Self Driving Car: behavioral Cloning Project

Vamshidhar Pandrapagada

December 2, 2017

Self Driving Car: Behavior Cloning Project

Introduction

This projects uses deep neural networks and convolutional neural networks to clone driving behavior.
Built in Keras, the model is trained on sequence of images and steering angles collected by driving/steering the car manually in a simulator. The model will predict a steering angle which is used to drive an autonomous vehicle.

Context Setting and Challenges

Training a neural network to build a self driving car is no easy task. Building a machine to mimic human driving requires gathering lot of data parameters. A simple convolutional neural network which has the ability to remember basic patterns in a sequence of high quality images will have more than a million parameters, let alone a network which generalized well and operates perfect in all driving conditions.

In this project, the task is to build a neural network which will only predict the steering angle based on the road conditions. We have access to the a simulator which can be used to manually drive and collect data (images) using good driving behavior.

Data Collection

Simulator generates sequence of images using 3 cameras mounted on the hood of the car (Center, Left and Right). Steering angle is calculated using the Center camera.

Let's look at a few images obtained using data collection.

We use 3 cameras on the car, the CENTER camera is always used to calculate the steer angle. For us to be able to use LEFT and RIGHT camera images a small correction factor of 0.2 is added to the center steer angle and tag the result to left camera. Similarly same factor of 0.2 is subtracted from the center steer angle and tag the result to right camera. This factor teaches the network to steer back to the center if vehicle is drifting towards the sides of the road.

This correction is very helpful when for example the vehicle sees a left curve, it tells the network to steer a little hard to the right to stay on course.

LEFT Steer Angle: 0.16

CENTER Steer Angle: 0.36

RIGHT Steer Angle: -0.04

LEFT Steer Angle: -0.21

CENTER Steer Angle: -0.01

RIGHT Steer Angle: -0.41

LEFT Steer Angle: -0.05

CENTER Steer Angle: 0.15

RIGHT Steer Angle: -0.25

Data Augmentation

Deep artificial neural networks require a large corpus of training data in order to effectively learn, where collection of such training data is often expensive and laborious. Data augmentation overcomes this issue by artificially inflating the training set with label preserving transformations. Recently there has been extensive use of generic data augmentation to improve Convolutional Neural Network (CNN) task performance.
Improving Deep Learning using Generic Data Augmentation

Data Augmentation is also a regularization technique to generate new training images from existing ones to boost the size of the training set. This will not only help the model learn better, but will also reduce over-fitting. The idea is to randomly adjust brightness, rotate/flip, vertical/horizontal shift every picture in the training set and add them back to the training set. This forces the model to be more tolerant to position and orientation of the key pixels in the image.

Using data from 3 cameras with correction factors applied to steering angles is our first step in Data augmentation, which we already did in previous step

Adjust Brightness of Images

In this step of augmentation, adjust the brightness of the images by selecting a random intensity level for each image. Intensity level is selected at random to simulate different sunlight conditions in a day. A darker intensity can be as close to driving at night.

Open CV reads images in BGR format. We first conver images from BGR to HSV(Hue-Saturation-Value) and randomly alter V value to change brightenss and finally convert the image back to RGB.
Drive.py in autonomous mode also gets the images from simulator using PIL image library which reads in RGB format.

Here are few examples of images after adjusting the brightness.

Random Image Flip

While driving in a loop the model tries to memorize the curves and can get biased if it frequently sees one type of a curve (LEFT/RIGHT). So in autonomous mode, the car tends to steer in that biased direction even if going straight is the best option.
To be able to generalize the model better, pick a random set of images and flip them counter clockwise. This way, model will observe both the curves and performs better in autonomous mode.

Here are few flipped images.

