Udacity SDC – P3: Behavioral Cloning Write-up Report

By: James Beasley

In this report 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 model (convolution neural network)
- **track1.mp4** and **track2.mp4** videos demonstrating the model in action (using the stable simulator)
- **writeup_report.pdf** summarizing the results

2. Submission includes functional code

Using the Udacity-provided **stable** simulator and my **drive.py** file, the car can be driven autonomously around **track 1** and **track 2** by executing:

```
python drive.py model.h5
```

Note that there is an assertion at the beginning of my **model.py** file that checks to ensure the current working directory is the **data** directory of the **Udacity-provided training data set**. So please ensure you execute the statement above within that directory.

The only modification I made to the Udacity-provided **drive.py** file was to boost the throttle when the car encounters a significant positive incline, such as the inclines in **track 2** of the **stable simulator**. (drive.py lines 37-40)

3. Submission code is usable and readable

My **model.py** file contains the code for training, validating, and saving the model. It also contains comments that explain how the code works.

The Keras "fit_generator" (model.py lines 271-277) leverages callbacks (model.py lines 263-268) to early-stop if the model isn't improving (based on validation loss) over successive epochs and also save the best model (also based on validation loss) over the course of the range of epochs defined (in my case 4 epochs). I converged on a max of 4 epochs based on a series of longer runs that on-average didn't improve past 4.

I have also made use of Python generators to yield batches of training and validation data without needing to load full training or validation sets into memory (model.py lines 258 and 260). The training generator randomly loads a batch of images at runtime from the Udacity-provided training data set, performs random transformations on those images (described later in this document), and yields a batch (model.py lines 197-217). The validation generator is similar, but walks through the validation set in chunks/batches (vs. assembling a batch randomly). (model.py lines 220-240)

Model Architecture and Training Strategy

1. An appropriate model architecture has been employed

My model consists of a convolutional neural network (based on the NVIDIA architecture) of 2 pre-processing layers (cropping & normalization), 5 convolutional layers, and 3 fully-connected layers (not including the output layer). It also leverages ReLU's for nonlinearity and dropout for regularization. See lines 94-140 of model.py. Greater detail can be found in the "final model architecture" section of this document.

2. Attempts to reduce overfitting in the model

To validate my model and ensure it wasn't overfitting/underfitting, I carved out a validation data set from the **Udacity-provided training data set** (model.py line 79). I also leveraged the "dropout" regularization scheme after each of the last two fully-connected layers to specifically reduce overfitting. (model.py lines 130 and 135)

My model was trained and validated with an eye toward minimizing training and validation loss, but there were no target values for those metrics (model.py lines 251-277). I would watch to ensure the training and validation loss were relatively close and not too high. My **model.h5** has a **training loss of:** 0.0374 and a **validation loss of:** 0.0240. Though not the lowest compared to other runs, it performed the best (in relation to smoothness in the simulator). The litmus test for the model was actually running it through the stable simulator and ensuring it not only stayed on the track, but took turns well, and wasn't too jittery with regard to steering.

3. Model parameter tuning

My model leverages an **Adam** optimizer, so the learning rate was not tuned manually (model.py line 254). I used a constant of **0.5** (fed as a parameter to the model creation function – **cnn_model**) for the dropout keep-probability in the last two layers. (model.py line 251)

4. Appropriate training data

To ensure my model was not afflicted with straight-driving bias, I dropped all zero degree steering angle training examples and focused on the left and right camera images from the Udacity-provided training data set to teach the model to recover back to the center of the track (model.py lines 30-59). Given the offsets used to simulate recovery (model.py lines 32-33), near-zero steering angles were reintroduced into the data set, but this time in a uniform way (model.py lines 51 and 55). See the "creation of the training set & training process" section for more detail on the training strategy.

Model Architecture and Training Strategy

1. Solution Design Approach

After reading the NVIDIA paper I decided to use their architecture as a base; adding ReLU activations to introduce nonlinearity and dropout regularization to combat overfitting. (model.py lines 94-140)

I had success in project 2 with adding dropout in the last two fully-connected layers so I started with that in this model and it worked well, so I didn't pursue further regularization.

As stated before, I didn't specifically target a training or validation value, but used training/validation loss to understand whether the model was working appropriately.

