Udacity CarND Behavioral Cloning project

Aleksander Czechowski

June 11, 2017

The purpose of this project was to train an algorithm to drive a car in a simulator, based on recordings of human-performed driving in this simulator. My algorithm is based on a convolutional neural network, which matches the input from vehicle's camera with the correct steering angles. The main goal of the project was to make the car drive one lap in autonomous mode, without leaving the road. The objective was achieved, as documented in the video SELF_DRIVING.mp4.

1 Rubric points

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

My project includes the following files:

- clone.py, which is a Python3 script that trains the neural network;
- a directory train_data, which contains data used during training
- model.h5, containing the trained network, that can drive the track;
- drive.py, which is the script that drives the car in autonomous mode;
- writeup/writeup.pdf, which is the project documentation you are reading now;
- a video file SELF_DRIVING.mp4, serving as a proof that the car can drive one loop using the network.

1.2 Submission includes functional code

To drive the car in autonomous mode using my simulator, run the simulator (Linux version) and execute

python drive.py model.h5

1.3 Submission code is usable and readable

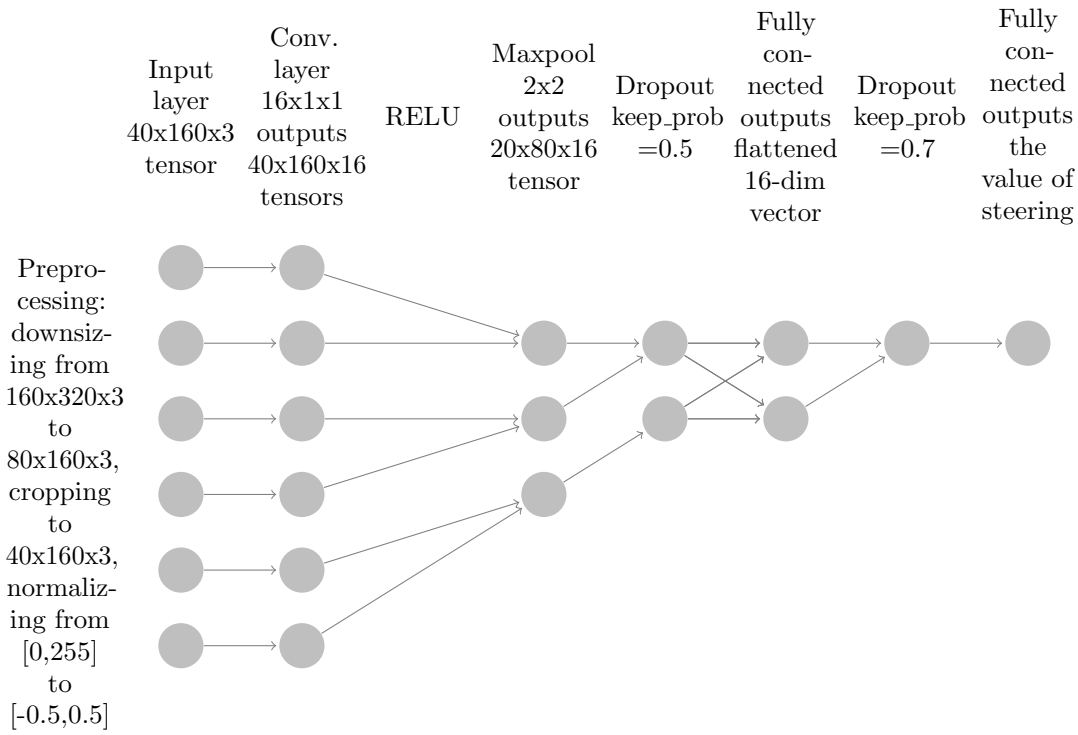
The file clone.py contains the pipeline I used for creating The code is quite short, commented and self-documenting.

The source roughly consists of four short parts: image preprocessing, the data generator configuration, the model architecture, and two lines of code which fit the model and save it.

2 Model Architecture and Training Strategy

2.1 An appropriate model architecture has been employed

Below is a schematic drawing of the network architecture:



2.2 Attempts to reduce overfitting in the model

The model is relatively compact, so it does not overfit much in the first place. However, two dropout layers and a maxpooling layer are in place to reduce overfitting. In addition, during data collection the car drove the track in both directions (additional data).

2.3 Model parameter tuning

The model uses an adam optimizer with self-adapting learning rate.

2.4 Appropriate training data

The training data was collected in order for the model to learn most efficiently. Therefore, rather than driving smoothly, which would result in a lot of 0 driving angles, I tried to perform many slight adjustments of the driving angles, and "bounce" off the edges of the road. This allowed the network to learn recovery maneuvers. For more details, see Subsection 3.3.

3 Model Architecture and Training Strategy

3.1 Solution Design Approach

Initially I tried more advanced architectures (VGG, NVIDIA SDC), but regardless of the preprocessing, filtering (eliminating constant steering data), and regularization steps, it would stop learning early and yield constant steering angles. Perhaps there were too many RELUs in these networks, and they would stop learning (see Karpathy's blog, section 'dying RELU's). I did not find time to debug these networks at a low level. It is certainly worth to have a look at that problem in the future.

I performed three preprocessing steps. First one was resizing the images, effectively reducing their resolution in half, which decreased the memory requirements for the model. Then, I normalized the input data to have mean=0 and sigma=1, which is a standard step and allows for faster learning and easier debugging. Both of these steps had to be repeated in the drive.py script. Finally, I cropped 25 pixels from the top and 15 pixels from the bottom, already at the model architecture level. The reason for cropping was that only the mid-part of the front-camera images contained relevant information on the road curvature.

3.2 Final model architecture

The final architecture was taken from a 'weird lame' model constructed by Cipher, which I found while exploring one of the discussions on Udacity board.

I tried to improve Cipher's network, but it seems to be already highly optimized and performs really well at the task. In the end, I obtained the best results by leaving it intact.

3.3 Creation of the Training Set & Training Process

The training data consists of 9095 images taken from the center camera. It was not necessary to use side cameras to emulate recovery driving. The car was being driven in both clockwise and counterclockwise directions along the track.

I tried to drive the car in a manner that would put it frequently in recovery-type situations:

During evaluation in the simulator, the vehicle still had problems falling into water before and during the bridge. Therefore, additional data had to be collected for driving before, and on the bridge:

Finally, I drove an extra lap with no recovery situations, to smoothen out the driving behavior.

Some more data was collected, but it decreased the performance rather than enhancing it. This data was excluded in driving_log.csv, to avoid having to finetune the model again.

The model was trained for 25 epochs, with batches of 20 images, using keras ImageDataGenerator, adam optimizer and the mean squared error as the loss function (since we are dealing with continuous range of the variable to predict). I did not split into training/validation/test sets, as I wanted to use all the data I collected. This was important for troubleshooting problems on specific parts of the track, I did not want the data I collected for that purpose to go into the validation part. Instead of validation accuracy, the evaluation of the model was performed in the simulator, by successfully completing a lap without driving off the road.

The training was performed on AWS g2.2xlarge server, so the training time was reduced by the use of a GPU.