Behavioral Cloning Project

9th September 2017

The goals 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 the training and validation sets.
- Test that the model successfully drives around track one without leaving the road.
- Summarize the results with a written report.

I will consider the rubric points individually and describe how I addressed each point in my implementation.

1 Required Files Submitted & Code Quality

The submission includes all required files that can be used to run the simulator in the autonomous mode:

- `drive.py` for driving the car in autonomous mode;
- `model.h5` containing a trained convolution neural network;
- `model.py` containing the script to create and train the model;
- `video.mp4` containing a video with the car driving in autonomous mode;
- `writeup.pdf` summarizing the results.

2 Code Quality

2.1 Functional Code

Using the Udacity provided simulator and the `drive.py` file, the car can be driven autonomously around the track by executing

```
python drive.py model.h5
```

with the speed up to 15 mph.

2.2 Usable and readable code

The `model.py` file contains the code for preparing data, training and saving the convolution neural network. The file shows the pipeline I used for training and validating the model. In addition, it contains self explanatory comments.

3 Model Architecture and Training Strategy

3.1 An appropriate model architecture has been employed

The data in my model is normalized using a Keras lambda layer (`model.py` line 115). My model consists of a convolution neural network with 5×5 and 3×3 filter sizes and depths between 24 and 64 (`model.py` lines 117-121). The model includes *RELU* activation functions to introduce nonlinearity in each convolution layer (`model.py` lines 117-121).

3.2 Attempts to reduce over-fitting in the model

The model is trained and validated on different data sets to ensure that the model was not over-fitting (`model.py` line 39). The training data set is represented by 80% of the input data, whereas 20% of the input data is associated with the validation data set. The model is tested by running it through the simulator and ensuring that the vehicle can stay on the track.

To reduce over-fitting, the model includes dropout layers between fully connected layers (`model.py` lines 124-129).

3.3 Model parameter tuning

The model uses an Adam optimizer (`model.py` line 132), therefore the learning rate is tuned automatically.

3.4 Appropriate training data

The training data is chosen to keep the vehicle driving on the road. To achieve this behavior, I use a combination of center lane driving, left and right lane driving (see the next section for details of the training data creation).

4 Architecture and Training Documentation

4.1 Solution Design Approach

To derive the correct model architecture, a number of steps was taken. I started with a neural network model with low complexity similar to *LeNeT-5*. This model could already be trained in the way that the car was driving a part of the track autonomously. Of course, since the mean squared error was too high on the validation set, the steering angles were not always predicted correctly, and the car fell off the track.

My next step was to use data preprocessing to get better data input. In this step the data was normalized and cropped inside the neural network. This in turn made the mean squared error lower. I also used all three cameras provided by the simulator with adjusted steering angles to avoid the car driving off the track.

To improve the steering angle prediction, I used the model published by the autonomous vehicle team from *Nvidia*. This model predicts the steering angles good enough, such that the vehicle is able to drive autonomously around the track without leaving the road.

I generalized training data by augmenting the collected images. Using a simplest augmentation technique, I added the flipped images with an inverted steering angle sign to the data set during the training process. Unfortunately, the training data set became too big to fit into the memory. Therefore, I included a generator function that split the data on batches. The size of batches equals to 32 such that the training data fits into the memory of my computer. In addition, to reduce the execution time, I installed the tensorflow library with GPU support and performed all operations on my Nvidia GPU. When fine-tuning the model, I configured an AWS instance with multiple GPUs and trained the model there.

At the end of the process, the vehicle was able to drive autonomously around the track without leaving the road in the both forward and backward direction (see `video.mp4`).

4.2 Final Model Architecture

The final model architecture is an extended version of the one presented by the *Nvidia* autonomous vehicle team. In addition to the features of the *Nvidia* model, my model includes the dropout layers between each fully connected layer to avoid over-fitting.

The first layer normalizes the input images mapping the RGB input from 0 to 255 to -0.5 to 0.5 (`model.py` line 115). The second layer crops the input images cropping top 67 pixels and bottom 25 pixels (`model.py` line 116). The image size changes from (160, 320, 3) to (68, 320, 3). The next five convolutional layers use the *RELU* activation function. The first 3 convolutional layers apply 5×5 filters (`model.py` lines 117-119), whereas the last two apply 3×3 (`model.py` lines 120-121).

After flattering the model with the next layer (`model.py` line 123), the set of fully connected layers are added with a dropout with probability 0.2 in between (`model.py` lines 124-130).

