Self-Driving Car Nanodegree

P3 – Behavioral Cloning

Raymond Ngiam
October 27th, 2017

I. Definition

Project Overview

Autonomous steering control is undoubtedly one of the most essential elements of a self-driving car. In the past, predating deep learning, approaches for solving this problem were primarily rule based systems based on human-designated features, such as lane markings, guard rails or other cars. These approaches decouple of the problem into two subsystems, namely feature extraction and steering action determination. They were not successful due to the lack of efficient algorithms to optimize the two subsystems and also the inability to make use of large amount of data due to hardware constraints at the time.

Nowadays, the main stream approach for solving this problem has shifted to end-to-end learning using deep learning. It no longer makes use of human-designated features to explicitly elicit useful features for downstream decision making. Instead, deep learning approaches like convolutional neural networks are used for feature extraction and steering angle prediction in one go. With the efficient optimization algorithms and ability to handle large amount of data, this approach has proven to be more effective in contrast to its predecessors.

Problem Statement

The goal of this project is to create a model that is capable of predicting car steering angles reliably and robustly, based only on camera image inputs. The tasks involved in this project include the following:

1. Use the simulator to collect data of good driving behavior
2. Build a convolution neural network in Keras that predicts steering angles from images
3. Train and validate the model with a training and validation set

4. Test that the model successfully drives around track one without leaving the road.

II. Background and Data Creation

Overview of the simulation environment

The simulator car contains 3 cameras which face towards the road. One camera is positioned at the center of the car, while the other two are placed at the left and the right respectively. The camera images are of the dimension 160 pixel by 320 pixel, and they are all color images, hence each containing 3 color channels. Figure 1 below shows example image of the three cameras.

Figure 1 – Left, center and right image of one of the training samples.

There are 4 control outputs for controlling the car, namely steering, throttle, brake, and speed. Out of the 4 outputs, the steering output is of particular interest for building a model to predict steering angles.

The steering input of the car simulator is ranged from -1 to +1, representing actual steering angle of range -25° to +25°. Negative steering angle represents steering to the left, while positive steering angle represents steering to the right.

For autonomous driving mode in the simulator, the car will be autonomously driven with a constant speed. The model only needs to predict the steering angle given the input feeds from the center camera only.

There are two tracks in the simulator, we are only going to use track one throughout this project.

Creation of the Training Set

To capture good driving behavior, 12 laps of driving on the track in normal direction are captured. During driving the car in the simulator, we try to keep the car to stay in the center of the road as much as possible.

Since the driving track has more left turns than right turns, the data acquired is biased with more samples with left turns. Hence, another 12 laps are captured while driving in reversed direction.

In training mode, the simulator captures the driving data, including camera images (left, center and right cameras), steering, throttle, brake and speed, at a rate of 12 frames per second, into a driving log. For the training data we acquired with 24 laps of driving, there are a total of 27,366 training samples. Table 1 below shows first 5 training samples in the driving log.

	center	left	right	steering	throttle	brake	speed
1	/home/ray/Desktop/data/IMG /center_2017_10_22_2...	/home/ray/Desktop/data/IMG /left_2017_10_22_23_...	/home/ray/Desktop/data/IMG /right_2017_10_22_23...	0.0	0.995159	0	4.828272
2	/home/ray/Desktop/data/IMG /center_2017_10_22_2...	/home/ray/Desktop/data/IMG /left_2017_10_22_23_...	/home/ray/Desktop/data/IMG /right_2017_10_22_23...	0.0	0.995159	0	5.980161
3	/home/ray/Desktop/data/IMG /center_2017_10_22_2...	/home/ray/Desktop/data/IMG /left_2017_10_22_23_...	/home/ray/Desktop/data/IMG /right_2017_10_22_23...	0.0	0.995159	0	6.893659
4	/home/ray/Desktop/data/IMG /center_2017_10_22_2...	/home/ray/Desktop/data/IMG /left_2017_10_22_23_...	/home/ray/Desktop/data/IMG /right_2017_10_22_23...	0.0	0.995159	0	7.800054
5	/home/ray/Desktop/data/IMG /center_2017_10_22_2...	/home/ray/Desktop/data/IMG /left_2017_10_22_23_...	/home/ray/Desktop/data/IMG /right_2017_10_22_23...	0.0	0.995159	0	8.699408

Table 1 – First 5 samples in the driving log.

