Behavioral Cloning

Writeup Template

You can use this file as a template for your writeup if you want to submit it as a markdown file, but feel free to use some other method and submit a pdf if you prefer.

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

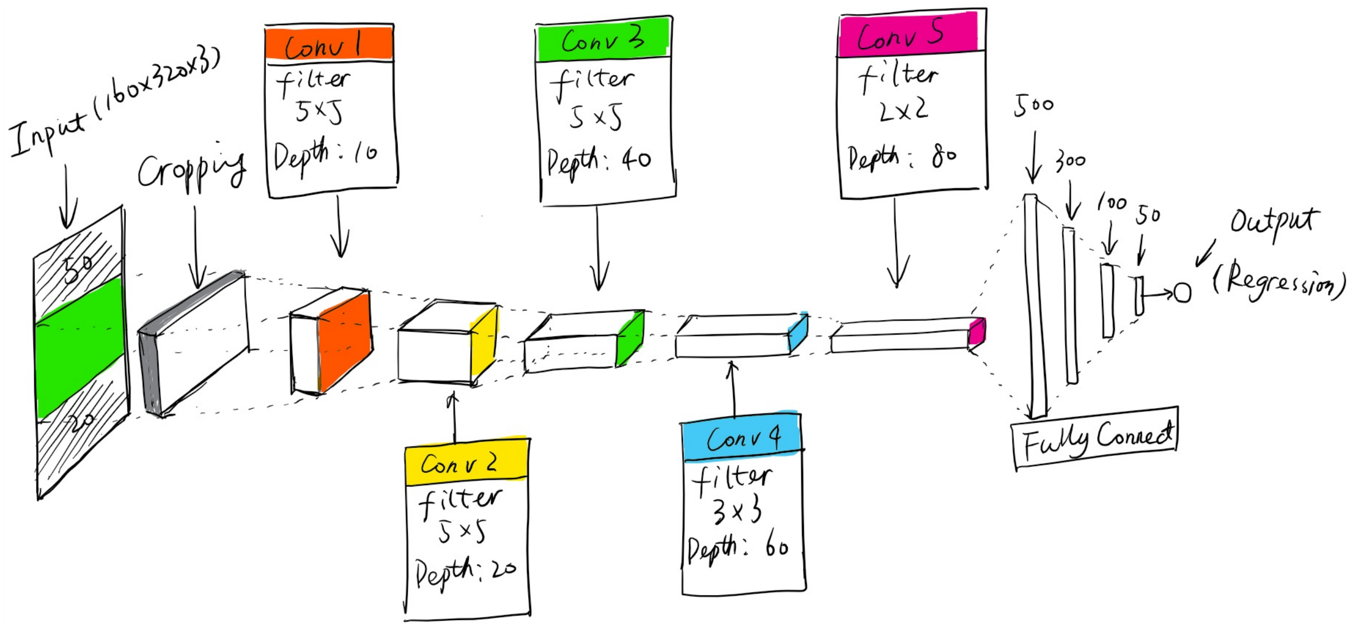
```

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



As the figure shows, my model consists of a series of convolution neural network with filter sizes ranges from 2x2 to 5x5 and depths between 10 and 80 (model_1.py lines 78-86)

To wipe off the useless information in the original images, a cropping layer was introduced right behind the input layer. It cuts out the top and bottom regions of the input images, in which the roads and lanes left.

After cropping, the data was then normalized using a Keras lambda layer(code line 77). The model also includes RELU activation method integrated in every convolutional layer to introduce nonlinearity (code line 78,80,82,84,86).

2. Attempts to reduce overfitting in the model

Since the training data was sufficient in this experiment, so the model tends to be underfitting rather than overfitting. The current model didn't take the dropout layer into consideration. Besides, the errors have showed that both the training error and validation error decreased with epochs, which means there was no evidence of overfitting.

3. Model parameter tuning

The model used an adam optimizer, so the learning rate was not tuned manually (model.py line 95).

4. Appropriate training data

Training data was chosen to keep the vehicle driving on the road. I used a combination of the following methods:

- Drive clockwise and counter clockwise
- Collect driving data of both senarios.(the lake and the forest)
- Record the process of recovering driving from off road to the main road in different situations to teach the autonomous car to stay on the road.

