Project: Behavioral Cloning

Akhil Waghmare

July 2017

Model Architecture and Training Strategy

1. *Model architecture*

   My model consists of a convolutional neural network with 5x5 filter sizes and depths between 24 and 48 followed by convolutions with 3x3 filter sizes and depth of 64 (model.py lines 65-69). The output is then flattened and fed into a feed-forward neural net (code lines 71-78).

   The model includes RELU layers to introduce nonlinearity (code lines 65-69), the data is normalized in the model using a Keras lambda layer (code line 62), and the input images are cropped (code line 63).

2. *Attempts to reduce overfitting*

   The model contains a dropout layer in order to reduce overfitting (model.py line 74).

   The model was trained and validated on different data sets to ensure that the model was not overfitting (code line 82). The model was tested by running it through the simulator and ensuring that the vehicle could stay on the track.

3. *Model paramter tuning*

   The model used an adam optimizer, so the learning rate was not tuned manually (model.py line 81).

4. *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, and driving both clockwise and counterclockwise around the track.

Architecture and Training Documentation

1. *Solution design approach*

   The overall strategy for deriving a model architecture was to start with an existing model which was known to be pretty good, train the original model with my data, make adjustments based on accuracy and overfitting, and then testing the model on the track.

   My first step was to use a convolution neural network model similar to the NVIDIA CNN (`https://devblogs.nvidia.com/parallelforall/deep-learning-self-driving-cars/`). I thought this model might be appropriate because it was previously used by NVIDIA to train a real self-driving car.

   In order to gauge how well the model was working, I split my image and steering angle data into a training and validation set. I found that my first model had a low mean squared error on the training set but a high mean squared error on the validation set. This implied that the model was overfitting.

   To combat the overfitting, I modified the model so that it included a dropout layer.

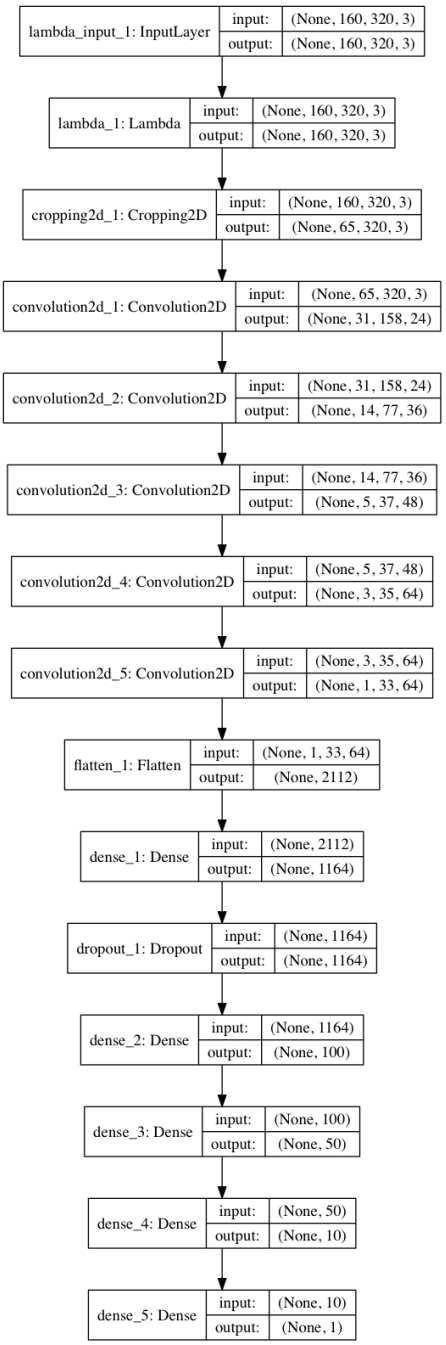
   The final step was to run the simulator to see how well the car was driving around track one. There were a few spots where the vehicle fell off the track. To improve the driving behavior in these cases, I

collected more training data around those specific spots, as well as augmented the dataset to further increase the number of training examples being fed in (see next section for specific augmentation performed).

At the end of the process, the vehicle is able to drive autonomously around the track without leaving the road.

2. *Final model architecture*

The final model architecture (model.py lines 82-100) consisted of a convolution neural network with the following layers and layer sizes:

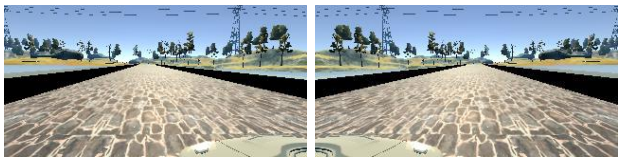


3. *Creation of the training set & training process*

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:

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 recover from veering too far left or right. These images show what a recovery looks like starting from the right side of the lane and working its way back to the center:

To augment the data sat, I also flipped images and angles thinking that this would increase the number of examples I fed into the network training. For example, here is an image that has then been flipped:



After the collection process, I had 6640 number of data points. After augmentation, this number multiplied by 6 to become 39840. I then preprocessed this data by normalizing the pixel values and cropping the images so that only the road portion remained.

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. The validation set helped determine if the model was over or under fitting. The ideal number of epochs was 5 as evidenced by the validation accuracy staying flat or starting to increase. I used an adam optimizer so that manually training the learning rate wasn't necessary.