BEHAVIORAL CLONING

This project is about training the autonomous car to drive through a path, with the help of neural networks, which is also called behavioural cloning.

Udacity provides a simulator wherein we drive the car through a particular track, collect the data, and feed it to a neural network model, to train the car. The neural network will watch our behaviour of driving the car. It studies the way we drive, and then use the knowledge to drive on its own, the next time. Hence, called behaviour cloning.

My project includes the following files:

- model.py containing the script to create and train the model
  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.
  https://github.com/anjanarajam/SELF_DRIVING_BEHAVIORAL_CLONING
- drive.py for driving the car in autonomous mode
- model.h5 containing a trained convolution neural network
- write_up.docx summarizing the results

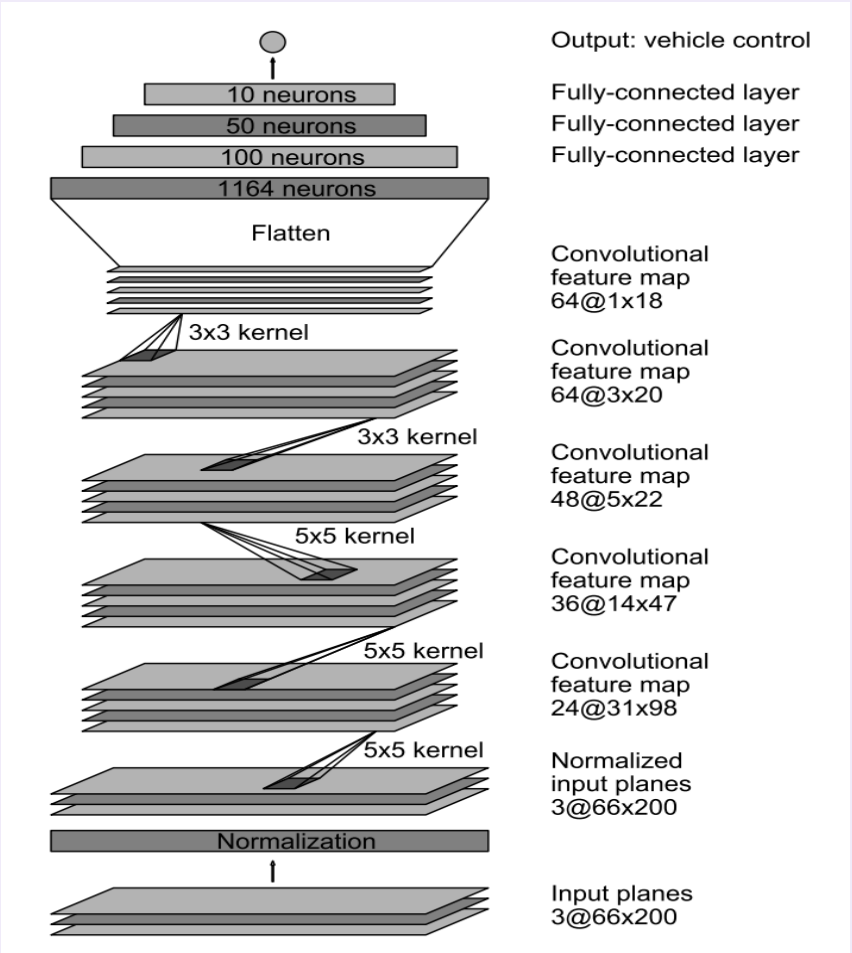
Using the Udacity provided simulator and my drive.py file, the car can be driven autonomously around the track by executing **python drive.py model.h5.**

FIGURE 1: CENTRE IMAGE

Model Architecture

The mode architecture that I used, is the efficient, **NVDIA convolutional network.**

- The input given to the network has dimensions 160x120x3.
- This network normalises the data before convolving. The data is normalised using Keras lambda layer.
- The normalised data is then given to 5 convolution layers, with filter size 5x5 for first three and 3x3 for the next two. The first three 5x5 layers, are subsampled to reduce the dimensions. The other two are convolved with same dimensions.
- The final output of convolution layers is given to four fully connected layers.
- This model Relu activation layers to introduce non linearities to the model.
- The output is a single output dense layer since we need just one output, which is the steering angle.



Training Data

A little about over fitting and underfitting

“A model that generalises well neither overfits or underfits.”

Before I get into the details of overfitting and underfitting, I would like to explain the term variance and bias.

Variance and Bias

Variance and bias are prediction errors of a model. We can avoid overfitting or underfitting of a model if we have proper understanding of variance and bias.

A model with high **bias** does not learn enough to generalise the data on the testing set. A bias is the difference between the average prediction of the model and the correct value. A high **Variance** learns too much during training that it cannot generalise the data again on the testing, which it has not seen before. However, it does well on the training data.

An **overfit** model has **high variance** and **low bias** on the training set which leads to poor generalization on new testing data. When a model learns too much, so much so that it also learns patterns not required, also called as noise, it has high variance. This happens when we leave the algorithm to learn for a long time. It’s like a student has mugged up the entire theoretical subjects, but when it came to lab exams, he could not apply that knowledge.

An **underfit model**, similarly has **low variance** and **high bias**, which means it does not have sufficient data to learn. This could either be, that enough training data is not fed or the training is stopped earlier.

