Behavioral Cloning

Writeup

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.md or 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 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

My model consists of a convolution neural network with 5x5/3x3 filter sizes and depths between 24 and 64 (model.py lines 53-58)

The model includes RELU layers to introduce nonlinearity, and the data is normalized in the model using a Keras lambda layer (code line 49).

2. Attempts to reduce overfitting in the model

The model contains dropout layers in order to reduce overfitting (model.py line 57).

The model was trained and validated on different data sets to ensure that the model was not overfitting (code line 67). 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 66).

4. Appropriate training data

Training data was chosen to keep the vehicle driving on the road. I used a combination of center lane driving in both the directions clockwise & anti-clockwise.

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

Model Architecture and Training Strategy

1. Solution Design Approach

The overall strategy for deriving a model architecture was to predict steering angles correctly so that the car stays on the center of the road.
My first step was to use a convolution neural network model similar to the LeNet architecture. I thought this model might be appropriate because it's been trained to detect complex patterns in an image. In order to gauge how well the model was working, I split my image and steering angle data into a training and validation set. To ensure that model isn't overfitting, I included in dropout layers.
Then I gradually increased the complexity of the model making it similar to architecture published by autonomous driving team at Nvidia. With this architecture I could clearly observe the improvement in the performance of the model.
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 went off the track. To improve the driving behavior in these cases, I iterated with the correction factor which I used to deduce left & right steering angles.
At the end of the process, the vehicle is able to drive autonomously around the track without leaving the road.

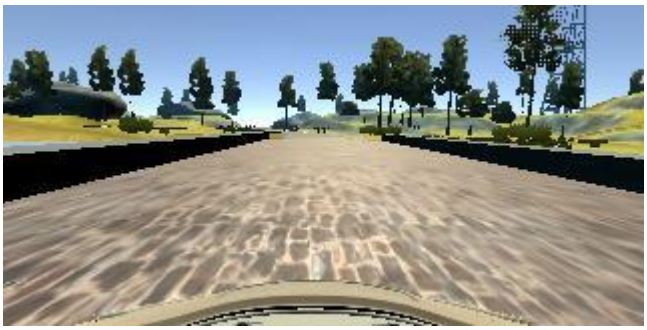
2. Final Model Architecture

The final model architecture (model.py lines 47-64) consisted of a normalization layer, followed by 5 convolutional layers, followed by 4 fully connected layers
Here is a visualization of the architecture.

Layer (type)	Output Shape	Param #	Connected to
lambda_1 (Lambda)	(None, 160, 320, 3)	0	lambda_input_1[0][0]
cropping2d_1 (Cropping2D)	(None, 65, 320, 3)	0	lambda_1[0][0]
convolution2d_1 (Convolution2D)	(None, 31, 158, 24)	1824	cropping2d_1[0][0]
convolution2d_2 (Convolution2D)	(None, 14, 77, 36)	21636	convolution2d_1[0][0]
convolution2d_3 (Convolution2D)	(None, 5, 37, 48)	43248	convolution2d_2[0][0]
convolution2d_4 (Convolution2D)	(None, 3, 35, 64)	27712	convolution2d_3[0][0]
dropout_1 (Dropout)	(None, 3, 35, 64)	0	convolution2d_4[0][0]
convolution2d_5 (Convolution2D)	(None, 1, 33, 64)	36928	dropout_1[0][0]
flatten_1 (Flatten)	(None, 2112)	0	convolution2d_5[0][0]
dense_1 (Dense)	(None, 100)	211300	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]

