Behavioral Cloning - Project Writeup

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. *I made no changes to this file.*
- **`model.h5`** : containing a trained convolutional neural network in Keras format.
- **`writeup_report.md`** : you are reading it or the pdf generated from it.

I *did not* include a `video.mp4` file because it goes against my religion to put heavy binary files inside github reports. I think this is an abuse. **The video of a whole lap driven by the trained network is available for the reader's enjoyment here:** https://youtu.be/pNxtBFdwN7k

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

File `model.py` contains the code for training and saving the convolution neural network and also the Python generator (function generator to do some data augmentation and feed images into it.) Function `search_correction_params` shows how I tried with different combinations of parameters `side_img_correct_mode` and `side_img_correct_param` which basically control how the angle is corrected for left/right images produced by `generator` during training.

Model Architecture and Training Strategy

1. An appropriate model architecture has been employed

My model uses essentially the same architecture shown in section 15, titled "Even More Powerful Network", of the project chapter.

The model is constructed in function `build_model` (`model.py lines` 152-176). It includes 3 convolutional layers with 5x5 kernels followed by 2 convolutional layers with 3x3 kernels, followed by 3 fully connected layers. The main differences with respect to the model shown in the video are:

- I included several dropout layers between the Conv layers, with dropout parameter `dropout_cs` .
- There are also a couple dropout layers between fully connected layer.
- The final layer has output of dimension 1, as we only want to predict the steering angle. I used "tanh" as the activation function of this layer to guarantee that the output was in the interval [-1,1]
- The model includes RELU layers to introduce nonlinearity (code line 20), and the data is normalized in the model using a Keras lambda layer (code line 18).

2. Attempts to reduce overfitting in the model

As mentioned above, the model contains dropout layers in order to reduce overfitting (function `build_model`). The model was tested by running it through the simulator and ensuring that the vehicle could stay on the track for a whole lap.

3. Model parameter tuning

The training procedure used an Adam optimizer. However, I did some tries with the default value 0.001 for the learning rate, used by keras but later lowered it to 0.0005, as it seem to produced a better validation loss. The learning rate is set as part of the `cfg` parameter dict (`model.py` line 55)

4. Appropriate training data

Training data was chosen to keep the vehicle driving on the road. I mostly used center lane driving, ocasionally recovering from the left and right sides of the road.

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 take advantage of the experience gained by NVIDIA engineers about how to design a neural network for driving a car and just copying their architecture.

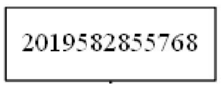
From the beginning I knew that overfitting could be an issue. So included significant dropout on my layers, focusing on the ones with the most parameters. As a comparison between the training and validation losses for up to 20 epochs showed, this proved rather effective in eliminating overfitting almost completely.

