Behavioral Cloning

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 summarizing the results
- Output.mp4 showing a lap of autonomous driving using the model

2. Submission includes functional code

Using the Udacity provided training data, my model.py file trains the model and saves it into model.h5 by executing

```
python model.py
```

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

As mentioned above, the model.py file contains the code for training and saving the convolutional neural network model. 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

After testing with several smaller, less complex models, I implemented the NVidia model that was described in the

course lectures (model.py lines 101-125)

This model performs normalization using a Keras lambda layer, followed by cropping and then five convolutional layers and finally five fully connected layers.

2. Attempts to reduce overfitting in the model

Training with a default ten epochs resulted in some minimal overfitting. Reducing the number of epochs to 5 (model.py line 130) corrected that condition. Eventually, the number of epochs was changed to 7 due to slight underfitting after some other changes in the model related to correction factors for right and left cameras.

3. Model parameter tuning

The model used an adam optimizer, so the learning rate was not tuned manually (model.py line 128).

4. Appropriate training data

After spending a significant amount of (real) time and GPU time attempting to create a dataset through manual driving, I reverted to use of the default dataset provided with the project. This allowed me to concentrate my time on model development and tuning to achieve a successful project.

Model Architecture and Training Strategy

1. Solution Design Approach

The overall strategy for deriving a model architecture was starting with a very simple model to test the data parsing and preprocessing and overall understanding of the project requirements and simulator use.

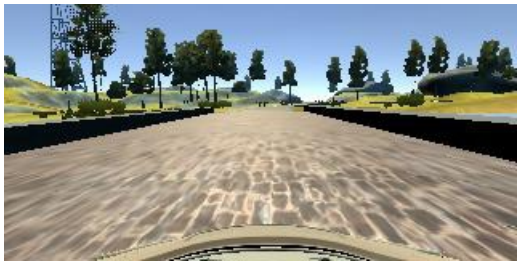
My first step was to use a model with just normalization and two fully connected layers.

In order to gauge how well the model was working, I split my image and steering measurements into a training (80%) and validation set (20%). The loss started very high but decreased quickly (although losses remained high even after 10 epochs). The resultant model simply drove the car in circles.

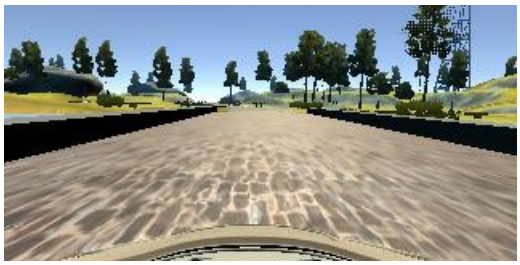
I experimented with the more complex LeNet network and augmented the data by flipping the images and measurements. This effectively doubled the dataset size.

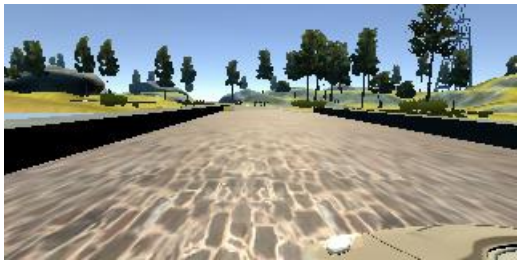
By taking note of the trees in the two images below, the flipping operation is apparent.

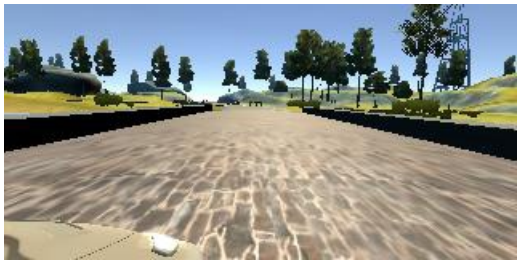
Not Flipped 

Flipped 

Model performance was only slightly improved and the car crashed before the first curve. Next, I cropped the image and augmented the data further by incorporating data from the left and right cameras and adjusting the steering measurements accordingly. This provided three times the data. 38572 images and measurements for training and 9644 for validation.

Center 

Left 

Right 

After training, this model showed improvement and drove the car past the bridge before crashing. Finally, I implemented the much more complex NVidia model (see below). After training, the car drove further crashing just past the bridge. I lowered the left and right camera steering measurement correction slightly to soften the right and left excursions. This model performed well but drove outside of the lane on the left once.

The correction factors for left and right were separated and modified slightly to provide a slight steering bias to the right. Also, in order to ensure underfitting wasn't occurring, the epochs were increased to 7. This produced a loss of less than 1% on the training data.

