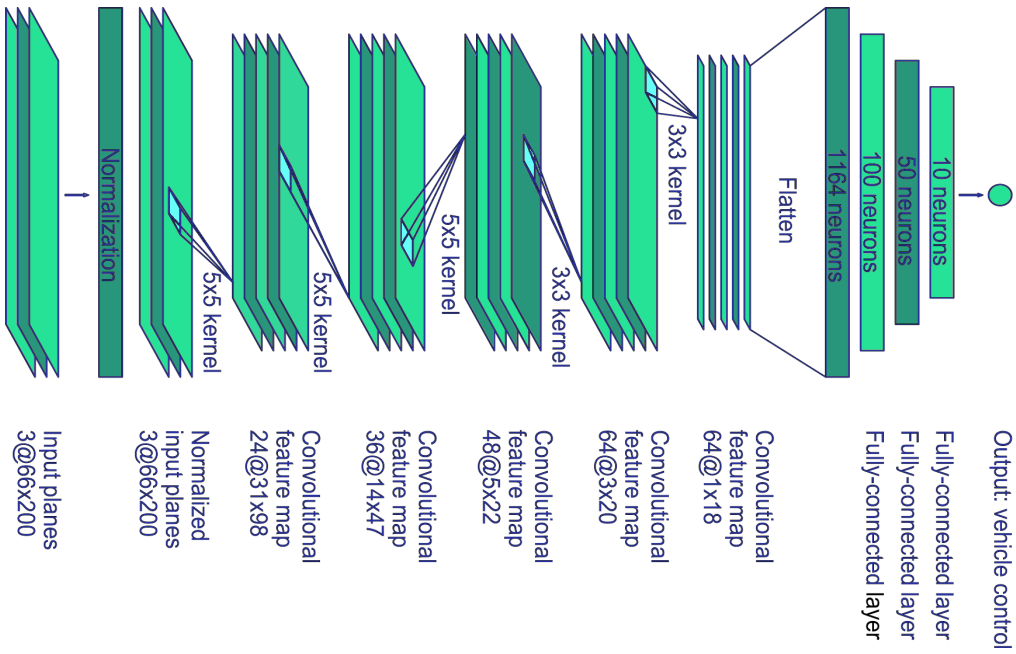
theoretically like driving on a new track. This helped with additional balancing data as this 'track' was in clockwise direction. Note this data was also subject to the adding left and right camera images and 'flipping' the images to 'balance' the balance data.

9) **Crop Images:** Used Keras 'Cropping2D' function to crop 70 & 25 pixels from the top and bottom respectively. This helped with reducing the parts of the image that the convolution actually needs to pay attention and not be distracted by unimportant trees, hills, skies, grass….etc.

Before Cropping

After Cropping

10) **Nvidia Network:** Next I implemented the Nvidia convolutional architecture (5 convNets + 4 fully connected layers, see the following image for a generic version). This design added additional layers of convolution to dive deeper to extract finer patterns in the images. And the additional fully connected layers allowed more permutations and parameters to better map to the cars steering angle with those images.



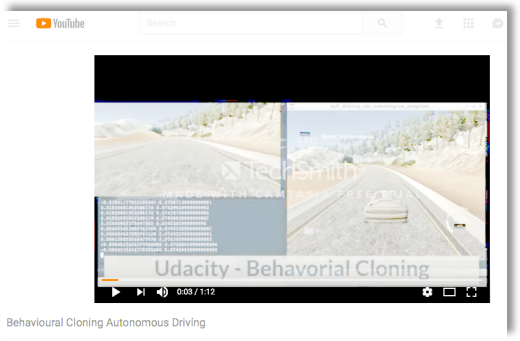
11) **Track 2**: I was going to add more data by driving on track 2 but I found collecting data extremely difficult to complete 'good' data without consistently driving off the road. It seems that it is more difficult to also control the steering angle and the throttle, break while at the same time negotiating the uneven terrain of the road that is althrough Track 2.

Simulation

As you can see from the video no tire left the drivable portion of the track surface. The car did veer off into the 'side paint' on some runs but most of the time it stayed in the 'safe zone'. The infrequency on capturing a great drive vs. a good drive was due to the randomness infused in using the Dropout rate of 50% or above. It seems using Dropout didn't guarantee I would never receive an imperfect drive (at least for me).

Nonetheless, my final drive model predicted steering angles that did not allow the car to pop up onto ledges or roll over any surfaces that would otherwise be considered unsafe (if humans were in the vehicle).

You can see in the copy of my driving video at this following YouTube link:



https://www.youtube.com/watch?v=TF7RxbQL_kc

This output is also captured in the video file named ‘run4-hd’ that was attached with this submission.

