Deploying Deep Learning Technology in the Robot Operating System

ZHIXI TANG

A dissertation submitted in partial fulfilment
of the requirements for the degree of
Master of Science in Artificial Intelligence
of the
University of Aberdeen.

Department of Computing Science

May 2, 2022

Declaration

No portion of the work contained in this document has been submitted in support of an application for a degree or qualification of this or any other university or other institution of learning. All verbatim extracts have been distinguished by quotation marks, and all sources of information have been specifically acknowledged.

Signed:

Date: May 2, 2022

Abstract

With the advent of deep learning, scientists have begun to explore how this new technology can benefit humans. Trying to combine new technology with existing technology is an efficient method of research for solving complex problems in existing application domains. Robots are considered to be machines that follow the rules designed in their program to solve problems. Even though they usually emulate human behaviour, it was not until progress was made into machine learning that robots could effectively be used in complex tasks where learning is required. With the development of neural networks, the idea that having a robot with a neural network might allow the robot to think like a human is beginning to be investigated. Even if robots cannot think like humans in a short period of time, if the efficiency of robots in solving practical problems can be improved by deploying deep learning technology, it may lead to novel applications of robotics and more widespread use of it in society.

In this dissertation, we present a description of the composition of robotic systems, the basic concepts of deep learning and neural networks, the tools used during the project, and the design and implementation of the experiment. We set out to build a prototype that can apply deep learning models to solve image recognition problems in robotic systems. To evaluate our prototype we perform an experiment with several objects and collect data that is then analysed to further explore the feasibility and practicability of deploying convolutional neural networks in the Robot Operating System (ROS).

Our results show that it is feasible to deploy deep learning to run on ROS. At the same time, we also summarise some practical factors that need to be considered in the deployment of deep learning technology in ROS. Moreover, we also describe the problems we encountered during development, and summarise some ideas on how to improve this approach in future work.

Acknowledgements

I would like to express my appreciation to some people for their support, encouragement, and guidance on this project. Firstly, I would like to thank Doctor Rafael Cardoso, my supervisor for the dissertation project, who provided me with many suggestions and guidelines throughout the whole project. Secondly, I would like to thank my friend Tjasa Recelj, a graduate student at the University of Aberdeen, who provided me with many suggestions on English writing. Thirdly, I would also thank my family, who gave me a lot of encouragement during the writing and coding period for this project.

Contents

Chapter 1

Introduction

When people talk about robots, the picture that enters people's minds for the first time is that of a machine with a human face. However, most people do not think that robots can think and make accurate decisions like humans do in science fiction movies. In general, people think of a robot as a set of machines that have been programmed to perform specific tasks. For example, the robot can use a series of algorithms to build a map in a closed and unfamiliar environment and navigate according to the map. Or in the manufacturing industry, according to a specific commodity template, a certain link is automatically produced, and so on.

Nowadays, the technology of Artificial Intelligence (AI) is rapidly improving. Especially with the advent of neural networks (a network of artificial neurons with weights representing the connections from biological neurons [15]), some complex algorithms and model structures have surpassed the ability of humans to manually calculate and analyse. Indeed, machine learning techniques have begun to be applied in our daily life, such as commodity classification, face recognition, text analysis, etc.

Research on combining machine learning technology with traditional robotics remains a very active field of research. Its main purpose is to investigate if machine learning can make robots more efficient and more "intelligent".

1.1 Motivation

Deep learning is one of the methods in machine learning [11]. Techniques in deep learning are used to get the information via learning the inherent orderliness and representations of data. Deep learning algorithms and methods are very useful and effective for parsing text, image, and voice data. In the practice of deep learning, a model may only solve a specific problem in a specific scenario. When one of the factors changes, the model is no longer effective. The model relies on static data for training, while in reality, the data is dynamic, so when the features of the data change, the previous model may fail, because the new data features do not exist in the learning memory of the model.

In the development of robotic systems, different robot applications rely on different hardware architectures, operating systems, and programming languages. In order to ensure that the robot can work normally, we need to ensure that the various components of the robot can work together in an orderly manner. If there is a problem with the compatibility between the robot's components, it is possible that there will be one or more problems with the timing of the process, likely

resulting in the robot not being able to work normally and the application failing. ROS (Robot Operating System) [30, 25] is a peer-to-peer, modular, open-source middleware for the development of robotic software that aims to solve the aforementioned problem.

In view of the above problems, the goal of our project is to explore how to combine robot technology and deep learning technology to let the robot solve the problem and evaluate the final result to obtain the practicability of the problem. In order to ensure the feasibility of the project at the beginning, we chose image recognition as the robot's application field. In this project, we will use a robot in a simulated environment to recognise dynamic streaming media (video) instead of a single image.

1.2 Project Objectives

The first objective described in this dissertation is to build a robot system in ROS consisting of applications that can solve the image/stream detection problem with deep learning technology. When the system is built, then we use some objects to let the robot detect samples from different angles and distances in the simulated environment.

The second objective described in this thesis is to analyse and evaluate the results given by the robot so that we can conclude whether combining the technology of robotic applications developed in ROS and deep learning is feasible and practicable.

1.3 Outline

The remainder of this dissertation is organised as following:

Chapter 2 : **Background and Literature Review** - Explain terminology and concepts related to this dissertation, including ROS, simulated environments, and Deep Neural Network.

Chapter 3 : **Methods and Implementation** - The description of what, how and why decisions have been executed through this project. This includes acquiring the pre-trained models, the design and implementation of robotic system, building the simulated environment, and the implementation of the experiments.

Chapter 4 : **Evaluation** - Presents the results obtained with quantitative and qualitative analysis.

Chapter 5 : **Conclusions** - Summarises the contributions in this dissertation, provides a critical analysis of the implemented work, defects, and possible future work for optimisation and further development.

Chapter 2

Background and Literature Review

This chapter introduces the basic concepts and terminologies of ROS, simulators, and deep learning. In each section, we also report some personal experiences on using some of the tools and techniques that we use to justify why we chose them. Finally, we introduce some related practical applications and projects.

2.1 ROS

In this section, we briefly introduce the concept of ROS. Secondly, we discuss the different versions of ROS and why we went with ROS2. Afterwards, we introduce the basic components of a ROS program and the main communication methods that are used in this project.

2.1.1 Basic concepts

The Robot Operating System (ROS) is an open-source robotics middleware suite using Apache 2.0 license [30]. ROS originated at Stanford University and released its initial version in 2007. After the construction and development of the ROS ecosystem, in 2010, Open Source Robotics Foundation (OSRF), the maintainer of ROS, open sourced the first official ROS version, "Box Turtle". Every year after that, OSRF releases a new version ROS version to fix and iterate on the functional defects of the previous version.

From the classical definition of a computing operating system, ROS is not considered an operating system [28]. Instead, it is a collection of software frameworks and application packages for robot development. It provides a variety of abundant software packages and tools to help developers quickly and efficiently design, develop, test, and deploy robot applications. ROS supports running both on simulated environment as well as physical-mechanical robots. ROS is not a real-time operating system, which in this case means that we can avoid the impact of dynamic data and environment on deep learning models.

Generally, we can split ROS into three parts as follows [26, 25]:

1. **Language-side and cross-platform tools**: for building and deploying basic ROS software to support the proper operation of various basic functions of ROS.
2. **ROS client package**: it enables developers to quickly build interfaces using a specific programming language (e.g., C++, Python, Lisp) to implement communication between components within ROS.
3. **Application packages**: can enable robots to implement various specific functions on the basis of the ROS client library, language, and cross-platform tools. For example, automatic navigation, map mapping, distance measurement, differential control, and so on.

Overall, the first part and second part are the basis of ROS. The developer can develop the third part to customise the robot. In this project, we build various functions to interact with other ROS basic functional packages to achieve the combined system.

2.1.2 ROS 1 and ROS 2

At the outset, ROS only supports a few languages such as C++, and Python2. Furthermore, ROS can only run (natively) on the Linux Ubuntu operating system. With the development of technology, various other programming languages, operating systems, hardware platforms, and technical standards began to appear and become popular. In this case, the maintainers of ROS decided to start developing the next generation of ROS in 2014. Therefore, the previous ROS is called ROS 1, and no main distributions of ROS 1 are planned to be released after May 2020. The new generation of ROS is called ROS 2 [27]. The first official version of ROS 2, "Ardent Apalone", was released in December 2017.

Compared with ROS 1, the reasons why I chose ROS 2 are shown in the following:

- **ROS API** — In ROS 2, all the basis functions are built by C/C++ in the package of RCL (ROS Client Library), the developer can use these functions with other client library, which is an API to interact with those functions. This means that developers are no longer limited to developing robots using limited programming languages. If developers want to use other programming languages, they do not need to reprogram the core of ROS, they just need to make a C/C++ binding to connect their own libraries with RCL.
- **Python and C++ Versions** — Python2 is no longer maintained. In ROS 1, only the last distribution, "Noetic", supports Python3. In ROS 2 all distributions already support Python3. For Cpp, ROS 1 was targeting Cpp 98, if the developers try to use another Cpp version, it might break other dependencies in ROS. ROS 2 supports Cpp 11, Cpp 14, as well as Cpp 17. ROS 2 has an advantage in this aspect because new versions of Cpp and Python introduced many useful internal functionalities, which can make development easier, quicker, and safer.
- **Communication management** — In ROS 1, when the robot is running, there is a node called ROS master, which is used to coordinate and manage the work between other nodes. All communication between child nodes must be scheduled by the ROS master. In such a case, when the ROS master node fails, the entire system fails. In ROS 2, there is no longer a ROS master node, each node itself manages and schedules its own behaviour. And when a node fails, it only affects the tasks that the node is responsible for, and will not affect the functions of other nodes in ROS. This update enables developers to quickly find faulty nodes and repair them, and makes it easier to deploy robots in a distributed system.
- **Launch file** — The launch file is also known as a pipeline, which connects each part of the application so different nodes can work together. Both in ROS 1 and ROS 2, if the structure of the system is very complex, it is very troublesome to start ROS nodes one by one, so by using the launch file we can start multiple nodes at once by writing some rules. In ROS 1, the launch file is implemented by writing a file in XML format. In ROS 2, the launch function is encapsulated into a Python library, so we can call it easily, and we no longer need to understand the grammar rules about XML for writing launch files.
- **Debug tools** — On the basis of the previous generation of debug tools from ROS, ROS 2

improves their functions that can be used to quickly check whether the interaction between the application and ROS is conformant with the standards.

- **Iterability and extensibility** — We want this project to be iterative and extensible. Therefore, using the latest release of languages, libraries, and frameworks will make the project source code more useful in the long term.

During this project I experimented with both generations of ROS, and ultimately, I chose ROS 2 to build the robotic system for this project due to the reasons listed above. Thus, unless otherwise specified, when ROS is mentioned in the remained of this document it represents ROS 2.

2.1.3 ROS architecture and communications

The architecture of ROS can be divided into many packages. A package can be divided into many nodes. A node is a basic element that composes ROS. The operation of ROS is essentially a process in which many internal nodes work together by adopting different communication methods [30]. Thus, to design a ROS application for our project, we have to understand the basic architecture of ROS and the major communication methods between ROS nodes.

ROS node

A ROS node is a module responsible for a single function, for example, one node for controlling the rotation of radar, one node for controlling the wheel motors, etc. Each node can send and receive data to other nodes via the basic ROS communication components, such as topics, services, actions, and parameters. A ROS application consists of many nodes working together. An executable (e.g., Python program, C++ program) can contain one or more nodes. Figure 2.1 illustrates how nodes work together.

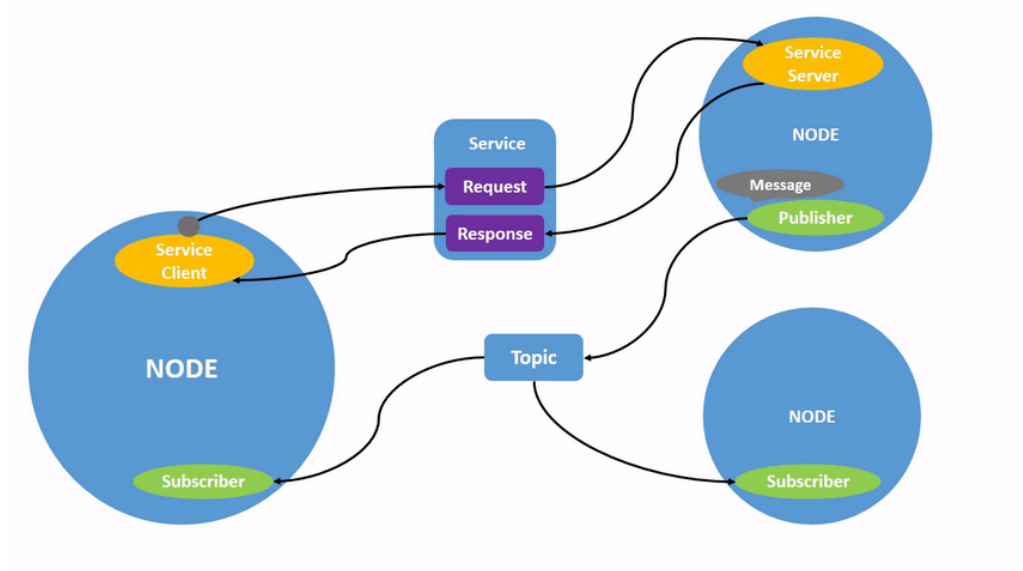

Figure 2.1: Nodes working together via topic and service.

ROS package

A ROS packages can be considered as a container for ROS nodes. A ROS package contains one or more ROS nodes capable of performing a specific function. Developers can organise the nodes

into one package and share this package to others. Other developers can then install the wrapped function through the package. During development, we have to create a work space to compile and store the basic ROS functional packages, and then we have to create a package before we start to program the customised function. Listing 2.1 shows the standard file tree of a ROS 2 project. The package_1 was built by C_make, package_2 was built by ament_python, and package_n is an empty package built by C_make.

```
workspace_folder/
    src/
        package_1/
            node_1.cpp
            CmakeLists.txt (default when package generated)
            package.xml (default when package generated)

        package_2/
            setup.py    (default when package generated)
            package.xml (default when package generated)
            package_2/
                node_2.py
        ...
        package_n/
            Cmakelist.txt
            package.xml
```

Listing 2.1: File tree of ROS.

In ROS, the cooperative work between various nodes relies on various communication methods, such as parameter, topic, subscriber, publisher, service, and action. In our project, we will rely on the following communication methods to build ROS nodes.

Interfaces

Interface can be considered as a data type or container of data. In ROS, different nodes transmit different types of data. The temperature sensor only transmit a float number to represent the Celsius degree so the interface can be *float64*. However, for a node which is responsible for navigation, the corresponding interface should be able to represent the position of robot. In this case, the interface is a container that contains three float numbers, which represent the robot's position on x-axis, y-axis, z-axis based on the coordinate frame. There are many built-in interfaces in ROS, and developers can call an interface according to different needs. Developers can also define their own interfaces, which are composed of many basic data structures. Table 2.1 shows the basic data types supported by custom interfaces in C++ and Python.

Topics communication

In ROS topic communication, the publisher is the data sender in a ROS node. The publisher sends information according to the pre-defined topic name and data interface. The subscriber receives the data in a ROS node. The subscriber will receive the data according to the pre-defined topic name and the data interface, same as the one that the publisher sends. The information published by the publisher is continuous, in real-time and uninterrupted. Therefore, in topic communication, the publisher is only responsible for publishing the information, and does not need to consider whether the information is received by the subscriber. Topic communication is widely used for

Type name	C++	Python
bool	bool	builtins.bool
byte	uint8_t	builtins.bytes*
char	char	builtins.str*
float32	float	builtins.float*
float64	double	builtins.float*
int8	int8_t	builtins.int*
uint8	uint8_t	builtins.int*
int16	int16_t	builtins.int*
uint16	uint16_t	builtins.int*
int32	int32_t	builtins.int*
uint32	uint32_t	builtins.int*
int64	int64_t	builtins.int*
uint64	uint64_t	builtins.int*
string	std::string	builtins.str
wstring	std::u16string	builtins.str

Table 2.1: Basic data types currently supported in ROS

real-time data transmission of various sensors in ROS. Such as ranging sensors, speed sensors, radar, etc.

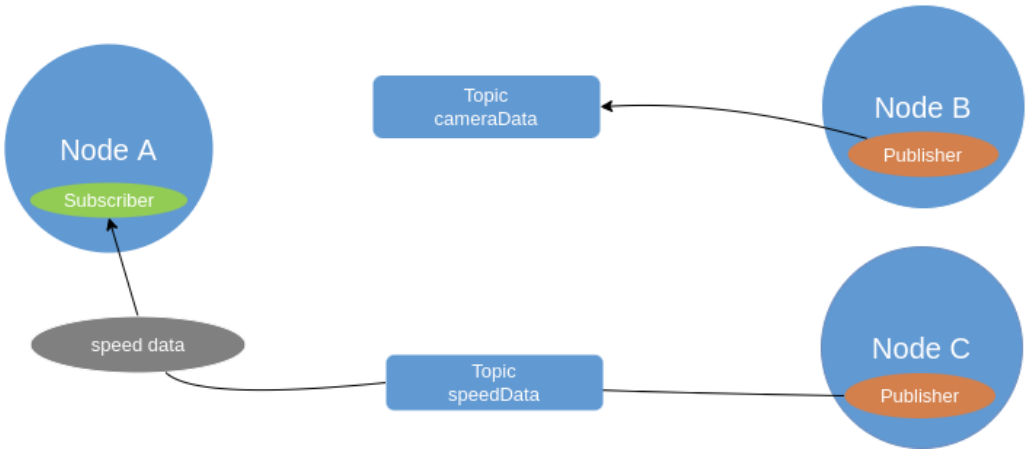

Figure 2.2: Node *A* only receives the speed message as the topic *speedData* has been assigned to it. Node *B* will still send the data to topic *cameraData* but no nodes would receive the data from it because we did not assign this topic to any nodes.

The topic is the most important factor for information exchange between nodes. The topic is what links the publishers and subscribers together so that the data can be transmitted correctly. For example, consider a node named *A* that is responsible for subscribing to and receiving data about the speed of the robot, a node named *B* that is responsible for publishing data about the robot's camera, and a node named *C* that is responsible for publishing data about the speed of the robot. At this time, there are two topics, one is *speedData*, the other is *cameraData*. We assign the topic *speedData* to *A* and *C* and assign the topic *cameraData* to *B*. At this time, *A* can accurately receive the speed data sent from *C*. It will not accept the data sent by node *B*, because the data in *cameraData* is not related to speed and node *A* did not subscribe to this topic. Figure 2.2 shows how topics connect the two nodes in the example and make them work correctly.

The communication of topics is not only one-to-one. A node can also send data to different topics at the same time, and a node can receive data from multiple topics at the same time as well. A topic can also be used by multiple receiving nodes to connect and communicate, but a topic accepts one type of data from one publisher. Figure 2.3 shows how a topic works in this multicast example.