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 find a structure most suitable for autonomous driving.

I builded several neural networks of different types, different architectures in Keras. Everytime I've finished one network, I put it in the simulator to test the performances.

- My first attempt is a simple fully connected neural network(Dense layer in Keras), whidh has 160x322x3 neurons in the input layer, one hidden layer and a single output node. This network terribely works in the autonomous mode since this type of structure can not grasp the 2-D characteristics in an image.
- Then I tried LeNet structure but with low depth convolutional layers. There are only 2 convolutional layers in my LeNet-like network. The depth of filters are 8 and 20. In simulation, it drives better than my first network only has Dense layers. But there still exists so many drawbacks, for instance, the car will be thrown out of the track. This usually happens on sharp corners.
- To improve the driving performance on corner, I use a convolution neural network model similar to the one proposed in NVIDIA's article. I thought this model might be appropriate because it has been verified on a real autonomous vehicle. The convolutional layer was deep enough to grasp the characteristics shown in the images. I think I've got the last shot, it works very well.

In order 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.

The final step was to run the simulator to see how well the car was driving around track one. Perfect! The vehicle is able to drive autonomously around the track without leaving the road. The video clip can be found in my submission.

During simulation, I found a tiny bug. The velocity of my vehicle, can not get steady, but has a large amount of fluctuation. To further investigate this, I checkout the drive.py file and found that the speed control is implemented by a PI controller. The speed setpoint is 9mph. So, that indicates that there is no connection between the network and the vehicle speed. If I want to fix this, I have to modify the PI controller's parameters. After trying to alter different combinations of Kp and Ki, I found it's hard to tune them without knowing the characteristics of the plant.

So, before the tuning, we should first know the mathmatical model of our plant, the vehicle. In this project, the MATLAB and Simulink software was introduced to do this task.

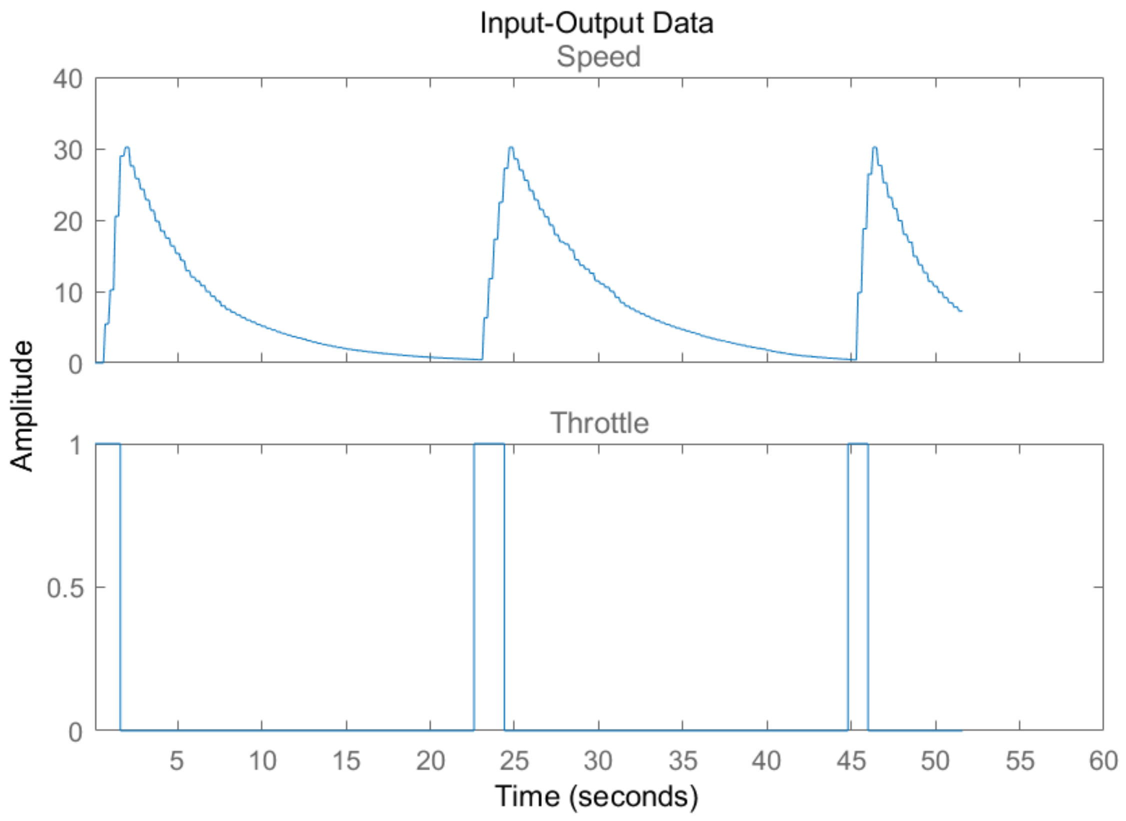
- First, the 'drive.py' file should be modified to record useful data and send my test instructions. The signals we used here are 'throttle' as the input and 'speed' as the output.

