Behavioral Cloning Project

Brent Wylie, April 12th, 2017

The goals / steps of this project were 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

In this writeup 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.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. I did not use a python generator as my system handled the data in memory just fine.

Model Architecture and Training Strategy

1. An appropriate model architecture has been employed

My model consists of a convolution neural network with 5x5 and 3x3 filter sizes and depths between 24 and 64. It is a reproduction of the CNN described in the Nvidia paper shown in the lesson.

The model begins with a cropping layer to reduce input sizes to main region of interest, and the data is normalized using a lambda layer. The model then uses three 5x5 convolution layers that subsample 2x2 (with depths 24, 36, 48 respectively), followed by two 3x3 convolution layers that subsample 1x1 (each 64 deep). All convolution layers introduce nonlinearity by using ReLU activation functions. Each convolution layer also uses 'valid' padding.

Following the convolution layer, the model has a flatten layer preceding the fully connected layers.

The model has a dropout layer for training in between the first dense layer with 100 output size, and the second dense layer with 50 output size. The dropout was set at 35% for training. The network ends with two dense connected layers of 10 and 1. None of the dense layers specify an activation, which according to Keras documentation means no activation was employed on these layers.

2. Attempts to reduce overfitting in the model

This model contains dropout layers in order to reduce overfitting (model.py lines 61). Only one set at 35% was needed to prevent overfitting. The model used an adam optimizer but set the learning rate to half of default, so 0.0005.

The model was trained and validated on different data sets to ensure that the model was not overfitting using the validation_split parameter of model.fit(). The model was tested by running it through the simulator and ensuring that the vehicle could stay on the track.

Some overfitting was initially observed once additional training data was added, and dropout + reduced LR + fewer epochs helped avoid overfitting.

3. Model parameter tuning

The model used an adam optimizer, but I found a reduction in learning rate by half (to 0.0005) was helpful to reduce overfitting.

4. Appropriate training data

Training data was chosen to keep the vehicle driving on the road. I used the sample data provided and created my own data driving centered in the lanes for all of the sharp turns. I also created data for one centered drive the opposite direction around the track to help generalize. I used the ‘three-camera’ approach described in the lesson to create recovery data from every existing piece of training data.

Architecture and Training Documentation

1. Solution Design Approach

My overall strategy for creating a working model was to follow the approaches demonstrated in the lesson. After getting the environment to work with a simple 1 layer fully connected network, I adopted the Nvidia-inspired model from the lesson and began training solely on the provided data (using just the single camera).

I ran this model in the simulator and I found it worked some of the time, but had trouble recovering. I chose this architecture because I knew it had been used successfully by others in the past.

Initially I evaluated the model purely by performance. I observed that the model did not have enough of an idea of what to do when it approached the edge of the road, so I moved towards using the three camera training approach to create recovery data for every existing piece of data. This essentially tripled the training and validation data size and I started to suspect some overfitting as the car seemed to react strangely around previously successful areas.

After adding more data for turns and a generalized opposite-direction run, the performance was good enough to start fine tuning the training, so I began plotting the Mean squared error (MSE) on the training sets vs the MSE on the validation sets. After a bit of playing with different network activation functions and initial weights, I confirmed that the model was overfitting the data.

I introduced dropout and reduced the learning rate and finally saw the car able to begin to successfully recover from turns.

2. Final Model Architecture

Here is a visualization of the final network (from output to input), the graphic is taken from the Nvidia paper in which the network was described and inherited from (in the Udacity lesson):

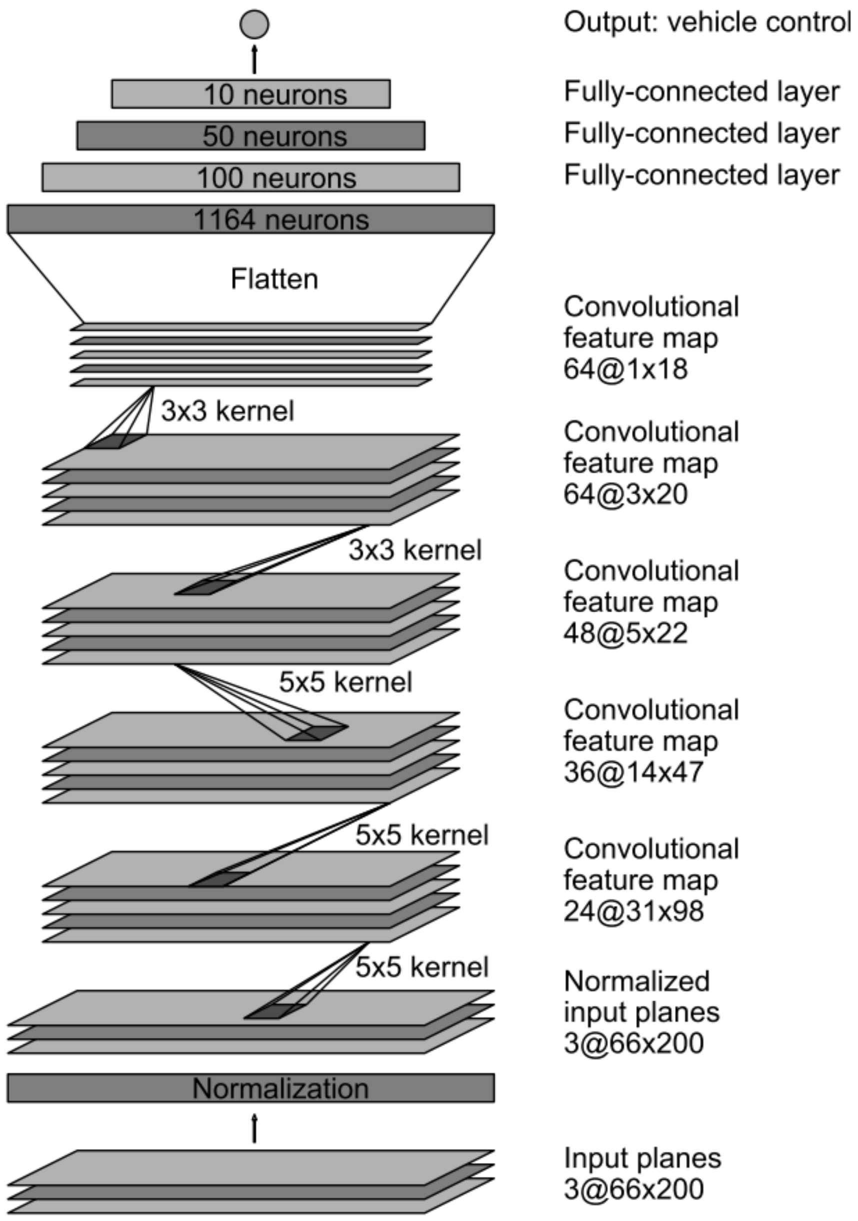

Figure 4: CNN architecture. The network has about 27 million connections and 250 thousand parameters.

3. Creation of the Training Set & Training Process

I noticed that the sample data provided was focused in only certain areas of the track. I decided to create three sets of additional training data on the sharp curves following the bridge. I also decided to create one lap of data headed the opposite way to help the model generalize.

Each training set recorded three images, one center, one at a left angle, and one at a right angle. The training data loading process placed all of these images together in a single queue in memory, alongside a corresponding steering value. The steering value for the left and right camera images was adjusted by plus-minus 0.225 degrees respectively.

With a validation split of 20%, I trained on 30304 samples, validated on 7577 samples. Shuffling was used before splitting.

I used the validation set along with some plotting of MSE loss to determine how badly the model was overfitting, if at all. I used these plots to determine that 3 epochs was the ideal time to stop training my model, and lowering the learning rate helped as well, even though it was an adam optimizer.