Behavioral Cloning - Make a simulator car drive like human*

Shanthan Suresh[1] [†]

February 20, 2017

1 Overview

In this project we train a simulator car to drive by itself on a race track. The main tasks are to drive the car on the track and then use a deep learning network to mimic the behavior. We use the simulator provided by Udacity to drive on the track and collect image data and the steering angles. Using this data, we train a convolutional neural network(CNN), to predict the steering angle. In the testing phase (also called autonomous phase), we feed the car camera images to the network which predicts the steering angle to direct the car on the lane of the test track.

2 Introduction

This is an interesting and challenging problem because we need to train the car to drive by itself and make sure that the network does not memorize the training data. This problem is called overfitting in deep learning terminology. To avoid the overfitting problem and make the network generalize to all conditions of the track, we resort to some augmentation techniques by which we synthesize artificial data to increase variability in the training data and

**This project was supported by Udacity and Nvidia Graphics Pvt Ltd

[†][1]Shanthan Suresh is with Nvidia Graphics Pvt Ltd, Bangalore, India shanthan.n2@gmail.com

simulate more possible road scenarios like varying brightness on the road, curved roads, hilly terrains. This technique of data synthesis also helps us to cover the scenario of car swerving away from the lane and recovering back.

For the deep learning network, we use the self driving solution proposed by Nvidia [1]. We use keras framework with tensorflow backend for training the model. We exploit the ***generators*** features of keras to read images from the disk, synthesize more data sample images on the fly and use these for training the network. The advantage of this method is that no extra memory space is required to store the augmented images.

We drove the car on the race track for 3 laps and this resulted in around 6500 samples with corresponding steering angles. For each sample we get three images, one each from center, left and right car cameras. We want to mention the fact that we use the images from center as well as the left and right cameras, to increase the variation in the data space. This helps to avoid overfitting.

For training the network we use adam optimizer which does not require much manual tuning of hyperparameters. We choose a learning rate of 0.0001 and train the network for 10 epochs with 200 samples per batch and 20000 samples per epoch.

3 Solution Design Approach

The overall project can be split into following parts:

- Data collection.
- Preprocessing, augmentation of the data.
- Defining and training the CNN.
- Validating the network model.

3.1 Data Collection

I used the car simulator provided by Udacity, for this project, to collect the image data. I collected the data from both the tracks, namely, the first track which is flat and the second one, which has a hilly terrain.

I used the computer keyboard control to drive around the tracks. I drove for 3 laps, to ensure that sufficient data is collected. Even though it?s the same track for 3 laps, there is slight variation in the driving which gives more data samples with more generalized behavior and avoids overfitting.

I drove over the track, staying right in the middle of the lane, thus steering angles remaining close to 0. I used preprocessing (described next) to generate data for recovering if car deviates from the lane.

I was able to generate around 20,000 samples from the centre, left and right cameras, after driving through the tracks.

Note that the simulator takes the steering angle as negative if car turns left and positive if the car turns right. This is just a convention to be followed.

3.2 Preprocessing

Although I drove for 3 laps and collected thousands of images, I ran the images through the training pipeline and observed that the validation loss does not decrease over epochs and also the car crashes by going straight in curved roads. This necessitated me to take a step back and think about this problem. I realized that the data I had collected has majority of samples with car staying on the lane (with 0 steering angle) and hence this behavior. To ascertain this, I plotted the histogram of the steering angles, shown in Fig. 1.

I did a bit of literature survey and motivated by [2], I decided to adopt some preprocessing on the images before feeding them to the network for training. The preprocessing pipeline is described in Fig. 2.

