Developing neural network navigation for service robots in dynamic environments

A thesis submitted to the Faculty of Science, Agriculture and Engineering for the Degree of Doctor of Philosophy in PhD in MECH(FT) 8090F programme

by

QUAN WU

School of Engineering,

Newcastle University,

Newcastle Upon Tyne, UK

February 2025

To my mom and dad for their unwavering support, understanding and encouragement over the years.

Abstract

Learning how to navigate autonomously in an unknown indoor environment whilst avoiding both static and dynamic obstacles is crucial for mobile robots. Traditional navigation systems in mobile robots lack the capability for autonomous learning. This study focuses on the performance of a navigation system in a wheeled mobile robot, utilizing a convolutional neural network (CNN). The evaluation was conducted through simulation using Webots software, where the environment was designed to include walls, floors, and objects to create boundaries and obstacles. LiDAR, compass, and bumper sensors were used to receive data from the simulated environment. These data sets were saved and then converted into image files as the input data for the CNN. The 15-layer deep CNN was designed in MATLAB Deep Network Designer APP. Two networks were trained: supervised and unsupervised. In the supervised version, user control of the robot was used to manually avoid obstacles whilst data was collected. This proved to be a time-consuming approach. To efficiently obtain a more comprehensive data set, an unsupervised approach was adopted in which the robot randomly started in different locations with random trajectories. With both avoidance and collision of obstacles occurring, two sets of data were collected labelled ‘good’ and ‘bad’. Both sets were then used to train the network. Validation during training proved to be an unreliable indicator of network performance due to the possible multiple outputs from a given data set. Therefore, all trained networks were tested in Webots using distance travelled and number of collisions as the performance metric. The supervised network demonstrated a success rate of 99.24% whilst the unsupervised network demonstrated a 95.43% success rate. The work demonstrates that supervised learning in a simulated environment is a suitable way of training CNNs.

The study then advanced to a dynamic environment by introducing several moving objects into the previous environment. These objects, equipped with several sensors, could avoid both the robot and static boxes. Initially, the same method from the static environment was applied, using 5 consecutive scans instead of one to detect moving objects. However, the trained network failed to make the robot avoid obstacles effectively. The algorithm was then improved in two subsequent versions: algorithm A increased the delay time between data

captures to enhance the difference in data, whereas algorithm B divided the original motion process into five steps with consistent movement decisions, enriching the distinctiveness and information of the input data. The final collision rate decreased from the initial 17.21% to 6.78% with algorithm B.

The final step involves constructing the physical robot and then testing the trained network. A two-wheeled differential robot equipped with a LiDAR sensor was built to test the basic SLAM process in a static environment, and an IMU model was included to determine the heading north, testing the trained network. Super beacons are used to record the trajectory. The trained network enables the real robot to navigate effectively, allowing it to consistently avoid obstacles and navigate to the north side of the room. Future objectives include implementing the trained network in a dynamic environment to further enhance the robot's navigation capabilities.

Acknowledgements

The completion of this Ph.D. dissertation represents the culmination of a challenging and rewarding academic journey, and I am profoundly grateful to the many individuals and organizations who have supported and guided me along the way.

First and foremost, I would like to extend my heartfelt thanks to my supervisors Dr John Hedley and Dr Francis Franklin. Their unwavering commitment to academic excellence, their sage advice, and their tireless dedication to my research have been instrumental in shaping this work. Their mentorship has not only broadened my knowledge but has also inspired me to strive for the highest standards in research.

I want to acknowledge the Newcastle University for providing a conducive academic environment and resources that were essential for the successful completion of this research. The opportunities for collaboration, access to libraries, and research facilities have been invaluable.

My fellow graduate students and colleagues have been a constant source of inspiration, intellectual exchange, and camaraderie. I cherish the friendships formed and the collaborative efforts that enriched my academic experience.

My family has been my unwavering source of love and strength throughout this endeavour. Their unwavering support, encouragement, and understanding, even during the most demanding times, have sustained me.

Lastly, I would like to especially thank my girlfriend Shiting Wang for her love, encouragement, and endless support.

List of publications

- Quan Wu, “Utilizing neural network for mobile robot navigation,” Oral Presentation, **ICRA@40: 40th Anniversary of the IEEE Int. Conf. on Robotics and Automation (ICRA)**, Rotterdam, Netherlands, Sept. 23-26, 2024.
- Quan Wu & John Hedley. ‘Deep Neural Network Training Approaches for Robot Navigation’ (in review)

List of Figures

List of Tables

List of Abbreviations

AI	Artificial intelligence
CNN	Convolutional Neural Network
CNND	Depth Conv Net
CNNVO	VO Conv Net
DQN	Deep Q-Network
DSG	Dynamic Scene Graph
EKF	Extended Kalman Filter
GAN	Generative Adversarial Network
GUI	Graphical User Interface
HSSH	The Hybrid Spatial Semantic Hierarchy
ICR	Instantaneous Centre Rotation
LC	Loop closure
LP	Loop Closing
LPD-Net	Large-scale place description network
LSTM	Long Short-Term Memory
MAC	Micro Aerial Vehicle
MCL	Monte Carlo Method
MLP	Multi-layer perceptron
MSE	Mean squared error

NDT	Normal Distributions Transform
PPO	Proximal policy optimization
PReLU	Parametric ReLU
ReLU	Rectified Linear Unit
ResNet	Residual Network
RNN	Recurrent Neural Network
SGANVO	Stacked Generative Adversarial Network
SLAM	Simultaneous Localisation And Mapping
SPP	Spatial pyramid pooling
STNN	Spatio-Temporal Neural Network
UAV	Unmanned aerial vehicle
VO	Visual odometry
YOLO	You Only Look Once

Contents

Chapter 1. Introduction

1. 1 Introduction

In the field of robotics, achieving autonomous navigation in dynamic environments remains an enduring challenge. Robots deployed in real-world scenarios often encounter highly dynamic and unpredictable surroundings, such as crowded urban streets, busy factories, or disaster-stricken areas. Effective navigation in these complex settings necessitates the integration of advanced technologies, and in recent years, artificial intelligence, particularly neural networks, has emerged as a transformative tool for enhancing the navigational capabilities of robots[1-8].

Neural network-based navigation represents a paradigm shift in the realm of robotics. By harnessing the power of deep learning, robots can acquire the ability to perceive their surroundings, make informed decisions, and adapt their movements in real-time, mirroring human-like cognitive processes. This approach has demonstrated remarkable success across various applications, including autonomous vehicles, aerial drones, and ground-based mobile robots[9-14].

Navigation in dynamic environments entails multifaceted challenges. It demands real-time sensor data processing, accurate modelling of dynamic obstacles and interactions, trajectory planning, and decision-making under uncertainty. Conventional navigation methods, often reliant on handcrafted algorithms, struggle to adapt to the complexity and variability of dynamic scenarios. In contrast, neural networks offer a data-driven approach to navigation. These artificial neural systems excel at recognizing patterns, making sequential decisions, and generalizing from extensive datasets. The concept of end-to-end learning, where a neural network learns to map sensory inputs directly to control actions, has gained prominence, enabling robots to acquire navigation skills through training, eliminating the need for explicit rule-based programming[15-18].

The development of neural network-based navigation systems for robots in dynamic environments holds immense significance across multiple domains. These systems can

substantially enhance the safety and efficiency of autonomous vehicles, allowing them to navigate congested traffic, unpredictable road conditions, and complex intersections with unparalleled precision. In logistics and warehouse automation, robots equipped with neural network navigation capabilities optimize their paths, resulting in more efficient item picking and transportation in crowded and dynamic environments. Moreover, in missions such as search and rescue or disaster response, adaptive robots equipped with neural network-based navigation can dynamically adapt to challenging terrains and obstacles to reach victims or perform reconnaissance tasks[14, 17, 19-21].

The increasing integration of robots into daily life, encompassing delivery drones, service robots in healthcare, and hospitality, demands robust navigation abilities. Neural network-based navigation systems ensure smooth and safe interactions in environments where humans and other moving entities coexist, ultimately enhancing user experiences[22, 23].

Despite their promise, neural network-based navigation systems face several challenges, including issues related to safety, robustness, interpretability, and generalization. Ensuring that neural networks can handle rare and extreme situations while providing interpretable decision-making remains a research priority. Bridging the gap between simulation and real-world deployment poses challenges due to domain shift issues.

Current trends in this field encompass hybrid approaches that combine traditional algorithms with neural networks, reinforcement learning for adaptive behaviour, and the exploration of neurobiologically-inspired navigation models. Moreover, addressing ethical and regulatory aspects of AI-powered robots in dynamic environments is of paramount importance[24-27].

1.1.1 Research Statement

The integration of neural network-based navigation systems in service robots presents a promising solution to the enduring challenge of autonomous navigation in dynamic environments. Despite significant advancements, existing systems often lack the adaptability, robustness, and continuous learning capabilities required to navigate complex and unpredictable real-world settings effectively. This research aims to bridge these gaps by

developing a comprehensive neural network-based navigation system that dynamically adjusts to environmental changes, ensuring smooth, safe, and efficient operation. By leveraging deep learning, this study seeks to enhance robots' ability to perceive their surroundings, make informed decisions, and adapt their movements in real-time, thereby improving their operational efficiency and safety.

1.1.2 Research Gap

Despite the advancements in robotic navigation, current systems are predominantly designed for static or minimally dynamic environments, relying heavily on predefined paths and static maps. These systems struggle to adapt to the complexity and variability of dynamic scenarios, such as crowded urban streets, busy factories, or disaster-stricken areas. Existing neural network approaches show potential but often lack real-time adaptability, robustness, and the capability for continuous learning. There is a significant gap in developing a navigation system that integrates real-time data processing, robust obstacle detection, adaptive path planning, and continuous learning to enhance the reliability and efficiency of robots in highly dynamic environments.

1.1.3 Research Problems to Solve

One of the primary challenges faced by existing navigation systems is their inability to adapt quickly to sudden changes in the environment, such as moving obstacles or varying terrain. To address this, the research aims to develop a neural network architecture capable of processing real-time sensor data and dynamically adjusting navigation strategies. This involves creating models that can predict and respond to changes instantly, ensuring smooth and continuous movement for service robots.

Efficient path planning in dynamic environments is another significant challenge due to the unpredictability of moving obstacles and changing paths. The proposed solution is to design algorithms that integrate deep reinforcement learning with traditional path planning methods. This integration will enable robots to find and follow the most efficient route while optimizing for factors such as travel time, energy consumption, and safety.

Reliable detection and avoidance of obstacles in real-time is critical for safe navigation but is often hindered by noisy sensor data and complex environments. Enhancing obstacle detection capabilities using advanced sensor fusion techniques that combine data from LIDAR, cameras, and ultrasonic sensors will be essential. Developing deep learning models that can accurately interpret this data to detect obstacles and predict their movements will ensure that robots navigate safely even in cluttered and unpredictable settings.

Many existing navigation systems lack the ability to learn from their experiences, leading to repeated mistakes and inefficiencies. To overcome this, the research will implement mechanisms for continuous learning, where the neural network updates its knowledge base and improves its decision-making process over time. This could involve the use of online learning algorithms and reinforcement learning techniques, allowing the navigation system to adapt to new and unforeseen scenarios.

Robustness against a variety of environmental conditions, such as different lighting, weather, and levels of clutter, is another critical issue for navigation systems. Developing robust neural network models that can generalize well across diverse conditions is essential. This includes training on large, varied datasets and implementing strategies like domain adaptation and augmentation to improve resilience and reliability.

Ensuring safe and efficient interaction with humans in shared spaces is a critical challenge for service robots. This research will address this by developing algorithms that enable the robot to predict human movements and intentions, facilitating smooth navigation around people. Integrating social navigation models and human behavior prediction into the navigation system will ensure that the robot can navigate without causing discomfort or hazards to humans.

By addressing these research problems show in table 1-1, this thesis aims to significantly advance the field of service robotics, enabling the deployment of robots that can autonomously and safely navigate dynamic environments. The ultimate goal is to enhance their usability and effectiveness in real-world applications, such as healthcare, hospitality, and domestic services. This will improve the quality of service provided by these robots and ensure a higher level of safety and reliability in their operations.

Table 1-1. Research problems and the proposed solution.

Research Problem	Description	Proposed Solution
Real-Time Adaptability	Systems struggle to adapt to sudden environmental changes.	Develop neural networks for real-time sensor data processing and dynamic navigation adjustments.
Path Planning and Optimization	Efficient path planning is difficult due to unpredictable obstacles and paths.	Integrate deep reinforcement learning with traditional methods for optimal route planning, considering various factors.
Obstacle Detection and Avoidance	Reliable real-time obstacle detection is hindered by noisy data and complex environments.	Enhance detection using sensor fusion and deep learning to interpret data and predict obstacle movements.
Learning from Experience	Systems often repeat mistakes and inefficiencies.	Implement continuous learning mechanisms to update and improve decision-making over time.
Robustness and Reliability	Systems need to operate under diverse environmental conditions.	Develop robust models trained on varied datasets, employing domain adaptation and augmentation.
Human-Robot Interaction	Ensuring safe interaction with humans in shared spaces is challenging.	Develop algorithms to predict human movements and intentions, integrating social navigation models.

1.2 Aim and Objectives

The aim of this project is to develop a robust convolutional neural network for autonomous robot navigation, capable of effectively solving the simultaneous localization and mapping

(SLAM) problem in dynamic indoor environment using both supervised and unsupervised learning techniques. The specific objects are to:

- Design and Construct a Physical Robot: Engineer a robot equipped with an array of sensors necessary for indoor navigation and SLAM. This robot will serve as the platform for deploying and testing the neural network in real-world scenarios.
- Develop a Simulation Environment: Create a detailed simulation environment that mirrors the real-world setting in which the robot will operate. This simulated platform should allow for both supervised and unsupervised control scenarios, and it should facilitate the collection of navigation data under various conditions.
- Convolutional Neural Network Development: Design a concise and efficient convolutional neural network using MATLAB. The design should focus on optimizing the architecture to process sensory data efficiently, ensuring low latency and high accuracy in real-time decision-making processes.
- Data Collection and Model Training: Implement data collection protocols for both the physical robot and the simulation, capturing a wide range of sensor inputs under different environmental conditions. Use this data to train the neural network, employing both supervised learning (with labelled data) and unsupervised learning (without labelled data) approaches.
- Dynamic Environment Testing: Enhance the simulation environment by introducing moving objects and varying environmental conditions to assess the neural network's performance in dynamic settings. This testing should evaluate the model's adaptability, accuracy, and reliability in real-time navigation and obstacle avoidance.
- Deployment and Evaluation on Physical Robot: Deploy the trained convolutional neural network on the real physical robot to navigate autonomously in dynamic indoor environments. Evaluate the network's performance based on predefined metrics such as navigation accuracy, computational efficiency, and adaptability to new environment conditions. This evaluation will determine the network's practical effectiveness and sufficiency in real-world applications.

1.3 Thesis Structure

This thesis is categorised into seven chapters as shown in Figure 1-1. The brief synopsis of reminder chapters is listed in the following:

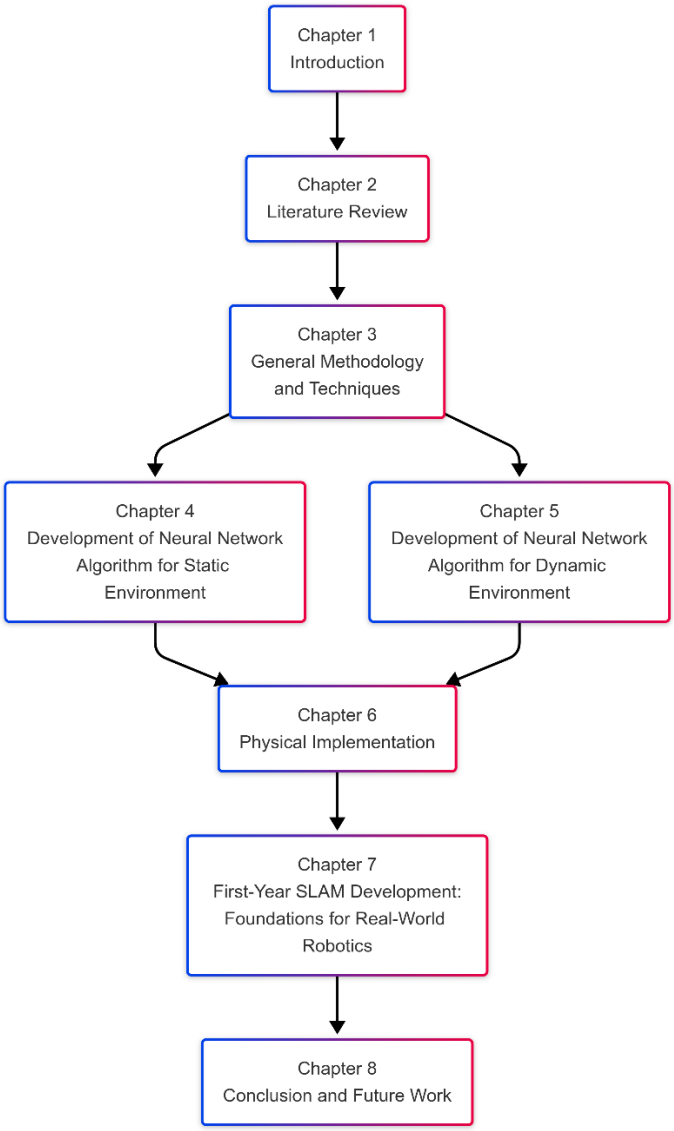


Figure 1-1. Structure of this thesis.

Chapter 1 gives a brief introduction regarding to the topic of this thesis, including the general background of the work, the aims, and objectives of the project as well as structure of the thesis.

Chapter 2 presents a review on the literature relevant to this project which consists of seven main sections: (1) Foundations and Development of Traditional Navigation Techniques; (2) Foundations of Neural Network Technology; (3) Perception and Processing of Dynamic Environments; (4) Application of Neural Networks in Navigation and Path Planning; (5) Comparison of Traditional Navigation Methods and Neural Network Navigation; (6) Integration of Traditional and Neural Network Navigation Techniques; (7) Challenges and Future Research Directions.

Chapter 3 provides a detailed simulation methodology of the Webots construction in this thesis. The simulation model is based on an actual small mobile robot which is built at the first year for the SLAM research. MATLAB is integrated with Webots for neural network training and data collection and processing.

Chapter 4 propose a convolutional neural network model for the robot training and navigation based on supervised learning and unsupervised learning. There are several algorithms towards the data collection and processing steps. Finally, the trained network navigation ability is tested with simulation as well as the data collection. The collision rate and the percentage of north travel are used to compare with different training input data.

Chapter 5 moves to dynamic case based on the Chapter 4. The improved neural network is implemented to detect the moving object and avoid obstacles. This part is still in progress and will be completed before the viva.

Chapter 6 introduces how the neural network trained in the simulation environment is applied to real mobile robot navigation. This chapter describes the physical robot platform, explains how the trained model is integrated with the robot's control system, and discusses the practical considerations of transitioning from simulation to reality.

Chapter 7 focuses on the first-year SLAM development and experimentation using the physical robot. A correlation matrix is employed to estimate the robot's position by comparing two consecutive map scans. The proposed SLAM approach is compared against

wheel odometry and MATLAB's built-in scan matching function. Lastly, this chapter explores early attempts to track objects in dynamic environments.

Chapter 8 concludes the research work in this thesis and proposes the recommendations for the future work.

Chapter 2. Literature review

2.1 Introduction

In this chapter, it provides a comprehensive overview of fundamentals and recent work on neural network and traditional navigation technologies for autonomous robot navigation. It provides an in-depth analysis of the foundational principles, development, and application of these technologies. This literature review is structured to first introduce traditional navigation methods, followed by neural network navigation methods, their perception and processing of dynamic environments, application in navigation and path planning, and finally a comparative analysis of the two approaches.

This chapter is classified as follows: Sec 2.2 reviews the historical development of traditional navigation techniques and summarizes their evolution over time. Sec 2.3 provides an overview of the foundational concepts of neural networks, tracing the milestones in their development within the field of artificial intelligence. Sec 2.4 discusses the techniques used in the perception and processing of dynamic environments, emphasizing the role of advanced sensors and algorithms in adapting to changing conditions. Sec 2.5 reviews the application of neural network technologies in navigation and path planning, detailing specific case studies and the outcomes of recent research. Sec 2.6 compares the efficacy and efficiency of traditional navigation methods with those based on neural networks, highlighting the advantages and disadvantages of each approach. Sec 2.7 explores the integration of traditional navigation techniques with neural network-based approaches, discussing synergies and enhanced capabilities that result from such integrations. Sec 2.8 outlines the main challenges currently facing the field of navigation technology and suggests directions for future research, with a focus on innovation and potential breakthroughs.

2. 2 Foundations and Development of Traditional Navigation Techniques

Traditional navigation methods have been the cornerstone of autonomous robotic systems, providing the essential frameworks and algorithms necessary for robots to move through and interact with their environments. These methods are primarily based on a combination of odometry, inertial measurement units (IMUs), and landmark-based navigation, each of which utilizes different sensors and computational techniques to determine the robot's position and map its surroundings.

2.2.1 Basic principles of Traditional Navigation

Odometry is one of the oldest and most fundamental methods in robotic navigation. It involves estimating a robot’s position and orientation by measuring the rotations of its wheels using encoders. This method is straightforward and computationally inexpensive, making it widely used in various robotic applications. However, odometry alone is prone to cumulative errors over time due to wheel slippage and other factors. These errors can accumulate and result in significant inaccuracies in the robot's perceived position, especially over long distances, or uneven terrain. The error propagation can be mathematically described using the odometry error model, which quantifies the drift based on factors such as wheel diameter inaccuracies and slippage[28].

IMUs complement odometry by providing data on the robot's orientation and acceleration through accelerometers and gyroscopes. IMUs can track the robot's movement in three-dimensional space, offering more detailed information on its trajectory. Despite their advantages, IMUs also suffer from drift over time, which can lead to inaccuracies in long-term navigation. The integration of accelerometer and gyroscope data using techniques such as Kalman filtering can partially mitigate this drift, but the challenge remains significant for extended operations[29].

Landmark-based navigation involves the robot identifying and using recognizable features in its environment to aid localization. These landmarks can be natural features like trees or artificial markers placed specifically for navigation purposes. By continuously referencing these landmarks, robots can correct positional errors that accumulate from odometry and

IMU measurements, significantly enhancing accuracy. Landmark-based navigation often employs algorithms such as the Extended Kalman Filter (EKF) for sensor fusion and data association, which improves the robustness of the system in dynamic environments[30].

2.2.2 Development of SLAM Technology

SLAM represents a revolutionary advancement in traditional navigation methods. SLAM allows a robot to create a map of an unknown environment while simultaneously keeping track of its location within that map[31]. This dual capability is essential for autonomous navigation in unstructured and dynamic environments.

There have been many methods towards the traditional SLAM problem, most of which can be divided into two areas: filter-based and optimization-based approaches (see Figure 2-1)[32].

