CarND-Project 3
Ayanangshu Das
April 17, 2017

Use Deep learning Cloning Driving Behavior

Data Preprocessing

Two steps
1. Lambda layer for image normalization: Divided each image by 255 which is the maximum value of an image pixel

2. Mean centralization: Once the image is normalized between 0 and 1, I mean centered the images by subtracting 0.5 from each image which will shift the mean from 0.5 down to zero

3. In order to ascertain how the model was working, I split the images and the steering angle data set into training and validation set in a ratio of 80:20

Doing the above image normalization reduced both the training and validation losses. However after few epochs, the validation loss started oscillating up and down hinting that the model might be overfitting the training data.

To combat overfitting, I introduced the following:

- Added dropout into the model
- Added maxpooling into the model
- Added subsampling in the convolutional layer
- Shuffled the data set across epochs

Final Model Architecture

I approached to use the LeNET architecture. The original LeNET architecture takes a 32x32x1 image. However the image in the training data is of the shape 160x320x3.

One good thing about convolutional networks is that they work on a wide range of input image sizes.

I used Mean Squared Error as the loss function instead of the cross entropy function as this model is a regression model instead of a classification model; that is why MSE is an appropriate loss function

What I intend to do is to minimize the error between the actual steering angle and the steering angle that the network predicts. And MSE is a good loss function for this.

By default, Keras train for 10 epochs. However, I see that the validation loss reduces for first 7 epochs and then climbs.

Since LeNET was a powerful architecture, I trained the model for 5 epochs. The loss decreased consistently, however slightly, throughout all the 5 epochs.

NVIDIA architecture

I used the NVIDIA architecture. The network consists of a normalization layer, which I already implemented with lambda normalization. This architecture consisted of 5 convolutional layer and four fully connected layer. The convolutional layer consisted of image subsampling which reduced the size of the images.

The model includes RELU layers to introduce nonlinearity.

The final model architecture (model.py) consisted of a convolution neural network with the following layers and layer sizes:

Conv1: Input of 160, 320, 3 and output of 80, 160, 24

The entire structure is as below:
Conv2:40, 80, 36
Conv3:20, 40, 48
Conv4:19, 39, 64
Conv5:18, 38, 64
FC1:1164
FC2:100
FC3:50
FC4:10

Total params: 47,112,567
Trainable params: 47,112,567
Non-trainable params: 0

I ran the simulator to see how well the car was driving around track one. There were places where the vehicle fell off the track... to improve the driving behavior in these cases, I took the following steps

Data Augmentation

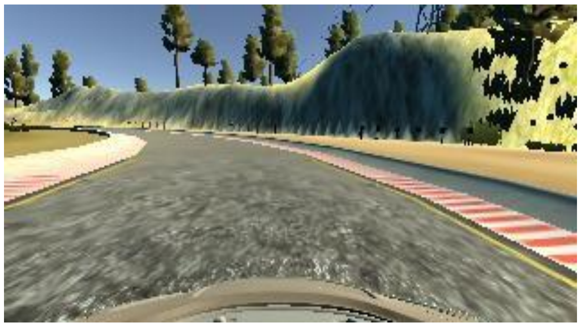
There are many ways to expand the training set. I chose to flip the existing images horizontally and add to data set.

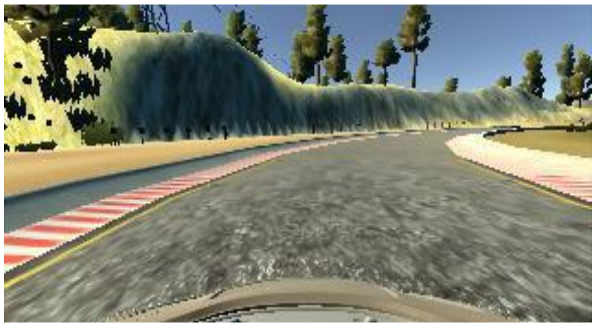
This caused the car to have a balanced data set which will enable the model to steer clockwise as well as counter clockwise.

I wanted to flip the images so that the training data set consists of generalized images.

This flipping caused the training data set to get images just mirror images of what was captured which helped the model to handle the turns just in the opposite directions.

Flipped Images



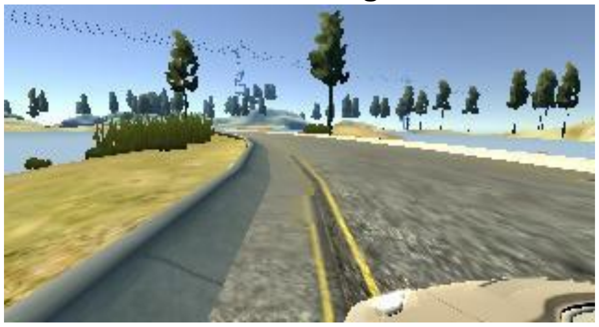


Using multiple camera images

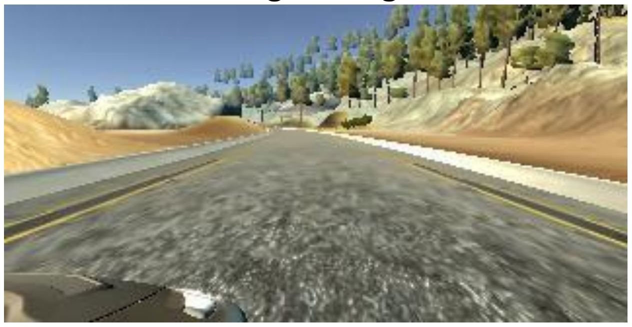
I wanted to use the side camera images for training too. However, I had to add a correction to the left_images and subtract a correction to the right images. I had to fine-tune the correction factor. I tried with values 0.2 and 0.15. However, it didn't give me satisfactory result.

