Submitted by Kyle Martin

Project 3
Behavioral Cloning

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

Quality of Code

My submission includes all required files and can be used to run the simulator in autonomous mode.

My project includes the following files:

File	Description
train.py	Contains the model and training script
drive.py	Uses a trained model to drive the car in the simulator
model.h5	Trained model for predicting steering angle given an image
project3_report.pdf	This file. A report on my learnings and results for this behavioral cloning project.
upload_to_aws.sh	Script to upload training data to AWS
download_run_model.sh	Script to download trained model from AWS and run drive.py
update_train_script.sh	Script to just update the training script on AWS with the local train.py script

Model Architecture and Training Strategy

General Learnings

The data collection process arguably was the hardest part of this project. It taught me that a great model alone, will *not* produce reliable and robust performance. In fact, having solid training and validation data is probably the most important piece of machine learning.

I also learned that it is an incredibly iterative process. Sometimes you think you've collected some great training data. It has low training and validation error but when the model is put to the test, it falls flat on its face. I realized how important it is to provide a 'gradient' of training data, where by the training set gives the model enough information to accurately interpolate between scenarios.

The Iterative Collection Process

Some image used in training

It took many iterations to find the right way to collect training data. The general process was:

1. Collect data with the simulator
2. Copy to AWS
3. Train the model
4. Download the model
5. Test the model in the simulator
6. Add more training data to handle edge cases

At first I tried recording myself driving a 2 laps while trying to stay in the center lane. This model performed decently but wasn't able to handle sharp turns.

The second training set I tried was driving around 4 laps with good driving behavior. This did help the car handle turns better but it still drove off the track a lot.

The next thing I tried was adding some training data by driving around the track counter-clockwise. Theoretically this should help the model generalize by reducing steering bias. I found that it actually made my model less confident and ended up lowering performance. I believe this is because I was steering with the car with the keyboard arrow keys. I noticed this this style of steering the car produce very a non-continuous steering angle data. Trying to create a model that has a smooth output with non-smooth training data is not very productive!

This lead me to start recording data with the mouse. I also tried to drive under 7 MPH since the training sampling rate is quite slow. I really took my time to drive slowly and ensure the car is in the middle of the lane. Surprisingly the model performed much better but it still crashed when it saw the bridge.

In order to prevent the model from crashing in certain spots of the track, I recorded data showing the car in move from an extremely bad position on the road into safe position on the road. While this did help the car avoid certain avoid from crashing in certain spots it actually made the model less confident and it still crash.

After many days of recording and creating dozens of training sets, I discovered one key element that I didn't take into account: interpolation. I was teaching the model how to behave in an ideal center lane case and also how to recover from extreme cases. *What I didn't do well was help the model understand how to incrementally recover from small mistakes. I was too focused on helping it recover after it too late!*

Once I realized this, I created a new training set with 1 lap of good behavior driving, 2 examples of good behavior around each turn, 6 examples of how to recover from minor veerings toward the sides of the road, and 3 examples of how to recover from sharp turns (when it was almost too late).

I like to call this a 'situation gradient' because it gives enough information to the model to properly interpolate between the different situations I've handed it. This model performed quite well and was able to drive the complete track. Hooray!

Ultimately, I discovered that one must be strategic and very aware of what you're signaling to the model. Even if your model has low validation error, it doesn't necessarily

indicate that performance will be high in testing. It's very important to ask at least these three questions:

1. "What will this extra data teach my model about the world?"
2. "Will this cause my model to begin to underfit or overfit?"
3. "Have I shown the model enough information to properly interpolate between the situations I've placed before it?"

Automated Tooling

In order to speed up the iterative process outlined above, I wrote several shell scripts to help automate some of these processes.

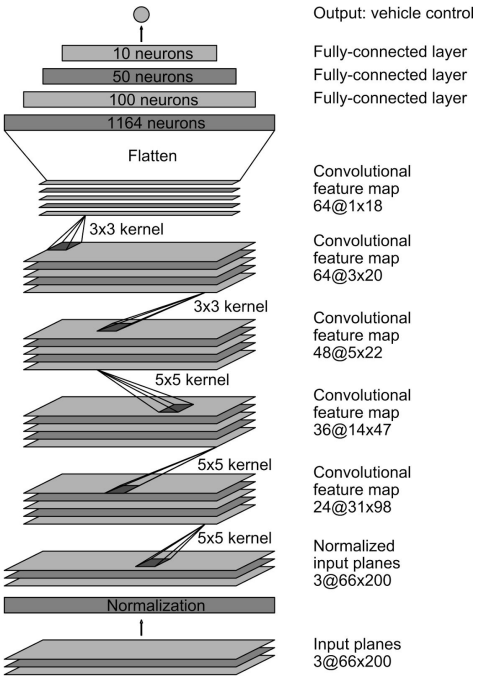
Mainly, the biggest pain points were:

1. Getting an updated dataset to the cloud
2. Getting an updated model from the cloud

So I decided to write some basic shell scripts to automate these tasks. I think a software business could be created to make this workflow for self-driving car companies easy.

Model Architecture

The architecture I ended up using was originally designed by folks at Nvidia for this same problem. The only difference is the image input size is 160x320x3. The architecture is fairly straightforward:



Here's a summary of the Keras model obtained by printing out mode.summary():

Layer (type)	Output Shape	Param #	Connected to
lambda_1 (Lambda)	(None, 160, 320, 3)	0	lambda_input_1[0][0]
cropping2d_1 (Cropping2D)	(None, 90, 320, 3)	0	lambda_1[0][0]
convolution2d_1 (Convolution2D)	(None, 43, 158, 24)	1824	cropping2d_1[0][0]
convolution2d_2 (Convolution2D)	(None, 20, 77, 36)	21636	convolution2d_1[0][0]
convolution2d_3 (Convolution2D)	(None, 16, 73, 48)	43248	convolution2d_2[0][0]
convolution2d_4 (Convolution2D)	(None, 14, 71, 64)	27712	convolution2d_3[0][0]
flatten_1 (Flatten)	(None, 63616)	0	convolution2d_4[0][0]
dense_1 (Dense)	(None, 100)	6361700	flatten_1[0][0]
dense_2 (Dense)	(None, 50)	5050	dense_1[0][0]
dense_3 (Dense)	(None, 10)	510	dense_2[0][0]
dense_4 (Dense)	(None, 1)	11	dense_3[0][0]

Total params: 6,461,691
Trainable params: 6,461,691
Non-trainable params: 0

Preventing Overfitting

In order to reduce overfitting, I experimentally determined that around 10,000 training samples and keeping training epochs to 3 was ideal.

Learning Rate Parameters

I chose to optimize with Adam over vanilla SGD for the following reasons. First, it Adam has been shown to outperform SGD many times over. Second, it's also been shown that SGD can take a long time to converge. Third, SGD introduces more training parameters and therefore more potential for bugs. More parameters also increases the maintenance burden for future engineers.

Simulation

Overall my final model performed decently when testing with the simulator. I really wish the simulator did not have the ability to drive the car with arrow keys. It lead my model to perform badly and it doesn't produce continuous measurement like you would actually get from a real car.