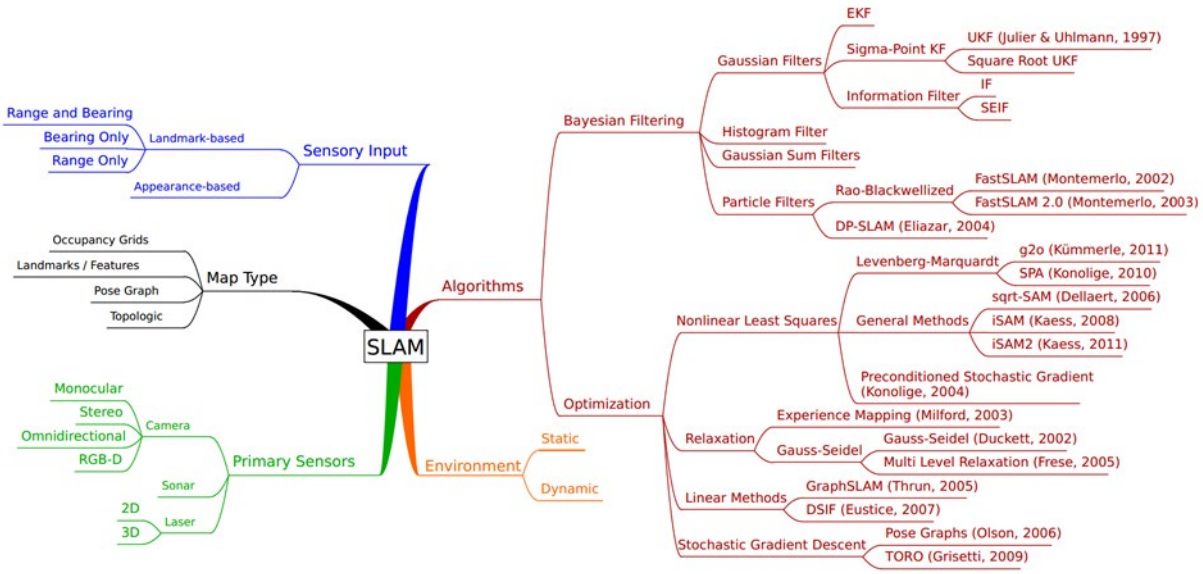


Figure 2-1. 1 A taxonomy of SLAM progress[32].

The exploration of SLAM problems initiated in the mid-1980s, receiving comprehensive analysis in two influential surveys by Durrant-Whyte and Bailey[33, 34]. During this aera, laser range finders emerged as the primary sensors, drawing significant attention from the research community. The 1990s saw the advent of EKF SLAM as a pioneering method for

addressing SLAM challenges. Nonetheless, EKF SLAM encountered limitations in large-scale environments due to computational demands and inherent nonlinearity issues. In response to these challenges, 2002 marked a significant advancement with the introduction of Fast SLAM[35]. This innovative approach, grounded in particle filter technology, revolutionized the field. Fast SLAM utilized a collection of weighted samples to accurately represent a robot's posterior state. This representation was continuously refined by recalculating the weight of each sample following data association, enhancing the model's reliability and accuracy. Further evolution in the field led to the creation of Gmapping, a synergistic blend of scan matching and grid mapping techniques. This approach rapidly gained popularity, becoming the preeminent algorithm for laser-based SLAM problems[36, 37]. The landscape of SLAM research underwent another transformation in 2010 with the proposal of graph-based SLAM. This method introduced least squares optimization techniques as a cornerstone approach, establishing them as the trending and predominant solutions in contemporary SLAM research[38].

- EKF SLAM

The EKF addresses the online SLAM challenge, which focuses on estimating the posterior probability of the robot's current position. EKF's approach involves maintaining a comprehensive state vector that encapsulates not only the robot's pose but also the positions of various environmental landmarks. To effectively manage the uncertainties associated with these state estimates, EKF employs an error covariance matrix. This matrix is crucial as it not only records the uncertainties of each state component but also captures the cross-correlations between the robot's pose and the landmarks' locations[28, 39]. This sophisticated approach allows EKF to provide a more nuanced and accurate understanding of the robot's environment and its interaction with it.

- Particle filter SLAM

Particle filters operate by maintaining a multitude of particles, each representing potential estimates of the robot's pose and the positions of various features in its environment. This method hinges on a dynamic process: as the robot moves and performs scanning operations, the particle set undergoes continual updates. This update process involves a crucial

'resampling' step[28, 39],which refines the particle estimates to better match the robot's observed state and environmental interactions. Building upon this foundation, an enhanced version of this methodology emerged, known as Fast SLAM. Fast SLAM represents a significant advancement in particle filter application, offering improved efficiency and accuracy in simultaneous localization and mapping tasks. This method's detailed framework and its advantages over traditional particle filtering techniques in SLAM are extensively discussed in the literature[35, 40].

- Graph-based SLAM

Graph-based SLAM, in contrast to EKF approaches, effectively addresses the full spectrum of the SLAM problem. This method employs a sparse graph structure for representing robotic measurement data. Within this graph, nodes symbolize the robot's various poses and sensory measurements, while edges signify soft constraints that link pairs of robot poses. Upon constructing this graph, the focus shifts to optimization techniques. These techniques are employed to meticulously ascertain the optimal arrangement of the robot's poses, ensuring a more accurate and comprehensive solution to the SLAM problem[28, 39].

2.2.2.1 Popular 2D laser SLAM algorithms

In recent years, a significant number of 2D laser SLAM algorithms have been made available as open-source projects, greatly facilitating accessibility and usage. Among these, Gmapping, Hector SLAM, and Google Cartographer stand out as the most renowned and widely used algorithms. Their open-source nature not only democratizes access but also provides an invaluable resource for those new to the field of SLAM. Beginners find these tools particularly user-friendly, as they enable a quicker and more intuitive understanding of the underlying principles and mechanics of SLAM technologies. The availability of these resources has thus played a pivotal role in fostering learning and experimentation in the domain of robotic mapping and navigation.

- Gmapping

The cornerstone of Gmapping is the Rao-Blackwellized particle filter, commonly known as Fast SLAM. This method employs particle filters, each representing a distinct environmental map, necessitating a substantial quantity of particles. However, the challenge of particle

depletion is an inherent issue in this approach. To address this, Gmapping innovatively reduces the particle count while preserving accuracy and quality. This is achieved through the computation of a precise proposal distribution that incorporates both robot movement and sensor observations. Furthermore, Gmapping employs a strategic threshold setting during the resampling step, effectively mitigating the problem of particle impoverishment, as outlined in references[36, 37]. This methodological refinement significantly enhances the efficiency and performance of the algorithm.

Conventional particle filter applications typically employ wheel odometry as the proposal distribution, primarily due to its computational simplicity across various robotic systems. However, in most scenarios, sensor data offers markedly higher precision compared to wheel odometry. This is illustrated in Figure 2-2, where the area representing significant likelihood is substantially smaller when utilizing accurate sensor data rather than wheel odometry. Consequently, this increased sensor accuracy substantially diminishes the number of particles needed for effective operation. The contrast between the two methods highlights the efficiency gains achieved through the integration of precise sensor technology in particle filter applications.

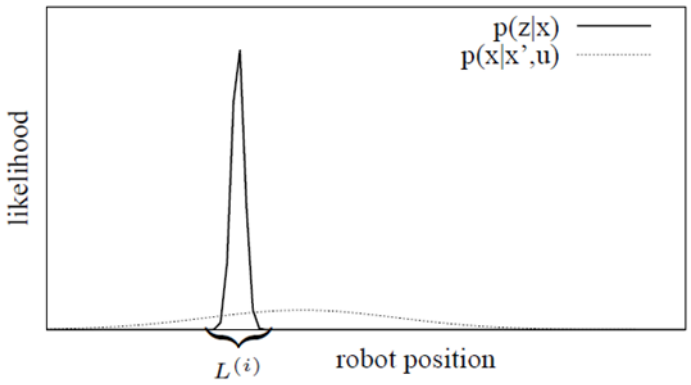


Figure 2-2. The two types of the motion model. Within the interval L^((i)) the product of using sensor information and wheel odometry is dominated by the observation likelihood. Accordingly the model of wheel odometry error can be reduced with an accuracy sensor [36].

Gmapping has undergone significant refinements, enhancing its suitability for small environment SLAM through reduced computational demands and heightened robustness. These improvements have been pivotal in minimizing processing and storage requirements, crucial for operational efficiency in constrained environments. However, its performance

tends to diminish in larger settings due to the escalating need for more particles, which proportionally increases both storage and computational burdens. Despite these limitations, Gmapping's effectiveness in 2D laser SLAM has not gone unnoticed. Its widespread popularity is evidenced by its extensive application in numerous contemporary products, underscoring its status as a leading algorithm in the field[41].

- Hector SLAM

Hector SLAM presents an innovative integration of a robust 2D SLAM system, leveraging scan matching, with a 3D navigation system anchored in inertial sensing. Central to its design is the reliance on high-accuracy LiDAR sensors, which are crucial due to the system's unique characteristic of not utilizing odometry information. This precision in distance measurement is vital. The accuracy and reduced reliance on odometry make Hector SLAM particularly suitable for applications in aerial robotics, especially in unmanned aerial vehicles (UAVs)[42]. A comprehensive illustration of this system's architecture and workflow is depicted in Figure 2-3.

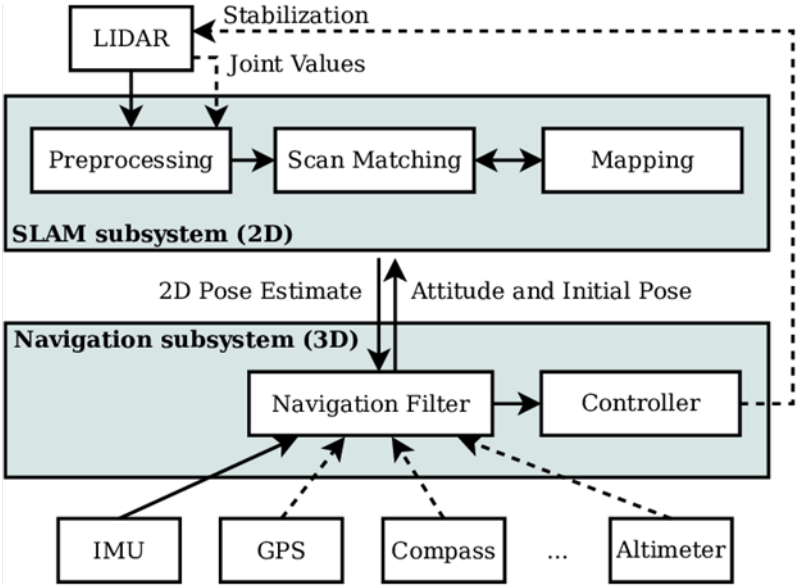


Figure 2-3. Overview of Hector SLAM system (dashed lines depict optional information)[42].

The Hector SLAM algorithm employs a bilinear interpolation method during the map access phase. This sophisticated approach determines if and how precisely a laser point maps onto the existing map, assessing the degree of correspondence. In addressing the challenges of

scan matching, the algorithm utilizes the Gaussian-Newton optimization method in its back-end processing. This method enhances the accuracy and efficiency of the scan matching process. Additionally, Hector SLAM incorporates a multi-resolution map representation strategy. This particular feature is pivotal in circumventing the pitfalls of local minima, a common challenge in map optimization[42].

In summary, Hector SLAM stands out for its capability to effectively navigate SLAM challenges in uneven terrains without relying on odometry data. This attribute marks a significant advancement in SLAM technology. However, it's important to note a limitation of the system. Specifically, Hector SLAM may encounter performance issues when operated with sensors that have a low scan rate, which can impede the system's overall effectiveness and accuracy[41].

- Google Cartographer

Google Cartographer adopts a scan matching methodology akin to that of Hector SLAM, but it further enhances accuracy with a real-time loop closure strategy designed to minimize scan matching errors. This innovative approach involves constructing submaps from a series of consecutive scans. These submaps then serve as reference points for aligning new scans, thereby establishing constraints for improved mapping accuracy.

A key feature of Google Cartographer's methodology is the utilization of a branch-and-bound algorithm. This algorithm efficiently facilitates real-time loop closure, a critical aspect in maintaining the integrity and continuity of the mapping process. As illustrated in Figure 2-4, these submaps are represented through a network of grid points and corresponding pixels[43]. This visual representation not only simplifies understanding but also aids in the effective application of the technology.

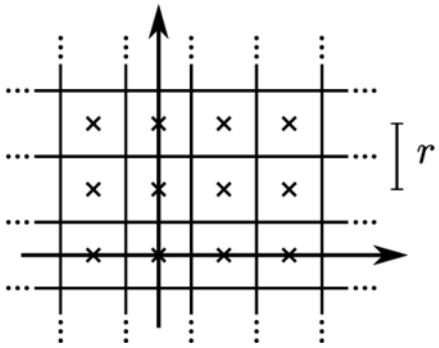


Figure 2-4. Grid points and associated pixels[43].

Through this blend of advanced scan matching and loop closure techniques, Google Cartographer stands out as a sophisticated tool in the realm of mapping and spatial navigation, offering a robust solution for accurate and dynamic environment mapping.

2.2.2.2 Semantic classification

The robot's position is categorized through a process known as 'semantic place labelling' or 'semantic place classification'[44]. Prior research has explored various classification strategies, primarily utilizing visual data captured by cameras[45, 46], and in some cases, fused data[47, 48] that integrates visual and range information from laser range finders. These approaches, however, faced challenges in maintaining consistent performance under varying illumination conditions, which are inherent to different lighting environments. Consequently, there has been a shift in research focus towards leveraging 2D laser data for more reliable classification of mobile robot locations, circumventing the limitations posed by reliance on visual data in diverse lighting scenarios.

Prior research has explored a range of strategies for semantic classification of robot locations using 2D laser data, leveraging techniques like supervised machine learning[49], graph theory[50], and clustering algorithms[51]. These methodologies have shown proficiency in distinguishing between rooms and corridors, yet they falter when it comes to accurately identifying doorways. To overcome this limitation, the implementation of deep learning approaches appears promising. These methods have the potential to efficiently extract unique features characteristic of doorways directly from unprocessed data. Considering this, a novel point-based deep learning framework, dubbed '2DlaserNet', has been developed. This architecture is specifically designed to enable the network to adeptly

learn and identify distinctive features from the raw point clouds generated by laser scans[6]. Unlike previous models, 2DlaserNet's advanced learning capabilities offer a more robust and accurate approach to the semantic classification of various locations, particularly in the challenging task of doorway detection.

2.2.3 Key Implementation Details

Implementing SLAM effectively involves integrating data from various sensors to create a comprehensive map of the environment. Sensors such as lidar, cameras, and sonar are commonly used to gather detailed information about the robot's surroundings. Lidar sensors, for instance, provide precise distance measurements to objects, which are crucial for creating accurate maps. Cameras offer visual data that can be processed to identify features and landmarks, while sonar sensors can detect obstacles and measure distances in environments where visual data may be insufficient. The fusion of these sensor modalities requires sophisticated algorithms to ensure the accuracy and reliability of the SLAM system[52].

A critical aspect of SLAM is data association, which involves matching observed features with previously mapped features to maintain accurate localization. This process requires sophisticated algorithms to handle the complexities of dynamic environments, where features can change or move over time. Effective data association ensures that the robot can update its position and map accurately, even in challenging conditions. Techniques such as the Joint Compatibility Branch and Bound (JCBB) and Random Sample Consensus (RANSAC) are commonly used for robust data association in SLAM[53].

Map optimization is another essential component of SLAM, where the map is continuously refined to reduce errors and improve accuracy. Techniques such as bundle adjustment and pose graph optimization are used to adjust the positions of the mapped features and the robot itself, ensuring that the overall map remains consistent and accurate. These optimization techniques are computationally intensive but are necessary for maintaining the reliability of the SLAM system over extended periods and across large environments. Advanced optimization techniques, such as incremental smoothing and mapping (iSAM),

provide real-time performance by incrementally updating the map and robot pose as new data is acquired[54].

Ensuring the robustness and scalability of SLAM systems is crucial for their deployment in real-world applications. Robustness refers to the system's ability to handle dynamic and unpredictable environments, while scalability pertains to the system's capability to operate efficiently in large-scale environments. Techniques such as multi-session SLAM, which allows for the merging of maps from multiple sessions, and distributed SLAM, where multiple robots collaboratively build a map, address these challenges. Additionally, the use of semantic information, where objects and landmarks are recognized and categorized, enhances the robot's understanding of its environment, improving both robustness and scalability[55].

In summary, traditional navigation methods, particularly SLAM, have laid a robust foundation for autonomous robotic navigation. These methods have evolved significantly over the past few decades, integrating advancements in sensor technology and probabilistic algorithms to enhance their accuracy and reliability. As we transition to exploring neural network-based navigation methods, it is essential to understand these traditional techniques' strengths and limitations to appreciate the innovations brought by neural networks in robotic navigation.

2. 3 Foundations of Neural Network Technology

Neural networks, inspired by the human brain's structure and functioning, have revolutionized various technological domains, including robotics. These networks consist of interconnected nodes or neurons, which process and transmit information, allowing machines to learn from data and make decisions. Neural networks are particularly powerful due to their ability to model complex, non-linear relationships in data, making them suitable for a wide range of applications. In the context of robotics, neural networks have been extensively used for tasks such as image recognition, sequence prediction, and decision-making, significantly enhancing the capabilities of autonomous systems[56].

2.3.1 Convolutional Neural Networks

CNNs are a class of deep learning algorithms specifically designed for processing structured grid data, such as images. CNNs leverage the spatial hierarchies in data through their architecture, which comprises convolutional layers, pooling layers, and fully connected layers. Convolutional layers apply filters to the input data to extract feature maps, highlighting essential patterns like edges, textures, and shapes. Pooling layers then down sample these feature maps, reducing the dimensionality and computational complexity while retaining critical information[57].

CNNs are widely used in robotic vision systems to interpret visual scenes, detect obstacles, and recognize landmarks. For instance, in autonomous navigation, a robot equipped with a CNN can process real-time camera feeds to identify objects and classify them, enabling obstacle detection and avoidance. This capability is crucial for navigating through complex environments where precise visual interpretation is necessary[58].

The architecture of CNNs typically includes multiple convolutional layers interspersed with activation functions like ReLU (Rectified Linear Unit), pooling layers, and finally fully connected layers that output the classification results. The convolutional layers perform feature extraction by applying learned filters to the input image, while the pooling layers reduce the spatial dimensions of the feature maps, making the network more computationally efficient and robust to variations in the input. The fully connected layers interpret the high-level features extracted by the convolutional layers to perform the final classification[59].

Training a CNN involves using a large dataset of labelled images and an optimization algorithm, such as stochastic gradient descent (SGD), to minimize the classification error. The backpropagation algorithm is used to compute gradients of the loss function with respect to the network parameters, which are then updated iteratively. Techniques like data augmentation, dropout, and batch normalization are employed to improve the generalization performance of CNNs and prevent overfitting[60].

2.3.2 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are designed to handle sequential data by maintaining a memory of previous inputs, making them well-suited for tasks involving temporal dependencies. Unlike feedforward neural networks, RNNs have connections that form directed cycles, allowing information to persist over time. This feature enables RNNs to capture the dynamic behavior of sequences, which is essential for applications like speech recognition, language modelling, and time-series prediction [61].

In robotic navigation, RNNs are used to predict the future states of a robot based on its past experiences. This predictive capability is particularly valuable in dynamic environments where the robot must anticipate changes and adjust its path accordingly. For example, an RNN can analyse a sequence of sensor readings to forecast potential obstacles, enabling the robot to plan its movements proactively [62].

RNNs consist of units that take an input vector and a hidden state vector from the previous time step to produce an output vector and a new hidden state vector. The recurrence relation allows the network to maintain a form of memory, enabling it to process sequences of variable length. However, standard RNNs suffer from issues like vanishing and exploding gradients, which make training difficult over long sequences [63].

Training RNNs involves using the backpropagation through time (BPTT) algorithm to compute gradients of the loss function with respect to the network parameters. Techniques like gradient clipping, which prevents gradients from growing too large, and advanced optimizers like Adam, are used to stabilize the training process. Regularization methods such as dropout and L2 regularization help prevent overfitting[64].

2.3.3 Long Short-Term Memory Networks

Long Short-Term Memory networks (LSTMs) are a specialized type of RNN that addresses the limitations of standard RNNs, particularly the issue of long-term dependencies. LSTMs

achieve this by incorporating memory cells and gating mechanisms that control the flow of information, allowing them to retain and utilize information over extended periods [65].

LSTMs are highly effective in navigation tasks that require long-term planning and context awareness. For instance, an LSTM can help a robot remember the layout of an environment it navigated hours earlier, enabling it to make informed decisions when revisiting the area. This long-term memory capability enhances the robot's ability to navigate complex environments with a higher degree of autonomy and reliability [66].

LSTMs introduce a more complex architecture that includes input, output, and forget gates, as well as a cell state that carries long-term information. The gates control the flow of information into and out of the cell state, allowing the network to retain or discard information as needed. This architecture mitigates the vanishing gradient problem, enabling LSTMs to learn long-term dependencies effectively [67].

Training LSTMs involves the same BPTT algorithm used for standard RNNs but benefits from the stabilized gradient flow due to the gating mechanisms. Techniques such as layer normalization and recurrent dropout are employed to enhance the robustness and generalization capability of LSTMs. Additionally, advanced regularization techniques like variational dropout and scheduled sampling are used to improve performance on sequential tasks [68].

2.3.4 Applications in Robot Navigation

Neural networks, especially CNNs, RNNs, and LSTMs, have been successfully applied in various robot navigation tasks, significantly improving their performance and autonomy. In obstacle detection and avoidance, CNNs process visual data to identify and classify obstacles, while RNNs predict their movement, allowing robots to navigate safely through dynamic environments. Path planning is another critical application, where LSTMs use historical navigation data to plan efficient paths, considering both immediate and future obstacles [69].

CNNs are extensively used for real-time obstacle detection and avoidance. By processing images from the robot's cameras, CNNs can detect and classify obstacles in the environment. This information is then used to generate safe paths that avoid collisions. Techniques such as

semantic segmentation, where each pixel in the image is classified into categories like road, obstacle, and free space, enhance the robot's understanding of its environment and improve navigation accuracy [70].

LSTMs are particularly useful for path planning in dynamic environments. By leveraging their ability to remember long-term dependencies, LSTMs can analyse past navigation data to predict future states and obstacles. This predictive capability allows robots to plan more efficient paths that account for both immediate and anticipated changes in the environment. For instance, an LSTM can help a robot navigate through a crowded warehouse by predicting the movement patterns of people and other robots [71].

Combining CNNs and LSTMs, robots can create and update maps of their surroundings in real-time, facilitating better navigation in unfamiliar environments. This capability is essential for autonomous vehicles, drones, and service robots that operate in dynamic and unpredictable settings. Techniques such as SLAM with neural networks integrate visual and depth data to build accurate 3D maps of the environment, enhancing the robot's ability to navigate and interact with its surroundings [72].

2.4 Application of Neural Networks in Navigation and Path Planning

Navigation and path planning are critical components of autonomous robot operation. Neural networks have significantly enhanced these processes by enabling robots to make intelligent decisions in complex and dynamic environments. This section analyses how neural networks are employed for path planning and navigation decision-making, highlighting their effectiveness in dealing with various obstacles.

Traditional path planning algorithms, such as A* and Dijkstra's, have been widely used in robotics. These methods, however, often struggle with dynamic and unpredictable environments where the robot must navigate through changing scenarios with various obstacles. Neural networks, particularly deep reinforcement learning (DRL) techniques, offer a more adaptable solution. DRL allows robots to learn optimal navigation strategies through trial and error, improving their ability to handle unforeseen obstacles[73, 74].