```
        # record simulation data to csv file
        if args.datalog_folder != '':
            with open('../logData/logdata.csv','w',newline='') as csvfile:
                writer = csv.writer(csvfile, delimiter=',')
                for i in range(len(speed_array)):
                    writer.writerow([speed_array[i], throttle_array[i]])
```

- To make the identification easier, I bypassed the original throttle PI control logics in "drive.py", use a 'step' like signal instead. The code was show below:

```
        if float(speed) > 25:
            throttle = 0
        elif float(speed) < 0.5:
            throttle = 1
        else:
            throttle = throttle
```

- The recorded input-output figure was shown like this:



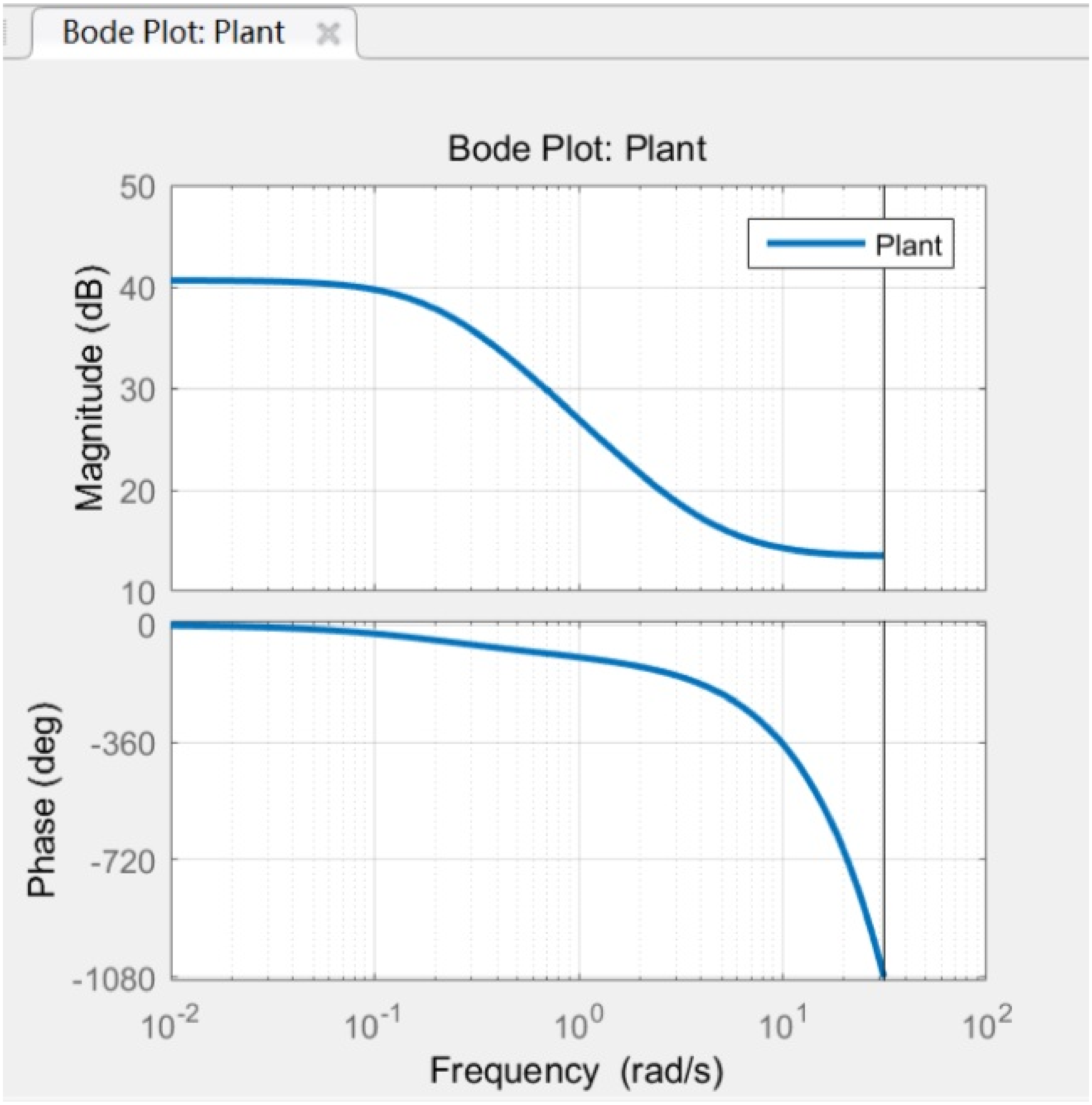
- Then, I employed the 'System Identification Toolbox'(a product of MATLAB) to identify the transfer function model that describes the vehicle's longitudinal dynamics. The following MATLAB code used recorded speed and throttle data to derive the transfer function of our plant,

```
data = iddata(speed, throttle, Ts)
sys = tfest(data, 1, 0, nan)
```

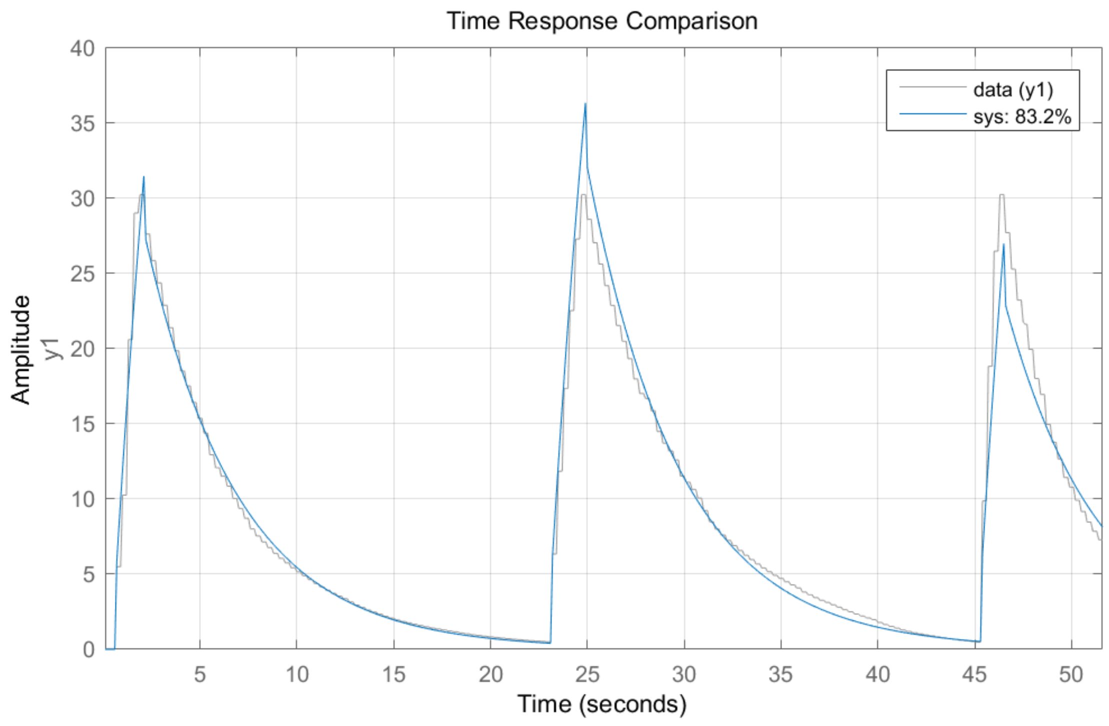
- The transfer function of the vehicle is shown below. It acts like a first order inertial system with a delay of 0.5 seconds.