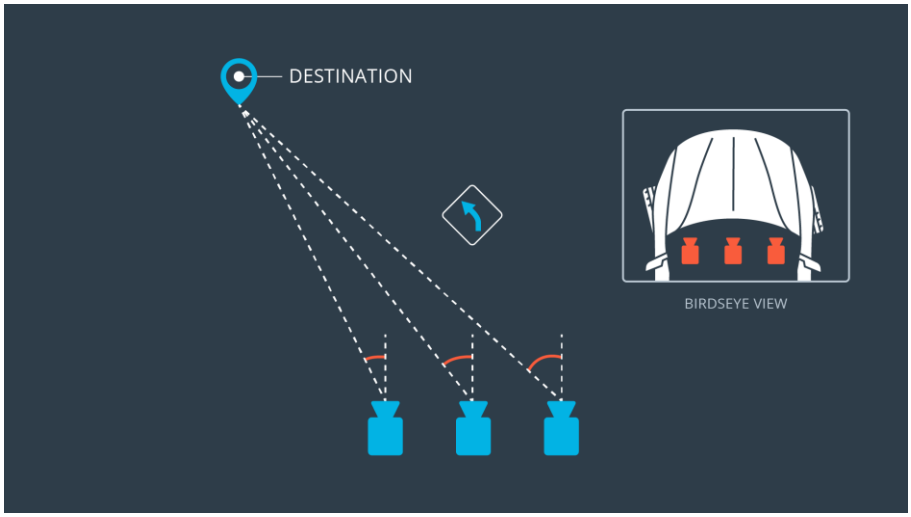
3. Creation of the Training Set & Training Process

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



To augment the data, I also captured the image while driving it in clockwise direction.
The simulator captures images from three cameras mounted on the car: a center, right and left camera. That's because of the issue of recovering from being off-center. This is how it's captured in real car too. For example, if you train the model to associate a given image from the center camera with a left turn, then you could also train the model to associate the corresponding image from the left camera with a

somewhat softer left turn. And you could train the model to associate the corresponding image from the right camera with an even harder left turn.



The image above gives a sense for how multiple cameras are used to train a self-driving car. This image shows a bird's-eye perspective of the car. The driver is moving forward but wants to turn towards a destination on the left.
From the perspective of the left camera, the steering angle would be less than the steering angle from the center camera. From the right camera's perspective, the steering angle would be larger than the angle from the center camera

In order to feed the camera images from left & right camera, I included a correction factor of 0.18 to adjust the steering angle which is captured for the center image.

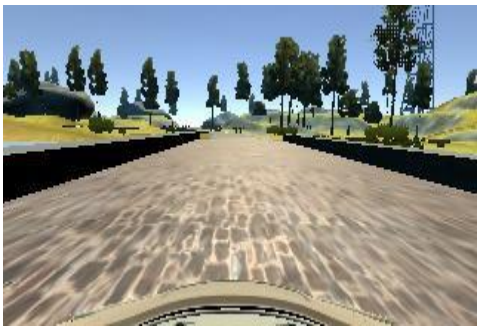
```
steering_left = steering_center + correction
steering_right = steering_center - correction
```

I specifically wanted to include in the left & right camera images so that I can teach the model how to steer if the car drifts off to the left or the right.

After the collection process, I had 8551 number of data points. This number increased by 3x after including in left & right camera images.
Further **preprocessing** steps are elaborated below:
1. **Normalizing the data** - For normalization, I added the lambda layer to my model. Within this lambda layer, I normalized the image by dividing each element by 255 which is the maximum value of the image pixel
2. **Mean centering the data** - Once the image is normalized to arrange between 0 & 1, I mean centered the image by subtracting 0.5 from each element which will shift the element mean down from 0.5 to 0.
3. **Cropping the data** - The image size is 160 pixel by 320 pixel images. Not all of these pixels contain useful information, however. In the image above, the top portion of the image captures trees and hills and sky, and the bottom portion of the image captures the hood of the car. Model will train faster if we crop each image to focus on only the portion of the image that is useful for predicting a steering angle. I used built in keras layers to perform the cropping inside of the model. In this case, I used cropping2D layer to remove the top 70 pixels and bottom 25 pixels.

Here is an example of an input image and its cropped version after passing through a Cropping2D layer:

Original image **Cropped image**



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 under fitting. The ideal number of epochs was 8 as validation error increased after this point. I used an adam optimizer so that manually training the learning rate wasn't necessary.