Deep reinforcement learning combines the reinforcement learning framework, where agents learn to make decisions by interacting with their environment, with deep learning, which allows for the processing of high-dimensional sensory inputs. This combination has proven effective in training robots to navigate complex environments autonomously. For instance, a robot can learn to avoid obstacles and navigate towards a target location by receiving rewards for successful actions and penalties for collisions[75].

One of the most prominent DRL algorithms is Deep Q-Network (DQN), which combines Q-learning with deep neural networks. In DQN, the Q-function, which represents the expected cumulative reward of taking an action in each state, is approximated using a neural network. The network parameters are updated using gradient descent to minimize the difference between the predicted and target Q-values[76].

Another advanced DRL technique is Proximal Policy Optimization (PPO), which optimizes the policy directly by updating the policy parameters to maximize the expected reward. PPO employs a surrogate objective function that ensures stable updates by limiting the deviation from the previous policy. This method has been shown to be effective in various robotic control tasks, providing a balance between exploration and exploitation[77].

In environments with multiple robots or agents, Multi-Agent Reinforcement Learning (MARL) is used to coordinate their actions. MARL extends single-agent reinforcement learning to scenarios where agents must learn to cooperate or compete to achieve their objectives. Techniques such as Multi-Agent Deep Deterministic Policy Gradient (MADDPG) enable agents to learn policies that consider the actions of other agents, leading to more efficient and coordinated behaviours[78].

Neural networks enhance navigation decision-making by providing robots with the ability to interpret complex sensory data and predict environmental changes. For example, a robot equipped with a CNN can analyse visual data to identify a clear path, while a LSTM uses past navigation experiences to predict potential obstacles. This combination enables robots to make more informed and accurate navigation decisions[79].

The integration of perception and control is crucial for effective navigation decision-making. Perception involves interpreting sensory data to understand the environment, while control

involves generating the actions required to navigate through it. End-to-end learning approaches, where neural networks are trained to map raw sensory inputs directly to control commands, have shown promise in achieving this integration. For instance, an end-to-end trained neural network can process camera images to directly output steering angles for a robot, streamlining the navigation process[80].

Hierarchical Reinforcement Learning (HRL) decomposes the navigation task into multiple levels of abstraction, allowing for more efficient learning and decision-making. In HRL, a high-level policy selects sub-goals or tasks, while low-level policies execute the actions required to achieve these sub-goals. This hierarchical structure enables robots to learn complex navigation strategies more efficiently by breaking down the task into manageable components[81].

Neural networks excel in environments with complex and dynamic obstacles. For instance, robots navigating through crowded spaces, such as warehouses or urban areas, benefit from the predictive capabilities of RNNs and LSTMs. These networks allow robots to anticipate the movements of people and other objects, ensuring smooth and efficient navigation[82].

Moreover, neural networks can adapt to changing conditions, such as variations in lighting or weather, maintaining high performance in diverse scenarios. This adaptability is crucial for applications like delivery drones and search-and-rescue robots, which must operate reliably in unpredictable and harsh environments[56].

Autonomous delivery drones must navigate through urban environments, avoiding obstacles such as buildings, trees, and power lines. Neural networks, particularly CNNs and LSTMs, play a vital role in enabling these drones to process real-time visual and sensor data, predict potential obstacles, and plan safe flight paths. The adaptability of neural networks allows these drones to operate in varying weather conditions and different times of day, ensuring reliable delivery services[71].

In underwater robotics, the environment is highly dynamic, and traditional navigation methods face significant limitations. Neural networks process sonar and visual data to map the seabed and avoid obstacles. For example, CNNs can analyse sonar images to identify

underwater features, while LSTMs predict the movement of marine life or water currents, enabling autonomous underwater vehicles (AUVs) to navigate safely and efficiently[55].

Dynamic path planning involves continuously updating the robot's path in response to changes in the environment. Neural networks, particularly DRL techniques, enable robots to adapt their paths in real-time, improving navigation efficiency and safety. Techniques such as Model Predictive Control (MPC) combined with neural networks provide a robust framework for dynamic path planning, allowing robots to predict future states and optimize their actions accordingly[62].

In autonomous vehicles, dynamic path planning is critical for navigating through traffic, changing lanes, and avoiding obstacles. Neural networks process data from multiple sensors, including cameras, lidar, and radar, to detect obstacles and predict their movements. The vehicle's path is then continuously updated based on these predictions, ensuring safe and efficient navigation[83].

Ensuring the robustness and safety of neural network-based navigation systems is essential. Techniques such as adversarial training, where neural networks are trained to handle intentionally perturbed inputs, enhance the robustness of these systems. Additionally, hybrid approaches that combine neural networks with traditional rule-based systems provide a balance between flexibility and predictability, ensuring safe operation in diverse scenarios[75].

The future of neural network-based navigation is likely to see advancements in several key areas. Transfer learning, where models trained on one task are adapted to perform another, holds promise for improving the efficiency of training neural networks for navigation tasks. Self-supervised learning, where models learn from unlabelled data, and continual learning, where models adapt to new information without forgetting previous knowledge, are also emerging areas of interest[84].

Advancements in hardware, such as more powerful GPUs and specialized accelerators like TPUs, will enable the deployment of more complex neural network models in real-time navigation systems. Additionally, the integration of neural networks with other emerging

technologies, such as quantum computing and edge computing, will further enhance the capabilities of autonomous systems[85].

Regulatory frameworks and safety standards will also need to evolve to ensure the safe deployment and operation of neural network-based navigation systems. Addressing ethical and societal implications, such as the impact on employment and the potential for bias in neural network models, will be critical for the widespread adoption of these technologies.

2.5 Comparison of Traditional Navigation Methods and Neural Network Navigation

Comparing traditional navigation methods with neural network-based approaches provides insights into their respective advantages, limitations, and optimal application scenarios. This section conducts a comparative analysis, highlighting key differences and potential synergies between these technologies.

2.5.1 Advantages of Traditional Navigation Methods

Traditional navigation methods, particularly SLAM, offer several advantages. SLAM and other traditional techniques have been extensively tested and validated in various applications, providing a proven reliability that is crucial for many industries. These methods are deterministic, meaning they produce predictable and repeatable results, which is vital for safety-critical applications such as industrial robotics and autonomous vehicles[34].

Additionally, traditional algorithms often require less computational power compared to complex neural network models. This lower computational requirement makes traditional methods suitable for resource-constrained environments, where high-performance computing resources may not be available. For instance, traditional SLAM can be implemented on low-cost hardware platforms, making it accessible for a wide range of applications[28].

2.5.2 Detailed Analysis of SLAM

SLAM algorithms can be divided into two main categories: feature-based and direct methods. Feature-based methods rely on extracting and matching distinct features from sensor data, such as corners or edges, to build a map and localize the robot. Popular feature-based SLAM techniques include ORB-SLAM, which uses Oriented FAST and Rotated BRIEF (ORB) features, and PTAM (Parallel Tracking and Mapping), which allows real-time tracking and mapping[86].

Direct methods, on the other hand, use the raw sensor data directly without extracting features. These methods, such as LSD-SLAM (Large-Scale Direct Monocular SLAM) and DSO (Direct Sparse Odometry), offer higher accuracy and robustness in environments with few distinct features. Direct methods can better handle dynamic environments and changes in lighting conditions, making them suitable for applications like augmented reality and autonomous driving[87].

Traditional navigation methods, particularly those based on SLAM, are designed to be resource efficient. They can be implemented on low-cost hardware, making them accessible for a wide range of applications, from small-scale robotics to large industrial systems. This resource efficiency extends to their scalability, as traditional methods can be deployed in large environments with minimal computational overhead[36].

One of the key strengths of traditional navigation methods is their robustness and reliability. These methods have been extensively tested and validated in various applications, ensuring that they produce predictable and repeatable results. This reliability is crucial for safety-critical applications, such as industrial robotics and autonomous vehicles, where any failure in the navigation system could lead to catastrophic consequences[88].

2.5.3 Advantages of Neural Network Navigation

Neural network-based navigation approaches bring unique benefits, particularly in their adaptability and robustness to noise. Neural networks can adapt to changing environments and learn from new data, which enhances their performance in dynamic scenarios. This adaptability is particularly beneficial in unstructured environments where traditional methods may struggle to maintain accuracy[56].

Neural networks are generally more robust to sensor noise and variations, improving navigation accuracy in real-world conditions. This robustness is essential for applications such as autonomous driving, where the operating environment can change rapidly due to weather conditions, traffic, and other factors. Neural networks can process large volumes of sensor data in real-time, enabling them to make more informed navigation decisions[75].

Neural network-based navigation systems leverage various deep learning architectures to process and interpret sensor data. CNNs are widely used for image and video processing, enabling robots to detect and classify objects in their environment. RNNs and LSTM networks are employed for sequential data processing, allowing robots to predict future states based on past experiences[89].

Another significant advantage of neural networks is their ability to perform end-to-end learning. This capability allows them to learn navigation tasks directly from raw sensory input, reducing the need for handcrafted features and models. For example, an end-to-end trained neural network can take raw images from a camera and directly output steering commands for a vehicle, simplifying the overall system design and potentially improving performance[90].

Neural networks excel in their ability to generalize from training data to new, unseen environments. This generalization capability is particularly important for navigation in dynamic and unstructured environments, where the robot may encounter situations that were not present in the training data. Neural networks can learn to adapt to these new scenarios by leveraging their deep architectures and large amounts of training data[84].

2.5.4 Limitations and Optimal Application Scenarios

While traditional navigation methods offer robustness and reliability, they also have limitations. One significant limitation is their reliance on accurate and consistent sensor data. In dynamic and unstructured environments, sensor noise and variations can degrade the performance of traditional methods. Additionally, traditional methods may struggle with scalability in highly complex environments, where the number of features or the size of the map becomes too large to handle efficiently[34].

Neural network methods are ideal for complex, dynamic, and unstructured environments where adaptability and robustness are essential. However, they require substantial computational resources for training and inference, which can be a limitation in resource-constrained settings. Moreover, neural networks often require large amounts of labelled training data, which can be time-consuming and expensive to obtain. Finally, the black-box nature of neural networks makes them difficult to interpret and debug, posing challenges for ensuring safety and reliability in critical applications[79].

Traditional navigation methods are best suited for well-structured and static environments where reliability and predictability are paramount. These methods excel in scenarios where the environment does not change significantly over time, and where computational resources are limited. For example, traditional SLAM is ideal for industrial robots operating in controlled factory settings, where the layout and features of the environment are known and relatively constant[36].

In contrast, neural network methods are ideal for complex, dynamic, and unstructured environments where adaptability and robustness are essential. These methods are particularly useful in applications such as autonomous vehicles, drones, and robots operating in unpredictable environments. For instance, neural networks can be used to navigate autonomous vehicles through busy city streets, where the environment is constantly changing and the robot must adapt in real-time[90].

2.6 Integration of Traditional and Neural Network Navigation Techniques

Integrating traditional navigation methods with neural network approaches holds the potential to significantly enhance robotic navigation capabilities. This integration leverages the complementary strengths of both techniques, providing enhanced accuracy, improved adaptability, and efficient resource utilization. This section explores the benefits and challenges of such integration, highlighting potential contributions to navigation in dynamic environments.

2.6.1 Complementary Strengths

Combining traditional and neural network-based techniques leverages their complementary strengths. Traditional navigation methods, such as SLAM, provide a reliable and well-understood framework for mapping and localization. They are deterministic and require less computational power, making them suitable for resource-constrained environments[34]. On the other hand, neural networks offer adaptability and robustness to noise, enabling robots to learn from new data and adjust to changing environments[56].

By integrating these approaches, robots can achieve enhanced accuracy and adaptability. For instance, neural networks can refine and improve the maps generated by traditional SLAM, leading to more accurate localization and mapping. This integration allows for the development of hybrid systems that combine the reliability of traditional methods with the flexibility of neural networks.

Hybrid navigation systems that combine SLAM with neural networks have shown promising results in various applications. For example, a robot might use traditional SLAM to create an initial map of the environment. This map provides a reliable baseline for navigation. The neural network can then be employed to update and refine this map based on new sensory data, such as changes in the environment or the appearance of new obstacles[86].

One practical application of hybrid navigation systems is in autonomous vehicles. These vehicles can use traditional SLAM to build a map of the road network and then employ neural networks to process real-time data from cameras, lidar, and radar. The neural networks can detect dynamic obstacles, such as pedestrians and other vehicles, and predict their movements. This integrated approach enhances the vehicle's ability to navigate safely and efficiently through complex urban environments[79].

Integrating traditional and neural network navigation techniques can also optimize computational resources. Traditional methods can handle basic navigation tasks, such as mapping and localization, while neural networks can be reserved for more complex decision-making processes. This division of labour allows for more efficient use of computational power, ensuring that the robot can operate effectively even in resource-constrained environments[94].

For example, in a warehouse setting, a robot might use traditional SLAM to navigate the static layout of the storage racks. When it encounters a dynamic obstacle, such as a moving forklift, the neural network can take over to predict the obstacle's movement and plan a safe path around it. This approach ensures that the robot can navigate efficiently while minimizing computational load.

2.6.2 Challenges and Solutions in Integration

System Complexity One of the main challenges of integrating traditional and neural network navigation methods is managing the increased system complexity. Combining multiple navigation techniques requires sophisticated algorithms to ensure seamless operation and effective communication between different components. Developing modular frameworks that allow for the independent development and testing of each component can help manage this complexity[55].

Data Fusion Effective data fusion is critical for integrating traditional and neural network navigation methods. Combining data from multiple sensors requires advanced algorithms to handle differences in data types, sampling rates, and noise characteristics. Techniques such as Kalman filtering, particle filtering, and neural network-based sensor fusion are commonly used to integrate data from various sources, ensuring accurate and reliable navigation[52].

Real-Time Processing Ensuring real-time processing capabilities is another challenge in integrating traditional and neural network navigation methods. Both approaches must process large volumes of data quickly to provide timely navigation decisions. Optimizing algorithms for speed and efficiency, and deploying them on specialized hardware such as GPUs and TPUs, can enhance real-time performance[90].

Robustness and Adaptability Maintaining robustness and adaptability in integrated navigation systems is essential. Traditional methods provide reliable baseline navigation, while neural networks offer adaptability to changing environments. Ensuring that the integrated system can switch between these methods seamlessly and adapt to new scenarios is crucial for robust operation. Developing hybrid architectures that can

dynamically adjust their behavior based on environmental conditions can help achieve this goal[84].

2.6.3 Future Research Directions

Adaptive Hybrid Systems Future research in integrated navigation techniques is likely to focus on developing adaptive hybrid systems that can dynamically switch between traditional and neural network methods based on the current environment and task requirements. These systems will use traditional methods for stable and well-mapped environments and switch to neural network-based approaches in more dynamic or unstructured scenarios[54].

Advanced Sensor Fusion Advancements in sensor fusion techniques will also play a critical role in the future of integrated navigation systems. Developing more sophisticated algorithms that can handle high-dimensional and noisy data from multiple sensors will enhance the accuracy and reliability of navigation. Machine learning-based sensor fusion techniques, which can learn optimal ways to combine data from different sensors, are an emerging area of interest[79].

Scalability and Efficiency Improving the scalability and efficiency of integrated navigation systems is another important research direction. As robots are deployed in larger and more complex environments, ensuring that navigation systems can scale accordingly without compromising performance is crucial. Techniques such as distributed SLAM, where multiple robots collaborate to build a map, and cloud-based processing, where computationally intensive tasks are offloaded to remote servers, can help achieve scalability[91].

Human-Robot Collaboration Enhancing human-robot collaboration is another key area of research. Developing integrated navigation systems that can understand and predict human behavior, and adapt their actions, accordingly, will improve the usability and acceptance of robots in shared environments. Techniques such as imitation learning and reinforcement learning, where robots learn from human demonstrations and feedback, can enhance human-robot collaboration[95].

2.7 Challenges and Future Research Directions

The integration of traditional and neural network-based navigation techniques presents numerous opportunities, but it also comes with significant challenges. This section addresses common challenges such as adaptability, system complexity, and resource demands, and discusses future research trends and potential technological developments that could enhance robotic navigation capabilities in dynamic environments.

2.7.1 Common Challenges

Adaptability One of the primary challenges in robotic navigation is ensuring that systems can adapt to new and changing environments. Traditional navigation methods often struggle with adaptability, as they rely on pre-mapped environments and can be limited by their ability to handle dynamic changes. Neural networks, while more adaptable, require substantial amounts of training data to perform effectively. Collecting and labelling this data can be time-consuming and resource-intensive[28].

Adaptability is particularly crucial in scenarios where robots must operate in environments that are constantly changing, such as urban areas or disaster zones. In such settings, robots must be able to learn and update their navigation strategies in real-time to handle new obstacles and changes in the environment. Developing more efficient training algorithms and techniques that can reduce the data requirements for neural networks is a critical area of research[56].

System Complexity Combining traditional and neural network navigation methods increases the overall complexity of robotic systems. Integrating multiple navigation techniques requires sophisticated frameworks that can manage the interactions between different algorithms and sensors. This complexity can pose significant challenges in terms of implementation, maintenance, and debugging[54].

Managing system complexity is essential to ensure that robotic systems remain reliable and efficient. Researchers are exploring modular approaches to system design, where different components of the navigation system are developed and tested independently before being

integrated. This approach can help reduce the complexity of the overall system and make it easier to manage and maintain[34].

Resource Demands Advanced neural networks require substantial computational power for training and inference, which can be a limitation in resource-constrained settings. Mobile robots and autonomous vehicles, for example, often have limited onboard computing resources and may struggle to run complex neural network models in real-time. This issue is particularly relevant for applications that require low latency and high throughput, such as autonomous driving[66].

To address this challenge, researchers are developing more efficient neural network architectures and training techniques that can reduce computational demands. Techniques such as model pruning, quantization, and hardware accelerators like GPUs and TPUs can enhance the efficiency of neural network-based navigation systems. Additionally, edge computing, where data processing is performed locally on the device rather than in the cloud, can help reduce latency and improve real-time performance[94].

2.7.2 Future Research Trends

Improved Learning Algorithms Future research in robotic navigation is likely to focus on developing improved learning algorithms that can enhance the performance and adaptability of neural network-based systems. Transfer learning, where models trained on one task are adapted to perform another, holds promise for improving the efficiency of training neural networks for navigation tasks. Self-supervised learning, where models learn from unlabelled data, and continual learning, where models adapt to new information without forgetting previous knowledge, are also emerging areas of interest[84].

These advanced learning algorithms can help reduce the amount of labelled data required for training and enable robots to adapt more quickly to new environments. Researchers are also exploring meta-learning, where models learn to learn by adapting quickly to new tasks with minimal training. This approach can significantly enhance the adaptability and efficiency of robotic navigation systems[96].

Robust Sensor Fusion Robust sensor fusion techniques are essential for improving navigation accuracy and robustness. By integrating data from diverse sensors, such as cameras, lidar, radar, and IMUs, robots can form a comprehensive understanding of their environment. Advanced sensor fusion algorithms can account for uncertainties and correlations between different sensors, enhancing the reliability of the navigation system[52].

Researchers are developing new sensor fusion techniques that leverage neural networks and probabilistic models to combine data from multiple sensors. These techniques can improve the accuracy of localization and mapping, especially in challenging environments with high levels of noise or dynamic changes. Additionally, exploring ways to integrate new types of sensors, such as thermal cameras or hyperspectral imaging, can provide additional information that enhances navigation capabilities[79].

Human-Robot Interaction Human-robot interaction is another crucial area of research. Understanding and predicting human behavior can facilitate smoother navigation in shared environments, such as hospitals, factories, and public spaces. Developing robots that can interact seamlessly with humans, understanding their intentions and adapting their behavior accordingly, is essential for enhancing usability and acceptance[97].

Researchers are exploring various approaches to improve human-robot interaction, including the use of social navigation algorithms that allow robots to navigate in ways that are socially acceptable and predictable to humans. Machine learning techniques, such as reinforcement learning and imitation learning, are being used to teach robots how to interact with humans based on observing human behavior and receiving feedback[95].

Edge Computing Edge computing offers significant potential for advancing robotic navigation. By processing data locally on the device rather than relying on cloud-based resources, edge computing can reduce latency and dependency on external networks. This approach is particularly beneficial for real-time navigation in dynamic environments, where timely responses are critical[85].

Researchers are developing edge computing platforms that are optimized for running neural network models on resource-constrained devices. These platforms combine efficient hardware accelerators with software optimizations to provide high-performance computing

capabilities in a compact and energy-efficient form factor. Edge computing can enhance the real-time capabilities of navigation systems and enable more robust and adaptive behavior[98].

Advanced Sensor Technologies Advancements in sensor technology can improve the durability and performance of navigation systems in harsh environments. For example, developing sensors with better resistance to environmental factors like dust, moisture, and temperature variations can provide more reliable data for navigation. Additionally, new types of sensors, such as quantum sensors, can offer unprecedented levels of accuracy and sensitivity[99].

Quantum sensors, for instance, can measure physical quantities with extreme precision, enabling more accurate localization and mapping. These sensors can be particularly useful in environments where traditional sensors struggle, such as underwater or space exploration. Integrating advanced sensor technologies with neural network-based navigation systems can significantly enhance their capabilities and reliability[93].

Quantum Computing Quantum computing is another promising area, offering the potential to solve complex navigation problems more efficiently. Quantum algorithms can process large volumes of sensor data and perform complex calculations at unprecedented speeds, significantly enhancing the capabilities of navigation systems. For example, quantum algorithms can optimize path planning by exploring multiple potential routes simultaneously, finding the most efficient path in a fraction of the time required by classical algorithms[100].

While quantum computing is still in its early stages, ongoing research is exploring how to leverage its capabilities for robotic navigation. Researchers are developing quantum algorithms tailored to specific navigation tasks and exploring hybrid systems that combine quantum and classical computing to achieve optimal performance[101].

Autonomous Systems and Ethical Considerations As robotic navigation systems become more advanced and widespread, addressing ethical and societal implications will be critical. Ensuring the safety and reliability of autonomous systems is paramount, especially in applications such as autonomous vehicles and drones. Developing regulatory frameworks

and safety standards that govern the deployment and operation of these systems will be essential for ensuring public trust and acceptance[102].

Researchers are also exploring ways to ensure that neural network models used in navigation are transparent and interpretable, allowing for better understanding and debugging of their behavior. Addressing potential biases in training data and ensuring that models are trained on diverse datasets can help prevent unintended consequences and promote fairness in autonomous systems[103].

2.8 Conclusion

This literature review has thoroughly examined the applications of both traditional and neural network-based navigation technologies in the realm of service robotics. By exploring the foundational principles, developments, and implementations of these technologies, the review has provided a comprehensive understanding of their respective advantages, limitations, and potential for integration. The following key points summarize the main findings:

Foundations of Traditional Navigation Methods: Traditional navigation techniques, particularly SLAM, have laid a robust foundation for autonomous robotic navigation. These methods are characterized by their reliability, predictability, and lower computational requirements, making them suitable for structured and static environments. However, they face challenges in adaptability and handling dynamic changes[34].

Neural Network Navigation: Neural networks offer significant advantages in adaptability and robustness to noise. They excel in dynamic and unstructured environments, processing large volumes of sensory data in real-time to make informed navigation decisions. However, their high computational demands and the need for extensive training data pose challenges in resource-constrained settings[56].

Perception and Processing of Dynamic Environments: Neural networks, particularly CNNs and RNNs, play a crucial role in real-time data processing and environmental interaction.

These technologies enable robots to perceive and adapt to changes in their surroundings, enhancing their ability to navigate safely and efficiently in dynamic environments[104].

