TRAFFIC SIGN DETECTION IN GAZEBO SIMULATION ENVIRONMENT USING YOLOV3

A Dissertation
Presented to
The Academic Faculty

by

Nikhil Prabhu

In Partial Fulfillment
of the Requirements for the Degree
MASTER OF SCIENCE IN ENGINEERING in the
KETTERING UNIVERSITY

Kettering University
December 2019

TRAFFIC SIGN DETECTION IN GAZEBO SIMULATION ENVIRONMENT USING YOLOV3

Approved by:

Dr. Jaerock Kwon, Advisor
Department of Electrical and Computer Engineering
Kettering University

Dr. Girma Tewolde
Department of Electrical and Computer Engineering
Kettering University

Dr. Diane Peters
Department of Mechanical Engineering
Kettering University

Date Approved: December 18, 2019

ACKNOWLEDGEMENTS

This thesis represents the capstone of my two years of combined academic work at Kettering University and my experience. My culminating graduate experience provided the opportunity for me to use the knowledge gathered and skill set acquired while at Kettering to complete a thesis of this magnitude.

Although this thesis represents the compilation of my efforts, I would like to acknowledge and extend my sincere gratitude to the following people for their valuable time and assistance, for without whom the completion of this thesis would not have been possible:

1. Firstly, I thank God for giving me the strength to get through all the difficulties I have experienced.
2. A special thanks to my family. Words cannot express how grateful I am to my most significant influencer “My Idol” my Mother. I cannot Thank you enough for supporting me with everything, It would not have been possible for me to complete my studies without your love, support, and encouragement.
3. A great appreciation to my advisor Dr. Jaerock Kwon, Associate Professor of Computer Engineering and Faculty Thesis Advisor, Kettering University who helped me in every way possible towards the completion of this thesis.
4. Dr.Girma Tewolde, Associate Professor of Computer Engineering, Kettering University.
5. Dr.Bassem Ramadan Department Head and Professor of Mechanical Engineering at Kettering University.
6. A special thanks to Dr. Scott Reeves, Dean of graduate studies, for his help and support during my difficult times.
7. Lastly, I would also like to acknowledge all the members of MIR lab, Ahmed Abdelhamed, Ninad Doshi and Balakrishna Yadav for their valuable time and support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND ABBREVIATIONS

YOLO	You Only Look Once
ROS	Robot Operating System
ADAS	Advanced Driver Assistance System
RADAR	Radio Detection and Ranging
LiDAR	Light Detection and Ranging
CNN	Convolutional Neural Network
CAT	Cognitive and Autonomous Test
SiL	Software in the Loop
HiL	Hardware in the loop
OGRE	Object-oriented Graphics Rendering Engine
SDF	Simulation Description Format
OSRF	Open Source Robotics Foundation
CAD	Computer Aided Design
URDF	Universal Robot Description Format
PID	Proportional-Integral-Derivative
ROI	Region of Interest
TP	True Positive
FP	False Positive
FN	False Negative
mAP	Mean Average Precision
IoU	Intersection Over Union
RMSE	Root Mean Squared Error

ABSTRACT

Research on the self-driving vehicle systems is constantly increasing. The design of an autonomous vehicle system is concentrated on building a robust and efficient framework that has the potential to handle dangerous traffic situations. There are many systems in place that provide driver assistance in various forms such as lane-keeping, collision avoidance, adaptive cruise control, etc. which are attempts to making our roads safer to drive on. This thesis is also an attempt on providing such a framework for autonomous driving system, that is focused on road safety. The aim of this thesis is to contribute to the autonomous vehicle research community by integrating a system capable of detecting and recognizing traffic signs and based on the detection take appropriate decisions for easier and safe driving. The framework for this thesis is developed in an open-source 3D simulator Gazebo. For the traffic sign recognition, I have made use of the You Only Look Once (YOLO) object detection system. Unlike most object detection systems, YOLO uses a sliding window over several locations within an image, which allows it to be extremely fast. These two tools are combined with the help of the Robot Operating System (ROS), which is a widely used middleware for robotics research. I have also made use of a neural network for training the vehicle for lateral motion control which is based on Nvidia's Pilot Net architecture. The framework was developed in the MIR lab at Kettering University. This thesis has been developed such that it is very easy to use by any individual who is interested in the research of autonomous vehicles. I have provided all the necessary requirements, tools, scripts and commands to the smallest detail, which will be needed to successfully build this from scratch. Apart from this, the weights obtained from training the neural networks, along with the labeled dataset used for training have also been provided. The results show that the detection procedure is satisfactory with a precision metric value of 0.98. Utmost care has been taken to have real-time deployment, hence all parameters have been optimized to make the system capable of running with very little computation.

CHAPTER 1. INTRODUCTION

In the recent years, autonomous vehicles have vastly changed from just being a research topic to a goal for every automotive company, advanced driver assistance systems (ADAS) is being a necessity for today's vehicle owners [1]. The possibility of having a system that can monitor your surroundings and keeping track of your safety is a must-have feature. The autonomous and intelligent systems are no more considered a myth but becoming a reality and one of the most vital research areas in the automotive industry these days. Such a system that is capable of entirely navigating autonomously in an environment and being aware of the surroundings without any human interaction is considered a challenging task [2].

The automotive industry has already started developing ADAS features for driver comfort and safety, making substantial progress towards fully autonomous cars. Advancements in technologies such as, Radio detection and ranging (RADAR), Light detection and ranging (LiDAR), cameras and their fusion provide a 360-surround view of the car, have enhanced the vehicle perception in a huge way. The amalgamation of these technologies will ensure extra safety and mobility, which will also play a significant role in providing enhanced mobility for children, elderly, disabled, and provides relief to the drivers during long journeys. In addition to an increase in driving safety of cars on normal busy roads and highways, there has been a lot of increase in driver comfort, thus significantly reducing the number of accidents. Even after all these technological advancements and reaching a high level of autonomy, there still is a great potential to improve driving safety, comfort, and efficiency, before the autonomous vehicles are

available commercially. Today, many huge universities, major companies, and also governments are working with full force to get a fully functional autonomous vehicle. Although the goal seems to be close, to have a proof of concept, there needs to be a minimum of at least 10 billion kilometers of autonomous driving to get to the stage of mass production. Even with 100 vehicles driving 24 hours a day seven days a week, this would still need about 225 years of driving [3]. Hence, a significant need for a simulator, that can speed up the process of validation of the driving technologies and at the same time reduce the costs for the testing is needed.

Research groups in academia and industry have built their own platforms to investigate problems that need to be solved in order to achieve full autonomy. The simulated platforms available today present a powerful solution to the issue of testing for a huge number of miles as now developers are able to safely and accurately test and validate the driving software prior to hardware testing. A simulated test environment is not just a virtual road with a virtual car, but it is a detailed and accurate blueprint of the city roads, highways, buildings, pedestrians, bicyclists, and any other traffic scenario model that a vehicle may encounter. Hence, apart from looking realistic, it also needs to act realistic, meaning, it needs to follow the laws of physics. It also must have the ability to simulate real-world conditions such as wind, weather, and lighting conditions that vary without any proper human predictions. For the virtual car to perform close to the real car, it needs to have sensing capabilities just as good as the real car. Hence, the need for realistic sensors, that can replicate realistic sensor feeds. The closer the sensor model gets to the real sensor, better the results will be obtained for testing the algorithms.

The purpose of this thesis is to develop a framework for object detection and recognition system for autonomous vehicles. All the work towards this thesis is done in the MIR (Mobile Intelligent Robotics) Lab, a research lab located at Kettering University. The software used for creating the framework is called Gazebo which is an open-source simulator, capable of simulating complex robots and environments. Gazebo runs on a robust physics engine and allows us to manipulate the physical properties of the models in the simulation to recreate urban scenarios. It also has the flexibility to set up and modify the sensors to provide signals that can be used to train driving strategies. Robot Operating System (ROS), was used as the backbone for communicating with all the vehicle sensors and the vehicle controller. ROS provides a distributed computing environment by which all communication between components in the system is seamlessly possible. ROS is being used by most of the automotive companies for implementing and testing ADAS features that contribute to building a fully automated vehicle.

This thesis also includes procedures for applications of the algorithms created for autonomous vehicle testing such as traffic sign detection and lateral control with the use of Neural Networks. Over the past decade, there has been significant progress in machine learning techniques such as Convolutional Neural Networks (CNN). With CNN it is easier to train the vehicle for pattern recognition, which enables the deployment of various safety-critical systems that are important for autonomous vehicles. In this thesis, I have used CNN for both traffic sign detection and lateral control. The CNN takes the sensor inputs and provides predicted outputs for signs and steering angles to control the vehicle. A detailed explanation regarding the working and implementation of the CNN inside the created framework is provided in the further sections.

1.1 Related Works

A lot of work is being carried out in the field of autonomous vehicles and most of the companies and universities. In this field of research, algorithms are first tested in a simulated environment, prior to testing in the real world. Although there are a number of tools for simulation of autonomous vehicles, it is very difficult to pick one tool for a particular development, as each tool has its own advantage and it is only possible to choose the right tool based on the application at hand. Here are a few research platforms, from which ideas and inspirations were drawn.

One of the very famous and prominent platforms for autonomous vehicle testing is the CAT Vehicle (Cognitive and Autonomous Test Vehicle) [4], which is an open-source SIL (Software in the Loop) modeling and simulation done in Gazebo using ROS. This research testbed comprises of a simulated model of Ford Explorer. It is designed to mimic the dynamics of the real vehicle. It includes simulated sensors and actuators with configurable parameters. It also has the option of multi-vehicle simulation to support vehicle to vehicle interaction. Data can be logged for examining scenarios.

Carla is an open-source simulator developed for autonomous vehicle research [5]. It is implemented over the Unreal Engine 4 (UE4). It simulates a simple interface between a dynamic world and an agent to interact in the world. The world is composed of all objects found in an urban scenario such as buildings, traffic signs, infrastructure, vehicles, pedestrians, etc. All objects are of a similar scale and modeled to reflect real-life sizes. It allows for flexible configuration of sensors. The number of sensors and the positions can be specified by the user. Carla provides a simulation engine to test three approaches to

autonomous driving, a classic modular pipeline and using deep learning, for imitation learning and reinforcement learning. It has various weather settings for testing in various weather conditions that the autonomous vehicles may have to perform in.

Airsim [52] is a similar open-source simulator, which is also built on the Unreal Engine, is cross-platform, which supports HiL with flight controllers for drones. It has been developed with a goal to serve as a platform for AI research to experiment with deep learning, computer vision, and reinforcement learning algorithms for autonomous vehicles. The vehicle model provided includes a vehicle as a rigid body with parameters such as mass, inertia, coefficients for linear a linear and angular drag, friction, etc. The vehicle is exposed to various physical phenomena like gravity, air density, air pressure, and magnetic field.

The dbw_mkz_simulation is an ADAS development kit created by the Dataspeed Inc. It predominantly focuses on simulation and visualization of vehicle models in virtual environments inside the Gazebo environment [6]. This ADAS kit is developed to test autonomous algorithms in SiL (Software in the Loop) and HiL (Hardware in the Loop) testing. There are some examples of lane-keeping in the simulated environment with multiple vehicle models, but the downside is that the user cannot use their own vehicle models and environment or testing as some core modules provided cannot be modified as the user requires.

There are other commercial simulators for autonomous vehicles created by the huge companies for research such as Constellation [53] created by Nvidia, Carcraft of Waymo, Prescan by Tass international, Carmaker by IPG automotive, etc. These simulation

software and many others similar to these are developed on top of a very powerful physics engines like Unity, PhysX, and the Unreal Engine. These all are excellent tools for testing and validation of autonomous vehicle algorithms. The issue with these is an expensive license, which cannot be afforded by everyone that is interested in research in the autonomous field. Hence, I created this framework, which is easily available to all and has the flexibility to create our own environment and vehicle models. The following sections will explain in detail how to design and implement this framework and how to apply it for autonomous vehicle applications such as traffic sign detection and lateral control.

1.2 Motivation

The introduction reveals the necessity of further research in the field of autonomous vehicles. Though there is already significant research completed, there is still scope for a lot of advanced research that needs to be done. Keeping this in mind, I decided to contribute to the autonomous vehicle research community via this thesis project. The thesis is based upon the development of an autonomous vehicle capable of navigating autonomously using sensor inputs for perception. The main problem faced in research of autonomous vehicles is the lack of a platform for testing algorithms, and this was the main motivation for this thesis. Here I aim to provide a complete and effective autonomous solution, that is effortless to use and at the same time straight forward for further development. The motivation has led me to develop a system fully capable of detecting and recognizing traffic signs. I have developed this framework to be flexible enough for building on top of and anybody interested in this research will be able to replicate the work and add more value to the existing framework.

1.3 Objectives

The objective of this thesis is to develop an autonomous vehicle capable of recognizing traffic signs and taking actions based on the sign recognized, inside a simulation environment. To achieve the objectives, I have broken the thesis into the following goals:

1. Develop a simulation environment to replicate real-world scenarios, including all essential elements like roads, buildings, traffic signs, and also moving objects such as pedestrians.
2. Develop a vehicle model in simulation with all the crucial sensors integrated within, necessary to achieve autonomous driving.
3. Develop an object detection and recognition system and provide it with the capability of recognizing traffic signs.
4. Develop a system capable of driving itself autonomously by perceiving its environment and taking decisions based on input.
5. Maintain reproducibility by making the research freely available with proper documentation for recreating this work and future development.

CHAPTER 2. METHODOLOGY

2.1 Simulation Framework

The start of this thesis is set off with the construction of a framework for the simulation environment followed by a vehicle model with all the necessary sensors integrated into the vehicle model. This section will explain the details of the framework design and the procedures that will help you replicate the software in the loop (SIL) model development of such a system.

Plugins and scripts written in Python and C++ are used in the development of this framework. The simulation environment used together with ROS kinetic is Gazebo 9. The further chapters of this thesis explain the procedure on how to design and train the object detection model and implementing it in Gazebo. Darknet is an open-source neural network framework, which is the backbone for the real-time object detection system YOLO. The system is trained to detect and recognize traffic signs inside the Gazebo.

2.1.1 ROS

ROS (Robot Operating System), is an open-source, meta-operating system for any robotic system, regardless of it being in a simulation or real world. It provides the services such as hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management. It also has tools and libraries which facilitate the obtaining, building, writing, and running code across multiple computers. The reason behind using ROS as middleware to maintain the modularity of the perceptions and decision-making modules [7].

2.1.1.1 ROS Installation

The first step in the implementation of this framework is to install the correct distribution of ROS. For the purposes of this thesis, the first requirement is the installation of ROS Kinetic. The installation process is well documented and mostly straight forward. Here's a summary of the necessary steps.

1. Setup your sources.list: Set up your computer to accept software from packages.ros.org.

```
$ sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main" > /etc/apt/sources.list.d/ros-latest.list'
```

2. Set up your keys: Before installing the ROS packages, you must acquire their package authentication key

```
$ sudo apt-key adv --keyserver 'hkp://keyserver.ubuntu.com:80' --recv-key C1CF6E31E6BADE8868B172B4F42ED6FBAB17C654
```

3. Update the Linux environment: Make sure your Debian package index is up-to-date for proper installation.

```
$ sudo apt-get update
```

4. Installing the ROS packages: Now finally the ROS software can be installed. The simplest approach is to perform a complete install of the core ROS system:

```
$ sudo apt-get install ros-kinetic-desktop-full
```

5. Initialize rosdep: Before you can use ROS, you will need to initialize rosdep. rosdep enables you to easily install system dependencies for the source you want to compile and is required to run some core components in ROS.

```
$ sudo rosdep init
```

```
$ rosdep update
```

6. Environment setup: It's convenient if the ROS environment variables are automatically added to your bash session every time a new shell is launched:

```
$ echo "source /opt/ros/kinetic/setup.bash" >> ~/.bashrc
```

```
$ source ~/.bashrc
```

2.1.1.2 Basics of ROS

The basic principle of a robot operating system is to run a great number of executables in parallel that need to be able to exchange data synchronously or asynchronously [8]. The following are terminologies used to describe the working principle of the ROS framework.

1. Node: A node is an instance of an executable. It uses ROS to communicate with other nodes. A node equates to a sensor, motor, processing or monitoring algorithm, and so on. Every node that starts running registers itself to the Master.