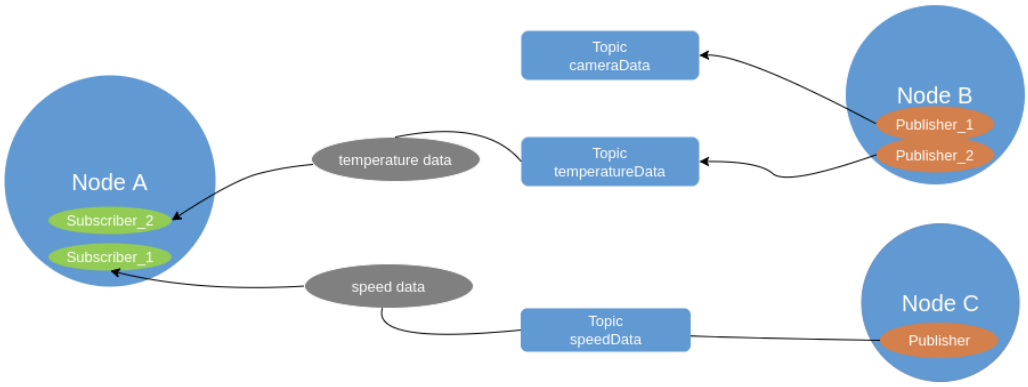

Figure 2.3: How topic communication works in multicast communication. The node *A* is subscribed to two topics *cameraData* and *speedData*, so it can receive the data of temperature and speed. Node *B* publishes to two topics so it can send camera data and temperature data. Publishers can only publish the data type that is specified by their topics.

Service communication

Service communication is the other major communication method in ROS. In service communication, the client sends requests to the server. The server receives the request, processes it, and then returns a response to the client. The response data usually contains the server's processing result of the request. In topic communication, the transmission of data is one-way, the data comes from the publisher and will arrive at subscribers. Unlike topic communication, in service communication, both client and server send data to each other through service. Moreover, service communication is non-persistent and non-real-time, the server only provides response data when a specific service is called. Developers can choose the server to respond to the client's request in real-time or perform asynchronous processing according to their needs. In general, choosing asynchronous communication is a good choice, in a complex robotic system, too many synchronous request operations can cause the process deadlock. Figure 2.4 shows how the client and server communicate in service communication.

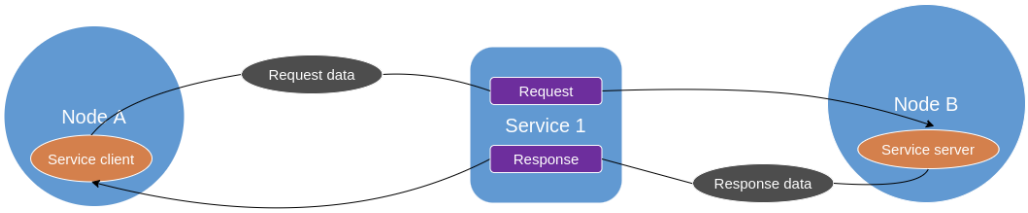

Figure 2.4: How service communication works between a client and server.

The client can also initiate requests to multiple servers at the same time, and these do not have to be necessarily the same requests. A server can accept requests from multiple clients and process them in the order they are received. The type of request is limited only by their interface

when the service is defined. Figure 2.5 shows how multi-clients communicate with multi-servers through services.

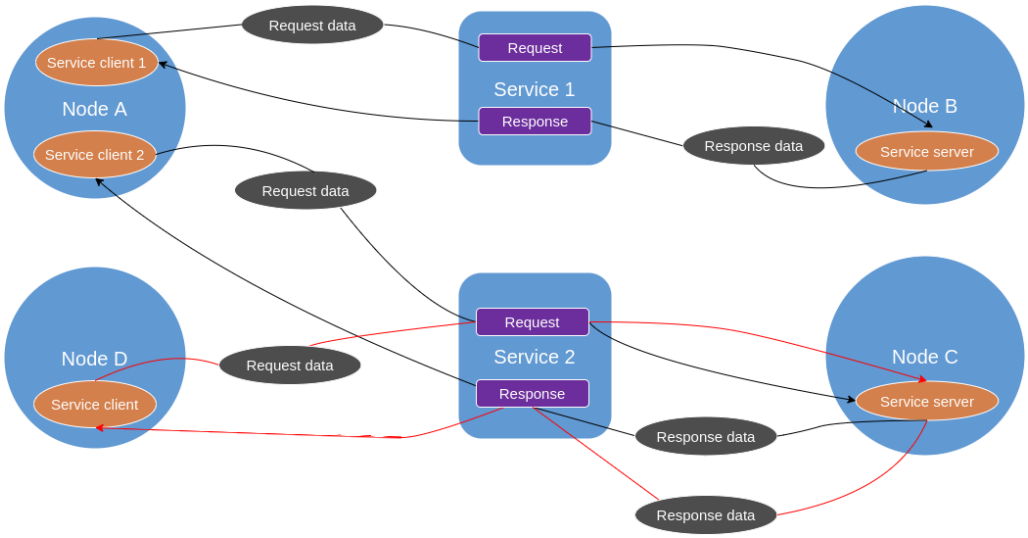

Figure 2.5: How multi-clients communicate with multi-servers through services. Node *A* has two service clients that will send requests to node *B* and node *C* and get a response from them. Node *C* has two clients can call its service, one is node *A*, the other one is node *D*.

In ROS, service communication can be used to check the status of each node, or as a signal, for example, to start or stop a certain action of the robot. Alternatively, there is another communication method called action library, that can be used to implement such functions with more control over the communication between client and server. However, we do not cover the details of it in this document since we did not use the action library during this project.

2.2 Simulation environment

In this section, we cover the basic concepts of simulation environments and the reason why simulators can be useful in robotic development. Afterwards, we introduce *Gazebo*, the 3D simulator that we use in this project, and describe key points that developers should notice when building a simulated environment in Gazebo.

2.2.1 Basic concepts

Simulators are widely used in robotic development as the simulations allows for the programmed scenario to be interactive and it facilitates the movement of the robot across diverse types of environments. Many physical objects and dynamic events in the real world can be simulated in a simulator, for instance, the wind direction and wind speed, the gravity, the inertia, and the collision of objects, among others. The physical attributes (size, weight, etc.) of the object to be tested are able to be programmed to be equal to the real object as well. Simulators have several advantages in robotic development, which include [5]:

1. Developers can save time and speed up the iterative process, developers can modify a function and check the performance right away in the simulated environment.
2. The experiment can be carried out in a secure environment, such as avoiding damage to a robot due to a collision.

3. Saving in the cost, since compared to real experiments designers do not need to spend too much money on the real hardware of the robot, as well as the labour costs of assembling, before it has been properly tested and developed.
4. Speed up the training process of machine learning algorithms. Machine learning algorithms usually require powerful performance hardware. It is hard to apply it to real robots due their limited onboard capabilities. Instead, a medium performance external computer can easily perform training that will then later on be added to the robot.

In real robotic development, experimenting with robots in the simulated environment before the deployment of the physical robot is always advantageous. There are many simulators adopted in the robotic development community, such as Webots, Gazebo, CoppeliaSim, NVIDIA ISAAC Platform for Robotics, ABB RobotSutio, etc. Some of them are open-source while others are proprietary. These simulators can simulate a 3D or 2D visible environment on the computer and provide various plugins to interact with robotic programs. In this dissertation, we use Gazebo to simulate the environment.

2.2.2 Gazebo

Gazebo is an open-source simulator [23, 24] specialising in robotic development [21]. Development started in 2004 as the part of Player Project. In 2011, Gazebo became an independent project and was maintained by Willow Garage. The community of Gazebo developers is very active, Gazebo has had a significant version iteration almost every year since 2013. Even though there are frequent versions of Gazebo being developed, Gazebo has maintained compatibility with ROS. For example, older robotic models are still executable when used with the latest distributions of ROS. Additionally, Gazebo has the following features:

1. **Dynamic simulation:** Gazebo supports many high-performance physical engines, including ODE, Bullet, SimBody, and DART.
2. **3D visible environment:** Gazebo has a graphical user interface that visualises the environment on the monitor so that developers can conveniently observe the simulation.
3. **Sensor simulation:** Gazebo supports the simulation of sensor data, as well as the noise of sensors.
4. **Extendable plugins:** Gazebo supports customised plugins. This greatly increases the extensibility of Gazebo, allowing robot developers to use different custom plugins according to different development needs.
5. **Various internal robotic models:** Besides using developers' robotic models, developers can use the internal Gazebo robotic models for building a prototype faster. Gazebo internal robotic models including PR2, Pioneer2 DX, TurtleBot, and so on.
6. **TCP/IP protocol:** Gazebo supports remote simulation, backend simulation, and local simulation with the communication method based on TCP/IP protocol.
7. **Cloud simulation:** Gazebo supports running on the cloud servers, such as Amazon, and Softlayer. Developers can deploy Gazebo on their own server as well.
8. **Friendly GUI and command-line tools:** Developers can control the simulation by using GUI or command-line tools of Gazebo.

Developers can build a Gazebo simulated environment by dragging and dropping default 3D

models in Gazebo GUI and the environment can be saved as an XML syntax file containing the parameters and attributes of the environment. Thus, developers can write files that conform to XML syntax to build the simulated environment as well. Generally, when the customised robotic model and complex attributes of the physical environment are required, writing XML syntax files is a good option. Gazebo simulated environment consists of three parts, which are the world, model, and plugin [17].

Gazebo world

Gazebo world is equivalent to a representation of a place where experiments are conducted with robots in the real world. The Gazebo world is defined as the items that exist in the environment and the environment's physical attributes. Items are visible elements that appear by default with physical properties, for example, buildings, semaphores, trees, and so on. Physical attributes are the invisible elements that can affect the behaviours and representations of items and robots in the environment. For example, wind, light, gravity, inertial, temperature, humidity, and so on.

Gazebo model

A Gazebo model refers to a 3D model of a visible object that appear in the Gazebo world. For example, robots, buildings, semaphores, trees, and so on. All the internal physical attributes of items are also programmed as a file that conforms to XML syntax. The file extension name is usually .urdf or .sdf. SDF file is recommended by the Gazebo community as it is originally supported by Gazebo. In Gazebo, an object consists of one or several simple geometric shapes, as well as possibly some links and joints.

A link in the simulation can represent inertia, collision and visual properties. It represents the geometric shape in the simulation and a link must be a child label of the item, with one item being capable of having multiple links. The joint connects links so all the links combine into an item. Figure 2.6 and Figure 2.7 shows how to simulate a robot arm inspired by a human arm.

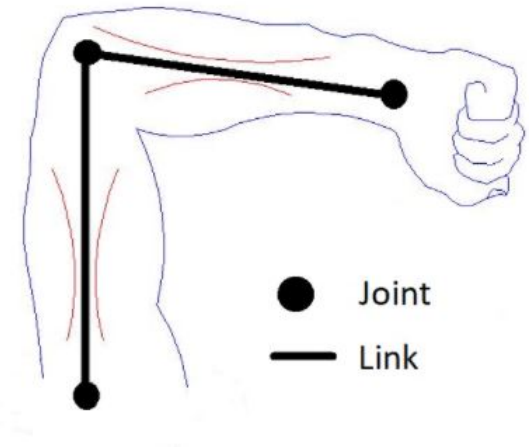

Figure 2.6: Using the concepts of links and joints to represent human arm. [1]

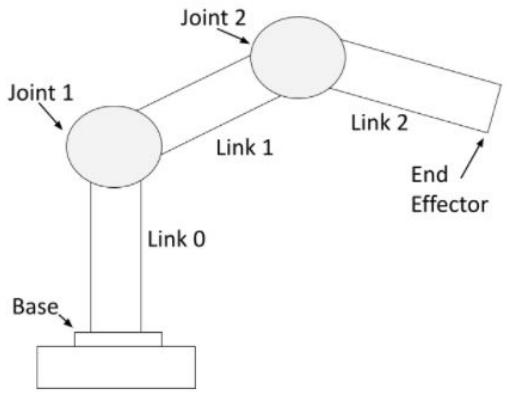

Figure 2.7: How to create a robotic arm using the concepts of links and joints. [1]

In order to better simulate objects and robots from the real world, developers can customise Gazebo models' appearance by using decoration files with format STL, DAE or OGJ. Decoration files can be easily created and saved by 3D modelling software, such as Blender, 3D Studio Max, etc. A Gazebo model is usually constructed as shown in Listing 2.2.

```
model_name/
    meshes/
        decoration_file_1.stl
        decoration_file_2.stl
    model.config
    model.sdf
```

Listing 2.2: Structure of a Gazebo model.

Gazebo plugins

The Gazebo plugins are the core parts that supports the robot model communicating with ROS [22]. The robot is also an item model, but in addition to links and joints, the robot model file also contains multiple plugins, so that the model in Gazebo can be connected to ROS. Usually, the plugins in the robot model include sensor plugins, status publishing, status subscription plugins, etc. The sensor plugin is responsible for simulating sensor data and publishing it to the relevant nodes in ROS. The state publishing plugin monitors the position and physical state of the robot model and publishes it to each node in ROS. The state subscription plugin accepts the information from the relevant nodes in ROS, thereby changing the current state of the robot in the Gazebo environment. The communication methods of all plugins are carried out in the ROS communication methods mentioned previously.

Item models, robot models and the world can be written in a single file. However, for convenience and modular programming, developers usually save them separately, and then connect each other using the world file, or load items and robots through the Gazebo interface.

2.3 Object detection using deep learning

In this section, we introduce the basic concepts of Deep Learning (DL), and then we discuss one of the common architectures of DL, Convolutional Neural Network (CNN), and how it works. Afterwards, we interpret its advantages in the implementation of object detection.

2.3.1 Basic concepts of DL

Deep learning (DL) is a subset of machine learning [11, 18]. Deep learning networks learn by discovering intricate structures in empirical data. By building computational models that contain multiple layers of processing, deep learning networks can create multiple levels of abstraction to represent data in deep learning models.

The low-level feature representation can be considered human-readable information. The high-level feature representation can be counted as abstract format information that humans cannot understand directly, such as multi-dimensional tensor, multi-dimensional vector, a series of parameters, or a combination of them, etc. In deep learning neural network, through multi-layer processing, after gradually transforming the initial low-level feature representation into a high-level feature representation, compared with the input data, the model is small and simple. And this simple model can be used to complete learning tasks such as complex classification [18, 11].

Deep learning can be considered as a general designation for a type of pattern analysis method. As far as the specific structure of deep learning neural network is concerned, we can conclude deep learning has the following types [6]:

1. Neural network system based on convolution operation, namely Convolutional Neural Network (CNN).
2. Multi-layer neuron neural networks based on series, encoding, and decoding. Including recurrent neural networks, auto-encoder, and sparse coding, which have received extensive attention in recent years.
3. The pre-training is carried out on a multi-layer self-encoding neural network. Then the model will optimise its weights by combining it with the discriminant information. Including Deep Belief Network (DBN).

Deep learning is different from traditional machine learning [4]. In machine learning, many traditional machine learning algorithms have limited learning capabilities, and the increasing amount of data cannot continuously increase the total knowledge learned by the model, while deep learning can improve performance by accessing more data. Traditional machine learning algorithms utilise structured and labelled data to make predictions, which means that features are usually defined from the input data in a structured format. Thus, data scientists have to do a lot of pre-processing work to organise the data into a structured format before feeding the data into the model. If the data volume is vast, the pre-processing work can be exhausting.

For instance, assuming we have a set of pictures of different animals. The task is to classify these pictures by 'cat', 'dog', 'elephant', etc. In traditional machine learning, the expert might spend a large amount of time designing and building a system to detect the features that represent an animal. Then label this hierarchy of features on each image and feed them to the machine learning model. In deep learning, it dislodges some data pre-processing steps which are typically involved with machine learning. As we mentioned, deep learning can process the data from the low level to the high level. In high-level data, it would contain particular features with abstract formats. Thus, the human does not have to manually label the specific data features. Deep learning algorithms can learn which physical features are the key (e.g., size, fur, ears, eyes, etc.) to telling apart an animal from another. We only have to provide an adequate amount of images to train the model, the deep learning model will then learn from the pixels of the image, and extract and categorise these pixels that represent physical features into groups.

Besides object detection, deep learning has excellent performance on other others task as well. Such as speech recognition [9], audio processing [29], and machine translation [37].

2.3.2 Convolutional neural network

CNN [2, 16] is the most common and important architecture in DL. Many other neural networks or problem solutions are based on it. In this project, we are going to use models based on CNN to perform image detection using a robot.

Basic concepts

Compared with other neural networks, CNNs have significant better performance with image inputs [2, 16]. In CNN networks, the data processing is called convolutional calculation. The CNN model extracts, groups, and memorises the features of the input image through the convolution operation, so as to achieve the learning process. Figure 2.8 shows how CNN works in an image classification task.

CNNs have three main types of layers, which are convolutional layer, pooling layer, and

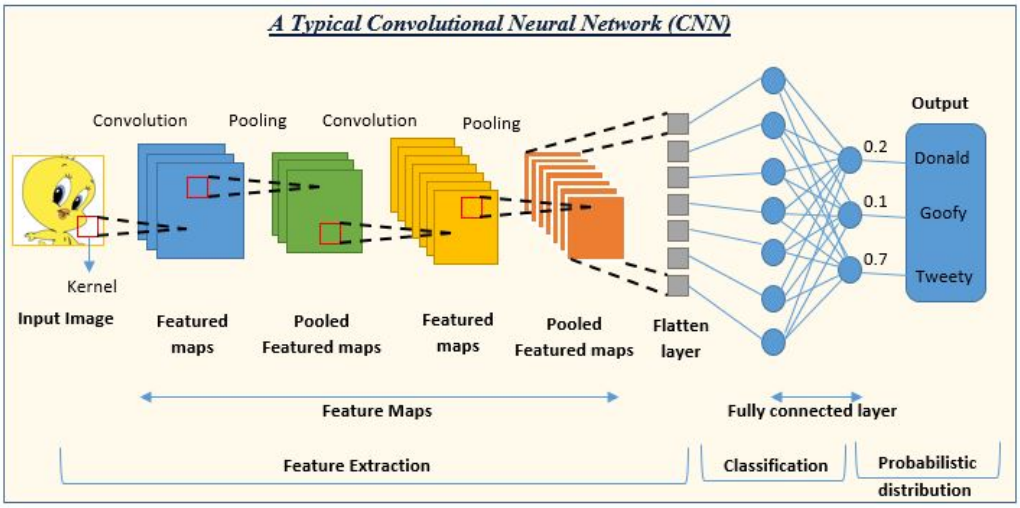

Figure 2.8: How CNN works in an image classification task. [34]

full-connected layer [2, 16]. The CNN starts from a convolutional layer and ends with a fully-connected layer. There can be many additional convolutional layers and pooling layers between them. The complexity of the CNN increases with each successive layer. The transmission of image data is transmitted layer by layer from the first to the last. The previous layers focus on learning smaller features of the image, such as colour, outline, and as the transfer process progresses, later layers learning larger features of the image, such as shape, size, until the expected object is recognised.

Convolutional layer

Convolutional layer [2, 16] is the key part of CNN. Most of computation of CNN is in convolutional layer, the computation is also known as convolutional calculation or convolutional operation. Normally, the filter is a two dimensional array of weights with size 3×3 or 5×5, it is customisable and determines the size of the receptive field. When the filter scans the image, it will cover an area of pixels of size equal to the filter size. Then a dot product will be applied between the pixels and the filter. This scanning process will go through the entire image. To a single channel image, each result will be fed into a feature map. To three channel image, each result is the summation of the dot product between filter and each channel of the image in each scanning. This calculation is also known as convolutional calculation. Figure 2.9 shows how convolutional calculation works with a single channel image as input.