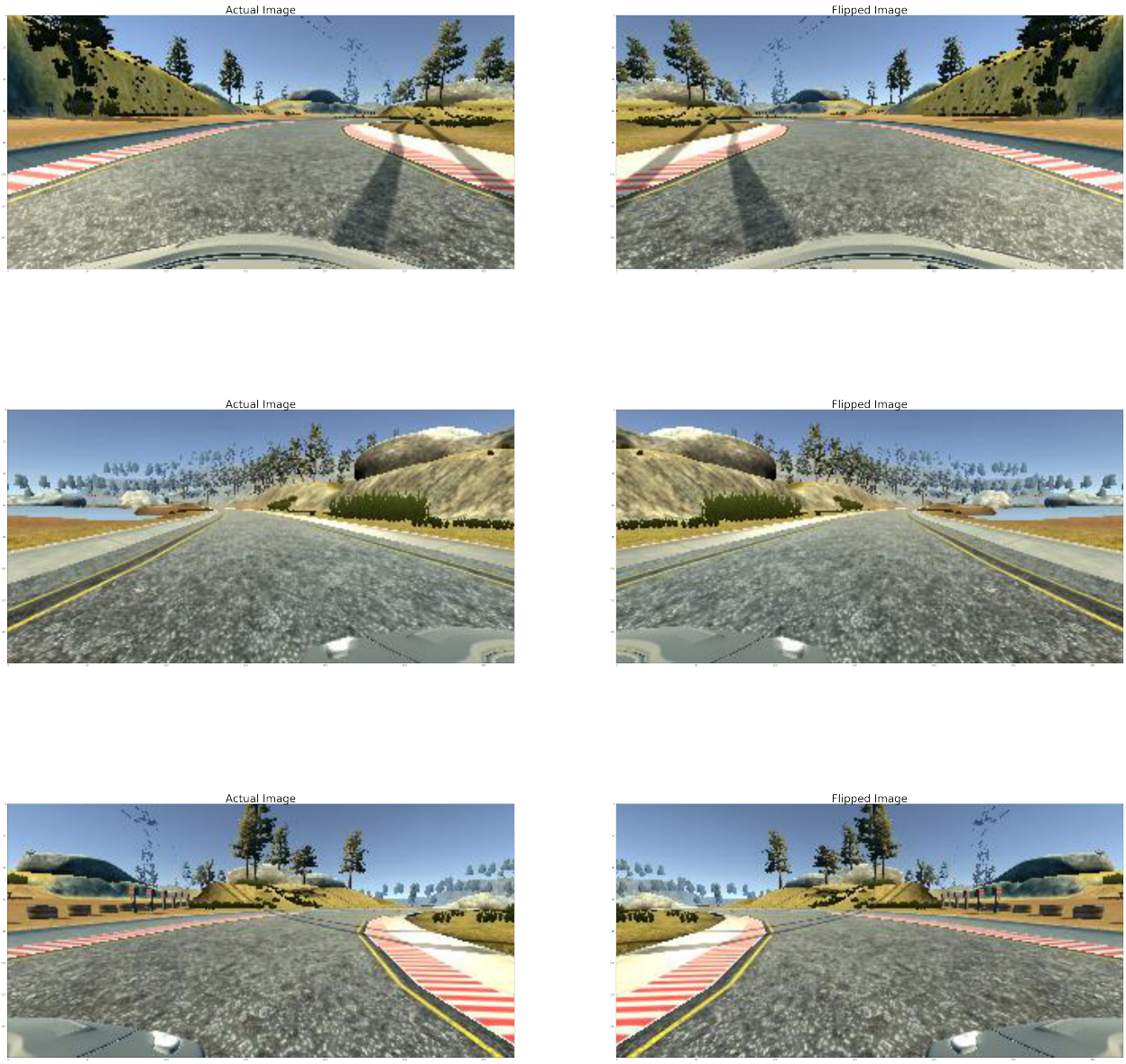


Figure 1: png

Random Veritical Horizontal Shift

This augmentation technique alters the position of an image to new position and orientation, generating new images to simulate the effect of car being is various positions and angles on the road.

Camera position changes with this shift, hence the steering angle should be rectified accordingly using a correction factor.