2. Master: Provides naming and registration services to the rest of the nodes in ROS, facilitating communication. Master makes it possible for nodes to find each other and exchange data. ROS master is started by the command `$ roscore`. You should allow the master to continue running for the entire time that you're using ROS.

3. Topic: A topic is a data transport system based on a subscribe/publish system. Nodes can publish messages to a topic or can subscribe to a topic to receive messages. One or more nodes can publish data on a topic, and one or more nodes can read data on that topic. Data is exchanged asynchronously by means of a topic and synchronously via a service. This notion of an asynchronous, many-to-many bus is essential in a distributed system situation.

4. Message: A message is a compound data structure used for publishing and subscribing to a topic. A message may comprise of a combination of primitive data types of character strings, Booleans, integers, floating-point, etc. Messages are a recursive structure. The message description is stored in package_name/msg/myMessageType.msg. This file describes the message structure.

5. Services: A service is for synchronous communication between two nodes and is a bi-directional channel between nodes. One node sends information to another node and waits for a response. Information flows in both directions. Service calls implement one-to-one communication. Each service call is initiated by one node, and the response goes back to that same node. The description of a service is stored in package_name/srv/myServiceType.srv. This file describes the data structures for requests and responses. The interfaces in ROS can be seen in Figure 1.

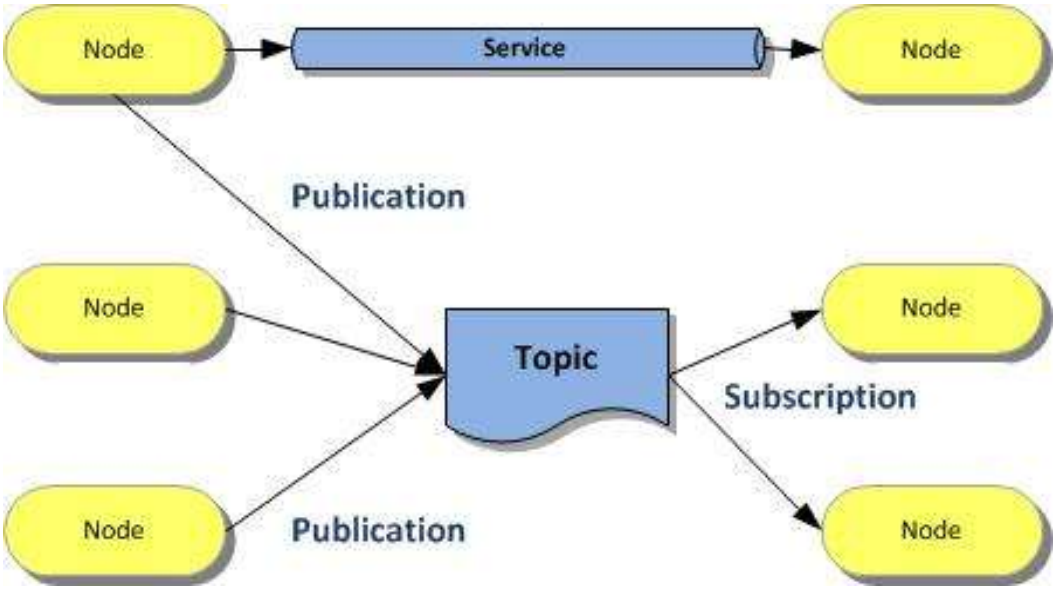

Figure 1. ROS interface [8]

2.1.2 Gazebo

The Gazebo is an open-source 3D simulation software, which is mostly used to test new code or new physical systems that may be too dangerous or expensive to test in the real world. Gazebo is a very useful tool as it gives you the capability to simulate robots accurately through defining the robot joint and links. Joint [11] is a point which is connecting two links, where links are the actual robot parts. Gazebo allows for testing of complex systems regardless of it being robotic or otherwise. It has many uses, including testing the dynamics of a control system before the system has been actualized.

Gazebo is supported by the ODE [12] Physics rendering engine. ODE is a high-performance library for simulating rigid body dynamics written in C/C++. Gazebo performs simulations with a high degree of fidelity and a decent graphical rendering feature. It uses Object-Oriented Graphics Rendering Engine (OGRE) [13], for rendering graphics. It allows the manipulation of aesthetic attributes, like geometric orientation,

lighting, etc. One of the most important aspects of Gazebo is that it allows the manipulation of the real-time factor, meaning, it allows the simulation to speed up or slow down. The procedure to install Gazebo is as follows:

1. Start with Setting up your computer to accept software from packages.osrfoundation.org

```
$ sudo sh -c 'echo "deb http://packages.osrfoundation.org/gazebo/ubuntu-stable `lsb_release -cs` main" > /etc/apt/sources.list.d/gazebo-stable.list
```

2. Setup keys for system to accept Gazebo package

```
$ wget http://packages.osrfoundation.org/gazebo.key -O - | sudo apt-key add -
```

3. Update the Linux environment to make sure your Debian package index is up-to-date for proper installation

```
$ sudo apt-get update
```

4. Install Gazebo using the command

```
$ sudo apt-get install gazebo9
```

5. Check your installation by executing,

```
$ gazebo
```

In order to be able to modify or create an environment, it is important to understand the file format that is used in Gazebo. Gazebo makes use of various file formats such as world file, SDF file [14], and URDF file [16]. Each of these files contains the necessary

information for describing the components of the simulation. I have provided a brief and short explanation of these files and their usage in this section.

A world description file contains all the elements existing in the simulation environment such as traffic lights, traffic signs, pedestrians, buildings, roads, and weather conditions, lighting conditions, etc. The world file is formatted using the file extension of .world. It follows a Simulation Description Format (SDF), which is written in XML format. The world file is read and executed using the Gazebo gzserver [15]. The gzserver is the component of Gazebo, which is responsible for the physics update loop and sensor data generation. It is considered the core of Gazebo. It has a counterpart, Gazebo gzclient, which provides a graphical user interface and visualization of the simulation. It also provides controls for the simulation properties. Together these two components make it possible to build and run robust and realistic simulations.

The file for performing simulation for robot models is the Universal Robot Description Format, or URDF. This file is also written in the XML format and is used heavily in ROS for simulation and testing. It is a supported file type for Rviz [17] and other ROS tools. The file is a tree structure of child links (like vehicle wheels and robot arms) connected to parent links (like the chassis and robot body) by a series of joints. This file is a description of how your system moves internally, and this will determine how it can interact with the environment. The capabilities of the file, however, are limited in comparison to the SDF. The SDF files are what Gazebo uses when performing simulation, and in fact, before importing one of the URDF models, Gazebo will convert it into an SDF. This file format includes more details like friction, damping, and environment properties like heightmaps and lights that are excluded from the URDF file [18]. For simplicity, most

of the objects existing in the world file are created in a separate model files. These model files are also written in SDF format. The purpose of having separate model files for each object is that it facilitates model reuse between different projects. There is a database available online for most of the gazebo models [19], that can be used to create a simulation environment.

Apart from these advantages, Gazebo is capable of simulating sensors, with the help of plugins. These plugins are a set of codes that help us to control the robot models, sensors, world properties, and even the way Gazebo simulation is performed [20]. A set of ROS packages named gazebo_ros_pkgs enhances the capacity by integrating Gazebo with ROS [21], making it highly suitable for robotic and autonomous applications.

2.2 Simulated Environment

The simulation is created to replicate urban layouts. The users will be able to use the environment as if they were inside an actual city layout. The simulation environment was adopted from a package named car_demo [22], which was created by the Open Source Robotics Foundation (OSRF). For the purposes of this research, I have made some modifications to the original package. These modifications are such as the addition of traffic signs, traffic lights, pedestrians, etc. Even the roads were modified by adding different lane markings. In Figure 2 shows a view of the simulated environment inside Gazebo.

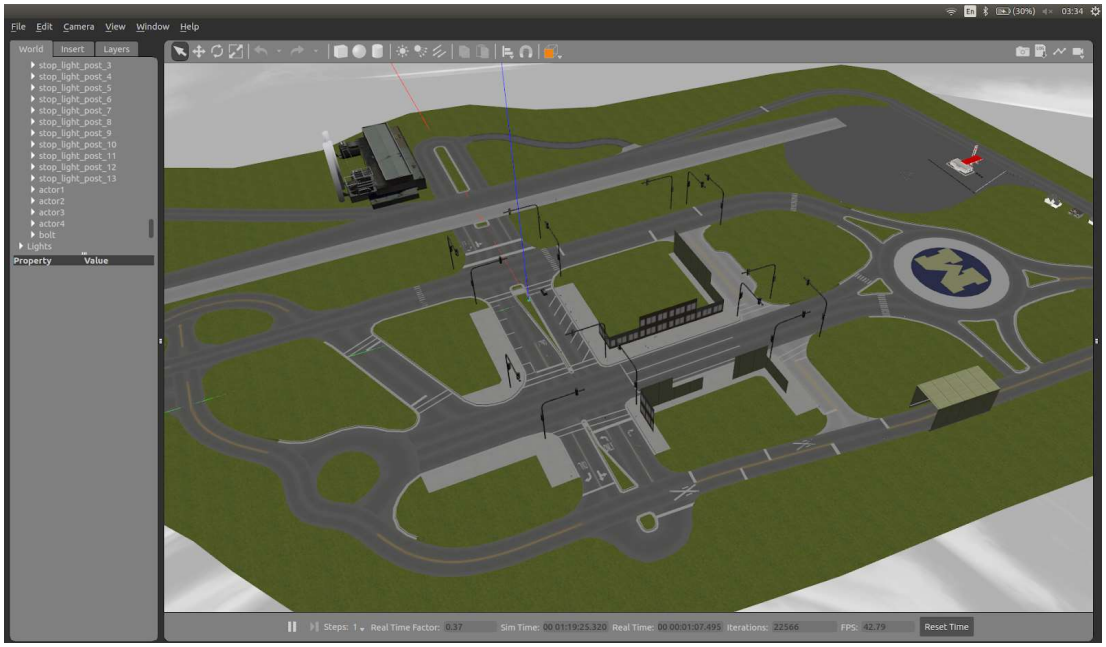

Figure 2. Mcity Model in the Gazebo

The process of modification begins with the creation of objects that will be imported into the simulation. In my case, I have created various road networks, traffic signs, etc. Each object was created with a separate file to maintain reproducibility. As mentioned in the earlier section, these files need to be in an SDF format. The SDF contains simulation details for the objects in the form of tags such as inertia, visuals, collision, etc. Most of these are self-explanatory as the SDF is a human-readable language. The main tags used for our purpose are, <visual>, <collision> tags. The <visual> tag defines how the object looks. This basically specifies the model shape and size. The <collision> defines how it interacts with other objects, robot models, meaning, if I want to create a scene with a vehicle crashing an object, then I need to specify the collision so that the simulator can interpret this information and apply the underlying physics engine and make the crash look realistic. Without this collision tag, the vehicle will just pass through the object. The SDF provides us with the ability to be very detailed, as there are tags such as <surface>,

<shader>, <inertia>, etc., which represent the friction of the object model, the light reflectivity, and the center of mass respectively. For the purposes of this thesis, I have decided to keep the models very basic and simple and hence I have only used visuals and collisions.

Upon creation of the SDF files, these will be added into the world file, inside which all the simulation features will be specified. This world file contains a complete definition of the environment. It specifies all simulation features, such as the lighting conditions, ground plane texture, wind, etc. In the next section, I have explained in detail how to create a model to use in Gazebo.

2.2.1 Traffic Sign Modelling

In this section, I will provide the steps required to model a traffic sign. The example shown here is for the creation of a stop sign. The first step in creating a stop sign for the Gazebo simulation is to create a 3D computer-aided design (CAD) model of the desired object. The CAD model is a complete description of the shape and size of the object. I completed the CAD design inside SolidWorks [23]. Once the CAD model ready, it needs the addition of surface texture to the model. I have made use of a software called Blender [24] for this reason. The texture is where the model gets proper visualization. Here it allows the addition of stop sign image, the surface texture and other details such as reflectivity, exposure, etc. I have used a tool called the UV editor, provided inside the Blender software to accomplish the task of adding textures. Upon completion of texture editing, the object model is exported in a COLLADA file format, which has a .dae extension. The reason for using this file format is easy compatibility with Gazebo.

Now that the model is textured and ready for use, it is required to create some folders and files that are specific to Gazebo simulations. This is due to some basic conventions kept in place by the Gazebo software, which are a necessity for the model to be accepted. Gazebo requires a specific directory to hold the model file, the material file, and the configuration file. The model directory can be seen in Figure 3.

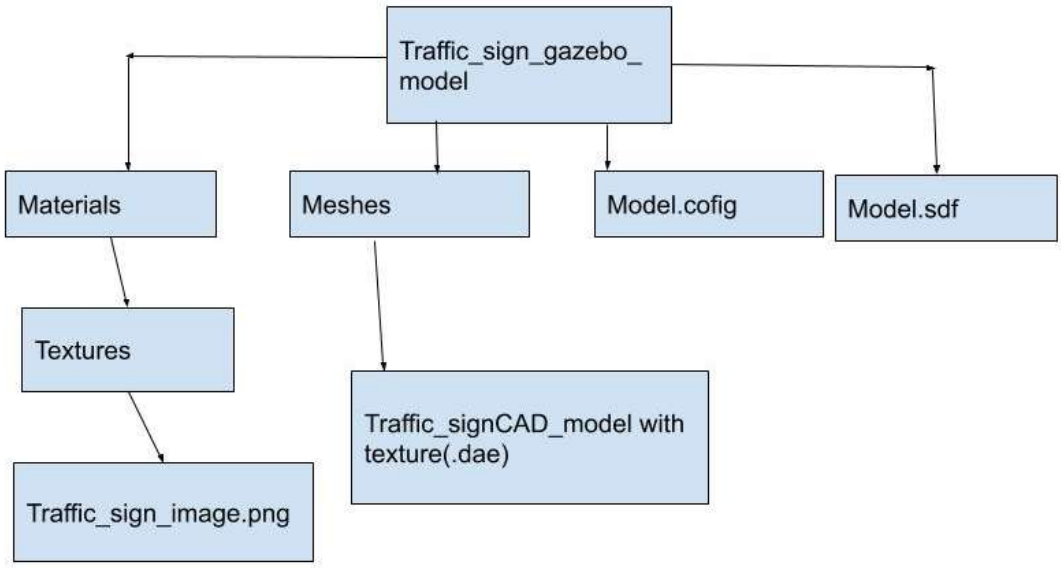

Figure 3. Gazebo model file directory

The COLLADA file(.dae) exported from blender is saved inside the traffic_signs/catkin_ws/src/car_demo/car_demo/models/stop/meshes directory. The corresponding image file, that defines the texture is placed inside traffic_signs/catkin_ws/src/car_demo/car_demo/models/stop/materials/textures directory. Next, step is to create two files, the model.config file and the model.sdf file.

The model config file contains information about the model name, author name, description and most importantly the version. I have provided the contents of the model config file below.

```
<?xml version="1.0"?>
<model>
  <name>speed_10</name>
  <version>1.0</version>
  <sdf version="1.5">model.sdf</sdf
  <author>
      <name>mir lab</name>
    <email>@kettering.edu</email>
  </author>
  <description>
   Stop Sign model
  </description>
```

The model SDF file, as explained in the previous section and mainly contains visual and collision information. Here I have provided a sample of the model SDF file.

```
<?xml version="1.0"?>
<sdf version="1.5">
  <model name="stop_sign">
      <link name="link">
      <collision name="collision">
      <geometry>
            <mesh>
            <scale>1 1 1</scale>
```

```
        <uri>model://stop_sign_gazebo_model/meshes/stop_sign.dae</uri>
              </mesh>
        </geometry>
        </collision>
        <visual name="visual">
        <geometry>
              <mesh>
              <scale>1 1 1</scale>

        <uri>model://stop_sign_gazebo_model/meshes/stop_sign.dae</uri>
              </mesh>
        </geometry>
        </visual>
        </link>
  </model>
</sdf>
```

Both these files need to be placed inside, traffic_signs/catkin_ws/src/car_demo/car_demo/models/stop directory. In Figure 4, shows how the stop sign looks inside the Gazebo environment. Following these conventions provided here is not only mandatory but also makes it easy to better understand the model and at the same time makes it very convenient to modify existing models or add new models. Any object model can be designed in Gazebo using these exact steps and following these conventions.