From Figure 2.9, we can observe that the output value in the feature map does not connect with the input values, it only connects with the receptive field, where the area is covered by the filter. Thus, convolutional layers are commonly considered as "partially connected" layers, as well as polling layers. The convolutional layer will use the same filter to scan an entire image, thus, the weight of the filter is fixed during this process. While in some processes such as backpropagation and gradient descent, the value of the weight would be adjusted. The size of the feature map is affected by three parameters, which are the following:

1. **Number of filters:** will affect the depth of the feature map. Using a different filter to scan the image would result in a different output. For instance, if there are three filters

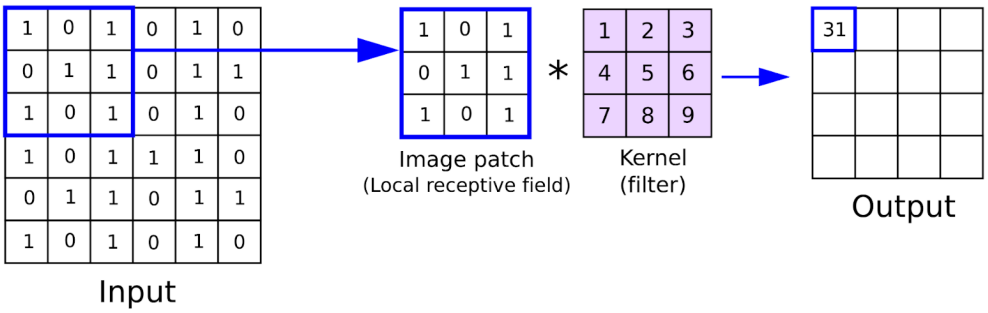

Figure 2.9: How convolutional calculation works with a single channel image as input. The output size of calculation is a dot product between filter and covered pixels (blue line). Similarly, if the input is a three channels image, then the output is the summation of dot product between the filter and covered pixels in each layer.

applied, then it will yield three different feature maps, the size of the feature map would be $(height, width, 3)$.

2. **Stride:** is the number of pixels that the filter moves over the input image. A larger stride would cause a smaller output, thus, the stride is usually one or two.
3. **Zero-padding:** is to extend the image size and fill the extended area with zero when the filter does not fit the image. There are three types of padding:
    - **Valid padding:** When the filter is in the image the convolutional calculation starts, this will cause the last convolution to be dropped if its dimension does not fit the image.
    - **Same padding:** When the centre of the filter and image pixel overlaps with each other the convolutional calculation starts, and the filter part that exceeds the image will be filled with zero. If the stride is one, then the output size is the same as the input size.
    - **Full padding:** When the image border is covered by the filter the convolutional calculation starts, and the filter part that exceeds the image will be filled with zero.

In CNNs, the first convolutional layer extracts the feature from the input. Afterwards, later CNN layers extract features from the outputs of the former CNN layers, and the features are more abstractive than the feature of the former layers. Thus, the convolutional computation creates a feature hierarchy within the CNN. This process is repeated until the fully connected layer is reached.

Pooling layer

The role of the pooling layer [2, 16] is to reduce the dimension of the input layer so the parameters in the input can be reduced. Similar to the convolutional layer, the filter in the pooling layer scans through the entire input but the filter does not have weights. Instead, the filter would apply an aggregation function to the values within the receptive field and feed the output to the feature map. The two major types of pooling are:

1. Max pooling — when the filter scans the input, it will select the maximum value within the receptive field and feed this value to the output.
2. Average pooling — when the filter scans the input, it will calculate the mean of values within the receptive field and send it to the output.

Figure 2.10 shows how max pooling and average pooling work. Even though a lot of information will be lost by using the pooling layer, it can help to reduce the complexity of the neural network, lower the risk of over-fitting, and improve computation efficiency.

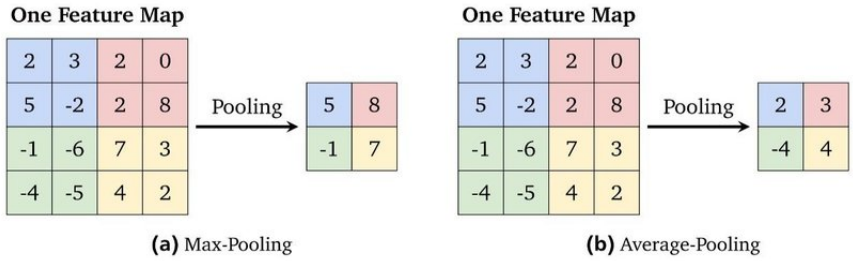

Figure 2.10: How max pooling and average pooling work. The filter size is 2×2 and stride is 2.

Full-connected layer

As mentioned earlier, the pixel values of the input image are not connected to the output layer in convolutional layers and pooling layers. Therefore, the extracted features should be processed so that each node in the output layer can be connected with input so that the CNN knows a feature belongs to a particular category. In a classification task, the full-connected layer [2, 16] applies a softmax activate function producing a probability from zero to one. This probability represents how likely the feature belongs to a certain category.

Using CNN for object detection

The performance of CNN on object detection outperforms other traditional machine learning methods, even other deep learning methods [13]. Researchers have produced many well-structured models for image processing problems, such as R-CNN [10], ResNet [12], Google Inception Net [39], and VGGNet [36]. As mentioned earlier, as an architecture of deep learning, compared with traditional machine learning methods such as SVM and KNN, the most significant advantage of CNNs is that it eliminates the feature extraction steps. Researchers only have to feed the input images into CNNs, and it will learn and extract image features at the same time. Besides, CNNs also have the following advantages when dealing with object recognition tasks:

- **Translation invariant** [8] — Once CNN learns a pattern from a position of the image, it can recognise the same pattern in other images. Compared with traditional machine learning methods, the factors such as camera lens, light, pose, and shelters can cause the model to not recognise patterns correctly. Compared with other deep learning models, such as fully-connected neural networks, if a pattern appears at a different position, the model would learn this pattern over again. This feature allows CNN to efficiently use data so it learns the same pattern using a small number of images.
- **Spatial hierarchies of patterns** [8] — As the layer progresses, the structure of CNN enables it to effectively learn complex and abstract visual concepts from images. For example, we have a picture of a cat, and in the CNN, the features learned by the previous layers are simple features such as the contours that make up the cat's facial elements. The features learned by the middle layers are the facial elements composed of these contours, such as ears, eyes, and nose. The features learned by later layers are various cats composed of facial elements.

- **Fewer memory requirements and easier training** [13] — Due to the convolution operation and its partial-connected characteristics, CNN occupies less memory and computing resources, so the training is faster. For the same object recognition task, a fully-connected neural network can also be applied. However, the output value of the fully connected layer network corresponds to the input value one-to-one, so the time spent and the computing resources occupied will be much larger. For example, for a picture of size 32x32 and a hidden layer with 1024 features, this requires 2^{20} parameters to operate, which will be a large overhead for computing resources.

2.4 Real-time object detection

Real-time object detection is the task of performing object detection in real-time with fast inference while maintaining a base level of accuracy [20]. In real-time detection, the input of the algorithm model is usually a data stream. The data stream consists of many single identical or different pictures, such as videos. Videos are many different pictures combined in a sequence and broadcast with very high frequency [3]. Therefore, in real-time detection, the algorithm model not only needs to classify the objects by executing a classification task but also needs to predict the position of the object by executing a regression task. The output of the algorithm model is also a data stream consisting of the prediction results of a single image. A single prediction result usually includes the category of objects, the probabilities, and the corresponding position of the object in the image. By processing the resulting data, we can synchronously display this information on the video. For instance, if an object is detected by the algorithm model, then the category and likelihood will be displayed synchronously in a video. A frame, also known as a box, will be displayed around the object to indicate the position of the object as well.

2.4.1 One-stage algorithms

We can categorise the current mainstream CNN algorithms for real-time object detection into two categories [40], one is the two-step algorithm based on R-CNN, Fast R-CNN, and Faster R-CNN. The other is the one-step algorithm based on RetinaNet, SSD and YOLO. In a two-step algorithm, the first step is using selective searching or Region Proposal Net to produce a possible target region. Afterwards, in the second step, the model would apply the classification function to the produced target region. The prediction accuracy of two-step algorithms is good but the detection speed is limited. The one-step algorithm only uses one neural network to detect the object borders and predict the possibility scores, which eliminates a step compared with two-step algorithms. Thus, its speed is higher but the accuracy can not be not satisfactory sometimes [40], especially when considering the limited hardware performance of robots, and the scenarios that may be encountered in real deployments. We adopt two one-step algorithms to perform our experiments on real-time object detection, which are SSD and Yolo.

2.4.2 SSD

Single Shot multibox Detector (SSD) was first proposed by Wei Liu et al. in [20]. The concept of SSD draws on the anchor point mechanism in R-CNN but it is a one-step algorithm. The backbone of the SSD uses the VGG16 neural network, on which the original two fully-connected layers are replaced by the fully-connected layers and four additional volume base layers that are added after that to extract image features. The final prediction result of the SSD model is a set of

bounding boxes with a fixed size, and the prediction scores of the target categories in the bounding boxes. The redundant bounding boxes are filtered out by the non-maximum suppression (NMS) algorithm, and the final remaining bounding box and score within it is the prediction result. Figure 2.11 shows the network architecture in SSD.

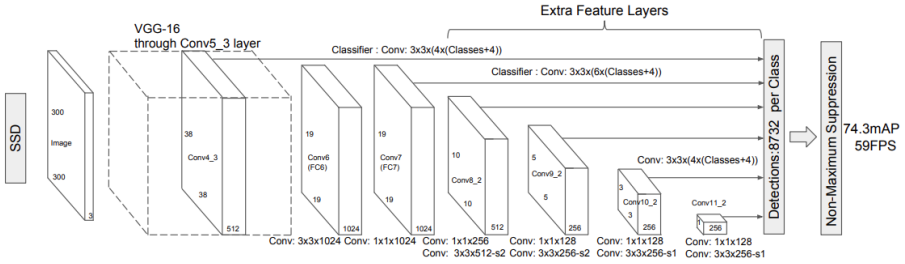

Figure 2.11: SSD network architecture [20].

2.4.3 YOLO

"You Only Look Once" (YOLO) was proposed by Joseph Redmon et al. in [31]. YOLO is a one-step algorithm, it only uses one neural network for the whole image. A single image is divided into small grids by YOLO, these small grids will be the input data for the training process. Afterwards, the probability value is obtained by predicting the category to which the small grids belong. Finally, the boundary of the predicted object is determined by the possibility scores of small grids. The backbone of YOLO can be a simple CNN structure, which is very succinct. YOLO has many variants as well, the latest YOLO variant is YOLO V5. Different backbones are adopted between the different variants, and there are also some methods and tricks applied within different variants to improve the speed and prediction accuracy of the model from the previous version. Figure 2.12 shows the architecture of YOLO.

2.5 Related Work

Deep learning is not often combined with ROS, but there has been some research in the area. There are not many projects that combine ROS 2 and deep learning publicly available, but there are some ROS 1 projects that combine deep learning technology, and those can provide interesting viewpoints and ideas that we can learn from.

The system of the shopping assistant robot [38] consists of four modules, namely cloud server, sensor, ROS, and hardware module. The sensor module is mainly responsible for transmitting images of commodities, and the depth sensor and the noise sensor record the environment where the robot is located respectively. The cloud server is responsible for deploying CNN to train

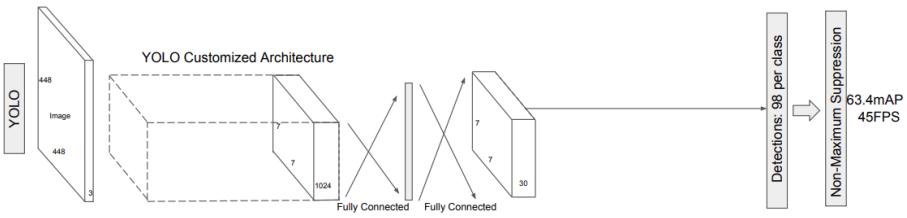

Figure 2.12: YOLO network architecture [20].

and detect commodities and environments. The hardware is responsible for executing the robot's actions. As the core of the system, ROS is responsible for interacting with the above three modules, requesting or transmitting data according to requirements, and making decisions to execute actions. For example, in a scenario where a robot is requested to pick up an item, ROS will send data to the hardware module and sensor module to request the robot to move to the shelf where the item is located. During the movement of the robot, the camera will record the images within the range, upload the picture to the cloud server through ROS, and the CNN in the cloud server will recognise the image and return the recognition result to ROS. When the target item is recognised, ROS will send a signal to drive the hardware to complete a fetching action.

In the target recognition system based on ROS 1, the robot can identify the objects within the range of the camera through R-CNN deployed on the cloud, and display the recognition results on the back end (such as a computer monitor) [7]. The system consists of four main nodes and several sensors, as well as robotic hardware. The four main nodes are the camera node, image processing node, image display node, and ROS master node. The ROS master is responsible for managing all nodes in the system, and the other three nodes need to register with the ROS master at startup. The camera node is responsible for capturing the environment image, and the image processing node processes the captured image and then transmits it to the image display node. The image display node is responsible for communicating with the R-CNN deployed in the cloud. The cloud processes the image, performs recognition, posts back the recognition results, and displays them on the backend.

Chapter 3

Methods and Implementation

This chapter describes the project design process, including the design of the prototype, robot model, simulated environment, and ROS nodes. As well as the selection of image detection models and detected objects.

3.1 Scenario design

In this project, we deployed two different CNN pre-trained models on a ROS 2 system. The robot can detect objects captured by its camera based on the pre-trained models. During the experiment, the detection result would be displayed in the backend synchronously, after each detection, and the related node saved the results as a CSV format file for further evaluation. Our aims were as follows:

- explore which CNN pre-trained models performs better;
- explore how different angles and distances affect detection;
- explore how different shapes of objects affect detection; and
- summarise the feasibility of deploying CNN on ROS 2 and further contribute to the availability of deep learning techniques that can be used in ROS 2.

Figure 3.1 illustrates our system. The figure is a top view of a Gazebo simulated environment. The object to be detected is placed in the yellow area with the size of 4×4, and the object size should not be able to exceed the yellow area. The robot detects the object from 8 positions, the coordinates of these positions are $(-6,0), (-4,0), (4,0), (6,0), (0,-6), (0,-4), (0,4)$, and $(0,6)$. The robot is placed at a 45-degree offset clockwise towards the target at the beginning. Then, when the detection process is started, the robot first performs a 90-degree counterclockwise rotation, then a 90-degree clockwise rotation, this process repeats twice. This ensures that the target object always appears within the camera range during rotation. The fixed rotation angle makes the measurement take the same amount of time at different angles, distances, and different models used.

3.2 Pre-trained model selection

We use two pre-trained models for the project. The first is a regular model which can be deployed on either computer, cloud, or mobile devices. Considering the limited storage space of real robots, we decided to choose a type of model that is specialised for mobile devices as the second model. As mentioned in Chapter 2.4, SSD and YOLO are state-of-the-art in object detection. Therefore, we adopted two variants as the pre-trained models based on SSD and YOLO. The special model

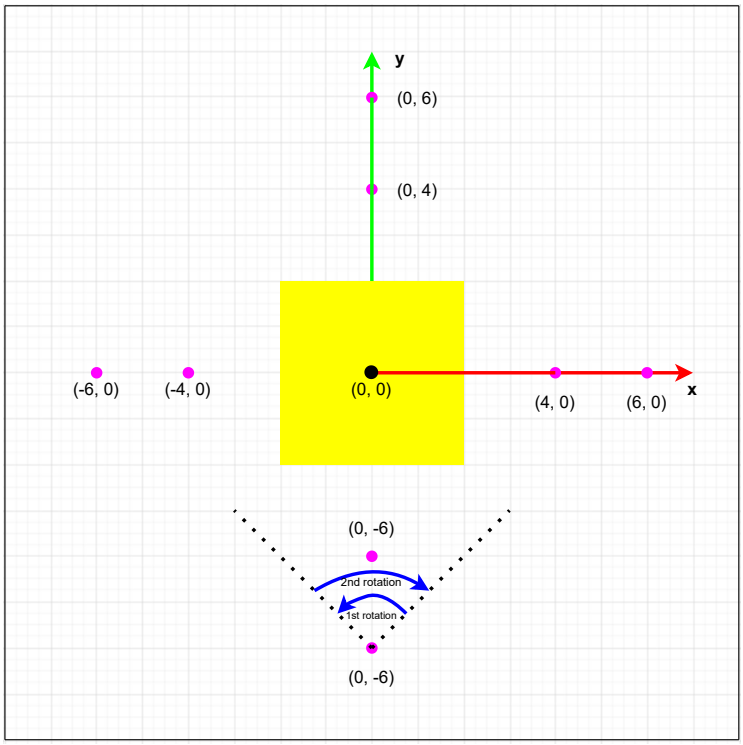

Figure 3.1: Scenario design. The red arrow is the x-axis, and the green arrow is the y-axis. The length and width of a small square are 0.25. A unit square is a combination of 16 small squares. At position $(0, -6)$, the black dotted line represents the direction where the robot is facing initially, and the blue arrows represent how the robot rotates.

is SSD-MobileNet V2 [14, 35], and the regular model is Yolo V3 [33]. Both models have been trained on the COCO dataset [19].

The SSD-MobileNet-V2 is a variant of the original SSD. In Chapter 2.4, we have shown that the original SSD adopts VGG16 as its backbone, also known as a feature extractor. However, the resulting size of the model is big. SSD-MobileNet-V2 [35] is a solution that combined single-shot detection and uses MobileNet V2 as its feature extractor. MobileNet [20] model is designed to be used in mobile applications, compared to VGG16, the size of MobileNet V2 is much smaller as well as having less number of parameters within the network. In the model processing, the size of the networks in memory also grows as the number of the parameters grows. Thus, by using MobileNet V2 as the feature extractor, we can save disk space and make the detection faster. MobileNet V2 is an optimisation based on MobileNet V1 [35], the structure of MobileNet V1 is piling depthwise convolutional layers on one another. MobileNet V2 introduced two new concepts, one is the data dimension expansion layer at the beginning of the model, and the other one is the data dimension compression layer at the end of the model. In the tensor operation, if the dimension of tensors is lower, the operation speed would be faster, but in convolutional operation, the filter can not extract enough features from low-dimension tensors. Therefore, the expansion layer will map the data from low-dimensional to high dimension, which can make the convolutional layers extract more features. The compression layer will compress the high-dimensional back to low-dimensional data, to ensure proper calculation speed. Figure 3.2 shows the architecture of MobileNet V2 and how it works.