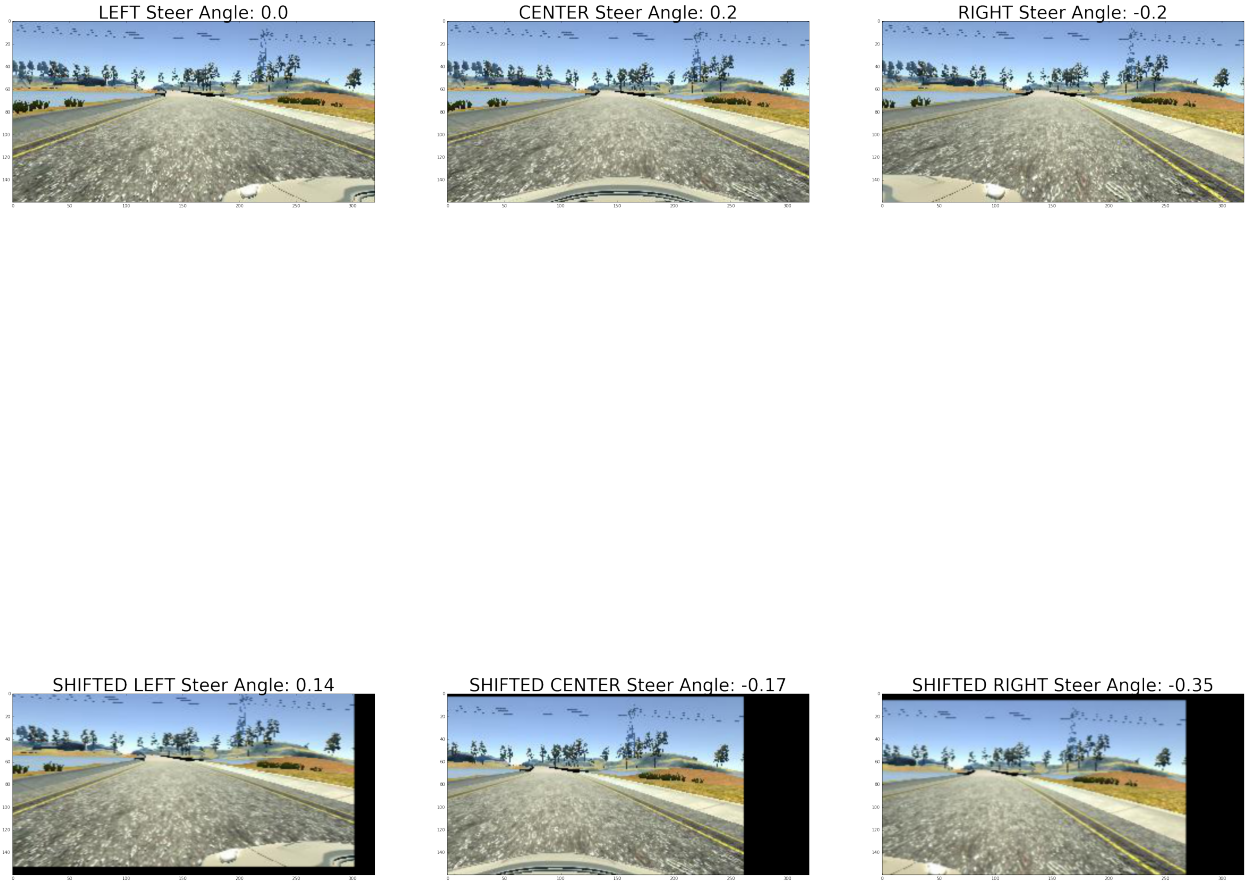
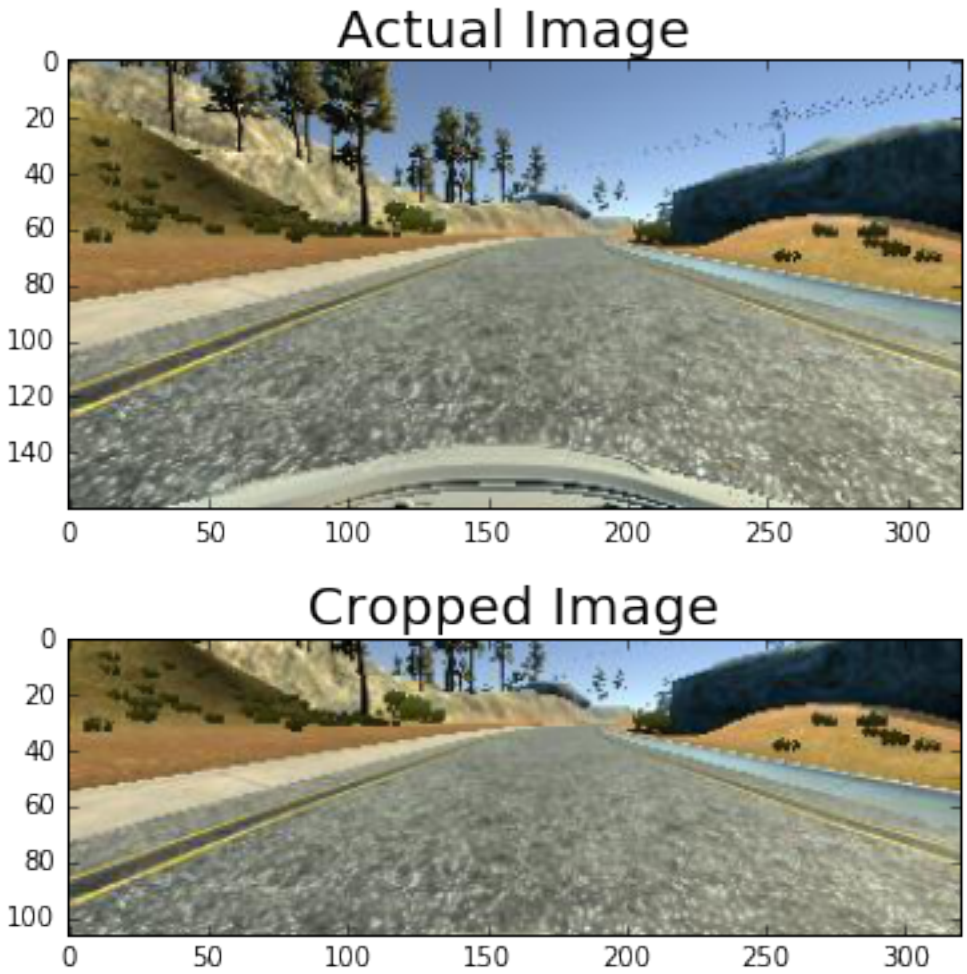