Project Bonus

I did attempt to use my trained model that learned from track 1 in Autonomous mode on Track 2, but I was not able to simulate the same behavioral cloning affects. As a result the model was not able to make my car stay on the road for this difficult track. One main reason for this is that I was not able to collect and train my model on any data on Track 2. Clearly, the result I got was expected.

Resources

Behavioral Cloning Cheatsheet
https://s3-us-west-1.amazonaws.com/udacity-selfdrivingcar/Behavioral+Cloning+Cheatsheet+-+CarND.pdf

Udacity Q&A tutotial Videos
https://www.youtube.com/watch?v=rpxZ87YFg0M&list=PLAwxTw4SYaPkz3HerxrHlu1Seq8ZA7-5P&index=4&t=0s

Cornell University Library – Adam Optimizer
https://arxiv.org/abs/1412.6980v8

Appendix A

Run #1: Initial run with only 3 images with basic flatten, dense layer:

```
Epoch 1/10
2/2 [==============================] - 0s 83ms/step - loss: 16804.0938 - val_loss: 447816512.0
Epoch 2/10
2/2 [==============================] - 0s 4ms/step - loss: 374906848.0000 - val_loss: 30377940.0
Epoch 3/10
2/2 [==============================] - 0s 4ms/step - loss: 24457790.0000 - val_loss: 91312656.00
Epoch 4/10
2/2 [==============================] - 0s 4ms/step - loss: 79831168.0000 - val_loss: 249584144.00
Epoch 5/10
2/2 [==============================] - 0s 4ms/step - loss: 215675008.0000 - val_loss: 188839216.00
Epoch 6/10
2/2 [==============================] - 0s 4ms/step - loss: 163637856.0000 - val_loss: 53569968.00
Epoch 7/10
2/2 [==============================] - 0s 4ms/step - loss: 47277808.0000 - val_loss: 351307.6875
Epoch 8/10
2/2 [==============================] - 0s 4ms/step - loss: 169856.8906 - val_loss: 55100648.00
Epoch 9/10
2/2 [==============================] - 0s 3ms/step - loss: 44940584.0000 - val_loss: 122922304.0
Epoch 10/10
2/2 [==============================] - 0s 3ms/step - loss: 101581376.0000 - val_loss: 121036568.0
```

Result: Car goes in a right continuous circle turn.

Run #2 : Using 60 images with Flatten + Dense layer with & epochs:

```
48/48 [==============================] - 0s - loss: 71091690.1354 - val_loss: 5062826.0000
Epoch 2/10
48/48 [==============================] - 0s - loss: 34900137.0000 - val_loss: 161365344.0000
Epoch 3/10
48/48 [==============================] - 0s - loss: 114933238.6667 - val_loss: 139792.3125
Epoch 4/10
48/48 [==============================] - 0s - loss: 11858077.0833 - val_loss: 98581616.0000
Epoch 5/10
48/48 [==============================] - 0s - loss: 74130096.0000 - val_loss: 24401796.0000
Epoch 6/10
48/48 [==============================] - 0s - loss: 12199188.0833 - val_loss: 21639568.0000
Epoch 7/10
48/48 [==============================] - 0s - loss: 29678626.6667 - val_loss: 36979008.0000
Epoch 8/10
48/48 [==============================] - 0s - loss: 28364493.3333 - val_loss: 360366.4375
Epoch 9/10
48/48 [==============================] - 0s - loss: 3861583.4115 - val_loss: 33841800.0000
Epoch 10/10
48/48 [==============================] - 0s - loss: 25025683.3333 - val_loss: 12006364.0000
```

Result: Failed - Car goes in a right continuous circle turn.

Run #3: 2 convolutional layers (6 & 16) each with 5x5 filter, with ‘relu’ activation.

```
Train on 96 samples, validate on 24 samples
Epoch 1/5
96/96 [==============================] - 4s 42ms/step - loss: 2692.9988 - val_loss: 1390.3322
Epoch 2/5
96/96 [==============================] - 3s 30ms/step - loss: 975.7389 - val_loss: 2019.8721
Epoch 3/5
96/96 [==============================] - 3s 30ms/step - loss: 1126.7056 - val_loss: 17.8548
Epoch 4/5
96/96 [==============================] - 3s 29ms/step - loss: 631.0599 - val_loss: 233.2281
Epoch 5/5
96/96 [==============================] - 3s 30ms/step - loss: 459.7389 - val_loss: 31.8012
```

Run #4: Output of 1x Flatten with 3 Dense (120,84,1) layers with 5 epochs:

```
Epoch 1/5
48/48 [==============================] - 3s 56ms/step - loss: 4222.4208 - val_loss: 5001.3457
Epoch 2/5
48/48 [==============================] - 2s 35ms/step - loss: 7428.3905 - val_loss: 1.6339
Epoch 3/5
48/48 [==============================] - 2s 34ms/step - loss: 2740.8566 - val_loss: 1349.6766
Epoch 4/5
48/48 [==============================] - 2s 34ms/step - loss: 4741.2178 - val_loss: 138.7078
Epoch 5/5
48/48 [==============================] - 2s 34ms/step - loss: 817.4260 - val_loss: 338.1927
```

