Udacity Self Driving Cars Nanodegree- Behavioral Cloning project

Objective:

To build a neural network for a Car to be able to drive (thru a Simulator) on a Test track by cloning the behavior of the car as observed in the Training dataset obtained by driving the car on the same Test track. The car should complete one lap of the track while remaining on the track at all times (preferably in the center) and not go off the track at any point of time.

There was an additional optional challenge for the car to be able to drive on a different track (than observed in the Training dataset).

Details on the project:

The neural network model needed to be built in a python program file "model.py". When this program was run; it needed to save the model in a "model.h5" file. This saved neural network model can then be invoked to run the Car Simulator by running the command, "python drive.py model.h5". The helper file "drive.py" was provided by Udacity, which we could have modified; if we needed to.

To collect the dataset; the following process needed to be followed: When the car is driven thru the Simulator in "Training" mode; it stores images in IMG folder (child folder created in the current folder) and details of the images are stored in the driving_log.csv file. The drving_log.csv file has 7 columns; in the following order: Center camera image file name, Left camera image file name, Right camera image file name, Steering (values between -1 and +1), Throttle (values between 0 and 1), Brake (value 0 for this project) and Speed. To build the model; we could use either only the Center camera image or all the three cameras images. In the output, only Steering value was relevant for this project; Throttle, Brake and Speed values were ignored.

Also, the project gave you the option to use the dataset provided by Udacity; instead of using the dataset collected by you driving the car around the track thru the Simulator. However, the project needed you to drive at least one lap thru the Simulator (in the "Training" mode) and include the resulting file "video.py" in the submission. The "video.py" was created from the dataset by using the command, "python video.py run1"; where "run1" is the folder where dataset is stored.

Approach:

Here are the steps I followed as I worked on the project:

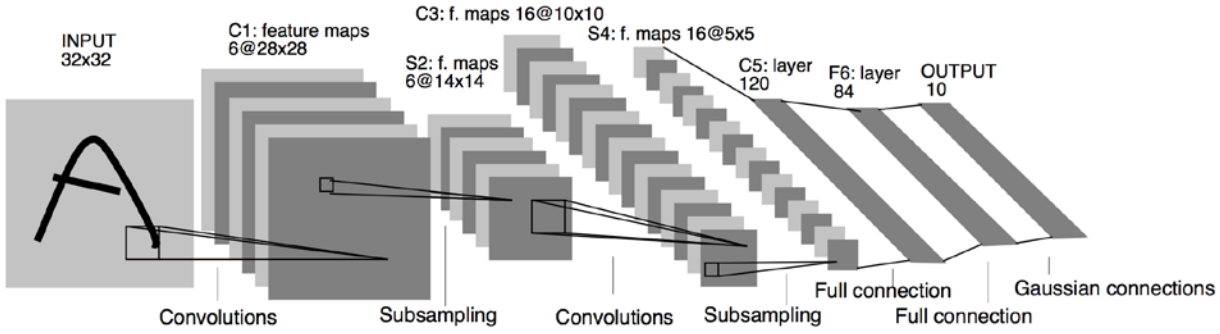
1. Access the Dataset. The "driving_log.csv" file is in "data" folder and the Images are in "data/IMG" folder. I used the Udacity provided dataset.
2. Read column 1 of the driving_log.csv file to get the Center camera file name. Append all the values in a features array X_train. Read column 4 of the driving_log.csv file to get the Steering value. Append all the values in a labels array y_train.
3. Build the neural network (see the following section, "**Building the neural network**").
4. Normalize the images (divide the image pixel values by 255 and then subtract 0.5 to get resulting values between -1 and +1); by adding a Keras Lambda layer before the Flatten layer. The steering angle measurements don't need to be normalized because they already are normalized; in the Udacity provided Training dataset.