Application in Navigation and Path Planning: Neural network-based navigation and path planning techniques, especially DRL, provide robots with the ability to learn optimal navigation strategies through trial and error. These techniques are particularly effective in handling complex, dynamic obstacles and making real-time adjustments to navigation paths[73].

Comparison of Traditional and Neural Network Methods: Both traditional and neural network-based methods have distinct advantages and limitations. Traditional methods are ideal for well-structured environments, while neural networks excel in dynamic and unpredictable settings. Integrating these approaches leverages their complementary strengths, providing enhanced accuracy and adaptability[28].

Integration of Navigation Techniques: The integration of traditional and neural network navigation methods holds significant potential for improving robotic navigation capabilities. Hybrid systems that combine SLAM with neural networks can achieve higher accuracy and adaptability, optimizing computational resources and enhancing performance in dynamic environments[86].

Challenges and Future Research Directions: Key challenges in robotic navigation include system complexity, data fusion, real-time processing, and robustness. Future research will focus on developing adaptive hybrid systems, advanced sensor fusion techniques, scalable and efficient navigation algorithms, and enhanced human-robot collaboration. Emerging technologies such as edge computing, advanced sensors, and quantum computing offer promising avenues for advancing the field[85].

The future of robotic navigation lies in the seamless integration of traditional and neural network-based methods, leveraging their respective strengths to create robust, adaptable, and efficient systems. Several key areas of research and development will drive this integration forward:

Adaptive Hybrid Systems: Developing systems that can dynamically switch between traditional and neural network methods based on the current environment and task requirements will enhance the versatility and efficiency of robotic navigation[84].

Advanced Sensor Fusion: Improving sensor fusion techniques to handle high-dimensional and noisy data from multiple sources will enhance the accuracy and reliability of navigation systems. Machine learning-based sensor fusion methods are a promising area of research[96].

Scalability and Efficiency: Ensuring that navigation systems can scale to larger and more complex environments without compromising performance is crucial. Distributed SLAM and cloud-based processing are potential solutions for achieving scalability[54].

Human-Robot Collaboration: Enhancing human-robot interaction by developing systems that can understand and predict human behavior will improve the usability and acceptance of robots in shared environments. Techniques such as imitation learning and reinforcement learning will play a key role in this area[95].

Emerging Technologies: Leveraging emerging technologies such as edge computing, advanced sensors, and quantum computing will significantly enhance the capabilities of robotic navigation systems. These technologies offer the potential for improved real-time performance, higher accuracy, and greater robustness[99].

Chapter 3. General methodology and techniques

3.1 Introduction

This chapter outlines the design and simulation process of an autonomous differential drive robot, focusing on the integration of hardware and software components. The robot is designed to navigate complex environments using advanced sensors, actuators, and control algorithms. Key components, such as the Hokuyo URG-04LX-UG01 LiDAR and Arduino Mega 2560, shown in table 3-1, are selected for their precision, reliability, and compatibility with the system's requirements.

The simulation is conducted using Webots and MATLAB, two powerful tools for testing and validating robotic systems. Webots provides a realistic virtual environment for evaluating the robot's performance, while MATLAB supports algorithm development and data analysis. The simulation environment is populated with obstacles to mimic real-world conditions, and the robot's navigation relies on a differential drive system for precise movement.

By combining hardware integration, simulation, and algorithmic development, this chapter lays the foundation for understanding the robot's capabilities and the steps taken to achieve reliable autonomous navigation.

Table 3-1. Robot components and the rationale for their selection.

Component	Reason for Selection
Hokuyo URG-04LX-UG01 LiDAR	High precision and reliability in distance measurement Fast response time for real-time data capture Compact and lightweight, suitable for mobile robots
Nema 17 Stepper Motors	Offers precise control over movement, essential for accurate navigation Known for durability and reliability
DRV8834 Low-Voltage Stepper Motor Driver Carrier	Supports efficient power management to extend operational life Allows up to 1/32 micro stepping for smooth and precise motor control
Arduino Mega 2560	Provides extensive I/O capabilities necessary for managing multiple sensors and actuators Benefits from strong community support and extensive libraries
XBee-PRO S1 Module	Ensures long-range wireless communication, essential for remote operations High reliability in data transmission ensures integrity and accuracy of sensor data and control commands
Minimum-9 v5 Gyro, Accelerometer, and Compass	Combines gyroscope, accelerometer, and compass for comprehensive motion sensing and orientation Compact size fits well in mobile robots

3.1.1 Hokuyo URG-04LX-UG01 LiDAR

The Hokuyo URG-04LX-UG01 LiDAR is chosen primarily for its precision and reliability in distance measurement. This accuracy is indispensable for creating detailed and trustworthy maps necessary for effective navigation and obstacle avoidance. Equipped with a scanning rate of 100 Hz, the Hokuyo LiDAR captures the environment rapidly, providing continuous, real-time updates. This quick data acquisition is crucial for the robot to make timely decisions and adapt quickly to dynamic changes in its environment, such as moving obstacles or alterations in the landscape.

Additionally, the LIDAR's compact and lightweight design is particularly suitable for mobile robotic applications. Its small size ensures that it can be easily integrated into the robot without significantly affecting the overall weight distribution or hindering its mobility. Moreover, the LIDAR offers a wide 240-degree scanning range, which enhances the robot's ability to detect obstacles not only directly ahead but also peripherally, providing a broad field of view that is beneficial for navigating through complex environments.

The ease of integration of the Hokuyo LIDAR with other system components, such as the Arduino Mega 2560 and the robot's sensor array, further underscores its suitability for the project. This ease of integration is vital for maintaining a streamlined design and operational efficiency within the robot's hardware setup.

These combined features make the Hokuyo URG-04LX-UG01 an excellent choice for the project, aligning well with the requirements for robust, real-time sensory data in autonomous navigation tasks.

3.1.2 Nema 17 Stepper Motors

Nema 17 stepper motors are selected for their precision in control, which is crucial for accurately navigating the differential drive wheeled robot. These motors provide fine control over the robot's movements, allowing for precise adjustments in position and speed that are essential in an environment where slight deviations can lead to significant navigational errors. The ability of stepper motors to move in exact increments is beneficial for path accuracy and repeatability, which are key in autonomous robotic applications.

Moreover, Nema 17 motors are known for their reliability and durability. In the context of autonomous robots, these characteristics ensure that the motors can withstand continuous operation under varying loads and environment conditions without failure. This durability is particularly important for field applications where maintenance opportunities may be limited, and operational reliability is a priority.

The compact size of the Nema 17 motors also contributes to their choice. Despite their robust functionality, these motors do not occupy excessive space, allowing for a more compact robot design. This compactness is vital for ensuring that the robot remains agile and can navigate through tight spaces, which is often required in cluttered or dynamically changing environments.

Their widespread use in both industrial and hobbyist robotics projects means that Nema 17 stepper motors have extensive support in terms of availability of spare parts, compatibility with a wide range of driver boards, and community knowledge. This makes troubleshooting and enhancements easier, which is a significant advantage during both the development and operational phases of the robot project.

This combination of precision, reliability, durability, and supportability makes Nema 17 stepper motors an ideal choice for the project, providing the necessary movement control capabilities required for sophisticated autonomous navigation.

3.1.3 DRV8834 Low-Voltage Stepper Motor Driver Carrier

The DRV8834 Low-Voltage Stepper Motor Driver Carrier is chosen for its ability to efficiently manage power and provide precise motor control, crucial for the autonomous navigation of the robot. This driver supports up to 1/32 microstepping, a feature that enhances the smoothness of the stepper motors' operation. Microstepping allows for finer control over motor movements, enabling the robot to navigate with higher precision and less mechanical noise, which is particularly beneficial in environments that require delicate manoeuvring.

Another significant advantage of the DRV8834 is its low-voltage operation, which aligns well with the power supply constraints typical in mobile robotics. This compatibility ensures that

the motor driver can operate effectively without requiring a heavy or bulky power supply, thus maintaining the robot's overall light weight and compact size. Efficient power management is also a standout feature of the DRV8834, which helps in minimizing power consumption during operations. This is essential for prolonging the robot's operational time on a single battery charge, a critical factor in autonomous applications where longevity and efficiency are priorities.

The DRV8834 also provides built-in protection against over-current, under-voltage, and over-temperature conditions, enhancing the reliability and safety of the motor operations. These protective features are vital for preventing damage to the motor and driver in scenarios where the robot might encounter unexpected mechanical resistance or operate in varying environmental conditions.

Furthermore, the ease of interfacing the DRV8834 with the control systems, specifically the Arduino Mega 2560, is another reason for its selection. This driver integrates seamlessly into the existing electronics framework, facilitating straightforward programming and control logic development.

The combination of precise control, efficient power management, and robust safety features makes the DRV8834 an excellent choice for driving the Nema 17 stepper motors in the robot, ensuring that the motor operations are both reliable and highly efficient.

3.1.4 Arduino Mega 2560

The Arduino Mega 2560 is chosen as the primary control unit for the differential drive wheeled robot due to its impressive array of input/output capabilities, which are crucial for managing the diverse sensors and actuators involved in this project. With its 54 digital I/O pins and 16 analogy inputs, the Arduino Mega provides ample connectivity options for interfacing with the robot’s multiple sensors including LIDAR, IMU, and motor drivers, as well as various communication modules.

Another key advantage of using the Arduino Mega 2560 is its robust processing power relative to its size and cost, which makes it a cost-effective choice for student and hobbyist

projects without sacrificing performance. The microcontroller is capable of handling complex computations required for real-time sensor data processing and decision-making, which are integral to autonomous navigation systems.

The Arduino platform also benefits from a vast community of developers and a wealth of shared libraries and tools, which significantly reduce the development time and increase the resources available for troubleshooting and enhancements. This extensive support helps in quickly implementing functionalities ranging from basic motor control to more advanced algorithms for sensor data fusion and path planning.

Moreover, the Arduino Mega 2560's compatibility with various shields and modules, such as USB host shields for connecting LIDAR sensors and wireless shields for XBee modules, makes it an adaptable choice for expanding the robot's capabilities. This flexibility allows for easy upgrades and modifications, which can be necessary as the project evolves or as new technologies become available.

The choice of Arduino Mega 2560 ultimately provides a balance between performance, flexibility, and community support, making it an ideal core for developing a versatile and capable autonomous robot.

3.1.5 XBee-PRO S1 Module

The XBee-PRO S1 module is selected for its exceptional wireless communication capabilities, which are essential for the remote operation and telemetry of the autonomous robot. One of the standout features of the XBee-PRO S1 is its extended range, capable of maintaining a reliable communication link over long distances (up to 1 mile line-of-sight). This range is particularly beneficial for operations in large or complex environments where the robot needs to maintain connectivity with the control station without direct line of sight.

The XBee-PRO S1 also supports high data rate transmissions, which is crucial for sending real-time sensor data, including LIDAR and IMU outputs, back to the host computer for processing. This ensures that there is minimal latency in the control feedback loop, allowing for timely and accurate responses to navigational commands and sensor inputs.

Another critical factor in choosing the XBee-PRO S1 is its robustness in various environmental conditions, which ensures reliable performance even in areas with potential interference or obstacles that could disrupt wireless signals. Its use of the 2.4 GHz frequency band is advantageous due to the widespread availability and compatibility with standard communication devices, facilitating easier integration into a variety of project environments.

Moreover, the XBee-PRO S1 is known for its ease of configuration and integration with existing hardware, such as the Arduino Mega 2560. It utilizes a simple serial communication interface, which allows for straightforward setup and programming, reducing development complexity and time. This module also includes advanced networking capabilities, such as mesh networking, which can be invaluable in scenarios where multiple robots need to communicate or when extending the communication range is necessary.

Choosing the XBee-PRO S1 enhances the robot's ability to communicate effectively over large distances and in challenging environments, ensuring reliable and continuous operation critical for autonomous navigation tasks.

3.1.6 Minimum-9 v5 Gyro, Accelerometer, and Compass

The MinIMU-9 v5 was chosen for its comprehensive integration of three critical sensors: a gyroscope, an accelerometer, and a compass. This combination offers a full suite of motion tracking capabilities, which is pivotal for enhancing the robot's navigation and positioning systems.

The gyroscope in the MinIMU-9 v5 provides angular rate measurements, which help in determining the orientation and rotational movement of the robot. This is essential for maintaining balance and stability during manoeuvres, particularly in environments where precision is crucial, such as navigating tight corridors or avoiding obstacles.

The accelerometer contributes by measuring the acceleration forces acting on the robot, which can be used to deduce travel speed and changes in motion. These measurements are crucial for dynamic speed adjustment and for initiating corrective actions when deviations from the intended path are detected.

The compass, or magnetometer, offers directional orientation, complementing the gyroscope and accelerometer by providing a stable frame of reference based on Earth's magnetic field. This ensures that the robot maintains its course over longer distances and under varying environmental conditions.

The compact size of the MinIMU-9 v5 is also a significant advantage, as it allows these sensors to be integrated into a small mobile robot without significantly impacting the weight distribution or the overall form factor. This integration supports the robot's agility and responsiveness.

Moreover, the MinIMU-9 v5 supports I2C communication, which facilitates easy and efficient data transfer to the main control unit, the Arduino Mega 2560. This communication protocol is ideal for high-speed data exchanges and simplifies the connectivity within the robot's electronic system.

The selection of the MinIMU-9 v5 Gyro, Accelerometer, and Compass is thus driven by its ability to provide comprehensive, reliable, and precise sensor data, crucial for advanced navigation tasks. This ensures that the robot can effectively perceive and interact with its surroundings, crucial for achieving robust autonomous navigation.

3.1.7 Webots Simulation Software

Webots was chosen as the simulation platform for its robust capability to create highly realistic and interactive environments for testing and validating robotic applications. Webots offers a physics-based simulation that accurately models the real-world physics, including gravity, friction, and collisions. This high-fidelity simulation is critical for ensuring that the scenarios and conditions the robot faces in the virtual environment closely mimic those it will encounter in the real world.

One of the standout features of Webots is its versatility in supporting a wide range of robotic models and sensors, allowing custom configurations that match the specific hardware used in the robot, such as the Hokuyo LIDAR and Nema 17 stepper motors. This capability ensures

that the behaviours and interactions observed in the simulation provide a reliable indicator of how the robot will perform physically.

Additionally, Webots supports various programming languages, including C, C++, Python, and Java, which aligns well with the development tools used in the robot's control systems. This flexibility facilitates seamless transitions from simulation testing to real-world application, allowing for consistent programming environments, and reducing the learning curve for developers.

Webots also includes advanced features such as automated testing and benchmarking tools, which are invaluable for systematic evaluation of the robot’s performance under various conditions. These tools help in conducting extensive and repeatable tests, ensuring comprehensive coverage of potential scenarios the robot might face.

Another important factor in selecting Webots is its active community and ongoing support. The community provides a wealth of resources, including tutorials, documentation, and forums where developers can exchange ideas and solutions. This support is crucial for troubleshooting and enhancing the robot's capabilities based on collective experiences and advancements shared within the community.

By leveraging Webots for simulation, the project benefits from accelerated development cycles, as issues can be identified and addressed virtually before real-world deployment. This significantly reduces the risk and cost associated with physical prototyping and testing, making Webots an ideal choice for rigorous and efficient developmental workflows.

3.1.8 MATLAB

MATLAB is selected for its comprehensive computational capabilities, particularly in handling complex data analysis, algorithm development, and system modelling, which are integral to the development of autonomous robotic systems. Its extensive suite of built-in functions and specialized toolboxes makes it particularly suited for tasks like signal processing, machine learning, and control systems design, which are central to this project.

MATLAB's Robotics System Toolbox is especially valuable for this application. It provides algorithms and functions for designing, simulating, and testing robot applications. These include motion planning, kinematics, and dynamics which are crucial for understanding and implementing the movements and behaviours of an autonomous robot. The toolbox also supports 3D simulation environments and automatic code generation, which allows for the seamless integration of algorithm development and testing with actual hardware implementation.

Another significant advantage of using MATLAB is its ability to handle and process large sets of data from various sensors such as LIDAR, IMU, and visual cameras. MATLAB's environment supports the visualization and detailed analysis of this data, which is critical for fine-tuning sensor fusion algorithms and improving the accuracy and reliability of the robot's navigation and perception systems.

MATLAB's Simulink, an add-on to MATLAB, offers a graphical interface for modelling, simulating, and analysing multidomain dynamical systems. This feature is instrumental in visually planning and iterating on complex control systems, providing a clear and interactive means of prototyping before committing to code, which enhances both the development speed and accuracy of the implemented algorithms.

Moreover, MATLAB's widespread use in both academia and industry for robotics and engineering projects ensures it is continually updated with the latest algorithms and features and supported by a vast community of users. This community provides an invaluable resource for troubleshooting, learning, and collaborating on complex problems, making MATLAB an ideal tool for cutting-edge robotic development.

By incorporating MATLAB into the workflow, the project leverages its powerful computational tools and extensive support network to enhance the development and operational effectiveness of the autonomous robot. This choice not only aligns with the technical needs of the project but also with the goal of maintaining a high standard of reliability and innovation in the robot's design and function.

MATLAB is favoured over Python or other programming languages for the autonomous navigation robot project due to its robust, specialized capabilities tailored to engineering and

robotics applications. It provides a vast array of advanced toolboxes, such as the Robotics System Toolbox, which streamlines tasks like robot modelling, simulation, path planning, and sensor fusion without the need to integrate multiple external libraries. Simulink, an add-on to MATLAB, offers a seamless visual programming environment for designing and simulating control systems, enabling intuitive modelling and automatic code generation. MATLAB's optimized performance in numerical computations and matrix operations makes it particularly efficient for processing large sensor datasets, which is critical for this project. Additionally, MATLAB supports real-time testing and direct interfacing with hardware such as Arduino and Raspberry Pi, simplifying hardware integration compared to Python, which often requires more setup and customization. Widely recognized in academia and industry, MATLAB benefits from extensive documentation, community support, and compatibility with industry standards, ensuring reliability and facilitating collaboration. Its educational support, including discounts and practical resources, further makes it an ideal choice for research and student projects. These advantages make MATLAB the preferred option for meeting the complex and precise requirements of autonomous robotic navigation.

3.2 System setup

This research utilized three different computers and their configuration are shown in Table 3-2. In the initial stages, simulations, and preliminary programming, along with the construction of the initial model, were conducted on a home PC and a laptop. This approach was primarily adopted due to the constraints imposed by the COVID-19 pandemic. However, it was evident that the laboratory computer offered superior performance for more advanced tasks. This included the execution of simulations involving data collection via Webots and the training of neural networks using MATLAB. The lab PC's enhanced capabilities facilitated more efficient and effective processing of these complex computational tasks.

Table 3-2. The configuration of three computers used for running the simulation.

	CPU	RAM	GPU
Home	AMD Ryzen 5 3600 6-Core	16 GB	Nvidia GeForce GTX 1660 SUPER
Laptop	Intel Core Processor i9-9880H 8-Core	32 GB	Nvidia Quadro P620
Lab	Xeon Gold 5217	192 GB	NVIDIA Quadro RTX A6000

3.2.1 Robot model building

In this simulation, the robot replicates the design from our first-year study: a two-wheeled differential robot equipped with two universal wheels at the front and rear. This robot features a triple-layer circular chassis configuration, with the topmost chassis hosting a Hokuyo LiDAR sensor. To streamline the model for simulation purposes, components like Arduinos, the battery, breadboard, and wiring are excluded. Building upon this simplified model, we've incorporated a collision sensor and a compass to enhance the simulation's functionality. The robot's bumper has been specifically designed to be both long and sizable, ensuring efficient detection of obstacles that may hinder its movement during operation.

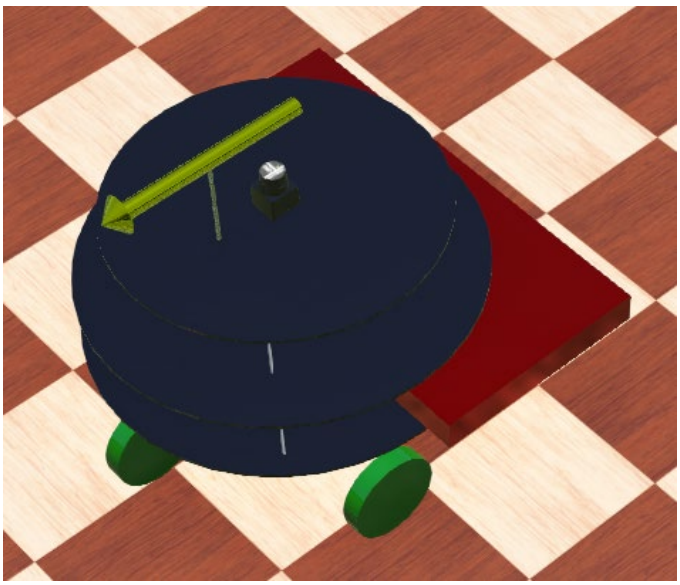


Figure 3-1. The model of simulated robot in Webots. It is a two-wheel differential robot with a LiDAR sensor, a bump sensor, and a compass on the top.

3.2.2 Environment building

The robotic navigation environment is designed as a spacious, enclosed rectangular area measuring 20 meters by 100 meters. This space is populated with a variety of boxes, each varying in size, strategically placed to simulate real-world obstacles. The flooring within this area features a distinct chequered parquetry pattern, with each square measuring 0.5 meters by 0.5 meters. This specific floor design not only enhances the aesthetic appeal but also significantly aids in the observation and analysis of the robot's movement patterns, providing clear visual markers for tracking and intuitive understanding of its spatial navigation.

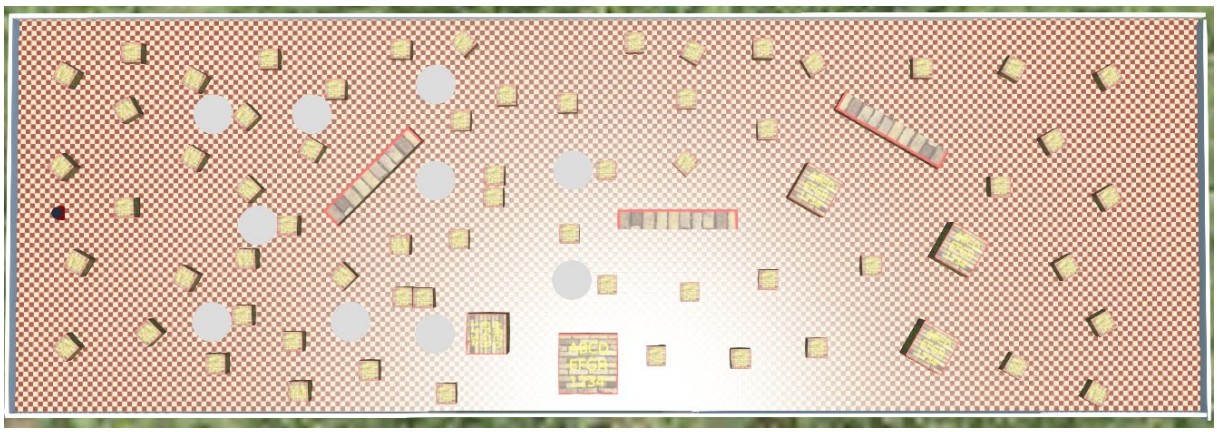


Figure 3-2. The simulated environment with plenty of different size boxes. The right-hand side is defined as the north direction. The 10 white circles represent the locations where the robot randomly restart. If the robot collides with a box, the bumper sensor will get a value then give the system an instruction to make the robot randomly start inside the ten white circles in a random position and direction.

