Udacity Self-driving Car P3: Behavioral Cloning

Behavrioal 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.

Files Submitted & Code Quality

1. Submission includes all required files and can be used to run the simulator in autonomous mode

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.pdf summarizing the results

2. Submssion 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.json --data_path [root path to training data]
```

3. Submssion code is usable and readable

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

1. An appropriate model arcthiecture has been employed

The network is very similar to Nvidia's paper about end-to-end training of self-driving car. The model is defined in function get_model(input_shape) in model.py

Here is a snapshot of the model structure:

Layer (type)	Output Shape	Param #	Connected to
bn1 (BatchNormalization)	(None, 66, 200, 3)	12	batchnormalization_input_1[0][0]
conv1 (Convolution2D)	(None, 31, 98, 24)	1824	bn1[0][0]
bn2 (BatchNormalization)	(None, 31, 98, 24)	96	conv1[0][0]
act1 (ELU)	(None, 31, 98, 24)	0	bn2[0][0]
conv2 (Convolution2D)	(None, 14, 47, 36)	21636	act1[0][0]
bn3 (BatchNormalization)	(None, 14, 47, 36)	144	conv2[0][0]
act2 (ELU)	(None, 14, 47, 36)	0	bn3[0][0]
conv3 (Convolution2D)	(None, 5, 22, 48)	43248	act2[0][0]
bn4 (BatchNormalization)	(None, 5, 22, 48)	192	conv3[0][0]
act3 (ELU)	(None, 5, 22, 48)	0	bn4[0][0]
conv4 (Convolution2D)	(None, 3, 20, 64)	27712	act3[0][0]
bn5 (BatchNormalization)	(None, 3, 20, 64)	256	conv4[0][0]
act4 (ELU)	(None, 3, 20, 64)	0	bn5[0][0]
conv5 (Convolution2D)	(None, 1, 18, 64)	36928	act4[0][0]
bn6 (BatchNormalization)	(None, 1, 18, 64)	256	conv5[0][0]
act5 (ELU)	(None, 1, 18, 64)	0	bn6[0][0]
flat (Flatten)	(None, 1152)	0	act5[0][0]
FC1 (Dense)	(None, 100)	115300	flat[0][0]
bn7 (BatchNormalization)	(None, 100)	400	FC1[0][0]
act6 (ELU)	(None, 100)	0	bn7[0][0]
FC2 (Dense)	(None, 50)	5050	act6[0][0]
bn8 (BatchNormalization)	(None, 50)	200	FC2[0][0]
act7 (ELU)	(None, 50)	0	bn8[0][0]
FC3 (Dense)	(None, 10)	510	act7[0][0]
output (Dense)	(None, 1)	11	FC3[0][0]

Total params: 253,775
Trainable params: 252,997
Non-trainable params: 778

As we can see, there are a total of 5 convolution layers and 3 FC layers. In addition to these, there are batch normalization layers in between every adjacent layers. ELU layers are chosen to introduce non-linearity.

As for the choice of these layers, first the complexity and number of conv layers and FC layers are kept the same as Nvidia’s paper. Since we are in a simpler situation here, if Nvidia can fit the model to a much larger dataset than us, then the model must be complex enough to fit to out simulated data.

The choice of Batch Normalization layers are used because if can drastically reduce training time with default learning rate. It has added benefit of regulation, as pointed by the original paper on Batch Normalization. Further more, Adam optimizer is used to further increase training speed and achieve smaller loss at the same time.

ELU layers are used because I suspect the “dead” RELU layers might be affecting my results. However I have no very strong evidence that this will be better than RELU at this stage.

2. Attempts to reduce overfitting in the model

As previously mentioned, Batch Normalization added some level of regularization. However, the problem that is more pertinent to this particular problem is the massive amount of zero steering angles in the training set. We want to avoid overfitting to these zeros.

Therefore, one important step is to drop the images with close to zero steering angles with certain probability. This way, the model would tend to fit into those images with non-zero steering angles. The probability is controlled by a function parameter called "pr_threshold" in the function called generate_from_directory(). The smaller “pr_threshold”, the more likely that a zero data point is dropped.

The idea is from this link:

https://chatbotslife.com/using-augmentation-to-mimic-human-driving-496b569760a9?source=user_profile---------6----------&gi=112f78fa9e29

Of course, the same as many machine learning problems, a validation set(10% of the whole training set) is kept aside from the training set as a metric to evaluate the training accuracy. The primary purpose for the validation set is though a monitor for overfitting/underfitting so that I know for how many epochs I should run. However, as many people on the discussion forum has pointed out, the loss on the validation accuracy might not be a very good indicator concerning self-driving performance when run on the simulator. I think that the ultimate test should be the autonomous mode in simulator using the trained model.

3. Model parameter tuning

The model used an adam optimizer, so the learning rate was not tuned manually. Shapes and sizes for the conv layers and FC layers are copied from Nvidia’s for the reason stated in the section “An appropriate model arcthiecture has been employed”. Another model parameter that this model carries is the probability with which data points with zero steering angles are dropped. If the car wiggles too much in the simulator, we can decrease this probability so that the model

learns more about driving straight. If the car drives off the safe zone, we can increase the probability so that the model learns more on how to drive back to the middle of the road.

4. Appropriate training data

I have used the dataset provide by Udacity. I have also generated a bunch of recovery data myself, by purposefully driving the car from the edge of the road to the middle of the road. For details about how I used the training data, see the next section.

Model Architecture and Training Strategy

1. Solution Design Approach

As mentioned in “Model Architecture and Training Strategy”, I adapt the model presented in Nvidia’s paper. As previously mentioned, batch normalization is a way to speed up the training and a mean of regularization.

The major problem I encounter is that when the car doesn’t steer when it should in the simulator, even I train the model with left and right camera (and adjust steering angle accordingly) and I have a relatively small loss on the validation set.

Then I decide to drop some of the data points with zero steering angles (as inspired by other people who are working on this project). The probability that these data points are dropped becomes another hyper-parameter of the model that I can tune. If the car drives off the safe zone, we can increase the probability so that the model learns more on how to drive back to the middle of the road.

After adding this drop-out methods, the car learns to turn appropriately most of the time. But here are a few bad spots. So I generated a bunch of recovery data myself.

After adding the recovery data, the car start to turn appropriately in the simulator.

2. Final Model Architecture

The model structure has been presented in section “An appropriate model arcthiecture has been employed”.

3. Creation of the Training Set & Training Process

I first tried to train the network with data provided by Udacity. Of course the whole data set is splitted into training and validation set randomly. Additionally, left-

camera images are adjusted by adding 0.15 to steering angle accordingly. Right-camera images are adjusted by substracting 0.15 from steering angle.

After the above training (<5 epochs, default learning rate used), there are a few bad corners where the car would drive off the track.

Then I added a few thousands of recovery data. However, only center images are used from these recovery data. After 4 epochs of training, the car can drive within the safe zone of track #1. The final square root loss on the test data is about 0.01.

Future Work

a. Try use regular normalization (x-128/255.0 as the first layer). With the methods that drop out zero angles randomly, does the model output a constant angles? If not, how is the learning speed compared with used batch normalization?

b. Use other image augmentation (brightness, shift, flipping) to see if the model can generates to the 2nd track.

c. Read carefully on the original paper on Batch Normalization