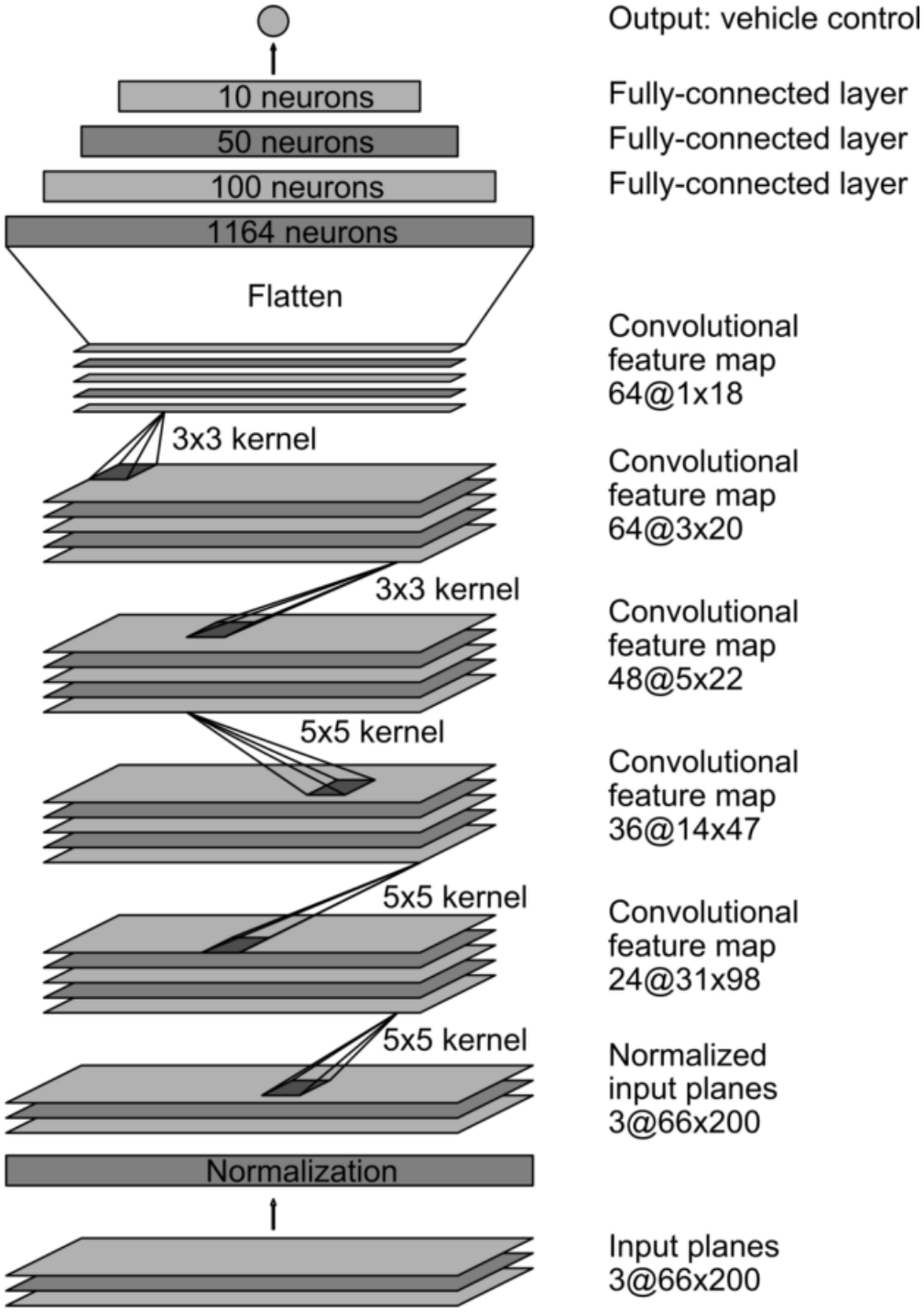
The model performance was still inadequate so both the right and left corrections were reduced and set to the same value. A further reduction in the factors was taken and a slight bias in the left factor was added. This caused the car to drive slightly right of center most of the time.

After these changes, the vehicle is able to drive autonomously around the track for a full lap without leaving the road.

The video file Output.mp4 shows one lap of autonomous driving using the trained model.

2. Final Model Architecture

The final model architecture (model.py lines 101-125) is described above. Here is a visualization of the architecture



3. Creation of the Training Set & Training Process

I utilized the provided training dataset but augmented the data by flipping the images and steering measurements. Additional augmentation was provided by using the left and right camera images saved in the data set. Steering measurements for the left camera were adjusted by adding a correction factor and for the right camera by subtracting the same correction factor. This factor provided a convenient hyperparameter for tuning. I started with 0.2 and after separating the factors tried 0.175 for the left correction and 0.185 for the right correction. These values produced over correction and after a little experimentation, the final factors for right and left that providing acceptable performance were 0.1 and 0.15, respectively.

I randomly shuffled the data set and put Y% 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 7 as evidenced by a loss that continued to decrease and indicated no overfitting since the loss didn't increase in later epochs. I used an adam optimizer so that manually training the

learning rate wasn't necessary.