Behavioral Cloning

Udacity Self-Driving Car Nanodegree Program

Sanyam Agarwal - June 1, 2018

Behavioral Cloning Project

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

Rubric Points

Here I will consider the rubric points individually and describe how I addressed each point in my implementation.

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 (this is modified, sets speed to 14MPH and changed image from RGB to YUV)
- Model.h5 containing a trained convolution neural network for cloning the driving behaviour on TRACK 1
- run1.mp4 video to drive the car on Track 1
- README.md summarizing the results

Submission includes functional code 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.h5
```

The model.py file contains the code for training and saving the convolution neural network. The file shows the pipeline I used for training and validating the model, and it contains comments to explain how the code works.

Model Architecture and Training Strategy

a) An appropriate model architecture has been employed

First I applied a very basic network with just a flatten layer and an output unit of 1 to make sure everything is working in a sense of reading the images and the corresponding steering angles and use keras to fit the model; as this is a regression network instead of a classification network, mean square error was used as the loss function instead of cross entropy. Idea is to minimize the difference between the steering measurement this network predicts and the ground truth steering measurement, I shuffle the data and use 20% of the data for validation set.

Then I tried with LeNet CNN network which takes 32x32x1 image only but as the Convolutional layer from keras works with wide ranges of images, I able to use the images here which are 160x320x3 without re-sizing them.

Finally I moved to PilotNet CNN published by NVIDIA.

b) Attempts to reduce overfitting in the model

The model was modified with a dropout layers to reduce overfitting.

It was trained and validated on different data sets to ensure that the model was not overfitting (see Data augmentation i.e. flip the image during cornering, steering angle adjustment, applied Gausian Blur/DownSample the image etc.). The model was tested by running it through the simulator and ensuring that the vehicle could stay and run smoothly on track

c) Model parameter tuning

The model used an adam optimizer, so the learning rate was not tuned manually

d) Appropriate training data

Initially I started training with the data provided by the project resource from Udacity which contains data from the track for complete track.

After some time working with the data I realized that the data was not sufficient and I started recording my training data and combined the resource data from Udacity and also the data which I recorded.

My data collection was mainly focused on:

- Smooth drive on curves

- Smooth drive on bridge

- Driving in reverse direction on track to make data generalized.

- Recovering from drifting off to side and then coming back to center.

The above data was recorded for multiple times during data recording.

After reading the data and collecting into single collection, I split the data into Train and Validation data, I use the 20% data for validation.

e) Solution 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 track without leaving the road.

f) 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-stride 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	Param #	Connected to
lambda_1 (Lambda)	(None, 160, 320, 3)	0	lambda_input_1[0][0]
cropping2d_1 (Cropping2D)	(None, 75, 320, 3)	0	lambda_1[0][0]
convolution2d_1 (Convolution2D)	(None, 36, 158, 24)	1824	cropping2d_1[0][0]
convolution2d_2 (Convolution2D)	(None, 16, 77, 36)	21636	convolution2d_1[0][0]
convolution2d_3 (Convolution2D)	(None, 6, 37, 48)	43248	convolution2d_2[0][0]
convolution2d_4 (Convolution2D)	(None, 4, 35, 64)	27712	convolution2d_3[0][0]
convolution2d_5 (Convolution2D)	(None, 2, 33, 64)	36928	convolution2d_4[0][0]
flatten_1 (Flatten)	(None, 4224)	0	convolution2d_5[0][0]
dropout_1 (Dropout)	(None, 4224)	0	flatten_1[0][0]
dense_1 (Dense)	(None, 100)	422500	dropout_1[0][0]
dense_2 (Dense)	(None, 50)	5050	dense_1[0][0]
dense_3 (Dense)	(None, 10)	510	dense_2[0][0]
