Behavioral Cloning

Write-up

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
- writeup_report.md or writeup_report.pdf summarizing the results

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 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 consists of a convolution neural network (model.py lines 54-70) inspired by NVIDIA's whitepaper from April 2016: "End to End Learning for Self-Driving Cars. The model has nine layers, including a normalization (Lambda) layer (code line 54), five convolutional layers (three with 5x5 filters and a stride of 2x2, and two non-strided layers with 3x3 filters) and three fully connected layers (100, 50 and 10). The model includes RELU activations to introduce nonlinearity.

2. Attempts to reduce overfitting in the model

The model contains dropout layers in order to reduce overfitting (model.py lines 63, 65, 67 and 69).

The model was trained and validated on different datasets to ensure that the model was not overfitting (code line 74). 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 73).

4. Appropriate training data

Training data was chosen to keep the vehicle driving on the road. I gathered data from the simulator in a number of ways:

- Track 1 center driving counterclockwise: 4 laps
- Track 1 center driving clockwise: 4 laps
- Track 1 recovery driving counterclockwise; record only returning to center

- Track 1 curves counterclockwise; record only smooth driving on sharp curves
- Challenge Track center driving counterclockwise: 1 lap
- Challenge Track center driving counterclockwise: 1 lap
- Udacity provided dataset

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 work my way up in terms of complexity to determine what level of sophistication (in terms of models) would be required to accomplish this task.

I started by building one of the simplest sequential keras models imaginable by flattening the input and passing it to a single dense layer. In order to gauge how well the model was working, I split my image and steering angle data into a training and validation set: 80/20%. I implemented the Adam optimizer and mean squared error for the loss function and kept all of these latter settings for future iterations.

The car in the simulator drove off the road very quickly so I knew pretty quickly that a much more powerful model would be needed to complete this task.

I went on to build the following models on my way to finding a successful CNN:

- normalized version of first model mentioned above
- LeNet architecture with same 80/20% split and Adam and MSE for model.compile
- LeNet + cropping
- LeNet + cropping + augmented data (images flipped horizontally)
- LeNet + cropping + augmented data + using all 3 cameras with 0.2 correction
- NVIDIA model as stated in aforementioned whitepaper

Ultimately, I utilized the NVIDIA model with augmented data (horizontally flipped), cropping (omitting 70 and 25 lines from the top and bottom, respectively) and Dropout, as I found that my model was overfitting for each model I tried. I suspect that my data set was far inferior to whatever NVIDIA is using so Dropout seems logical in my case. Validation error didn't change much over the course of 10 epochs but the 10th was the lowest at 0.009.

The final step was to run the simulator to see how well the car was driving around track one for each of these models. LeNet with at least cropping and augmented data seemed to do quite well but couldn't quite make its way around the track, even after many may attempts. In the end, the NVIDIA architecture seemed much better suited for this task and it's what was required to get the car all the way around the track, given the data set I created.

2. Final Model Architecture

The final model architecture (model.py lines 54-70) consisted of a convolution neural network with the following layers and layer sizes:

- Input: 160x320x3 images
- Lambda normalization to a range of -0.5 to 0.5
- Crop to omit 70 from top and 25 from bottom of images
- 5x5 convolutional layer (24), stride of 2x2 with RELU activation
- 5x5 convolutional layer (36), stride of 2x2 with RELU activation
- 5x5 convolutional layer (48), stride of 2x2 with RELU activation
- 3x3 convolutional layer (64), non-strided with RELU activation
- 3x3 convolutional layer (64), non-strided with RELU activation
- Flatten
- Dropout (0.5)
- Fully connected layer (100)
- Dropout (0.5)
- Fully connected layer (50)
- Dropout (0.5)
- Fully connected layer (10)
- Adam optimizer
- Mean Squared Error for loss function

3. Creation of the Training Set & Training Process

To capture good driving behavior, I first recorded four laps on Track 1 using center lane driving. Here is an example image of center lane driving:

I then recorded the vehicle recovering from the left side and right sides of the road back to center so that the vehicle would learn to bring itself back to the middle of the track whenever it strays from center. These images show what a recovery looks like, from the left and right sides of the track:

- Track 1 center driving counterclockwise: 4 laps (4439x3 images)
- Track 1 center driving clockwise: 4 laps (4711x3 images)
- Track 1 recovery driving counterclockwise; record only returning to center (8814x3)
- Track 1 curves counterclockwise; record only smooth driving on sharp curves (1441x3)
- Challenge Track center driving counterclockwise (heavy braking): 1 lap (5148x3)
- Challenge Track center driving counterclockwise (heavy braking): 1 lap (5776x3)
- Udacity provided dataset

I created eight different datasets using different variations of the above, which was extremely time consuming. Ultimately, I ran my model on a combination of half of the Track 1 center driving counterclockwise (2 laps) + half of the Track 1 center driving clockwise (2 laps) + everything else listed above, all combined into one batch.

The dataset I used had (15002x3) number of data points. I then preprocessed this data by normalizing and doubled the dataset by using flipped versions, to give a total of 30004 images, or 90012 images if all three cameras were utilized. I had compatibility issues with mismatched versions of Keras when trying to use all three cameras with a correction factor of 0.2.

I finally randomly shuffled the data set and put 20% of the data into a validation set. I used this training data for training the model. I used 10 epochs and the adam optimizer so that manually training the learning rate wasn't necessary.