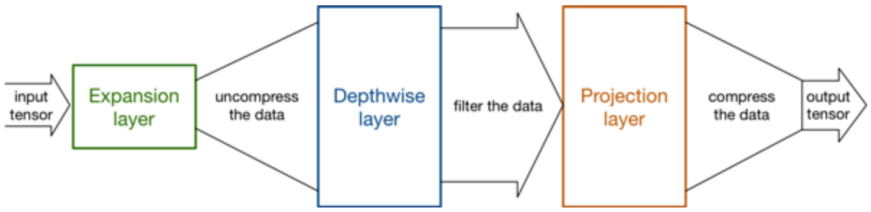

Figure 3.2: The architecture of MobileNet V2 and how it works. [41]

YOLO V3 [33] is a variant of YOLO. Compared with its previous version, YOLO V2 [32], YOLO V3 adopts Darknet-53 [33] as its skeleton, which can make the neural network deeper and improve the extraction of image features. YOLO V3 also did other optimisations to increase the prediction speed as well, including multi-scale predictions. Darknet-53 is a convolutional neural network written in C and CUDA which is 53 layers deep. Even though its structure is very deep, it adopts a large number of Resnet residual structures, so it can avoid the problem of gradient vanishing. Figure 3.3 shows the network architecture in YOLO V3.

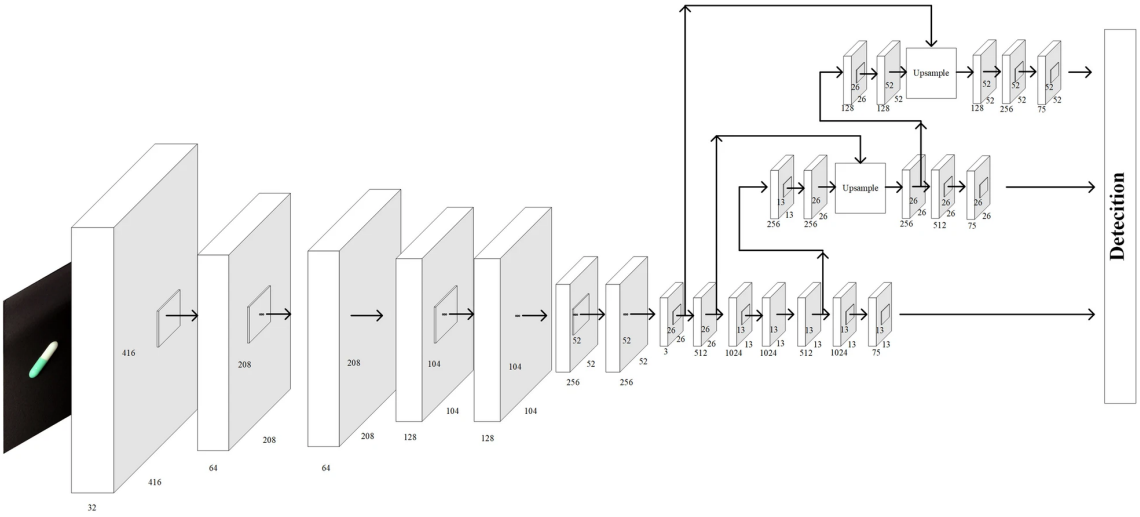

Figure 3.3: The architecture of YOLO V3 [40].

3.3 Objects selection

In SSD-MobileNet V2, the model uses pictures from 88 categories for training. In YOLO V3, the number of image categories is 80. Therefore, in this experiment, we need to select objects from the intersection of them. Then, select a 3D model to be used in the simulated environment. We selected six objects with different original sizes and different shapes for detection so that we can explore the detection performance of models for different targets in our experiments. These six objects are: bus, car, person, chair, monitor, and suitcase. The bus and the car are relatively large size and sharp outline objects. The person and the chair are objects with sharp outlines. And the monitor and the suitcase are objects with relatively simple outlines. The 3D models for these objects were obtained from open-source Gazebo repositories. After choosing the appropriate models, we edit the model file to scale the model proportionally so that its size does not exceed the yellow area from Figure 3.1. The conversion between the original size and the converted size

is shown in Table 3.1.

Model Name	Position Occupied (Original)	Scale Ratio
Bus	x:(-2, 2), y:(-6, 6)	1: 0.003125
Car	x:(-2, 2), y:(-2, 2)	1: 1
Suitcase	x:(-1, 1), y:(-1, 1)	1: 2
Monitor	x:(-0.5, 0.5), y:(0, 0)	1: 1.5
Chair	x:(-1, 1), y:(-1, 1)	1: 2
Person	x:(-1, 1), y:(-1, 1)	1: 2

Table 3.1: The re-scaling ratio of object models. The column "Model name" represents the name of object. The column "Position Occupied (Original)" represents the occupied area where the object original spawned. The column "Scale Rate" represents the ratio at which we scale the model according to the original size of the model.

3.4 Design of robot model

The design of the robot is not critical for this project, as we focus on exploring the feasibility and practicability of CNNs deployment on ROS. However, the robot should be able to meet some essential requirements, which are mobility and a camera. The camera ensures that the robot is able to capture the image of the object so that the CNNs model can recognise it. The mobility ensures that the robot is able to rotate itself at the required angle. Luckily, the Turtlebot Waffle robot meets our requirements and is readily available in ROS and Gazebo. TurtleBot is a low-cost robot kit with open-source software available on ROS. Thus, we can use the simulated Waffle robot for rapid deployment and development. Waffle is a small robot with a camera and two drive wheels. Waffle uses Raspberry Pi as its motherboard, which is equipped with a 32-bit ARM processor. In addition, Waffle is also equipped with a Bluetooth module, ranging radar, battery and other accessories. The linear velocity of the waffle is 0.26 m/s, and the turning velocity is 1.82 rad/s (approximately 104.27 deg/s). The detailed specifications of Waffle is shown in Table 3.2. More importantly, the Gazebo Waffle model integrated the ROS and Gazebo plugins for camera and wheels so that we can write node to control the robot.

3.5 Design of simulated environment

There are not many requirements for the simulation of the environment. For the experiments, we place the object and the robot following the scenario design shown in Figure 3.1. Thus, we only need an open and clear space to put them. We can use Gazebo to simulate such an environment. In the simulated environment, the gravity is approximately 9.8 m/s. There are no other physical elements that affect the environment, such as humidity, wind, noises, etc.

3.6 Design of ROS nodes

The design of the ROS nodes is the core part of this project. As previously mentioned, there is no ROS master to manage and register for nodes in ROS 2. We only have to consider how many modules are in the robotic system and design each module separately. According to the scenario, we divide the robot system into four modules, each module consisting of a ROS node. Figure 3.4 shows the composition of this system and how the nodes work with each other.

Maximum translational velocity	0.26 m/s
Maximum rotational velocity	1.82 rad/s (104.27 deg/s)
Maximum payload	30kg
Size (L x W x H)	281mm x 306mm x 141mm
Weight (+ SBC + Battery + Sensors)	1.8kg
Threshold of climbing	10 mm or lower
Expected operating time	2h
Expected charging time	2h 30m
SBC (Single Board Computers)	Raspberry Pi
MCU	32-bit ARM Cortex®-M7 with FPU (216 MHz, 462 DMIPS)
Remote Controller	RC-100B + BT-410 Set (Bluetooth 4, BLE)
Actuator	XM430-W210
LDS(Laser Distance Sensor)	360 Laser Distance Sensor LDS-01 or LDS-02
Camera	Raspberry Pi Camera Module v2.1
IMU	Gyroscope 3 Axis, Accelerometer 3 Axis
Power connectors	3.3V / 800mA, 5V / 4A, 12V / 1A
Expansion pins	GPIO 18 pins, Arduino 32 pin
Peripheral	UART x3, CAN x1, SPI x1, I2C x1, ADC x5, 5pin OLLO x4
DYNAMIXEL ports	RS485 x 3, TTL x 3
Audio	Several programmable beep sequences
Programmable LEDs	User LED x 4
Status LEDs	Board status LED x 1, Arduino LED x 1, Power LED x 1
Buttons and Switches	Push buttons x 2, Reset button x 1, Dip switch x 2
Battery	Lithium polymer 11.1V 1800mAh 19.98Wh 5C
PC connection	USB
Firmware upgrade	via USB or via JTAG
Power adapter (SMPS)	Input : 100-240V, AC 50/60Hz, 1.5A @max Output : 12V DC, 5A

Table 3.2: Specifications of Waffle.

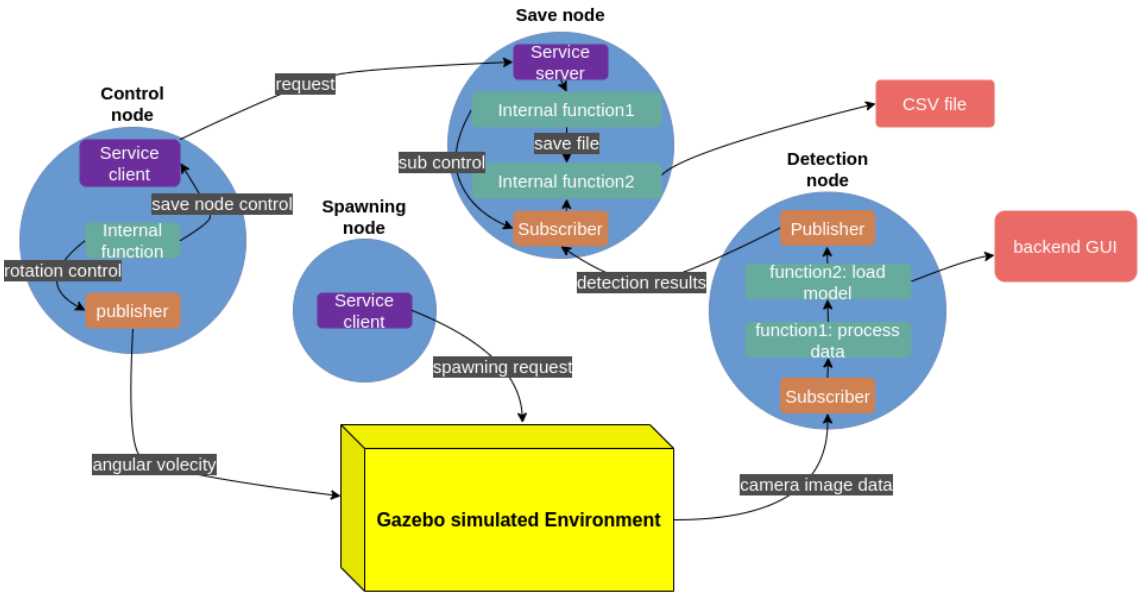

Figure 3.4: The structure of the system and how nodes work with each other.

3.6.1 Spawning node

The spawning node is responsible for spawning the selected object and the robot in the Gazebo simulated environment. When Gazebo is launched, it will launch several services to interact with ROS. The spawning node sends the request to one of these services, SpawnEntity, to spawn new entity in the simulated world. The request includes information of spawned entity's position, the model file, name of entity, etc. Therefore, we can use spawning node to spawn our selected objects and the robot at required positions as represented in Figure 3.1.

3.6.2 Control node

The control node is responsible for controlling the motion of the robot and performing actions. The control node consists of a publisher, a service client, and an internal processing function. When executing the control node, the publisher publishes the angular velocity to the Gazebo robot velocity subscriber to rotate the robot with the angular velocity at 0.182 rad/s. At the same time, the service client of the control node sends a request to the save node with the boolean value *True* to instruct the save node to start working. An internal function is responsible for computing the angle that the robot has rotated so that the control node can control the rotation clockwise or counterclockwise. When the rotation action is finished, the publisher sends a request with boolean value *False* to instruct the save node to stop working. We use service communication to control the save node, alternatively, the action library could also be used to have more control over the communication.

3.6.3 Save node

The save node is responsible for receiving detection results data sent by the detection node and receiving requests sent by the controller node. It saves the detection results according to the request value from the controller node. The save node is composed of a service server, a subscriber, and two internal functions. When the save node is launched, it is in an idle state, which is not receiving data from the detection node. The service server receives a request including a boolean value from the control node. If the boolean value is *True*, then the subscriber of the save node starts subscribing to the corresponding topic to receive detection results from the detection node. If the boolean value is *False*, then the subscriber disconnects to the detection node to stop receiving the results and then calls another internal function to save the received results as a CSV file.

3.6.4 Detection node

The detection node is responsible for capturing the image from Gazebo, processing the image data to the correct format (e.g., png, jpeg, etc.) that the pre-trained model is able to read, and applying the DL model to detect the image, finally, sending the detection result to the save node. The detection node consists of a subscriber, a publisher, and internal functions. When the detection is launched, it starts receiving image data that the robot camera captures from the Gazebo environment. One of the internal processing functions parses the received data to a readable format for the pre-trained model. Another function is responsible for applying the pre-trained model to detect the object and generate detection results. At the same time, this function launches a GUI at the backend so that users can observe detection results synchronously. Finally, the publisher publishes the results on a the respective topic.

3.7 Experiment implementation

To follow the workflow described in the last section, we use multiple terminals on the computer running the simulation to check the output of each step. We divided the implementation of the experiment into the following 5 steps. We repeat the steps at each position for each selected object to collect all of the result data.

1. Launch Gazebo simulated environment.
2. Spawn the object in the simulated environment.
3. Spawn the robot in the simulated environment. Figure 3.5 shows the robot and an object in the Gazebo simulated environment.
4. Launch the save node. In this step, there is no visual change in the simulated environment but the terminal shows the service client is created and waiting for the request from the control node.
5. Launching the detection node. In this step, a GUI is launched to show what image the camera is capturing from the simulated environment. At the same time, the model is applied to detect whether there is an object in the environment. The node then sends the detection results to the topic `detect_result`. The output terminal of the detection node shows that the node is receiving the image from the camera. However, at this time, the subscriber of the save node is disconnected from the publisher of the detection node. Therefore, there is no change in the output terminal of the save node. There are no changes in the simulated environment either.
6. Launch the control node. After the control node is launched, it sends a request to it so that the save node connects with the detection node and the robot starts rotate with angular velocity at 0.182 rad/s. During the rotation, the terminal of the save node and the control node displays related information, the GUI displays the object and the corresponding detection result synchronously as shown in Figure 3.6. When the rotation action is done, the control node sends a request to the save node for it to stop receiving results and save the received results as a CSV file. Afterwards, the control node shuts down.

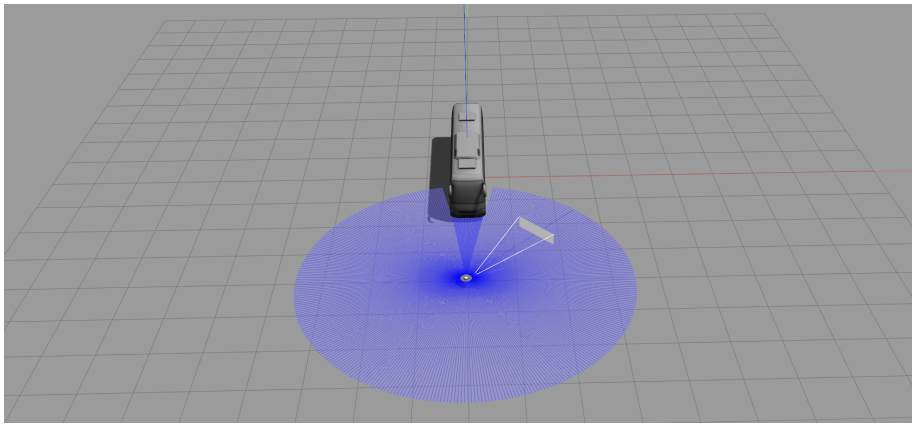

Figure 3.5: Robot and bus object are ready in the Gazebo simulated environment.

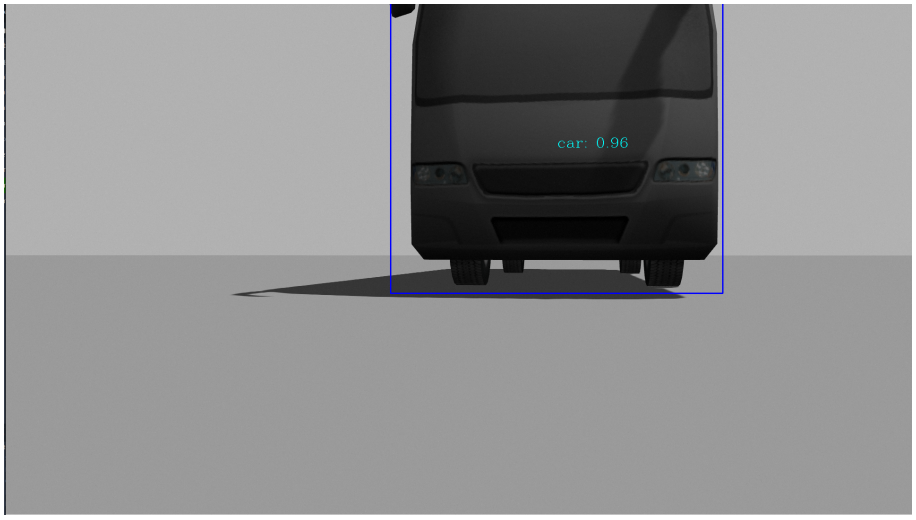

Figure 3.6: The GUI displaying the object and detection result during an ongoing rotation.

Chapter 4

Evaluation

This chapter contains the experiments used to evaluate our prototype. First, we describe what are the measurements collected in the experiments. Second, by processing the raw collected data, we made a table to analyse the detection speed when using different models so that we can speculate whether the CNN models are fast enough for deployment. The further sections describe how the data was processed, and how the processed data was used for further evaluations of model sensitivity and detection accuracy between the two models so that we can discuss whether the CNN models' performance is capable of solving image detection problems using ROS.

4.1 Result description

The position of the robot in each experiment is different and each iteration of the experiment would produce a result file in CSV format. For a single object, we used two CNN models to detect the object at 8 different positions. Thus, for the six objects, in total we collected 96 CSV files. Each file contains a table with the columns named category and score. The category column represents the category predicted by the CNN for the target, and the score column represents the probability of the CNN model for the prediction, also known as confidence. The table consists of the detection result of the image stream, the values in each row represent the detection result of a single image. Table 4.1 shows the raw result when the robot is at position (0, -4) and is trying to detect the bus by using the SSD pre-trained model.

category	**score**
none	0.0
none	0.0
none	0.0
...	...
person	0.51
person	0.65
person	0.76
...	...
none	0.0
none	0.0

Table 4.1: Raw detection result when robot detects person at position (0, -4) by using SSD pre-trained model.

4.2 Detection speed

In the experiments, we set the robot to rotate at a constant angular velocity of 0.182 rad/s, so the time for the robot to rotate is equal in each experiment. Therefore, we can calculate the length of the image stream obtained by using different CNN models to measure the detection speed of using different models. After we process the raw results by using Python library pandas[1], we can generate Tables 4.2 and 4.3 which represent the length of the image stream for each position when using the SSD model and the YOLO V3 model to try to detect each object. The columns represent which object the robot is trying to detect. The rows indicate the coordinate position where the robot is trying to detect the object. In an row header, the first number represents the abscissa, the second number represents the ordinate, and the letter "m" is the abbreviation of minus. We can see the length of the image stream by checking the object and the corresponding position. For example, when using the SSD model to detect the bus at position (4, 0), the image stream length is 460.

	car	**bus**	**person**	**chair**	**suitcase**	**monitor**
m6_0	415	439	438	442	424	426
m4_0	380	434	417	406	415	417
4_0	374	460	424	425	418	426
6_0	368	402	452	446	422	457
0_6	385	435	412	421	400	433
0_4	366	441	423	440	404	426
0_m4	372	404	433	443	443	442
0_m6	374	420	413	409	449	431

Table 4.2: The length of the image stream at each position when using SSD model to try to detect each object.

	car	**bus**	**person**	**chair**	**suitcase**	**monitor**
m6_0	188	174	187	179	198	186
m4_0	173	182	183	178	192	179
4_0	173	212	174	181	183	177
6_0	170	169	182	182	178	166
0_6	167	177	178	202	172	184
0_4	165	167	183	188	182	179
0_m4	168	161	187	183	183	174
0_m6	166	180	181	181	178	177

Table 4.3: The length of the image stream at each position when using YOLO V3 model to try to detect each object.

