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

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.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 approach consists of two different model to finish the job. The first model is using LeNet and the second model is referring the structure by NVidia.

Both models consist of a convolution neural network with 5x5 / 3x3 filters sizes and depths between 24 and 120 (model.py lines 74-94)

The model includes RELU layers to introduce nonlinearity (code line 75, 77, 85-88), and the data is normalized in the model using a Keras lambda layer (code line 66).

2. Attempts to reduce overfitting in the model
The model limits the epochs for the training to reduce overfitting (model.py lines 99). Also, the model added dropout layers to prevent overfitting (line 91/94).

The model was trained and validated on different data sets to ensure that the model was not overfitting (code line 100). 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 98). The losses that Adam optimizer aims to reduce is mean square error.

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. When the car is driving in the center, it will mostly face the trained images scenarios from the center camera. When the car is closer to the two sides, the car mostly faces the trained images scenarios taken from the two sides cameras.

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

Model Architecture and Training Strategy
1. Solution Design Approach
My first step was to use a convolution neural network model like the LeNet I thought this model might be appropriate because it has good infrastructure of identifying images and it did a good job in our previous traffic sign recognition project.

In most part of the track this trained model by LeNet works fine, but there is one exception. When the track has no right-side road mark, the car is driving confusingly. The whole track contains two parts that has no right-side road edge, the first one shows up right after the car get off the bridge and the second one shows up right before the end of the track.

This may be caused by the addition of two max pooling layers in the LeNet model, we may lose two much information there. Another possibility maybe overfitting as well. So, I moved to the Network built by NVidia and had some modification there.

The overall strategy for deriving a model architecture was to using convolution network to extract different layers of images out of each training image. With multiple convolutional layers, features of edges, lines, shapes even actual item can be identified into different feature labels.

To gauge how well the model was working, I split my image and steering angle data into a training and validation set. I found that my first model had a low mean squared error on the training set but a high mean squared error on the validation set. This implied that the model was overfitting.

To combat the overfitting, the model limits the epochs for the. The model added dropout layers to prevent overfitting. Also, the model was trained and validated on different data sets to ensure that the model was not overfitting (code line 99).

At the end of the process, the vehicle can drive autonomously around the track without leaving the road.