Since I could not identify the correct correction factor, adding the left and right camera images couldn't help my model. Hence I didn't use the side camera images and relied on having the center images only.

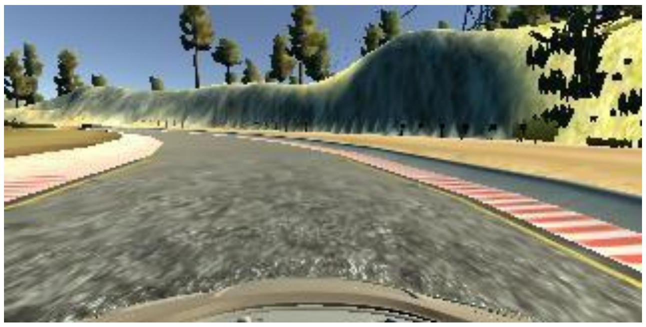
Left Image



Right image



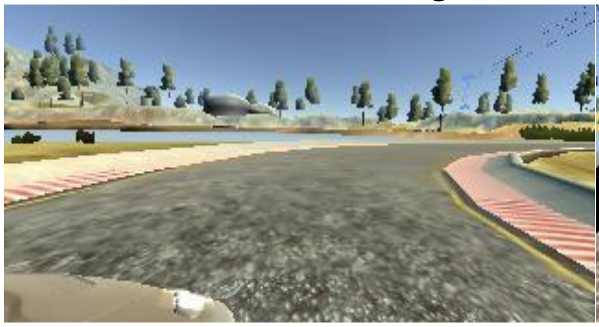
Center Image



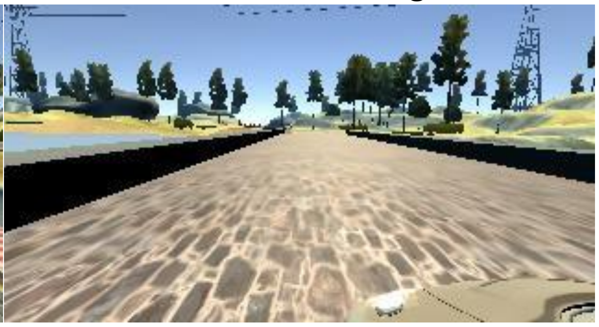
Further refinement

I collected more training data. I gathered images for situations where the car veered too much to one side and then recovered to the center.

Car Veered from Left to Right

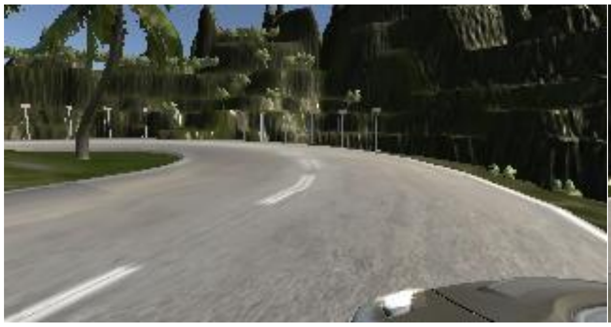


Car Veered from Right to Left

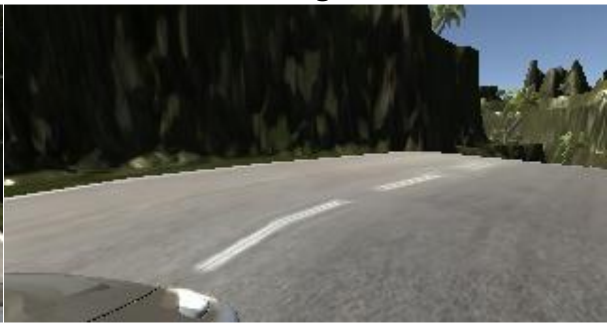


Data Collected from the other Track

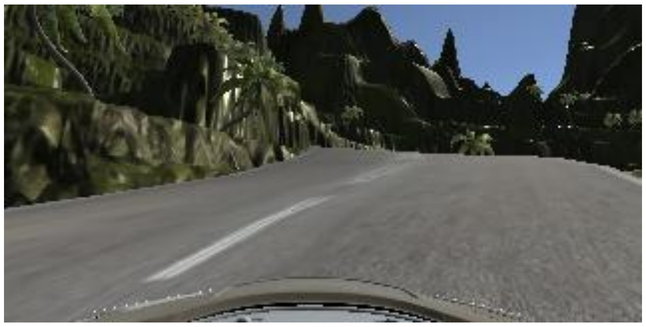
Left



Right



Center



After the collection process, I had **8036** number of data points. I randomly shuffled the data and put **20%** of the data to the validation set.

Hence the number of data points in the training data set became **6428** and the in the validation set it became **1608**.

I didn’t use python generators in my pipeline as the number of images was not huge and was fine to load in memory and train.

I used an adam optimizer so that manually training the learning rate wasn't necessary.

I could see that both the training and validation loss has decreased to .0088 and .0098 respectively.

I wanted to see if adding more training data helped the model, which to my surprise, didn’t.

At the end of the process, the vehicle is able to drive autonomously around the track without leaving the road

Next Steps

1. My model didn’t do well on the other track. I plan to do the following
    a. Collect more data
    b. Add left and right images
    c. Need to check if training the data with greyscale images helps
    d. Add inception layer

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 convolution neural network
* writeup_report.md or writeup_report.pdf summarizing the results
* video.mp4 to show the video of how the car ran in my model in autonomous driving
* clearer video link: the link to a shared drive of a clearer video:
https://drive.google.com/open?id=1IxU9oTpBQAUHTn-CpbSY5iGIjdU0UPPUEA