We can make observations from Tables 4.2 and 4.3 that the image stream length of SSD is always longer than YOLO V3 when the object and position are the same. Moreover, we can get the average stream lengths when using SSD and YOLO V3 are 419 and 179 by using the following equation:

$$avg\_length = \sum_{pos=m\_6_0}^{0\_m6} \sum_{obj=car}^{bus} length(pos, obj)$$

[1] `https://pandas.pydata.org/`, Accessed May 6 2022.

In the experiment, the angular velocity of the robot is 0.182 rad/s, and the angle that the robot rotated is $90 \times 4 = 360$ degrees, which is $\frac{\pi}{2} \times 4 = 2\pi$ radians so the robot spent 34.5 seconds on rotation. Thus, we can know the average detection speed of the SSD model is 12 images per second and the average detection speed of the YOLO V3 model is around 5 images per second. The detection speed of the SSD model is 2 times faster than the YOLO V3 model.

We can summarise from the above calculations and observations that deploying a simple structured CNN model on ROS enables the robot to detect objects faster. Conversely, deploying a CNN model with a complex structure on ROS will result in slower object detection by the robot.

4.3 Sensitivity

The sensitivity evaluation is to evaluate whether the the detection node loaded with a CNN model is sensitive enough to detect when the camera captures an object. This part of the evaluation is intended to explore the sensitivity of the detection node, so we consider the node as having good sensitivity as long as the node can make a prediction when an object is in the range of the camera, regardless of whether the node can correctly predict the category of object.

Before starting the evaluation, we need to do some processing on the raw results. In some rows of the original tables, e.g., Table 4.1, the category is none and the score is 0.0. We consider such results to be invalid and remove them from the original table. There are two reasons for invalid results:

1. The detection node was not able to recognise there is an object if the object was in the range of the camera.
2. The object was not in the range of the camera. In this instance, the detection node could have been able to detect the presence of the object. However, as the angle of the camera kept changing during the rotation of the robot, the range that the camera can capture the image was changing as well. Thus, in some periods of the rotation, the detection node published results with the none category and the 0.0 score. Figure 4.1 shows two-time points when the object is not in the range of the camera.

After eliminating the invalid results from all original result tables, we have 48 processed tables. In the previous section, we know that the time for the robot to complete the entire rotation is 34.5 seconds. If the detection node is sensitive enough, the GUI launched by the detection node to display the detection results should normally display the detection result a few times in the entire rotation (regardless of a correct or an incorrect prediction). During the entire rotation process, we stipulate that the GUI displays the detection results at least once every three seconds on average. Therefore, in the original result table, there should be at least 11 valid results, that is, if the detection node is sensitive for an object at a certain position, the number of rows is at least 11 in a processed table. For convenience, we set the minimum number of rows as 10. If the number of rows in a processed table is less than 10, we can say that the detection node is not sensitive to make object detection at that position, and vice versa. Thus, we produce the following Table 4.4 and Table 4.5. The definitions of column and row headers of Table 4.4 and Table 4.5 are the same as Table 4.2 and Table 4.3. The boolean value in the cells of Tables 4.4 and 4.5 represent whether the model was sensitive to detecting an object at a certain position. The value *True* means sensitive and the value *False* means not sensitive. For example, in Table 4.4, the value at the cell

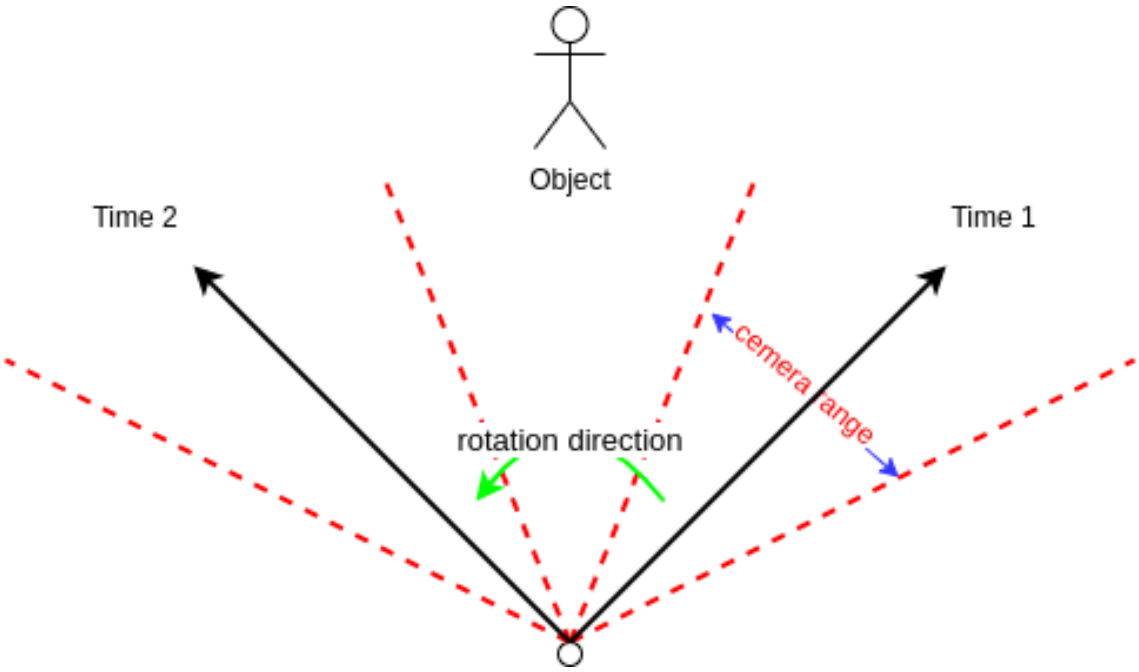

Figure 4.1: The two-time points when the object is not in the range of the camera. The black arrows represent the front of the camera, the red dotted line on both sides of the black arrow represents the range of the camera, and the green arrow represents the rotation is counterclockwise.

with column "chair" and row "m6_0" is *False*, which means that the detection node loaded with the SSD model was not sensitive to detecting the chair at the coordinate position (-6, 0).

	car	**bus**	**person**	**chair**	**suitcase**	**monitor**
m6_0	True	True	True	False	True	False
m4_0	True	True	True	True	True	False
4_0	True	True	True	True	True	False
6_0	True	True	True	False	True	False
0_6	True	True	True	True	True	True
0_4	True	True	True	True	True	True
0_m4	True	True	True	True	True	True
0_m6	True	True	True	True	True	True

Table 4.4: The sensitivity of the detection node at each position when using the SSD model to try to detect each object.

Based on the results from Table 4.4 and Table 4.5 we can observe that both CNN models are not sensitive to detecting the "monitor" object. The reason for this could be the object model itself, or the CNN model was not well trained for that object. Moreover, we can count the number of *True* in the table to separately calculate the likelihood that the two detection nodes are able to predict objects that appear in the camera throughout the experiments.

$$P(SSD) = \frac{48-6}{48} \times 100\% = 87.5\%$$

$$P(YOLOV3) = \frac{48-13}{48} \times 100\% = 73.0\%$$

Thus, we can comment that when the object appears within the range of the camera. In 87.5%

	car	bus	person	chair	suitcase	monitor
m6_0	True	True	True	False	True	True
m4_0	True	True	True	True	False	False
4_0	True	True	True	True	False	False
6_0	True	True	True	True	False	True
0_6	True	False	True	True	False	False
0_4	True	True	True	True	False	False
0_m4	True	True	True	True	True	False
0_m6	True	True	True	True	True	False

Table 4.5: The sensitivity of the detection node at each position when using the YOLO V3 model to try to detect each object.

of the cases, the detection node loaded with the SSD model can detect the presence of the object accordingly, and in 73.0% of the cases, the detection node loaded with the YOLO V3 model can detect the presence of objects accordingly.

4.4 Detection accuracy

The detection accuracy is the proportion in the valid results that the detection node made the correct prediction for the object category in one experiment. In each processed result table of the previous section, with a detection node at a position for detecting an object, we can count the number of correct predictions by judging whether the value of the category in each row is consistent with the object name. Thus, we can make two new Tables 4.6 and 4.7. These two tables represent the detection accuracy of the detection node loaded with SSD and YOLO V3 model at each position for each object. The definitions of column and row headers are the same as in previous table. The value in the cell represents the detection accuracy. If the value is 0.0000, then it represents there is no correct prediction in that experiment. For example, when using the detection node loaded with the SSD model, we can say the detection accuracy at coordinate position (-6, 0) for the object "person" is 1.0, and there is no correct prediction for the object "chair" at the coordinate position (-6, 0).

	car	bus	person	chair	suitcase	monitor
m6_0	0.5371	0.4186	1.0000	0.0000	0.0000	0.0000
m4_0	0.0060	0.5681	0.9962	1.0000	0.0000	0.0000
4_0	0.4014	0.3776	1.0000	0.7692	0.0294	0.0000
6_0	0.0329	0.5484	0.9930	0.0000	0.0000	0.0000
0_6	0.0096	0.0000	1.0000	0.0000	0.0000	0.0000
0_4	0.0492	0.0000	0.9891	0.0000	0.0000	0.0000
0_m4	0.0000	0.0000	0.9714	0.0000	0.5000	0.0000
0_m6	0.1301	0.2391	0.9937	0.2212	0.0662	0.0000
avg_acc	0.1301	0.2391	0.9937	0.2212	0.0662	0.0000

Table 4.6: The detection accuracy of the detection node loaded with SSD model in each experiment.

We can conclude from the results shown in Tables 4.6 and 4.7 that the detection node does not have good accuracy on some objects, especially on the detection of the monitor, suitcase, and bus objects. The average accuracy of the SSD detection node and the YOLO V3 detection node

	car	bus	person	chair	suitcase	monitor
m6_0	0.9871	0.8687	1.0000	0.8000	1.0000	0.0000
m4_0	1.0000	0.8927	1.0000	0.9839	0.0000	0.0000
4_0	1.0000	0.6492	1.0000	0.8917	0.0000	0.0000
6_0	1.0000	0.4730	1.0000	0.9910	0.0000	0.0000
0_6	1.0000	0.0000	1.0000	0.9767	1.0000	0.0000
0_4	0.6800	0.0000	1.0000	0.8247	0.0000	0.0000
0_m4	1.0000	0.0000	1.0000	0.9213	1.0000	1.0000
0_m6	1.0000	0.0000	1.0000	1.0000	1.0000	0.0000
avg_acc	0.9584	0.3604	1.0000	0.9237	0.5000	0.2500

Table 4.7: The detection accuracy of the detection node loaded with YOLO V3 model in each experiment.

can be calculated with the following equation:

$$acc(model), model \in \{SSD, YOLO\_V3\} = \frac{\sum_{k,k\in\{objects\}} avg\_acc(k)}{6}$$

The average accuracy of the SSD detection node is 0.275, which is very low. The average accuracy of the YOLO V3 detection node is 0.665, much better than the SSD detection node. Thus, loading a CNN model with a more complex model on the detection node might be able to improve the detection accuracy.

4.5 Detection confidence

In the result tables, besides the column "category", there is a column named "score", which represents the probability of the correct prediction made by the loaded CNN model, also known as confidence. Thus, in this section, we explore the detection confidence when the detection node made a correct prediction, and we try to infer the effect of the model on the confidence of detected nodes by comparing the confidence of the two CNN models.

Using the tables processed in Section 4.3, we are able to calculate the average confidence at each time of the experiment by calculating the mean of all correct prediction's confidence. Therefore, we get the average confidence of the SSD detection node and YOLOV3 detection node in each experiment as shown in Table 4.8 and Table 4.9. The column and row headers are the same as in previous table. The value in each cell represents the average confidence when the CNN model of the detection node has made a correct prediction of an object at a certain position. The value of each cell in the last row represents the average confidence of the detection node for the same object at 8 different positions.

Based on Tables 4.8 and 4.9, we can observe that when the YOLO V3 detection node made a correct prediction, its confidence is high. Thus, the detection node loaded with a complex structure CNN model can produce a relatively reliable prediction. With the SSD detection node, even if it made a correct prediction, its confidence is just over 50% (on average). This value indicates that the predicted results are not very reliable. Confidence values are especially relevant when the robot needs the prediction results with high confidence value to make a decision. In these scenarios, using prediction results with low confidence may lead to accidents or failures.

	car	bus	person	chair	suitcase	monitor
m6_0	0.6591	0.5496	0.6401	0.0000	0.0000	0.0000
m4_0	0.5750	0.5707	0.7143	0.5445	0.0000	0.0000
4_0	0.6474	0.5727	0.6932	0.5350	0.5000	0.0000
6_0	0.5529	0.5906	0.5600	0.0000	0.0000	0.0000
0_6	0.5400	0.0000	0.5525	0.0000	0.0000	0.0000
0_4	0.5689	0.0000	0.6252	0.0000	0.0000	0.0000
0_m4	0.0000	0.0000	0.6182	0.0000	0.5370	0.0000
0_m6	0.5600	0.0000	0.5545	0.0000	0.0000	0.0000
avg_confi	0.5129	0.2855	0.6198	0.1349	0.1296	0.0000

Table 4.8: The average confidence of the detection node loaded with SSD model in each experiment.

	car	bus	person	chair	suitcase	monitor
m6_0	0.9023	0.7847	0.9513	0.5825	0.6496	0.0000
m4_0	0.9542	0.9334	0.9489	0.5825	0.6496	0.0000
4_0	0.9797	0.8780	0.9792	0.9445	0.0000	0.0000
6_0	0.9861	0.7560	0.9919	0.8892	0.0000	0.0000
0_6	0.9291	0.0000	0.9945	0.9195	0.6375	0.0000
0_4	0.8254	0.0000	0.9709	0.8060	0.0000	0.0000
0_m4	0.9281	0.0000	0.9886	0.7030	0.5960	0.0000
0_m6	0.9477	0.0000	0.9960	0.8676	0.6524	0.0000
avg_confi	0.9316	0.4190	0.9777	0.7868	0.3984	0.0000

Table 4.9: The average confidence of the detection node loaded with YOLO V3 model in each experiment.

Chapter 5

Conclusion

In this final chapter, we first summarise our contributions and offer some concluding remarks regarding the results observed in the experiments. Afterwards, in view of the limitations of this project, we describe some interesting extensions to our prototype and some possible solutions for optimisation in the future.

5.1 Conclusion

In this dissertation, we first elaborated on the composition of ROS applications, the concepts of simulation environments, the concepts of deep learning, and the composition of CNNs. Secondly, we described the state-of-art on image detection field based on CNNs and the related work of deploying deep learning technology in ROS. Thirdly, we describe design and implementation concepts of our prototype for deploying deep learning technology in ROS. Then, we carried out the experiments by deploying two CNN pre-trained models. Finally, we compared and analysed the results of the experiments to further explore the feasibility and achievability of deploying deep learning technology in ROS.

Observing the results of the experiments, we can conclude that it is feasible to deploy deep learning technology run on ROS 2. This is relevant because this is not something that has been actively explored, since most deployments of deep learning in ROS has been based around ROS 1. From the evaluation of detection speed, we can conclude that the detection speed of the detection node is related to the complexity of the CNN model. The more complex the CNN model the node loads, the slower the detection speed it has. From the evaluation of sensitivity, we can conclude that the CNN model with a simpler structure could make the detection node more easily sense the presence of the object. Inversely, a node with a complex structure may not be able to respond to the presence of the object. By evaluating the detection accuracy and confidence, we summarise that the prediction accuracy of the node with a simpler structure CNN model is lower, as well as the corresponding confidence even if the prediction is correct. Oppositely, the node with a more complex and deeper structure has much better performance.

Combining the above conclusions, we can further conclude that even though it is feasible to deploy the deep learning technology in ROS, we still need to consider the complexity of the model, the prediction accuracy and the corresponding confidence of the model when deploying. When solving real-world problems, decisions resulting from low precision and low confidence can cause significant losses, and low operating speeds can lead to inefficient problem-solving. Therefore, we need to find a balance between them in order to deploy deep learning technology on ROS to solve

practical problems.

5.2 Future Work

During the development of the prototype and the experiments, some optimisation issues and possible extensions were identified. We summarise three aspects and propose some future work related to our findings.

First, the diversity of models is limited at the moment, since we only used two pre-trained CNN models for the experiments. In the future, we can use other pre-trained deep learning models which have different structures so that we can explore the support of ROS to the different deep learning models with varied types of architecture.

Second, in our experiment the scenario is simple. We only designed one scenario for single object detection. In the future, we can experiment with complex application scenarios, such as hospitals, streets, factories, etc. Additionally, we can try to use our ROS node to detect multiple objects at the same time. This would allow us to we explore the performance limit of the CNN model when running on ROS by comparing it with the the number of objects in the environment. For example, we can measure how how many objects in the scene it takes for the CNN model to start failing.

Third, in our detection results, we found that the detection node was not able to correctly detect the object when the object could be captured by the camera. We assume that there might be two reasons for this issue, one is the loaded CNN model was not well trained so it could not be able to make predictions according to the object feature. The other possible reason is that the image style of objects we used in the simulated environment is different from the image style which was used to train the model. The data used to train the model is from the COCO data set, the images were captured from the real world, while the simulated objects we used follow a 3D animation style. Thus, future work includes either carrying out the experiments on a real robot to detect objects in the real world or use transfer learning methods to train an optimised model with images of 3D Gazebo models. In this case, we can further explore whether the low detection accuracy and confidence are due to insufficient CNN model training, or the image style of the object. Through this method, we can also explore how to improve the accuracy of the CNN model so that we can deploy a simple structure model on ROS.

Reference

[1] Addison, Sears, C. (2020). Automatic Addison - Build the future. `https://automaticaddison.com/robot-state-publisher-vs-joint-state-publisher/` Accessed April 26 2020.

[2] Albawi, S., Mohammed, T. A., and Al-Zawi, S. (2017). Understanding of a convolutional neural network. In *2017 international conference on engineering and technology (ICET)*, pages 1–6. Ieee.

[3] Apostolopoulos, J. G., Tan, W.-t., and Wee, S. J. (2002). Video streaming: Concepts, algorithms, and systems. *HP Laboratories, report HPL-2002-260*, pages 2641–8770.

[4] Ardabili, S., Mosavi, A., Dehghani, M., and Várkonyi-Kóczy, A. R. (2019). Deep learning and machine learning in hydrological processes climate change and earth systems a systematic review. In *International conference on global research and education*, pages 52–62. Springer.

[5] Banks, J., II, J. S. C., Nelson, B. L., and Nicol, D. M. (2010). *Discrete-Event System Simulation, 5th New Internatinal Edition*. Pearson Education.

[6] Bhardwaj, A., Di, W., and Wei, J. (2018). *Deep Learning Essentials: Your hands-on guide to the fundamentals of deep learning and neural network modeling*. Packt Publishing Ltd.