3.2.1 Adding angle offset

For the preprocessing steps mentioned below we add an offset to the steering angle. As mentioned earlier, we make use of the ledt and right camera images as well. Since the left and right cameras are away from the lane center, using these images simulate the scenario of lane recovery is the car has swerved from the center. For this recovery, we need to add an offset to the steering angle which is positive for left camera images and negative for right camera images. The magnitude of the offset is approximated using the radius and arc relationship in the circle, given by,

$$s = r\theta \tag{1}$$

where, s is the distance of the left/right camera from lane center, r is the

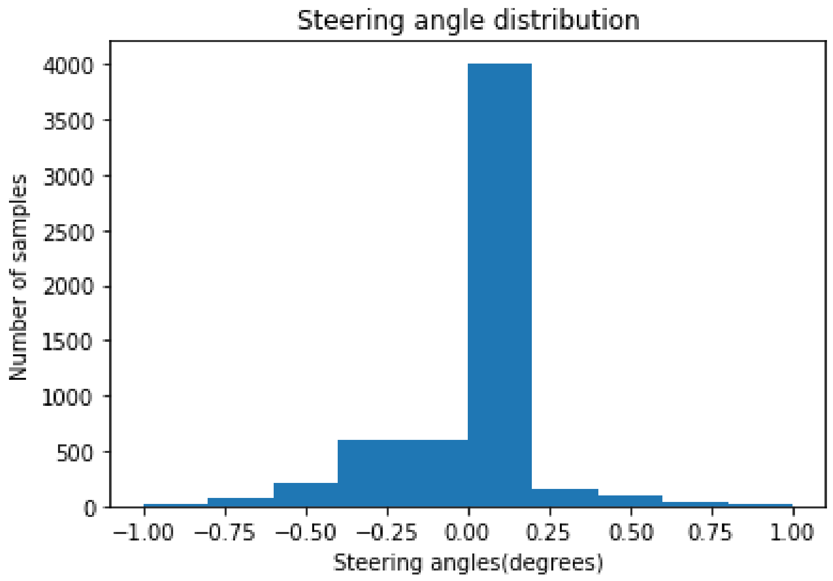

Figure 1: Steering angle distribution acrosss data samples. Note that majority of the images have 0 steering angles.

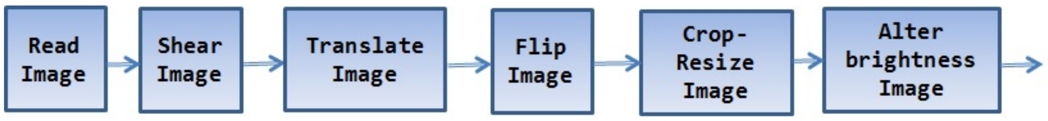

Figure 2: Preprocessing pipeline. The data images are passed through a set of transformations before feeding to the network for training

distance before coming back to lane center and θ is the offset angle that we add. This is illustrated in Fig. 3. We choose empirical values for s and r.

We use the same approximation for most of the preprocessing steps.

3.2.2 Read Image

We sample the data images to read a random image. The random image could be center, left or right camera image. This is illustrated in Fig.4.

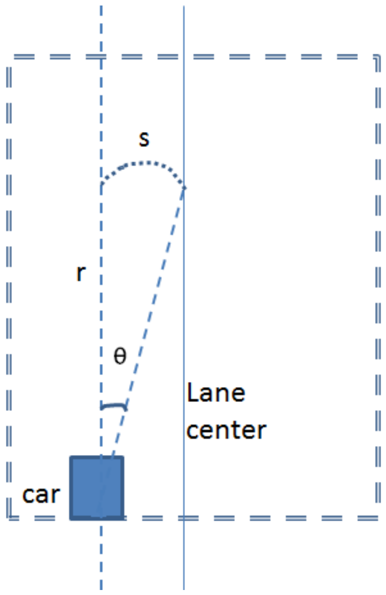

Figure 3: Steering angle offset approximation. We add an offset to steering angle for left and right images, as well as other preprocessing steps.

3.2.3 Shear Image

I apply the shear transformation on the image. In affine transformation, all parallel lines in the original image will still be parallel in the output image [3]. This transformation is applied on the lower half of the image which makes the straight line appear curved. I also add an offset angle corresponding to the amount of shear. This transformation simulates the behavior of car swerving from the lane and trying to recover back.

