Behavior Cloning Project

1. Model Architecture and Training Strategy

a) Employed Model Architecture

- The network architecture used is like the end-to-end convolutional neural network used by NIVIDIA with dropouts used in after the fully connected layers
- The model consists of 5 convolutional layers with filter sizes of 5x5 and 3x3 and depth ranging from 24 to 64. Also, before the convolution layer, there is one cropping layer followed by a normalization layer
- After the convolutional layers, there is a flatten layer followed by the 4 fully-connected layers with dropout in the first 2 fully-connected layers.
- The model uses the RELU activation in the convolutional layers to introduce non-linearity and the data is normalized in the model using a Keras lambda layer

2. Attempts to reduce overfitting in the model

- The model contains two dropout layers in the fully-connected (layers) to reduce overfitting and the data training was intensive and was done in sets. The training data and validation set were from both the tracks to generalize the model. The model was tested by running through the simulator and ensuring that the vehicle stays on the track

3. Model Parameter Tuning

- The model uses the Adam Optimizer, hence the learning rate was not tuned manually
- The batch size used was 32 which was the default
- The dropout in the fully-connected layers was iterated with different dropout probabilities. The final dropout probabilities used are 0.8 and 0.7. These were finalized by seeing the validation and the training loss

4. Training Data

- Varied training sets are used to train the model. The main idea was train the model, see where it fails, save the model weights and use the same model to train on a newly recorded data, this approach can be termed as learning in steps
- The training data can be classified in to the following categories

a) Center Lane Driving
b) Lane recovery driving
c) Track 2 driving to generalize the model
d) Error prone turn data

e) Driving in the reverse direction to remove the left turn bias in the data

The details of the training data are discussed in the upcoming sections.

5. Solution Design Approach

The overall strategy for deriving the model architecture was to reduce overfitting and obtain a model which is not biased.

- The first model that was used for training that was the one that I used in the traffic sign classifier project. The architecture that I used in the traffic sign classifier was an evolved version of the LeNet architecture.
- The model architecture took a lot of time to train the performance was not good as the car went out of the track.
- So, other model architectures were explored and I tried the end-to end CNN used by Nvidia. The starting point was to use the default one without dropouts and it was tuned gradually.
- The final model performed well and failed at some turns so the model weights were saved and was trained a new separate data to refine the weights
- To further update the weights, I collected more training data at scenario's where the car was going out of the track.
- I used the left and the right images and added and subtracted the correction value = 0.25 to the steering value from the left and the right images respectively. This also balanced the dataset.
- With the collection of more data, the model trained well and traversed the track completely at a low speed of 9mph. My target for the project was to run it at 30mph.
- When I changed the speed to 30mph in the drive.py file, the car started to oscillate in the track but traversed without going out of the track. The oscillations are shown in the figure below

- I realized that the training had become biased, so I loaded the trained model, and I collected more data but only used the center images and tried to drive in the middle of the road. I recorded two laps of the track.
- After this iteration, the car stopped oscillating but went out of the track at a sharp turn. The scenario is shown below.

- So, I collected data at the specific turn and again trained the model by loading the previous model.
- The car finally drove with very minor oscillations.

- Also, I reduced the resolution of the simulator window and chose the 'fastest' option in the graphics quality tab to reduce latency and maximize performance.

6. Final Model Architecture

- The final model architecture is shown in the table and the model is pictorially visualized using Keras. The actual architecture used is shown in figure 2 below

Sno	Layer	Description
1	Cropping layer	RGB Image 32x32x3 Input, Cropping (50,20), (0,0)
2	Normalization Later	(x/255) -0.5
3	Convolution 5x5	Depth =24, with relu activation
4	Convolution 5x5	Depth =36 with relu activation
5	Convolution 5x5	Depth =48 with relu activation
6	Convolution 3x3	Depth =64 with relu activation
7	Convolution 3x3	Depth =64 with relu activation
8	Flatten Layer	
9	Fully-connected 1	Dropout = 0.8 ,
10	Fully-connected 2	Dropout = 0.7 ,
11	Fully-connected 3	
12	Fully-connected 4	Output =1

