Behavioral Cloning

Writeup Template

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

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 `sh 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.

Model Architecture and Training Strategy

1. An appropriate model architecture has been employed

I used the NVIDIA network architecture recommended as one of the powerful nets in Udacity lectures. My model consists of:

1. Normalization layer that takes 64x64 images and scales the values to the range of -0.5 to 0.5.
2. 3 Convolution layers with 5x5 kernels with 24, 36, 48 filters respectively. Each Convolutional layer uses RELU activation for introducing nonlinearity.
3. 2 Convolutional layers with 3x3 kernels with 64 filters each. Each Convolutional layer uses RELU activation for introducing nonlinearity.
4. Flatten layer
5. Fully connected layer with 100 neurons
6. Dropout layer for reducing overfitting with a drop factor of 0.5
7. Fully connected layer with 50 neurons
8. Dropout layer for reducing overfitting with a drop factor of 0.5
9. Fully connected layer with 10 neurons
10. Final fully connected layer with one output neuron

Here is the summary of the model produced by calling `model.summary()` on my Keras implementation:

```
____________________________________________________________________________________________________
Layer (type)                     Output Shape          Param #     Connected to
====================================================================================================
lambda_1 (Lambda)                (None, 64, 64, 3)     0           lambda_input_1[0][0]
____________________________________________________________________________________________________
convolution2d_1 (Convolution2D)  (None, 30, 30, 24)    1824        lambda_1[0][0]
____________________________________________________________________________________________________
convolution2d_2 (Convolution2D)  (None, 13, 13, 36)    21636       convolution2d_1[0][0]
____________________________________________________________________________________________________
convolution2d_3 (Convolution2D)  (None, 5, 5, 48)      43248       convolution2d_2[0][0]
____________________________________________________________________________________________________
convolution2d_4 (Convolution2D)  (None, 3, 3, 64)      27712       convolution2d_3[0][0]
____________________________________________________________________________________________________
convolution2d_5 (Convolution2D)  (None, 1, 1, 64)      36928       convolution2d_4[0][0]
____________________________________________________________________________________________________
flatten_1 (Flatten)              (None, 64)            0           convolution2d_5[0][0]
____________________________________________________________________________________________________
dense_1 (Dense)                  (None, 100)           6500        flatten_1[0][0]
____________________________________________________________________________________________________
dropout_1 (Dropout)              (None, 100)           0           dense_1[0][0]
____________________________________________________________________________________________________
dense_2 (Dense)                  (None, 50)            5050        dropout_1[0][0]
____________________________________________________________________________________________________
dropout_2 (Dropout)              (None, 50)            0           dense_2[0][0]
____________________________________________________________________________________________________
dense_3 (Dense)                  (None, 10)            510         dropout_2[0][0]
____________________________________________________________________________________________________
dense_4 (Dense)                  (None, 1)             11          dense_3[0][0]
====================================================================================================
Total params: 143,419
Trainable params: 143,419
Non-trainable params: 0
```

2. Attempts to reduce overfitting in the model

The model contains dropout layers in order to reduce overfitting. The model also uses L2 weight regularization on convolutional and fully connected layers.

The model was trained and validated on different data sets to ensure that the model was not overfitting. 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 Architecture and Training Strategy

Solution Design Approach

The main strategy from the start was to see what the minimum data preprocessing would be necessary to teach the car how to drive around the track. Ideally, I was hoping to get the network to learn the behavior with just enough of the "right" data.

Right away I chose NVIDIA's network that many people recommended. I assumed I should be able to get it to work and I can focus my time and energy on the data. I made minimal changes to the network, and iterated through different data collection and processing approaches. I only added regularization and dropout layers because I noticed right away that often the training loss would go down while the validation loss remained the same or increased.

After every change I would run the model and see how it drives around the track. The goal was to achieve 30MPH without touching any yellow lines.