In the simulation, ten circular zones, each with a 1-meter radius and marked in white, designate the robot's potential starting points under a triple-collision scenario. Within these circles, the robot's initial position is randomized. Surrounding these areas are taller boxes, functioning as obstacles to challenge the robot's navigation capabilities. Additionally, the experiment is bounded by four walls, effectively constraining the robot's movement to the interior space. This setup ensures a controlled environment where the robot's decision-making and mobility can be rigorously tested under various obstructive conditions.

(a) (b)

Figure 3-3. (a) The defined four walls and plenty of boxes in the simulation. (b) The 10 white circles for marking the robot random start position.

3.2.3 Simulation language and roles

In this research, the primary programming language utilized is MATLAB, a robust and versatile tool ideal for image processing and computer vision tasks. MATLAB, an acronym for 'Matrix Laboratory,' is a high-level programming language with an interactive environment, developed by MathWorks. It's extensively used across engineering, mathematics, science, and other disciplines for its capabilities in numerical computation, data analysis, algorithm development, modelling, and simulation. MATLAB's wide array of built-in functions and its adaptability make it an essential resource for a global audience of researchers, engineers, and scientists.

Originating in the late 1970s from the work of Cleve Moller, MATLAB has developed into a comprehensive software suite with an array of applications. Its unique syntax, particularly efficient for matrix and vector operations, makes it exceptionally useful for tasks in linear algebra and numerical computing.

A key advantage of MATLAB is its user-friendly graphical user interface (GUI) and its interactive development environment. This design makes MATLAB accessible even to those with limited programming expertise, enabling interactive task execution, data exploration, and result visualization.

Furthermore, MATLAB's extensive suite of toolboxes and add-ons greatly enhances its functionality, covering a broad spectrum of fields including signal processing, image processing, control system design, machine learning, and more. These toolboxes allow researchers and engineers to efficiently tackle complex challenges and streamline their workflows.

Moreover, MATLAB's compatibility with a variety of hardware devices, such as microcontrollers and data acquisition systems, broadens its applicability. It is effectively used in diverse applications, from control system prototyping to scientific experimentation across various fields[106].

Algorithm:

In this research, the algorithm is built around three core decision-making processes for robotic movement: moving forward, turning left, and turning right. Each decision is uniquely defined by the differential wheel speeds. Specifically, when moving forward, both wheels operate at a uniform speed of 1.0. For turning left, the algorithm reduces the left wheel's speed to 0.5 while maintaining the right wheel's speed at 1.0. Conversely, when turning right, the left wheel speed is set at 1.0, and the right wheel is slowed to 0.5. A critical aspect of this research is the movement duration, which is quantified as 100 program loops. This parameter ensures that the robot's movement, whether forward, left, or right, is executed for a consistent duration across all scenarios, enabling standardized assessment and comparison of the robot's navigational efficacy under varying wheel speed conditions.

In the supervised control mode, the robot, guided by human operators, avoids obstacles, necessitating data recording only before each movement. This study specifically focuses on directing the robot northward. To gather comprehensive data, we initiated movements from 10 randomly chosen starting positions. After the robot consistently moved straight north several times, reset its position to one of the previously selected starting points. Through this method, over 20 hours, a total of 51,903 data is collected.

In contrast, the unsupervised control mode entailed allowing the robot to navigate randomly. This randomness naturally led to potential collisions, requiring a structured approach to data collection:

- **Good Data:** Save data only when the robot avoided collisions for five consecutive steps (collision value = 0, indicating no collision).
- **Bad Data:** Data from a step resulting in a collision (collision value = 1) was classified as 'bad'. Post-collision, the robot was either moved back to its position five steps prior—if the collision occurred after at least five steps—or reset to a new random starting position if the collision happened sooner.

Allow the computer host to execute the simulation continuously for a duration of roughly three days. During this period, ensure the monitors remain off to optimize performance.

Over this extended simulation run, the system is expected to accumulate a substantial dataset, specifically totalling 1,850,657 data points. This approach is designed to maximize data collection efficiency while minimizing potential disruptions and energy consumption.

Table 3-3. Values in saved data. It contains a value for collision (1) or no collision (0), three values for robot position, three values for compass heading, the decision for the movement: move forward (1), turn left (2) and turn right (3). The last one is the distance values for the LiDAR data, which includes 667 range values.

Name of each parameter				
collision	values	com	decision	urg041x_values

The simulation tasks were executed using Webots, a versatile open-source, multi-platform desktop application specifically designed for robotic simulation. Webots offers a comprehensive development environment that facilitates the modelling, programming, and simulation of robots. The initial phase of the simulation involved constructing the fundamental geometric structure of the robot. Following this, we integrated a bumper sensor to enable effective collision detection. Additionally, a compass, visually represented by a needle in figure 12, was incorporated to ascertain both the robot's orientation and measurement headings. These elements played a crucial role in gathering the essential data needed for training the neural network.

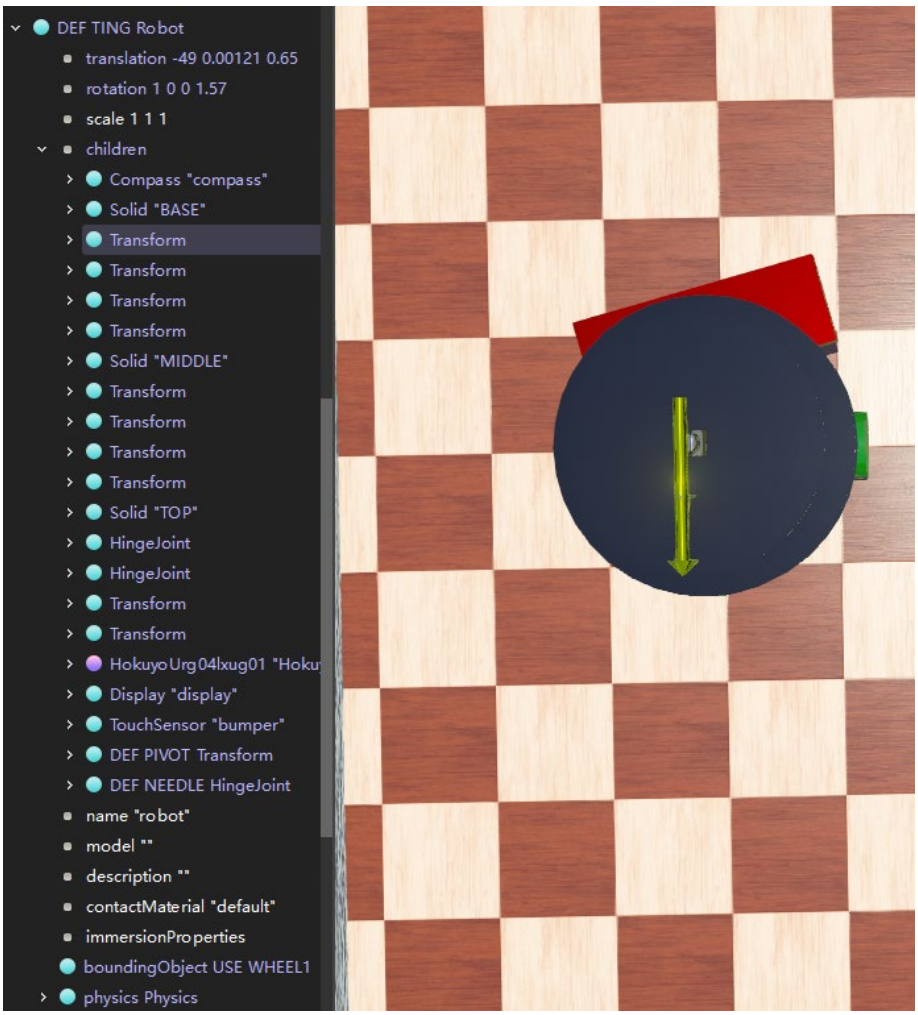


Figure 3-4. TING Robot model and the structures in the children list. The floor appearance is 'Chequered Parquetry' with different colour grids in same size. It can be used to visually observe the movement of robot.

The simulation setup encompasses a $20m \times 100m$ area, scattered with variously sized and positioned boxes. These boxes are strategically placed so that no two adjacent boxes are more than 5m apart, ensuring constant LiDAR sensor engagement. During the simulation, the needle point consistently aligns northward. Under supervisor control, the robot is programmed to return to the starting point upon reaching the northern boundary of the simulation floor.

Our simulation employs the Hokuyo URG-04LX-UG01 model in Webots, mirroring the equipment used on the actual robot. In this environment, LiDAR range values are extracted using the 'wb_lidar_get_range_image' function. These values are stored as an array of single

precision floating point numbers, each representing the range of a specific measurement point[107]. However, the LiDAR scan data, primarily distance information from each point cloud, offers limited variability for neural network training. To enhance this, we integrate the angle of each distance relative to the compass heading. By setting the northward target angle as zero, we construct a $667 \times 2 \times 1$ matrix, comprising both range data and corresponding headings, to serve as the input for the neural network training process.

3.4 Robot motion

In this study, the robot's decision-making is limited to three actions: moving forward, turning left, and turning right. This functionality is enabled by its design as a differential wheeled robot, which features two independently operated wheels located on opposite sides of its body. This configuration allows the robot to alter its course by adjusting the rotational speed of each wheel relative to the other, thereby eliminating the need for separate steering mechanisms. Typically, robots employing this drive system are equipped with one or more castor wheels. These additional wheels play a crucial role in maintaining the robot's balance and stability, preventing it from tipping over during movement[108].

3.4.1 Kinematics of differential drive robot

The navigational trajectory of a robot is intricately linked to the rotational dynamics of its two primary driven wheels. This mechanism is fundamental to its directional control. When these wheels are propelled at an identical speed and in the same direction, the robot is designed to move along a linear, straight path. This uniform motion ensures stable and straightforward navigation. Conversely, a distinct manoeuvre occurs when the wheels are rotated at the same speed but in opposite directions. In this scenario, the robot engages in a rotational movement, pivoting around its central axis point[109]. This capability allows for precise turning and positioning, essential for complex navigational tasks and manoeuvring in restricted spaces.

The figure presented below illustrates the kinematics of differential drive for a mobile wheeled robot. This system is characterized by three key generalized coordinates: the robot's planar position represented by x and y coordinates, and its angular orientation θ, which is measured relative to the X-axis[110]. The kinematic model is crucial for understanding the robot's movement mechanics, as it determines how the robot's wheel speeds translate into motion across the plane and changes in its directional heading. This understanding is fundamental for programming precise movements and navigational strategies in robotic applications.

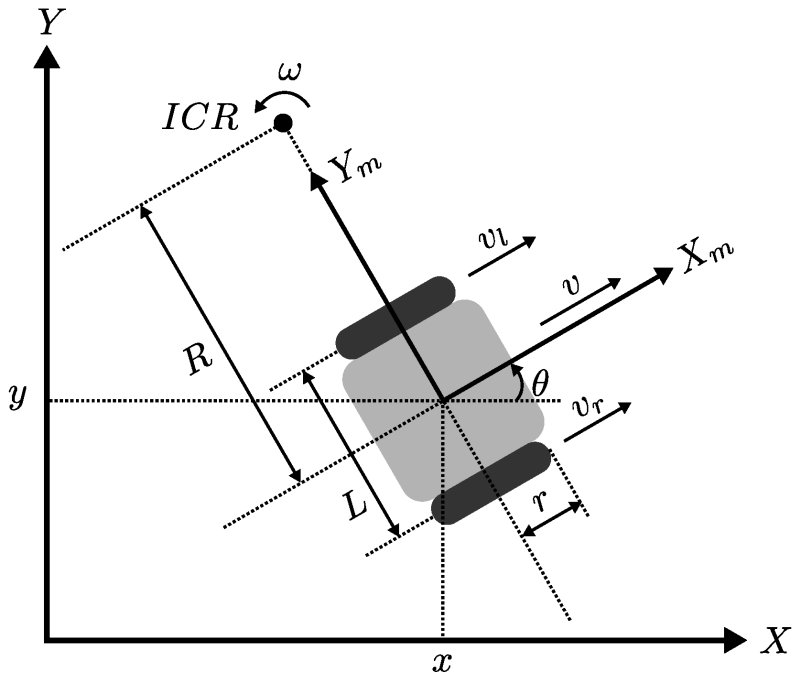


Figure 3-5. Geometry of differential drive wheeled mobile robots[110].

The design of the robot integrates wheels with a radius *r* and a chassis width *L*. During motion, the wheels rotate at specific speeds, generating a tangential velocity *v* that propels the robot forward. Simultaneously, the robot exhibits rotational motion characterized by an angular velocity *ω* and a turning radius *R*, cantered at the Instantaneous Centre of Rotation (ICR). These parameters are interrelated as follows:

$$\dot{x} = v\cos(\theta) \tag{1}$$

$$\dot{y} = v\sin(\theta) \tag{2}$$

$$\dot{\theta} = \omega \tag{3}$$

where $\dot{x} = dx/dt$ and $\dot{y} = dy/dt$ represent the components along the X and Y axes, respectively, and $\dot{\theta} = d\theta/dt$ is the angular velocity, with t denoting time[110].

From figure , the tangential velocity v can be derived using the relation[110]:

$$v = R\omega \tag{4}$$

Likewise, the ground contact speed of the left wheel v_l and the right wheel v_r can be expressed in terms of R, L and ω as[111]:

$$v_l = R_l\omega = \left(R - \frac{L}{2}\right)\omega \tag{5}$$

$$v_r = R_r\omega = \left(R - \frac{L}{2}\right)\omega \tag{6}$$

Where R_l and R_r are the rotation radius of each wheel with respect to ICR.

Solving these two equations for ω and R leads[112]:

$$\omega = \frac{v_r - v_l}{L} \tag{7}$$

$$R = \frac{L}{2}\left(\frac{v_r + v_l}{v_r - v_l}\right) \tag{8}$$

Substituting these values into Equation (4) yields:

$$v = \frac{v_r + v_l}{2} \tag{9}$$

It is clear that v and ω depend on the tangential velocities of the wheel v_l and v_r. In turn, the above velocities can be determined by the rotation speeds of the wheels ω_l and ω_r as $v_r = r\omega_r$ and $v_l = r\omega_l$[110]. The robot kinematics in local body coordination can thus be written as[39]:

$$\begin{bmatrix} \dot{X_m} \\ \dot{Y_m} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} \mathrm{v}X_m \\ \mathrm{v}Y_m \\ \omega \end{bmatrix} \triangleq \begin{bmatrix} \frac{r}{2} & \frac{r}{2} \\ 0 & 0 \\ -\frac{r}{L} & \frac{r}{L} \end{bmatrix} \begin{bmatrix} \omega_l \\ \omega_r \end{bmatrix} \tag{10}$$

Finally, applying a coordinate transformation allows to express the kinematics in global coordinates[113]:

$$\begin{bmatrix} \dot{\mathrm{x}} \\ \dot{\mathrm{y}} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 \\ \sin\theta & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} V \\ \omega \end{bmatrix} \tag{11}$$

Where V and ω are the control variables[39, 109-116].

3.5 Conclusion

This chapter detailed the design and simulation of an autonomous differential drive robot, highlighting the critical interplay between hardware integration, simulation tools, and algorithmic development. The choice of components, such as the Hokuyo URG-04LX-UG01 LiDAR and Nema 17 stepper motors, was guided by their precision, reliability, and compatibility with the system's requirements. These components not only ensured accurate sensing and movement but also allowed for seamless integration with the Arduino Mega 2560, the primary control unit.

The simulation environment, developed using Webots and MATLAB, provided a robust platform for testing and validating the robot's capabilities. Webots offered a realistic virtual setting with carefully designed obstacles and environmental elements that mimicked real-world conditions. MATLAB complemented this by enabling advanced algorithm development, data analysis, and sensor fusion. The integration of both tools resulted in a comprehensive approach to evaluating the robot's navigation and decision-making processes.

The development of simplified algorithms for the robot's movements, including forward motion, left turns, and right turns, underscored the importance of differential drive kinematics in autonomous navigation. The kinematic models demonstrated how wheel velocities influence the robot's trajectory and orientation, providing a foundation for precise control strategies.

The hardware and software selection for this project was thoroughly analysed to ensure compatibility and efficiency. The Hokuyo LiDAR, with its high scanning precision and broad field of view, proved crucial for real-time obstacle detection. The Nema 17 stepper motors, coupled with the DRV8834 motor drivers, delivered smooth and reliable motion control, which is essential for navigating cluttered and dynamic environments.

The kinematic models developed in this chapter provided a theoretical basis for understanding the robot's movement. By relating wheel velocities to the robot's global position and orientation, these models allowed for effective algorithm development and ensured the accuracy of the simulated movements.

The simulation environment successfully replicated a variety of scenarios, offering a controlled space for testing the robot's capabilities. The chequered parquetry floor design, strategically placed obstacles, and randomized starting positions allowed for a thorough evaluation of the robot’s navigation algorithms. The use of supervised and unsupervised control modes further enhanced the comprehensiveness of the data collected, which will be instrumental in refining the robot's decision-making processes.

This chapter laid the groundwork for understanding the integration of hardware components, kinematic modelling, and simulation tools in developing an autonomous differential drive robot. The selection of components was guided by a clear understanding of the project's requirements, focusing on precision, reliability, and compatibility.

The kinematic equations and their implementation in simulation provided a strong theoretical framework for controlling the robot's movements. By translating these equations into algorithms and testing them in a realistic virtual environment, this research bridged the gap between theoretical design and practical application.

The simulation results demonstrated the robot's ability to navigate through complex environments, adapt to obstacles, and execute precise movements. These findings provide a solid foundation for future enhancements, including the integration of advanced decision-making algorithms, machine learning techniques, and real-world testing.

Overall, this chapter underscores the importance of a systematic approach to the design and simulation of autonomous robots, ensuring that each component and process contributes to the goal of achieving reliable and efficient autonomous navigation.

Chapter 4. Development of neural network algorithm for static environment

4.1 Introduction

This chapter focuses on the development and implementation of a neural network algorithm designed for operation in a static environment, utilizing supervised learning. The neural network was developed using MATLAB's Deep Learning Toolbox, with data collected through simulations in Webots, a widely used robotic simulation environment. The goal of this work is to demonstrate the effectiveness of neural networks in solving tasks within a controlled, unchanging environment, where variables such as object placement and environmental layout remain fixed throughout the simulation.

A static environment presents both challenges and advantages compared to dynamic settings. While the lack of time-varying factors simplifies certain aspects of problem-solving, it requires the neural network to effectively learn patterns and achieve high performance in tasks such as object classification, navigation, or detection. The chapter outlines the process of data collection in the Webots environment, followed by the design, training, and evaluation of the neural network within MATLAB.

In the Webots simulation, the static environment includes fixed obstacles and object configurations intended to emulate real-world applications like indoor robotic navigation or object recognition. Data gathered from these simulations, which includes sensor readings and visual information, was used to train the neural network in MATLAB's Deep Learning Toolbox. The aim is to develop a model that can accurately interpret this static environment and make decisions based on the available input.

Results and analysis presented in this chapter provide insights into the performance of the neural network, including its accuracy, robustness, and any limitations when applied to a static, simulated environment. These findings offer a foundation for further research into

extending the approach to more complex, dynamic environments, which will be discussed in subsequent chapters.

4.2 Convolutional neural network design in MATLAB

The process of developing a CNN using MATLAB encompasses a multi-faceted approach. Initially, it involves meticulous data preparation, where the data is curated and formatted to be compatible with CNN requirements. Following this, the network design phase is undertaken, which is a critical step where the architecture of the CNN is conceptualized and configured. This architecture includes layers, functions, and parameters tailored to the specific needs of the research.

Subsequent to the design phase, the focus shifts to training the CNN. This stage is crucial as it involves feeding the prepared data into the network and iteratively adjusting the network's parameters to minimize errors and improve accuracy. This iterative process is guided by optimization algorithms and loss functions, enabling the network to learn from the data effectively.

Finally, evaluation is a pivotal step in assessing the performance of the CNNs. This involves testing the trained model on a set of data that it hasn't encountered during training, which helps in determining the model's generalization ability and accuracy in real-world scenarios.

Throughout these stages, MATLAB's Deep Learning Toolbox is utilized as a key resource. This toolbox provides a comprehensive suite of functions and tools that facilitate the creation, training, and evaluation of CNNs. Its user-friendly interface and advanced capabilities significantly enhance the efficiency and effectiveness of the CNN development process for this research.

4.2.1 Data preparation

Initially, the raw data needs to be transformed into image format. This conversion can be approached in two distinct ways:

1. The first method involves exclusively utilizing the distance values obtained from the LiDAR sensor. In this approach, the resultant image data should be formatted to a resolution of 667 x 1 pixels. This method focuses on the spatial information provided by the LiDAR sensor, capturing the distance measurements in a linear image format.
2. The second option integrates angular data from the compass readings with the LiDAR range values. By combining these two data types, the image data becomes more comprehensive, resulting in a resolution of 667 x 2 pixels. This dual-dimensional approach not only incorporates the spatial data from the LiDAR but also adds orientation information from the compass, offering a richer and more informative image representation.

Both methods provide unique perspectives on the data, with the first prioritizing simplicity and direct distance measurements, while the second offers a more complex view by including directional context.

4.2.2 Neural Network Architecture

Develop the CNNs architecture by employing MATLAB's 'convnet' function. This process involves defining various layers of the network. Key components include convolutional layers, which are the core building blocks, capturing the spatial features from the input. Batch normalization layers follow, which normalize the activations and gradients propagating through the network, improving the training speed and stability. Next are the ReLU (Rectified Linear Unit) layers, responsible for introducing non-linearity into the model, allowing it to learn more complex patterns. Subsequent layers include pooling layers, which reduce the spatial dimensions (height and width) of the input volume for the next convolutional layer. They are instrumental in decreasing the computational load, memory usage, and in preventing overfitting. Fully connected layers come afterwards, which connect every neuron in one layer to every neuron in the next layer, primarily used for classifying the features learned by the convolutional layers based on the training dataset. Lastly, the output layer is defined, which is crucial for the network to make predictions. Additionally, MATLAB's

Deep Network Designer, a widely used app for deep learning, offers similar functionalities for network design. An example of this in action is the construction of AlexNet, a well-known convolutional neural network.