III. Methodology

Data Preprocessing

To recap, the simulator will only provide the center image as input of our model during autonomous mode. For a given center image, the model is required to predict the desired steering command output. Hence, left and right images play no role during autonomous mode. However, we can still make use of them during training.

The left and right images are essentially equivalent to the center viewpoint of the car from slightly deviated positions, either deviated to the left or to the right. Hence, we can use these images as the center image input by correctly compensating the steering output.

For the left images, the car's viewpoint is closer to the left border of the road, thus we compensate the original steering output by adding +0.5 (i.e. equivalent to steering 12.5° to the right.

For the right images, the car's viewpoint is closer to the right border of the road, thus we compensate the original steering output by adding -0.5 (i.e. equivalent to steering 12.5° to the left.

The assigned steering outputs for the three images in the example above are as shown in the Figure 2 below:

Figure 2 – Assigned steering outputs for the three images.

Another way of augmenting the data is flipping the images horizontally and at the same time negating the steering outputs. Doing this, we will have another set of training data that is from a mirrored version of the existing driving track. Figure 3 below shows flipped version of the above 3 images and their respective steering output:

Figure 3 – Flipped version of the three images and their assigned steering output.

The cameras in the simulator capture 160 pixel by 320 pixel images. However, not all of these pixels contain useful information.

The top portion of the image captures trees and hills and sky, and the bottom portion of the image captures the hood of the car.

The next step of the data preprocessing is to crop each image to focus on only the portion of the image that is useful for predicting a steering angle. The top 50 rows and bottom 20 rows of the images are cropped as shown Figure 4 below.

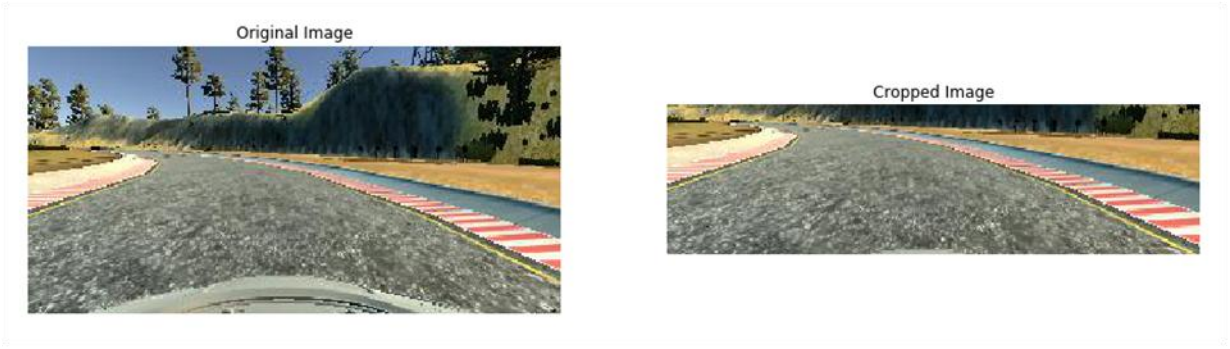

Figure 4 – Original image and cropped image.

Finally, the cropped images are normalized to have means of zero and equal variance by the following transformation:

$$g(x,y) = \frac{g(x,y)}{255.0} - 0.5$$

With these preprocessing steps in place, we would have 6 times more data than our original training samples, i.e. 6 x 27,366 = 164,196 samples.