[7] Chang, Y.-H., Chung, P.-L., and Lin, H.-W. (2018). Deep learning for object identification in ros-based mobile robots. In *2018 IEEE International Conference on Applied System Invention (ICASI)*, pages 66–69.

[8] Chollet, F. (2021). *Deep learning with Python*. Simon and Schuster.

[9] Deng, L. and Platt, J. (2014). Ensemble deep learning for speech recognition. In *Proc. interspeech*.

[10] Girshick, R., Donahue, J., Darrell, T., and Malik, J. (2014). Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 580–587.

[11] Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep learning*. MIT press.

[12] He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778.

[13] Hijazi, S., Kumar, R., Rowen, C., et al. (2015). Using convolutional neural networks for image recognition. *Cadence Design Systems Inc.: San Jose, CA, USA*, 9.

[14] Hongkun, Y., Chen, C., Xianzhi, D., Yeqing, L., Abdullah, R., Le, H., Pengchong, J., Fan, Y., Frederick, L., Jaeyoun, K., and Jing, L. (2020). TensorFlow Model Garden. `https://github.com/tensorflow/models` Accessed April 26 2020.

[15] Hopfield, J. J. (1982). Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences of the United States of America*, 79(8):2554–2558.

[16] Kim, P. (2017). Convolutional neural network. In *MATLAB deep learning*, pages 121–147. Springer.

[17] Koenig, N. and Howard, A. (2004). Design and use paradigms for gazebo, an open-source multi-robot simulator. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*, volume 3, pages 2149–2154 vol.3.

[18] LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. *nature*, 521(7553):436–444.

[19] Lin, T.-Y., Maire, M., Belongie, S., Hays, J., Perona, P., Ramanan, D., Dollár, P., and Zitnick, C. L. (2014). Microsoft coco: Common objects in context. In *European conference on computer vision*, pages 740–755. Springer.

[20] Liu, W., Anguelov, D., Erhan, D., Szegedy, C., Reed, S., Fu, C.-Y., and Berg, A. C. (2016). Ssd: Single shot multibox detector. In *European conference on computer vision*, pages 21–37. Springer.

[21] Meyer, J., Sendobry, A., Kohlbrecher, S., Klingauf, U., and Stryk, O. v. (2012). Comprehensive simulation of quadrotor uavs using ros and gazebo. In *International conference on simulation, modeling, and programming for autonomous robots*, pages 400–411. Springer.

[22] Mingo Hoffman, E., Traversaro, S., Rocchi, A., Ferrati, M., Settimi, A., Romano, F., Natale, L., Bicchi, A., Nori, F., and Tsagarakis, N. G. (2014). Yarp based plugins for gazebo simulator. In *International Workshop on Modelling and Simulation for Autonomous Systems*, pages 333–346. Springer.

[23] Open Robotics, O. (2022a). Gazebo. `https://github.com/ros2` Accessed April 4, 2022.

[24] Open Robotics, O. (2022b). Gazebo - a dynamic multi-robot simulator. `https://github.com/osrf/gazebo` Accessed April 4, 2022.

[25] Open Robotics, O. (2022c). Ros - robot operating system. `https://www.ros.org/` Accessed April 4, 2022.

[26] Open Robotics, O. (2022d). Ros cores. `https://github.com/ros/ros` Accessed April 4, 2022.

[27] Open Robotics, O. (2022e). Version 2 of the robot operating system (ros) software stack. `https://github.com/ros2` Accessed April 4, 2022.

[28] Peterson, J. L. and Silberschatz, A. (1985). *Operating system concepts*. Addison-Wesley Longman Publishing Co., Inc.

[29] Purwins, H., Li, B., Virtanen, T., Schlüter, J., Chang, S.-Y., and Sainath, T. (2019). Deep learning for audio signal processing. *IEEE Journal of Selected Topics in Signal Processing*, 13(2):206–219.

[30] Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., Ng, A. Y., et al. (2009). Ros: an open-source robot operating system. In *ICRA workshop on open source software*, page 5. Kobe, Japan.

[31] Redmon, J., Divvala, S., Girshick, R., and Farhadi, A. (2016). You only look once: Unified, real-time object detection. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 779–788, Los Alamitos, CA, USA. IEEE Computer Society.

[32] Redmon, J. and Farhadi, A. (2017). Yolo9000: better, faster, stronger. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7263–7271.

[33] Redmon, J. and Farhadi, A. (2018). Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767*.

[34] Saily, S. (2020). Convolutional Neural Network: An Overview. `https://www.analyticsvidhya.com/blog/2022/01/convolutional-neural-network-an-overview/` Accessed April 26 2020.

[35] Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., and Chen, L.-C. (2018). Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4510–4520.

[36] Simonyan, K. and Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.

[37] Singh, S. P., Kumar, A., Darbari, H., Singh, L., Rastogi, A., and Jain, S. (2017). Machine translation using deep learning: An overview. In *2017 international conference on computer, communications and electronics (comptelix)*, pages 162–167. IEEE.

[38] Su, H., Zhang, Y., Li, J., and Hu, J. (2016). The shopping assistant robot design based on ros and deep learning. In *2016 2nd International Conference on Cloud Computing and Internet of Things (CCIOT)*, pages 173–176.

[39] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. (2015). Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9.

[40] Tan, L., Huangfu, T., Wu, L., and Chen, W. (2021). Comparison of retinanet, ssd, and yolo v3 for real-time pill identification. *BMC medical informatics and decision making*, 21(1):1–11.

[41] Xing, Yuan, H. (2020). Explain MobileNetV2 in detail. `https://zhuanlan.zhihu.com/p/98874284` Accessed 02 May 2020.

Appendix A

Manual and Documentation of ROS Application

In appendix A, we will introduce the steps of how to configure the computer environment so that we can run the our project experiment code from scratch. It including ROS Foxy environment installation, essential ROS and Python packages/libraries installation. In the instruction of carrying out the experiment, we described the function of each package as well.

A.1 ROS Foxy Installation

In this section, we will introduce the installation steps of ROS Foxy from scratch. As well as the step of verifying whether the installation is successful.

A.1.1 Prerequisites

- A local computer with Ubuntu 20.04 operating system.
- Python 3.8 has been installed on the computer.
- CUDA and CUDNN has been installed on the computer.

A.1.2 Set locale

Firstly, we have to set the locale to ensure that our computer locale supports ‘UTF-8‘ by type the commands shown in list A.1 to the terminal:

```
$ locale  # check for UTF-8

$ sudo apt update && sudo apt install locales
$ sudo locale-gen en_US en_US.UTF-8
$ sudo update-locale LC_ALL=en_US.UTF-8 LANG=en_US.UTF-8
$ export LANG=en_US.UTF-8

$ locale  # verify settings
```

Listing A.1: Set the locale for computer

A.1.3 Setup installation sources

In this step, we have to add the sources of ROS Foxy, so that the computer would be able to know where to download the installation package and executing the installation. Typing the commands shown in list A.2 to the terminal for adding the sources:

```
$ sudo apt update && sudo apt install curl gnupg2 lsb-release
$ sudo curl -sSL https://raw.githubusercontent.com/ros/rosdistro/master/ros
    .key  -o /usr/share/keyrings/ros-archive-keyring.gpg
```

```
$ echo "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/
    keyrings/ros-archive-keyring.gpg] http://packages.ros.org/ros2/ubuntu $
    (source /etc/os-release && echo $UBUNTU_CODENAME) main" | sudo tee /etc
    /apt/sources.list.d/ros2.list > /dev/null
```

Listing A.2: Set the source for computer

A.1.4 Install ROS Foxy packages

In this step, we simply type the following commands shown in list A.3 to install ROS Foxy on the local computer. During the installation, the output should be displayed in the terminal.

```
1 $ sudo apt update
  $ sudo apt install ros-foxy-desktop
```

Listing A.3: Install ROS foxy Desktop packages

A.1.5 Environment setup

The default install path of ROS foxy is '/opt/ros/foxy/'. Thus, to let our computer is able to reads and executes commands from the ROS. We can type the following command shown in list A.4 to the terminal:

```
1 $ source /opt/ros/foxy/setup.bash
```

Listing A.4: Set up the environment

Alternatively, to avoid type the source command every time when we restart the computer, or launch a new terminal. We can edit the '.bashrc' file, in this case, every time when we open a new terminal, the shell would automatically execute the above command. Using command in list A.5 to edit the '.bashrc' file and paste the above command at the end of bash file.

```
1 $ vim ~/.bashrc
```

Listing A.5: Type the command to edit the bash file

A.1.6 Verify the installation

By far, we have already installed the ROS foxy on the local computer. We can launch two demo nodes separately in two terminals to verify whether ROS can work on the computer by typing the commands in list A.6.

```
1 $ source /opt/ros/foxy/setup.bash
  $ ros2 run demo_nodes_cpp talker
3
  $ source /opt/ros/foxy/setup.bash
5 $ ros2 run demo_nodes_py listener
```

Listing A.6: Verify whether the ROS can work

After executing the command in list A.6, if the installation is successful, we should see the output message "send" in the publisher node and the output message "received" in the other terminal. If the installation is failed, please check the ROS official documentation for more help. `https://docs.ros.org/en/foxy/Installation/Ubuntu-Install-Debians.html`

A.2 Installation of Virtual Environment

By now, Anaconda does not very compatible support for ROS 2. While Virtual Environment has a better support for ROS 2. Thus, we will install the virtual environment to run our project.

A.2.1 Install virtualenv and virtualenvwrapper packages

The packages of ”virtualenv” and ”virtualenvwrapper” can be installed via pip. Since the pip is already configured on the local machine. We simply execute commands shown in the list A.7. The ”virtualenv” packages include the cores of build a virtual development on the local computer, the ”virtualenvwrapper” packages provides command line to let the developers can manage the virtual environment conveniently.

```
$ pip3 install virtualenv
$ pip3 install virtualenvwrapper
```

Listing A.7: Install the ”virtualenv” and ”virtualenvwrapper” packages

A.2.2 Configure virtual environment

When the installation is completed. We edit bash file to make the commands could be able to work with the shell environment of computer. Simply open the editing using command in list A.5. Then paste the commands shown in list A.8 at the end of bash file.

```
export WORKON_HOME=$HOME/.virtualenvs
export PROJECT_HOME=$HOME/Devel
source /usr/local/bin/virtualenvwrapper.sh
export VIRTUALENVWRAPPER_PYTHON=/usr/bin/python3$
```

Listing A.8: Commands pasted to bash file to configure virtual environment

Save the changes and quit from eiding. Then use the command in list A.9 to refresh the bash file in the terminal.

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

Listing A.9: Refresh the bash file

If we see the output in the terminal is similar with the output in list A.10. It means that we have successfully installed and configured virtual environment.

```
virtualenvwrapper.user_scripts creating /home/jey/.virtualenvs/premkproject
virtualenvwrapper.user_scripts creating /home/jey/.virtualenvs/
    postmkproject
virtualenvwrapper.user_scripts creating /home/jey/.virtualenvs/initialize
virtualenvwrapper.user_scripts creating /home/jey/.virtualenvs/
    premkvirtualenv
virtualenvwrapper.user_scripts creating /home/jey/.virtualenvs/
    postmkvirtualenv
virtualenvwrapper.user_scripts creating /home/jey/.virtualenvs/
    prermvirtualenv
...
```

Listing A.10: Successful output for virtual environment installation and configuration

A.3 Experiment

In this section, we will firstly introduce ROS-packages we need to use in the experiment. Secondly, we introduce how to create a work space and build a new virtual environment for the experiment. Thirdly, we introduce how to carry out the experiment step by step.

A.3.1 Install ROS-packages

As we use Gazebo as the simulator in the experiment, so we have to install some packages so that Gazebo can interact with ROS. In the terminal, typing the command in list A.11 to install those packages.

```
$ sudo apt install ros-foxy-gazebo*
```

Listing A.11: Install additional ROS-packages for better interaction between ROS and Gazebo

A.3.2 Building workspace and virtual environment

We direct to the directory where you want to build the workspace for ROS. Then using the command in list A.12 to create a folder as the workspace of our experiment and enter it.

```
$ mkdir -p ~/colcon_venv/src
$ cd ~/colcon_venv/
```

Listing A.12: Create ROS workspace

Secondly, in the workspace, we create a virtual environment named ”env” and activate it by executing commands shown in list A.13.

```
# Make a virtual env and activate it
$ virtualenv -p python3 ./venv
$ source ./venv/bin/activate
# Make sure that colcon does not try to build the venv
$ touch ./venv/COLCON_IGNORE
```

Listing A.13: Create virtual environment in ROS workspace

A.3.3 Install Python packages in virtual environment

In this step, we install some essential python packages into the virtual environment to assist the execution of experiment. Using the command in list A.14

```
(venv) pip3 install python3-opencv yolov34py-gpu
```

Listing A.14: Install Python packages into the virtual environment

A.3.4 Paste the experiment packages and compiling

Firstly, unzipping the compressed file, then we get five packages. Which are:

- **experiment** - The package contains the node files of the save node and the control node.
- **project_interfaces** - The package contains the customised interfaces used for node communication in the experiment
- **recogniser_ssd** - Where the detection node loaded with the SSD model is, as well as the weight file of model, configuration files.

- **recogniser_yolo** - Where the detection node loaded with the YOLO V3 model is, as well as the weight file of model, configuration files.
- **spawning** - Where the spawning node is.

We copy each package into the path "path_of_workspace/src/" by executing the command in list A.15. Alternatively, we can paste these packages by dragging the file icon to the destination path in the GUI.

```
(venv) copy -rf path_of_package/package_name path_of_workspace/src/
```

Listing A.15: Paste package to ROS workspace source directory

Afterwords, we execute commands shown in list A.16 to direct into the workspace and compile the packages we just pasted.

```
(env) cd path_of_workspace
(env) source ~/.bashrc
(env) colcon build
```

Listing A.16: Direct to workspace and compiling the packages

A.3.5 Carrying out the experiment

In each experiment, we execute the ROS program by six steps. As an example, what follows describes how we detect the bus at coordinates (-4, 0) with the detection node loaded with the SSD model and YOLO V3 model.

To monitor the output in each step, we suggest to use multi-windows terminal, such as Terminator (`https://gnometerminator.blogspot.com/p/introduction.html`). Each time when we open a new terminal, we would type the commands shown in list A.17. Otherwise, the execution might be failed.

```
$ source ~/.bashrc
$ source path_of_workspace/venv/bin/activate
$ source path_of_workspace/install/setup.bash
```

Listing A.17: Source bash files when open a new terminal each time

Step 1: Simulate the environment

Firstly, we launch the Gazebo simulated environment by type the command shown in list A.18. Then a Gazebo GUI launched with grids. Figure A.1 shows the Gazebo simulated environment is launched, the figureA.2 shows the corresponding output in the terminal.

```
(env) ros2 launch spawning launch_gazebo_empty_world.launch.py
```

Listing A.18: Step 1 command: launch Gazebo

Step 2: Spawning the object

Secondly, spawning the object at the original point of the spherical coordinates frame by executing the command in list A.19. We can spawn any selected object by executing the command template shown in list A.20. Figure A.3 shows the change in the Gazebo environment and the figure A.4 shows the output in the terminal.

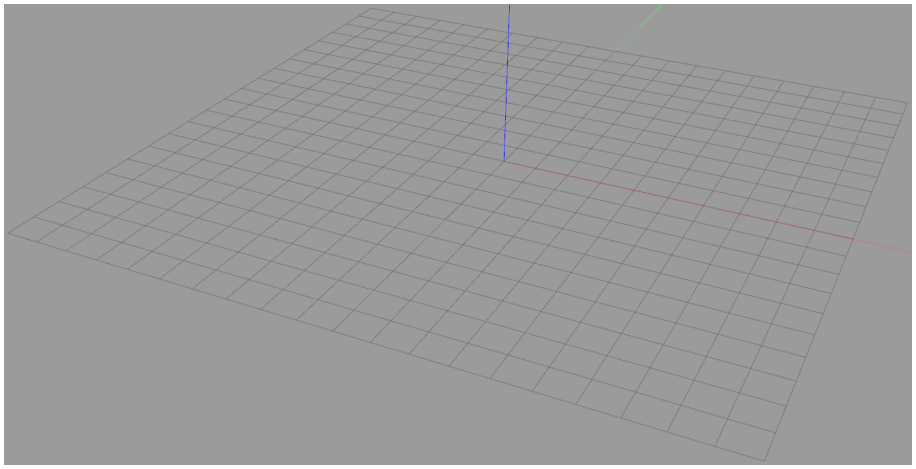

Figure A.1: Step 1: The Gazebo GUI with empty environment

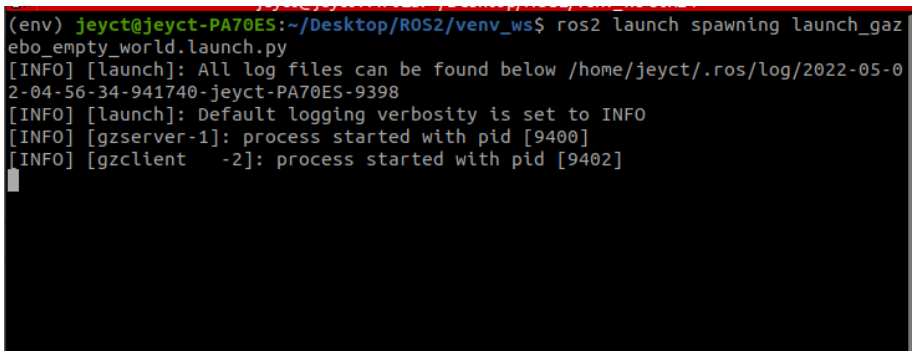

Figure A.2: Step 1: Terminal output of step 1

```
(env) ros2 run spawning spawn_single_item.py bus
```

Listing A.19: Step 2 command: spawning object

```
(env) ros2 run spawning spawn_single_item.py object_name
```