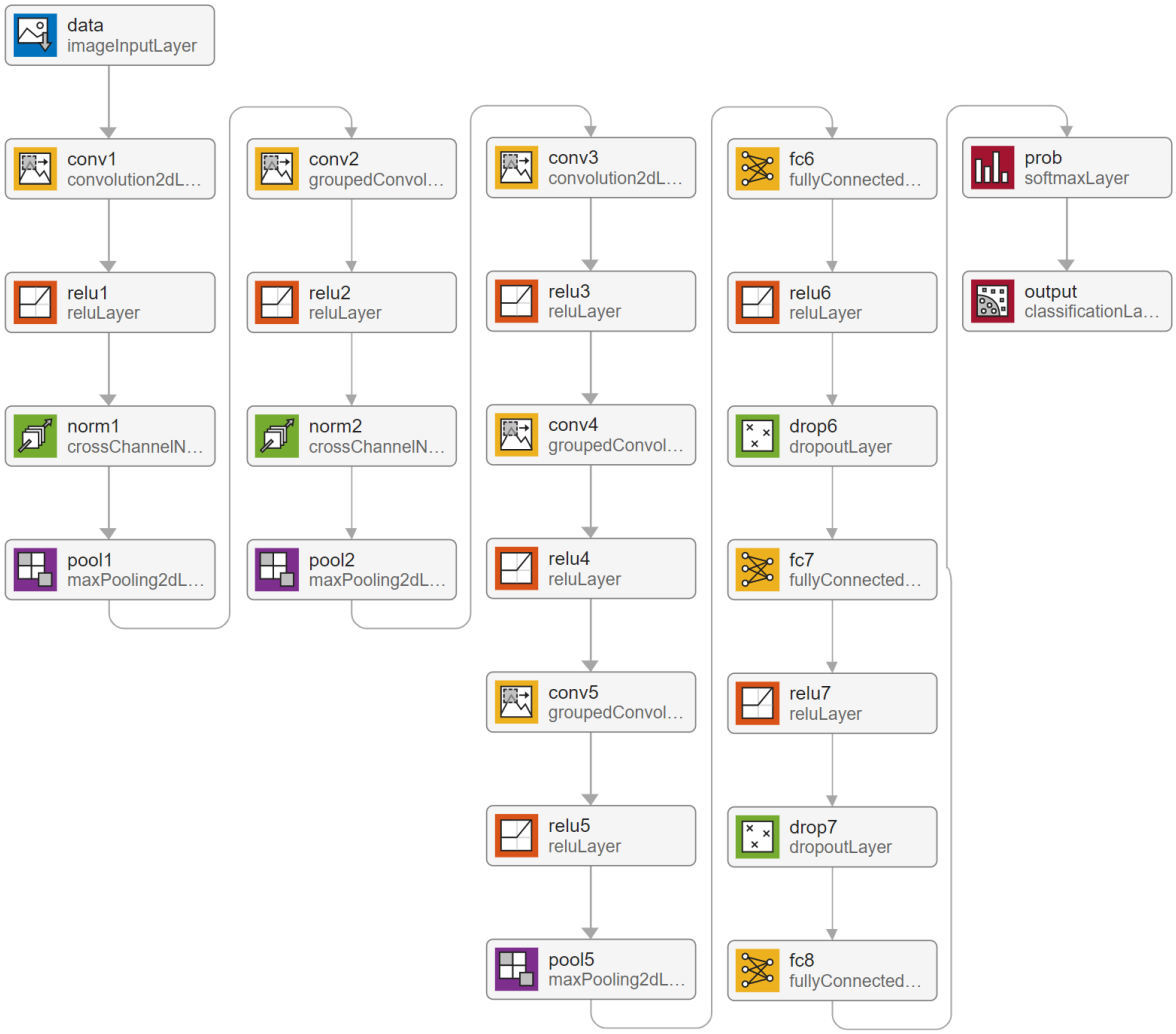


Figure 4-1. Alex Net structure in MATLAB Deep Network Designer. It is a convolutional neural network that is 8 layers deep.

The Deep Network Designer offers an expansive, zoomed-out perspective of the entire neural network[123], providing a comprehensive view of its architecture. Typically, networks like AlexNet are configured to process standard RGB images, with input dimensions of 227 x 227 x 3. However, the input data in the current research diverges significantly, characterized

by one or two elongated columns rather than conventional image formats. Addressing this unique data structure, the network design for this study has been tailored accordingly. It involves innovative adjustments to accommodate the atypical shape and size of the input data, ensuring optimal processing and analysis. This custom network architecture is meticulously crafted to effectively handle the specific characteristics of the research data, setting it apart from standard neural network designs.

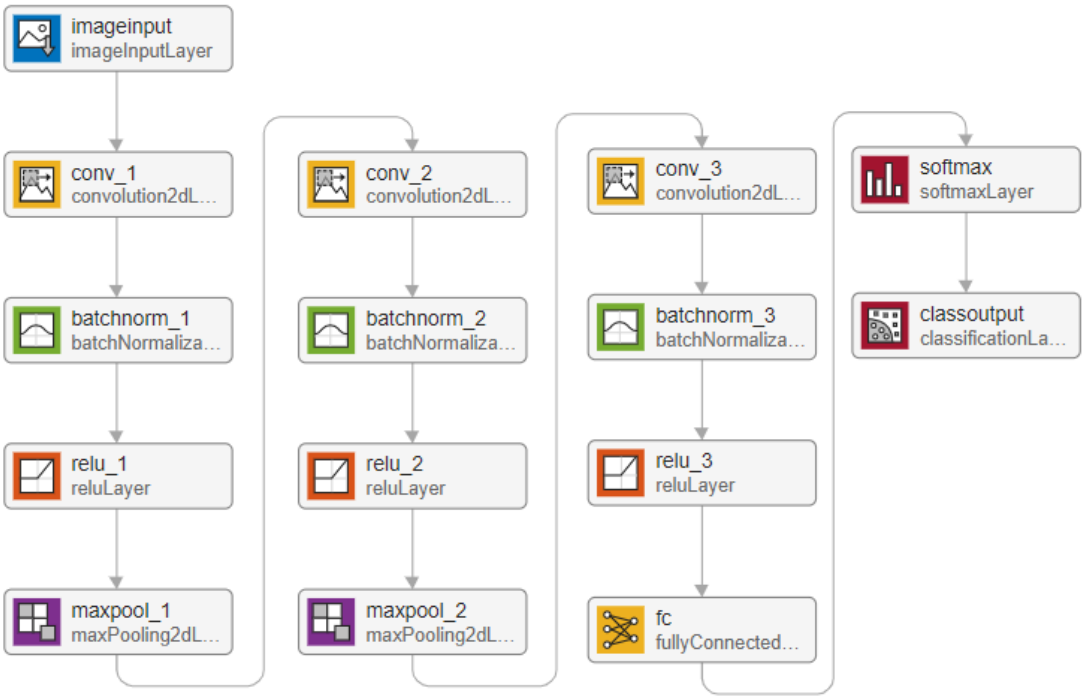


Figure 4-2. The structure of the convolutional neural network designed in this project.

The appendix A will include the MATLAB program code for the convolutional neural network structure. This code details the configuration of the network layers, specifically addressing the input layer dimensions. For the test involving only LiDAR data, the input layer is configured to accept dimensions of 1 x 67 x 1. In contrast, when the test incorporates both LiDAR data and compass readings, the input layer dimensions are adjusted to accommodate a 2 x 67 x 1 input size. This distinction is critical for ensuring accurate data processing and effective neural network performance, tailored to the specific requirements of each data set.

Table 4-1. The structure of CNN highlighting the parameters used for each layer. Oversized filters allow for modification of the input data without needing to alter the remaining layers of the CNN.

Type	Description
Image Input	imageinput: 1×67×1 images with 'zerocenter' normalization or imageinput: 2×67×1 images with 'zerocenter' normalization
2-D Convolution	conv_1: 8 3×3 convolutions with stride [1 1] and padding 'same'
Batch Normalization	batchnorm_1
ReLU	relu_1
2-D Max Pooling	maxpool_1: 2×2 max pooling with stride [2 2] and padding 'same'
2-D Convolution	conv_2: 16 3×3 convolutions with stride [1 1] and padding 'same'
Batch Normalization	batchnorm_2
ReLU	relu_2
2-D Max Pooling	maxpool_2: 2×2 max pooling with stride [2 2] and padding 'same'
2-D Convolution	conv_3 32 3×3 convolutions with stride [1 1] and padding 'same'
Batch Normalization	batchnorm_3
ReLU	relu_3
Fully Connected	fc: 3 fully connected layer
Softmax	Softmax
Classification Output	classoutput

4.2.3 Justification for layers selection

In the proposed architecture for processing LiDAR data alongside compass heading information, a variety of layers and design choices are incorporated to effectively capture spatial hierarchies and improve predictive accuracy, crucial for tasks such as object detection and navigation in autonomous vehicles or robotics. This architecture is tailored for handling 1D sequences of LiDAR data, which is structured as a 67x1 vector, encapsulating distance measurements and angular orientation from the compass.

The network begins with an 'imageInputLayer' designed to accommodate the specific shape of the input data ([1 67 1]). Although typically used for image data, using this layer for LiDAR sequences enables the utilization of CNN techniques, which are adept at capturing spatial and temporal patterns through localized receptive fields. This approach leverages the CNN's

ability to process input data in a format that, while not traditionally image-based, benefits from the spatial structuring of convolutional operations.

The choice of convolutional layers with 3x3 kernels and 'same' padding ensures that the dimensionality of the output from these layers remains unchanged relative to the input dimension. This is crucial for maintaining the integrity of spatial relationships within the data across the network. Starting with 8 filters and increasing to 16 and then 32 in subsequent layers allows the network to capture more complex features at each level, with deeper layers abstracting higher-level features from the raw input provided by the LiDAR sensors.

Each convolutional layer is followed by a ‘batchNormalizationLayer’ and a ‘reluLayer’. Batch normalization stabilizes the learning process by normalizing the output of the previous layer, reducing internal covariate shift, and allowing higher learning rates without the risk of divergence. The ReLU activation function introduces non-linearity to the learning process, enabling the network to learn complex patterns in the data. This combination is well-suited for deep networks where vanishing or exploding gradients might impede learning in the absence of such stabilizing mechanisms.

‘maxPooling2dLayer’ is used after selecting convolutional and activation layers, with a 2x2 pool size and 'same' padding to reduce the spatial dimensions of the feature maps gradually. This down sampling technique helps in reducing the computational complexity of the network, while also making the network invariant to small translations in the input data.

The network transitions from convolutional layers to a ‘fullyConnectedLayer’ with three outputs, corresponding to the categorical targets expected from the system. The use of a fully connected layer here is to collate the high-level features learned by the convolutional layers and map them to the output space. This is followed by a ‘softmaxLayer’ which outputs probabilities for each class, making the model's predictions interpretable as confidence levels across the possible categories. Finally, a ‘classificationLayer’ computes the loss for a multi-class classification problem, facilitating the training process by providing a differential measure of the error in predictions.

The exclusion of dropout layers in the final architecture (despite being commented out in the provided code) may be driven by the need to maintain all learned features due to the relatively small size of the input data and the model's capacity. Dropout is typically used to prevent overfitting by randomly dropping units during training, which might not be necessary or could lead to underfitting in scenarios where the complexity of the model is well-balanced with the complexity of the task and the amount of data available.

While not specified in the question, the number of epochs, learning rate, and other hyperparameters in training a neural network of this design should be chosen based on empirical performance metrics such as loss and accuracy on a validation set, alongside considerations of computational efficiency and convergence behavior observed during training.

The chosen learning rate of 0.0001 is relatively low, which is particularly advantageous for training deep neural networks on complex datasets where a finer control over the model adjustments is necessary. A lower learning rate ensures that the model updates its weights gradually, avoiding drastic changes that could lead to divergence in the loss function. This is crucial when using a model structure with multiple layers and operations such as batch normalization and ReLU activations, as it helps in achieving a stable convergence.

For the specific context of LiDAR data, where the input features can have subtle yet critical variations important for tasks like object recognition or navigation, a lower learning rate allows the model to iteratively learn these nuances without skipping over them. This meticulous approach can lead to better generalization on unseen data, a critical factor in autonomous systems where predictive accuracy and reliability are paramount.

Setting the epoch count to 20 strikes a balance between adequate training time and computational efficiency. Training for too few epochs might leave the model underfitted, wherein it has not learned enough about the data features to make accurate predictions. Conversely, too many epochs could lead to overfitting, especially if the model starts learning not only the legitimate data characteristics but also the noise and anomalies in the training data.

For deep learning models dealing with spatial data such as those from LiDAR, each epoch represents an opportunity for the model to pass over the entire dataset, refining its understanding and adjusting its weights accordingly. The decision to limit the epochs to 20 likely stems from empirical observations suggesting that further epochs do not significantly improve performance on a validation set. This could be indicative of the model reaching its learning capacity under the current configuration, or it might be a preventive measure against overfitting, particularly in scenarios where additional data augmentation or regularization techniques (like dropout) are not employed extensively.

The combination of a low learning rate and a moderate number of training epochs is reflective of a cautious approach aimed at achieving model robustness and reliability. This training strategy is suited to applications where the cost of an incorrect prediction is high, and therefore, precision in the learning process is prioritized over speed. It is also indicative of an environment where the available computational resources are balanced against the need for timely deployment, making it a practical choice in many professional settings.

In summary, the chosen learning rate and epoch count for this neural network architecture are aligned with the goals of achieving high accuracy and reliability in processing LiDAR data, supporting the overarching objectives of safety and efficiency in autonomous navigation and related applications.

4.3 Training the Neural Network with Supervised learning

Supervised learning in robot navigation involves training a robot to predict its actions or make decisions based on labelled data, typically provided by human operators or sensors. This process is akin to teaching a robot to interpret its surroundings, understand the environment, and navigate through it by leveraging a dataset of historical examples. These examples serve as a guide for the robot to generalize its understanding of different situations it may encounter[28].

The utilization of supervised learning in robot navigation has witnessed significant advancements in recent years. The integration of deep learning techniques, particularly CNNs for perception tasks and RNNs for decision-making, has substantially improved the navigation capabilities of robots. These advancements have been pivotal in applications such as autonomous vehicles, drones, and mobile robots, making them more adaptive and responsive in various settings[56].

4.3.1 Data collection

For the supervised learning in this project, the robot is driven manually to avoid the obstacles. There are only three decisions in this research: move forward, turn left, and turn right. In order to get more dataset for different directions, reset the robot in random position inside 10 circles after moving towards north for several times. In this case the robot should learn how to navigate to north while avoiding obstacles. Each move data is saved in MATLAB file with robot position, compass heading, and LiDAR sensor data. However, human control of robot movement is quite boring and time-consuming. The total number of data is only 51,903 data after 22 hours driving the robot. A normal neural network training requires large amounts of data, and we propose 2 strategies to get more data based on the raw data.

- Random value: in the random one, any point that is closer than 200 gets a random number of 0 to 9 added to it (this loop 10 times to create the additional files). This is effectively adding noise to the dataset (always away from the robot).
- Systematic: for the systematic one, the objects that are less than 200 are moved back from 0 displacement up to 9 displacements thereby creating the 10 extra files (essentially you are pushing each object back slightly and using that as a new data set).

The two methods code will be attached at appendix. By using these two methods, the number of the data can get up to 5,190,300 data set.

4.3.2 Data pre-processing

The data set is divided into 12 parts: F1~F12 folders based on the decision and the compass heading, shown in Table 4-2.

Table 4-2. 12 folders for each direction with different decision.

Folder name	**F1**	**F2**	**F3**	**F4**	**F5**	**F6**	**F7**	**F8**	**F9**	**F10**	**F11**	**F12**
Decision-compass	1N	1E	1S	1W	2N	2E	2S	2W	3N	3E	3S	3W

This thesis proposes a method to change the percentage of directional data for network training to test the navigation capabilities. It is clearly from the figure 4-3 below, in addition to F1, (F6, F7, F11 and F12) can make the robot move towards the north. Therefore, the training results are compared by adjusting their percentage in the training data.

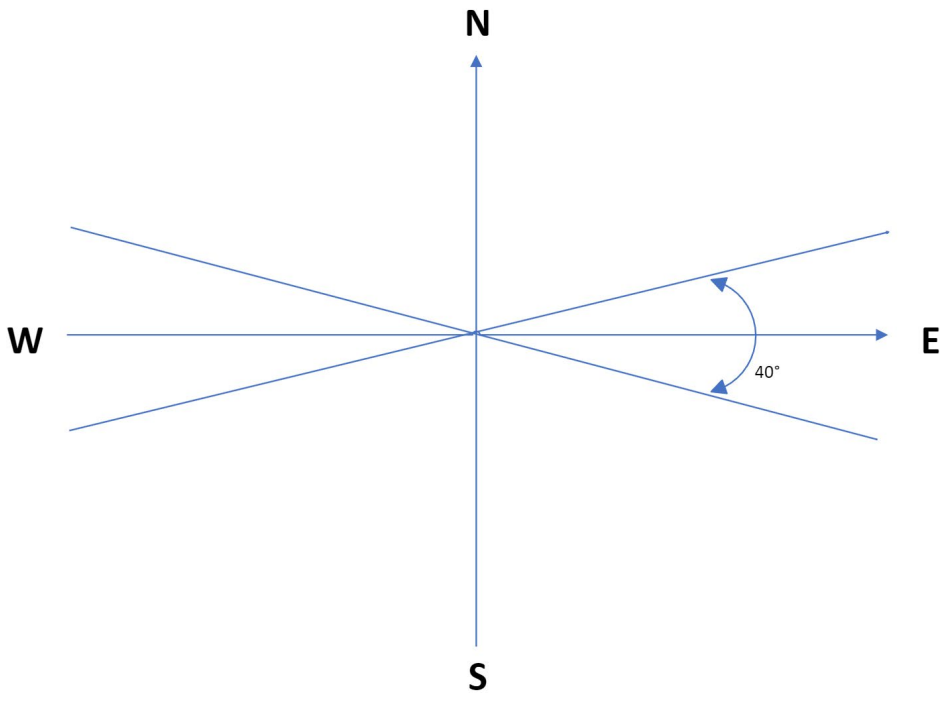


Figure 4-3. The local coordinates of the robot are divided into four parts (NWSE). Multiplied by three motion groups (decision 1,2,3), there are a total of 12 groups of datasets.

In the first step, we use 0%,20%,50% and 80% for the percentage of directional data to use in the formula below. Take two-thirds of the data in the south direction and the rest are in

the east (decision 2 turn left) and west (decision 3 turn right). The input data of the three groups are all 20,000 in the end. The 4 groups are shown in Table 4-3, Table 4-4, Table 4-5 and Table 4-6 below.

Table 4-3. 0% of directional data to use data selection. They will subject neural networks to different training. The total training data is 20,000 for each decision group.

*Percentage of directional data to use: **0**%*

Folder Name	Decision-compass	Specific data	Random data	Total	Subtotals
F1	1N	0	5000	5000	
F2	1E	0	5000	5000	
F3	1S	0	5000	5000	
F4	1W	0	5000	5000	20000
F5	2N	0	5000	5000	
F6	2E	0	5000	5000	
F7	2S	0	5000	5000	
F8	2W	0	5000	5000	20000
F9	3N	0	5000	5000	
F10	3E	0	5000	5000	
F11	3S	0	5000	5000	
F12	3W	0	5000	5000	20000

Table 4-4. 20% of directional data to use data selection. They will subject neural networks to different training. The total training data is 20,000 for each decision group.

*Percentage of directional data to use: **20**%*

Folder Name	Decision-compass	Specific data	Random data	Total	Subtotals
F1	1N	4000	4000	8000	
F2	1E	0	4000	4000	
F3	1S	0	4000	4000	
F4	1W	0	4000	4000	20000
F5	2N	0	4000	4000	
F6	2E	2667	4000	6667	
F7	2S	1333	4000	5333	
F8	2W	0	4000	4000	20000
F9	3N	0	4000	4000	
F10	3E	0	4000	4000	
F11	3S	1333	4000	5333	
F12	3W	2667	4000	6667	20000

Table 4-5. 50% of directional data to use data selection. They will subject neural networks to different training. The total training data is 20,000 for each decision group.

*Percentage of directional data to use: **50**%*

Folder Name	Decision-compass	Specific data	Random data	Total	Subtotals
F1	1N	10000	2500	12500	
F2	1E	0	2500	2500	
F3	1S	0	2500	2500	
F4	1W	0	2500	2500	20000
F5	2N	0	2500	2500	
F6	2E	6667	2500	9167	
F7	2S	3333	2500	5833	
F8	2W	0	2500	2500	20000
F9	3N	0	2500	2500	
F10	3E	0	2500	2500	
F11	3S	3333	2500	5833	
F12	3W	6667	2500	9167	20000

Table 4-6. 80% of directional data to use data selection. They will subject neural networks to different training. The total training data is 20,000 for each decision group.

*Percentage of directional data to use: **80**%*

Folder Name	Decision-compass	Specific data	Random data	Total	Subtotals
F1	1N	16000	1000	17000	
F2	1E	0	1000	1000	
F3	1S	0	1000	1000	
F4	1W	0	1000	1000	20000
F5	2N	0	1000	1000	
F6	2E	10667	1000	11667	
F7	2S	5333	1000	6333	
F8	2W	0	1000	1000	20000
F9	3N	0	1000	1000	
F10	3E	0	1000	1000	
F11	3S	5333	1000	6333	
F12	3W	10667	1000	11667	20000

4.3.3 Training the CNN

The input layer size is [2 67 1] this part and the network structure is mentioned in previous chapter. We use 0.0001 initial learn rate and 20 for the max epochs. Just show the training result of 0% one as an example.

Table 4-7. The structure of CNN highlighting the parameters used for each layer. Oversized filters allow for modification of the input data without needing to alter the remaining layers of the CNN.

Type	Description
Image Input	**imageinput: 1×67×1 images with 'zerocenter' normalization** **or** **imageinput: 2×67×1 images with 'zerocenter' normalization**
2-D Convolution	**conv_1: 8 3×3 convolutions with stride [1 1] and padding 'same'**
Batch Normalization	**batchnorm_1**
ReLU	**relu_1**
2-D Max Pooling	**maxpool_1: 2×2 max pooling with stride [2 2] and padding 'same'**
2-D Convolution	**conv_2: 16 3×3 convolutions with stride [1 1] and padding 'same'**
Batch Normalization	**batchnorm_2**
ReLU	**relu_2**
2-D Max Pooling	**maxpool_2: 2×2 max pooling with stride [2 2] and padding 'same'**
2-D Convolution	**conv_3 32 3×3 convolutions with stride [1 1] and padding 'same'**
Batch Normalization	**batchnorm_3**
ReLU	**relu_3**
Fully Connected	**fc: 3 fully connected layer**

Softmax	**Softmax**
Classification Output	**classoutput**

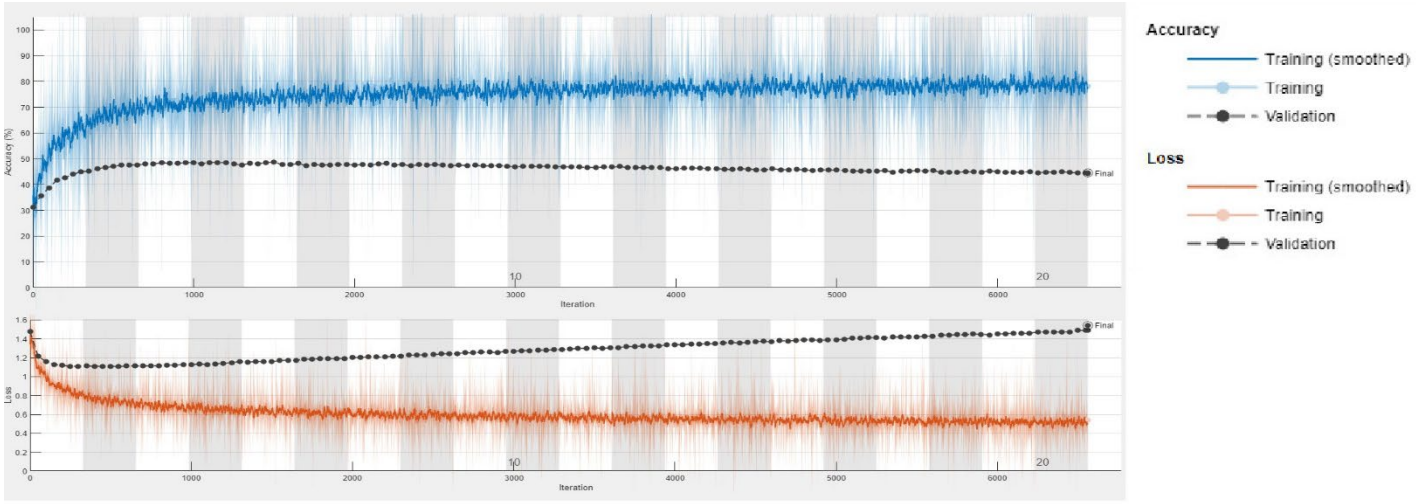


Figure 4-4. The proposed convolutional neural network training process and result.

