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
- writeup_report.pdf summarizing the results
- run1.mp4 – Video showing successful run on track 1

The model is able to successfully drive around track 1 with car being very close to the center of the track throughout the run. The model is able to keep running for hours.

The model.py file contains the code for preprocessing the data, 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. Choosing the training data

In my opinion this project truly highlights the importance of choosing proper training data as the model is very sensitive to the training data choice. In the begining I spent over a week collecting my own training data using keyboard, mouse and joystick across track 1 and track 2. With the data I collected my model was able to run for a little bit till it veered off track. To solve this I collected recovery data by driving from the sides of the road back to center and recording just that as suggeted in the lectures. However my recovery data worsened the performance of my model. At this point I looked through the forums and found links to a few blogs (referenced at the end) that suggested using Udacity data and image augmentation. So I ended up relying heavily on the sample data provided by Udacity. I also collected a little bit of data on one piece of the track where my model was having challenges (~450 center images). So to summarize I used the following two sources of data:

1. Udacity data (8036 center images with same no. Of left and right images)

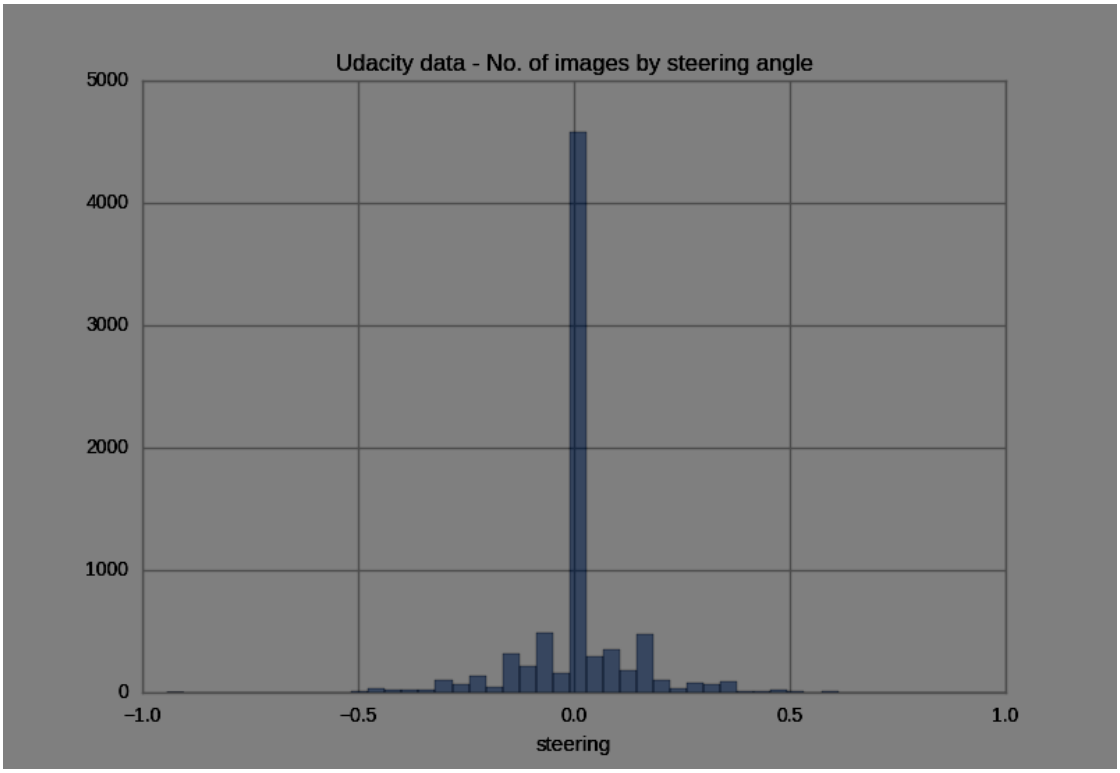
2. My data on one particular turn (~450 center images with same no. Of left and right images)

Characteristics of the data:

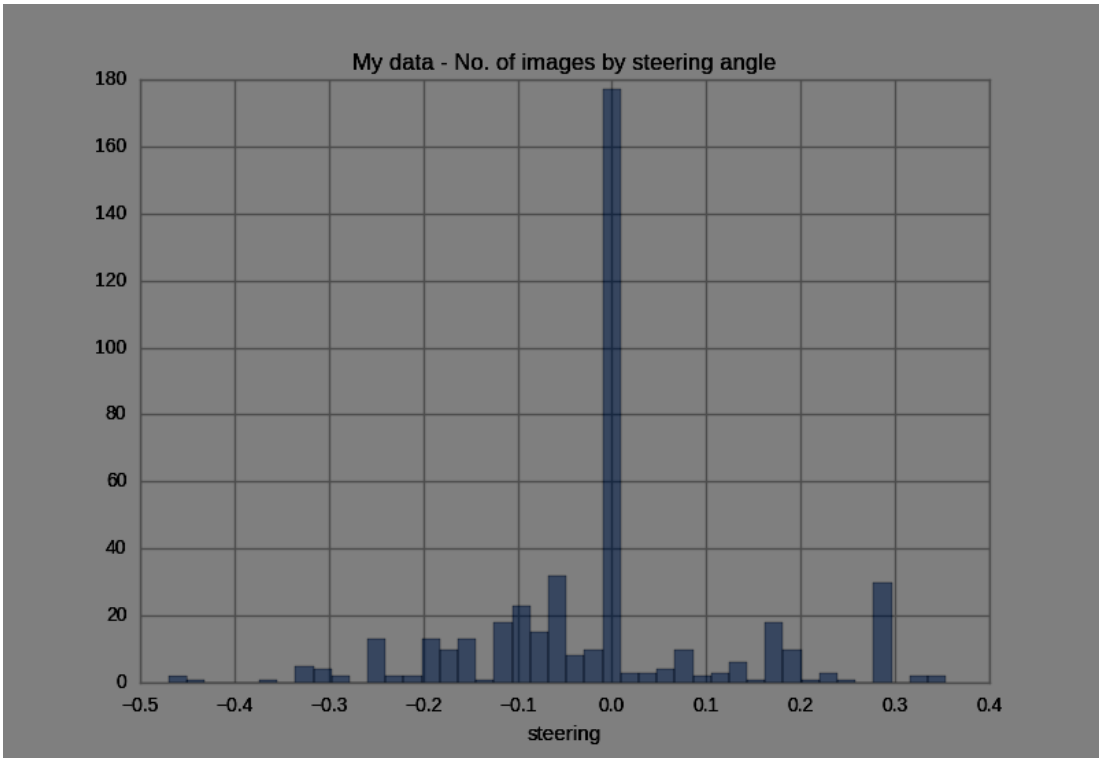
* All images are 320x160 pixels

* The data is heavily skewed to zero steering as shown below:

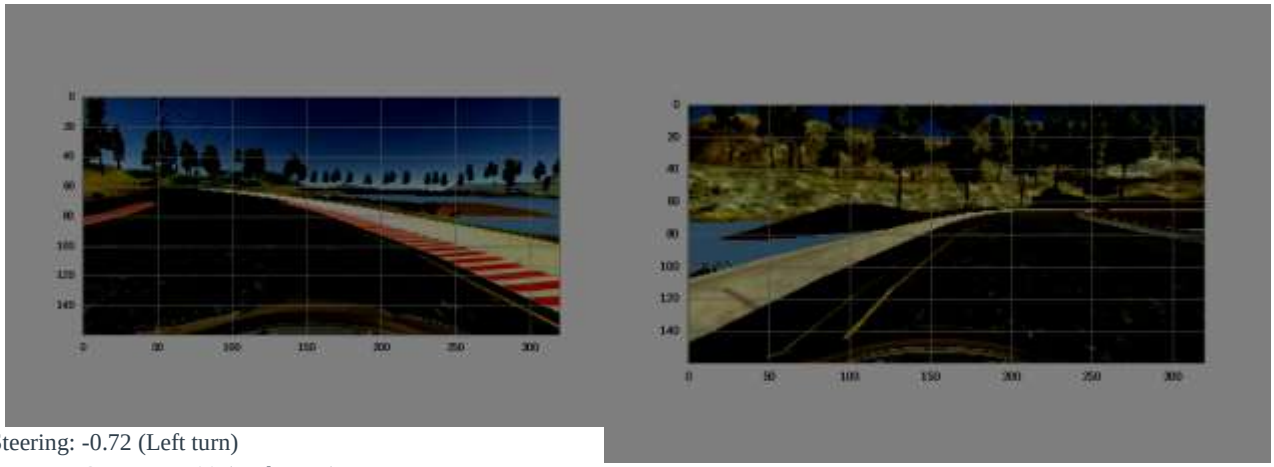
My own data had slightly less skew and more left turn data as it was



generated driving across left turn right after the bridge



Below are a few samples of images for Udacity's dataset



Steering: -0.72 (Left turn)

Steering: 1.00 (Right turn)

Besides center, left and right images a csv file is generate that contains steering, throttle, speed and brake. The steering value which is what needs to be predicted by the model has values b/w -1.0 and 1.0 as shown in the graph above.