Listing A.20: Spawning any selected object by change the object name

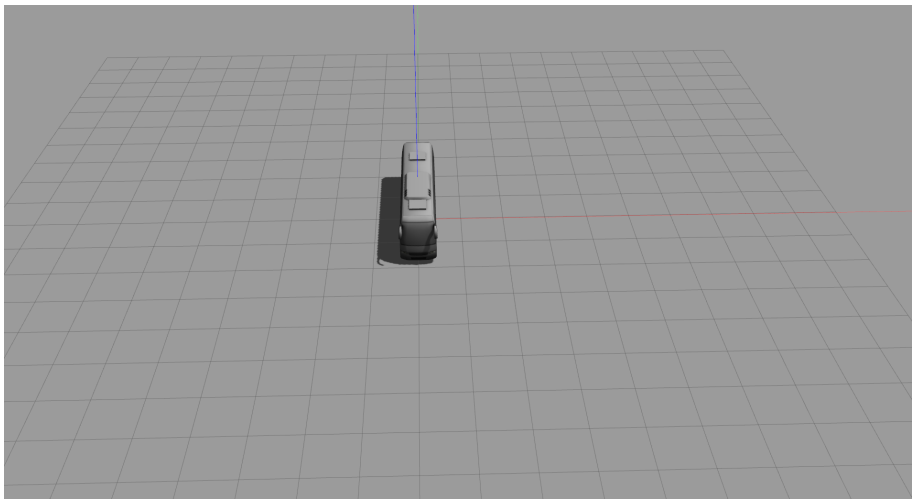

Figure A.3: Step 2: The changes in simulated environment

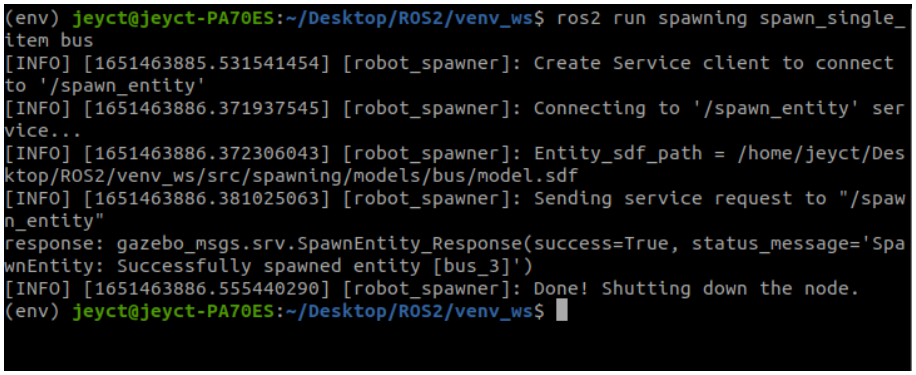

Figure A.4: Step 2: The terminal output of step 2

Step 3: Spawning the robot

In step 3, we spawning the robot at a specific position with the command template shown in list A.21, in this instance, the robot would be spawned at coordinate (-4, 0), so in command A.21, we simply change the ”x” and ”y” to corresponding coordinate. Please note that we use ”m” to represent minus. Figure A.5 shows the robot was spawned at (0, -4), figure A.6 shows the corresponding output in the terminal.

```
(env) ros2 launch spawning launch_robot_x_y.launch.py
```

Listing A.21: Step 3 command: spawning robot

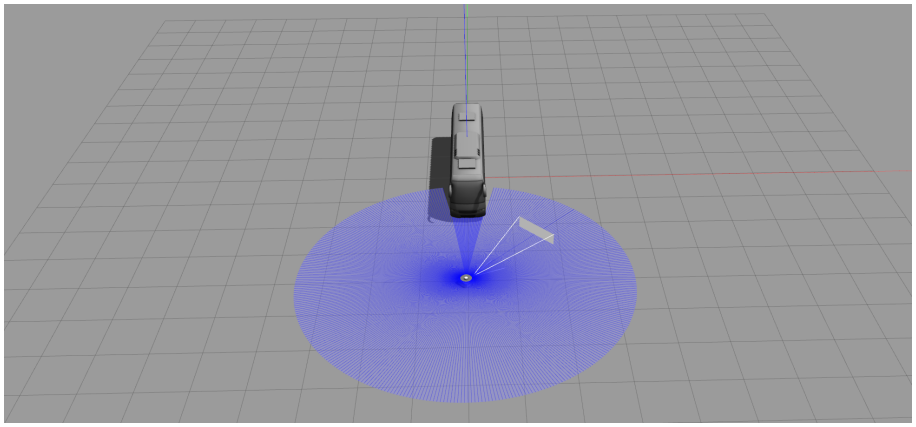

Figure A.5: Step 3: The robot is launched in step

```
(env) jeyct@jeyct-PA70ES:~/Desktop/ROS2/venv_ws$ ros2 launch spawning launch_rob
ot_0_m4.launch.py
[INFO] [launch]: All log files can be found below /home/jeyct/.ros/log/2022-05-0
2-05-00-27-725897-jeyct-PA70ES-10469
[INFO] [launch]: Default logging verbosity is set to INFO
[INFO] [spawn_robot_model-1]: process started with pid [10471]
[spawn_robot_model-1] [INFO] [1651464028.228628445] [robot_spawner]: Create Serv
ice client to connect to '/spawn_entity'
[spawn_robot_model-1] [INFO] [1651464028.281334493] [robot_spawner]: Connecting
to '/spawn_entity' service...
[spawn_robot_model-1] [INFO] [1651464028.281686242] [robot_spawner]: Entity_sdf_
path = /home/jeyct/Desktop/ROS2/venv_ws/src/spawning/bot_models/turtlebot3_waffl
e/model.sdf
[spawn_robot_model-1] [INFO] [1651464028.297482734] [robot_spawner]: Sending ser
vice request to "/spawn_entity"
[spawn_robot_model-1] [INFO] [1651464028.601603895] [robot_spawner]: Done! Shutt
ing down the node.
[INFO] [spawn_robot_model-1]: process has finished cleanly [pid 10471]
(env) jeyct@jeyct-PA70ES:~/Desktop/ROS2/venv_ws$
```

Figure A.6: Step 3: The terminal output

Step 4: Launching the save node

Launching the save node by typing command shown in list A.22. Figure A.7 shows the output in the terminal. In this step, there is no visual change in the simulated environment but the terminal shows the service client is created and waiting for the request from the control node.

```
(env) ros2 run experiment result_saver
```

Listing A.22: Step 4 command: launching the save node

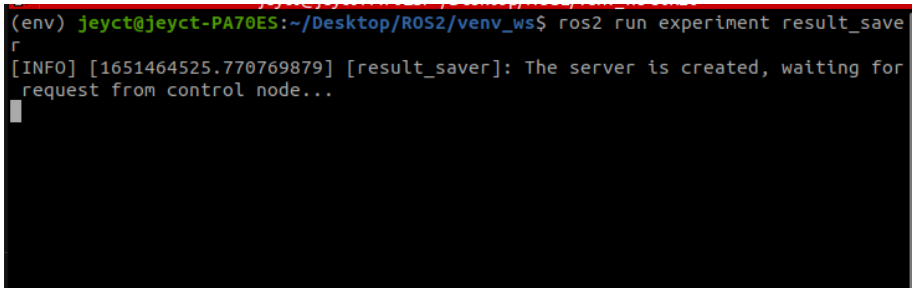

Figure A.7: Step 4: The terminal output

Step 5: Launching the detection node

We launching the detection node loaded with the SSD model by executing command in list A.23, if we want to launch the SSD detection node, then we replace ”model_type” to ”ssd” in command A.23. If we want to launch the YOLO V3 detection node, then we replace ”model_type” to ”yolov3” in command A.23. In this step, a GUI would be launched to show what image the camera captured from the simulated environment. At the same time, the model is applied to detect whether there is an object in the environment and the node would send the detection results to topic detect_result. The output terminal of the detection node is shown that the node is receiving the image from the camera. However, at this time, the subscriber of the save node is disconnected from the publisher of the detection node. Therefore, there is no change in the output terminal of the save node. There is no changes in the simulated environment neither. Figure A.8, A.9, and A.10 shows the GUI, terminal output of the detection node, and the terminal output of the save node.

```
(env) ros2 run recognizer_model_type reg_model_type
```

Listing A.23: Step 5 command: launching the SSD detection node

Step 6: Launching the control node

We launch the control node to start the experiment by type command in list A.24. After the control node is launched, it sends a request to it so that the save node connects with the detection node, figure A.12 shows that the save node starts to receive the detection results from the detection node. When the rotation action is done, the control node sends a request to the save node to let it stop receiving results and save the received results as a CSV file. Afterwards, the control node shutdown itself.Figure A.11 shows the detection node detected the object with the going of rotation. Figure A.13 shows the working process of the control node in the terminal, figure A.14 shows the save node responses to the request of the control node.

```
(env) ros2 run experiment movement_control
```

Figure A.8: Step 5: The launched GUI

```
[INFO] [1651464738.980002318] [image_publisher]: Receiving video frame from Gazebo world...
[INFO] [1651464739.146444931] [image_publisher]: Receiving video frame from Gazebo world...
[INFO] [1651464739.312725812] [image_publisher]: Receiving video frame from Gazebo world...
[INFO] [1651464739.470580207] [image_publisher]: Receiving video frame from Gazebo world...
[INFO] [1651464739.628565159] [image_publisher]: Receiving video frame from Gazebo world...
[INFO] [1651464739.797357944] [image_publisher]: Receiving video frame from Gazebo world...
[INFO] [1651464739.958113419] [image_publisher]: Receiving video frame from Gazebo world...
[INFO] [1651464740.120839945] [image_publisher]: Receiving video frame from Gazebo world...
[INFO] [1651464740.291198155] [image_publisher]: Receiving video frame from Gazebo world...
[INFO] [1651464740.472998969] [image_publisher]: Receiving video frame from Gazebo world...
[INFO] [1651464740.633612957] [image_publisher]: Receiving video frame from Gazebo world...
[INFO] [1651464740.792229511] [image_publisher]: Receiving video frame from Gazebo world...
[INFO] [1651464740.956654347] [image_publisher]: Receiving video frame from Gazebo world...
[INFO] [1651464741.126760706] [image_publisher]: Receiving video frame from Gazebo world...
[INFO] [1651464741.286045098] [image_publisher]: Receiving video frame from Gazebo world...
[INFO] [1651464741.443131344] [image_publisher]: Receiving video frame from Gazebo world...
[INFO] [1651464741.599614468] [image_publisher]: Receiving video frame from Gazebo world...
[INFO] [1651464741.766559462] [image_publisher]: Receiving video frame from Gazebo world...
[INFO] [1651464741.928469769] [image_publisher]: Receiving video frame from Gazebo world...
[INFO] [1651464742.094711016] [image_publisher]: Receiving video frame from Gazebo world...
```

Figure A.9: Step 5: The terminal output of detection node

```
(env) jeyct@jeyct-PA70ES:~/Desktop/ROS2/venv_ws$ ros2 run experiment result_save
r
[INFO] [1651464525.770769879] [result_saver]: The server is created, waiting for
 request from control node...
```

Figure A.10: Step 5: The terminal outoput of save node

Listing A.24: Step 6 command: launching the control node

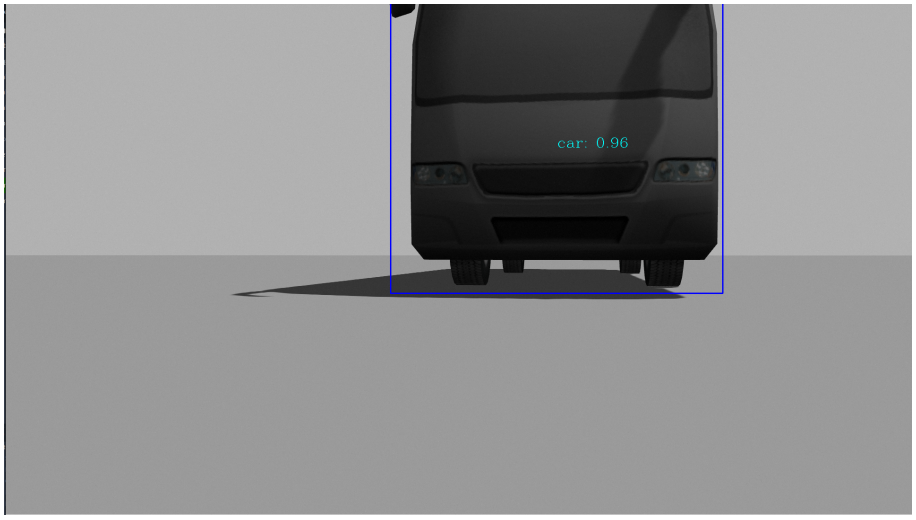

Figure A.11: Step 6: The detection result shown in Gazebo GUI

```
[INFO] [1651464803.947609949] [result_saver]: Get category: car, corresponding score: 0.96
[INFO] [1651464804.169009579] [result_saver]: Get category: car, corresponding score: 0.97
[INFO] [1651464804.411221646] [result_saver]: Get category: car, corresponding score: 0.97
[INFO] [1651464804.621198824] [result_saver]: Get category: car, corresponding score: 0.95
[INFO] [1651464804.835442001] [result_saver]: Get category: car, corresponding score: 0.96
[INFO] [1651464805.052355883] [result_saver]: Get category: car, corresponding score: 0.92
[INFO] [1651464805.276600346] [result_saver]: Get category: car, corresponding score: 0.93
[INFO] [1651464805.472494967] [result_saver]: Get category: car, corresponding score: 0.95
[INFO] [1651464805.677511093] [result_saver]: Get category: car, corresponding score: 0.63
[INFO] [1651464805.885009358] [result_saver]: Get category: car, corresponding score: 0.52
[INFO] [1651464806.110083182] [result_saver]: Get category: car, corresponding score: 0.76
[INFO] [1651464806.332784319] [result_saver]: Get category: car, corresponding score: 0.94
[INFO] [1651464806.529708773] [result_saver]: Get category: none, corresponding score: 0.0
[INFO] [1651464806.746574078] [result_saver]: Get category: none, corresponding score: 0.0
[INFO] [1651464806.948984570] [result_saver]: Get category: car, corresponding score: 0.58
[INFO] [1651464807.158392282] [result_saver]: Get category: car, corresponding score: 0.91
[INFO] [1651464807.366960065] [result_saver]: Get category: car, corresponding score: 0.75
[INFO] [1651464807.584137069] [result_saver]: Get category: car, corresponding score: 0.72
[INFO] [1651464807.782951847] [result_saver]: Get category: car, corresponding score: 0.67
[INFO] [1651464807.989023260] [result_saver]: Get category: car, corresponding score: 0.72
[INFO] [1651464808.189741778] [result_saver]: Get category: car, corresponding score: 0.73
[INFO] [1651464808.395876768] [result_saver]: Get category: car, corresponding score: 0.62
[INFO] [1651464808.617277309] [result_saver]: Get category: car, corresponding score: 0.93
[INFO] [1651464808.832972244] [result_saver]: Get category: car, corresponding score: 0.98
[INFO] [1651464809.054832573] [result_saver]: Get category: car, corresponding score: 0.9
[INFO] [1651464809.262146056] [result_saver]: Get category: car, corresponding score: 0.93
[INFO] [1651464809.468986045] [result_saver]: Get category: car, corresponding score: 0.95
[INFO] [1651464809.669354016] [result_saver]: Get category: car, corresponding score: 0.97
```

Figure A.12: Step 6: The save node starts to receive detection result

By far, we have done one experiment for object bus at coordinate (-4, 0). Similar, by changing the position of robot in step 3, and the CNN model in step 5 for 47 times, we finished all the experiments.

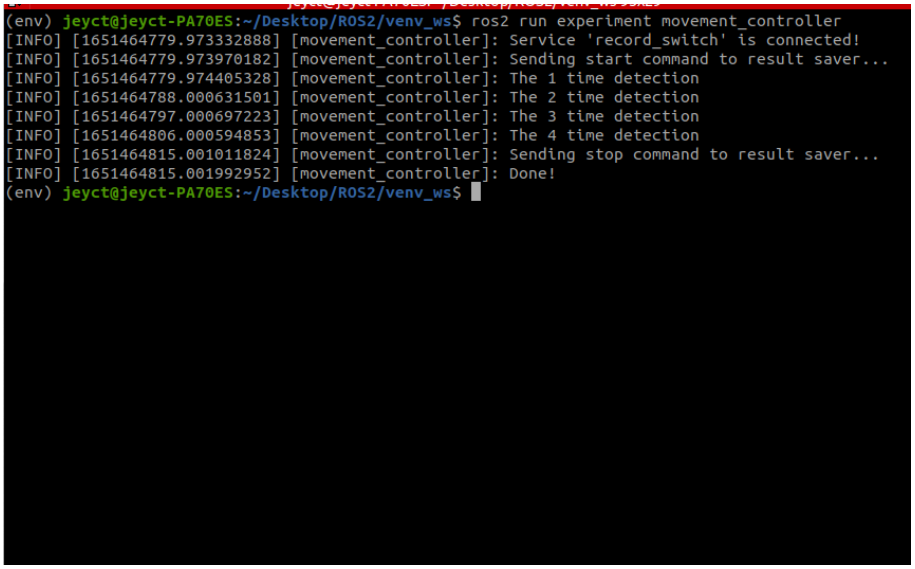

Figure A.13: Step 6: The working process of the control node. The control node sends request to the save node, then execute rotation, when rotation is done, the control node sends request ot the save node, and shutdown itself

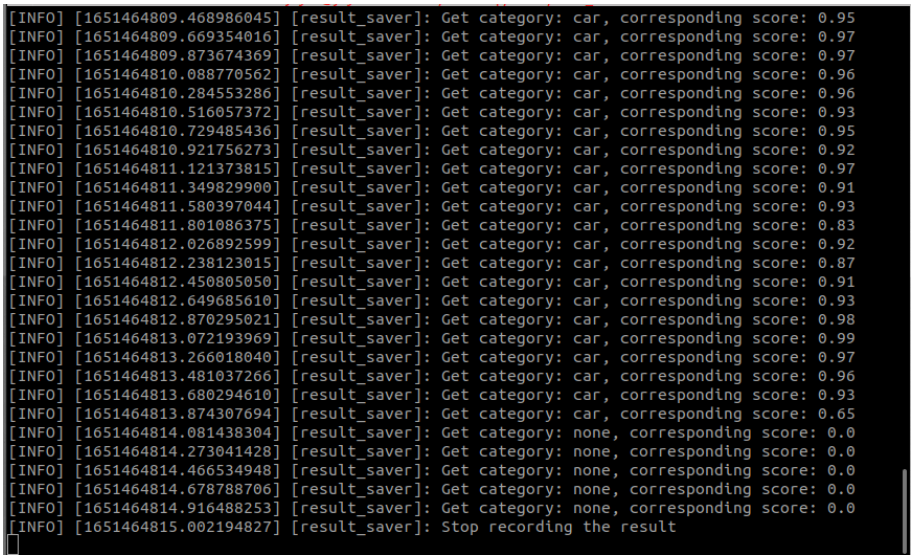

Figure A.14: Step 6: The save node stops receiving results and save the results to CSV file

Appendix B

Evaluation Process

In this appendix B, we show the evaluation process we have written in Jupyter Notebook. For the original file and source code, please check evaluation_process.ipynb in the submission.

1. Result Description

For the same object, the robot will eventually generate 8 csv files from different angles and different distances, because we have 6 objects to be detected so we did $6 \times 8 = 48$ experiments, the total number of files is 48 as well.

The filename of the result file identifies the detected object and the coordinates of the position where the robot detected it. In the file, it contains the results the model detects the target during the rotation. The detection result includes the category of the object to be detected, and the probability of predicting the category, also known as the confidence.

In [15]:
```
import pandas as pd
```

In [16]:
```
df = pd.read_csv('experiment_results_ssd/person/results_performance_person_0_m
```

In [17]:
```
df
```

Out[17]:

	category	score
0	none	0.0
1	none	0.0
2	none	0.0
3	none	0.0
4	none	0.0
...	...	...
428	none	0.0
429	none	0.0
430	none	0.0
431	none	0.0
432	none	0.0

433 rows × 2 columns

2. Model speed

In this experiment, as all the nodes are same. The only major element that affects the detection speed is model. As we know, for an object, the robit rotates same angle at each position so the time of rotation is same. Thus, we will explore the model speed by comparing how many data streams each pre-trained model produced in a fixed time.