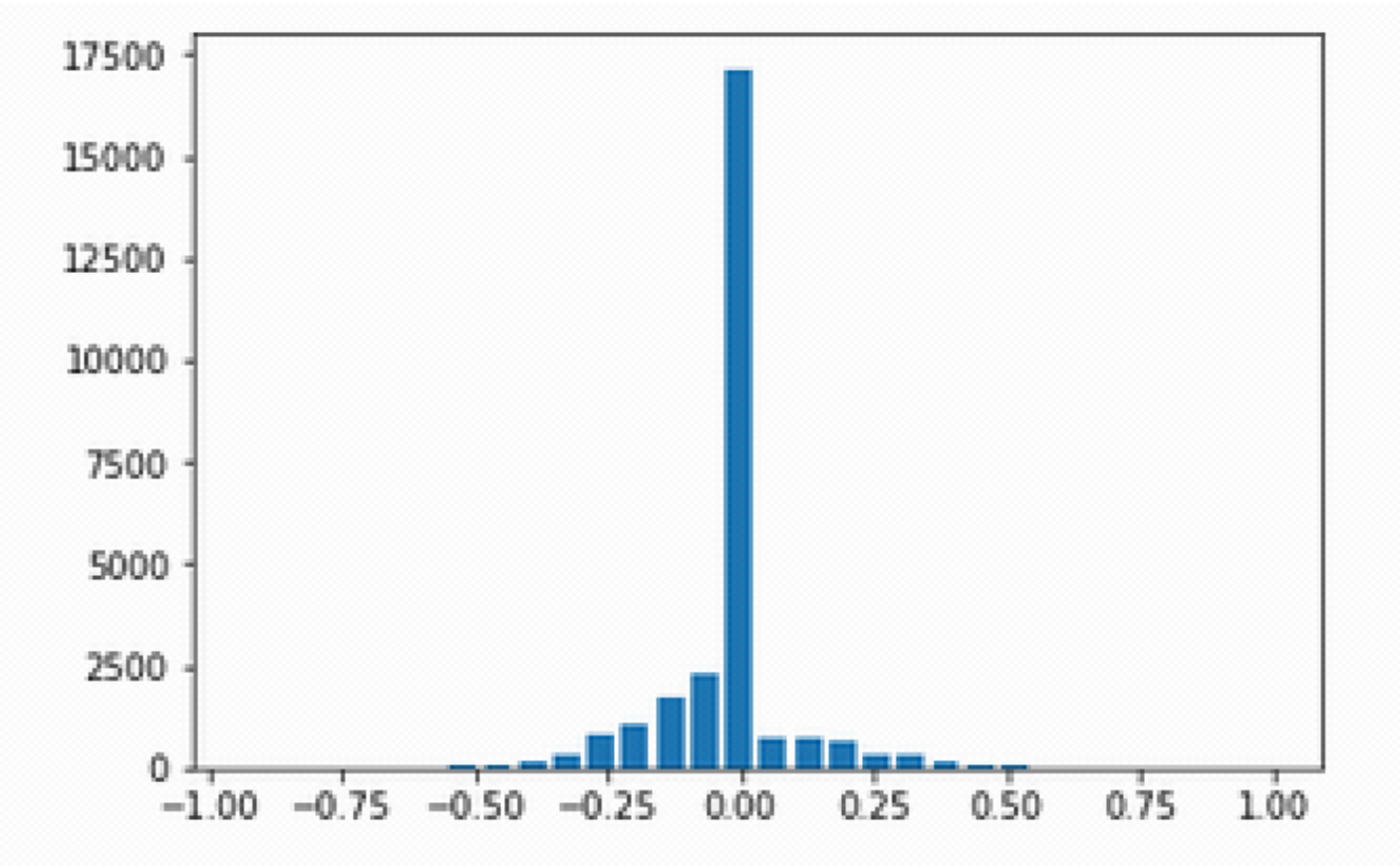
All of the progress happened by collecting training data and using different data processing techniques, so I describe all the steps I took below.

Creation of the Training Set & Training Process

1. When I first started training, my car would drive fine until it hit the sharp curves. It would try to turn but not enough, which would lead it to often exit the curve. My assumption was that it simply didn't learn how to handle curves sufficiently and that it needed more data in those sections of the road. I took two approaches to increasing the turning data:

   a) I simply collected more turning data by driving the turns

b) I kept only 50% of data points where steering angle measurement was zero. I played around with this percentage and ended up at 50. Eliminating too much of straight-driving data would cause the car to perform poorly on the straight segments of the road by swerving left and right constantly, instead of simply driving straight ahead.



2. Training only on Udacity's data didn't seem sufficient because the car performed rather poorly. Simply collecting more data points by driving laps around increased the performance significantly. The car could get around the first curve and onto the bridge, and drive the bridge without any issues.