Figure 4. Stop Sign in Gazebo Environment

Once all the objects are created successfully added in the proper directory, the objects will be reflected in the gazebo models list under the Insert tab, on the left side panel as seen in Figure 5. From here the objects can be simply dragged and dropped to model the simulation environment as desired. This feature is very convenient when it is needed to design a large city model.

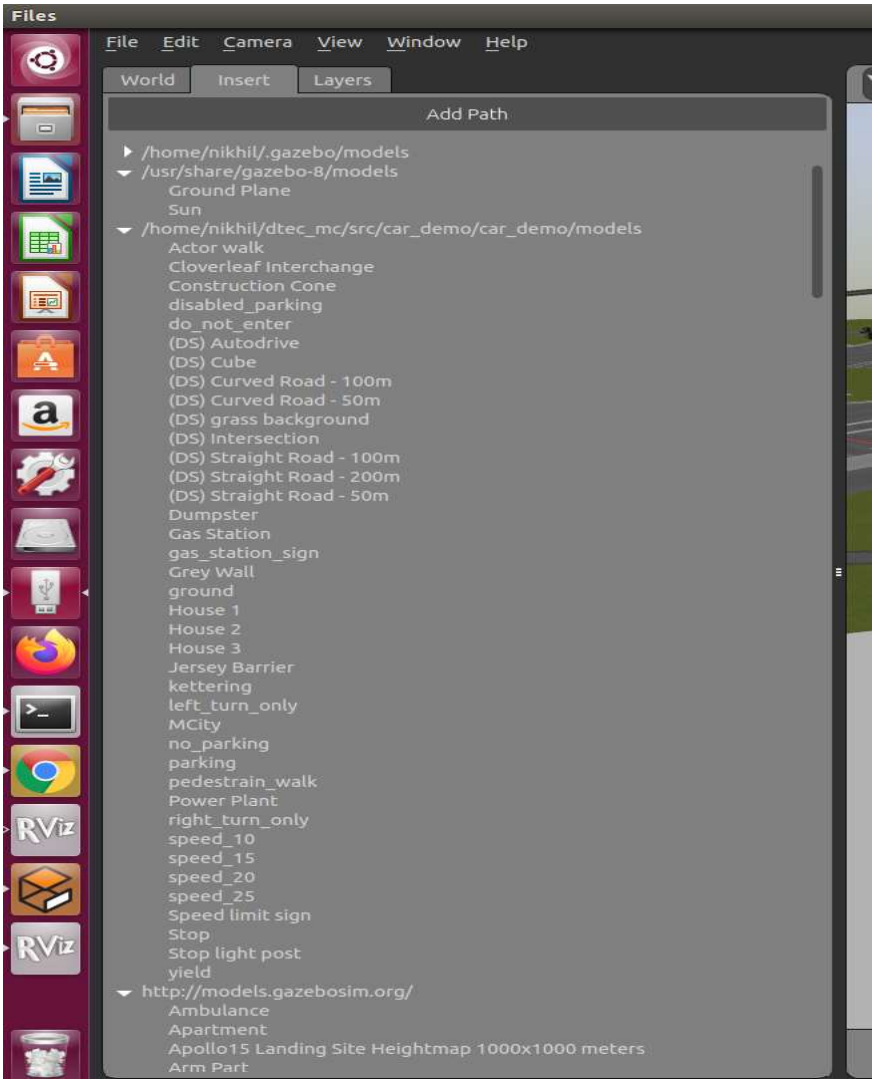

Figure 5. Gazebo object panel.

2.2.2 Pedestrian Modelling

There is a wide variety of Gazebo models available for download from the ORSF website and these include traffic lights, walking/standing pedestrians, SUV, houses and many more. These can be added to our simulation in a similar fashion described in the Traffic sign modeling section. Once the objects have been successfully added, it is easy to move the objects around. This section will explain how this can be achieved. For modeling

moving objects in the gazebo environment, such as a pedestrian, the trajectory of the moving object needs to be specified, within the world file. For better understanding, I have provided a snippet of the world file containing a moving pedestrian below.

```
<actor name="actor1">
      <plugin name="actor_collisions_plugin"
filename="libActorCollisionsPlugin.so">
      <scaling collision="LHipJoint_LeftUpLeg_collision" scale="0.01
0.001
0.001"/>
      <scaling collision="LeftUpLeg_LeftLeg_collision" scale="8.0 8.0
1.0"/>
      <scaling collision="LeftLeg_LeftFoot_collision" scale=" 8.0 8.0
1.0 "/>
      </plugin>
      <skin>
      <filename>walk.dae</filename>
      <scale>1.0</scale>
      </skin>
      <animation name="walking">
      <filename>walk.dae</filename>
      <scale>1.000000</scale>
      <interpolate_x>true</interpolate_x>
      </animation>
      <script>
      <loop>true</loop>
      <auto_start>true</auto_start>
      <trajectory id="0" type="walking">
```

```
        <waypoint>
          <time>10.000000</time>
          <pose>-116.363 -196.789 0.316128 0 0 -0.00514</pose>
          <waypoint>
          <time>11.500000</time>
          <pose>-116.363 -196.789 0.316128 0 -0 1.57</pose>
          </waypoint>
          <waypoint>
          <time>11.500000</time>
          <pose>-116.363 -196.789 0.316128 0 0 -0.00514</pose>
          </waypoint>
      </trajectory>
      </script>
      </actor>
```

Waypoints can be provided for the pedestrian under <trajectory> tag in the world file. Here it is seen that the pedestrian, named as actor1 has been added in a time loop, with a fixed trajectory. The <time> tag sets the time it will take to reach each waypoint. I have also added other properties such as collision so that the crashes are realistic, and vehicle does not pass through the pedestrian. This method can be used to move any object, such as a car, bike or truck.

2.3 Vehicle Model

As discussed earlier, I require a vehicle model with integrated sensors in order to achieve autonomous driving. The car_demo package taken from the OSRF's model repository comes along with a simulated model of the Toyota Prius. For this thesis, I have

converted this model into a Chevrolet Bolt. The reason behind conversion is to make this framework usable by the Kettering University students involved in Auto Drive Challenge [25]. The vehicle model can be seen below in Figure 6. The modifications made to the Prius model for conversion into Bolt have been described in this section.

Figure 6. Chevrolet Bolt model in Gazebo.

It is already seen that a world file contains the information necessary for simulating an environment. Similarly, for the vehicle model simulation and visualization, A universal robotic description format (URDF) model needs to be created. A URDF is written in an XML macro language and is referred to as Xacros. This format is widely used in ROS to describe robot models. It enables the creation of property and property blocks which can be repeated to create similar parts such as wheels. The URDF is structured like a tree, with one root link which is usually called the base_link. And other links branch out from the root link. These branches are comprised of links and joints. The links define a rigid body, which contains the information of inertia, visuals and collision property of the rigid body.

It can be a simple shape such as a sphere, box, cylinder or a .dae file, which is a COLLADA file directly imported as a mesh. All links are joint to the base_link using joints. Joints can be of various types like revolute, continuous, fixed, floating, prismatic or planar. The joints specify the root link and the branched link connected to it, which are referred to as the parent link and the child link.

The specifications of the car can be changed or manipulated in the URDF file. In order to modify the vehicle model, it is necessary to know a few details about the URDF file. These are mainly essential elements enclosed in a tag. These are described as follows:

1. Link: A link describes the physical characteristics of a part of the model. It contains all the aspects necessary to describe the part body such as the model mesh. Usually, parts with movable joints are considered as different links. Elements without movable joints are grouped under the same link.

2. Collision: It encapsulates a geometry that is used for checking the interactions between two objects. The collision of an object can be a simple shape or a mesh which accurately simulates the body geometry. A collision will consume computational resources without any needed necessity. A link may contain many collision elements.

3. Visual: This is a visual description of the part that is described within the link. As explained earlier, it is the 3D shape of this link.

4. Inertia: Inertia describes the dynamic properties of the link, such as mass, rotational inertia matrix. This is mainly used when simulating high fidelity physics as it also requires a lot of computational resources.

5. Sensor: This element is used to simulate the sensor in the simulation environment. It contains all the properties that are needed to describe the working of a sensor inside a simulation. More on sensors will be explained in the Simulated Sensors section.

6. Joint: A joint is the connecting point between two links, it describes the connection pivot between a parent and a child link. It also controls other parameters such as joint limits, the axis of rotation of this element. In simple words, joint describes how two parts move in simulation.

7. Plugin: A plugin is a set of codes that can be used to control almost all the elements in the simulation. This element is used to monitor and control your model in the simulation environment.

All the above-mentioned elements are put together in a model URDF file. In Figure 7 shows the structure of the vehicle URDF.

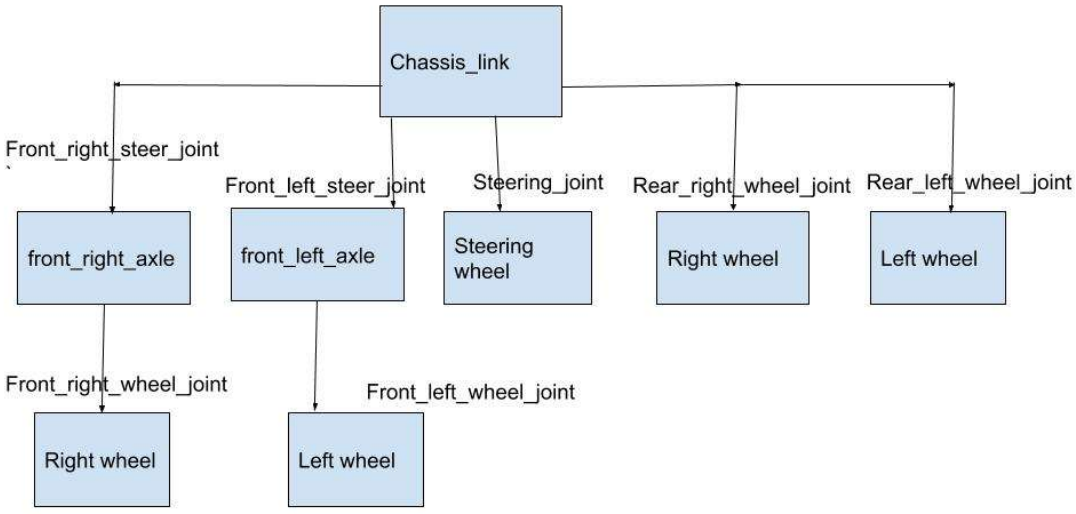

Figure 7. Vehicle URDF structure

Now that the elements that make up a URDF file are known, I will explain the process of importing the vehicle model. The process for importing a vehicle model in Gazebo is the same as importing any object model, as it has been already seen in the Traffic sign modeling section. The difference is that the files need to be added to the robot description directory and, it does not require special configuration and SDF files for simulating robot models using URDF. I first created a 3D CAD model, add texture and export as a COLLADA file. The meshes along with the textures are placed inside traffic_signs/catkin_ws/src/car_demo/bolt_description/meshes/bolt. While the URDF file needs to be placed inside the traffic_signs/catkin_ws/src/car_demo/bolt_description/urdf directory. The meshes are linked to the robot model inside the URDF by providing the path of the directory where the meshes are saved. This is more clearly understood from the snippet of the URDF provided below.

```
<link name="chassis">
     <visual>
     <origin xyz="0 0.05 0.05" rpy="1.57 0 0"/>
       <geometry>
     <mesh filename="package://bolt_description/meshes/bolt/body.dae"
  scale="0.064 0.064 0.064"/>
     </geometry>
     </visual>
     <collision name="chassis">
     <origin xyz="0.0 0.05 0.625" rpy="0 0 0"/>
     <geometry>
     <box size="1.7526 2.1 0.95"/>
```

```
      </geometry>
      </collision>
      <collision name="front_bumper">
      <origin xyz="0.0 -2.0 0.458488" rpy="0.0 0 0"/>
      <geometry>
      <box size="3 1 0.566691"/>
      </geometry>
      </collision>
  </link>
```

The above-seen snippet shows how links are defined inside a URDF file. It also shows the import of the mesh file. The path to the mesh file is added in the <geometry> tag. This enables the URDF file to simulate the part in the simulation. I have added another snippet for the same URDF showing the relation between joints and links.

```
<joint name="front_right_wheel_joint" type="continuous">
      <parent link="fr_axle"/>
      <child link="front_right_wheel"/>
      <origin xyz="0.2 0.55 0" rpy="0 0 0"/>
      <axis xyz="1 0 0"/>
  </joint>
```

In the above snippet, Shows the connection between the parent and child link via a continuous joint. Apart from this, it defines the origin of the joint and its axis of rotation. Here the connection between the front wheel and the front axle can be seen. Depending upon the purpose, any of the following type of joints can be used to establish a connection:

1. Revolute: A revolute joint is a 1 DOF (degree of freedom) joint that allows rotation about a single axis. It can be used to describe an arm or hinge joint. It is capable of multiple rotations but allows for joint limitations so that you could restrict the motion to a 90-degree arc.

2. Continuous: A continuous joint rotates around an axis with no boundary conditions It is best represented by a wheel, which can rotate continuously.

3. Prismatic: A prismatic joint is also a 1 DOF linear translating joint. It allows for translation along an axis, but not the rotation around the axis.

4. Fixed: A fixed joint is used when you want a child link to be incapable of movement with respect to the parent link. In other words, the link is fixed. The movement is totally locked.

5. Floating: As the name suggests, a floating joint is not directly connected and allows 6 Degree of freedom movement.

6. Planar: A planar joint allows movement in a plane perpendicular to a specific axis. These types of joints may be used to define the motion of a piston.

Pay attention when working on the CAD model for the vehicle as to always keep the body separate from the wheels. The reason behind this is to maintain proper joint movements between each element. For example, if you decided to add the wheel and the vehicle body in the same 3D CAD model, you wouldn't be able to see any wheel rotation or assign a separate link for the wheels, because it is already grouped on one DAE file. For

avoiding this error, I have maintained the vehicle body, steering wheel, and the tires, and all other movable parts have a separate (.dae) file.

I have described all the links, joints, necessary elements, the directories where the files need to be added and also a brief on how everything is connected together. The converted vehicle model (Bolt) is provided along with the original Prius model, which will make it easier for the reader to compare how things have changed from the original. The next section describes the addition of sensors to the vehicle model.

2.3.1 Simulated Sensors

The key to any simulation for autonomous vehicles is the virtualization of the sensors used, as all the decision-making is dependent on the sensor feedback. Our vehicle model is equipped with different sensors, that exists in most of the autonomous vehicles. I have integrated the sensors with readily available plugins that are used to set the parameters for each of the sensor models, enabling them to represent the real sensor performance. Some of the attributes of these sensors like the frequency, range, the data output can all be changed using sensor plugins. The sensor plugins are added to the URDF file. Here, the name of the topic, sensor type, range, resolution, etc can be defined. The sensor locations can vary depending on the type of car, view angle, the purpose of the sensor, etc. within the URDF file.

2.3.1.1 Camera

The camera sensor integrated with the vehicle is an eight-bit stereo camera. The camera model is already available in the Gazebo model database and can be easily adopted

into simulation with any modifications needed. I will explain in detail the changes made to the default camera provided by Gazebo. The resolution of the camera is 800x800 pixels. I have integrated 4 cameras in total, 3 in front and 1 in the back. A camera frame named camera link is used to publish the camera sensor messages. The naming changes depending on the location of the camera, for example, the front camera is named front_camera_link and publishes the sensor messages over the topic /bolt/front_camera/image_raw. The camera plugin used to control the characteristics is called libgazebo_ros_camera.so and the plugins are saved inside traffic_signs/catkin_ws/src/car_demo/car_demo/plugins directory. Below I have provided a snippet of the URDF file showing the camera information.

```
<joint name="camera_joint" type="fixed">
      <axis xyz="0 1 0" />
      <parent link="chassis"/>
      <child link="front_camera_link"/>
      <origin xyz="0 -0.7 2.52" rpy="0 0 -1.57"/>
  </joint>
  <gazebo reference="camera_link">
      <sensor type="camera" name="stereo_camera">
      <update_rate>30.0</update_rate>
      <camera name="center">
      <horizontal_fov>1.3962634</horizontal_fov>
      <image>
            <width>800</width>
            <height>800</height>
```

```
            <format>R8G8B8</format>
        </image>
        <clip>
            <near>0.02</near>
            <far>300</far>
        </clip>
        <noise>
        <type>gaussian</type>
            <mean>0.0</mean>
            <stddev>0.007</stddev>
        </noise>
        </camera>

        <pluginname="stereo_camera_controller"filename="libgazebo_ros_cam
era.so">
        <alwaysOn>true</alwaysOn>
        <updateRate>0.0</updateRate>
        <cameraName>stereo_camera</cameraName>
        <imageTopicName>/bolt/front_camera/image_raw</imageTopicName>
        <cameraInfoTopicName>camera_info</cameraInfoTopicName>
        <frameName>camera_link</frameName>
        <hackBaseline>0.07</hackBaseline>
        <distortionK1>0.0</distortionK1>
        <distortionK2>0.0</distortionK2>
        <distortionK3>0.0</distortionK3>
            <distortionT1>0.0</distortionT1>
        <distortionT2>0.0</distortionT2>
```

```
        </plugin>
        </sensor>
    </gazebo>
```

The snippet provided above displays all the details of the camera sensor. The pixel information is provided in the <height> and <width> tags. The format of the image is in RGB format, contained inside the <format> tag. The range of the camera can also be changed by setting desired values in the <near> and <far> tags. I have also added Gaussian noise to the camera to make it more realistic [26]. The noise is parameterized using standard deviation, and all the noise information is contained inside <noise> tag. It can also be seen that the most important aspect, that controls the camera, is the plugin. The plugin can be manipulated to modify various attributes such as update rate, distortion, camera info topic, etc. The camera is connected to the chassis_link via a fixed joint, with no rotation or any other type of movement.