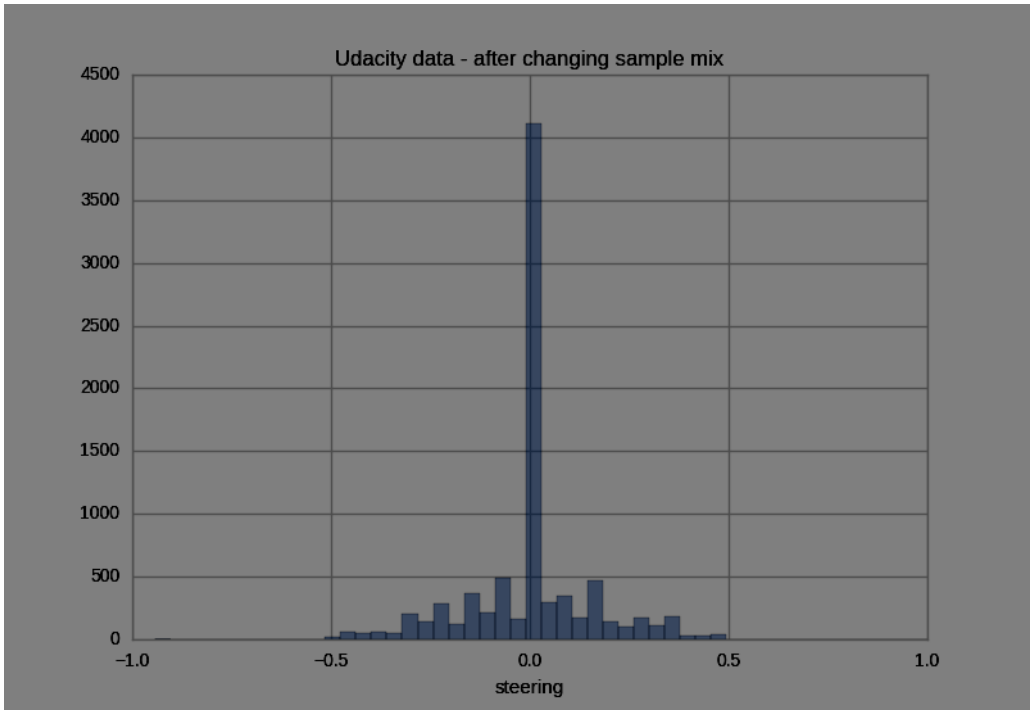
2. Preprocessing the training data

Since my data only contained driving across the center of the road, I did the following preprocessing to my data:

1. Upsampled left and right turns (model.py lines 56-71): As shown in the graph above my model contained only a little bit of non-zero steering data, so I upsampled left and right turns by doubling the sample to improve the ability to predict turns.

2. Downsampled near zero steering (model.py lines 73-76): I downsampled near zero steering by discarding 10% of the data. This parameter was tuned.

Below is a distribution plot of Udacity's data after making changes discussed above. It is slightly less skewed towards zero.



3. Upsampled my own data (model.py lines 112-115): I drove around the turn (right after the bridge) where my model was running into the dirt road once. Then I upsampled this 4 times to generate data for driving around the same turn multiple times.

While upsampling, I randomly changed the recorded steering b.w (-0.01 and 0.01).

4. Included left and right images (model.py lines 159-180): To enable recovery I used data from all 3 cameras. As suggested in the lectures I applied a steering correction factor of 0.20. This parameter was tuned. Furthermore I randomly chose only one of the images (center, left or right) for training in an epoch.

5. Flipping images (model.py lines 182-188): The track has a left turn bias. To correct for that the images, were randomly flipped and the steering was reversed. I got best performance by randomly flipping 80% of the images.

6. Randomly change brightness (model.py lines 190-202): To generate more data and reduce overfitting as well as to allow the model to generalize to track 2, I randomly changed brightness by updating the v channel of hsv.

7. Random shifts in X direction(model.py lines 205-219): This was done to simulate the effect of car being at different positions on the road, and add an offset corresponding to the shift to the steering angle.

8. Crop images(model.py line 278): Within the Keras model, a cropping layer was added to remove 60 pixels from the top (correspoding to the tree line and horizon) and 20 pixels from the bottom to remove the car's hood.