5. Augment the data by flipping the images (use OpenCV flip function) and multiply measurements by -1.0. I had to undo this step because after augmenting the data, the network accuracy decreased; which is counterintuitive and I don't know the reason why. One possibility is that the Udacity provided training dataset was already augmented and augmenting it again just worsened the network accuracy.
6. Use left and right camera images in addition to the center camera image. File names of left and right camera images are in columns 2 and 3 respectively in the driving_log.csv file. Adjust the left camera and right camera steering measurements by a correction coefficient; so that the left camera measurement will be (measurement+coefficient) and the right camera measurement will be (measurement-coefficient); the center camera image steering measurement need not be corrected. After trial and error; I arrived at a correction coefficient of 0.2
7. Crop the images; about 70 top pixels of the image is the scenery (trees, mountains, etc.) and about 25 bottom pixels of the image is the hood of the car. So, we crop the images to exclude these redundant portions of the image. This not only improves the accuracy of the network but also makes it learn faster because of the reduced size of the images.
8. Compile the model: use model.compile Keras function; with parameters being Adam Optimizer (for gradient descent) and Loss function as mean squared error.
9. Run the model: use model.fit Keras function; with parameters being X_train, y_train and validation_split=0.2 (meaning the model splits off 20% of Training dataset to use as Validation dataset).

Building the neural network:

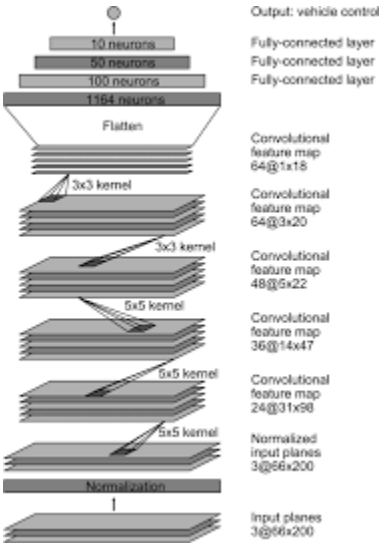
To build the Neural network; I started with the most basic model and then kept adding to it; compiling the model and running the Simulator (in Autonomous mode) for each model and observing the results to see if they match the Objective :

1. First, I started with a very basic Neural Network in Keras: Flatten the images (start with a Keras Sequential model and then add a Flatten Layer) and pass that to a Dense layer with a single output node to predict the steering angle. Input shape of images is (160, 320, 3).
2. Next, I tried the LeNet model for the neural network architecture. This model architecture is as per the picture (obtained from the Internet) below:



The architecture summary is:
Conv1 (5x5x6) ->Max Pool ->Conv2 (5x5x16) ->Max Pool->Flatten->FC 1 (120) ->FC2 (84) ->Output (1)

3. The LeNet architecture did not satisfy the Project objective; so I tried the Nvidia End to End driving neural network next. This model architecture is as per the picture (obtained from the Internet) below:



This network has a Normalization layer followed by 5 Convolutional layers; then output is Flattened followed by 4 Fully Connected layers. A stride of 2x2 is used between each of the Convolutional layers. The architecture summary is:
Input-Normalization-Conv1 (5x5x24) -> Conv2 (5x5x36) -> Conv3 (5x5x48) -> Conv4 (3x3x64) Conv5 (3x3x64) -> Flatten->FC 1 (100) ->FC2 (50) -> FC 3 (10) ->Output (1)

Results:

With the above overall approach, Nvidia End to End driving neural network worked better than the LeNet model and **met the Project Objective.**

However, the car did not drive well on Track 2 (the Optional challenge track) and veered off the track after a few seconds.

Thus, the model does not generalize well, i.e. it does not do well on tracks not seen in Training. I tried to use Dropout regularization after the Fully Connected layers to reduce the Variance; but it did not seem to help for the Car to drive on Track 2.

Alternative possible approach:

An alternative possible approach that may yield better results on Track 2 (or other tracks unseen in the Training set) is to use Transfer Learning.

Transfer learning involves leveraging models (like VGGNet, GoogLeNet, etc.) already developed by the industry. These models have been trained on large datasets, like ImageNet. The Transfer Learning approach involves taking a pre-existing model like GoogLeNet; lopping off the final layer to be replaced with a layer having the number of outputs needed (1, in our case, for the steering angle prediction). The weights of the layers (except the final layer) can be either frozen (for a small data set, such as ours, so it

does not overfit the Training set) or can be initialized with their existing values (for large data set) and then trained. The weights of the final layer will need to be trained, in either case.

I tried to use the Transfer Learning approach; but was not successful in implementing it because of some errors that I encountered.