SELF DRIVING CAR NANODEGREE – TERM 1 / PROJECT3 – Behavioral Cloning

ADITHYA RANGA
March 2016 – COHORT STUDENT

The goals / steps of this project are the following:

- Use the simulator to collect data of good driving behavior
- Build, a convolution neural network in Keras that predicts steering angles from images
- Train and validate the model with a training and validation set
- Test that the model successfully drives around track one without leaving the road
- Summarize the results with a written report

APPROACH:

Firstly I choose to try training and validation of a simple LeNet architecture on the data provided by Udacity. This worked Ok but the data was not generalized and only confined to track – 1 and also the network architecture was not so powerful.

I choose to generate my own additional data using the simulator for both the tracks – while making sure I drive in both the directions to have a good data set with equal distribution of data with left and right steering angles. Additional data augmentation and pre-processing techniques were used to generate more training and validation data sets. I finally choose to train the images using the NVidia's model architecture with some modifications and techniques to reduce overfitting.

The final project package includes:

Model.py – Code to train the network and save the weights

Drive.py – used to drive the car autonomously in the simulation (modification – added data pre-processing function for input images before using model weights).

Video1 –track1 autonomous driving.

Video2 – track 2 autonomous driving.

DATA COLLECTION:

The dataset provided by Udacity is only from track – 1. To generalize the training data and make sure the model performs well on both the provided tracks the data was hence collected from both the tracks.

LEFT / CENTER / RIGHT CAMERA – TRACK 1

LEFT / CENTER / RIGHT CAMERA – TRACK 2

As shown in the figures as we are driving around the tracks data is recorded from all the three cameras: center, left and right. I did make use of all the three camera views for training. This is implemented using a constant offset correction in the code. (*steering_corr = 0.2).*
Such that:

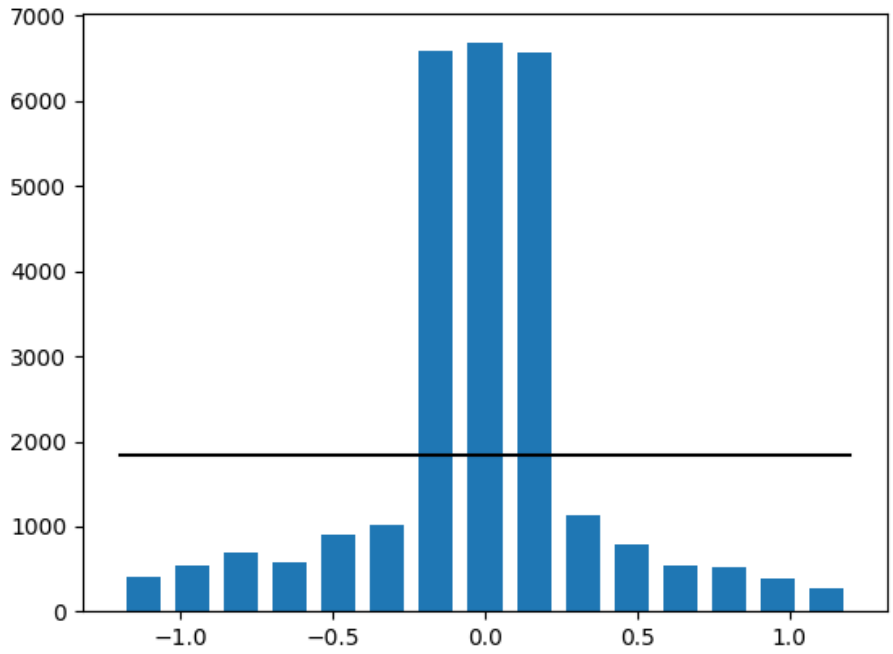
Steering_angle (right) = Angle – steering_corr
Steering_angle (left) = Angle + steering_corr

The generated data from the simulator consists about 27663 samples from all the three camera views used later towards training and validation sets.
I did drive 4 time (2 on each track):

- Drive 1: track 1 normal
- Drive 2: Track 1 – in reverse direction (to make sure we have equal left and right turns as track one normal direction is biased more with left only turns)
- Drive 3: Track 2 – normal
- Drive 4: Track 2 – reverse direction

DATA DISTRIBUTION:

Using the numpy.histogram function I could generate the histogram plot for the simulation data I generated. I did randomly choose to divide the steering angles into like 15 bins. The histogram shows that most of the generated data is biased around straight driving or very small steering angle inputs. This could bias the training and the model might not be work well on curved roads for track – 2 especially.



I went with the original data set and the distribution as shown without throwing out any data points to give a try with the following approach.

DATA SPLIT & TRAINING DATA AUGMENTATION:

Before I augment any data I did shuffle the data and also split 20% of the data into a validation set using train_test_split from sklearn.model_selection. This way I was only going to augment my training data and not the validation set.

Length of training / Validation data after split: (22130 – Training) (5533 – Validation)

Data augmentation was a very important step here to generate additional data from the original data set. This is a step to reduce any overfitting and also helps generalize the model a bit. There are many methods to generate this augmented data set – change brightness, transform the training images, add jitter, flip images etc.
Here I implemented data augmentation by flipping the images about the vertical axis and likewise multiplying the steering angles by -1 (so we have to steering in the opposite direction).