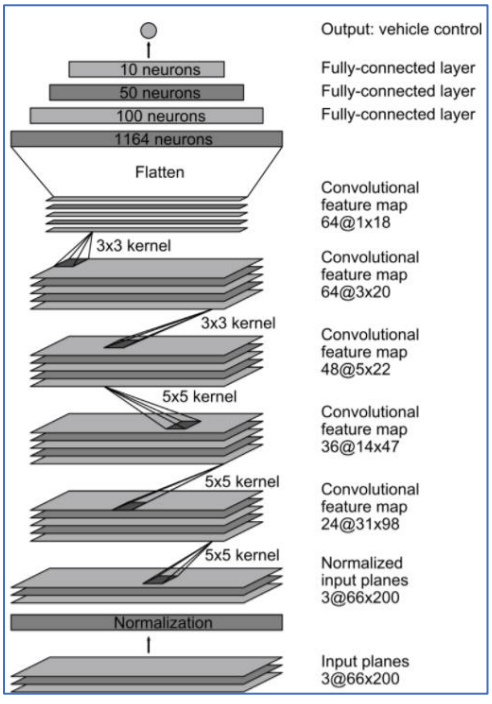


Figure 1 NVIDIA architecture

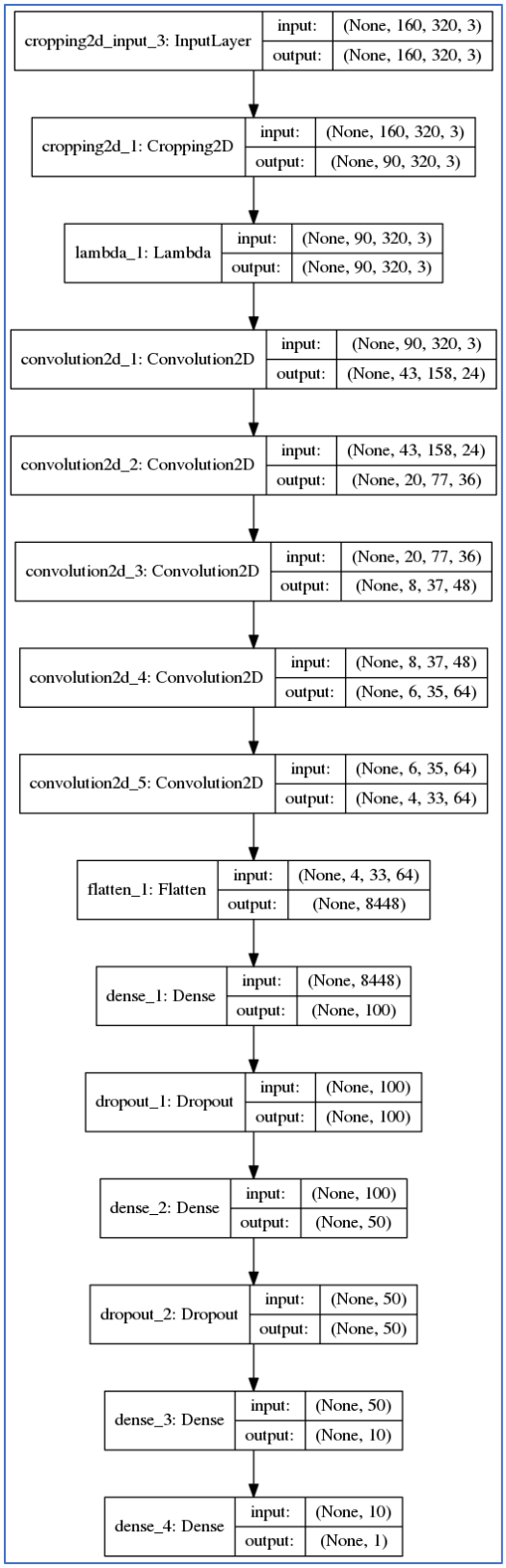


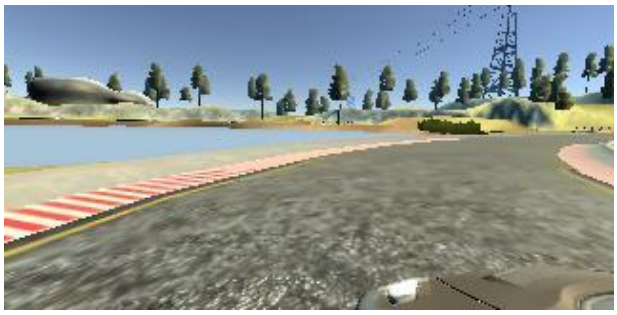
Figure 2 Actual Model Architecture

7. Creation of the Training Set and Training Process:

As discussed in section 4, the training data is divided into 4 categories.

a) Center Lane driving- Only the center images are used. An Example image is shown below:

b) Lane recovery driving- The left and the right image were used and extra data was recorded at steep turns



c) Track 2 Driving – To generalize the model, data from track two also recorded

