Behavioral Cloning Project

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
- Summarize the results with a written report

Rubric Points

Here I will consider the rubric points individually and describe how I addressed each point in my implementation.

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.hd5** containing a trained convolution neural network
   - **writeup_report.pdf** summarizing the results
   - **run_center_cam_60fps.mp4** center camera view of autonomous run captured using video.py
   - **run_3rdperson_15fps.mp4** 3rd person view of autonomous run captured using kazam

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

   > **python drive.py model.h5**

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**

   Model is implemented in the file **model.py**, function **networkModel()** : lines **108-168.** Details about the architecture is provided in the following section.

2. **Attempts to reduce overfitting in the model**
   The model contains dropout layers in order to reduce overfitting.
   Drop out is done in the Dense layers (layers 7, 8) (model.py, lines147-159).

   The model was trained and validated on different data to ensure that it was not overfitting (model.py, line-180).

   The model was tested by running it through the simulator and ensuring that the vehicle could stay on the track. Used keras model.fit(…) function to split train and validation set as (80%-20%) respectively. (model.py, line 191)

3. **Model parameter tuning**
   The model used an Adam optimizer, so the learning rate was not tuned manually (model.py, line 188). Batch size was set as 256 and trained for 100 epochs using Keras and backend as Tensorflow

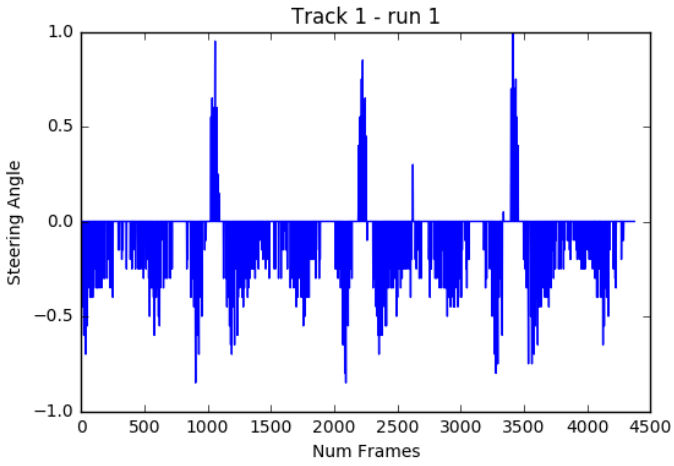
4. **Appropriate training data**
   Training data was chosen to keep the vehicle driving on the road. I used a combination of center lane driving, reverse track driving, recovering from the left and right sides of the road, weaving through straight roads.
   For details about how I created the training data, see the next section.

Model Architecture and Training Strategy

1. **Solution Design Approach**

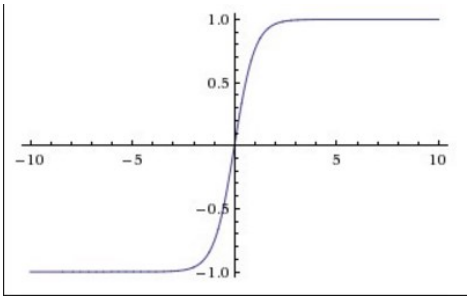
The overall strategy for deriving model architecture was to figure out how to map captured images and steering angle from Train Run by designing a convolution neural network. Convolution neural networks are very effective in learning image features on its own provided enough training data is fed to the network with proper hyper parameter tuning. Popular networks such as AlexNet, GoogLeNet, ResNet, VGG are great in object classification of 1000's of classes, but I felt that these are very big networks for this solution. I wanted to create a small enough network which can train faster and be robust enough to handle corner cases such as recovery, sharp turns etc. For Track 1 particularly, there are no other vehicles on the road, no pedestrians, cyclists, traffic signs, traffic lights etc. Also the elevation of the road is zero and the road bends into nicely manageable curves. These were motivation enough to build a simple network to accept input images and predict steering angles. When we run on actual roads with many vulnerable users, lane markings, traffic rules etc… It's important to start with a pre-trained network and fine tune for a given problem.

Firstly I wanted to drive a couple of laps in Track1 and visualize how the steering angles are captured. Even though the game window shows a range of -25 (left) to +25 (right) the actual value being recorded is normalized to a range of -1(left) and +1(right) as shown below,



This is good because the tanh activation function has a range of -1 to +1 to any value thrown at it.

$$f(x) = \tanh(x) = \frac{2}{1 + e^{-2x}} - 1$$

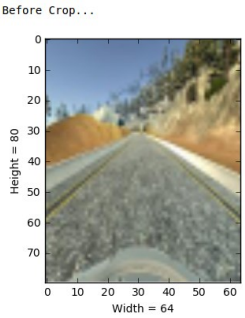


(Source: Wikipedia)

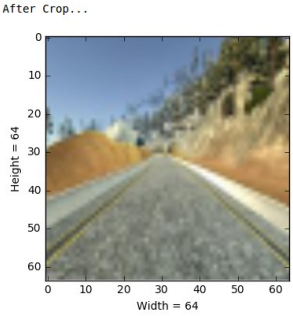
Next I had to take care of input dimensions. When we record in training mode, the image which gets captured is of resolution (160 rows and 320 columns). This is a big image to be fed into a tiny network so I downsized the input to (80 rows and 64 columns) and cropped bottom 16 rows to remove the hood. These conversions changes the aspect ratio of the input image fed to the network

Original

Resized



Cropped



Once the input image dimensions and output value is defined, the important work was to build a simple enough network in between. The network should be quick and also robust enough to recover from corners and have good generalization and avoid overfitting. Details of trial runs input preprocessing techniques and network detail are described in next points.

In order to gauge how well the model was working, I split my image and steering angle data into a training and validation set. During initial runs, keeping network structure the same, the mean-squared-error loss for the train set was pretty close to zero (loss = 0.0001), but the validation error was pretty high (0.06). This indicated that the network might have over-fit and needs more data to generalize. Also to make the connections more robust I introduced drop out layers in Dense connections and set the connection keep probability at 50%.

Post this the validation loss reduced to (loss= 0.032). This was also by adding more training samples such as driving in reverse, weaving through straight lines etc. Post this the model was saved as ‘model.hd5’

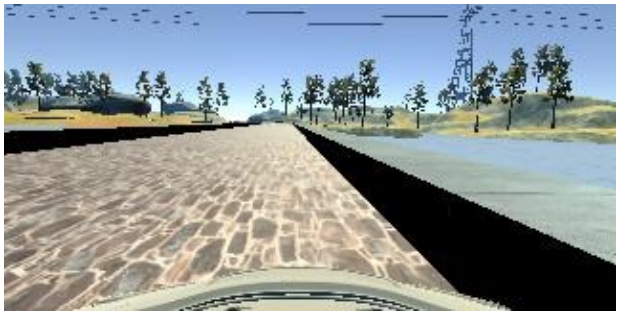
Next step was to try out the autonomous mode in simulator by running

> python drive.py model.hd5

At this point, the car was able to take the turns pretty confidently but at one particular place, while crossing the bridge, the car entered the bridge with a skewed angle, hit the wall of the bridge and got stuck. This happened because the network was not trained for “Recovery”.

To fix this, I recorded multiple sequences where the car tries to move away from the bridge wall and into the center lane. Care was taken, not to include scenes where car is approaching the boundary of wall, as this was not normal behavior. Below are some images where the car starts at boundary wall and moved away, towards the center of bridge.

1



2

3

4

Care was taken to modify drive.py file by resizing the input from (160x320) to (80x64) and cropped to (64x64) before feeding to the model (drive.py, lines: 71-74)

At the end of the process, the vehicle is able to drive autonomously around the track without leaving the road. Especially on the bridge, where the car drives effortlessly on the center lane.

2. **Final Model Architecture**

The final model architecture (model.py lines 108-164) consisted of a convolution neural network with the following layers and layer size. I wanted to try a good balance of convolution layers and dense layers with pooling layers in-between reshaping the feature size and helping with slight translation in-variance.

Interestingly, keeping only the last layer as tanh activation function and remain as ReLU did not train well. Output was saturated to +1 value (hard right) for all input images. This could be because we are not allowing –ve valued features at all, all the way to the last layer using ReLU activation. By modifying the network to have ReLU in convolution layers and tanh in dense layers solved this issue an I was able to get a good range of -1 to +1 steering angels.

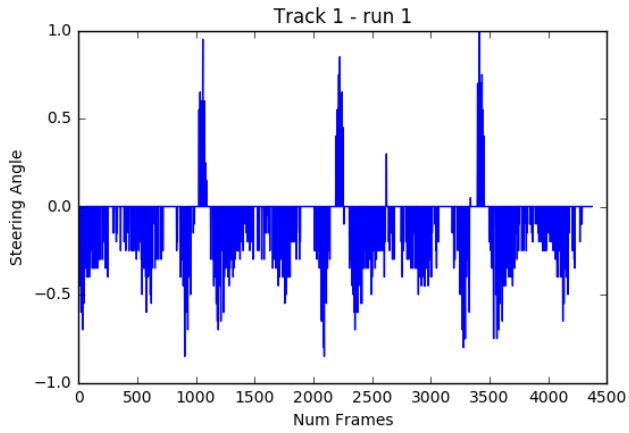
Layer ID	Layer Type	Input Dimensions	Output Dimensions
1	Convolution-5x5	64x64x3	60x60x16
	Max Pooling-2x2	60x60x16	30x30x16
	ReLU activation		
2	Convolution-3x3	30x30x16	28x28x32
	Max Pooling-2x2	28x28x32	14x14x32
	ReLU activation		
3	Convolution-3x3	14x14x32	12x12x64
	Max Pooling-2x2	12x12x64	6x6x64
	ReLU activation		
4	Convolution-3x3	6x6x64	4x4x128
	Max Pooling-2x2	4x4x128	2x2x128
	ReLU activation		
5	Flatten	2x2x128	1024
6	Dense	1024	512
	Tanh activation		
7	Dense	512	128
	Tanh activation		
	Dropout = 50%		
8	Dense	128	32
	Tanh activation		
	Dropout = 50%		
9	Dense	32	1
	Tanh activation		

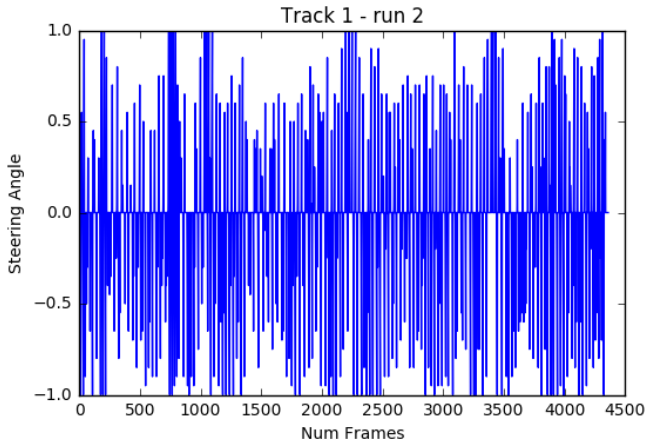
3. Creation of the Training Set & Training Process

To capture good driving behavior, I recorded 2 separate runs to start with,

Run 1 - I first recorded three laps on track 1 using center lane driving. The distribution of steering angles shows that the track turns mostly to the left with just three occasions where track is turning towards right.

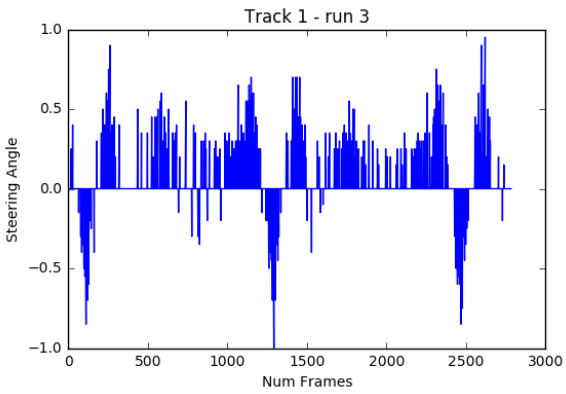
Run 2 - I then recorded three more laps where the car is weaving through the track both in straight lines and in curves. The steering angle distribution shows that it covers a good range of steering angle values.