The visualization of the architecture is constructed using *keras.utils.visualize_util*:

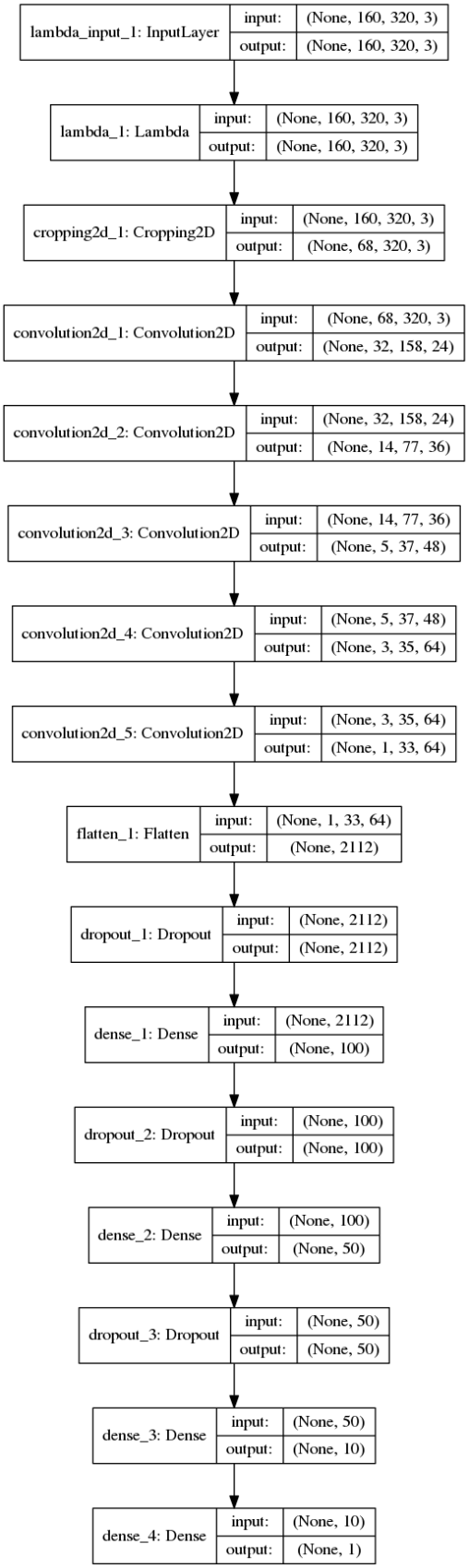

Figure 1: Model architecture

4.3 Creation of the Training Set & Training Process

To capture good driving behavior, I first recorded two laps on track one using center lane driving. Here is an example image of center lane driving:

(a) Left camera (b) Center camera (c) Right camera

Figure 2: Center lane driving

The training data with center lane driving only is of course not sufficient. There is always a situation when the neural network has to predict how to recover from the left or right sides of the road back to the center. I model this recovering by recording the vehicle driving on the left and right sides of the road applying the correction steering angles during the model training. The left, center and right cameras are taken into account. These images show what a left lane driving looks like:

(a) Left camera (b) Center camera (c) Right camera

Figure 3: Left lane driving

The right lane driving looks as follows:

(a) Left camera (b) Center camera (c) Right camera

Figure 4: Right lane driving

Therefore, the correction steering angles in left, center and right lane driving are as follows:

	Left Camera	Center Camera	Right Camera
Left Lane Driving	+0.35	+0.25	+0.15
Center Lane Driving	+0.1	0	−0.1
Right Lane Driving	−0.15	−0.25	−0.35

Therefore, the absolute value steering angle correction, $|\alpha|$, is proportional to the distance from the vehicle center to the center of the track, l:

$$\frac{d|\alpha|}{dl} > 0.$$

This means that if the vehicle drives on the left side of the road, the algorithm sets the high absolute value of steering angle with the positive sign. If the vehicle drives on the right side of the road, the algorithm also sets the high absolute value of the steering angle but with the negative sign. In both cases the corrected steering angles bring the vehicle back to the lane center.

To augment the data set, I also flipped images and angles because this gives more data points.

After the collection process, I had 6267 data points of the center lane driving, 2904 – of the left lane driving and 3168 – of the right lane driving. I then preprocessed the images by normalizing it mapping values from 0 to 255 to the interval between -0.5 and 0.5.

I finally randomly shuffled the data set and put 20% of the data into a validation set. Other 80% I used as training data for training the model. The validation set helped determine if the model was over or under fitting. The ideal number of epochs was 5 that provided already a low mean squared error. I used an Adam optimizer so that manually training the learning rate wasn't necessary.

5 Simulation

No tire leaves the drivable portion of the track surface. The car does not pop up onto ledges or roll over any surfaces that would otherwise be considered unsafe (if humans were in the vehicle).