2.3.1.2 LiDAR

One of the most used vehicle sensors for autonomous driving is LiDAR. I have integrated the vehicle model with a Velodyne LiDAR, which is also provided free in the Gazebo model database. The process for integration and modification of the LiDAR model is similar to the integration of the camera. The Velodyne LiDAR is mounted on top of the vehicle and it publishes data at a rate of 10 Hz. The sensor frame is named velodyne_link, which is used to publish the LiDAR sensor messages. The sensor information is published over the topic, /bolt/center_laser/scan. The LiDAR characteristics is also controlled using a LiDAR plugin, and is called libgazebo_ros_block_laser.so. The plugin is saved inside

the traffic_signs/catkin_ws/src/car_demo/car_demo/plugins directory. All this information is contained inside the vehicle URDF file and a snippet is provided below.

```
<joint name="velodyne_joint" type="fixed">
        <axis xyz="0 0 0"/>
        <origin xyz="0 -0.7 2.52" rpy="0 0 -1.57"/>
        <parent link="chassis"/>
        <child link="velodyne_link" />
 </joint>
<gazebo reference="velodyne_link">
        <sensor name='velodyne_sensor' type='ray'>
        <visualize>0</visualize>
        <update_rate>10</update_rate>
        <ray>
        <scan>
        <horizontal>
        <samples>1875</samples>
        <resolution>1</resolution>
        <min_angle>-3.14159</min_angle>
        <max_angle>3.14159</max_angle>
        </horizontal>
        <vertical>
        <samples>16</samples>
        <resolution>1</resolution>
        <min_angle>-0.261799</min_angle>
        <max_angle>0.261799</max_angle>
        </vertical>
```

```
</scan>
<range>
<min>0.055</min>
<max>80</max>
<resolution>0.01</resolution>
</range>
<noise>
<type>gaussian</type>
<mean>0</mean>
<stddev>0.1</stddev>
</noise>
</ray>
```

In the snippet above the LiDAR is connected to the chassis_link via a fixed joint with a rotation of zero. The update rate of the LiDAR is set to 10 Hz and has a resolution of 0.01 m. the Vertical scan has a minimum angle of -3.141 and the and maximum angle of 3.14, measured in radians, while the horizontal scan has a minimum angle of -0.261 and a maximum angle of 0.261. All angles are measured in radians and are contained within the <horizontal> and <vertical> tags. The minimum range is set to 0.055 m and has a maximum range of 80 m. To make the LiDAR data more realistic, I have added Gaussian noise, just like the camera sensor. These parameters are saved inside the URDF file and contribute to the sensor performance. Based on the application needs, changes can be made in this file and in turn change the performance.

2.3.2 Vehicle Controller

The vehicle model is now fully equipped with the necessary sensors, the last step is to control the vehicle movements. Here I have controlled the vehicle in Gazebo using plugins. The type of controller used is a proportional-integral-derivative (PID) controller. The PID controller is integrated inside the plugin and I have adopted the default plugin that was accompanied with the Car_demo package. This plugin was written to control the Prius model, and I have made small modifications to it for controlling our vehicle. The modification includes changes to the topics for the throttle, brake, and steering. I have created a custom message structure for the topic to publish and subscribe to these values. It is created with a name control.msg and saved inside traffic_signs/catkin_ws/src/car_demo/bolt_msgs/msg directory. The throttle and brake values have a range of 0-1 while the steering values vary from -1 to 1. The Plugin also has configurations for keyboard controls for vehicle controls and uses ASCII code [27] for keyboard buttons. I have currently configured the controls to run with W, A, S, D set to control throttle, left, brake and right respectively. There are other aspects that can be controlled such as max speed, max torque within the plugin which have not been modified and are set to default. A snippet of the code has been provided below.

```
<gazebo>
        <plugin name="bolt_drive" filename="libboltPlugin.so">
        <chassis>chassis</chassis>
        <front_left_wheel>front_left_wheel_joint</front_left_wheel>
        <front_right_wheel>front_right_wheel_joint</front_right_wheel>
  <front_left_wheel_steering>front_left_steer_joint</front_left_wheel_steering>
```

```
        <front_right_wheel_steering>front_right_steer_joint</front_right_wheel_steering>
        <back_left_wheel>rear_left_wheel_joint</back_left_wheel>
        <back_right_wheel>rear_right_wheel_joint</back_right_wheel>
        <steering_wheel>steering_joint</steering_wheel>
        <chassis_aero_force_gain>0.63045</chassis_aero_force_gain>
        <front_torque>859.4004393000001</front_torque>
        <back_torque>0</back_torque>
        <front_brake_torque>1031</front_brake_torque>
        <back_brake_torque>687</back_brake_torque>
        <max_speed>80</max_speed>
    <min_gas_flow>8.981854013171626e-05</min_gas_flow>
        <gas_efficiency>0.371</gas_efficiency>
        <battery_charge_watt_hours>291</battery_charge_watt_hours>
        <battery_discharge_watt_hours>214</battery_discharge_watt_hours>
        <max_steer>0.6458</max_steer>
        <flwheel_steering_p_gain>1e4</flwheel_steering_p_gain>
        <frwheel_steering_p_gain>1e4</frwheel_steering_p_gain>
        <flwheel_steering_i_gain>0</flwheel_steering_i_gain>
        <frwheel_steering_i_gain>0</frwheel_steering_i_gain>
        <flwheel_steering_d_gain>3e2</flwheel_steering_d_gain>
        <frwheel_steering_d_gain>3e2</frwheel_steering_d_gain>
        </plugin>
  </gazebo>
  <gazebo>
<plugin name="joint_state_publisher"
filename="libgazebo_ros_joint_state_publisher.so">
        <!-- <robotNamespace>/bolt</robotNamespace> -->
```

```
        <jointName>rear_right_wheel_joint, rear_left_wheel_joint,
front_right_wheel_joint, front_left_wheel_joint, front_right_steer_joint,
front_left_steer_joint, steering_joint</jointName>

        <updateRate>100.0</updateRate>

        <alwaysOn>true</alwaysOn>

        </plugin>

         </gazebo>
```

In the snippet above the name of the modified plugin is seen, the bolt_drive. This plugin is saved inside the traffic_signs/catkin_ws/src/car_demo/car_demo/plugins directory. All joints have been specified here and are connected to the chassis link. The changeable vehicle parameters such as torque, brake torque, max speed, max steer, etc. can also be seen. I have left these values unchanged as testing these parameters is the focus of this thesis. Finally, A joint state publisher [28] is used to calculate all the transformations between all the defined joints in the URDF. With the help of an ignition math package, that provides a library to deal with all the math operations needed for robotics applications [29]. The main function of a joint state publisher is to watch the joints; for instance, it was used to move all wheels with the same velocity and to main the same frequency between the wheels.

2.4 Visualization of Environment

2.4.1 Rviz

Visualization of sensor information is an imperative part of the development and debugging process. I have used a tool called Rviz provided by ROS for sensor data visualization. It is a very powerful and flexible tool and can be customized depending on

the application. It provides options to add and delete sensor topics, robot models and transformation frames. It also allows to edit sensor properties such as color, resolution, update rate, etc. In Rviz the links connecting the vehicle model can be visualized. In Figure 8 the car model inside Rviz, with all the active sensors displayed is shown.

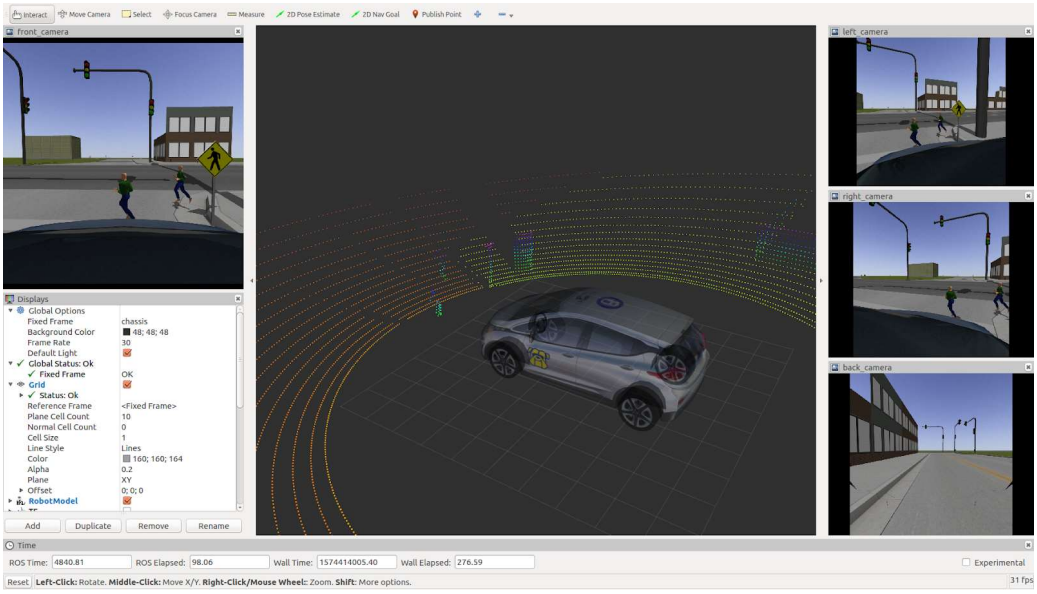

Figure 8. Vehicle model and sensor information

In the figure shown above, shows the vehicle model, images form 4 camera and the LiDAR scan data. I have provided a custom configuration for Rviz, that displays all the camera info and LiDAR scan upon launch. This configuration file named bolt.rviz, which is saved inside traffic_signs/catkin_ws/src/car_demo/car_demo/rviz.

2.4.2 ROS Launch

I have so far covered all the necessary elements that need to come together to simulate the environment, visualize sensors, etc and all the nodes and topics that are needed to have these elements running. I would need to open many terminals to start each node

separately. Fortunately, ROS provides a mechanism for running all essentials using a single file called a launch file [30]. Launch files are very common in ROS to both users and developers. As they provide a convenient way to start up multiple nodes and a master, as well as other initialization requirements such as setting parameters. Launch files have an extension .launch and use a specific XML format. The command used to open a launch file is roslaunch. I have provided a sample of the launch file below.

```
<?xml version="1.0"?>
<launch>
  <arg name="model" default="$(find bolt_description)/urdf/bolt.urdf"/>
  <arg name="rvizconfig" default="$(find car_demo)/rviz/bolt.rviz" />
  <param name="robot_description" textfile="$(arg model)"/>
  <include file="$(find gazebo_ros)/launch/empty_world.launch">
    <arg name="use_sim_time" default="true"/>
    <arg name="verbose" value="true"/>
    <argname="world_name" value="$(find car_demo)/worlds/mcity_modified.world"/>
  </include>
  <node  pkg="robot_state_publisher" type="robot_state_publisher"
name="robot_state_publisher" >
    <!-- <remap from="robot_description" to="different_robot_description" /> -->
    <!-- <remap from="joint_states" to="/prius/joint_states" /> -->
  <node name="spawn_urdf"  pkg="gazebo_ros" type="spawn_model" args="-param
robot_description -urdf -x 3 -y -12 -z 0.5 -model bolt"/>
  <node name="rviz" pkg="rviz" type="rviz" args="-d $(arg rvizconfig)" required="true" />
</launch>
```

Above seen are the various nodes launched such as Rviz, robot state publisher, etc. The launch file also spawns the robot model inside the simulation environment at the specified location. I have provided multiple launch files in the directory, traffic_signs/catkin_ws/src/car_demo/car_demo/launch. The contents of launch life must be contained inside a pair of <lunch> tags. Similarly, all nodes are contained inside <node> tag. It will be shown how to use launch files to launch all the packages used in the thesis, in the later sections.

2.5 Traffic Sign Detection and Recognition

In the previous chapters I have explained the process of creating a simulation environment for autonomous vehicles. This section contains the details regarding implementation of traffic sign detection and recognition. Here is a detailed description of the network architecture, data collection, performance matrix and training. The training results and the network performance have also been discussed in this section.

2.5.1 Darknet and YOLOv3

You only look once (YOLO), which implies that it only takes one glance at an image and predicts what objects are present and where they are located. It is a very computationally efficient and one of the fastest object detectors, which are the underlying reasons for selecting YOLOv3 [31] in our thesis. It is a state-of-the-art real-time object detection system, that uses a unified network for predicting bounding boxes and class probabilities. YOLO runs the entire network on the image at once without breaking the

image into regions, making it ideal for use in real-time detection as required in autonomous vehicle applications. This object detection system is based on the regression method [32], which predicts the classes and the bounding boxes for the objects in the entire image in one run of the algorithm. Its network design is inspired by the GoogLeNet [33] model for image classification. The base model of YOLO can process images at 45 FPS, while the smaller version of the network, the tiny-yolo [34], can process at 155 FPS, at the expense of giving up on the accuracy. This smaller version is designed for quicker execution by removing some layers from the original neural network used in YOLOV3. The YOLO object detection system highly generalizable and performs well with new unexpected inputs. This unique quality makes it reliable for using it for the purpose of traffic sign detection as it is more likely to give better predictions during bad weather conditions or poor visibility. YOLO makes use of the darknet network architecture for training which will be discussed in The Darknet Architecture section.

The major innovation YOLO brought when it came about was the fact that it is capable of performing the detections in one go, which is why it is very fast and performant. While other approaches usually employ a pipeline of tasks, like passing on the image some classifier(s) to detect stuff in different locations and/or utilizing some other added methodologies. During this study, several detectors were compared, and it was found that the most impressive performance out of all the available detectors were, R-CNN [35], Faster-RCNN [36] and YOLO. Most of these detection systems are highly accurate, and at the same time fast at detection. The only drawback seen in these detectors is the amount of computational power they require. The concept behind, R-CNN and Faster R-CNN, is to make use of region of interest (ROI) proposal methods to first create potential bounding

boxes in an image and then run a classifier on these proposed boxes but for both cases, running a classifier thousands of times over an image implies thousands of neural network evaluations to produce detections. This demands many computational resources, prevents generalization and can introduce a micro-processing-delay, turning the whole system into a pseudo-real-time system. What's more, it needs a post-processing phase to refine the multiple bounding boxes, eliminate duplicate detections and rescore the boxes based on the number of objects of the scene.

It is very hard to have a fair comparison among different object detectors. Amongst the various object detector types, there are a number of aspects that affect the performance, such as the type of network, input image size, training dataset, data augmentation, etc. To make matters even worse, the technology evolves so fast that comparison becomes out of date very quickly. Hence, for real-life applications, it is best to choose a type of detector that suits best for your requirement, making a choice to balance accuracy and speed [37]. The goal of this thesis is recognition of traffic signs in Gazebo, which requires a lot of computation. Considering these facts, I made a difficult choice and decided to go with YOLOv3 for object detection and recognition.