Result - Actually this made the both losses

Run #5: Added ‘flipped’ images increasing the dataset:

```
38572/38572 [==============================] - 98s - loss: 1.0075 - val_loss: 0.0149
Epoch 2/5
38572/38572 [==============================] - 93s - loss: 0.0108 - val_loss: 0.0115
Epoch 3/5
38572/38572 [==============================] - 94s - loss: 0.0084 - val_loss: 0.0116
Epoch 4/5
38572/38572 [==============================] - 93s - loss: 0.0073 - val_loss: 0.0120
Epoch 5/5
38572/38572 [==============================] - 94s - loss: 0.0066 - val_loss: 0.0125
```

Result: Car is actually able to drive thru several bends in the road but crashes at first major left turn.

Run #6: Added ‘left’ & ‘right’ images increasing the dataset:

38572/38572 [==============================] - 51s - loss: 0.3938 - val_loss: 0.0126
Epoch 2/5
38572/38572 [==============================] - 47s - loss: 0.0113 - val_loss: 0.0117
Epoch 3/5
38572/38572 [==============================] - 47s - loss: 0.0103 - val_loss: 0.0120
Epoch 4/5
38572/38572 [==============================] - 47s - loss: 0.0098 - val_loss: 0.0122
Epoch 5/5
38572/38572 [==============================] - 47s - loss: 0.0095 - val_loss: 0.0122

Run #7: Run new 42K images, ‘flipped’ and ‘left’ & ‘right’ images increasing the dataset:

42336/42336 [==============================] - 112s - loss: 1.6638 - val_loss: 0.0109
Epoch 2/5
42336/42336 [==============================] - 103s - loss: 0.0039 - val_loss: 0.0035
Epoch 3/5
42336/42336 [==============================] - 104s - loss: 0.0015 - val_loss: 0.0025
Epoch 4/5
42336/42336 [==============================] - 104s - loss: 0.0010 - val_loss: 0.0023
Epoch 5/5
42336/42336 [==============================] -104s -loss: 8.3441e-04 - val_loss: 0.0024

Result: okay driving but still runs off the road

Run #8: Run on 55K of opposite direction on Track 1

run #1
55382/55382 [==============================] - 144s - loss: 0.8000 - val_loss: 0.0073
Epoch 2/5
55382/55382 [==============================] - 136s - loss: 0.0029 - val_loss: 0.0039
Epoch 3/5
55382/55382 [==============================] - 136s - loss: 0.0018 - val_loss: 0.0034
Epoch 4/5
55382/55382 [==============================] - 136s - loss: 0.0015 - val_loss: 0.0024
Epoch 5/5
55382/55382 [==============================] - 136s - loss: 0.0012 - val_loss: 0.0022

run #2
55382/55382 [==============================] - 71s - loss: 0.0126 - val_loss: 0.0174
Epoch 2/5
55382/55382 [==============================] - 64s - loss: 0.0074 - val_loss: 0.0176
Epoch 3/5

55382/55382 [==============================] - 64s - loss: 0.0054 - val_loss: 0.0140
Epoch 4/5
55382/55382 [==============================] - 64s - loss: 0.0044 - val_loss: 0.0135
Epoch 5/5
55382/55382 [==============================] - 63s - loss: 0.0036 - val_loss: 0.0117

Result: Model is slightly overfits. Car drives about a minute and crashes into water.

Run #9: Run on 18K and added 2x Dropout (50%)

14395/14395 [==============================] - 19s - loss: 0.0161 - val_loss: 0.0144
Epoch 2/5
14395/14395 [==============================] - 17s - loss: 0.0060 - val_loss: 0.0155
Epoch 3/5
14395/14395 [==============================] - 17s - loss: 0.0043 - val_loss: 0.0163
Epoch 4/5
14395/14395 [==============================] - 17s - loss: 0.0035 - val_loss: 0.0177
Epoch 5/5
14395/14395 [==============================] - 17s - loss: 0.0030 - val_loss: 0.0182

33595/33595 [==============================] - 40s - loss: 0.0172 - val_loss: 0.0153
Epoch 2/5
33595/33595 [==============================] - 38s - loss: 0.0115 - val_loss: 0.0162
Epoch 3/5
33595/33595 [==============================] - 38s - loss: 0.0098 - val_loss: 0.0129
Epoch 4/5
33595/33595 [==============================] - 38s - loss: 0.0087 - val_loss: 0.0133
Epoch 5/5
33595/33595 [==============================] - 38s - loss: 0.0077 - val_loss: 0.0123