3.2.4 Translate Image

Next, I apply the translation operation on the sheared image. This simulates the effect of car being away from the lane center. Corresponding to the laterally shifted image, we also adjust the steering angle by adding an offset.

3.2.5 Flip Image

The images that I generated by driving through the track had greater number of samples where the car turns left(negative steering angle). So, to evenly

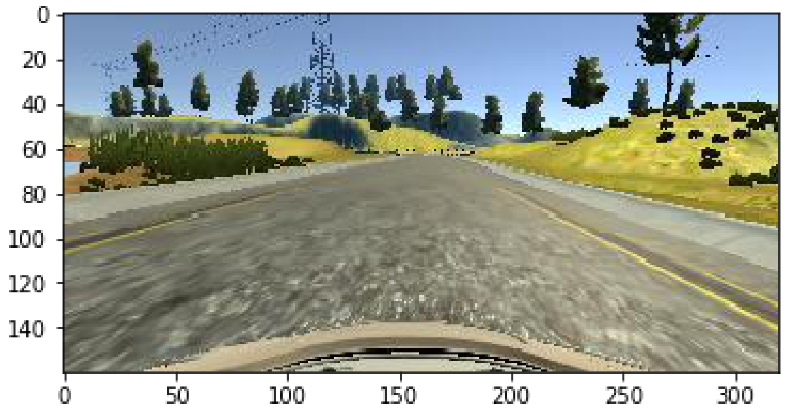

Figure 4: Read an arbitrary image. In the first step of preprocessing, an arbitrary image is read from the data samples, which could be from center, left or right camera.

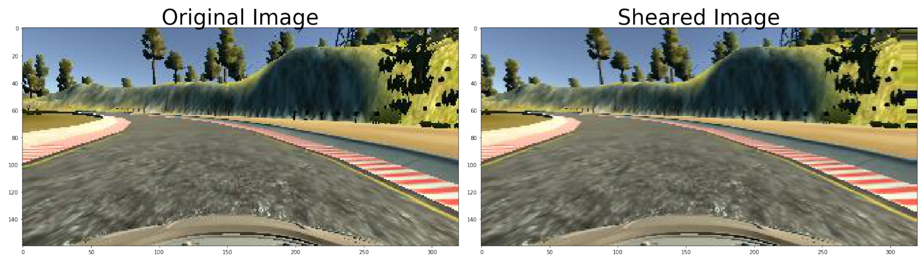

Figure 5: Sheared image. The input image is passed through an affine transformation which shears the bottom half of the image.

distribute the data, I flip the images (based on tossing of a coin), to generate more cases with positive steering angles. Flipping is done around the vertical axis.

3.2.6 Crop and resize Image

I crop the images to remove a portion of top and bottom of the image. The top will contain the horizon and the bottom contains the bonnet of the car. Both of them do not play a role in deciding the steering angle. Removing this redundancy simplifies the job of the network in choosing the best features for predicting the steering angle.

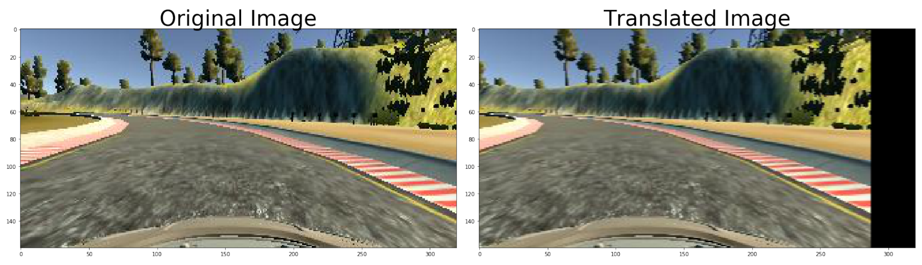

Figure 6: Translated image. The input image is passed through a translate operation.