2.5.1.1 Darknet Architecture

YOLOv3 uses a variant of Darknet, called the Darknet-53[38], which is the backbone of the system. Darknet-53 mainly consists of residual blocks and 53 convolutional layers. A residual block is a block consisting of a pair of 3x3 and 1x1 convolutional layers together with a shortcut connection. A full overview of the Darknet-53 architecture can be found in Figure 9. For the task of detection, 53 more layers are

stacked onto it, giving us a 106 layer fully convolutional underlying architecture for YOLOv3. The network down samples the image by a factor called the stride of the network [39], meaning if the stride of the network is 32, then an input image of size 416 x 416 will yield an output of size 13 x 13. Generally, the stride of any layer in the network is equal to the factor by which the output of the layer is smaller than the input image to the network.

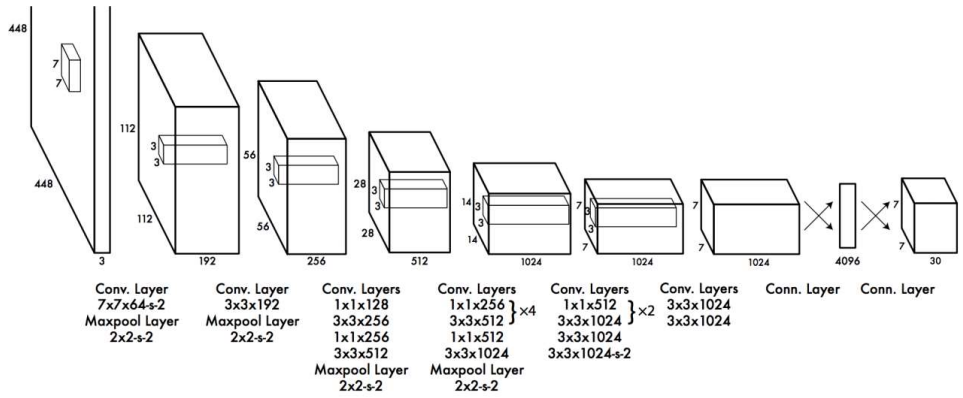

Figure 9. Darknet 53 Network Architecture [38]

To perform the detection procedure, the system divides the input image into an S x S grid and a single convolutional neural network (CNN) predicts simultaneously B multiple bounding boxes for each grid cell, their confidence value and their class probabilities for each region. The B bounding boxes are weighted by the predicted probabilities and the non-maximal suppression technique is applied to fix multiple detections when objects near the border of multiple cells are localized by these near-cells. Delving into the Yolo network design, its architecture is inspired by the GoogLeNet model for image classification but instead of having 22 layers deep CNN, it has 24 followed by 2 fully connected layers. To reduce the features space from preceding layers, 1 x 1 convolutional layers are alternated between layers.

In this section, I have only provided a very brief explanation of the network architecture, keeping in mind the scope of this Thesis. This should be sufficient to understand how the object detection works in YOLOv3. The network architecture has been kept the same as changing any layers in the network may result in an unforeseen effect on the overall performance, however, I have made some modifications in the training parameters. These modifications and the reasons for these have been explained in the Training for Traffic Sign section.

2.5.1.2 Performance Matrix

Now that the network is designed is known, it is necessary to understand how to evaluate the performance of the system. For this, an understanding of the performance matrix for the system is needed. Here I have provided the details on the same.

1. Loss function: YOLO predicts multiple bounding boxes per grid cell as seen in the Darknet Architecture section. YOLO uses a sum-squared error between the predictions and the ground truth to calculate loss. The loss function comprises:
    a. Classification Loss: This function calculates the error in classifying an object. If an object is detected, the classification loss at each cell is the squared error of the class conditional probabilities for each class.
    b. Localization Loss: The localization loss measures the errors in the predicted boundary box locations and sizes. Here only count the box responsible for detecting the object. We do not want to weight absolute errors in large boxes and small boxes, meaning, a 2-pixel error in a large box is the same for a

small box. To partially address this, YOLO predicts the square root of the bounding box.

c. Confidence loss: This function provides a measure of the objectness of the box. And the objectness score is predicted using logistic regression. It is 1 if the bounding box prior overlaps a ground truth object by more than any other bounding box prior. Only one bounding box prior is assigned for each ground truth object.

2. Intersection Over Union: Intersection over union is a metric evaluating the overlap of two bounding boxes. It calculates the ratio between the intersection between predicted bounding box and ground truth, over their union. This measure helps find how accurate the bounding boxes are predicted.
3. Recall and Precision: Recall is used to calculate how well a model finds all the positives and precision measures how accurate the model is, how well does it find all the relevant objects. For this, it is necessary to understand what True Positives (TP), False Positives (FP) and False Negatives (FN) mean. The True Positives are the number of positive observations that were predicted as positive. The False Positives are the number of negative observations that were falsely predicted as positive and the False Negatives are the number of positive observations that were falsely predicted as negative. Recall is the ratio of TP over the sum of TP and FN, whereas Precision is measured as the ratio of TP over the sum of TP and FP.
4. Mean Average Precision (mAP): Mean average precision (mAP) is an object detection metric that multiple data sets used to compare different object detectors. mAP is calculated by the area under the precision versus recall curve. Thus, mAP

is calculated by interpolating all points on the precision versus recall curve and then calculating the area under the curve.

5. F1 Score: The F1 score is a weighted average of precision and recall. The F1 score a model can achieve will fall in between 0 and 1, where 0 is the minimum and 1 is the maximum. The higher the score the better the model performs.

2.5.1.3 YOLO/Darknet Installation

As mentioned in the earlier sections, Darknet is a framework to train neural networks, it is open source and written in C/CUDA and serves as the basis for YOLO. Darknet is used as the framework for training YOLO, meaning it sets the architecture of the network. Before training the network for traffic signs, using darknet, check for all the prerequisites. The requirements are as mentioned below and can be downloaded from the links provided with it

CMake >= 3.8 (https://cmake.org/download/]

CUDA 10.0: (https://developer.nvidia.com/cuda-toolkit-archive]

OpenCV < 4.0: (https://opencv.org/releases.html]

cuDNN >= 7.0 for CUDA 10.0 (https://developer.nvidia.com/rdp/cudnn-archive]

GPU with CC >= 3.0: (https://en.wikipedia.org/wiki/CUDA#GPUs_supported]

Once all the requirements are installed, clone the following git repository, build and run darknet from git repository as follows:

```
$ git clone https://github.com/AlexeyAB/darknet.git
```

```
$ cd darknet

$ make
```

2.5.2 Network Parameter Modifications

To train YOLO for traffic signs, the neural network parameters need to change according to the data set. First, the configuration of the net was adjusted to the case at hand. The first step was changing the batch size from 1 to 64. This variable defines the number of samples (images) that will be propagated through the network at each iteration. Choosing a batch size number smaller than the total amount of training images has some advantages: less memory space is required during the training procedure because the network is trained with fewer samples and it is also trained faster because the weights are updated after each propagation. On the other hand, the smaller the batch variable is, the less accurate the estimation of the gradient will be. By default, a stochastic batch (batch size equal to 1) is set but a batch size of 64 deals better with the YOLO architecture network [40]. Also, the number of maximum iterations was changed: the max_batches variable is set multiplying 2,000 by the number of classes. That is because it is considered that 2,000 iterations per class are needed to train the network properly. In our case there are 10 classes, so 20,000 iterations will be needed in order to train the model. The learning rate is left by default: 0.001. After the 80% and 90% of the maximum iterations value the learning rate will be adjusted, so the steps variable is set as '3800'. Finally, one more change was performed. The image resolution was decreased from 412x412 to 224x224 in order to increase the precision and to allow the net to detect small objects. I have explained in detail the changes made by the Training Parameters in the next section.

2.5.2.1 Training Parameters

The network parameters for darknet training is saved in a configuration file. This contains a detailed illustration of the block by block network layout. To train for Traffic Signs, it is required to configure these parameters to suit the needs, that is to detect and recognize traffic signs in a simulated environment. Start by creating an empty file traffic_sign.cgf. This file is saved inside the directory /traffic_signs/darknet/cfg. Then copy the contents of the file yolov3.cfg to the traffic_sign.cgf configuration file and make the changes in the layers and other training parameters such as batch-size, filters, max_batches, etc as shown in Table1. Here I have explained the changes made in the traffic_sign.cfg file and why these changes are necessary for training of the traffic signs in our thesis.

Batch size: The training process of YOLO updates the weights iteratively by learning from the mistakes it makes during training. It will take a lot of computational power and training time if all images in the dataset are used at one iteration to update the weights, so a small subset of the dataset is used in one iteration. This is the batch size. For our small dataset, I have set 'batch = 64'. During training, our network generates new weights after iterating over 64 images and the training results were acceptable.

Subdivisions: The training process highly depends on the GPU (Graphical Processing Unit) capability and memory. Training process can get crashed if the GPU runs out of memory, fortunately, Darknet allows the user to set the subdivision parameter, which processes only a fraction of the batch size in one iteration and GPU memory and computation is conserved. I have trained traffic signs with 'subdivisions = 8'.

Classes: The classes are the number of categories that will be detected. I have created 10 different traffic signs for our thesis, so the number of classes is 10. Set 'classes =10'.

Filters: Filters represent the number of convolutional kernels in a layer. The filters depend on the classes and the number of masks, which are indices of the anchors. Since I am not making any changes to the number of anchors, the filter size can be calculated by, filters= (classes + 5) x3. In this case, filters are 45. Change the filter for each of the [convolutional] layers that are seen before all 3 [yolo] layers. Set 'filters = 45'.

Momentum: The changes in the weights in every iteration are a result of the learning. The rate of change of the weights is dependent on the momentum of the network. Extremely large or extremely small fluctuations will result in slower learning. The training speed can be improved if it is set to optimum momentum. I have trained with 'momentum=0.9'.

Decay: The decay parameter also controls the rate of change of weights. I have set the decay to the default value, i.e. 'decay = 0.0005'. Decay is mainly used to address the case of overfitting in the training.

Max_batches: To complete the training I need to specify how many iterations need to be completed. This can be done by setting the max_batch parameter. For good results, the number of iterations should be relative to the number of classes being trained for classification. According to the creators of Darknet, it is advisable to train for at least 2000 times the number of classes. In this case, there are a total of 10 classes, so I trained such that a safe number of iterations is reached, with 'max_batch = 40000'.

Image parameters: A crucial parameter for training is the size and type of image used in the dataset. The input image has width, height, and number of channels. The width and height specify the pixel size of the image and the channels are color information. The larger the image size, it will take longer to train. I have trained with images of 3-channel RGB and size of 224X224 pixels, i.e. I have set 'width=224', 'height=224' and 'channels=3'

Learning parameters: The learning rate controls how aggressively the learning takes place of the current iteration. At the start of the training, there is very little information about the features that the network is learning, hence the learning rate needs to be high and as the network starts learning many features, the rate of learning gradually can be gradually decreased. I have used 'learining_rate=0.001'.

Steps: The steps parameter controls the rate at which the learning rate for the training process decreases. Steps represents the number of iterations after which the learning rate will change, in our training, I use the default value for steps. Set 'Steps=3800'.

There are many other parameters such as scales, burn-in, angle, saturation, etc. which can be changed depending upon the need for the training process. Changing these parameters sometimes have a significant impact on the way the network learns. All these features have been kept default and no changes are made. Table 1 shows the adjusted parameters for the configuration used for our training process.

Table 1. Configuration values of the network training

Parameter	Value
Batch size	64
Subdivisions	8
Width	224
Height	224
Channels	3
Momentum	0.9
Decay	0.0005
Saturation	1.5
Exposure	1.5
Hue	0.1
Learning Rate	0.001
Max Batches	40000
Steps	3800

2.5.3 Data Acquisition

One of the most strenuous issues that is faced during the training of neural networks is getting the right data in the right format. The right type of data means collecting data that correlates to the type of training at hand. Since our task was to detect and classify traffic signs inside a simulation environment, I needed to have training images that are collected from a simulation environment. Hence, I chose to create our own data set. The data for our training is collected using the simulated front camera of the simulated vehicle model.

The images are collected using a python script which saves the camera data using OpenCV. To collect the images, launch the world in Gazebo, with all the traffic signs placed at different locations as it is seen in an urban environment. Once the world and the Vehicle model are launched, run the ros node data_collection.py in a new terminal. The python script creates a ROS node. This ROS node subscribes to the camera topic, wherein the camera topic publishes the camera image at 30Hz. The script makes use of OpenCV to save the ROS images and the images are saved in .jpg format. The rate at which the images can be changed inside the python script. I have made use of the timestamp in order to provide a distinct name to each image. This script also collects the steering angle of the vehicle, but I have commented out this part as I do not require the steering angles to train for Traffic sign detection. The steering angle will be used for training the vehicle for lateral control and this is explained in the Integration with Gazebo section. The location to save the collected images is provided along with the script in the terminal.

The python script for collection of data is provided in the ROS package which can be downloaded from GitHub [55]. The data collection can be achieved by following the simple steps provided.

1. Launch the simulated environment in which I have added the traffic signs.

```
$ rosslaunch car_demo mcity_modified.launch
```

2. In a new terminal, launch the python script using the following command,

```
$ cd catkin_ws
```

$ rosrun data_collection data_collection.py traffic_signs/data/nikhil This will run the script and save the images. If the folder does not exist a new folder is created with the time stamp(yyyy-mm-dd-hh-mm-ss) set as the name inside the path provided.

3. Drive the Vehicle in the environment, specifically in the regions where the signs have been placed to collect the desired images.
4. Open the folder where the images are saved and delete all the images with no signs. Also, delete the duplicates.
5. Copy the images to traffic_signs/darknet/data/obj

For better training for classification, the dataset for training must be diverse, containing a variety of different types of images. In order to achieve this, the image for the signs needed for training was collected at various distances, angles. The vehicle and the signs were placed at varying positions in terms of height, skew, etc, to have a diverse set of images. The images collected need to be filtered to remove unwanted extra images. The images that need to be deleted are mainly duplicates and the images that do not have any traffic signs. Then I need to create a text file containing the names of all the images in the final dataset. This is explained in detail in the Data Annotation section. The dataset I used for training consists of 700~1,000 images per class which gave us satisfactory results for traffic sign detection and classification. Table 2 shows the total number of images collected in each class.

Table 2. Total number of images per class for training

Class Name	No. of Images
Stop Sign	832
Yield Sign	759
Pedestrian Sign	812
Speed limit 5 Sign	703
Speed limit 10 Sign	710
Speed limit 15 Sign	823
Speed limit 20 Sign	809

Speed limit 25 Sign	813
Right Turn Sign	717
Left Turn Sign	726

2.5.3.1 Data Annotation

A critical part of training neural networks for classification of any category is data annotation. It is simply the task of labelling the data for the learning neural network to recognize recurring patterns in the annotated data. The data annotation for our dataset I have used 2D bounding boxes, which are imaginary boxes drawn on images along with the class information for each image in the dataset. The data annotation for training YOLO needs to be in the following format,

<class-id> <center-x> <center-y> <width> <height>

The <class-id> represents the class of the object, <center-x> and <center-y> are the x and y coordinates of the center of bounding box, the <width> and <height> are the width and height of the bounding box.

There are several tools available for data annotation like 'LabelImg', Labelbox, BBox-Label-Tool', 'Yolo Mark', Gengo AI, MATLAB Image ground truth labeler, etc., The tool used for this thesis is 'Yolo Mark' and it is shown in Figure 10. This tool was chosen as it saves the label data in the format required by default. To install, build and run Yolo Mark from the git repository, perform the following steps:

```
$ git clone https://github.com/AlexeyAB/Yolo_mark
```

$ cmake .

$ make

Yolo_mark requires to have all the images in a specific location. All the images collected during the data acquisition must be copied to this folder, traffic_signs/Yolo_mark/x64/Release/data/img.

Next, 3 special files are required. They are of the extension, .data, .names, and .txt. The contents of these files will be described in the Training Setup section.

Finally, run the following command, to open a GUI and start labeling the data.

$ ./linux_mark.sh

The annotation using Yolo Mark will generate a .txt file, which is the label file for each image in the same directory and with the same name as the image.

