
Behavioral Cloning Project

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 one without leaving the road
- Summarize the results with a written report

Rubric Points

Here 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 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.

Model Architecture and Training Strategy

1. An appropriate model architecture has been employed

The overall strategy for deriving a model architecture was to start with a simple, yet powerful network.

My first step was to use the LeNet-5 architecture. I thought this model might be appropriate because it works well with images. Since the car's cameras capture video footage and a video is just a series of images, I figured this would be a good starting point.

Eventually, I decided to go with an architecture similar to the one NVIDIA used for it's end-to-end deep learning for self-driving cars.

2. Attempts to reduce overfitting in the model

The model contains a dropout layer in order to reduce overfitting (model.py line 79).

The model was trained and validated on different data sets to ensure that the model was not overfitting. As a good rule of thumb, it is a good idea to use 80% of the data for the training set and 20% for validation set, so I used the sklearn library to split the data that way. The model was tested by running it through the simulator and ensuring that the vehicle could stady on the track.

3. Model parameter tuning

The model used an adam optimizer, so the learning rate was not tuned manually (model.py line 86).

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,

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

Model Architecture and Training Strategy

1. Solution Design Approach

In order to gauge how well the model was working, I split my image and steering angle data into a training set and a validation set. I found that my first model did a decent job but needed improvement.

After trying different model architectures, I decided to go with an architecture that NVIDIA used for it's end-to-end deep learning for self-driving cars. The mean squared error on the training set decreased after each epoch but actually increased on the third and fourth epochs on the validation set. This implied that the model was overfitting.

To combat the overfitting, I modified the model by adding a dropout layer and reducing the number of epochs.

The final step was to run the simulator to see how well the car was driving around track one. There were a few spots where the vehicle fell off the track. One of these was at the turn after the bridge. In order to improve the driving behavior in this case, I collected additional data around that point to provide even more examples of good driving there.

At the end of the process, the vehicle is able to drive autonomously around the track without leaving the road.

2. Final Model Architecture

The final model architecture (model.py lines 68-84) consisted of a convolutional neural network.

Here is a summary of the architecture:

Layer	Description	Output
Input	160x320x3 RGB image	
1.Lambda	Normalize	160x320x3
2.Cropping	70 pixels from top 25 pixels from bottom	65x320x3
3.Convolution	5x5 kernel, 2x2 stride, RELU activation	31x158x24
4.Convolution	5x5 kernel, 2x2 stride, RELU activation	14x77x36
5.Convolution	5x5 kernel, 2x2 stride, RELU activation	5x37x48
6.Convolution	3x3 kernel, RELU activation	3x35x64
7.Convolution	3x3 kernel, RELU activation	1x33x64
8.Flatten		2112
9.Fully Connected		100
10.RELU		100
11.Dropout	keep_prob = 0.5	100
12.Fully Connected		50
13.RELU		50
14.Fully Connected		10
15.RELU		10
16.Fully Connected		1

3. Creation of the Training Set & Training Process

To capture good driving behavior, I first recorded two laps on track one using center lane driving. Here is an example image of center lane driving:

I, then, recorded the vehicle recovering from the left side and right sides of the road back to center so that the vehicle would learn to what to do if it ever got off-center.

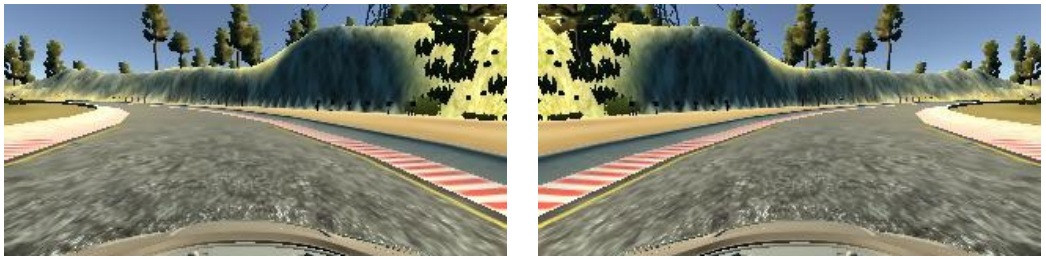
Then I repeated this process on track two in order to get more data points.

In order to have even more data, I decided to augment the data by using the images from the left and right cameras. This was done in lines 30-44 of the model.py file. This tripled my number of data points.

Here is an example of what the left, center, and right camera images look like:

I noticed that the track only had left turns. As a result, the model became great at making left turns but horrible at making right turns. In an attempt to remedy this, I decided o augment the data set by flipping the images and angles.

Here is an image that has then been flipped:



After flipping the images, I had 3 times as many data points.

I, finally, randomly shuffled the data set and put 20% of the data into a validation set.

The only preprocessing that I did normalizing the pixel values using a lambda layer as the first layer in my model (model.py line 69). After that, I used a cropping layer because I figured that the sky at the top of each image wouldn’t offer any valueable data for our model to learn.

I used this training data for training the model. The validation set helped determine if the model was overfitting or underfitting. I started with 5 epochs but ended up changing that to 3 due to overfitting. I used an adam optimizer so that manually training the learning rate wasn't necessary.