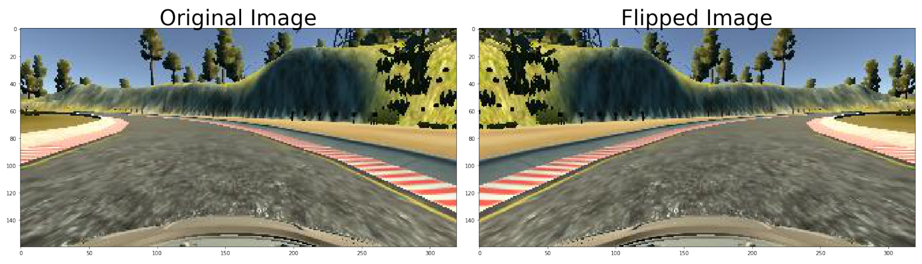

Figure 7: Flip image. The input image is passed through a flip operation which rotates about the vertical axis.

3.2.7 Alter brightness Image

In the last step of preprocessing I alter the brightness of the image. This step simulates driving the car in different lighting conditions and adds variation to the training data. In order to alter the brightness, I convert the image from RGB to HSV format, to modify the *V-value* channel.

3.3 Model Architecture

For the network I used the Nvidia solution for self driving cars [1], which is a reasonable size network for predicting the steering angle. I felt this is a very good point to start with, because, the authors have experimented with several network parameters before arriving at this network. As the authors mention in the paper, they do not explicitly decompose the problem into lane marking detection, path planning, control etc, instead they let the network figure out the best features required for predicting the steering angle. The other advantage of this network is that it is reasonably small network

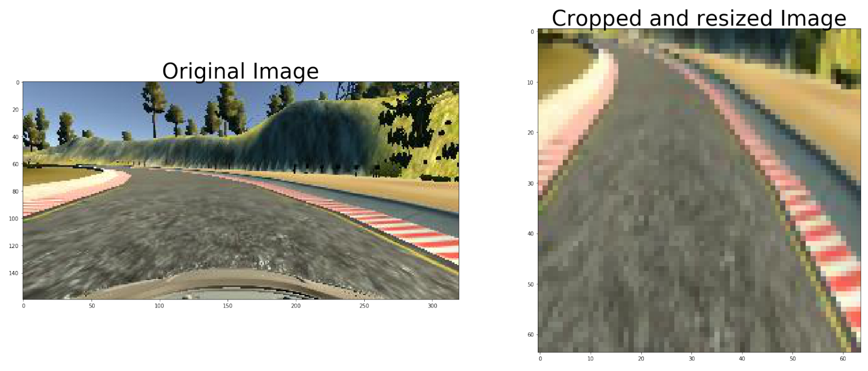

Figure 8: Crop and resize image. The input image is passed through a crop and resize operation which crops and resizes the image.

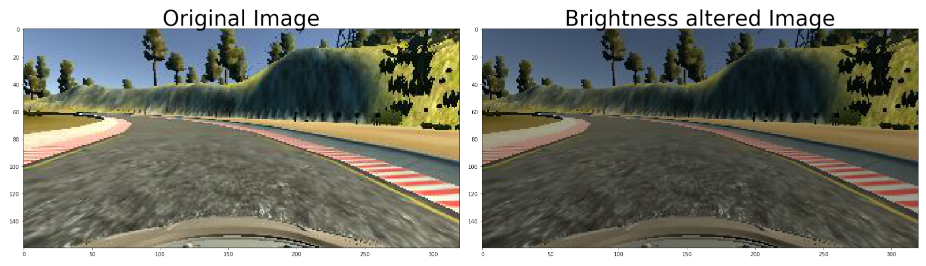

Figure 9: Brightness altered image. The input image is passed through an brightness alter transformation which changes the brightness by a random value.

allowing us to train with the limited data available. I had to make minor modifications to the network in terms of input normalization and adding extra dropout layers, to make the network perform better.

3.3.1 Split into training and validation data

In order to judge the performance of the network, I split the data that I had collected into training and validation, in the ratio of 80:20.