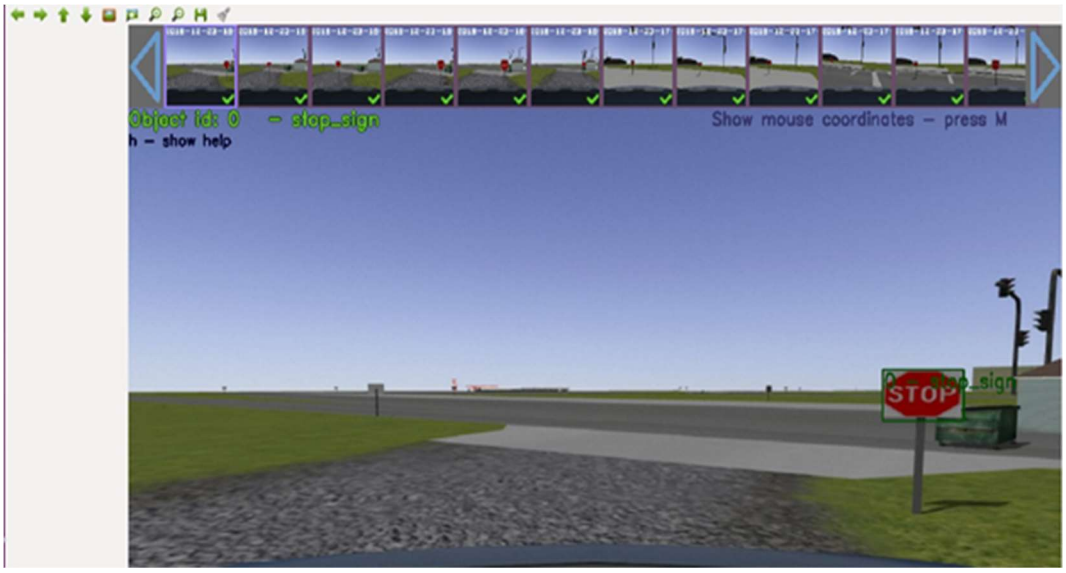

Figure 10. Data annotation in Yolo Mark.

2.5.4 Training for Traffic Sign

This section particularly deals with training the network to detect and recognise traffic signs. It consists of training setup, training process and the training results. Here we have also provided all the necessary files and datasets for recreating the results obtained.

2.5.4.1 Training Setup

The darknet package that I have used for training YOLO for custom object detection needs information regarding the specifications of the data that needs to be trained. These include the information about the classes, the names of the classes, training and validation datasets along with the relevant path to their locations inside the package. In this section, it is seen what these files are, how to create these files and what locations these files need to be saved.

Training and validation list: These files have the list of the files that are to be used for the training and validation. It has a similar function to that of a .csv file which is used for training most of the neural networks. Here, specify the list of images that will be trained. Start by creating an empty text file with the name traffic_sign_train.txt. And save this file inside the directory /traffic_signs/darknet/data/obj. Now populate this file with the list of images collected and labeled. A sample of the contents of the traffic_sign_train.txt is seen below.

```
data/obj/2018-12-10-00-16-34-023577.jpg
data/obj/2018-12-10-00-16-35-032199.jpg
data/obj/2018-12-10-00-16-36-041377.jpg
data/obj/2018-12-10-00-16-37-050892.jpg
```

data/obj/2018-12-10-00-16-38-060369.jpg

data/obj/2018-12-10-00-16-39-070252.jpg

data/obj/2018-12-10-00-16-44-115532.jpg

data/obj/2018-12-10-00-16-47-141772.jpg

data/obj/2018-12-10-00-16-50-173511.jpg

Inside the text file, each line needs to contain the image name along with the path to the directory where the images are saved. Failing to do so will lead to image not being found for training and it will throw errors.

Names file: The names file contains the names of the classes for which the detector will be trained. I started by creating a file Traffic_sign.names and save it inside the directory, /traffic_signs/darknet/data. In this file, I have added all the class names that I want to detect in this system. The contents of traffic_sign.names are seen below. Keep in mind that each class name is written in a new line as to not mix-up with the classes.

stop_sign

pedestrain_walk

parking

right_turn

left_turn

yield

speed_10

speed_15

speed_20

Speed_25

Data file: The data file provides information about all the necessary requirements for the training. It specifies the number of classes and provides the absolute paths for the Train list, Validation list, and the Names file. The classes specified in this file must match the total number of classes in the Names file. The last line in the data file is the path to the backup directory. The intermediate weights generated during the training process are stored in this directory. Start by creating an empty file traffic_sign.data and save it inside the directory, /traffic_signs/darknet/data. Now populate the file with all the information mentioned above.

```
classes= 10
train  = data/traffic_sign_train.txt
valid  = data/traffic_sign_train.txt
names = data/traffic_sign.names
backup = backup/
```

The last step before the training is to download pre-trained weights for the convolutional layer. This can be done using the command,

```
$ wget https://pjreddie.com/media/files/darknet53.conv.74
```

The pre-trained weights are needed for training even though the weights may not contain any information on objects that you are trying to detect. Using a pre-trained model from convolutional weights will learn faster during training as it will not have to start learning from scratch. These weights need to be saved into the directory /traffic_signs/darknet.

2.5.4.2 Training

The last step to train is to add the images for the training. These images must be added in the directory, /traffic_signs/darknet/data/obj. For training the network, the images

must be accompanied with the annotation files generated in the Data Annotation section. Take utmost care as to not have any missing annotation files, else it may cause errors for training.

Finally, start the training using the following commands:

```
$ cd darknet
```

```
$ ./darknet detector train data/traffic_sign.data cfg/traffic_sign.cfg darknet53.conv.74
```

For training with mAP(mean average precision) or to save the train.log, add '-map' flag or '-train.log' flag along with the command for training. Weights are preset to be saved after every 100 iterations in the directory /backup. This allows us to stop training at any point and restart it using the latest weights. The training is set to stop after 40,000 iterations are completed, but the training can be manually stopped if the average loss(error) does not decrease for many iterations. As mentioned earlier the model needs to be trained for 2,000 iterations per class, but not less than 4,000 iterations in total.

In my training, I stopped the training manually upon reaching 14,000 iterations, as the average loss value was not showing any significant change. They may not be the case with other users. The training may have a lot of unknown effects from the type of system used, and the type of data used.

2.5.4.3 Training Results

Stopping the training process trying to avoid the underfitting or overfitting phenomena, is not a trivial task. Darknet's site details that 2,000 iterations per class are

usually enough, but if the average loss during the training does not decrease significantly between consecutive iterations it should also be considered the training is nearly complete. It is also remarked that it is important to get the weights at the Early Stopping Point and in order to choose them the last few weights files are compared. This comparison is performed by evaluating the validation set previously uploaded and getting the weights file that has the highest mean Average Precision (mAP) and the highest Intersection over Union (IoU) value. mAP is the mean of the average value of the precision across all recall values and is an evaluation metric of object detectors that measures the overlap ratio between the ground truth and the predicted boundaries. These are popular metrics to measure the accuracy of the object-detection algorithms and have been discussed in the Performance Matrix Section. Table 3 shows the results of the evaluations of the 10 created weights files (one after every 1,000 iterations) after a total of 22 hours of training.

Table 3. Table comparing the obtained weights from the training procedure

Weight Files	mAP	IoU
2000 iterations	59.01	51.01
3000 iterations	67.59	76.32
4000 iterations	86.23	83.50
5000 iterations	92.56	85.39
6000 iterations	98.78	85.67
7000 iterations	99.04	86.43
8000 iterations	99.12	86.21

Taking the results shown in Table 3, the weights file corresponding to the 7000th iteration has the highest maP and IoU values. These weights are the chosen ones to perform the detection process in the Gazebo simulator. All these weights will be tested on a test

data set analyzing their performance and other metrics such as the Precision, the recall, and the F1-Score. Moreover, True Positives (TP), False Positives (FP) and False Negatives (FN) rates are also discussed.

On completion of the training, apart from mAP and loss, the weights were tested on new data to check for results of detection. The testing can be done using the command:

```
$./darknet detector test data/Traffic_sign.data cfg/traffic_sign.cfg traffic_sign_XXXX.weights
```

A test set of 1346 images extracted from Gazebo was created alongside the training set. The total number of images added in each set can be seen in table 4.

Table 4. Total number of images per class for testing

Class Name	No. of Images
Stop Sign	148
Yield Sign	129
Pedestrian Sign	137
Speed limit 5 Sign	124
Speed limit 10 Sign	132
Speed limit 15 Sign	145
Speed limit 20 Sign	135
Speed limit 25 Sign	142
Right Turn Sign	126
Left Turn Sign	128

Apart from the mAP and the IoU, there are three more metrics there were chosen for evaluation of the weights, to be used in the final package. These metrics were the

precision, the Recall, and the F1-Score. The precision measure is the ratio of correctly predicted positive observations to the total predicted positive observations. The recall metric is the proportion of how many of the total amount of positives observations have already been correctly predicted as positive. Finally, the F1-Score is a measure used to seek a balance between the Precision and Recall, in other words, it is the weighted average of these metrics. To get these measures, three rates must be obtained: True Positives (TP), False Positive (FP) and False Negative (FN) rates. A comparison of these values is seen in Table 5

Table 5. Performance Metrics obtained with the test data set

Weight File	Precision	Recall	F1-Score	TP	FP	FN
2000 iterations	0.74	0.74	0.74	1331	462	470
3000 iterations	0.82	0.81	0.82	1628	312	223
4000 iterations	0.88	0.90	0.87	1719	127	96
5000 iterations	0.92	0.90	0.90	1778	65	34
6000 iterations	0.95	0.97	0.95	1798	5	4
7000 iterations	0.98	0.99	0.98	1799	2	2
8000 iterations	0.98	0.98	0.98	1798	7	9

The data from the above table shows the results from the metrics which have previously discussed. These results were obtained from the weight files produced during the training phase. Analyzing these results and the ones obtained in the validation test (Table 3), it can be seen how the values regarding the weights file on the 7000th iteration

are the best ones. This justifies the reason for the weight on the 7000th iteration chosen to be integrated into the object-detection system. In Figure 11 shows the traffic signs recognized within a bounding box, along with the probability in the class predicted.

Figure 11. Traffic signs recognized in Gazebo

It is very important to verify the results by applying it where it is needed, and check the output. Hence I placed some random traffic signs in front of the vehicle model and checked for recognition by feeding the camera image to the network. As it can be seen in the above figure, the network is trained very well as it is recognized the yield sign, the stop sign, and the speed limit sign and predicts a 100% probability for each. This validates our results and brings an end to this section.

2.6 Integration with Gazebo

As it was very clearly mentioned in the beginning that the purpose of this thesis was to develop a system capable of detecting and recognizing traffic signs. Not only that, I intended to provide the ability for anyone to be able to recreate the work done and further develop this research. So far, I have shown you the methods to develop all the essential elements that are a part of this system. The final step to achieve the goals of this thesis is to put together all these elements into one system. This chapter explains in detail, the procedure needed to be followed, in order to achieve detection inside a simulation environment.

2.6.1 Darknet_ros

The heart of this system lies inside the traffic sign detection system. In order to achieve the detection and recognition in a Gazebo simulation environment, I needed a ROS Package, that is capable of running the darknet architecture. This is achieved by making use of a ROS package called the darknet_ros [41]. This an open-source package that can be readily cloned from the GitHub. The following dependencies are needed in order to successfully build this package. The links for download are provided:

1. OpenCV. [https://opencv.org/]

2. Boost. [https://www.boost.org/]

The procedure for successfully installing these libraries can be found on the official websites for OpenCV [42] and Boost [43]. These are also open sourced and are freely

available for anyone to download. Once these have been setup, start by building the ROS package using the following commands.

1. Clone and build the darknet_ros package from Github.

```
$ git clone --recursive git@github.com:leggedrobotics/darknet_ros.git
```

2. Build the ros workspace including darknet_ros package.

```
$ cd ..
```

```
$ catkin_make -DCMAKE_BUILD_TYPE=Release
```

This Darknet package can be run on the CPU and GPU, but the GPU is about 500 times faster than the CPU. In order to run the GPU version of Darknet you will require an Nvidia GPU and you will have to install CUDA [44]. During the installation process, the CMakeList.txt file will automatically detect for CUDA installation and build for CPU or GPU version accordingly.

2.6.1.1 Detection messages

As mentioned earlier the darknet_ros is a ROS package developed for object detection using YOLO in any system that runs on top of ROS. The package takes the input from the camera image of the object detection system. It may be a robotic camera or a simulated camera. This is achieved by subscribing to the camera data as a ROS node. In this Framework, I have provided the essential knowledge required to publish and subscribe to a ROS topic. The darknet_ros package publishes three topics, which contain all the necessary information required for detection and recognition. A brief description of the

information contained in the messages, such as the topic name, message structure, data type, etc is provided below.

1. Object_detector: This publishes the information on the number of objects detected. It saves the data type used is a standard ROS of the type integer, std_msgs:: Int8.

2. Bounding_boxes: This message is a custom-built message which publishes an array of bounding boxes that gives information about the position and size of the bounding box in pixel coordinates. This message is a struct, and has the following declaration,

a. string Class;

b. float64 probability;

c. int64 xmin, ymin, xmax, ymax;

Bounding box and its coordinates have already been explained in detail in the Data Acquisition section.

3. Detection_image: This message publishes an image of the detection image including the bounding boxes. The data type will be similar to the input of the camera image, sensor_msgs:: Image.

2.6.1.2 Traffic Sign Detection Parameters

In order to detect our custom objects, I need to provide the weights generated from the training. I have already provided the procedure to train for custom objects, and generate weights for the custom objects in the Training for Traffic sign Section. These trained

weights need to be copied over and placed in the directory: catkin_ws/src/darknet_ros/darknet_ros/yolo_network_config/weights/.

As seen previously in the section on Training Results, I analyzed and compared the output from multiple weight files and selected traffic_sign_7000.weight as the best-trained weight. I have placed this file into the above-mentioned folder. Along with the weight file. The network configuration file is also needed. This configuration file contains the details of the network parameters. I need to provide the same network parameters used for training to obtain successful detections. For training to detect traffic signs, I made use of the traffic_sign.cfg as our network configuration file. The network configuration file needs to be placed in the directory: catkin_ws/src/darknet_ros/ yolo_network_config/cfg/.

Next step is to link the cfg and the weight files to be read by the darknet_ros package. Make use of a YAML file [45] for linking. The YAML is a human-readable data-serialization language which is very commonly used for configuration files. This file contains detection related parameters such as the names of the detection classes, network configuration file, the weights file, and the threshold value. The threshold value is set between 0 and 1 and it determines the minimum probability below which the detections will not be displayed, meaning, only the detections that have a probability value over the set threshold value will be displayed to the user. In this thesis, I have set threshold value = 0.95. This file needs to be placed in the directory: /traffic_signs/catkin_ws/src/darknet_ros/darknet_ros/config. Create a new YAML file named yolo_traffic_sign.yaml and add the above mentioned details in this file. Snippet of the yolo configuration YAML file is provided below.

```
yolo_model:
  config_file:
    name: traffic_sign.cfg
  weight_file:
    name: traffic_sign_7000.weights
  threshold:
    value: .95
  detection_classes:
    names:
      - stop_sign
      - pedestrain_walk
      - parking
      - right_turn
      - left_turn
      - yield
      - speed_10
      - speed_15
      - speed_20
      - speed_25
```

2.6.1.3 ROS Parameters

To run the system in the ROS it is necessary to set up the basic communication requirements in the form of publishers and subscribers. As mentioned in the detection messages section, I need to subscribe to the camera topic, in this case I have subscribed to

/bolt/front_camera/image_raw. This topic is published by the vehicle sensor model and contains the image data of the front camera. More camera topics can always be added or the camera image can be changed from the front camera to the back camera, depending on our application. I will only be using the front camera for our purpose.

Moving on to the publishers, there are 3 topics published by this ROS package that provides the detection information. The published topics are as these:

1. Object Detector: This topic contains the information on the number of objects detected in the Camera, the topic is published with the name darknet_ros/found_object.
2. Bounding boxes: It contains an array of bounding boxes that gives information on the position and size of the bounding box in pixel coordinates. The format of the topic is taken from the bounding_box message and is already explained in the previous section. It is published under, darknet_ros/bounding_boxes. The bounding box information can be used to locate the object in the Image frame and further used for any application desired.
3. Detection Image: This is simply the camera topic taken from the vehicle sensor being published back along with the bounding box. It basically superimposes the camera image, with the bounding box coordinates obtained from the detection.

subscribers:

camera_reading:

topic: /bolt/front_camera/image_raw

```
    queue_size: 1
  publishers:
object_detector:
    topic: /darknet_ros/found_object
     latch: false
   bounding_boxes:
    topic: /darknet_ros/bounding_boxes
    latch: false
   detection_image:
    topic: /darknet_ros/detection_image
    latch: true
```

The topics seen above need to be added into a new YAML file with name ros_traffic_sign.yaml, and save this file inside the directory /traffic_signs/catkin_ws/src/darknet_ros/darknet_ros/config, along with the Yolo configuration file.