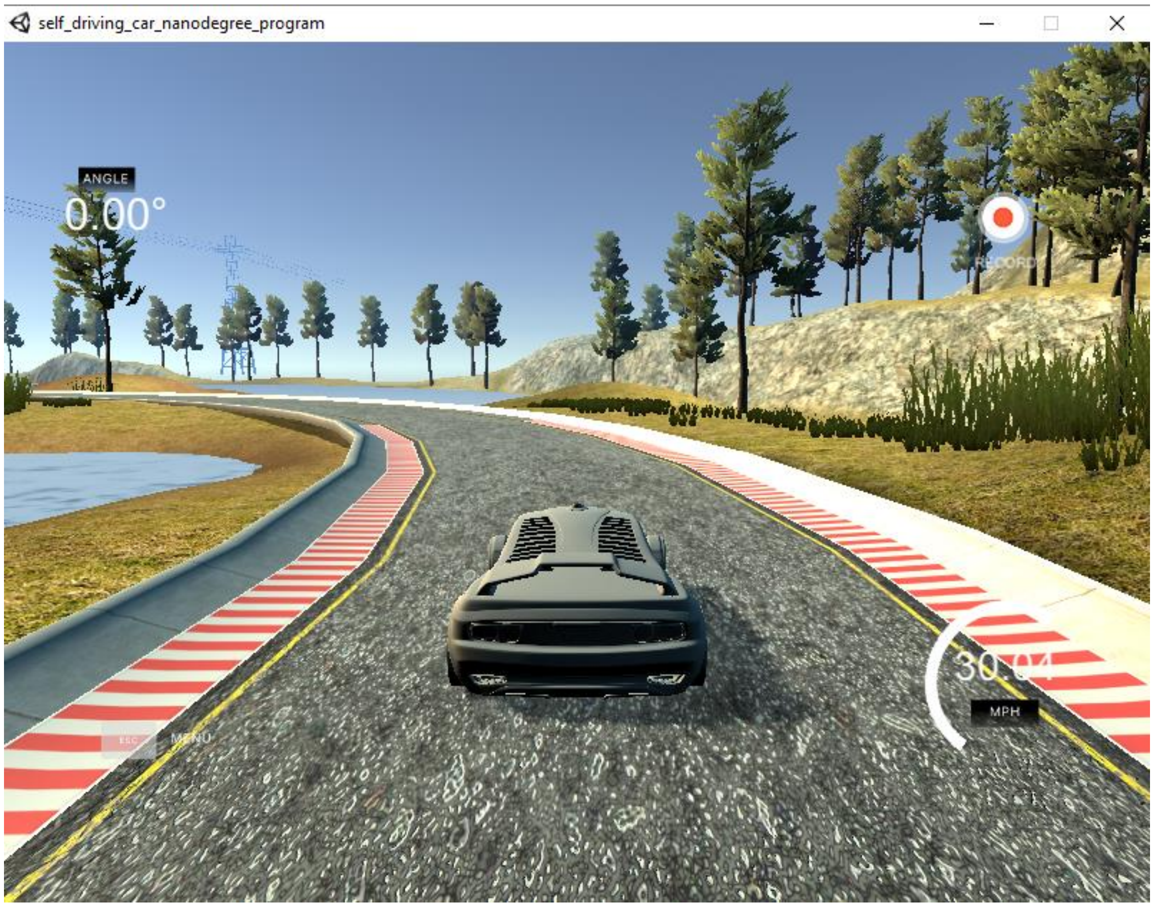
2. Final Model Architecture

The final model architecture (model.py lines 64-94, with model_option=2) consisted of a convolution neural network with the following layers and layer sizes:

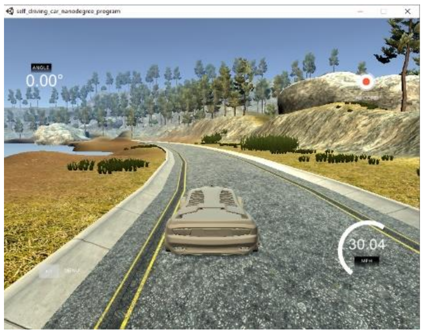
- Convolution Layer: filter(5x5), stride(2x2), 3 features to 24, with RELU activation
- Convolution Layer: filter(5x5), stride(2x2), 24 features to 36, with RELU activation
- Convolution Layer: filter(5x5), stride(2x2), 36 features to 48, with RELU activation
- Convolution Layer: filter(3x3), stride(2x2), 48 features to 64, with RELU activation
- Flatten Layer
- Dropout Layer with dropping rate of 10%
- Dense Layer, reduce feature labels to 100
- Dense Layer, reduce feature labels to 50
- Dropout Layer with dropping rate of 10%
- Dense Layer, reduce feature labels to 10
- Dense Layer, reduce feature labels to 1 (steering wheel angle)

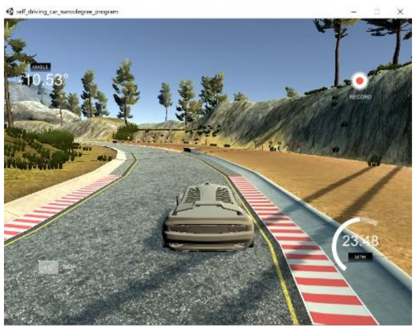
3. Creation of the Training Set & Training Process

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:



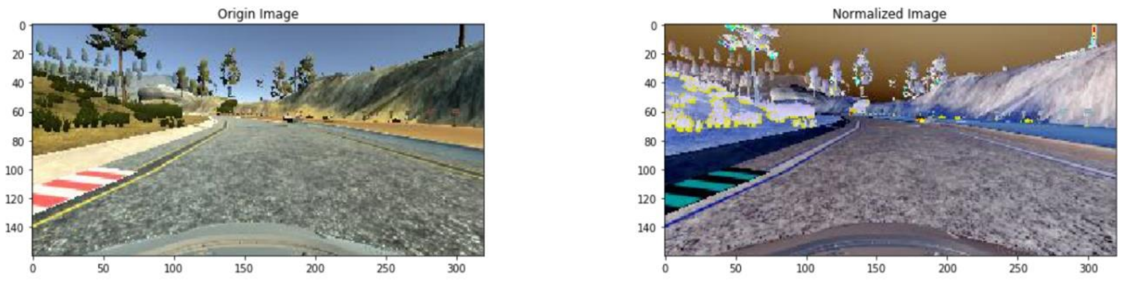
I then recorded the vehicle recovering from the left side and right sides of the road back to center so that the vehicle would learn to recover the position in the center of the track. These images show what a recovery looks like starting from left side and right side:





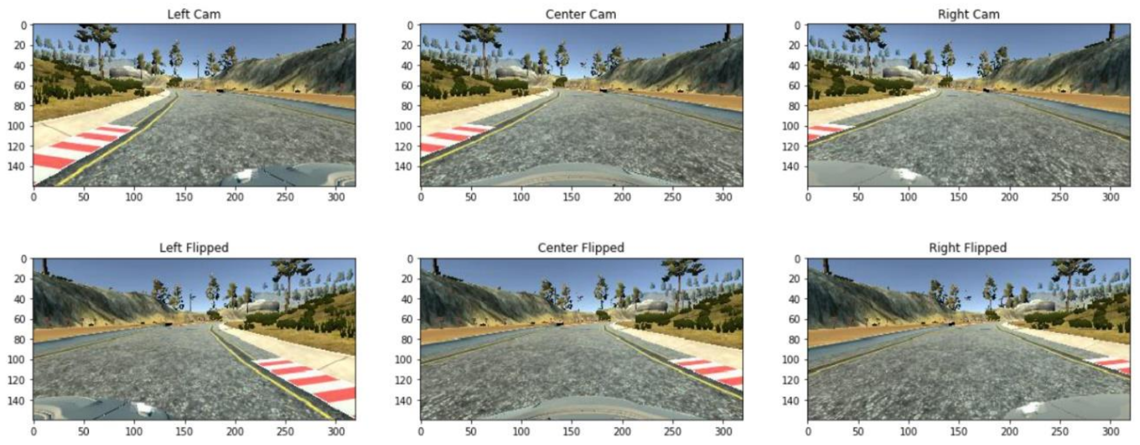
Then I repeated this process on track two to get more data points.

Before processing the training model, I added normalization to help normalize the data set, like in the following image.



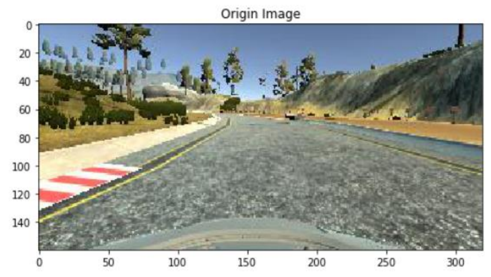
To augment the data set, I also flipped images and angles thinking that this would compensate the situations that the car needs turn right, because in this track, the car is turning left mostly.

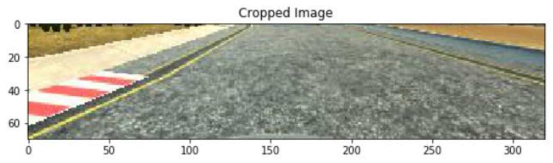
Also, the car has three front cameras, using all the three cameras can help our car handle more scenarios like driving too close to the edges. In the following figure, three images from different cameras along with their flipped images with be shown in the following figure.



It is worth noting that a correction factor must be added on top of the measurement data from the left and the right camera, because when we talk about steering wheel angle, it is respect to the center vison only. So, for the left/right camera image I applied 0.2/-0.2 as the correction factor from try and error. This factor can be calculated from a bunch of distances for example the camera position and car axis length, which is not covered in this project.

Before putting my data into the network model, I did one more step – cropping. Because around 1/3 from the top of each image is having trees, mountains, etc, which will harm instead of help the training since that info are useless. So, I cropped the top 70 pixels of each images and bottom 20 pixels for the similar reason.





Therefore, after the collection of 2 tracks data, I had 15534 images of data, after applying two more cameras data, I had 46604 images of data, and after applying flipping, I finally had 93204 images of data.

I finally randomly shuffled the data set and put 20% of the data into a validation set, so every time there will be 18641 images of data forming the training 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 2, because if too many epochs are applied the network seems to be trained overfitting. I used an Adam optimizer so that manually training the learning rate wasn't necessary.

For the complete self-driving video please check the video file “run1.mp4”.