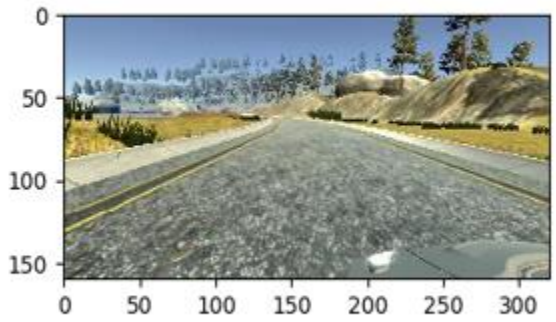
This operation to augment the images will make sure we have equal distribution of steering left and right data.

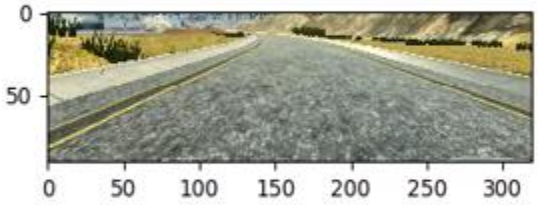
Length of training data after augmentation: 44260

DATA PROCESSING:

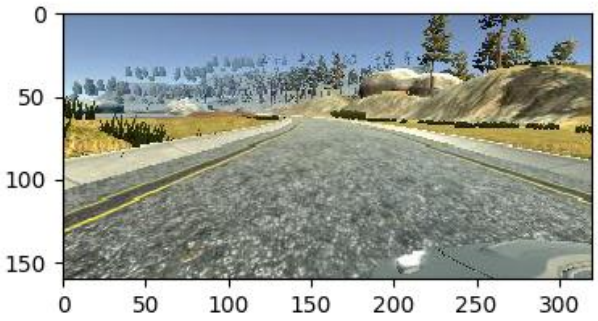
First of all I cropped the images from all the cameras by removing the top 20 pixels and also the bottom 50 pixels. The reason for cropping the top 20 pixels is mainly because it consists of background noise from trees and sky. The bottom 50 pixels are covered with the hood of the vehicle and not much feature left to extract.

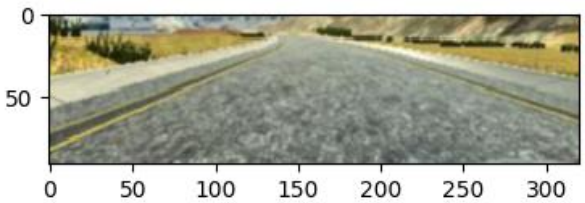
The figures below shows the original and cropped images:





I used a gaussian filter in the processing stage of filter size 3x3 to filter any additional noises in the images. This step was considered after looking closely at some of the noisy images from left and center camera especially on track 2.
The figure below shows gaussian filter being applied on the cropped original image:

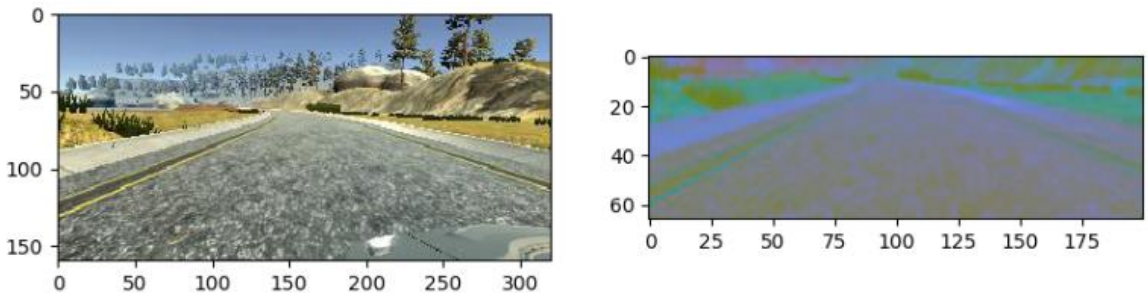




Finally the network architecture used in this project is from the NVidia's end – to – end learning model (https://devblogs.nvidia.com/parallelforall/deep-learning-self-driving-cars/). The input for

this network architecture are images of size 200x66x3 and are split into YUV plane. Hence I included the resizing and conversion from BGR to YUV plane step during pre-processing.