With regard to synthetic data creation, the transforms I focused on where the same base set that worked for me in project 2, with the addition of y-axis image flipping. These transformations were, random x/y translation, random image brighten/darken, and as stated before, flipping images about the y-axis (model.py lines 148-194). I also instituted a rule that, although multiple transformations could be done to a single training image, these transformations must be distinct (can't be repeated, such as two rounds of translation to the same image). I accomplished this by removing transformation options as they occurred (model.py lines 186-192).

After I had the pipeline in place I decided to run it with the simulator to see if I was even in the ballpark. I was proud that right out of the gate, the first time I ran the model in the **stable simulator** I was able to complete **track 1** (though some of the turns were close to the edge) and got all the way to the tight right turn on **track 2** before crashing; definitely gave me a confidence boost that I was close.

I had started out with a small amount of data per epoch for that initial test (5120 training examples – 20 * batch size of 256) and no validation examples. So given that my model was pretty close, I decided to up my data per epoch vs. tweaking anything else. I upped the training examples per epoch to **25600** (100 * batch size of 256) and **5120** (20 * batch size of 256) validation examples and on my next run made the tight right turn on track 2 and was able to successfully complete both tracks! This also helped tighten up my driving on track 1.

After that, the only parameters I tweaked were the offsets that I applied to the left and right camera images when folding them into the training data (to be used as center camera training examples). I experimented with fixed values (0.15, 0.2, 0.25, 0.3), but then converged on randomly assigning an offset within a **uniform distribution in the interval [0.2, 0.25]**. This helped tighten up my straight driving and also smoothed my turns. (model.py lines 32 and 33)

2. Final Model Architecture

In this section I'll describe my model architecture in detail:

To begin with, the first two (pre-processing) layers are: a layer that crops the input image from 160x320x3 down to 66x200x3 (which the NVIDIA architecture expects), and a normalization layer that ensures the pixel values are within the interval [-0.5, 0.5] (model.py lines 98 and 100).

I briefly investigated resizing images vs. just cropping, but didn't witness a benefit beyond memory savings.

A depiction of the cropping layer is shown below:

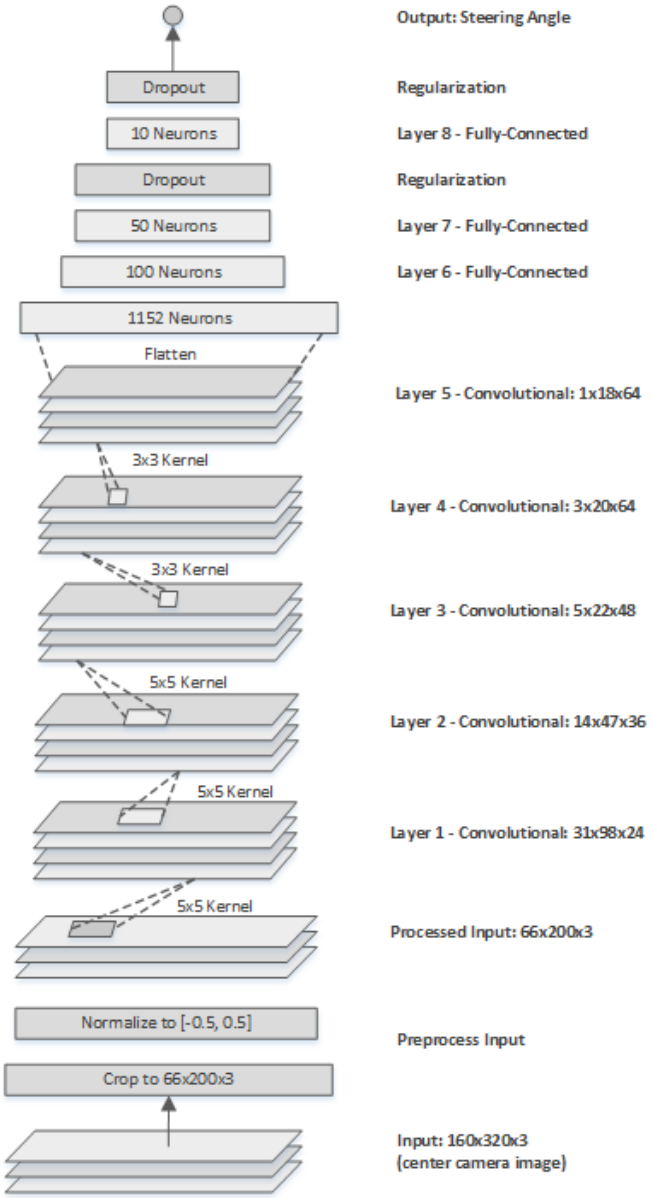
```
Input shape: (160, 320, 3)
Output shape: (66, 200, 3)
```

The 5 convolutional layers are: first, 3 layers with 5x5 filters (using valid padding) and depths between 24 and 48 (model.py lines 104, 108, and 112), followed by 2 layers with 3x3 filters (also using valid padding) each with a depth of 64 (model.py lines 116 and 120).