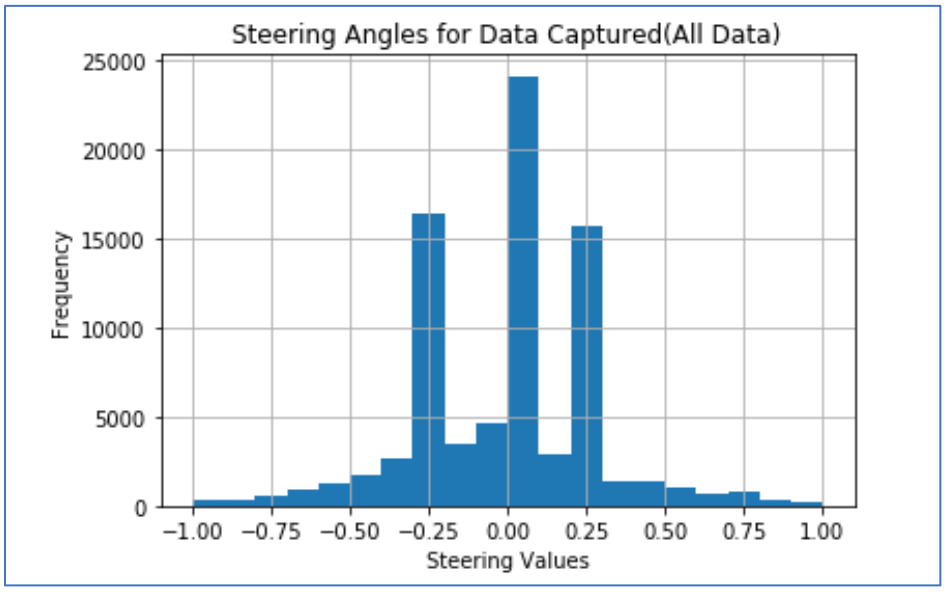
d) Error prone data:

e) Reverse Direction Driving- The training data was collected in the reverse direction of the track as well to avoid the left turn bias in the data

After the collection process, I had 81,957 data points. The data preprocessing was done to convert the BGR images into RGB.

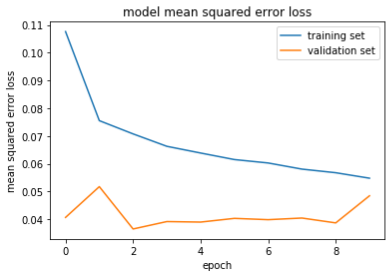
I finally used randomly shuffled data and put 20% of the data into a validation set. For several cases I also used 30 % validation when doing a fresh training.