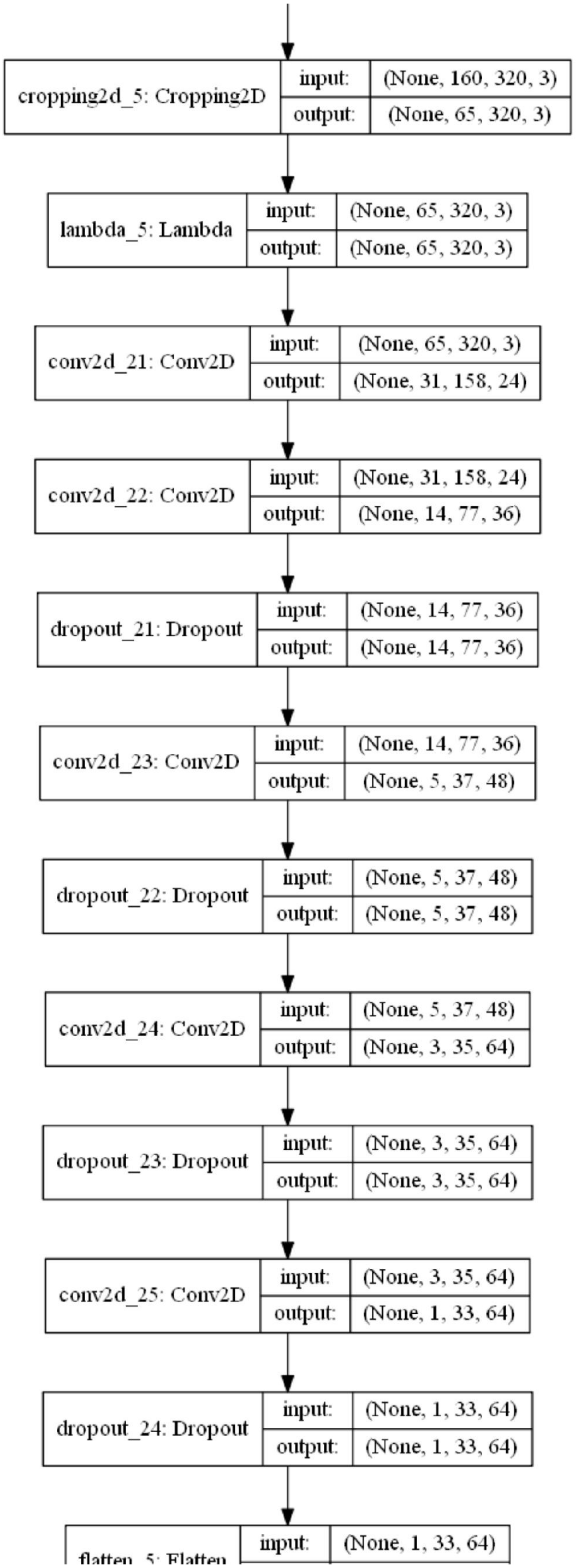
The final step was to run the simulator to see how well the car was driving around track one. There were a few spots where there was visible swerving, indicating some instability in the steering angle produced by the model. There is also a spot in which the car gets close to one of the side banks of the bridge, which looks unsafe. This is due to me making a similart mistake when producing the training data. However, the car manages to avoid the collision and, in fact, it never falls off the track.

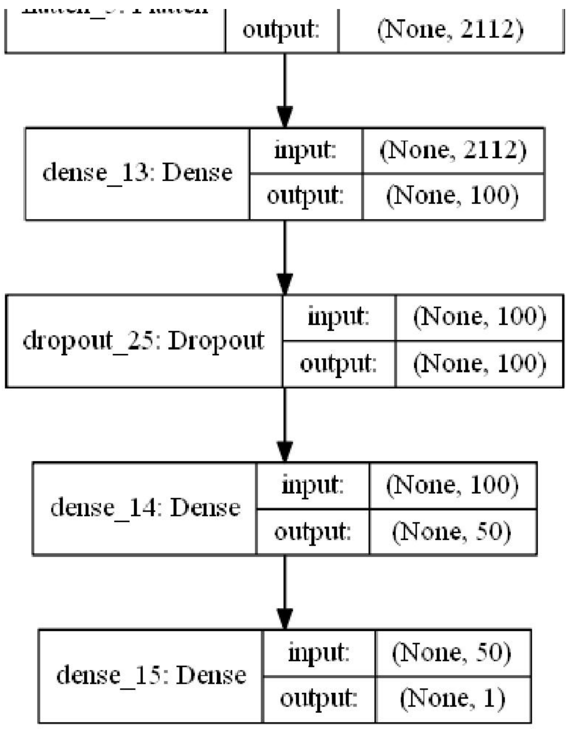
At the end of the process, the vehicle is able to drive autonomously around the track without leaving the road.

2. Final Model Architecture

The final model architecture (`model.py` lines 154-178) consisted of a convolutional neural network with the following layers indicated in the following figure:

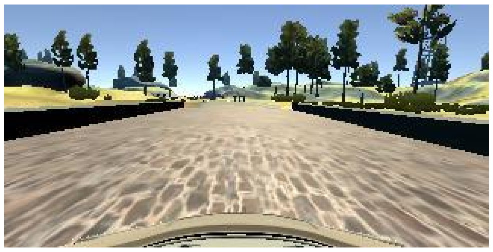



cropping2d_5: Cropping2D
input: (None, 160, 320, 3)
output: (None, 65, 320, 3)
lambda_5: Lambda
input: (None, 65, 320, 3)
output: (None, 65, 320, 3)
conv2d_21: Conv2D
input: (None, 65, 320, 3)
output: (None, 31, 158, 24)
conv2d_22: Conv2D
input: (None, 31, 158, 24)
output: (None, 14, 77, 36)
dropout_21: Dropout
input: (None, 14, 77, 36)
output: (None, 14, 77, 36)
conv2d_23: Conv2D
input: (None, 14, 77, 36)
output: (None, 5, 37, 48)
dropout_22: Dropout
input: (None, 5, 37, 48)
output: (None, 5, 37, 48)
conv2d_24: Conv2D
input: (None, 5, 37, 48)
output: (None, 3, 35, 64)
dropout_23: Dropout
input: (None, 3, 35, 64)
output: (None, 3, 35, 64)
conv2d_25: Conv2D
input: (None, 3, 35, 64)
output: (None, 1, 33, 64)
dropout_24: Dropout
input: (None, 1, 33, 64)
output: (None, 1, 33, 64)
flatten_5: Flatten
input: (None, 1, 33, 64)

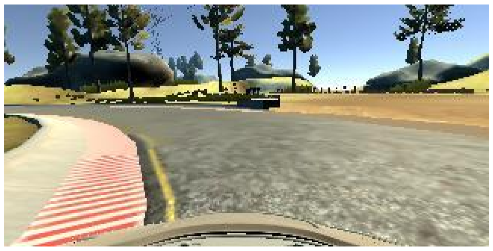
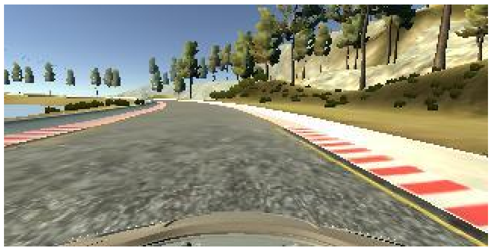


3. Creation of the Training Set & Training Process

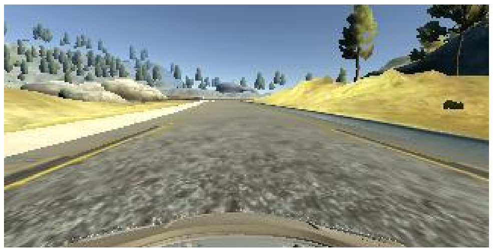
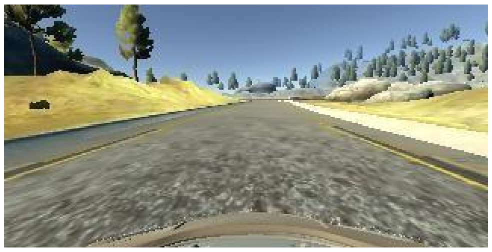
To capture good driving behavior, I first recorded two laps on track one using center lane driving. Here is an example image of center lane driving:



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 go back to the center after having deviated. These images show what a recovery looks like starting from the left and right respectively.

To augment the data sat, I also flipped images and angles thinking that this would ... For example, here is an image that has then been flipped:

After the collection process, I had around 20000 data points including sideviews from left and right cameras.

Instrumental to my solution was the approach taken for producing target angles for images from the left and right cameras. I tried two different parameterized approaches, called `side_img_correct_mode` s in the code.

- Perform 'additive' correction. That is, change the angle as $\alpha -> \alpha \pm \gamma$, where γ is a constant such as 0.1 or 0.5 and the sign is chosen as $+$ if the image is from the left camera and the as $-$ if the images is from the right.
- Perform 'multiplicative' correction. That is, change the angle as $\alpha -> \alpha(1 \pm \gamma)$, where, again, γ is a constant such as 0.1 or 0.5 and the sign is chosen as to make the angle more negative or more positive depending on its current sign and wether the camera is the left or right hand side camera. The details of the logic here are in the code inside the generator function where the nested functions `correct_left` and `correct_right` are defined (lines 240 through 265)

Side camera images where also mirrored and the angle correspondingly flipped after the correction applied.

After training the model with a few choices of the γ parameters (called `side_img_correct_param` in the code) for each correction mode and a seeing most of the resulting models crash and -- almost -- burn, or get stuck against virtual tires, a couple of models actually managed to make a full lap. The one I am submitting corresponds to the 'multiplicative' correction with $\gamma = 0.25$

The generator does shuffling of the whole data set (line 268).

In function train_model I split the data set into 70% train, 24% validation and 6% test, although, in the end, I am not using the test set for anything...

I ran training always for 20 epochs as it seemed more than enough for both training and validation error curves to flatten at around 0.03 MSE, which I think was pretty decent...