The next step is to create a launch file for launching all the necessary ROS nodes. The launch file starts new nodes, starts a master node, provide the locations for the weights file, network config file, YAML file, and also linked it to open Gazebo world

file where I have implemented the Traffic sign detection. I have provided multiple launch fine with different purposes. All the launch files are stored inside the directory, traffic_signs/catkin_ws/src/car_demo/car_demo/launch.

2.6.2 Lateral Control

Apart from Traffic Sign detection, one of the other objectives of this thesis was to make use of the detections and apply it for controlling the vehicle speed. It would not be of much use to have object detection and not apply it for any meaningful application. Hence, I also worked on Lateral Control [46], so as the vehicle can be justified to be Autonomous. There are a lot of ways to achieve autonomy, one such method is by providing the vehicle with lane keep assist and programming it to move at any speed that it detects on the road signs.

Now I have all the elements necessary to accomplish our objectives are ready for deployment. The final step, as mentioned above, is to achieve lateral control. For this, I have made use of a package called Mir_torcs [56], which was created by Prof. Jaerock Kown. The concepts underlying this work is based on Behavioral Cloning [47], for lateral motion control of autonomous vehicles. This package is open sourced and readily available on Github. A link is provided for the same. The work done in Mir_torcs is towards lateral control and also speed control and the underlying Neural Network is based on NVIDIA's Pilot Net [48]. The model is designed to learn from driver behavior. The network is fed with vehicle camera images, along with the steering and throttle values. The network learns to drive autonomously from the inputs provided for training.

Mir_torcs was designed to clone the speed and steering from the driver's behavior. I have made some modifications to the Mir_torcs package to suit our needs. Since only need this for lateral motion control, I only provided camera images with corresponding steering angles, as the speed will be set based on the specified speed limit as seen on the road signs. The certain changes were also made in the network. The reason I made these changes is that, in this thesis, I wanted to replicate the exact network architecture implemented in the NVIDIA's Pilotnet. In order to do this, I have deleted the following layers from Mir_torcs network architecture as these layers are not part of the PilotNet.

1. Max Pooling layer
2. Dropout layer

After the modifications have been made I can train the network. Since I needed an environment with a single lane, I created a new environment in Gazebo. This new environment consists of a double lane road. The steps for creating the track, data collection and training the network have been provided.

2.6.2.1 Test Track

To collect data and train for lateral control, I needed a test track. To create the test track I adopted the road segments from data speed [58]. These road segments were modified by changing the number of lanes. The modified track is a double lane road, separated by dashed white line. The lane ending is marked by yellow line. These modifications were made keeping in mind the training needs. I needed to have clear lane markings for better training of the network. The road segments can be seen in Figure 12.

These segments were added to the existing Gazebo model list so that the road segments are readily available for use.

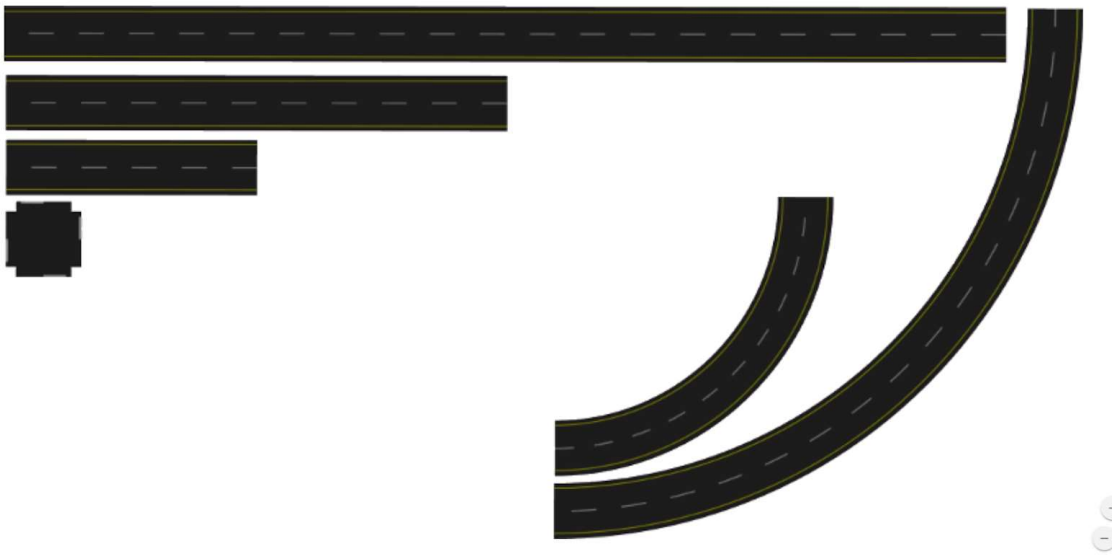

Figure 12. Road Segments for Test Track [58]

Upon adding the road segments I was able to see the road segments these inside the Gazebo Object panel. Now, all I need to do is drag and drop the segments into an empty Gazebo world. I have tried to maintain an equal number of left and right turns for better training of lateral control. The track design is completely dependent on the user needs and creativity. The final track is shown in Figure 13. I have also added the Traffic signs to this track that I modeled earlier as this track will be later used to test the application for speed control.

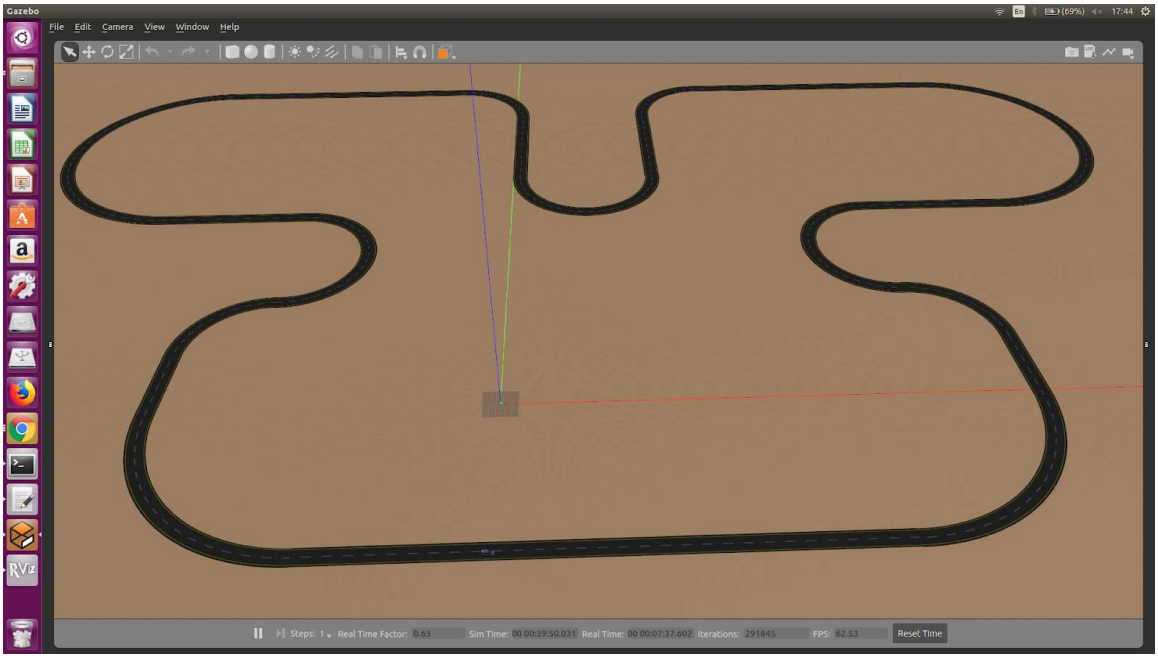

Figure 13. Final Test Track

2.6.2.2 Data collection

The data collection for lateral control requires steering input, hence I made use of Logitech G920 dual-motor feedback driving force steering wheel, accompanied with pedals, gear shifter, and car seat. This controller was selected as I needed precise steering angle input for training and it was very difficult to achieve the same using a keyboard or a regular gaming joystick.

For the collection of training images, launch the Gazebo environment which has the above-designed test track along with the Vehicle model by running the command $roslaunch car_demo test_track.launch. Then run the python script data_collection.py. using the following command, $rosrun data_collection data_collection.py traffic_signs/data/nikhil/yyyy-mm-dd-hh-mm-ss in a new terminal, along with the path to save images. This has already been used before in the Data Acquisition section. As

explained earlier in the Data Acquisition section the python script creates a ROS node that subscribes to the camera topic. I have made one more modification to the python script, here I have adjusted the camera topic to publish the camera images at 45Hz, as I will need more images of the lanes for better training. I have also enabled the feature to save the steering angle of the vehicle as it was earlier commented out for the data collection for Traffic signs. This script will save the images with the steering angle of the vehicle at that moment. The steering angle values range from -1 to 1. The steering angle values are saved in a text file along with the image name for ease of investigation. Then the vehicle was driven on the track and completed multiple laps. The driver needs to be very focused as not to make any mistakes. The driver has to maintain the lane at all times as this data will be provided to the network to learn from the driver's behavior. This means, if the driver makes a mistake in driving my leaving the lane, then the network will do the same and will not be able to maintain lane-keeping efficiently.

Finally, once the data collection is done, pre-process the data before feeding it to the network for training. For pre-processing, I carefully browsed through all the images and deleted the images in which I saw driver errors, such as lane departures, oversteer, and under-steer. The data_collection.py script has been modified to facilitate for image cropping.The images are cropped to eliminate the redundant information from the image, such as the sky and the vehicle bonnet, which is covered by the camera. There were two reasons for this. Firstly, these regions that have been cropped out provide no valuable input for training. And secondly, cropping will reduce the image size which will eventually help reduce the training speed. A comparison of the two types of images is seen in Figure 14.

Figure 14. Camera Input (left) and the cropped area(right)

2.6.2.3 Training For Lateral Control

For training the model, I followed the process described in the Mir_torcs package. Before training begins, it is suggested to change the training parameters. These changes can be made inside const.py. The parameters include the batch size, subdivisions, image size, etc. These parameters are similar to the parameters used in YOLO training and are already explained in the Network Modifications section. For the simplicity of training, I will keep the training parameters to the default values provided by the developers.

Finally, I can start training. The script used for training is train.py. Care needs to be taken so that the training data and csv file that contains the steering angles need to be placed in the same folder, and the file location needs to be provided for training. The training can be started with the command.

```
$ cd neural_net
```

```
$ python train.py traffic_signs/data/nikhil/yyyy-mm-dd-hh-mm-ss
```

Upon completion of training, the script outputs the weights file and the network model file along with a graph. The weight file has an extension of .h5 and contains information on the training. The weights have already been explained in the Training for Traffic signs section. The second file generated is a .json file, which contains the information of the network mode. The generated files will have the same name as the folder inside which the images are saved. These files will be used in later sections for running the vehicle autonomously. These files are saved in the same folder where training data is saved. The graph generated contains information of the RMSE(root mean squared error) [49], against the number of epochs/ iterations. RMSE is a very frequently used measure to determine the training quality [50]. This graph can be analyzed to determine the quality of the training. The network is considered to be trained well when the training and validation error eventually begin to converge with an increase in the iterations. The results of our training can be seen in Figure 15.

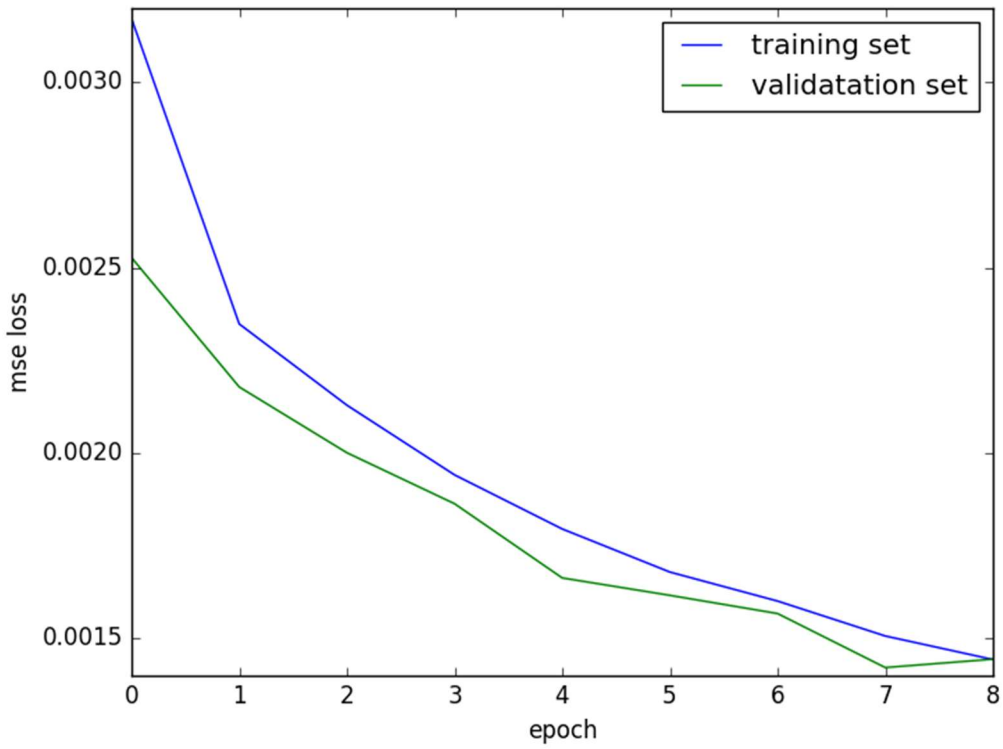

Figure 15. RMSE vs Epochs

This ends the lateral control section and in the next section, I will integrate all the elements that I created up until this point.

2.6.3 Speed Controller

In the objective section of this thesis, it is disclosed that the main motivation that leads to this thesis is to train a system to be able to detect and recognize Traffic signs in Gazebo and take decisions accordingly. So far I have provided details on creating a Gazebo simulation, training for Traffic sign detection and recognition, and also a lateral controller. These are all the necessary components required to create an Autonomous vehicle capable of navigating based on the input taken from the traffic signs. The last component necessary is a Speed controller that will take the predictions from the traffic sign detector, lateral

controller, make decisions for vehicle control and publish these into Gazebo. This section explains in detail how to accomplish this task. I created a python script called, run_neural_darknet.py with an aim to complete this task.

The run_neural_darknet.py script subscribes to the topic /bolt/front_camera/image_raw, which provides the data from the vehicle front camera. The camera images will be used by the lateral controller, hence I need to crop and resize the images to match the images provided for training of lateral controller. The image processing is done by calling the script Image_converter.py. This contains CvBridge [51], which converts between ROS Image messages and OpenCV images. The images will be saved in the variable name Image for later use.

The processed images are now ready for use by drive_run.py. The drive_run script takes two inputs, image data and training weights. The first input provided is the Image variable that was generated from the Image_converter.py. The drive_run script will load the model, run the weights and predict the steering angles. The steering angle values are saved with the variable name prediction. The prediction values will be published via a ROS node. The published values are then used by the vehicle controller for lateral control.

Finally, the run_neural_darknet.py subscribes to the topic /darknet_ros/bounding_boxes. As explained in the Detection messages section, this topic publishes the class, probability, and the bounding box information. The script saves the information in two variables, Class and Bounding_box. Based on the input predicted class, the script publishes the throttle and brake values as a ROS node. The throttle and brake values are hardcoded for each detected sign inside the script. For every prediction, the Class

variable is matched against the preset throttle and brake values and finally, sent to the vehicle controller for driving the vehicle. Apart from controlling the speed, the script also prints the class name and warning messages for all classes. The warning messages are printed within the terminal.

All the components need to be run simultaneously to achieve autonomous driving. Here are the steps needed to be followed.