The images captured in the car simulator are large even after image cropping, i.e. 90 x 320 x 3. Storing 164,196 cropped 24-bit color images would take over 14 GB. Not to mention that normalization can change data types from uint8 to float32, which can increase the size of the data by a factor of 4.

Hence, we will not preprocess the training data all at once. Instead, we will make use of generators which we can pull pieces of the data and process them on the fly only when required.

A generator function is defined to read samples from the driving log, and perform the preprocessing steps in the pipeline just right before image cropping (model.py lines 37-62), thus generating 6 samples from each log instance. The batches from the generator are then fed to Keras Cropping2D layer for cropping and Lambda layer for normalization (model.py lines 83-84).

Model Architecture

The model used is an adaptation of the Nvidia's End to End Learning for Self Driving Cars framework [1], which is trimmed down to have lesser feature depth in the convolutional layers. Full configuration of the model is as depicted below:

```
INPUT: [90x320x1]
LAYER 1:  INPUT    -> CONV-5-1-12  -> ELU     -> BATCHNORM -> MAXPOOL-2-2                [43x158x12]
LAYER 2:  LAYER 1  -> CONV-5-1-24  -> ELU     -> BATCHNORM -> MAXPOOL-2-2                [19x77x24]
LAYER 3:  LAYER 2  -> CONV-3-1-48  -> ELU     -> BATCHNORM                               [17x75x48]
LAYER 4:  LAYER 3  -> CONV-3-1-48  -> ELU     -> BATCHNORM                               [15x73x48]
LAYER 5:  LAYER 4  -> FLATTEN                                                            [52560]
LAYER 6:  LAYER 5  -> FC-100       -> ELU     -> BATCHNORM -> DROPOUT                    [100]
LAYER 7:  LAYER 6  -> FC-50        -> ELU     -> BATCHNORM -> DROPOUT                    [50]
LAYER 8:  LAYER 7  -> FC-10        -> ELU     -> BATCHNORM -> DROPOUT                    [10]
LAYER 9:  LAYER 8  -> FC-1         -> TANH                                               [1]

with:
          CONV-[patch size]-[stride]-[depth]
          MAXPOOL-[patch size]-[stride]
          FC-[output node]
```

The first two layers start with a convolution operation of patch size 5, stride 1 and 'valid' padding, followed by an Exponential Linear Unit (ELU) activation. Batch normalization [2] is then applied to the activations. Batch normalization is added to ensure the activations are well centered at zero with equal variance before input to the next layer. It serves to make the optimization well conditioned and thus improves learning performance.

It is then followed with a max pooling of patch size 2, stride 2 and 'valid' padding (model.py lines 85-90).

Layer 3 and 4 start with a convolution operation of patch size 3, stride 1 and 'valid' padding, followed by an Exponential Linear Unit (ELU) activation and batch normalization (model.py lines 91-94)

The output from layer 4 is flatten at layer 5, and then fed into three fully connected layers, i.e. layer 6, 7 and 8. These fully connected layers have 100, 50, and 10 nodes with ELU activation respectively. Batch normalization and dropout are then applied to these layers (model.py lines 95-104). Dropout is added here for regularization of the model and to reduce overfitting.

Finally, the output of layer 8 is connected to layer 9, a fully connected layer of 1 nodes with 'tanh' activation (model.py lines 105). Since the steering output is in the range of -1 and +1, 'tanh' activation is used as it squashes the logits into the range of -1 and +1.

Model Training

The loss functions used for the training of the model is the mean squared error, defined as follows:

$$loss = \frac{1}{N}\sum_{i=1}^{N}(p_i - t_i)^2$$

where p_i *= i-th prediction of the model (tanh output)*

t_i *= i-th target steering output*

N *= number of training samples*

The model is trained with an Adaptive Moment Estimation (Adam) Optimizer [3] (model.py lines 106-108). Adam optimizer computes adaptive learning rates for each parameter based on exponentially decaying averages of past gradients and past squared gradients.