3. There were certain areas of the road where the cas struggled more. One was the curve right after the bridge, it kept popping to the right and out of the road, but that got resolved by simply driving that section over and over again and giving the model that data to train on.

4. The second trouble area was the right turn after the bridge. This makes sense because when you look at the data distribution above, the steering angle is skewed heavily to the left. To compensate and obtain more data for the right turn, I call the random_flip method that flips the image along the vertical axis 50% of the time for larger steering angles and switched the sign of the steering angle. This provides us with the data that simulates the right turn based on the left turn images. I also recorded more data by driving the right turn in the training mode.

5. All the images were cropped to get rid of the sky and the car at the bottom, this right off the bat improved

the network performance.

6. All the images were resized to 64x64 as some students suggested, and this sped up the training time and made iterating on the model more efficient.

7. All the images were normalized to the -0.5 to 0.5 range in the first lambda layer of the network.

Recovery training

This was a big topic in the forums, in the medium posts I read about this project as well as the lectures. How do we teach the car to know how to recover? By recover we mean to stear back to the middle of the road when it finds itself at the edges of the road. The two approaches proposed were to either record recovery events in the training mode or to use left and right cameras with some constant factor to simulate recovery events.

The first approach seemed a bit tedious. The second I have tried and spent quite a bit of time on it, but for some reason I just couldn't get it to work. If I chose a steering angle constant that was larger, the car would start to oscillate quite a bit and eventually become unstable flying out of the road. If I chose a steering angle that was small, then there would be not much of a recovery event obtained. A lot of students suggested using 0.25 as an angle that worked for them, but in my case it caused oscillactions. Eventually I gave up and went for a bit of a different approach.

The approach I took is similar to using left and right cameras but instead uses only the central photo, and then shifts the image horizontally to the left or to the right by cropping the sides. Function that does this is in `model.py` called `random_shift`. It takes the possible ranges of the shift, -25 to 25 (if -25, it would crop the image from 0 to image*width-50, which would be similar to having a camera on the left) (if 25 it would crop the image from 50 to image*width, which would be similar to having a camera on the right) and it crops the image so that the road is shifted to the left or to the right with some random shift of pixels in that range.

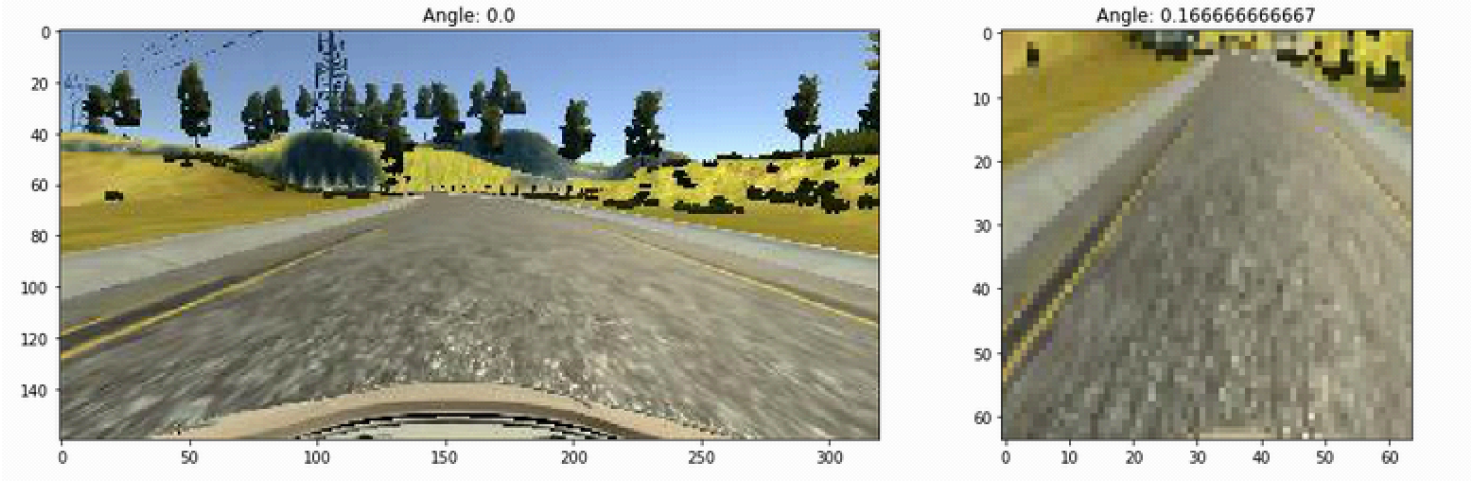
Of course, when we shift the image, we need to change the steering angle as well to simulate that the car was responding to the conditions in the shifted image, so we adjust the angle by the amount of the shift divided by some adjustable constant. The constant of 3 seemed like it performed well after exploring a different values.

