Udacity Self-Driving Car Nanodegree
Project 3: Behavioral Cloning
by Graham Arthur Mackenzie

The goals / steps of this project are the following:
- Use the simulator to collect data of good driving behavior
- Build a convolutional 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 1 without leaving the road
- Summarize the results with a written report

Rubric Points

Here I will consider the rubric points (https://review.udacity.com/#!/rubrics/432/view) individually and describe how I addressed each point in my implementation.

I. 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 convolutional neural network
- `writeup_report.pdf` (this document) summarizing the results
- `run1.mp4` of my successful run around track 1

2. Submission includes functional code

Using the Udacity provided simulator and the output of my `model.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 convolutional 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.

II. Model Architecture and Training Strategy

1. An appropriate model architecture has been employed

My model consists of a convolutional neural network with some 3x3 and some 5x5 filter sizes, and depths between 24 and 64 (`model.py` lines 123-127.)

The model includes RELU layers to introduce nonlinearity (`model.py` lines 123-127), and the data is normalized in the model using a Keras lambda layer (`model.py` line 115.)

2. Attempts to reduce overfitting in the model

The original LeNet model contained dropout layers in order to reduce overfitting.

The model was trained and validated on different data sets to ensure that the model was not overfitting (namely, the one provided and the one I collected.)

The model was tested by running it through the simulator and ensuring that the vehicle could stay on the track.

3. Model parameter tuning

The model used an Adam optimizer, so the learning rate was not tuned manually (`model.py` line 137.)

4. Appropriate 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 focusing on driving smoothly around curves. These three types of driving were done in both directions around each of the two tracks to mitigate any inherent chiral bias.

For details about how I created the training data, see the next section.

III. Model Architecture and Training Strategy

1. Solution Design Approach

The overall strategy for deriving a model architecture was to follow the steps laid out by David. :)

My first step was to use a convolutional neural network model similar to the LeNet. I thought this model might be appropriate because I have been told it would be by people who know a lot more than I do. :) I eventually evolved to the Nvidia network for the same reason.

In order to gauge how well the model was working, I split my image and steering angle data into a training and validation set (duh. :) Alright, enough following the boilerplate laid out in the writeup report template… ;)

Initially, rather than engaging in data augmentation via flipping the images and using multiple camera angles, I thought that I would simply collect and make use of my own data. Collecting the data proved easy enough. As mentioned in II.4 above, I did (in both directions around both tracks) a combination of two or three laps of center lane driving, one lap of recovery driving from the sides, and one lap focusing on driving smoothly around curves. Then I set aside this data and simply got my model up and running on the provided data.

Per David's suggestion of being, "the first obvious step to improve the model," I preprocessed the data by normalizing and mean-centering it with a Keras Lambda layer.

Then I cropped out the sky, trees, mountains, and hood of the car via a Cropping2D layer.

In preparation for transitioning from working on the provided data set to the set I had gathered (which was almost 4x larger) I got the generator() method up and running, so that the model could handle the data in batches.

Then I instituted color space conversion from BGR, such as cv2.imread() engages in, to RGB, which is what drive.py is expecting to see.

At this point, I got hung up for quite some time on an apparently minor code tweak that needed to happen? (more on this later.) It manifested itself as this error

```
ValueError: Error when checking model input: expected lambda_input_1 to have 4
dimensions, but got array with shape (32, 1)
```

preceded by an error associated with whichever function I happened to be using for color space conversion.

I couldn't puzzle out if this was in fact a problem with the color space conversion function, the generator method, or some combination thereof. But (for better or worse) this problem did not manifest itself when the same code was applied to the smaller, provided data set. Thus, I set aside working on the larger data set for the nonce, and went back to what works.

Because my larger data set was effectively unavailable to me, I went back to increasing my data set via augmentation.

Using all three camera angles allowed my car to make it all the way around the track! But (unacceptably) drive up on the curb at places.

Doubling the data size by adding horizontally flipped versions of the images improved my loss numbers somewhat, but did not noticeably change the above. It was thus obvious that there was some further tweak in my code that was going to need to happen in order to take full advantage of the extra data.

Digging into the issue of why my model kept failing on the larger data set I had culled, I found many references on the forums to how:

```
        import sklearn
        …
        yield sklearn.utils.shuffle(X_train,ytrain)
```

apparently needs to be changed to:

```
        from sklearn.utils import shuffle
        …
        yield shuffle(X_train,y_train)
```

This did help my Epochs get much further along, but training would still hang on, apparently, the combination color conversion / model input ValueError mentioned above.

Worse than this, the loss rate -- by the time training on the larger data set failed -- was no longer even close to the results we had been achieving on the smaller data set (that is, we had been able to get down to ~ 0.014 on the smaller data set, while the larger one was closer to 0.09 when it was hanging.)

Somewhat out of desperation, and somewhat out of an uncertainty of what else I could work on, I decided to implement the image size reduction that others had been mentioning on the forums. I used cv2.resize() for the task, which is not an approach I have seen elsewhere / had to research myself. Because I was now shrinking the images by half, I had to similarly reduce the dimensions of the Cropping2D layer so that it wouldn't chop off more of the now smaller images than was desired.

With this new functionality implemented, I tore my hair out a bit longer trying to get it to finish training on the larger data set. Finally, I went back to test it on the smaller data set (as I had been occasionally doing) just to make sure it still actually worked, and all of a sudden the validation loss rate got down to 0.0105 by the end of 3 Epochs!

At which point, the vehicle was able to drive autonomously around the track without leaving the road. :)

2. Final Model Architecture

The final model architecture (`model.py` lines 112-134) consisted of a convolutional neural network with the following layers and layer sizes …

Lambda layer for preprocessing
Cropping2D layer for cropping
Convolution2D layer 24 depth, 5x5 filter
Convolution2D layer 36 depth, 5x5 filter
Convolution2D layer 48 depth, 5x5 filter
Convolution2D layer 64 depth, 3x3 filter
Convolution2D layer 64 depth, 3x3 filter
Flattening layer
Fully connected Dense layer depth 100
Fully connected Dense layer depth 50
Fully connected Dense layer depth 10
Fully connected Dense layer depth 1

3. Creation of the Training Set & Training Process

To capture good driving behavior, I first recorded three laps on track 1 using center lane driving. I then recorded the vehicle recovering from the left and right sides of the road back to the center so that the vehicle would learn to do the same. Then, per the suggestions in lesson 16. More Data Collection, I also added one lap focusing on driving smoothly around curves. I next did all of the above in the opposite direction, in order to counterbalance the turn bias of (eventually) both tracks. Then I repeated this process on track 2 in order to get more data points.

To augment the data sat, I also flipped the images and angles thinking that this would help, insofar as more data = good. After the collection process, I had almost 84,000 additional data points. I then preprocessed this data by adding a Lambda layer, to normalize and mean-center it, as described above (II.1 and III.1). 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 underfitting. The ideal number of Epochs was 3, as evidenced by the fact that that's how many I was using when my model produced acceptable results at last. (Forgive me, I struggle to keep a straight face when filling in boilerplate. :) If this hadn't worked, I was prepared to up the number of Epochs to 5. I used an Adam optimizer so that manually training the learning rate wasn't necessary.

Thanks so much for taking the time to read my words/code! :) GAM