The steering value histogram is shown in the figure which shows that the data is balanced and has a good spread

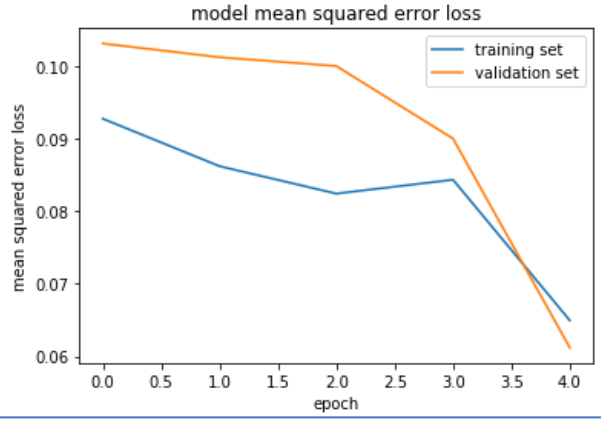


8. Training Process

```
Train on 39823 samples, validate on 9956 samples
Epoch 1/10
39823/39823 [==============================] - 436s - loss: 0.1076 - val_loss: 0.0407
Epoch 2/10
39823/39823 [==============================] - 436s - loss: 0.0755 - val_loss: 0.0518
Epoch 3/10
39823/39823 [==============================] - 436s - loss: 0.0707 - val_loss: 0.0365
Epoch 4/10
39823/39823 [==============================] - 435s - loss: 0.0663 - val_loss: 0.0392
Epoch 5/10
39823/39823 [==============================] - 436s - loss: 0.0639 - val_loss: 0.0390
Epoch 6/10
39823/39823 [==============================] - 436s - loss: 0.0615 - val_loss: 0.0403
Epoch 7/10
39823/39823 [==============================] - 436s - loss: 0.0603 - val_loss: 0.0399
Epoch 8/10
39823/39823 [==============================] - 435s - loss: 0.0581 - val_loss: 0.0405
Epoch 9/10
39823/39823 [==============================] - 435s - loss: 0.0568 - val_loss: 0.0387
Epoch 10/10
39823/39823 [==============================] - 439s - loss: 0.0548 - val_loss: 0.0485
dict_keys(['loss', 'val_loss'])
```



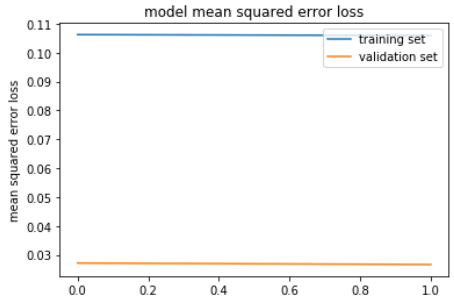
```
Train on 40696 samples, validate on 17442 samples
Epoch 1/5
40696/40696 [==============================] - 477s - loss: 0.0927 - val_loss: 0.1031
Epoch 2/5
40696/40696 [==============================] - 472s - loss: 0.0862 - val_loss: 0.1012
Epoch 3/5
40696/40696 [==============================] - 470s - loss: 0.0824 - val_loss: 0.1000
Epoch 4/5
40696/40696 [==============================] - 507s - loss: 0.0843 - val_loss: 0.0900
Epoch 5/5
40696/40696 [==============================] - 489s - loss: 0.0649 - val_loss: 0.0612
dict_keys(['val_loss', 'loss'])
```



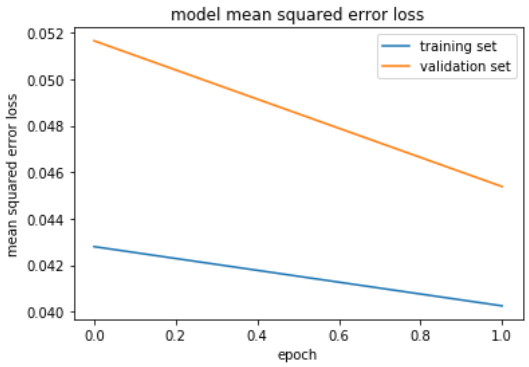

```
Train on 44144 samples, validate on 11036 samples
Epoch 1/6
44144/44144 [==============================] - 533s - loss: 0.1036 - val_loss: 0.0286
Epoch 2/6
44144/44144 [==============================] - 533s - loss: 0.0689 - val_loss: 0.0311
Epoch 3/6
44144/44144 [==============================] - 533s - loss: 0.0693 - val_loss: 0.0233
Epoch 4/6
44144/44144 [==============================] - 532s - loss: 0.0602 - val_loss: 0.0335
Epoch 5/6
44144/44144 [==============================] - 534s - loss: 0.0581 - val_loss: 0.0280
Epoch 6/6
44144/44144 [==============================] - 522s - loss: 0.0564 - val_loss: 0.0233
dict_keys(['val_loss', 'loss'])
```