dense_4 (Dense)	(None, 1)	11	dense_3[0][0]

Total params: 559,419
Trainable params: 559,419
Non-trainable params: 0

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

I prefer to use RELU instead of ELU based on the recent experiment bellow, https://ctjohnson.github.io/Capstone-Activation-Layers/ which summarized the comparison of different activation layers as below,

Activation Test Accuracy Training Time (per epoch)

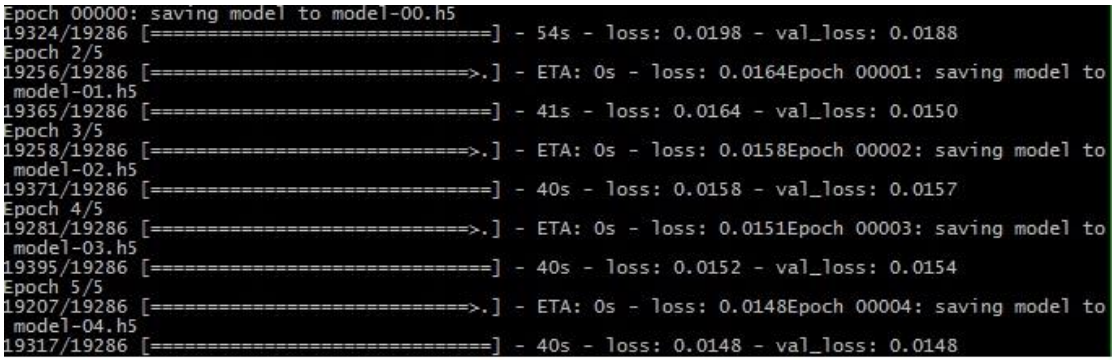
Relu	90.25%	76s
Elu	84.36%	76s
Soft Plus	02.85%	78s
TanH	83.41%	78s
Sigmoid	5.7%	90s
Linear	83.6%	131s
Soft Sign	85.95%	82s
Hard Sigmoid	5.7%	85s

g) Creation of the Training Set & Training Process

To capture good driving behavior, I first recorded two laps using clockwise and counter-clockwise. To reduce the bias towards cornering I added a small offset to the left side camera images and deducted from the right camera images. If the steering angle is more than 0.03 in any direction(left or right) and not steer in the right direction more than 0.30 , a flipped image was feed to the network to recover from over drifting. See the section 'Data Augmentation' bellow for details.

As explained in Udacity course material i.e. "Explanation of How Multiple Cameras Work", It is not necessary to use the left and right images to derive a successful model, Recording recovery driving from the side of the road is also effective.

I took the first approach and used the left and right camera images to record the recovery.



Converting the image from RGB to YUV (as suggested by Nvidia) and down-sample it (by applying a Gaussian Blur) makes the car drive much smoother.

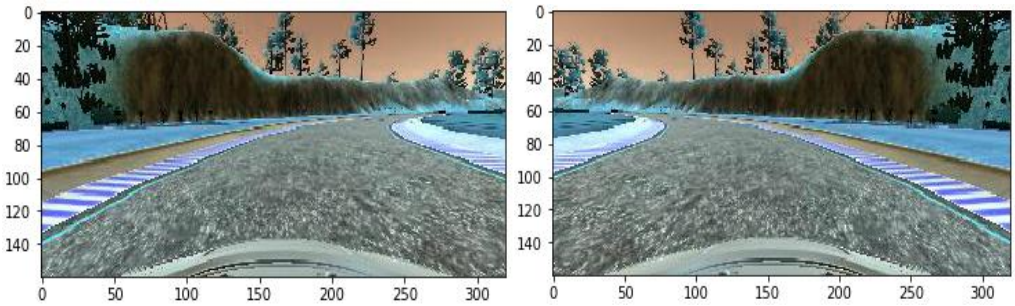
It is also imparitive to do the same conversion on drive.py where the image was received from the simulator in RBG format.

- **Data Augmentation:**

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.

Multiple Camera Images There are multiple cameras on the vehicle, and I used map recovery paths from each camera. For example, while the model was trained to associate a given image from the center camera with a left turn, it was also trained to associate the corresponding image from the left camera with a somewhat softer left turn and to associate the corresponding image from the right camera with an even harder left turn. This is applied on line 63 to 73 of model.py. This way, the car has been taught how to steer if the car drifts off to the left or the right.

Flipping the Images I added more data by flipping the images for those where the absolute value of steering is > 0.03 and < .30 (by experiment this value was selected) to prevent over-fitting . 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 then been flipped:



After the collection process, 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.

I used this training data for training the model. The validation set helped determine if the model was over or under fitting. The ideal number of epochs was 5 for the track.

Conclusion

I tried to keep the network simple like the pilotNet while cover both track. I would say Udacity 'Self Driving' course material for deep neural network and convolutional neural networks was quite good enough to clone the driving behaviour on both track successfully. It was also hinted where I could have make the final layer to output 3 things; the Steering Angle, speed and break measurement which then I could use to drive the car more dynamically with different speeds; faster in the case of straight road or slower by applying the break on a downhill situation. I also found it by adding a dropout layer allow me to train the network further and produce a stable driving.

Further coarse of action will be to run the model on Track2. Current model is underfit on track 2. Due to time crunch in office not able to work on Track 2 much.

References

- https://www.youtube.com/watch?v=rpxZ87YFg0M
- http://selfdrivingcars.mit.edu/
- http://images.nvidia.com/content/tegra/automotive/images/2016/solutions/pdf/end-to-end-dl-using-px.pdf
- http://jacobgil.github.io/deeplearning/vehicle-steering-angle-visualizations
- http://medium.com/udacity/teaching-a-machine-to-steer-a-car-d73217f2492c
- http://chatbotslife.com/using-augmentation-to-mimic-human-driving-496b569760a9