Figure 2: png

Image Preprocessing

This steps helps in removing noise from the image. If you observe the images plotted above, almost 1/5th of the image from the top is the sky and around 20 pixels from the bottom is the hood of the car. These pixels provide no added value to the neural network. Cropping the image to get rid of these pixels will help the neural network look only at the road as the car moves.
Here's an example how an image looks before and after cropping.



Collected Data

Data Collection and Augmentation resulted in a total of 23,760 images. A total of 17,856 were used for training the CNN and 5,904 images were used as Validation set

Model Pipeline

Once we have the necessary training images ready (after the data augmentation), construct the model pipeline to train the neural network.

Data Normalization : As for any data-set, image data has been normalized so that the numerical rangeof the pixels is between -1 and 1.
Model Pipeline in Brief: Inputs size fed to the model pipeline is 160 x 320 x 3 and cropped to get rid of noise (SKY and HOOD of the car) to get to a size of 88 x 320 x 3.
This architecture used here very similar to the one published by autonomous vehicle team in NVIDIA. End-to-End Deep Learning for Self-Driving Cars.

The model follows a The All Convolutional Net. Max-pooling layers are simply replaced by a convolutional layer with increased stride without loss in accuracy. This yielded competitive or state of the art performance on several object recognition datasets (CIFAR-10, CIFAR-100, ImageNet).