$$G(s) = e^{-0.5s} \frac{23.12}{s + 0.2235}$$

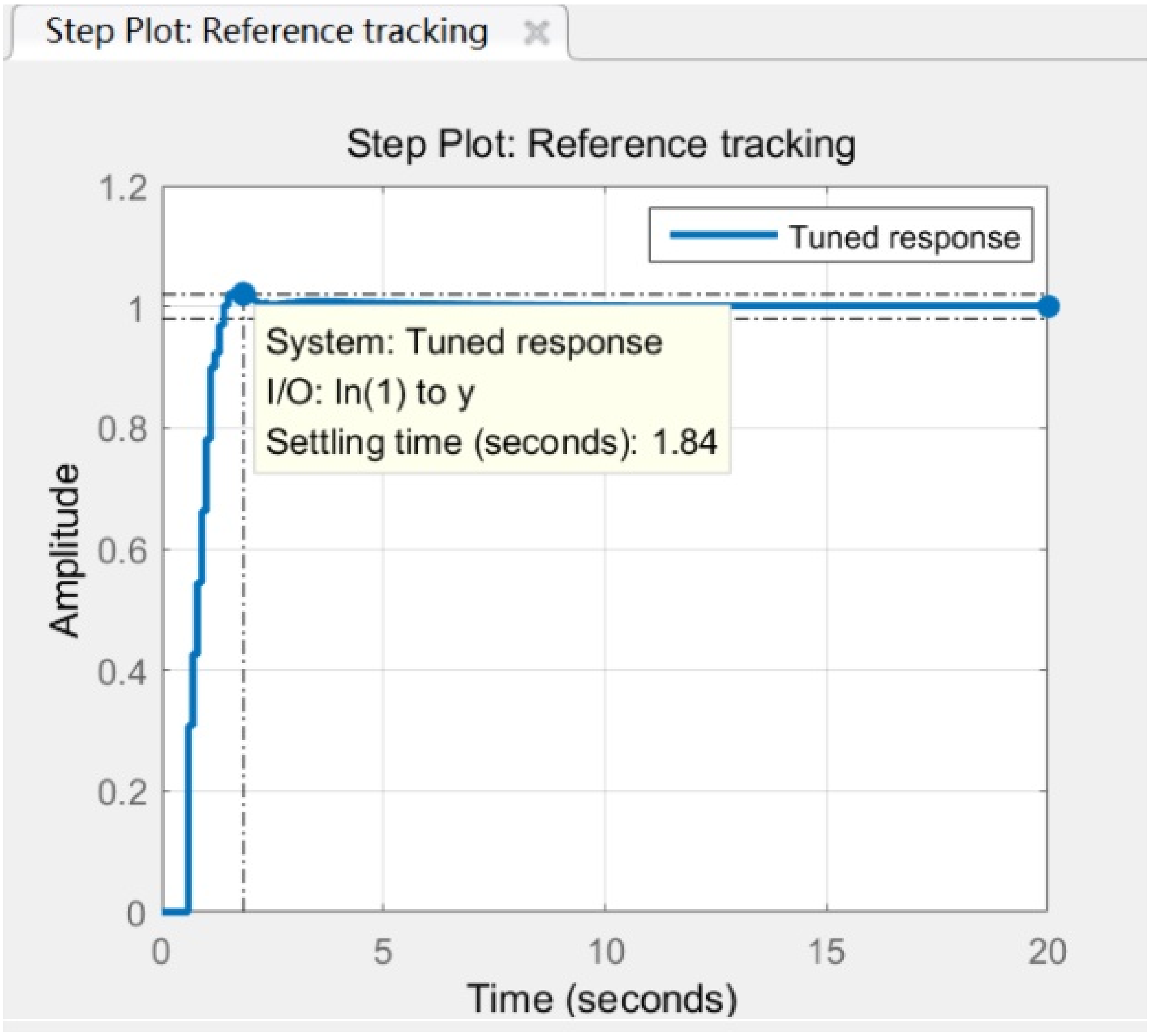
- The bode plot of the plant also can be derived.



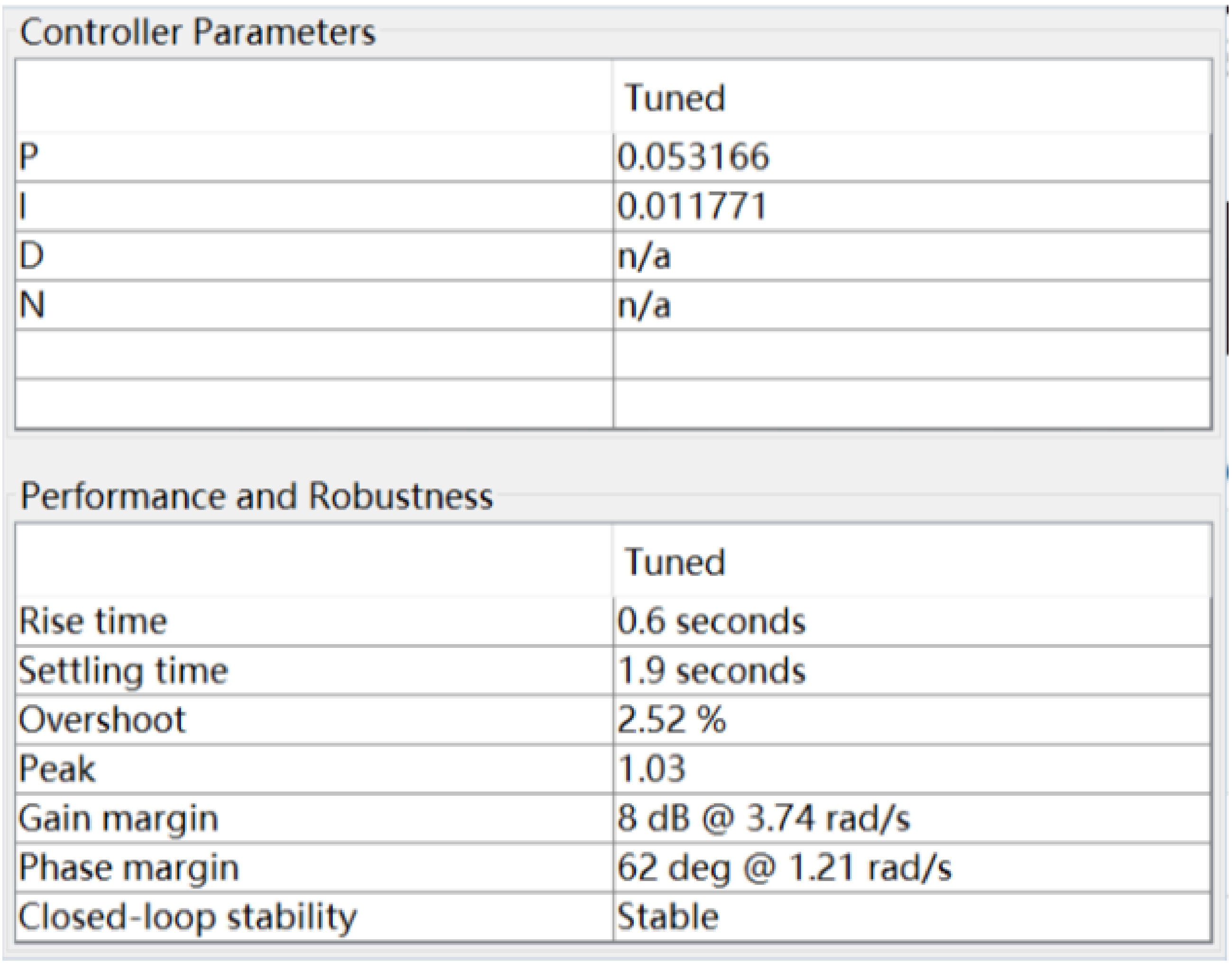
- The similarity of identified model and the recorded data was shown in the following plot. Fits well.



- So I use this plant model to design my PI controller in the Simulink PID tuning block. After fine tuning the Kp and Ki, the closed loop step response looks like:



- The followng figure shows the final PID parameters and characteristics of our tuned system.



Controller Parameters

	Tuned
P	0.053166
I	0.011771
D	n/a
N	n/a

Performance and Robustness

	Tuned
Rise time	0.6 seconds
Settling time	1.9 seconds
Overshoot	2.52 %
Peak	1.03
Gain margin	8 dB @ 3.74 rad/s
Phase margin	62 deg @ 1.21 rad/s
Closed-loop stability	Stable

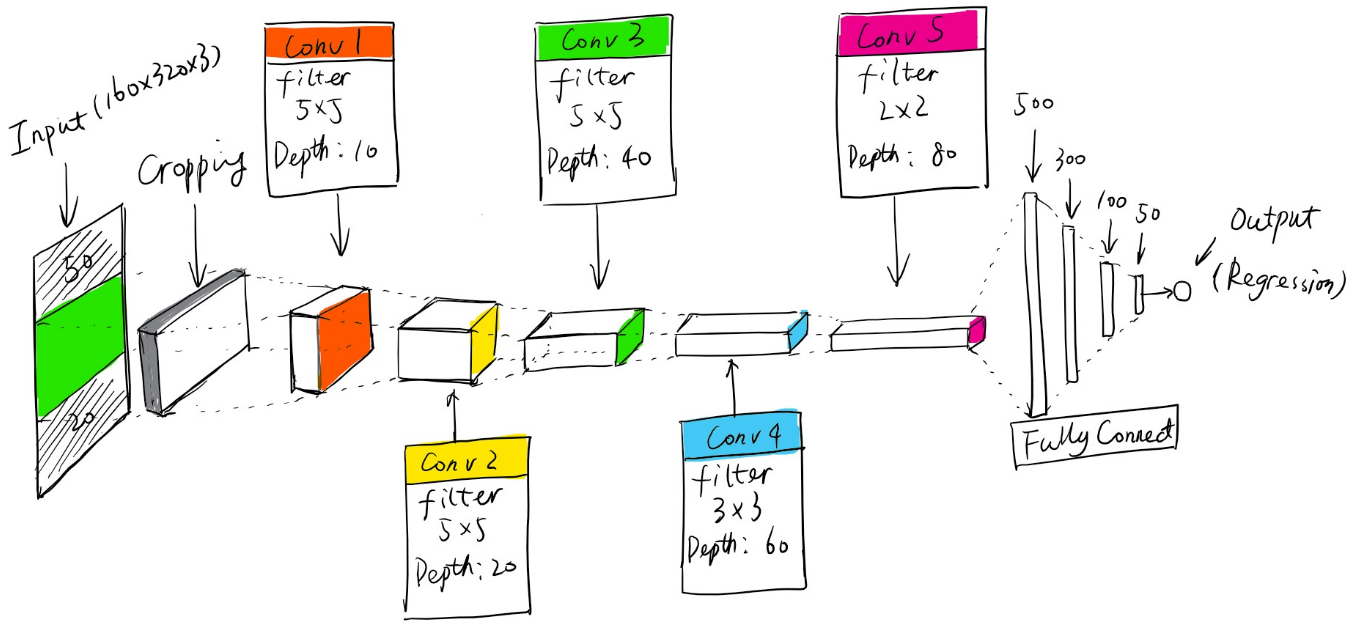
- Adopting the parameters of PI controller, the final code was shown in the current 'drive.py' in my submission. After running in the simulator, the speed control performs perfect, vehicle speed smoothly reached the target speed of 15 mph within 2 seconds and almost no overshooting. This indicating that the identified model fits well!
- **Problem sovled :)**

2. Final Model Architecture

The final model architecture (model.py lines 18-24) consisted of a convolution neural network with the following layers and layer sizes.

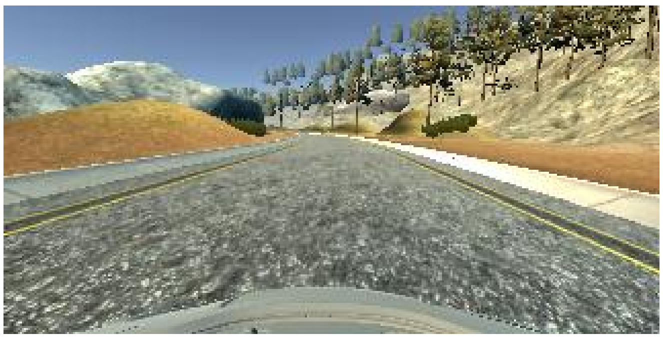
```
model.add(Convolution2D(10,5,5, activation='relu'))
model.add(MaxPooling2D())
model.add(Convolution2D(20,5,5, activation='relu'))
model.add(MaxPooling2D())
model.add(Convolution2D(40,5,5, activation='relu'))
model.add(MaxPooling2D())
model.add(Convolution2D(60,3,3, activation='relu'))
model.add(MaxPooling2D())
model.add(Convolution2D(80,2,2, activation='relu'))
```

Here is a visualization of the architecture (note: visualizing the architecture is optional according to the project rubric)



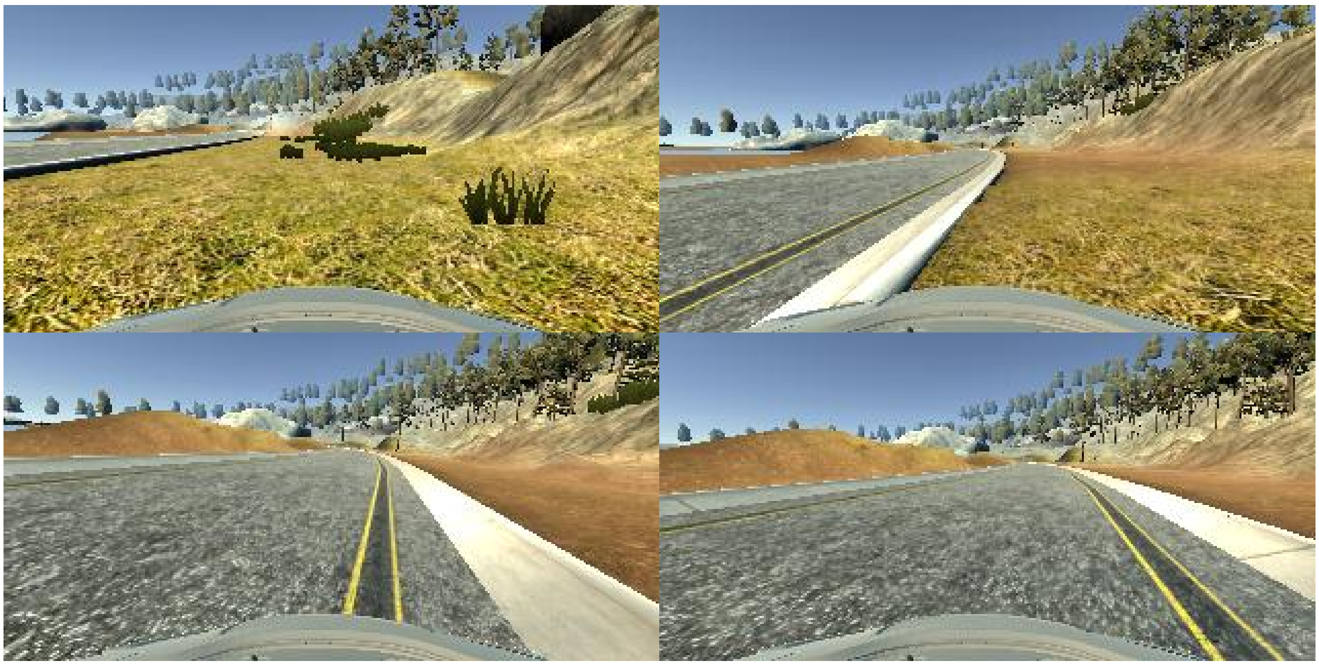
3. Creation of the Training Set & Training Process

To capture good driving behavior, I first recorded two laps on track No.1 using center lane driving along the track(counter clockwise). Here is an example image of center lane driving:



Then, I turned around, heading to the opposite direction(clockwise). This time, I recorded one lap.

After that, I also recorded the vehicle recovering from the left side and right side of the road back to center so that the vehicle would learn to recovery driving when it makes some deviation. These images show what a recovery looks like starting from the right side of the road, and then move back to the center:



I meant to repeat the above 3 steps for the second track to get more data points. But this track is to hard to handle for me, I only do the 1st step, driving on this track for just a lap.

To augment data set, I also flipped all of the images and angles thinking that this would help the model to generalize better.

After the collection process, I had 54638 number of data points. I then tailored the images to remove the useless part of the recorded images while reducing their size. I then preprocessed this data by normalize all the input data between 0 and 1.

I finally randomly shuffled the data set and put 20% of the data (13660 samples) 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 9 as evidenced by both of the training error and validation error are decreasing. I used an adam optimizer so that manually training the learning rate wasn't necessary.