In [18]:
```
from keras.models import load_model
model = load_model('model.h45')

history_object=model.fit(X_train, y_train, validation_split=0.2, shuffle=True,nb_epoch=2,verbose=1)
model.save('model.h47')

### print the keys contained in the history object
print(history_object.history.keys())

### plot the training and validation loss for each epoch
plt.plot(history_object.history['loss'])
plt.plot(history_object.history['val_loss'])
plt.title('model mean squared error loss')
plt.ylabel('mean squared error loss')
plt.xlabel('epoch')
plt.legend(['training set', 'validation set'], loc='upper right')
plt.show()
```

```
Train on 48771 samples, validate on 12193 samples
Epoch 1/2
48771/48771 [==============================] - 605s - loss: 0.1062 - val_loss: 0.0272
Epoch 2/2
48771/48771 [==============================] - 604s - loss: 0.1059 - val_loss: 0.0267
dict_keys(['val_loss', 'loss'])
```

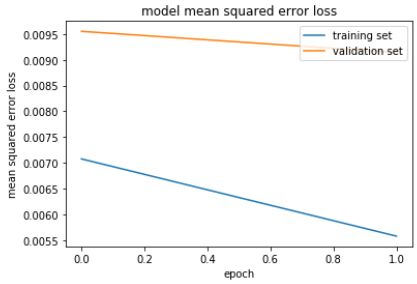



```
Train on 16881 samples, validate on 4221 samples
Epoch 1/2
16881/16881 [==============================] - 185s - loss: 0.0428 - val_loss: 0.0517
Epoch 2/2
16881/16881 [==============================] - 186s - loss: 0.0403 - val_loss: 0.0454
dict_keys(['loss', 'val_loss'])
```



```
from keras.models import load_model
model = load_model('model.h50')

history_object=model.fit(X_train, y_train, validation_split=0.2, shuffle=True,nb_epoch=2,verbose=1)
model.save('model.h51')

### print the keys contained in the history object
print(history_object.history.keys())

### plot the training and validation loss for each epoch
plt.plot(history_object.history['loss'])
plt.plot(history_object.history['val_loss'])
plt.title('model mean squared error loss')
plt.ylabel('mean squared error loss')
plt.xlabel('epoch')
plt.legend(['training set', 'validation set'], loc='upper right')
plt.show()
```

```
Train on 3872 samples, validate on 968 samples
Epoch 1/2
3872/3872 [==============================] - 40s - loss: 0.0071 - val_loss: 0.0096
Epoch 2/2
3872/3872 [==============================] - 40s - loss: 0.0056 - val_loss: 0.0091
dict_keys(['loss', 'val_loss'])
```




```
model = load_model('model.h60')

history_object=model.fit(X_train, y_train, validation_split=0.2, shuffle=True,nb_epoch=3,verbose=1)
model.save('model.h62')

### print the keys contained in the history object
print(history_object.history.keys())

### plot the training and validation loss for each epoch
plt.plot(history_object.history['loss'])
plt.plot(history_object.history['val_loss'])
plt.title('model mean squared error loss')
plt.ylabel('mean squared error loss')
plt.xlabel('epoch')
plt.legend(['training set', 'validation set'], loc='upper right')
plt.show()
```

```
Train on 8089 samples, validate on 2023 samples
Epoch 1/3
8089/8089 [==============================] - 89s - loss: 0.0074 - val_loss: 0.0073
Epoch 2/3
8089/8089 [==============================] - 89s - loss: 0.0072 - val_loss: 0.0073
Epoch 3/3
8089/8089 [==============================] - 89s - loss: 0.0071 - val_loss: 0.0071
dict_keys(['loss', 'val_loss'])
```

The final model was saved in as model.70 and was trained for only one epoch and I used the adam optimizer so the manual training of the learning rate wasn't necessary