The figure below shows images resized to 200x66x3 and YUV:

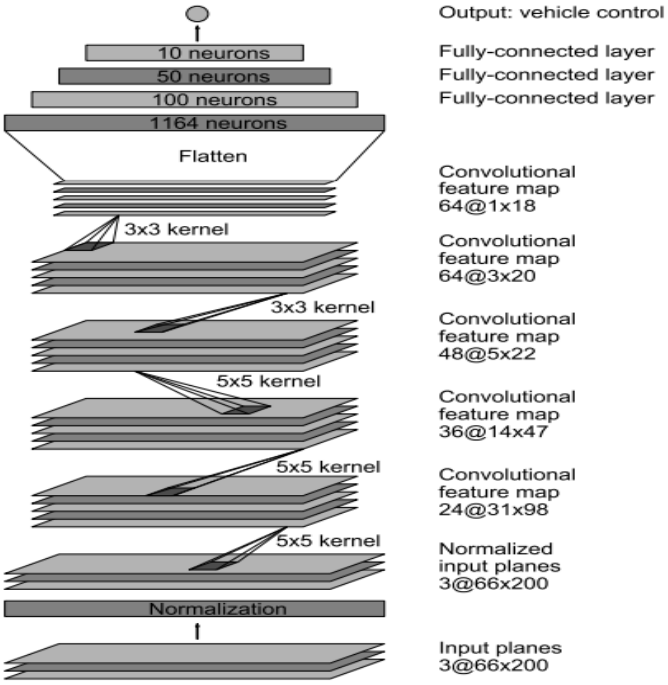


This was the final input for the network architecture. All the training and validation datasets went through this initial data pre-processing step.

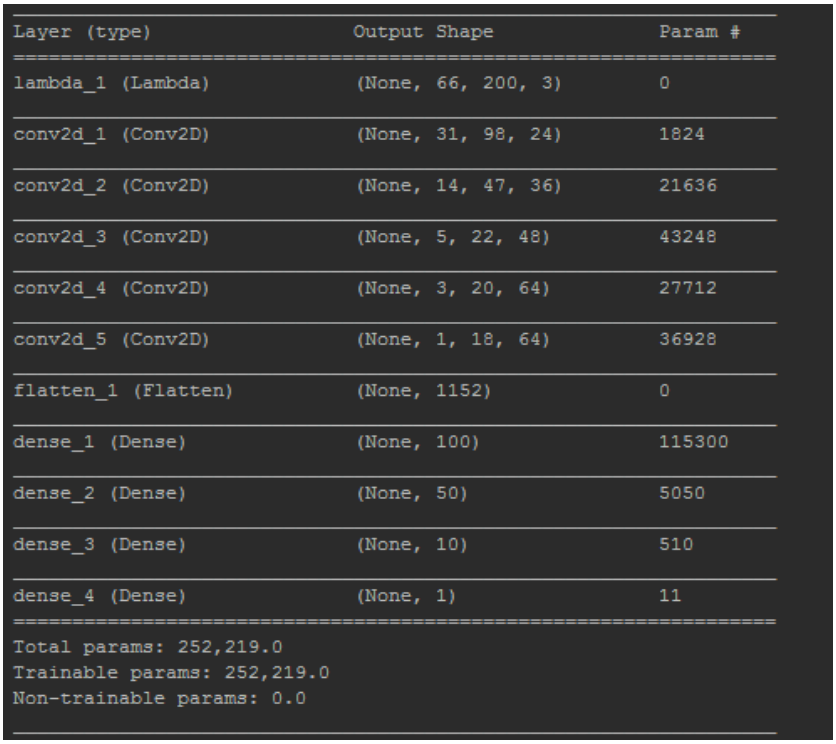
Importantly this function is included in the drive.py code too – because we need to perform the same conditioning on the new images that use the model weights during validation.

SECTION – 3: MODEL ARCHITECTURE & TRAINING:

The final model architecture that was used for this project is from the NVidia's published paper on end-to-end learning. The architecture contains a total of 9 layers with first 5 convolutional layers followed by fully connect ones. Following picture summarizes the model architecture:



The following image shows the model summary as implemented with the total number of parameters (252,219). The NVidia paper does not mention about any strategies to reduce overfitting and use of any activation functions in the layers. But for this final model I used “ELU” activation function on each convolutional and fully connected layer.



Layer (type)	Output Shape	Param #
lambda_1 (Lambda)	(None, 66, 200, 3)	0
conv2d_1 (Conv2D)	(None, 31, 98, 24)	1824
conv2d_2 (Conv2D)	(None, 14, 47, 36)	21636
conv2d_3 (Conv2D)	(None, 5, 22, 48)	43248
conv2d_4 (Conv2D)	(None, 3, 20, 64)	27712
conv2d_5 (Conv2D)	(None, 1, 18, 64)	36928
flatten_1 (Flatten)	(None, 1152)	0
dense_1 (Dense)	(None, 100)	115300
dense_2 (Dense)	(None, 50)	5050
dense_3 (Dense)	(None, 10)	510
dense_4 (Dense)	(None, 1)	11

Total params: 252,219.0
Trainable params: 252,219.0
Non-trainable params: 0.0

To reduce overfitting I choose to include l2 normalization (0.001) in each of the convolution and fully connected layers.

Optimizer – Adam

Number of Epochs: 20 (Going over 20 / 23 I see that I was running into overfitting issues)

Batch Size: 100

I choose to use Keras ‘fit’ method to train the images. (Could implement the fit_generator method and use the generator on each batch).

ADDITIONAL:

To further improve the model the following could be done:

- Generate more training data – using simulator or additional augmentation techniques.
- Try Dropouts while using additional augmented data – to avoid overfitting.
- Image by image cropping and filtering for better feature extraction.