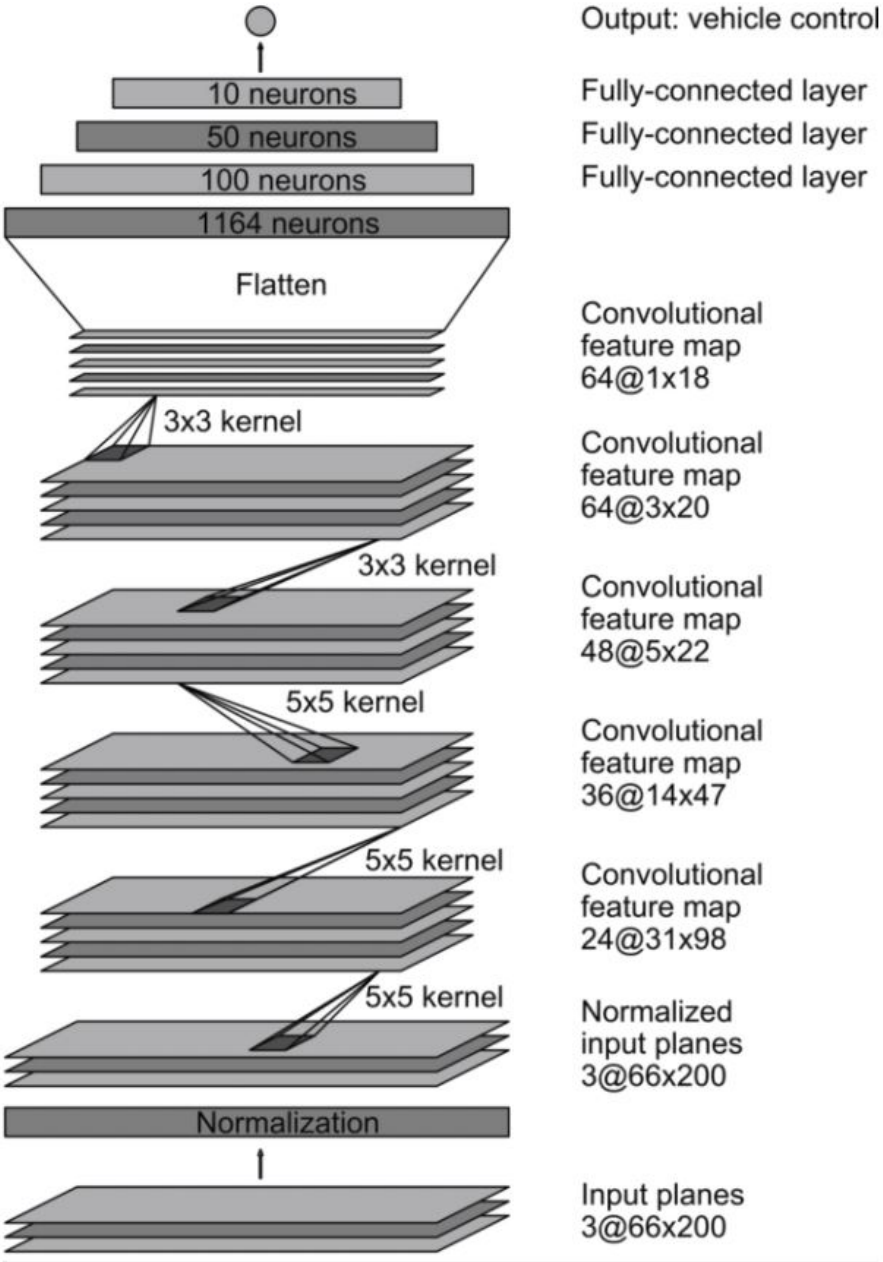


Figure 5: CNN architecture. The network has about 27 million connections and 250 thousand parameters.

Figure 3: NVIDIA ARchitecture

After several attempts, **Spatial Dropout** generalization on third and fourth convolutions followed by regular dropout on fisth convolution provided least loss on validation set.

Our network is fully convolutional and images exhibit strong spatial correlation, the feature map activations are also strongly correlated. In the standard dropout implementation, network activations are "dropped-out" during training with independent probability without considering the spatial correlation.

On the other hand Spatial dropout extends the dropout value across the entire feature map. Therefore, adjacent pixels in the dropped-out feature map are either all 0 (dropped-out) or all active. This technique proved to be very effective and improves performance

Maxpool layer is used only on the last convolution layer with regular drop out.

Model Pipeline Construction:

1. Input Image: 160 x 320 x 3
2. Cropped Image: 88 x 320 x 3
3. Normalization Lambda Layer
4. Convolution 1: Kernel size = 5, Feature maps = 24, Strides = 1, Padding = SAME, Output Image: 88 x 320 x 24, Weights Initializer = Truncated Normal, Activation = ELU
5. Convolution 2: Kernel size = 5, Feature maps = 24, Strides = 2, Padding = SAME, Output Image: 44 x 160 x 24, Weights Initializer = Truncated Normal, Activation = None
6. Convolution 3: Kernel size = 5, Feature maps = 36, Strides = 1, Padding = SAME, Output Image: 44 x 160 x 36, Weights Initializer = Truncated Normal, Activation = ELU
7. Convolution 4: Kernel size = 5, Feature maps = 36, Strides = 2, Padding = SAME, Output Image: 22 x 80 x 36, Weights Initializer = Truncated Normal, Activation = None
8. Convolution 5: Kernel size = 5, Feature maps = 48, Strides = 1, Padding = SAME, Output Image: 22 x 80 x 48, Weights Initializer = Truncated Normal, Activation = ELU
9. Convolution 6: Kernel size = 5, Feature maps = 48, Strides = 2, Padding = SAME, Output Image: 11 x 40 x 48, Weights Initializer = Truncated Normal, Activation = None
10. Spatial Drop out with probability 0.3
11. Convolution 7: Kernel size = 3, Feature maps = 64, Strides = 1, Padding = SAME, Output Image: 11 x 40 x 64, Weights Initializer = Truncated Normal, Activation = ELU
12. Spatial Drop out with probability 0.3
13. Convolution 8: Kernel size = 3, Feature maps = 64, Strides = 1, Padding = SAME, Output Image: 11 x 40 x 64, Weights Initializer = Truncated Normal, Activation = ELU
14. Maxpool Layer 1: Kernel size = 1, Strides = 1, Padding = 'VALID', Output Image: 11 x 40 x 64
15. Drop out probability 0.3
16. Flatten : Output = 28160 neurons, Activation = ELU
17. Drop out probability 0.3
18. Fully Connected Layer 1: Output = 100 neurons, Activation = ELU
19. Fully Connected Layer 2: Output = 50 neurons, Activation = ELU
20. Fully Connected Layer 3: Output = 10 neurons, Activation = ELU
21. Output Fully Connected Layer: Output = 1 neurons