1. Launch the Gazebo simulation environment using the command,

   $ roslaunch car_demo test_track.launch

2. Launch the Traffic sign Detection: This step will launch the Traffic sign detection system. Use the command,

   $ roslaunch darknet_ros traffic_sign.launch

3. Run the run_neural_darknet.py script: This step will run the python script explained above. The training weights and default throttle value need to be provided as arguments to run the script. Use the command,

   $rosrun run_neural_darknet.py run_neural_darknet.py /traffic_signs/data/nikhil/yyyy-mm-dd-hh-mm-ss 0.1

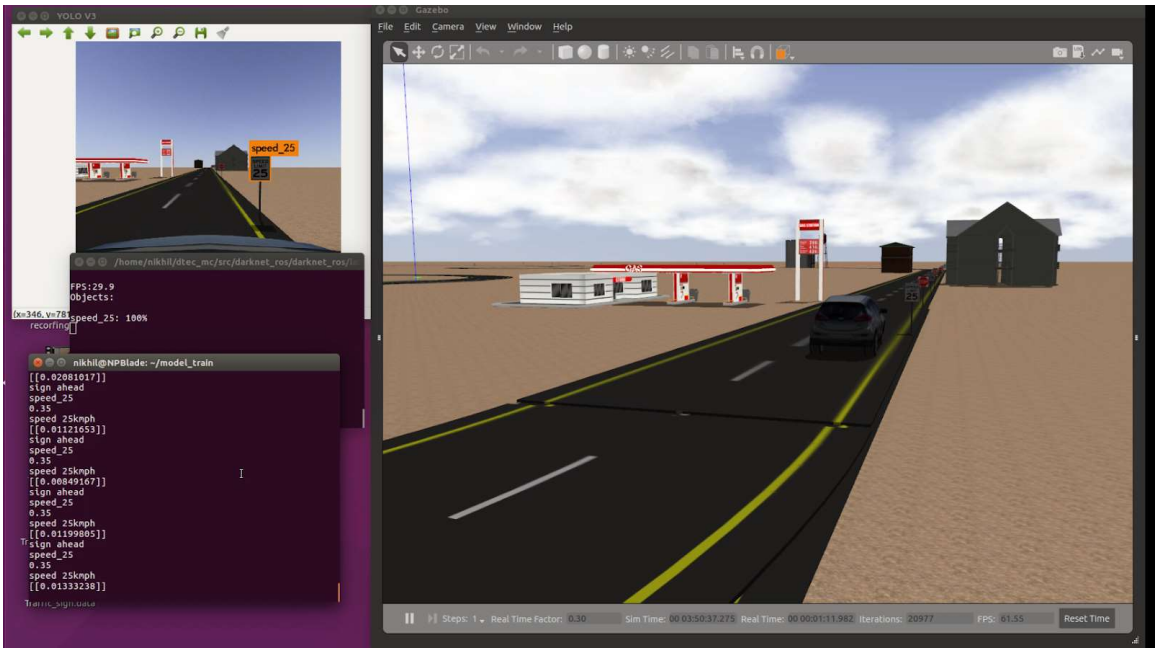

Figure 16. Traffic Sign Recognition and Speed Control

This completes all the procedures and steps needed to make this thesis work. I now have a Simulation environment with signs and a vehicle model, sign recognition using YOLO, vehicle lateral control and a speed control application. The next section is a brief discussion of the conclusion and future works.

CHAPTER 3. CONCLUSION

Even though there have been many studies and numerous projects completed regarding the traffic sign detection and recognition, needless to say with satisfactory results, none of these projects were integrated on Gazebo. This simulator is supported by a large community of users and researchers who constantly contribute to the development of this platform but, in spite of this, no Gazebo labeled images data set exist. For performing this thesis, a real-time CNN was trained in order to detect and recognize traffic signs. For this, three data sets were created to train, validate and test the network. These images can be freely downloaded at the GitHub site for use of anyone who is interested to further continue working towards developing this thesis. The dataset provided contains the same amount of labeled and unlabeledf images, with the purpose to make the model as robust as possible to False Positive cases.

On completion of training, I obtained multiple weight files. These weight files were compared, and the files with the highest mAP and IoU values were selected. The trained model was then integrated with the Gazebo simulator, which got the images from a camera sensor mounted on the simulated model of the Chevy Bolt. Apart from the detection, one other application was implemented, the Speed Controller, which deals with the application of Brake/Throttle depending on the detected sign. This was achieved with the use of a Python Script that directly sends commands to the Vehicle Control module.

Analyzing the performance and the overall results obtained in the thesis, the following conclusions can be extracted. First, the initial research objective of traffic sign detection and recognition inside a simulated environment was achieved. In addition to this, vehicle

speed control was also implemented, providing a cognitive driving ability. Moreover, this thesis highly contributes to the autonomous research community. The performance of this thesis is encouraging and has implications for further research on self-driving studies. Particularly, implications in order to make research on the traffic sign recognition field and the applications that can be deployed to help the driver with the purpose to make the driving process easier and safer.

CHAPTER 4. REFERENCES

[1] Çiftçioğlu, Özer, and Sevil Sariyildiz. "Data Sensor Fusion for Autonomous Robotics." *Serial and Parallel Robot Manipulators - Kinematics, Dynamics, Control and Optimization*, March 30, 2012. https://doi.org/10.5772/33139.

[2] "Driver Assistance Technologies | NHTSA." Accessed November 23, 2019. https://www.nhtsa.gov/equipment/driver-assistance-technologies.

[3]Intelligence, Business Insider. "10 Million Self-Driving Cars Will Be on the Road by 2020." Business Insider. Accessed November 23, 2019. https://www.businessinsider.com/report-10-million-self-driving-cars-will-be-on-the-road-by-2020-2015-5-6.

[4]Bhadani, Rahul Kumar, Jonathan Sprinkle, and Matthew Bunting. "The CAT Vehicle Testbed: A Simulator with Hardware in the Loop for Autonomous Vehicle Applications," April 12, 2018. https://doi.org/10.4204/EPTCS.269.4

[5]"Dosovitskiy - CARLA An Open Urban Driving Simulator.Pdf." Accessed November 23, 2019. http://proceedings.mlr.press/v78/dosovitskiy17a/dosovitskiy17a.pdf.

[6]."DataspeedInc / Dbw_mkz_simulation — Bitbucket." Accessed November 23, 2019. https://bitbucket.org/DataspeedInc/dbw_mkz_simulation/src/default/.

[7]"ROS/Introduction - ROS Wiki." Accessed November 23, 2019. http://wiki.ros.org/ROS/Introduction.

[8]Mazzari, Vanessa. "ROS - Robot Operating System." *Génération Robots - Blog* (blog), March 26, 2016. https://www.generationrobots.com/blog/en/ros-robot-operating-system-2/.

[9]"Catkin/CMakeLists.Txt - ROS Wiki." Accessed November 23, 2019. http://wiki.ros.org/catkin/CMakeLists.txt.

[10]"O'Kane - 2014 - A Gentle Introduction to ROS.Pdf." Accessed November 23, 2019. https://cse.sc.edu/~jokane/agitr/agitr-letter.pdf.

[11]"WPI-Robotics-SolidWorks-to-Gazebo.Pdf." Accessed November 23, 2019. https://blogs.solidworks.com/teacher/wp-content/uploads/sites/3/WPI-Robotics-SolidWorks-to-Gazebo.pdf.

[12]"Open Dynamics Engine." Accessed November 23, 2019. https://www.ode.org/.

[13]"Open Dynamics Engine." Accessed November 23, 2019. https://www.ode.org/.

[14]"SDF Specification." Accessed November 23, 2019. http://sdformat.org/spec.

[15]“Gazebo : Tutorial : Gazebo Components.” Accessed November 23, 2019. http://gazebosim.org/tutorials?tut=components&cat=get_started.

[16]“Urdf - ROS Wiki.” Accessed November 23, 2019. http://wiki.ros.org/urdf.

[17]“Rviz - ROS Wiki.” Accessed November 23, 2019. http://wiki.ros.org/rviz.

[18]“Introduction to URDF — Industrial Training Documentation.” Accessed November 23, 2019. https://industrial-training-master.readthedocs.io/en/melodic/_source/session3/Intro-to-URDF.html.

[19]“Osrf / Gazebo_models — Bitbucket.” Accessed November 23, 2019. https://bitbucket.org/osrf/gazebo_models/src/default/.

[20]*Understanding the Gazebo Plugins - Mastering ROS for Robotics Programming - Second Edition*, 2018. https://subscription.packtpub.com/book/hardware_and_creative/9781788478953/7/ch07lvl1sec68/understanding-the-gazebo-plugins.

[21]“Gazebo : Tutorial : ROS Overview.” Accessed November 23, 2019. http://gazebosim.org/tutorials?tut=ros_overview&cat=connect_ros.

[22]*Osrf/Car_demo*. C++. 2017. Reprint, Open Source Robotics Foundation, 2019. https://github.com/osrf/car_demo.

[23]Zhang, Changfu, Zhuangde Jiang, Dejiang Lu, and Taian Ren. “3D MEMS Design Method via SolidWorks.” In *2006 1st IEEE International Conference on Nano/Micro Engineered and Molecular Systems*, 747–51, 2006. https://doi.org/10.1109/NEMS.2006.334887.

[24]Foundation, Blender. “Blender.Org - Home of the Blender project - Free and Open 3D Creation Software.” *Blender.Org* (blog). Accessed November 23, 2019. https://www.blender.org/.

[25]“AutoDrive Challenge.” Accessed November 23, 2019. https://www.sae.org/attend/student-events/autodrive-challenge/.

[26]“Gazebo : Tutorial : Intermediate: Sensor Noise.” Accessed November 23, 2019. http://gazebosim.org/tutorials?cat=guided_i&tut=guided_i3.

[27]“ASCII.” On *Wikipedia*, October 24, 2019. https://en.wikipedia.org/w/index.php?title=ASCII&oldid=922763882.

[28]“Joint_state_publisher - ROS Wiki.” Accessed November 23, 2019. http://wiki.ros.org/joint_state_publisher.

[29]“Ignition Math: Ignition Math.” Accessed November 23, 2019. https://ignitionrobotics.org/api/math/4.0/index.html.

[30]“Launch Files — ROS Tutorials 0.5.1 Documentation.” Accessed November 23, 2019. http://www.clearpathrobotics.com/assets/guides/ros/Launch%20Files.html.

[31]Redmon, Joseph, and Ali Farhadi. “YOLOv3: An Incremental Improvement.” *ArXiv:1804.02767 [Cs]*, April 8, 2018. http://arxiv.org/abs/1804.02767.

[32]Doan, Tri, and Jugal Kalita. “Selecting Machine Learning Algorithms Using Regression Models.” In *2015 IEEE International Conference on Data Mining Workshop (ICDMW)*, 1498–1505, 2015. https://doi.org/10.1109/ICDMW.2015.43.

[33]Szegedy, Christian, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. “Going Deeper with Convolutions.” *ArXiv:1409.4842 [Cs]*, September 16, 2014. http://arxiv.org/abs/1409.4842.

[34]Pedoeem, Jonathan, and Rachel Huang. “YOLO-LITE: A Real-Time Object Detection Algorithm Optimized for Non-GPU Computers.” *ArXiv:1811.05588 [Cs]*, November 13, 2018. http://arxiv.org/abs/1811.05588.

[35]Hsu, Shih-Chung, Chung-Lin Huang, and Cheng-Hung Chuang. “Vehicle Detection Using Simplified Fast R-CNN.” In *2018 International Workshop on Advanced Image Technology (IWAIT)*, 1–3, 2018. https://doi.org/10.1109/IWAIT.2018.8369767.

[36]“Refining Faster-RCNN for Accurate Object Detection - IEEE Conference Publication.” Accessed November 23, 2019. https://ieeexplore.ieee.org/document/7986913.

[37]Hui, Jonathan. “Object Detection: Speed and Accuracy Comparison (Faster R-CNN, R-FCN, SSD, FPN, RetinaNet And….” Medium, March 26, 2019. https://medium.com/@jonathan_hui/object-detection-speed-and-accuracy-comparison-faster-r-cnn-r-fcn-ssd-and-yolo-5425656ae359.

[38]Redmon, Joseph, and Ali Farhadi. “YOLOv3: An Incremental Improvement.” *ArXiv:1804.02767 [Cs]*, April 8, 2018. http://arxiv.org/abs/1804.02767.

[39]Deng, Jia, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. “ImageNet: A Large-Scale Hierarchical Image Database.” In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, 248–55, 2009. https://doi.org/10.1109/CVPR.2009.5206848.

[40]Johnson, Rie, and Tong Zhang. “Accelerating Stochastic Gradient Descent Using Predictive Variance Reduction.” In *Advances in Neural Information Processing Systems 26*, edited by C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, 315–323. Curran Associates, Inc., 2013. http://papers.nips.cc/paper/4937-accelerating-stochastic-gradient-descent-using-predictive-variance-reduction.pdf.

[41]*Leggedrobotics/Darknet_ros*. C++. 2017. Reprint, ETH Zurich Legged Robotics, 2019. https://github.com/leggedrobotics/darknet_ros.

[42]“OpenCV.” Accessed November 23, 2019. https://opencv.org/.

[43]“Boost C++ Libraries.” Accessed November 23, 2019. https://www.boost.org/.

[44]“NVIDIA CUDA Installation Guide for Linux.” Concept. Accessed November 23, 2019. http://docs.nvidia.com/cuda/cuda-installation-guide-linux/index.html.

[45]“YAML.” On *Wikipedia*, November 13, 2019. https://en.wikipedia.org/w/index.php?title=YAML&oldid=925982021.

[46]Jiang, Jingjing, and Alessandro Astolfi. “Lateral Control of an Autonomous Vehicle.” *IEEE Transactions on Intelligent Vehicles* 3, no. 2 (June 2018): 228–37. https://doi.org/10.1109/TIV.2018.2804173.

[47]Sharma, Shobit, Girma Tewolde, and Jaerock Kwon. “Behavioral Cloning for Lateral Motion Control of Autonomous Vehicles Using Deep Learning.” In *2018 IEEE International Conference on Electro/Information Technology (EIT)*, 0228–33, 2018. https://doi.org/10.1109/EIT.2018.8500102.

[48]Bojarski, Mariusz, Philip Yeres, Anna Choromanska, Krzysztof Choromanski, Bernhard Firner, Lawrence Jackel, and Urs Muller. “Explaining How a Deep Neural Network Trained with End-to-End Learning Steers a Car.” *ArXiv:1704.07911 [Cs]*, April 25, 2017. http://arxiv.org/abs/1704.07911.

[49]Yao, Yuan, Lorenzo Rosasco, and Andrea Caponnetto. “On Early Stopping in Gradient Descent Learning.” *Constructive Approximation* 26, no. 2 (August 1, 2007): 289–315. https://doi.org/10.1007/s00365-006-0663-2.

[50]Ashourloo, Davoud, Hossein Aghighi, Ali Akbar Matkan, Mohammad Reza Mobasheri, and Amir Moeini Rad. “An Investigation Into Machine Learning Regression Techniques for the Leaf Rust Disease Detection Using Hyperspectral Measurement.” *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing* 9, no. 9 (September 2016): 4344–51. https://doi.org/10.1109/JSTARS.2016.2575360.

[51]“Cv_bridge — Cv_bridge 0.1.0 Documentation.” Accessed November 23, 2019. http://docs.ros.org/melodic/api/cv_bridge/html/python/index.html.

[52]Shah, Shital, Debadeepta Dey, Chris Lovett, and Ashish Kapoor. “AirSim: High-Fidelity Visual and Physical Simulation for Autonomous Vehicles.” *ArXiv:1705.05065 [Cs]*, July 18, 2017. http://arxiv.org/abs/1705.05065.

[53]NVIDIA Developer. “NVIDIA DRIVE Constellation,” February 14, 2019. https://developer.nvidia.com/drive/drive-constellation.

[54]“Documentation | CMake.” Accessed December 10, 2019. https://cmake.org/documentation/.

[55]GitHub. “Jrkwon/Nikhil.” Accessed December 10, 2019. https://github.com/jrkwon/nikhil.

[56]Kwon, Jaerock. *Jrkwon/Mir_torcs*. Python, 2018. https://github.com/jrkwon/mir_torcs.

[57] ResearchGate. “On the Lane Detection for Autonomous Driving: A Computational Experimental Study on Performance of Edge Detectors.” Accessed December 20, 2019. https://www.researchgate.net/project/On-the-Lane-Detection-for-Autonomous-Driving-A-Computational-Experimental-Study-on-Performance-of-Edge-Detectors.

[58] “DataspeedInc / Dbw_mkz_simulation — Bitbucket.” Accessed November 23, 2019. https://bitbucket.org/DataspeedInc/dbw_mkz_simulation/src/default/.