3.3.2 Avoid overfitting

I trained the Nvidia network without any modifications in the network layers. I observed that the validation loss was not decreasing with epochs, although training loss was decreasing. This appeared to be the problem of overfitting

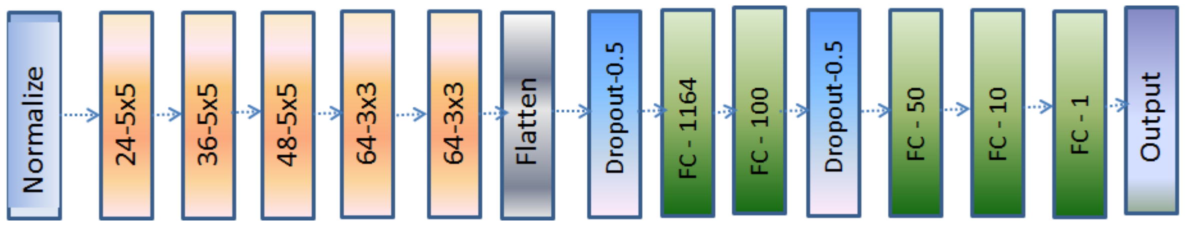

Figure 10: Network model. We make minor modifications to the Nvidia network by introducing few dropout layers in the last few layers of the network.

and so I added a couple of dropout layers in the last few layers of the network, with which validation loss started decreasing over epochs.

3.4 Model Validation

This was the final step, where I had to run the simulator in autonomous mode and drive the car on the test track. I saw that the car crashes after some time, because of a large turn. I sensed that this could be because of large offset being added to the steering angle in the preprocessing step. I went back and decreased the offsets being added to the steering angles at all the phases of preprocessing and then retrained the network. With this change, the car was successfully able to drive without crashing.

4 Rubric Criteria

In this section we address the points related to rubric criteria, for this project.

Are all required files submitted?
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 convolutional neural network(CNN).
* model.h5 containing the network architecture.
* writeup_report.pdf summarizing the results.

Is the code functional?

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.json

Is the code usable and readable?

The model.py file contains the code for training and saving the convolutional neural network. It also contains the pipeline that I used to train and validate the model. Also, I have added comments to explain how the code works.

Has appropriate model architecture been employed for the task?

Yes, I have chosen the Nvidia solution for the model architecture, which has been tested well on real world data.

Has an attempt been made to reduce overfitting of the model?

Yes, I have introduced a couple of dropout layers in the final layers of the Nvidia CNN, which reduce the overfitting.

Have the model parameters been tuned appropriately?

I have used adam optimizer, thereby not worrying much about hyperparameters tuning.

Is the training data chosen appropriately?

Yes, I have driven the car on both the tracks (flat and hilly) for several laps, so that I generate wide variety of data for the training process. Also I have stayed on the track while driving (thereby 0 steering angle) and then used preprocessing to simulate turning left, right, and lane recovery. Also, I am using images from the left and right cameras, as well, to introduce variability in the captured data.

Is the solution design documented?

Yes, this writeup has detailed explanation of the solution proposed for the project.

Is the model architecture documented?

Yes

Is the creation of the training dataset and training process documented?

Yes, I have a section in the writeup, which explains about the data collection process.

Is the car able to navigate correctly on test data?

Yes, The car is able to drive on the test track successfully.

5 CONCLUSIONS

In this project we train a CNN to drive a simulator car on the race track. We colect data by driving through the track (training track) for a couple of laps and then use the data to train the network. We then validate our trained network model on the test track and are successful in driving through the test track without any crash.

References

[1] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, K. Zieba, "End to End Learning for Self-Driving Cars", Apr 2016. URL: https://arxiv.org/abs/1604.07316

[2] V. Yadav, "https://chatbotslife.com/using-augmentation-to-mimic-human-driving-496b569760a9#.7v3qtf1w1"

[3] "http://docs.opencv.org/3.0-beta/doc/py_tutorials/py_imgproc/py_geometric_transformations.html"