The model also leverages **ReLU** activations to introduce nonlinearity at every layer (excluding the pre-processing and output layers). These activations are considered inherent within each layer and not called out specifically in my model architecture diagram (depicted below).

The Keras weight-initialization scheme I chose to use for each layer (example on line 104 of model.py) was "**lecun_uniform**", which is scaled by the square root of the number of inputs. (a scheme I used on project 2 with good success)

Below is a visualization of my final model architecture:



Below is the Keras model summary of my final model architecture:

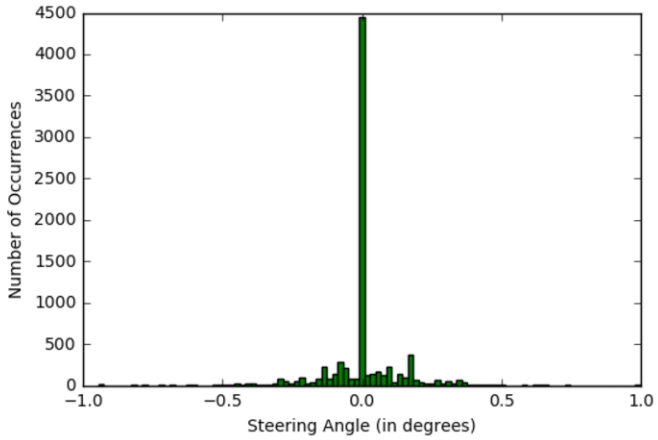
Layer (type)	Output Shape	Param #	Connected to
cropping2d_1 (Cropping2D)	(None, 66, 200, 3)	0	cropping2d_input_1[0][0]
lambda_1 (Lambda)	(None, 66, 200, 3)	0	cropping2d_1[0][0]
convolution2d_1 (Convolution2D)	(None, 31, 98, 24)	1824	lambda_1[0][0]
convolution2d_2 (Convolution2D)	(None, 14, 47, 36)	21636	convolution2d_1[0][0]
convolution2d_3 (Convolution2D)	(None, 5, 22, 48)	43248	convolution2d_2[0][0]
convolution2d_4 (Convolution2D)	(None, 3, 20, 64)	27712	convolution2d_3[0][0]
convolution2d_5 (Convolution2D)	(None, 1, 18, 64)	36928	convolution2d_4[0][0]
flatten_1 (Flatten)	(None, 1152)	0	convolution2d_5[0][0]
dense_1 (Dense)	(None, 100)	115300	flatten_1[0][0]
dense_2 (Dense)	(None, 50)	5050	dense_1[0][0]
dropout_1 (Dropout)	(None, 50)	0	dense_2[0][0]
dense_3 (Dense)	(None, 10)	510	dropout_1[0][0]
dropout_2 (Dropout)	(None, 10)	0	dense_3[0][0]
dense_4 (Dense)	(None, 1)	11	dropout_2[0][0]

Total params: 252,219
Trainable params: 252,219
Non-trainable params: 0

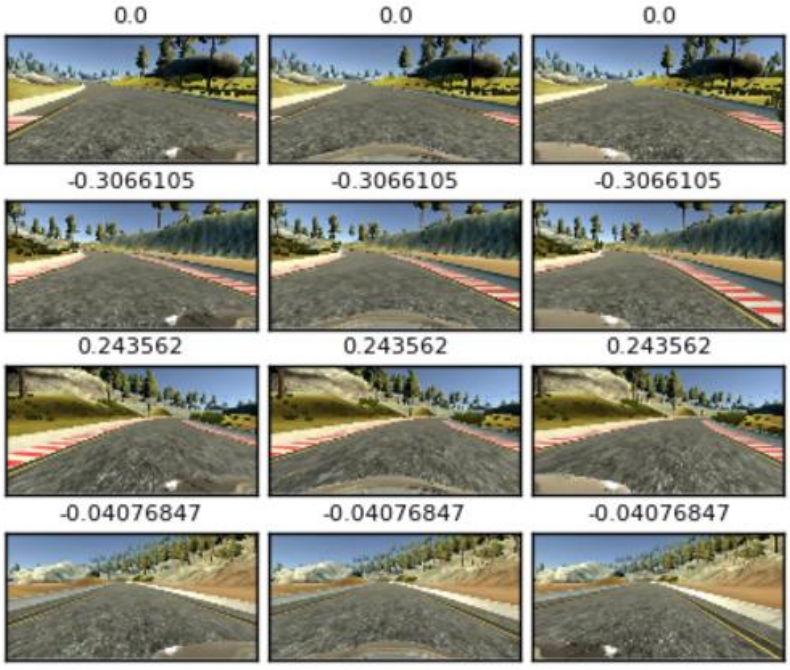
3. Creation of the Training Set & Training Process

