Behavioral cloning project:

Overview:

The overall idea of this project is to clone the human behavior to drive the car autonomously.

In this project, a deep convolutional neural network was deployed to clone driving behavior, this will be used to train, validate and test a model using Keras. The model will output a steering angle to an autonomous vehicle.

Udacity has provided a simulator where a car can be steered around a track for data collection. Image data and steering angles are used to train a neural network and then use this model to drive the car autonomously around the track.

The steps performed are:

* Use the simulator to collect data of good driving behavior, especially recovery paths

* Build a convolution deep neural network in Keras that predicts steering angles from images

* Train and validate the model with a training and validation set

* Test the model which can predict steering angle for images sent from simulator running in autonomous mode.

* Summarize the results with a written report

Files submitted and how to run the project:

My project submission includes following files:

* model.py: Contains code for a deep neural network to train and validate

* drive.py: For driving the car in autonomous mode

* model.h5: A trained convolution neural network for cloning the driving behavior on TRACK 1

* output_video.mp4: Video to drive the car on Track 1

* Behavioral_cloningREADME.pdf: Descriptive writeup on the project

Submission includes functional code

Using the Udacity provided simulator and my drive.py file, the car can be driven autonomously around track1 by executing

python drive.py model.h5

Submitted code is readable and usable:

The model.py file contains the code for training and saving the convolution neural network and also the image preprocessing pipeline.

Model architecture:

1. Design Approach

My first step was to use a convolution neural network model similar to leNet, I thought this model might be appropriate and it is simple but then I learned as part of this self driving course that leNet was developed within the mind set of classifying digits and can take only 32x32 pixel images, the ability to process higher resolution images require larger and more convolutional layers and I found this model was overfitting during the validation.

In contract, PilotNet from Nvidia was developed on the base to find the region in input images which makes the steering decisions, they call it the salient objects, structure in camera images those are not relevant to driving, as they described in the paper here, https://arxiv.org/pdf/1704.07911.pdf, also this capability is derived from data without the need of hand-crafted rules.

In this self driving regression network we wanted to minimize the mean square error instead of reducing the cross-entropy which is for the classifying network and I decided to use PilotNet which gave better result in the given project.

In order to gauge how well the model was working, I split my image and steering angle data into a training and validation set(20%). The network must also learn how to recover from any deviation or the car will drift off the road slowly. The training data is therefore augmented with additional images from left and right camera which shows the car in different shifts from the center of the lane and rotations from the direction of the road.

At the end of the process, the vehicle is able to drive autonomously around track1 without leaving the road.

2. Final Model Architecture

I used the NVIDIA'S PilotNet cnn network with an extra dropout layer added after the convolutional layers. According to NVIDIA, the convolutional layers are designed to perform feature extraction, and are chosen empirically through a series of experiments that vary layer configurations. They use strided convolutions in the first three convolutional layers with a 2×2 stride and a 5×5 kernel, and a non-strided convolution with a 3×3 kernel size in the final two convolutional layers.

I found it helped reducing the overfitting of these model in the given track by adding a dropout layer after the flatten layer. Filters depth in this network is between 24 to 64.

Here is a visualization of the Architecture

Layer(type) Output Shape

lambda-1 (Lambda) (None, 160, 320, 3)
cropping2d-1 (Cropping2D) (None, 65, 320, 3)
conv2d-1 (Conv2D) (None, 31, 158, 24)
conv2d-2 (Conv2D) (None, 14, 77, 36)
conv2d-3 (Conv2D) (None, 5, 37, 48)
conv2d-4 (Conv2D) (None, 3, 35, 64)
conv2d-5 (Conv2D) (None, 1, 33, 64)
flatten-1 (Flatten) (None, 2112)
dropout-1 (Dropout) (None, 2112)

dense-1 (Dense) (None, 100)
dense-2 (Dense) (None, 50)
dense-3 (Dense) (None, 10)
dense-4 (Dense) (None, 1)

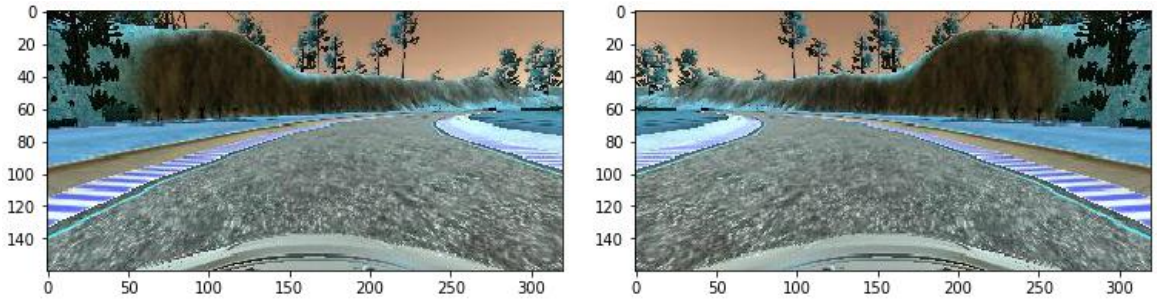
The model includes RELU layers to introduce nonlinearity and the data is normalized in the model using a Keras lambda layer

3. Data processing pipeline:

Data Augmentation:

For cropping the Image I use keras Cropping2D function to Crop the tree, hills, Sky from top 70 and the hood of the car from bottom 25 to avoid any extra noise on the fly. I also used left and right camera images with a correction factor 0.2 on steering angle.

I added more data by flipping the images for those where the absolute value of steering is > 0.43 to prevent overfitting . An effective technique for helping with the left or right turn bias involves flipping images and taking the opposite sign of the steering measurement. For example, here is an image that has been flipped:



Geometric Translation of an Image I shifted the original images on right, left , up and down but after experiment I did not find this helpful with the training data I collected, one of the reason could be when I shift the image, it replaced the shifted location with a black background which I should avoid by resizing the image.

After data processing, I had for Track 1, 19286 images for training and 4822 images for validation

I randomly shuffled the data and used python generator to feed the data into the model in a memory optimized batched fashion with a batch size 64.

I used this training data for training the model. The validation set helped determine if the model was over or under fitting. Number of epochs was 5 for track 1. Adam optimizer was used to avoid manual training of the learning rate.

Conclusion:

I used Udacity provided training data for simplicity and also with the assumption that it covers all the recovery paths and corner cases of human driving. I learned that pilotNet works better with an added dropout layer and also would like to appreciate the GPU time to speed up the train process of the model over cpu. Due to limitation on provided GPU time, I didn't train the model with training data from track2. Submitted model works only well with track1.

The training and validation loss looks like:

```
Epoch 1/5
19286/19286 [==============================] - 1880s 97ms/step - loss: 0.0074 - val_loss:
0.0146
Epoch 2/5
19286/19286 [==============================] - 1879s 97ms/step - loss: 0.0022 - val_loss:
0.0144
Epoch 3/5
19286/19286 [==============================] - 1887s 98ms/step - loss: 0.0015 - val_loss:
0.0144
Epoch 4/5
19286/19286 [==============================] - 1892s 98ms/step - loss: 0.0013 - val_loss:
0.0153
Epoch 5/5
19286/19286 [==============================] - 1884s 98ms/step - loss: 0.0011 - val_loss:
0.0142
```