This approach worked better for me that using left and right camera images, possibly because of the randomization that introduces a more even distribution of shift, unlike having to extremes, left and right that cause oscillations.

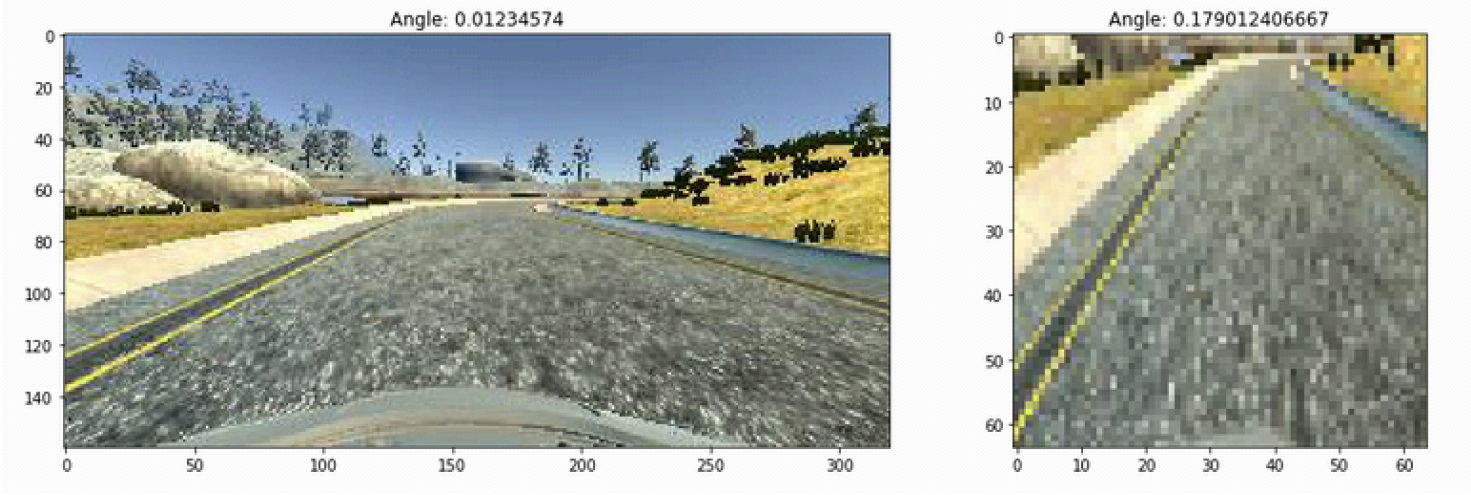
Visualizing Data

Here is the visualization of the before and after of the entire data processing pipeline. On the left are the input images, and on the right are the cropped, scaled and shifted images with the maximum shift to the left, just to demonstrate the simulation of the recovery angles.

Here is the example for the input image of Angle 0. It results in the image as if the car was driving more on the left and the resulting steering angle positive, indicating the car should turn towards the right to turn towards the center.



Here is an example of the car already turning right, and us creating an image that appears as though the car was even more to the left, driving near the lane. Angle is appropriately scaled to a larger right turn, simulating a recovery event from the left to the right.



These are just extreme examples of the "left shift", but as mentioned before, the amount of shift is randomized for each sample.

Of course all of that data was shuffled and divided into training and validation sets.

Results

Final implementation works well on the first track. At speeds of 30 MPH ends up touching the side line occasionally. If left to run for a long time it eventually gets out of the road, I'm assuming because it becomes

unstable at some point and the controller starts to overcompensate.

The model doesn't work on the second track. I assume it's because my data lacks any brightness enrichment, and the second track's light and road are quite different.