9. Resizing of images (model.py line 281): Within the Keras model, a lambda layer was used to resize images to 64x64. This was found to not adversely affect model performance while significantly reducing the number of neurons required in the model.

10. Normalizing data (model.py line 283): Within the Keras model, a lambda layer was used to normalize inputs b/w -0.5 and 0.5

3. Use of data generators for training and validation

In order to reduce the amount of data to be stored in memory and to speed up processing, data generators were used for training and validation. The code for these was suggested in the lectures.

The training generator created a sample of 20k images by selecting one of left, right or center images and performing several augmentations on it as discussed in section 2.

The validation generator created a sample of 2k images by only selecting the center image without any augmentations. Validation data was left untouched so that it closely resembled the data that would be generated in the simulator when running the model in autonomous mode.

4. Neural Network Architecture

Below is the detailed architecture of my model (model.py lines 302-331):

The model uses a cropping, resize and a normalization layer as described in the previous section. The first layer is a convolution layer of kernel size 1x1 and a depth of 3 and the goal of this layer is so the model can figure out the best color space. Following this the model uses 3 convolution layers each followed by RELU activation and a maxpool layer of size (2x2). The first convolution layer has a kernel size of 3x3, stride of 2x2 and a depth of 32. The second convolution layer has a kernel size of 3x3, stride of 2x2 and a depth of 64. The third convolution layer has a kernel size of 3x3, stride of 1x1 and a depth of 128.

After this the output is flattened. Dropout of 50% is applied and then there are 2 dense layers of 128 neurons. The final layer is an output layer of 1 neuron. All the layers are followed by RELU activation to introduce non-linearity.

Layer (type)	Output Shape	Param #
Zero Padding_1	(None, 162, 322, 3)	0
Cropping2d_1 (Cropping)	(None, 82, 322, 3)	0
lambda_1 (Resize)	(None, 64, 64, 3)	0
lambda_2 (Normalize)	(None, 64, 64, 3)	0
convolution2d_0 (Color conv)	(None, 64, 64, 3)	12
convolution2d_1 (Convolution2D)	(None, 32, 32, 32)	986
activation_1 (Activation)	(None, 32, 32, 32)	0
maxpool_2d (Pool1)	(None, 31, 31, 32)	0
convolution2d_2 (Convolution2D)	(None, 16, 16, 64)	18496
relu2 (Activation)	(None, 16, 16, 64)	0
maxpool_2d (Pool2)	(None, 8, 8, 64)	0
convolution2d_3 (Convolution2D)	(None, 8, 8, 128)	73856
activation_2 (Activation)	(None, 8, 8, 128)	0
maxpool_2d (Pool3)	(None, 4, 4, 128)	0
Flatten_1	(None, 2048)	0

Layer (type)	Output Shape	Param #
dropout	(None, 2048)	0
Dense_1	(None, 128)	262272
Activation	(None, 128)	0
dropout	(None, 128)	0
Dense_2	(None, 128)	16512
Dense_3 (Output)	(None, 1)	129

Total parameters: 372,173

To reduce overfitting the model uses dropout, as well as training/validation split. Finally the model uses Adam optimizer with a learning rate of 1e-4. This learning rate was tuned.

4. Hyper parameter tuning

Firstly the training data going into the model was tuned by changing the parameters involved in up-sampling, downsampling, steering correction and image augmentation randomness.

Within the neural network the following parameters were tuned:

1. Neural network structure – Number of convolution, max pool and dense layers

2. Learning rate in the optimizer

3. No. Of epoches – I found a value b/w 5-8 worked best. All the intermediate models were saved using Keras checkpoint (model.py lines 329-333) and tested in the simulator

4. Training samples per epoch – I found 20k to be the optimal number

I found that validation loss was not a very good indicator of the quality of the model and the true test was performance in the simulator. However models with very high validation loss performed poorly. But within different epochs, models with higher validation loss could have better performance.

Final Discussion

This was a very challenging and time consuming assignment. However I learned a ton and finally being able to drive the car autonomously was a great achievement. Here are a few things that made this assignment challenging:

1. Generating recovery data in the simulator was challenging. Driving too much to the side and then recovering added a lot of fluctuation and adversely affected the performance of the model. The key was to do “gentle recovery” but this was hard to do in practice

2. Validation loss was not a very good indicator of model performance and testing every optimization in the simulator was very time consuming.

3., there were still a lot of new things to learn – Lambda layer, Resizing layer, cropping layer, data generators.