The validation accuracy remains around 50% after 500 iterations and the final accuracy is 44.54%. 0% means that we don't specifically choose the data we want, and it should have the similar results with the lidar data only for the input image (which will be tested later). The other groups results are shown in the table below.

Table 4-8. The accuracy of different training group. The number is the percentage of the directional data for use while considering heading north.

Group	0%	20%	50%	80%
Accuracy	44.54%	46.46%	45.55%	67.23%

4.3.4 Testing and deployment

The next step is testing the model on new data that the robot hasn't seen before to assess its real-world performance. The trained network results are export as a SeriesNetwork MATLAB data. Then use the net to decide with the Lidar sensor data based on the previous code. Since the input data size is reduced for the network training, we need to resize the LiDAR scan data size as well. In the test part, we save both the good and bad data to calculate the obstacle avoidance rate and the northward navigation capability. The test simulation takes more than 10 hours each case and the robot start at the bottom of the environment. It only used the trained network for navigation. The results are shown in the figure below.

Table 4-9. The test data for the supervised learning with both LiDAR and Compass reading with different percentage of directional data to use (0%,20%,50%,80%).

Percentage	Percentage of north travel	No. of good	No. of bad	B/(B+G)	B/Distance
0%	**83.02%**	223197	1708	**0.76%**	**1.87%**
20%	**86.12%**	205803	1861	**0.90%**	**2.18%**
50%	**87.71%**	286035	6065	**2.08%**	**5.10%**
80%	**86.11%**	203988	10118	**4.73%**	**12.09%**

The robot demonstrates superior obstacle avoidance capabilities, particularly notable in scenarios with 0% data selection. Enhanced by manual-driven supervised learning, the robot is adept at navigating northward. This ability is directly proportional to the volume of north-oriented data employed in its training. However, there is a noticeable trade-off: as the training data becomes increasingly specific in its directional focus, the robot's proficiency in avoiding obstacles diminishes correspondingly.

4.3.5 Debugging and Fine-Tuning

For the future work of supervised learning, more percentage of directional data to use will be tested (aim to finish each 10%). When the robot encounters an obstacle, it is almost able to avoid it. Therefore, during supervised learning, we appropriately adjust the distance to avoid obstacles so that the car can make a turn earlier.

All in all, robots under supervised learning have excellent obstacle avoidance and navigation capability. However, in reality, many situations are in unknown environments, and it is impossible to train robots through supervised learning. Therefore, we need to introduce unsupervised learning.

4.4 Training the Neural Network with Unsupervised learning

Unsupervised learning is a subfield of machine learning that has gained significant prominence in recent years due to its potential to impart a degree of self-sufficiency to robots. Unlike supervised learning, where a robot is trained using labelled data, unsupervised learning allows a robot to explore and understand its environment by discerning inherent patterns and relationships in the data it gathers, without the need for explicit human-provided labels. This not only reduces the dependency on extensive, manually labelled datasets but also equips robots with adaptability and versatility crucial for navigating in previously unseen and unstructured environments[125].

The scope of applications for unsupervised learning in robot navigation is diverse and encompasses areas such as autonomous vehicles, warehouse automation, agricultural robotics, and even space exploration. For example, self-driving cars employ unsupervised learning to extract useful information from sensor data, recognize and respond to traffic conditions, and make real-time decisions. Similarly, autonomous drones and agricultural robots utilize unsupervised learning to map fields, detect crop health, and optimize farming operations.

4.4.1 Data collection

The data collection for unsupervised learning is totally random. Give the robot random decision then move. The rule here is shown in the diagram below. If five consecutive decisions are correct (no collisions), the first data is saved as the correct data. But if a collision occurs and the robot has moved more than five steps, the supervised function in the Webots will be used to flash the robot back to their it was five steps ago. Then randomly choose one of the remaining two decisions to move on and start a new count with that (called decision_once here). If the robot still collides later and decision_once value is less than 5, it will return to the first position and use the reaming last decision. If is still fails, the robot will restart at one of random starting points.

Table 4-10. The algorithm for unsupervised learning in static environment.

Algorithm: Unsupervised learning in static environment	
1	Initialize the value: start = 1, result =0, movement_time = 0, decision= 0, decision_once = 1;
2	If movement_time = 0 & result =0 Reda the robot position, compass reading and LiDAR scan and save all these data into Data_1; Give random decision and set the movement_time = 100; End
3	If touch_sensor_value(bumper) > 0 Reset the robot; End
4	If touch_sensor_value(bumper) = 0 & result = 1 If movement_time > 0

```
    movement_time – movement_time =1;
  End
  If movement_time = 0
    If decision_once >= 5
      decision = decision + 1;
      Save Data_5;
    Else
       decision_once = decision_once + 1;
    End
  Data_5 = Data_4;
  Data_4 = Data_3;
  Data_3 = Data_2;
  Data_2 = Data_1;
  Data_1 = [];
  End
End
```

Since no human intervention is required, the program can keep running from morning to night. After running for several days and nights, 1,391,118 data were obtained. The next step is transferring these data into the input data for network training.

4.4.2 Data pre-processing

The data pre-processing part for unsupervised learning is similar to the supervised learning. The main difference is that there is enough data to not require additional processing, and the bad data set can be used to do an additional network training for robot navigation. In this part, these groups of data will be tested:

- Lidar data only with both good and bad data
- Lidar data and compass heading good and bad data
- Lidar data and compass heading good data only with different percentage of directional data to use (0%, 20%,50%,80%)

Table 4-11. The data collected for each group during unsupervised learning.

FOLDER NAME	DECISION- COMPASS	DATA SET
F1	1N	46,904
F2	1E	192,404
F3	1S	62,510
F4	1W	152,342
F5	2N	93,691
F6	2E	385,369
F7	2S	130,449
F8	2W	315,484
F9	3N	49,419
F10	3E	202,292
F11	3S	64,872
F12	3W	154,921

4.4.3 Training the CNN

For the unsupervised learning, the same amount of input training data with supervised learning is used (20,000). Figure 4.4.3 is the training result for both lidar and compass data with 0% of directional data to use. It is clear that the accuracy of unsupervised one is quite smaller than supervised learning.

The network for lidar data only is a little different from previous one. The structure of the network is shown in Figure 4-5 below.

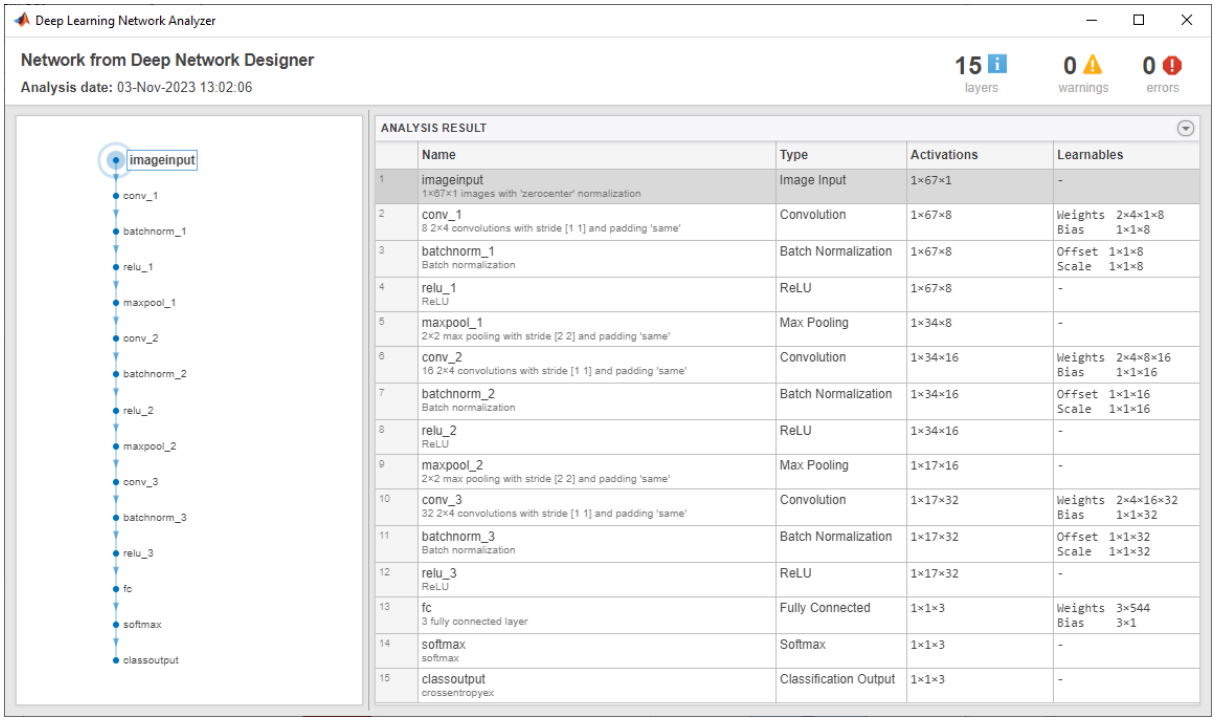


Figure 4-5. The analysis result for the proposed neural network for lidar data only.

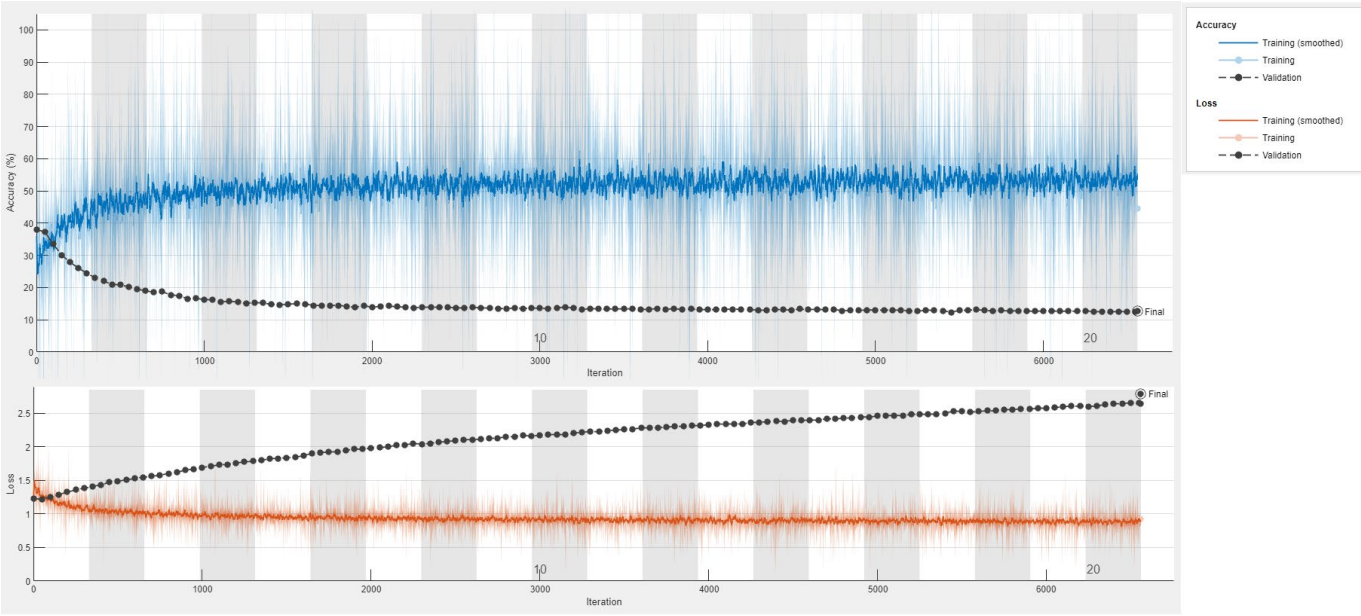


Figure 4-6. The group (0%) network training process and result.

However, when it comes to the lidar data only training, the result is shown in Figure 4-7 below. It has the similar accuracy with supervised learning.

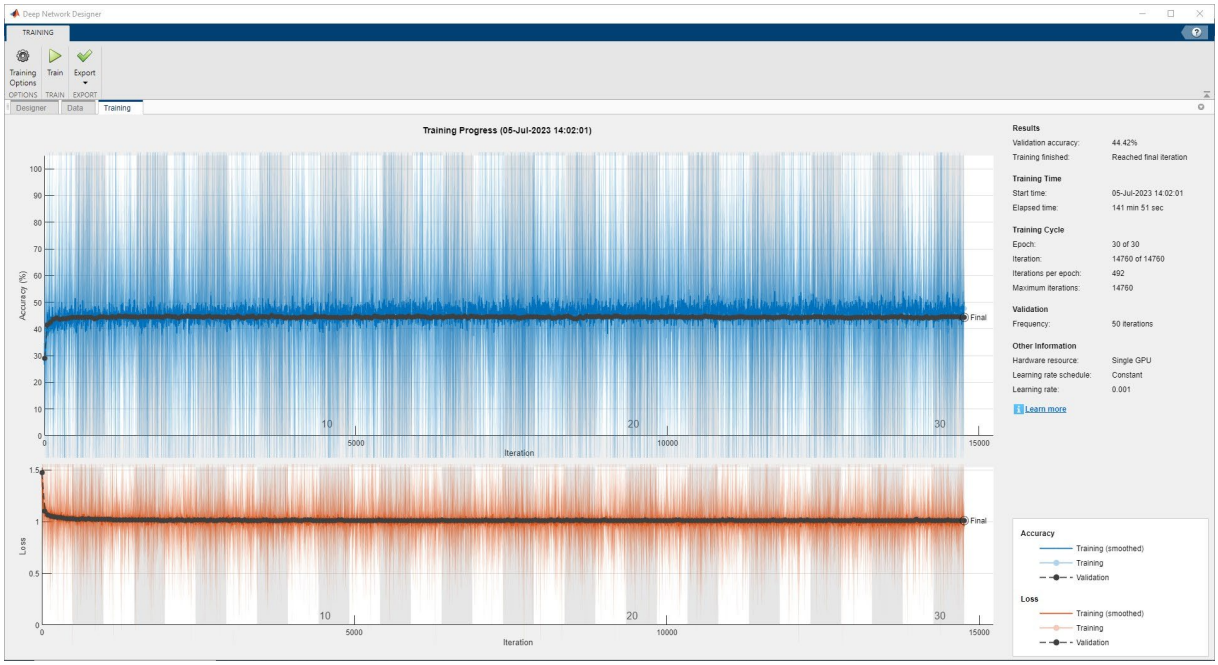


Figure 4-7. The group (LiDAR data only) network training process and result.

Table 4-12. The accuracy of each group (LiDAR data only and 0%,20%,50%,80% of the directional data).

	LIDAR DADA ONLY	LIDAR DATA & COMPASS HEADING			
GROUP	-	0%	20%	50%	80%
ACCURACY	44.42%	12.69%	17.49%	24.59%	49.05%

With more use of towards north's data, the network training accuracy is getting bigger. This shows that when using more data that turns north, the difference between the data becomes smaller. It is funny that if we use only lidar data, the difference between each data is smaller than combining LiDAR data and compass heading.

4.4.4 Testing and development

Based on the previous different group data set, there are several comparison tests in this part.

- LiDAR data and compass heading good data only with different percentage of directional data to use (0%, 20%,50%,80%)

Table 4-13. The original data for LiDAR data and compass heading good data only with different percentage of directional data to use (0%,20%,50%,80%).

Percentage	Percentage of north travel	No. of good	No. of bad	B/(B+G)	B/Distance
0%	**53.61%**	212668	10192	**4.57%**	**12.73%**
20%	**74.56%**	37763	2219	**5.55%**	**14.55%**
50%	**75.67%**	61048	5271	**7.95%**	**21.66%**
80%	**76.73%**	59352	6407	**9.74%**	**26.69%**

- LiDAR data only with both good and bad data

Table 4-14. The original data for LiDAR data only with both good and bad data.

Percentage	Percentage of north travel	No. of good	No. of bad	B/(B+G)	B/Distance
-	**42.18%**	479209	22004	**4.39%**	**11.44%**

- LiDAR data and compass heading good and bad data

Table 4-15. The original data for LiDAR data and compass heading good and bad data.

Percentage	Percentage of north travel	No. of good	No. of bad	B/(B+G)	B/Distance
-	**53.93%**	212485	9760	**4.39%**	**11.78%**

Arrange the data in a table below, focusing only on obstacle avoidance and northward navigation. Define groups with ABC and add 'percentage of directional data to use' in the Lidar & compass good data only sections.

Table 4-16. The north travel and collision rate results for each group test.

	Data type	Percentage of directional data to use	% of north travel	Collision Rate (B/(B+G))
A	LiDAR only good and bad data	-	42.18%	4.39%
B	LiDAR & compass good and bad data	0%	53.93%	4.39%
C	LiDAR & compass good data only	0%	53.61%	4.57%

D	LiDAR & compass good data only	20%	74.56%	5.55%
E	LiDAR & compass good data only	50%	75.67%	7.95%
F	LiDAR & compass good data only	80%	76.73%	9.74%

The group A is for lidar dada only test with both good and bad data as the input. The group B is a comparison test contain lidar data and compass heading with both good and bad data. Then group C to F use the same data from group B, excluding bad data. Each percentage of directional data are tested in these groups. With higher amount of directional data used for training, the robot obstacle avoidance ability has become worse.

4.4.5 Debugging and Fine-Tuning

From the data in the previous section, it is obvious that the obstacles avoidance abilities of these groups are very similar. They have less than 10% collision rate and good northward navigation capabilities.

The future plan here is trying to increase the steps from 5 to 10 in the data selection part. If the robot is so close to the obstacle, it cannot avoid the box with any decision. The robot should make the correct decision when seeing an object from a certain distance to avoid it.

The robot always starts at the bottom (south) and the environment is a rectangle. Most of the correct data obtained by the robot is moving towards the north duo to obstacle avoidance.

4.5 Conclusion in static neural network

This chapter delved into the design, development, and evaluation of neural networks applied to static environments, with a focus on autonomous robotic navigation. Through the use of MATLAB's Deep Learning Toolbox and Webots simulation environment, a neural network architecture was constructed, trained, and tested to assess its effectiveness in handling tasks such as navigation and obstacle avoidance. The study specifically concentrated on the application of CNNs in supervised and unsupervised learning settings, comparing their performance in terms of accuracy, robustness, and adaptability.

The supervised learning phase focused on the challenge of training the robot to navigate northward while avoiding obstacles. By manually driving the robot and collecting sensor data, including LiDAR and compass readings, a comprehensive dataset was formed. Two data augmentation techniques—random and systematic—were employed to overcome the limited quantity of collected data, increasing the dataset from approximately 50,000 to over 5 million data points. The neural network architecture, carefully designed to process the non-standard input shape of 1D LiDAR data, demonstrated varying degrees of success depending on the percentage of northward-oriented data used in training. The network achieved its highest accuracy of 67.23% when 80% of the training data was biased towards northward navigation, albeit with a trade-off in obstacle avoidance performance.

Key insights emerged during the supervised learning experiments, including the realization that training the neural network with a higher percentage of directional data improved its navigation ability in that direction but reduced its general obstacle avoidance capability. This finding underscored the importance of balancing directional bias with general navigational tasks, particularly when training neural networks for real-world applications like autonomous navigation. Furthermore, the analysis of training parameters such as learning rate and epoch count highlighted the benefits of cautious adjustments in neural network training, ensuring model stability and preventing overfitting.

In the unsupervised learning phase, the robot operated without human guidance, navigating randomly within the static environment, and autonomously collecting data. This approach yielded over 1.3 million data points, providing a robust dataset for training the neural

network without manual intervention. However, the results of unsupervised learning were less promising than those obtained from supervised learning, with lower overall accuracy and a noticeable drop in performance when compass data was combined with LiDAR data. This suggested that the complexity added by compass data was not beneficial in an unsupervised learning context where the network struggled to generalize from the unlabelled data.

Comparative testing between supervised and unsupervised learning further illuminated the differences in performance. Supervised learning, particularly when combined with a structured data augmentation strategy, demonstrated superior accuracy and obstacle avoidance capabilities. In contrast, unsupervised learning showed promise in terms of scalability and autonomy but required further refinement to achieve comparable performance levels. Additionally, the unsupervised learning experiments highlighted the potential for using large, unstructured datasets for robot training in real-world environments, where labelled data is often scarce or unavailable.

It concluded with an acknowledgment of the limitations observed during testing, particularly in terms of the balance between task-specific accuracy and obstacle avoidance. These findings suggest that while the current neural network architecture is effective in static environments, improvements can be made by incorporating more sophisticated strategies such as fine-tuning the percentage of directional data used during training or enhancing the network’s capacity to adapt to new environments.

Future work will explore extending this approach to dynamic environments, where variables such as object placement and environmental conditions change over time. This will involve the integration of unsupervised learning techniques that enable the robot to adapt to previously unseen environments without relying on pre-labelled datasets. Additionally, fine-tuning methods such as increasing the training steps or adjusting the obstacle avoidance distance are expected to further enhance the robot's decision-making and navigation capabilities.

In summary, this chapter laid a strong foundation for using CNNs in static robotic navigation tasks, showcasing the effectiveness of supervised learning while identifying areas where

unsupervised learning could be improved. The findings provide valuable insights into the challenges and opportunities of applying deep learning techniques in robotic navigation, particularly in terms of how data preparation, network architecture, and training strategies impact performance in both controlled and autonomous settings.

Chapter 5. Development of neural network algorithm for dynamic environment

In the realm of robotics and artificial intelligence, the navigation of autonomous robots within dynamic environments presents an intricate and ever-evolving challenge. The dynamism of the environment, characterized by moving objects and unpredictable obstacles, demands innovative solutions that extend beyond traditional approaches. At the intersection of computer vision and robotics, CNNs have emerged as a transformative technology, revolutionizing the way robots perceive and navigate dynamic surroundings.

Traditionally, robot navigation has been predominantly designed for static environments. Approaches such as SLAM and path planning algorithms have demonstrated their efficacy in controlled settings with stationary obstacles. However, as robots move into real-world, dynamic scenarios – ranging from busy city streets to warehouses filled with automated machinery – the inadequacies of these traditional methodologies become evident. Dynamic environments pose unique challenges that demand dynamic solutions, as robots must not only navigate but also anticipate and adapt to a constantly changing landscape.

This growing need for adaptability and real-time decision-making in dynamic environments has given rise to dynamic CNNs as a promising technology. Dynamic CNNs, building upon the foundation of their static counterparts, have demonstrated their potential in addressing the multifaceted requirements of robot navigation. These networks are equipped to handle the complexities of moving objects, delivering real-time object detection, scene understanding, and improved obstacle avoidance capabilities. As a result, dynamic CNNs have the potential to significantly enhance the autonomy and safety of robots navigating through environments marked by perpetual transformation.

One of the significant advantages of dynamic CNNs lies in their capacity to discern and track objects in real-time, allowing robots to respond swiftly to changes in their surroundings. For instance, the YOLO (You Only Look Once) family of real-time object detection systems, exemplified by YOLOv4 [58], has set new benchmarks in terms of accuracy and processing

speed. Such advancements are pivotal for enabling robots to identify and respond to dynamic objects, whether they be pedestrians crossing the street, vehicles merging into traffic, or packages moving on a conveyor belt.