Since Adam optimizer is used, learning rate of the model is not tuned explicitly. Initial learning rate of the Adam optimizer is set as the default value of 0.001 in Keras.

The full training data of size 164,196 is then split 9:1 into training and validation set, which are of size 147,774 and 16,422 respectively. The model is trained and validated on different data sets to ensure that the model is not overfitting.

Due to large size of the training set, the training of the model is implemented in mini-batches with generator (model.py lines 110-114). The batch size of the generator is 32. Since each sample in the driving log provides 6 different training cases, this is equivalent to 192 training samples per batch. The training of the model will be executed for 5 epochs.

Model Selection

Although we are not tuning the learning rate of the model, we still have one hyperparameter to fine tune, i.e. dropout rate of fully-connected layers.

Grid search is used to explore the effect of dropout rate which is discretized into 4 possible values, namely 0.2, 0.4, 0.6 and 0.8.

Dropout rate with the lowest validation loss will be selected as the final model.

IV. Results

Model Evaluation

The result of the dropout rate grid search optimization is as shown in Table 2.

	Dropout Rate	Training Loss	Validation Loss
1	0.2	0.0888	0.0899
2	0.4	0.0964	0.0813
3	0.6	0.1078	0.0897
4	0.8	0.1436	0.0923

Table 2 – Grid search optimization result for dropout rate of fully-connected layers.

The best model is the one with dropout rate 0.4.

The training and validation loss of best model are as shown in Figure 5.

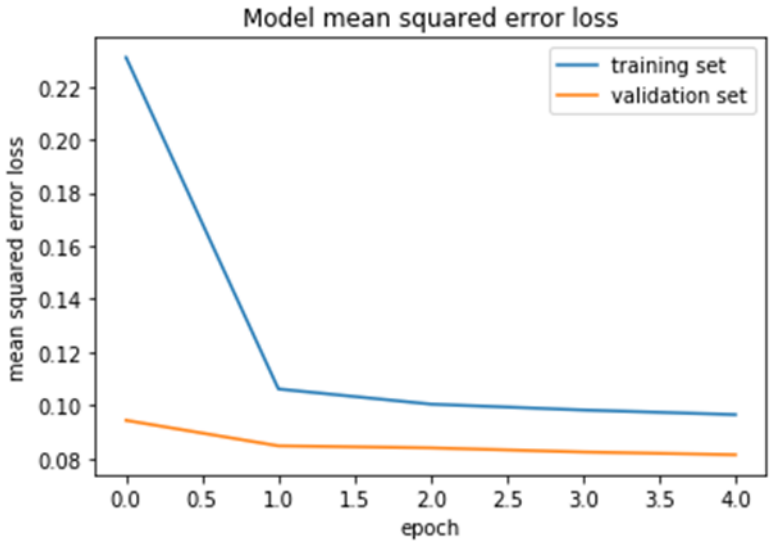

Figure 5 – Training and validation loss of the best model.

The final model has lower mean squared error on the validation set over the training set across all 5 epochs. This implied that the model generalizes well and there is no overfitting.

After training, the best model is tested by running it through the simulator. The car is able to stay on the track consistently (See the video file 'run1.mp4').

V. Conclusion

Reflection

In this project, a convolutional neural network is built with Keras for prediction of steering angles.

One important implementation detail in this project is the usage of generators for preprocessing data on the fly. This is especially crucial when the data is of large size and cannot possibly fit into the available memory of the computer.

References

[1] Bojarski, Mariusz, et al. "End to end learning for self-driving cars." arXiv preprint arXiv:1604.07316 (2016).

[2] Ioffe, Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift." arXiv preprint arXiv:1502.03167 (2015).

[3] Kingma, Ba. "Adam: A method for stochastic optimization." arXiv preprint arXiv:1412.6980 (2014).