I had planned to create my own training data using the stable simulator, but was unable to get my PS3 controller to work with my laptop. My trials while using the keyboard arrow keys alone produced driving data that was very erratic (my slow laptop also contributed), so I decided to instead leverage the **Udacity-provided training data set**. This proved to work just fine for me. In the following diagrams, you can see some of the data exploration I did on it:

```
Num drive log steering angles: 8036
Min drive log steering angle: -0.9426954
Max drive log steering angle: 1.0
```



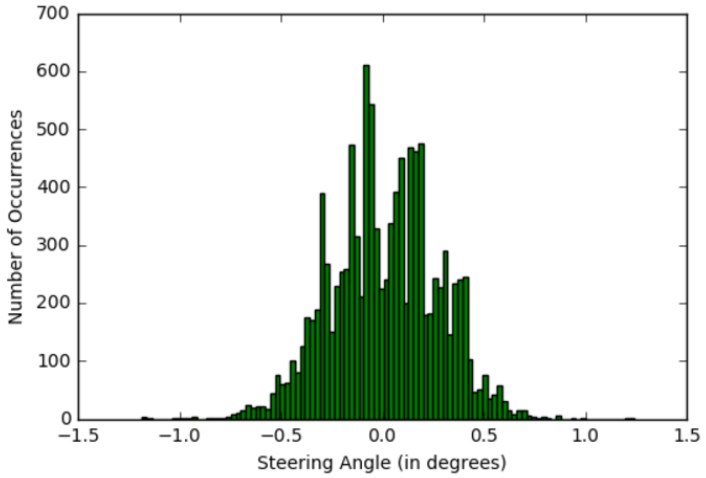
Below are four (unaltered) training examples, with left, center, and right camera images for each:



As discussed before, to deal with the bias toward straight driving (large zero-degree steering angle count depicted above) I decided to drop all zero-degree steering angle training examples. I then folded in the left and right camera images, adding an offset randomly sampled from a uniform distribution of interval [0.2, 0.25] for the left camera and subtracted an offset randomly sampled from a uniform distribution of interval [0.2, 0.25] for the right camera. (model.py lines 30-59)

Below is what the distribution looks like after dropping all zero-degree steering angle training examples and folding in the left and right camera images. This also gave a nice boost to my data set (from the original ~8k to now ~11k).

```
Num drive log image paths: 11025
Num drive log steering angles: 11025
Min drive log steering angle: -1.18403625678
Max drive log steering angle: 1.24423246256
```



After that was complete, I then shuffled the training data set (model.py line 74) and then carved out a validation set (20% of the training set - ~2200 images) to use in verifying the integrity of the model during training (model.py line 79). Also to see if there were inherent indications of overfitting/underfitting.

Python generators came in really handy to build synthetic data on-the-fly during training (vs. building it all beforehand). As stated earlier, I leveraged many of the transformations I used in project 2 (except for flipping images about the y-axis). I tried to keep the amount of transformation to a minimum: +-15 pixels (drawn from a uniform distribution of those bounds) for x/y translation (adjusting steering angle for x translations with a static offset multiplier of 0.003 – converged upon via experimentation), and a V-channel (from HSV) multiplier of +0.2 to +1.0 (drawn from a uniform distribution of those bounds) for image brightness adjustment (brighten or darken). When flipping images I also inverted the sign of the steering angle.

(model.py lines 148-177) – All in all, these transformations (introduced during training) really helped the model generalize to the second (unseen) track.

Below is a sampling (referred to as a batch below) of images randomly chosen and transformed by the training generator (model.py lines 197-217). There are translated (x/y shifted), flipped, and brightened/darkened images. **Original** represents before and **transformed** represents after. Note that between 1 and 3 transformations can take place per image. (model.py lines 180-194)

Randomly chosen batch - original:

Randomly chosen batch - transformed: