Self-Driving Golf Cart Documentation

Release 0.1.5

Neil Nie

Nov 06, 2020

Contents

Welcome! This is an open source self-driving development platform aimed for rapid prototyping, deep learning and robotics research. The system currently runs on a modified electric golf cart, but the code could work on a real car as well. Here are our goals:

Goals:

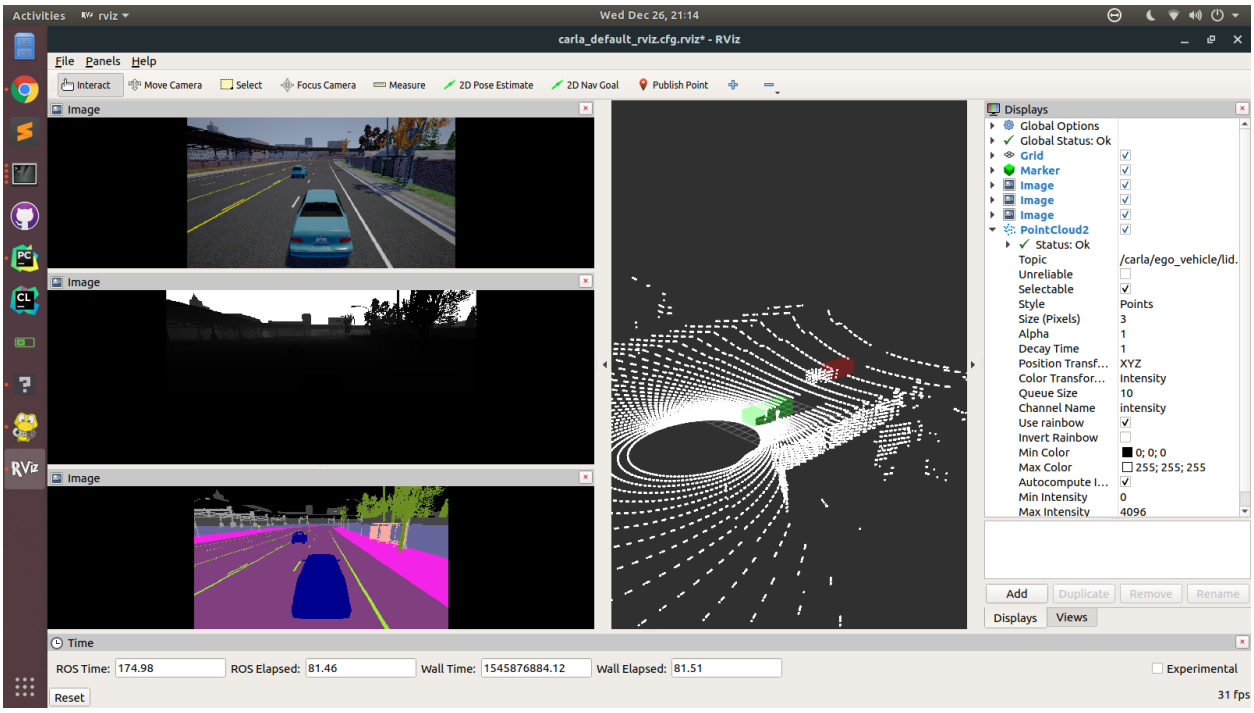
- Research and develop a deep learning-driven self-driving car.
- The vehicle should be able to navigate from point A to point B autonomously within a geofenced area.

CHAPTER 1

Tutorials

1.1 Getting Started

1.2 Running the Simulation



Building a self-driving car is hard. Not everyone has access to expensive hardware. I am currently trying to integrate this project with the CARLA self-driving simulator. If you are interested in CARLA, please refer to the following documentation. (The current ROS system in this project can only partially run on the CARLA simulator)

1.2.1 Build Unreal Engine

Please note that Unreal Engine repositories are private. In order to gain access you need to **add your GitHub username when you sign up at www.unrealengine.com.**

Download and compile Unreal Engine 4.18. Here we will assume you install it at "~/UnrealEngine_4.18", but you can install it anywhere, just replace the path where necessary:

```
$ git clone --depth=1 -b 4.18 https://github.com/EpicGames/UnrealEngine.git ~/
↪UnrealEngine_4.18
$ cd ~/UnrealEngine_4.18
$ ./Setup.sh && ./GenerateProjectFiles.sh && make
```

Check Unreal's documentation "Building On Linux" if any of the steps above fail.

1.2.2 Installing Carla

To install Carla the simulator is simple. Just head over to their releases page on Github and download the latest pre-built release. At the time of writing, the latest release can be found here: https://drive.google.com/open?id=1JprRbFf6UlvpqX98hQiUG9U4W_E-keiv

1.2.3 Running the Simulation

1. Setting Up:

1.1. Install CARLA

Download the compiled version of the CARLA simulator from here

Please refer to the CARLA documentation or open an issue if you have any questions or problems.

1.2. Install the CARLA Python API

At this point, you should have downloaded the compiled version of the CARLA simulator.

```
$ sudo easy_install <path/to/carla/>/PythonAPI/<your_egg_file>
```

Just as an example, for me, the command is this:

```
$ sudo easy_install '/home/neil/carla/PythonAPI/carla-0.9.2-py2.7-linux-x86_64.egg'
```

Please note that you have to put in the complete path to the egg-file including the egg-file itself. Please use the one, that is supported by your Python version. Depending on the type of CARLA (pre-build, or build from source), the egg files are typically located either directly in the PythonAPI folder or in PythonAPI/dist.

Check the installation is successfull by trying to import carla from python:

```
$ python -c 'import carla;print("Success")'
```

You should see the Success message without any errors.

1.3. Install other requirements:

```
$ sudo apt-get install python-protobuf
$ pip install --user simple-pid
```

If you have followed the instructions closely, you should be ready to use CARLA and the simulation_package

2. Running

2.1. Compile the project

Navigate to the ros directory of the self-driving golf cart project:

```
$ cd [PROJECT_DIRECTORY]/ros
```

Then enter & run the commands:

```
$ catkin_make
$ source devel/setup.bash
```

2.2. Launch CARLA

First run the simulator. For the full and latest documentary, please always refer to the carla official website. (see carla documentation: http://carla.readthedocs.io/en/latest/). I have created a simple script. Enter the following command and run the script:

```
$ rosrun simulation launch_carla.sh
```

Wait for the message:

```
$ Waiting for the client to connect...
```

2.3. Start the ros bridge & rviz

```
$ roslaunch simulation carla_client_with_rviz.launch
```

You should see a new rviz window. You can setup the vehicle configuration config/settings.yaml.

This launch file also make use of the CARLA Python API script `manual_control.py`. This spawns a vehicle with role_name='hero' which is interpreted as the ego vehicle as defined by the `config/settings.yaml`.

The launch file also **further spawn 30 other vehicles** using `spawn_npc.py` from CARLA Python API. Then those vehicles will show up also on ROS side.

CHAPTER 2

ROS

2.1 About ROS

Below you will find information about all the ROS packages, nodes, topics used in this project.

2.1.1 Terminology

ROS (robot operating system): a collection of software frameworks for robot software development. It provides services designed for hardware abstraction, low-level device control, implementation of commonly used functionality, message-passing between processes, and package management.

ROS Nodes a process that performs computations. Nodes are combined together into a graph and communicate with one another using streaming topics, RPC services, and the Parameter Server.

2.1.2 Packages & Nodes

Here is a list of packages. Underneath each package are nodes in that package.

simulation

The major purpose of the simulation package is to connect our self-driving system to CARLA simulator. To run the package, please refer to the documentation [here](./src/simulation/README.md).

The simulation package can also run simulated camera inputs using the camera_sim_node

Nodes:

```
$ carla_client
$ camera_sim_node
```

Launch Files:

```
$ carla_client.launch
$ carla_client_with_rviz.launch
$ carla_client_with_rqt.launch
$ start_camera_sim.launch
```

autopilot

The autopilot package is the brain of the self-driving car. It uses end-to-end deep learning to predict the steering, acceleration and braking commands of the vehicle. while subscribes to the camera feed. (Node currently functioning) The Arduino subsribes to the steering_cmds and controls the steering accordingly.

Nodes:

```
$ autopilot
$ visualization
```

Publishes (the autopilot node):

```
$ /vehicle/dbw/steering_cmds/
$ /vehicle/dbw/cruise_cmds/
```

Subscribes (all nodes):

```
$ /cv_camera_node/image_raw
$ /cv_camera_node/image_sim
```

object_detection

YOLO (You Only Look Once) realtime object detection system.

Nodes:

```
$ object_detection_node
```

Publishes:

```
$ /detection/object/detection_visualization/
$ /detection/object/detection_result
```

Subscribes:

```
$ /cv_camera_node/image_raw
```

segmentation

Semantic segmentation node. Deep learning, ConvNets

Nodes:

$ segmentation_node

Publishes:

```
$ /segmentation/visualization/
$ /segmentation/output
```

Subscribes:

```
$ /cv_camera_node/image_raw
```

cv_camera

The cameras are the main sensors of the self-driving car.

Nodes:

```
$ cv_camera_node
```

Publishes:

```
$ /cv_camera_node/image_raw
```

driver

This is the main package of the project. It pulls together all the individual nodes to create a complete self-driving system.

**Nodes: $ drive

gps

Used for localization. Currently using the Adafruit GPS module, serial communication.

Nodes:: $ gps_receiver $ nmea_topic_driver $ nmea_topic_serial_reader

The GPS package manages and publishes the data received from a GPS module connected via serial. The package

Publishes:

```
$ /sensor/gps/fix
$ /sensor/gps/vel
```

osm_cartography

Nodes:

```
$ osm_client
$ osm_server
$ viz_osm
```

This package broadcasts and processes .osm files. OSM files are OpenStreetMap files which contain detailed information about the environment, such as coordinates of roads, building and landmarks. Currently, the main function of the package is to broadcast the osm info to rviz for visualization. (Node currently functioning)

2.1.3 ROS Topics for visualization

```
$ /visual/steering/angle_img
$ /visual/detection/object/bbox_img
$ /visual/detection/lane/marking_img
$ /visual/segmentation/seg_img
```

CHAPTER 3

Modules

3.1 End-to-End Steering

3.1.1 Introduction

In 1989, ALVINN, the self-driving car (truck) made by Dr. Dean Pomerleau and his team, drove around the Carnegie Mellon campus. According to Pomerleau, The vehicle was powered by a CPU slower than the Apple Watch. The car used a fully connected neural network to predict the steering angle of the car in real time. Fast forward twenty years, NVIDIA proposed a novel method that combines Pomerleau's idea with the modern GPU, giving NVIDIA's car the capability to accurately perform real-time end to end steering prediction. Around the same time, Udacity held a challenge that asked researchers to create the best end to end steering prediction model. This project is deeply inspired by that competition, and the goal is to further the work in behavioral cloning for self-driving vehicles.

3.1.2 How to Run Inferencing

1. Clone/download this repository

2. Download the pre-trained weights here: https://drive.google.com/file/d/19DR2fIR6yl_DdqQzPrGrcvbp_MxXC0Pa/view?usp=sharing. 3.:

```
$ cd <YOUR REPO DIRECTORY>
```

Then in your own Python program or Python console:

```
$ from steering_predictor import SteeringPredictor
$ predictor = SteeringPredictor('<SAVED MODEL PATH>', '<MODEL TYPE>')
$ steering_pred = predictor.predict_steering_angle(<IMAGE>)
```

Please note that the possible model types are *rgb* and *flow*. The input image must be an RGB image of any size. For more information, please refer to the code comments and documentations.

3.1.3 I3D Model Architecture

Motives

NVIDIA's paper used a convolutional neural network with a single frame input. I believe that, even though this architecture yielded good results, the single frame CNN doesn't provide any temporal information which is critical in self-driving. This is the motive behind choosing the i3d architecture, which is rich in spacial-temporal information.

Model

The input of the network is a 3d convolutional block, with the shape of *n * weight * height * 3*. *n* is the length of the input sequence. Furthermore, the network also uses nine inception modules. The output layers are modified to accommodate for this regression problem. A flatten layer and a dense layer are added to the back of the network.

Results

During training, RMSE (root mean squared error) is used for the loss function. Udacity's open source driving dataset is used for training and testing. RMSE is also the benchmark for the validation results. For the results of the Udacity challenge, please click [here](https://github.com/udacity/self-driving-car/tree/master/challenges/challenge-2).

Model Type | Parameters | Training Loss| Validation Score|
———————— |:————:| ————:| ————:|
Single Frame ConvNet | – million | — | 0.1320 |
i3d 32 Frame RGB | 12.2 million | 0.0299 | 0.0862 |
i3d 64 Frame RGB | 12.2 million | 0.0430 | **0.0530** |

The i3d model architecture proved that spacial-temporal information could drastically improve the performance of the behavioral cloning system. After fewer than 80K steps of training, the network's validation loss scored half of the validation score of the single frame CNN.

Performance

This good performance comes at a cost. On the one hand, the 32 frames i3d network's inference time on a GTX 1080 is 0.07 seconds, making the realtime frame rate ~15fps. On the other hand, the 64 frames network's inference time is 0.13 seconds, which makes the frame-rate ~7fps. One of the best ways to improve performance is to improve the hardware of the system. A fast multicore CPU with hyper-threading would drastically improve the inference speed.

3.2 Semantic Segmentation

3.3 Drive by Wire

CHAPTER 4

Information

4.1 License

4.2 Contact

If you have any questions, comments or concerns about the code, please email me at contact@neilnie.com.