Attempts to reduce overfitting or underfitting in the model

The training data, taken from simulator is stored in a csv file, consisting of images (centre left and right) and steering angles. The role of the network is to match the images in the car to that of steering angle. At first, I took one entire lap of the track, to collect

the training data. The result was that the car would just go off track at the start itself. I realised the model was underfitting (the training loss was more than validation loss), to which I decided to increase the training data by taking:

1) Images from all angles: Since the car went off track, I decided to take in images, from all angles, left, right and centre.

FIGURE 2: LEFT IMAGE

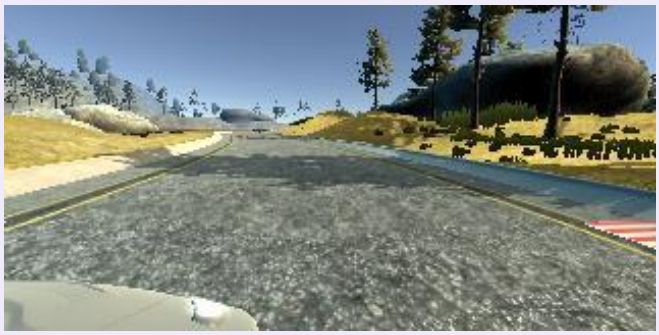


FIGURE 3: RIGHT IMAGE

2) Flip Images and measurement: The track in the simulator is biased towards left turn. So, the images had to be flipped, to give the network images of turning right. And the steering angle is taken as a negative of its value

FIGURE 4: FLIPPED IMAGE

3) Steering angle augmentation: Since the network matches the images of the car to the steering angle, the wobbly nature of the car was corrected by adding plus 20 degree when the car turned left and minus 20 degree when car turned right.
4) Image cropping: In the image, even the surroundings are displayed, which includes sky or trees. We need only the lane, so the image is cropped, just enough to see the lanes.

FIGURE 5: CROPPED IMAGE

5) Recovery data: A lot of times the car went off track at specific areas. To overcome that issue, I recorded more data by turning the car towards centre in the problematic areas.

FIGURE 6: RECOVERY IMAGE

FIGURE 7: RECOVERY IMAGE FROM PROBLEM AREA

After augmenting the images like above, the training data is increased with varieties of images, which helped me a great deal in successful training of the network.

As far as overfitting (training loss should be less than validation loss) is concerned,

1. hyper parameters tuning like that of batch size and number of epochs.
2. Added dropout of 0.25 between fully connected layers.
3. L2 regularization.
4. Tuned learning rate as low as 0.0001 while compiling.

Batch size and number of epochs - I took a batch size of 32 and number of epochs as 3. Through research I found that taking a small batch size, not very small, helps in better learning. Number of epochs depends on how much you want to train the network for the learning process to prevent overfitting.

Training Strategy

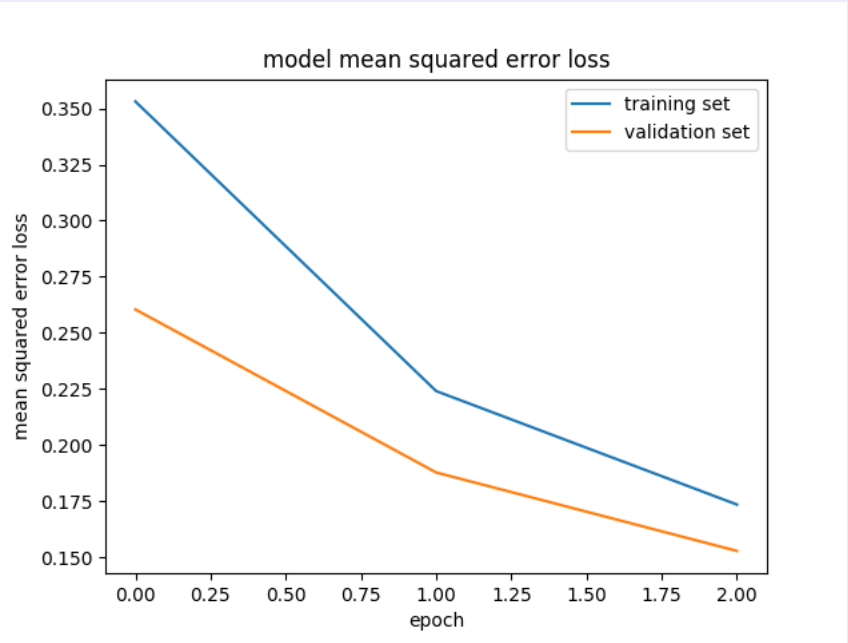
1) The data set is split into training set (80 %) and validation set (20 %).
2) Before feeding the data to the network, the data is pre-processed by normalising the data, using lambda layer. Lambda layer can take each pixel in the model and normalise it.
3) For training the network, **Keras** is used to build and train the model.
4) The optimizer used was **Adams optimizer**, using the commonly used regression loss function called **mean square error,** with a learning rate as low as 0.0001.

Python Generator

Python generator has been my saviour in getting through my project. The training time was huge without python generator, since I had collected lot of data for the training. Python generator also helped me train the data batch wise. Generators are iterators that work with large amounts of data. It does not store the entire dataset in memory at once, it takes portions of data and

processes, which is memory efficient. While iterating, it stores the previous state the training using yield batch wise.

Model Mean squared error loss:



Final video output:

Conclusion

I conclude that the main task for me in this project for me was only the collection of appropriate training data and usage of python generator. At the end of the process, the vehicle can drive autonomously around the track without leaving the road.