In [40]:
```
# Define a function to count the number of results
def result_counter(results, model_type):

```

```
    objects_name = ['car', 'bus', 'person', 'chair', 'suitcase', 'monitor']
    positions = ['m6_0', 'm4_0', '4_0', '6_0', '0_6', '0_4', '0_m4', '0_m6']
    new_df = pd.DataFrame(columns=objects_name, index=positions)

    for result in results:
        obj, position = get_obj_pos(result)

        df = pd.read_csv(result, index_col='Unnamed: 0')
        num_row = df.shape[0]
        new_df.at[position, obj] = num_row

    new_df.to_csv(os.path.join('./process_result', model_type + '_result_count
```

In [41]:
```
ssd_result_names = get_file_names('./experiment_results_ssd')
yolo_result_names = get_file_names('./experiment_results_yolov3')

# Execute the function for ssd and yolov3
result_counter(ssd_result_names, 'ssd')
result_counter(yolo_result_names, 'yolov3')
```

In [42]:
```
df_ssd = pd.read_csv('./process_result/ssd_result_count.csv', index_col='Unnam
df_ssd
```

Out[42]:

	car	bus	person	chair	suitcase	monitor
m6_0	415	439	430	442	424	426
m4_0	380	434	417	406	415	417
4_0	374	460	424	425	418	426
6_0	368	402	452	446	422	457
0_6	385	435	412	421	400	433
0_4	366	441	423	440	404	426
0_m4	372	404	433	443	443	442
0_m6	374	420	413	409	449	431

In [43]:
```
df_yolov3 = pd.read_csv('./process_result/yolov3_result_count.csv', index_col=
df_yolov3
```

Out[43]:

	car	bus	person	chair	suitcase	monitor
m6_0	188	174	187	179	190	186
m4_0	173	182	183	178	192	179
4_0	173	212	174	181	183	177
6_0	170	169	182	182	178	166
0_6	167	177	178	202	172	184
0_4	165	167	183	188	182	179
0_m4	168	161	187	183	183	174
0_m6	166	180	181	181	178	177

In [44]:
```
avg_counts_ssd = df_ssd.sum().sum() / 48
```

```
avg_counts_yolov3 = df_yolov3.sum().sum() / 48
print(f"The Average number of detection results in SSD is: {int(avg_counts_ssd
```

```
The Average number of detection results in SSD is: 419.
The Average number of detection results in Yolov3 is: 179.
```

We can observe from the above tables that the average result number in an experiment produced by the SSD model is 419. The average result number in an experiment produced by the Yolov3 model is 179. In a fixed period, the SSD model is $\frac{179}{429} \times 100\% = 42.7\%$ faster than Yolov3 model.

3. Data Processing

When the robot is at the initial position, the direction is 45 degrees offset to the object. The object is not in the capture range of the camera. E.g. In figure 1, we can see there is no object displayed in the GUI. When the control node is launched, during the process of robot rotation, the object enters the range of the camera, as it is shown in figure 2. Thus, between the period that the initial direction and the direction the object can enter into the camera range, The rotation of the robot starts at the same time as the start of the save node to receive data, so the detection result is invalid from the start time to the time when the object enters the range that the camera can capture. Similarly, in the process of completing the 90-degree rotation, the detection result of the time when the object is about to leave the camera range and the time when the robot completes the 90-degree rotation is also invalid. In the project code, we define that the return value of the detection result of this class is None, and the score is 0. Therefore, we need to remove this part of invalid data. In addition, when detecting some objects, the pre-trained model may not be able to detect the object in the whole process, so all the return values of the detection results are invalid. Therefore, in order not to eliminate the results sent because the model itself did not detect the object , we only filter the detection results with valid data greater than 50% of the total data. Figure 3 shows which part of invalid data we elimited.

In [18]:
```
import os
import pandas as pd
from tqdm import tqdm
import numpy as np
```

In [29]:
```
import time

round(time.time(), 2)
```

Out[29]:
```
1651789852.1
```

In [19]:
```
# Get all the result files name
def get_file_names(path):
    # param: path - there are two categories results, one is detected by ssd,
    # so we have to indicate which kind of results we are going to process
    file_names = [os.path.join(root, name) for root, _, files in os.walk(path)
    return file_names

ssd_result_names = get_file_names('./experiment_results_ssd')
yolo_result_names = get_file_names('./experiment_results_yolov3')
```

```
print(f"There are {len(ssd_result_names)} SSD results")
print(f"There are {len(yolo_result_names)} Yolov3 results")
```

```
There are 48 SSD results
There are 48 Yolov3 results
```

In [23]:
```
ssd_result_names
```

Out[23]:
```
['./experiment_results_ssd/chair/results_performance_chair_m6_0.csv',
 './experiment_results_ssd/chair/results_performance_chair_0_m4.csv',
 './experiment_results_ssd/chair/results_performance_chair_0_m6.csv',
 './experiment_results_ssd/chair/results_performance_chair_4_0.csv',
 './experiment_results_ssd/chair/results_performance_chair_0_4.csv',
 './experiment_results_ssd/chair/results_performance_chair_6_0.csv',
 './experiment_results_ssd/chair/results_performance_chair_m4_0.csv',
 './experiment_results_ssd/chair/results_performance_chair_0_6.csv',
 './experiment_results_ssd/car/results_performance_car_0_m4.csv',
 './experiment_results_ssd/car/results_performance_car_0_4.csv',
 './experiment_results_ssd/car/results_performance_car_0_m6.csv',
 './experiment_results_ssd/car/results_performance_car_m4_0.csv',
 './experiment_results_ssd/car/results_performance_car_4_0.csv',
 './experiment_results_ssd/car/results_performance_car_6_0.csv',
 './experiment_results_ssd/car/results_performance_car_0_6.csv',
 './experiment_results_ssd/car/results_performance_car_m6_0.csv',
 './experiment_results_ssd/bus/results_performance_bus_0_m6.csv',
 './experiment_results_ssd/bus/results_performance_bus_0_m4.csv',
 './experiment_results_ssd/bus/results_performance_bus_0_6.csv',
 './experiment_results_ssd/bus/results_performance_bus_6_0.csv',
 './experiment_results_ssd/bus/results_performance_bus_4_0.csv',
 './experiment_results_ssd/bus/results_performance_bus_m4_0.csv',
 './experiment_results_ssd/bus/results_performance_bus_m6_0.csv',
 './experiment_results_ssd/bus/results_performance_bus_0_4.csv',
 './experiment_results_ssd/person/results_performance_person_m4_0.csv',
 './experiment_results_ssd/person/results_performance_person_0_m6.csv',
 './experiment_results_ssd/person/results_performance_person_6_0.csv',
 './experiment_results_ssd/person/results_performance_person_m6_0.csv',
 './experiment_results_ssd/person/results_performance_person_4_0.csv',
 './experiment_results_ssd/person/results_performance_person_0_4.csv',
 './experiment_results_ssd/person/results_performance_person_0_m4.csv',
 './experiment_results_ssd/person/results_performance_person_0_6.csv',
 './experiment_results_ssd/suitcase/results_performance_suitcase_m6_0.csv',
 './experiment_results_ssd/suitcase/results_performance_suitcase_0_4.csv',
 './experiment_results_ssd/suitcase/results_performance_suitcase_0_m4.csv',
 './experiment_results_ssd/suitcase/results_performance_suitcase_6_0.csv',
 './experiment_results_ssd/suitcase/results_performance_suitcase_m4_0.csv',
 './experiment_results_ssd/suitcase/results_performance_suitcase_4_0.csv',
 './experiment_results_ssd/suitcase/results_performance_suitcase_0_6.csv',
 './experiment_results_ssd/suitcase/results_performance_suitcase_0_m6.csv',
 './experiment_results_ssd/monitor/results_performance_monitor_6_0.csv',
 './experiment_results_ssd/monitor/results_performance_monitor_4_0.csv',
 './experiment_results_ssd/monitor/results_performance_monitor_0_4.csv',
 './experiment_results_ssd/monitor/results_performance_monitor_0_m6.csv',
 './experiment_results_ssd/monitor/results_performance_monitor_0_6.csv',
 './experiment_results_ssd/monitor/results_performance_monitor_m4_0.csv',
 './experiment_results_ssd/monitor/results_performance_monitor_0_m4.csv',
 './experiment_results_ssd/monitor/results_performance_monitor_m6_0.csv']
```

In [31]:
```
# Define a function to get the object name, position according to the result f
def get_obj_pos(result_path):
```

```
    split_file_name = result_path.split('/')[-1].split('_')
    obj = split_file_name[2]
    position = split_file_name[3] + "_" + split_file_name[-1].replace('.csv',
    return obj, position

# Define a function to eliminate the none value in the file, and save them as
def eliminate_none(results_list, model_type):
    # eliminate every result file
    for result in tqdm(results_list):
        object_name, position = get_obj_pos(result)
        df = pd.read_csv(result, index_col="Unnamed: 0")    # open the result
        # df = df.drop(columns=['Unnamed: 0'])
        for index, row in df.iterrows():
            if row['category'] == 'none':
                df = df.drop([index])
        df.to_csv(os.path.join('./cleaned_data', model_type, object_name + "_"
```

In [32]:
```
# eliminate the none for ssd results
eliminate_none(ssd_result_names, "ssd")
# eliminate the none of yolov3 results
eliminate_none(yolo_result_names, 'yolov3')
```

```
100%|██████████| 48/48 [00:03<00:00, 12.36it/s]
100%|██████████| 48/48 [00:01<00:00, 29.75it/s]
```

4. Sensitivity

The detection rate is used to measure the proportion of objects in the environment detected by the pre-trained model in all experiments. During the detection process, the model may not be able to identify the object according to the image captured by the camera. At this time, in the result published by the detection node, the category is none and the confidence is 0. We regard this result as failed. In a single experiment, if the proportion of failed results reache more than 50%, we determine that the robot cannot detect the object at this position. By analyzing the data, we can get the following table:

In [33]:
```
import pandas as pd
```

In [34]:
```
# Get the cleaned files_paths of ssd
clean_ssd_results = get_file_names('./cleaned_data/ssd')

# Get the cleaned files_paths of yolov3
clean_yolov3_results = get_file_names('./cleaned_data/yolov3')
```

In [35]:
```
# Define a function that can return a new dataframe with the columns are the o
def create_new_df():
    objects_name = ['car', 'bus', 'person', 'chair', 'suitcase', 'monitor']
    positions = ['m6_0', 'm4_0', '4_0', '6_0', '0_6', '0_4', '0_m4', '0_m6']
    new_df = pd.DataFrame(columns=objects_name, index=positions)

    return new_df
```

In [36]:
```
# Define a function to evaluate whether the detection is success or failed
def detect_evl(result_paths, model_type):
    # Define the objects name and positions as the rows and index of table
```

```
    objects_name = ['car', 'bus', 'person', 'chair', 'suitcase', 'monitor']
    positions = ['m6_0', 'm4_0', '4_0', '6_0', '0_6', '0_4', '0_m4', '0_m6']
    empty_df = pd.DataFrame(index=positions, columns=objects_name)
    for result in result_paths:
        df = pd.read_csv(result)
        split_result = result.split('/')[-1].split('_')
        obj = split_result[0]
        pos = split_result[1] + '_' + split_result[-1].replace('.csv', '')

        # If the number of rows is less than 10, we count it as a failed detec
        # If the number of rows is greater than 10, we count it as a success d
        if df.shape[0] < 10:
            empty_df.at[pos, obj] = False
        else:
            empty_df.at[pos, obj] = True
    empty_df.to_csv(os.path.join('./process_result', model_type + '_detection_
```

In [37]:
```
# detect for ssd
detect_evl(clean_ssd_results, "ssd")

# detect for yolov3
detect_evl(clean_yolov3_results, "yolov3")
```

In [38]:
```
# Open the detection status of ssd
df_ssd = pd.read_csv('./process_result/ssd_detection_rate.csv', index_col='Unn
df_ssd
```

Out[38]:

	car	bus	person	chair	suitcase	monitor
m6_0	True	True	True	False	True	False
m4_0	True	True	True	True	True	False
4_0	True	True	True	True	True	False
6_0	True	True	True	False	True	False
0_6	True	True	True	True	True	True
0_4	True	True	True	True	True	True
0_m4	True	True	True	True	True	True
0_m6	True	True	True	True	True	True

In [39]:
```
# Open the detection status of yolov3
df_yolov3 = pd.read_csv('./process_result/yolov3_detection_rate.csv', index_co
df_yolov3
```

Out[39]:

	car	bus	person	chair	suitcase	monitor
m6_0	True	True	True	False	True	True
m4_0	True	True	True	True	False	False
4_0	True	True	True	True	False	False
6_0	True	True	True	True	False	True
0_6	True	False	True	True	False	False
0_4	True	True	True	True	False	False
0_m4	True	True	True	True	True	False
0_m6	True	True	True	True	True	False

We can observe from above tables that the success detection of SSD model is 42 out of 48 detections, the success detection of Yolov3 model is 13 out of 48 detections. The success rate of SSD and Yolov3 in the detection is:

$$DetectRate_{ssd} = \frac{42}{48} \times 100\% = 87.5\% \setminus DetectRate_{yolov3} = \frac{35}{48} \times 100\% = 73.0\%$$

5. Accuracy

In section 3, we evaluated when the camera capture the image in the environment, whether the model can percept there is an object. In this section, we are going to evaluate the accuracy of detection, which means how many correct detections the model made in the experiment. We will evaluate from two aspects, in the below tables, we can see the accuracy in each detection.

In [45]:
```
def accuracy_evl(cleaned_results, model_type):
    # Create a new dataframe to store the processed result
    new_df = create_new_df()
    # iterate the detection results
    for result in cleaned_results:
        # Get the object name and robot position
        obj = result.split('/')[-1].split('_')[0]
        position = result.split('/')[-1].split('_')[1] + '_' + result.split('/

        # Open the detection result
        df = pd.read_csv(result, index_col='Unnamed: 0')
        # num of results
        num_det = df.shape[0]
        # calculate how many correct detection
        detections = df['category'].value_counts()
        if obj not in detections.index:
            new_df.at[position, obj] = 0.0
            continue
        correct_rate = detections[obj] / num_det
        new_df.at[position, obj] = correct_rate
    new_df.loc['avg_acc'] = new_df.sum() / 8
    new_df.to_csv(os.path.join('./process_result', model_type + "_detection_ac
```

In [46]:
```
clean_ssd_results = get_file_names('./cleaned_data/ssd')
clean_yolov3_results = get_file_names('./cleaned_data/yolov3')
```

```
accuracy_evl(clean_ssd_results, 'ssd')
accuracy_evl(clean_yolov3_results, 'yolov3')

ssd_acc_df = pd.read_csv('./process_result/ssd_detection_accuracy.csv', index_
yolov3_acc_df = pd.read_csv('./process_result/yolov3_detection_accuracy.csv',
```

In [47]:
```
ssd_acc_df
```

Out[47]:

	car	bus	person	chair	suitcase	monitor
m6_0	0.537162	0.418605	1.000000	0.000000	0.000000	0.0
m4_0	0.005988	0.568116	0.996241	1.000000	0.000000	0.0
4_0	0.401384	0.377581	1.000000	0.769231	0.029412	0.0
6_0	0.032864	0.548387	0.992908	0.000000	0.000000	0.0
0_6	0.009615	0.000000	1.000000	0.000000	0.000000	0.0
0_4	0.049180	0.000000	0.989091	0.000000	0.000000	0.0
0_m4	0.000000	0.000000	0.971429	0.000000	0.500000	0.0
0_m6	0.004545	0.000000	1.000000	0.000000	0.000000	0.0
avg_acc	0.130092	0.239086	0.993708	0.221154	0.066176	0.0

In [48]:
```
yolov3_acc_df
```

Out[48]:

	car	bus	person	chair	suitcase	monitor
m6_0	0.987097	0.868687	1.0	0.800000	1.0	0.0
m4_0	1.000000	0.892655	1.0	0.983871	0.0	0.0
4_0	1.000000	0.649215	1.0	0.891667	0.0	0.0
6_0	1.000000	0.472973	1.0	0.990991	0.0	0.0
0_6	1.000000	0.000000	1.0	0.976744	1.0	0.0
0_4	0.680000	0.000000	1.0	0.824742	0.0	0.0
0_m4	1.000000	0.000000	1.0	0.921348	1.0	0.0
0_m6	1.000000	0.000000	1.0	1.000000	1.0	0.0
avg_acc	0.958387	0.360441	1.0	0.923670	0.5	0.0

From the tables we can tell the both model could not detect monitor very well. Both models have good accuracy for the detection on person. The yolov3 model has better performance than SSD as its average accuracy on other objects are higher than SSD. For further exploration, we will check the average confidence in each experiment.

In [74]:
```
def confidence_evl(cleaned_results, model_type):
    # Create a new dataframe to store the processed result
    new_df = create_new_df()
    # iterate the detection results
    for result in cleaned_results:
        # Get the object name and robot position
        obj = result.split('/')[-1].split('_')[0]
```

```
        position = result.split('/')[-1].split('_')[1] + '_' + result.split('/

        # Open the detection result
        df = pd.read_csv(result, index_col='Unnamed: 0')
        # num of results
        num_det = df.shape[0]
        # calculate how many correct detection
        detections = df['category'].value_counts()
        if obj not in detections.index:
            new_df.at[position, obj] = 0.0
            continue
        # correct_rate = detections[obj] / num_det
        # new_df.at[position, obj] = correct_rate
        confidences = []
        for index, row in df.iterrows():
            if row['category'] == obj:
                confidences.append(row['score'])
        avg_confidence = np.array(confidences).sum() / detections[obj]
        new_df.at[position, obj] = avg_confidence
        # new_df.loc['avg_confi'] = new_df.sum() / 8
    # new_df.loc['avg_acc'] = new_df.sum() / 8
    new_df.to_csv(os.path.join('./process_result', model_type + "_confidence.c
```

In [75]:
```
confidence_evl(clean_ssd_results, 'ssd')
confidence_evl(clean_yolov3_results, 'yolov3')
```

In [76]:
```
ssd_confi = pd.read_csv('./process_result/ssd_confidence.csv', index_col='Unna
yolov3_confi = pd.read_csv('./process_result/yolov3_confidence.csv', index_col
```

In [77]:
```
ssd_confi
```

Out[77]:

	car	bus	person	chair	suitcase	monitor
m6_0	0.659119	0.549630	0.640077	0.000000	0.000	0.0
m4_0	0.575000	0.570663	0.714340	0.544507	0.000	0.0
4_0	0.647414	0.572734	0.693172	0.535000	0.500	0.0
6_0	0.552857	0.590588	0.559643	0.000000	0.000	0.0
0_6	0.540000	0.000000	0.552540	0.000000	0.000	0.0
0_4	0.568889	0.000000	0.625221	0.000000	0.000	0.0
0_m4	0.000000	0.000000	0.618162	0.000000	0.537	0.0
0_m6	0.560000	0.000000	0.554519	0.000000	0.000	0.0

In [78]:
```
yolov3_confi
```

Out[78]:

	car	bus	person	chair	suitcase	monitor
m6_0	0.902288	0.784651	0.951311	0.582500	0.649600	0.0
m4_0	0.954194	0.933354	0.948934	0.711967	0.000000	0.0
4_0	0.979711	0.877984	0.979224	0.944486	0.000000	0.0
6_0	0.986115	0.756000	0.991917	0.889182	0.000000	0.0
0_6	0.929077	0.000000	0.994538	0.919524	0.637500	0.0
0_4	0.825441	0.000000	0.970887	0.806000	0.000000	0.0
0_m4	0.928106	0.000000	0.988629	0.703049	0.595909	0.0
0_m6	0.947724	0.000000	0.995966	0.867619	0.652353	0.0