Behavioral Cloning

by Sergiy Fefilatyev

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.

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.
- `project_report.pdf` summarizing the results.
- `video.mp4` – output of the video collected in autonomous mode for three laps in default (counter-clockwise direction)
- `video_clockwise.mp4` – output of the video collected in autonomous mode for three laps in clock-wise direction.

2. 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
```

3. Submission 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 of modified NVIDIA end-to-end driving network 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 architecture has been employed.

My model mimics NVIDIA end-to-end driving architecture with some modifications. (see NVIDIA's paper).

The model includes five convolutional layers, with ReLu-nonlinearity and four fully-connected layers. I also have a per-processing layer that normalizes and centers the data as well as crops only central portion of the image. The data is normalized in the model using a Keras lambda layer (code line 67). For cropping I used Keras Cropping2D layer (see line 68). Instead of max-pooling sub-sampling (as part of Convlution2D layer) is used.

The modifications to NVIDIA architecture is in introduction of Dropout layers before the first three fully-connected layers. The dropout probability was chosen to be 0.5 . Dropout layers are found on lines 75-78.

The model includes RELU layers to introduce nonlinearity (code line 20) as part of Keras Convolution2D layers (lines 69-73).

2. Attempts to reduce over-fitting in the model.

The model contains dropout layers in order to reduce over-fitting (model.py lines 75-79).

The model was trained and validated on different data sets to ensure that the model was not over-fitting (code line 60-61). The model was tested by running it through the simulator and ensuring that the vehicle could stay on the track.

3. Model parameter tuning.

The model used an Adam optimizer, so the learning rate was not tuned manually (model.py line 84).

4. Appropriate training data

Training data was chosen to keep the vehicle driving on the road. I used a combination of center lane driving, recovering from the left and right sides of the road. Most of the efforts were spend on smooth riding on the center of the track.

For details about how I created the training data, see the next section.

Model Architecture and Training Strategy

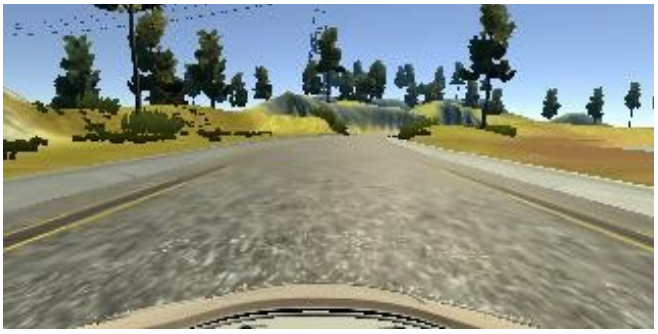
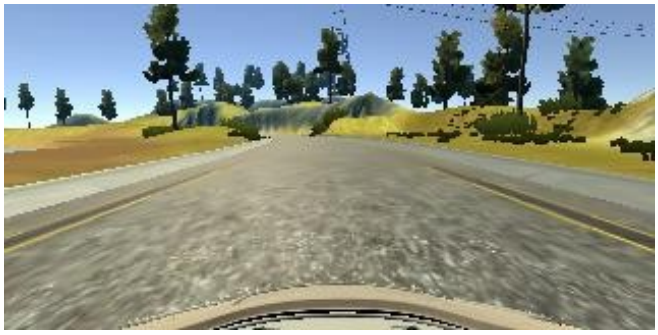
1. Solution Design Approach.

The overall strategy for deriving a model architecture was to obtain the simplest/smallest architecture that would allow smooth central driving on the first track. I focused on a minimal project implementation and, thus, did not consider the second track.

My first step was to use a convolution neural network model similar to a simple LeNet architecture. It was working mostly fine, but I was still getting a pretty big loss and rapid steering commands. I changed it to NVIDIA architecture as a proved design for three-camera system. Three camera-system allows much easier training collection as the the central-lane-based data collection generates corrective steering actions. The corrective steering action for left and right cameras I found suitable was equal to 0.1. The following table shows an example of left,center, and right images taken at the same time from the car.



To augment data further I flip the input image horizontally and change the polarity of the steering correction. For example, here is an image that has then been flipped:

Even with the LeNet architecture I noted much better regularization when using DropOut layer. For NVIDIA architecture was was getting training and validation loss almost equal.

In order to gauge how well the model was working, I split my image and steering angle data into a training and validation set. NVIDIA architecture with Dropout was providing almost equal loss in training and validation splits. Partially it is also explained that dominant part of the data consisted of centrally driven tracks with very smooth consistent steering correction. A bit part in minimizing the loss was selection of corrective steering offset for left and right camera so that they could mimic central camera.

The final step was to run the simulator to see how well the car was driving around track one. In first iterations, there were a few spots where the vehicle fell off the track. But after collecting quality data

and augmenting the data with flips and side cameras the vehicle is able to drive autonomously around the track without leaving the road. I tried the model for the clock-wise direction as well and it drove perfectly.

2. Final Model Architecture

The final model architecture (model.py lines 18-24) consisted of a convolution neural network with the following layers and layer sizes.

Input (160, 320, 3)
Lambda ((x / 255.0) - 0.5)
Cropping2D ((70,25), (0,0))
Convolution2D (24,5,5)
Subsampling (2,2)
ReLu
Convolution2D (36,5,5)
Subsampling (2,2)
ReLu
Convolution2D (48,5,5)
Subsampling (2,2)
ReLu
Convolution2D (64,3,3)
ReLu
Convolution2D (64,3,3)
ReLu
Dropout(0.5)
Dense(100)
Dropout(0.5)
Dense(50)
Dropout(0.5)
Dense(10)
Dense(1)

3. Creation of the Training Set & Training Process.

In order to make model more generalizable I took the step to augment the data. First of all, I collected the reverse track by driving a car in clock-wise direction. I put a lot of effort on making sure all my drives are centralized with very smooth steering. It turned out most of the correct training comes from this smooth steering, keeping the car in the center and using images from left and right cameras with corrective measurement.

I used a generator for providing batches of data. My PC allows keeping the dataset entirely in memory, but the solution with a generator (lines 21-52) is more elegant and allows scaling the application to much larger volumes of data. I used Keras `fit_generator()` method for providing the data with the generator. My batch sizes were equal 192 (where 32 is the number of original central images and the rest would come from side cameras and flips).

The total number of images I used for training was over 80K of which only 24K were provided by Udacity. The dataset had almost equal number of frames collected by driving clock and counter-clock wise. I finally randomly shuffled the data set and put 20% of the data into a validation set.

One important aspect of the dataset is that my own data was collected in the lowest quality. After training the network and testing it in autonomous mode I noticed it was working perfectly when was tested in the same quality of images. But if I were going to a better quality imagery the cam could veer-off.

NOTE TO REVIEWER: please test my network in the lowest resolution and lowers quality video as that I the settings I trained it with.