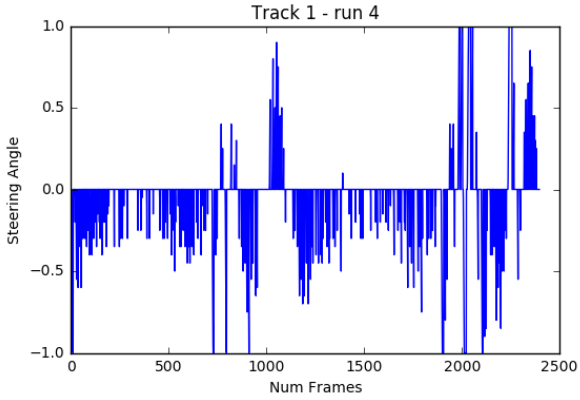




Run 3 - To improve robustness, I turned around the car (with some struggle, good to keep a button/keyboard shortcut to toggle forward or reverse tracks) and drove in the opposite direction. From the Run3 steering angle distribution its evident that I am driving in the opposite direction as compared to Run1. This helps bringing in some robustness to the network and not assume that the track mostly turns towards left.

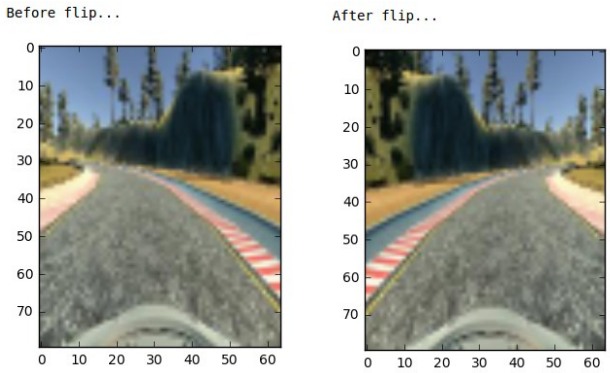
Run4 – Was captures after the car stopped on the bridge and hit a wall. Run 4's initial few frames have good driving behavior till it reaches the bridge and then there were several recovery sessions that was recorded. Proper care was taken that the car's approach towards the wall is not recorded. Only recovery from the wall towards the center lane was captured. The process was repeated in both right hand side and left hand side of the bridge and also along other track locations.





Although I recorded few sequences from Track 2, I observed that it was affecting the autonomous run in Track1. This is mainly because in Track2, I was driving along the median of the road which has dotted lanes. In Track1, there is not median line so with the network trained along with Track 2 data, the car used to catch the left most or right most line and not recover from it. Although additional training data would have helped, I couldn't spend enough time recording more sequences, so focused mainly on Track 1.

To augment the data sat, images were flipped along with steering angles. In total I ended up with ~23 thousand samples



I finally randomly shuffled the data set and set aside 20% of the data into a validation set. But this was done using keras, mode.fit() function while training. I used the adam optimizer which automatically decayed the learning rate from an initial 0.001 (default). For computing loss I used mean square error (mse) to keep a check on training and validation loss. MSE indicated overfitting of network, and suggested to add more test cases and introduce Dropout layers from initial network structure in the dense layers. With a batch size of 256 images and trained over 100 epochs, I was able to achieve a validation loss of 0.032. But this is only indicative of how well the network is learning but to test what is really happening, the model was tested several times in autonomous mode.

Interesting point to note is, once the network is trained, for each frame input the steering angle is a value from 0deg position. There is no history that is maintained between frames. Ideally, steering angels between frames should be delta differences from previous position. These temporal characteristics, could not be leveraged in plain CNN. LSTM cells and RNN's are shown to be effective in doing this, thereby taking a smooth trajectory in autonomous mode. Hopefully we get to experiment with RNN and LSTM in future projects and revisit this project later with RNN tools.