Convolution 6 and Convolution 7 use Spatial Dropout. Convolution 8 and Flatten use regular Dropout regularization.
All layers use the weights initially being initialised from a truncated normal distribution with a 0 mean and a standard deviation of 0.01.

Hyper Parameters

1. The number of epochs used: 45
2. Learning Rate: 0.01.
3. Batch size : 32

4. Momentum: 0.9

Weights updated using back propagation and stochastic gradient descent optimizer. Learning rate exponential decay was applied with global_step value computed as (learning_rate / epochs).
When training a model, it is often recommended to lower the learning rate as the training progresses, which helps the model converge and reach global minimum.

Training

Training the neural network with large number of images loaded into memory may slow down the entire process. Data generator functions in python are used to mitigate this problem by reading the required set of images in chunks using the batch size.

Keras supports a function model.fit_generator which trains the network using the small size chunk of images at a time yielded by the generator function and destroys them saving memory.
After training for 45 epochs on 17,856 images using a batch size of 32, the model yielded the following results.

```
Epoch 40/45
243s - loss: 0.0200 - val_loss: 0.0211
Epoch 41/45
243s - loss: 0.0197 - val_loss: 0.0205
Epoch 42/45
242s - loss: 0.0199 - val_loss: 0.0208
Epoch 43/45
243s - loss: 0.0195 - val_loss: 0.0206
Epoch 44/45
243s - loss: 0.0198 - val_loss: 0.0209
Epoch 45/45
242s - loss: 0.0195 - val_loss: 0.0205
dict_keys(['val_loss', 'loss'])
```

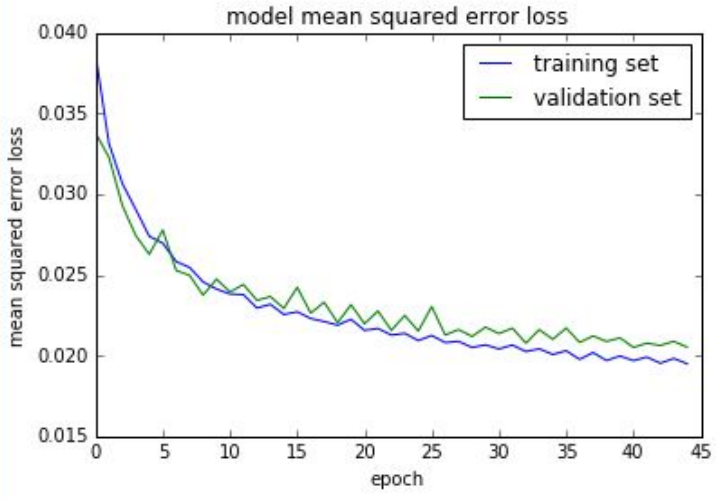


Figure 4:

Model Performance

Training Data was collected on Track 1. Using the model's steering predictions, the autonomous car was able to drive well following the course of the track.

Next Steps and Improvements

This project has tested my skills on Deep learning and CNNs to the core.

There is still a lot of room for improvement and in my mind are a few listed below:
1. How does the car perform on real world data? Record a video by setting up similar cameras in my car and train the network to check if the model gives similar results. 2. Train and improvise the model on Track 2 which is much more challenging (has more turns, loops and various driving conditions) 3. Ability to make the car drive only on right side of the road?

As a final thought, one of the best projects I've worked on till date. At the same time, Requires a lot of patience to get everything right as we try to train the network using different augmenting techniques and hyper parameter settings.