As this field advances, dynamic CNNs are not limited to mere object detection but extend to scene understanding as well. Spatio-Temporal Neural Networks (STNN) and Dynamic Scene Graphs (DSG)[126] are innovative models that embrace the temporal dynamics of the environment. These models enable robots to predict the future state of dynamic objects, providing a foundation for improved path planning and collision avoidance.

In addition to addressing the inherent challenges of dynamic environments, dynamic CNNs can be further customized through transfer learning and fine-tuning. By pre-training these networks on general object detection datasets and then fine-tuning them for specific environments, robots can adapt to the unique characteristics of their operational surroundings.

Furthermore, the integration of sensor fusion, involving data from various sensors like LIDAR, cameras, and radar, plays a pivotal role in enhancing the capabilities of dynamic CNN-based navigation systems. Sensor fusion offers a comprehensive and accurate perception of the environment, vital for informed decision-making in dynamic scenarios. Research in this area, such as "Multi-Sensor Fusion for Robot Perception" [3, 127], further underscores the potential of dynamic CNNs when combined with multi-modal sensor data.

the advent of dynamic CNNs has the potential to redefine the landscape of robot navigation in dynamic environments. These networks empower robots with the ability to perceive, understand, and adapt to constantly changing surroundings. As the field continues to advance, it is important to acknowledge the remaining challenges, such as real-time processing, robustness in adverse conditions, and the ability to navigate complex, cluttered environments.

5.1 First try in dynamic environment

The neural network-based navigation in dynamic environment is quite complex. The similar method of navigation in static environment is implemented at the first time.

5.1.1 Data collection in dynamic environment

In order to create the dynamic environment, several moving robots are added to the original environment. The moving robots are equipped with sensor. Once it detects an obstacle in front, it will stop and rotate at a certain angle to ensure that there are no obstacles in front before moving forward. The moving robots speed is smaller than the 'Ting' robot. Cause the original environment is quite large for dynamic research, a small, closed environment is created for testing dynamic CNN. The specific environment is shown in Figure 5-1, Three moving robots and three boxes are symmetrically distributed in the center of the environment. The robot will start at random positions then do the random motion to collect data. The data saving guidelines here are the same as the static one. The only difference is that when encountering an obstacle, both the 'Ting' robot and the moving robots will flash back to their previous positions. Each time the robot collects data, it will stop moving and wait for the LiDAR sensor to scan five times. The five consecutive scan data are saved in one data file.

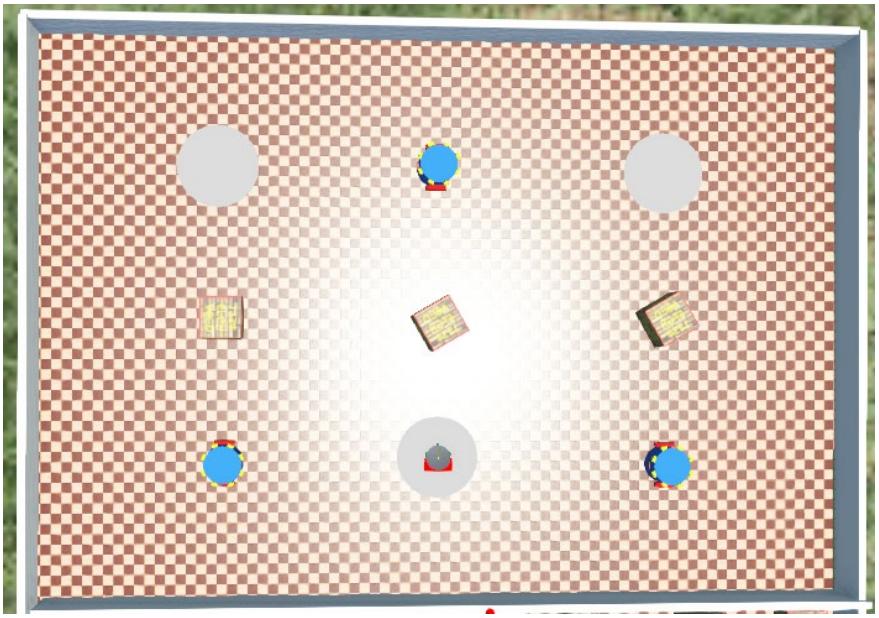


Figure 5-1. The simulation environment for unsupervised learning in dynamic environment. Three static boxes and three moving robots(objects).

The algorithm for the data collection is shown in table 5-1, let the computer run the simulation for a few days to get enough data. Since the dynamic environment is more complex and requires more data than the previous simulation in static environment.

Table 5-1. Algorithm for the first try in dynamic environment.

Algorithm: Dynamic environment first simulation	
1	Initialize the value: start = 1, result =0, movement_time = 0, decision= 0;
2	If movement_time = 0 & result =0 Reda the robot position, compass reading and LiDAR scan; Loop 5 times to get 5 consecutive scans then save all these data into Data_1;

```
    Give random decision and set the movement_time = 100;
  End
3 If touch_sensor_value(bumper) > 0
    Reset the robot;
  End
4 If touch_sensor_value(bumper) = 0 & result = 1
    If movement_time > 0
      movement_time – movement_time =1;
    End
    If movement_time = 0
      Decision = decision + 1;
      Save Data_(decision);
    End
  End
```

5.1.2 Data pre-processing

The data collection part takes around 2 days to complete, and the dataset is show in the table below.

Table 5-2. The number of datasets collected for dynamic study.

Group	1	2	3
Data	83,854	168,195	83,906

5.1.3 Training the CNN

The structure of neural network for dynamic study is show in Figure 5-2 and Figure 5-3 below. The input layer is changed to 3-D image input with value [1 67 5 1], which includes five image data.

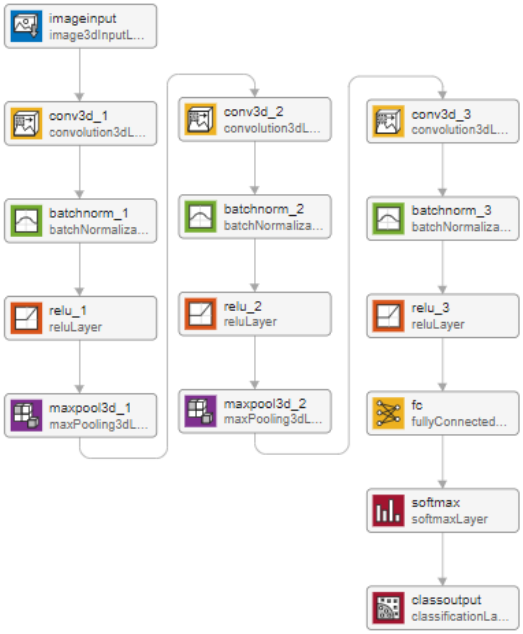


Figure 5-2. The diagram of the proposed neural network in dynamic study.

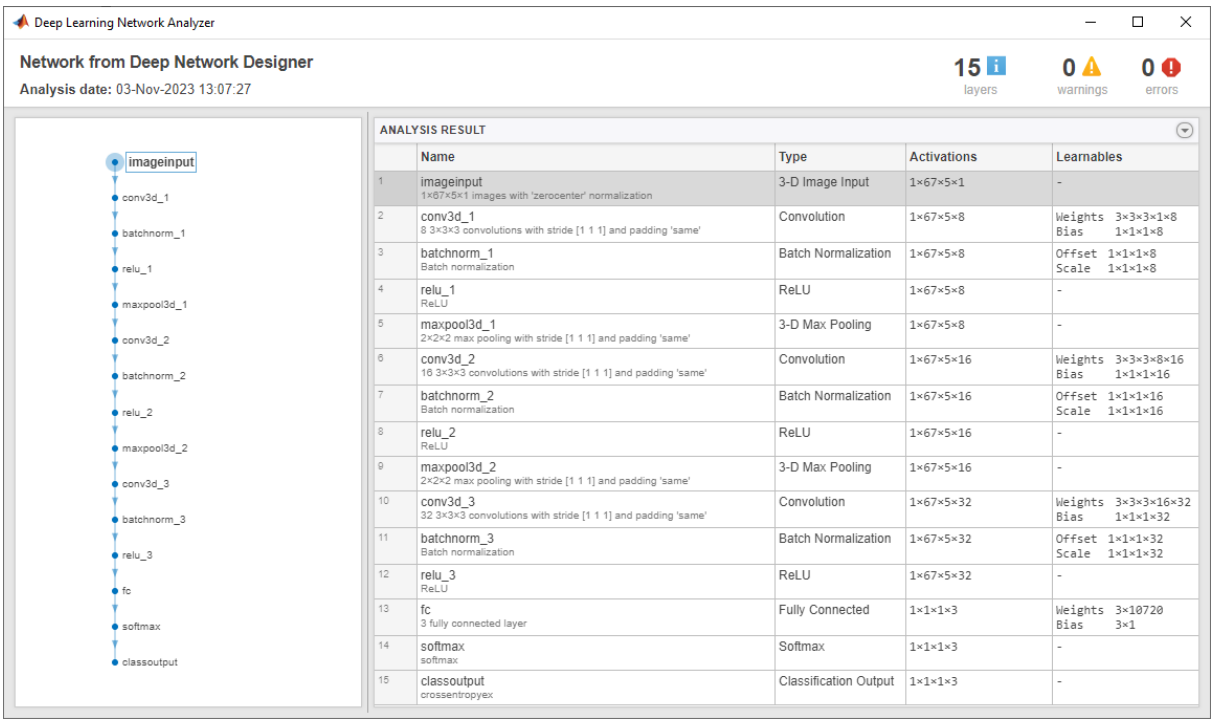


Figure 5-3. The detailed information of each layer for the proposed neural network in dynamic study.

The CNN training result is shown in Figure 5-4 below. The accuracy is stable at around 36% after 2 epochs and the training time is quite large with more than 1632 mins (27.2 hours!).

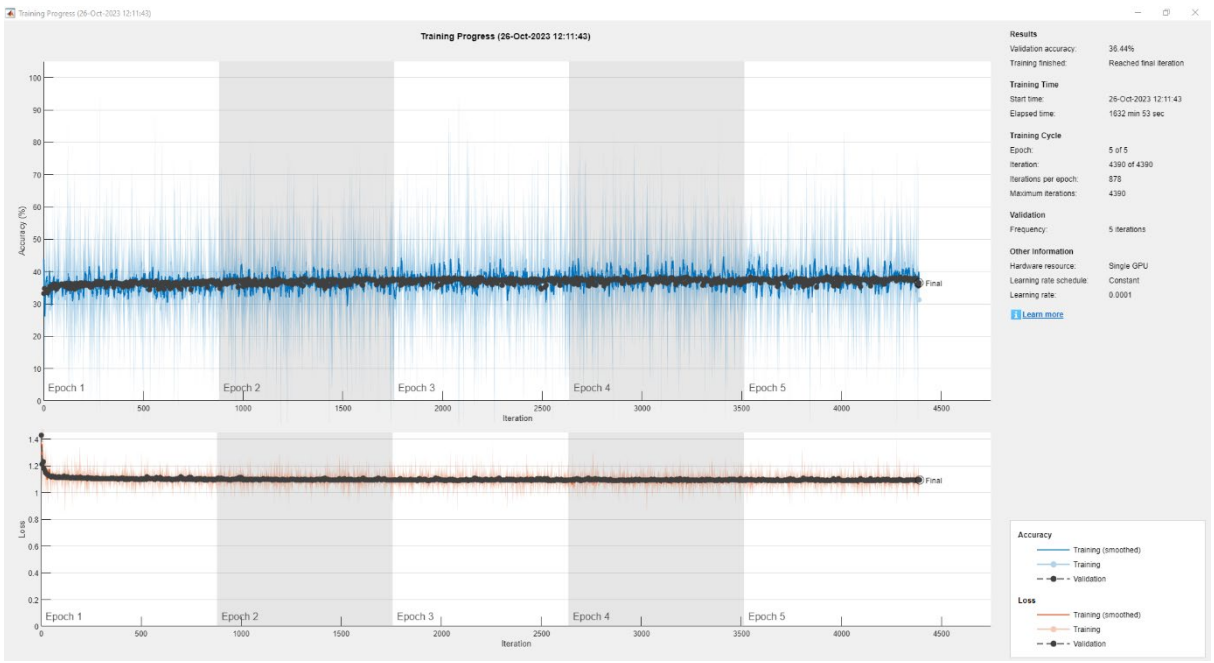


Figure 5-4. The neural network training process and result in dynamic case.

5.1.4 Test and development

In this step, the trained network is used for robot navigation in the same dynamic environment as data collection part. The result is shown in Table 5-3 below.

Table 5-3. The result of testing the trained network navigation.

	No of good	No of bad	Collision Rate
Dynamic	348	22	**17.21**%

5.2 Improved algorithms in dynamic environment

Based on the previous first try in dynamic environment, the following improvements are added to the simulation:

- Improve the differences between data during the data collection phase. The first plan is to add a delay time during the consecutive 5 scans and of course keep the robot stationary while waiting for the delay. It will definitely increase the difference between each data and the network could have a good training with these data. Another strategy is to change the algorithm used to get the input images. The same decision is used for 5 consecutive movements and collect these scans to generate the input images. The movement time in this part needs to be reduced relatively.
- Increase the complexity of the environment by adding more static boxes and moving objects. The previous input data lack the information for the moving objects which makes the training results much bad.

The improved environment is shown in Figure 5-5 below, with 6 more boxes and 3 more moving robots. The ‘Ting’ robot still random start inside the 3 white circles.

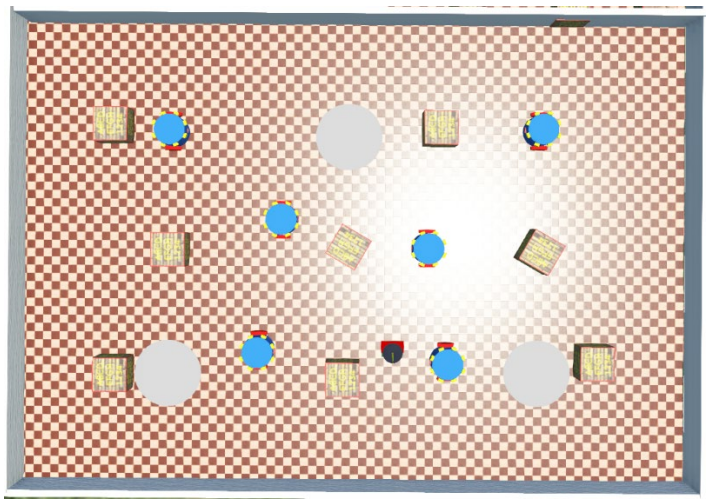


Figure 5-5. Improved environment with more static and moving objects for the dynamic case study.

5.2.1 Improved algorithm A by increasing the delay time

The improved algorithm A by adding delay time during each scan is shown in Table 5-4 below.

Table 5-4. The improved algorithm A for dynamic case with increasing the delay time.

Algorithm A: Dynamic environment with delay time	
1	Initialize the value: start = 1, result =0, movement_time = 0, decision = 0;
2	If movement_time = 0 & result =0 Reda the robot position, compass reading and LiDAR scan; The robot stays here and set a wait time = 20; Loop 5 times to get 5 consecutive scans then save all these data into Data_1; Give random decision and set the movement_time = 100; End
3	If touch_sensor_value(bumper) > 0 Reset the robot; End

```
4       If touch_sensor_value(bumper) = 0 & result = 1
          If movement_time > 0
            movement_time – movement_time =1;
          End
          If movement_time = 0
            decision = decision + 1;
            Save Data_(decision);
          End
        End
```

After sorting the data, a total of 219,488 data was obtained for this part. The original data becomes 2,580,860 after the MATLAB additional data processing. The input data for the improved Algorithm A is shown in Table 5-5 below and the CNN training result is shown in Figure 5-6.

Table 5-5. The input data for the improved algorithm A.

	Group 1	**Group 2**	**Group 3**
Original data	54,481	109,910	55,097
Additional data	641,380	1,292,230	647,250

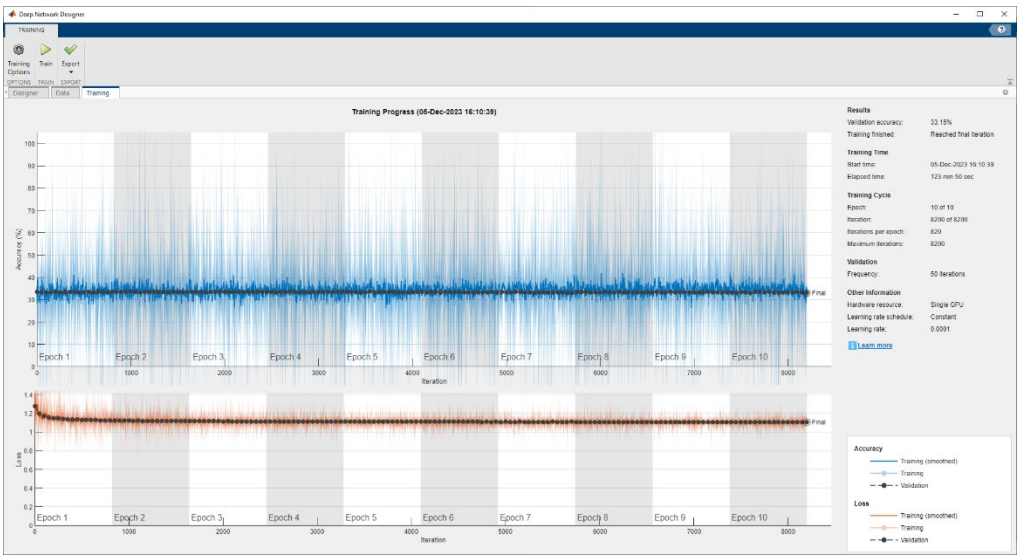


Figure 5-6. The training result with the improved algorithm A.

5.2.2 Improved algorithm B by using the same decision

Another method is to divide the original 100 movement time into five 20 movement time and use the same decision for the five movements. In this case, more information will be stored in the input data and the neural network will be better trained. The improved algorithm B is shown in Table 5-6 below.

Table 5-6. The improved algorithm B for dynamic case with same decision in 5 consecutive scans.

Algorithm B: Dynamic environment with the same decision in 5 consecutive scans	
1	Initialize the value: start = 1, result =0, movement_time = 0, decision_once = 0, newdecision = 0;

2	If movement_time = 0 & result =0
	Reda the robot position, compass reading and LiDAR scan;
	newdecision = newdecision + 1;
	If newdecision = 6
	decision = decision + 1;

```
        newdecision = 1;
      End
      Save the data into Data_(decision);
      Give random decision and set the movement_time = 20;
    End
3   If touch_sensor_value(bumper) > 0
      Reset the robot;
    End
4   If touch_sensor_value(bumper) = 0 & result = 1
      If movement_time > 0
        movement_time – movement_time =1;
      End
      If movement_time = 0
        Save Data_(decision)_(newdecision);
      End
    End
```

The original data and the additional data are shown in Table 5-7, the additional data is generated by the same code in static one. On this basis, 100,000 and 200,000 data are selected as input data to train the neural network. The training results are shown in Table 5-8 and Figure 5-7.

Table 5-7. The original and additional input data for the improved algorithm B.

	Group 1	Group 2	Group 3
Original data	78,881	158,005	78,451
Additional data	1,567,370	3,151,910	1,562,890

Table 5-8. The training results for the two data set:100,000 and 200,000.

Data set	Validation Accuracy
100,000	99.96%
200,000	99.99%

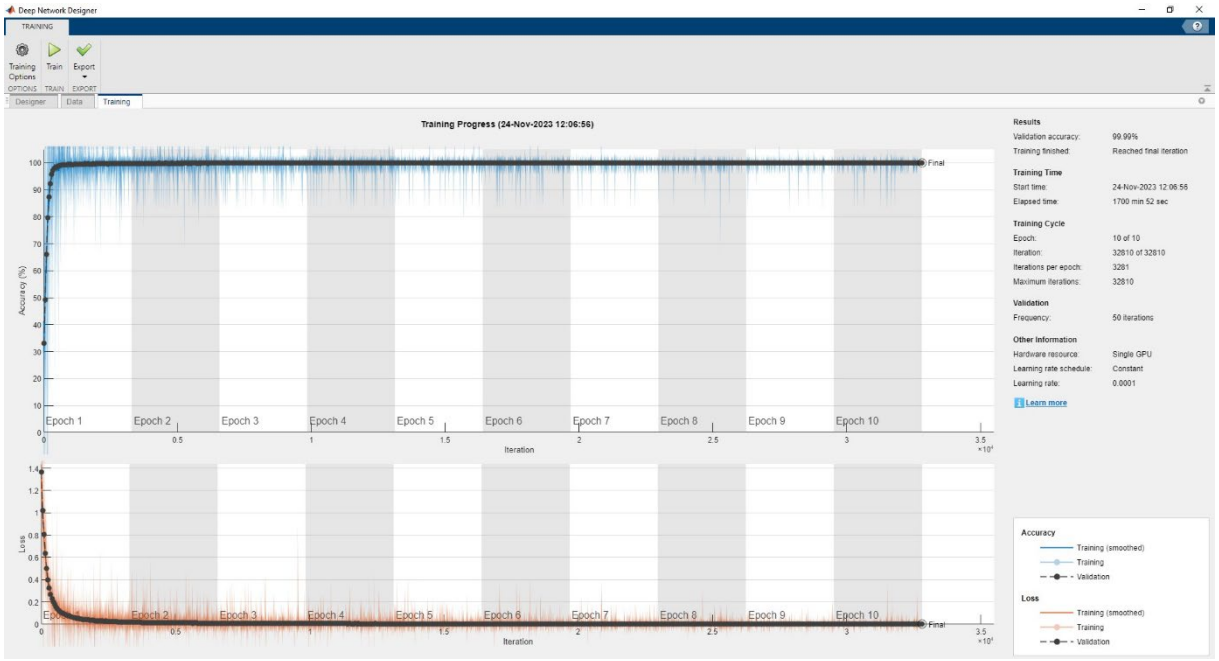


Figure 5-7. The training result with the improved algorithm B.

Table 5-8 shows the that no matter how the data set is expanded, the training accuracy is close to 100% which means over learning or over fitting. Therefore, the expected performance of the trained neural network is definitely not good, and the neural network

test part proves this. The robot cannot avoid the moving objects and always has the same decision.

5.2.3 Overfitting in neural network

In the realm of robotics, neural networks have emerged as pivotal tools for enabling autonomous navigation in dynamic environments. This application is particularly challenging due to the unpredictability and complexity inherent in real-world settings. A critical issue that often impedes the robustness of these neural network models is overfitting. Overfitting occurs when a model, trained extensively on a specific dataset, loses its generalization capabilities, making it less effective in real-world scenarios. This phenomenon is especially detrimental in robot navigation, where adaptability to new and unforeseen conditions is crucial. Addressing overfitting is therefore not just a theoretical concern but a practical necessity to enhance the reliability and efficiency of robots in dynamic settings[128].

5.2.3.1 Understanding and identifying overfitting in neural network

Overfitting in neural networks is a condition where a model learns the training data too well, including its noise and outliers, leading to poor performance on new, unseen data. This issue is particularly pronounced in complex models with large numbers of parameters, a common characteristic of networks designed for robot navigation in dynamic environments. The training process of a neural network involves adjusting its weights to minimize the difference between predicted and actual outcomes. In overfitting, the model becomes overly specialized to the training data, losing its ability to generalize. This results in high accuracy on training data but poor predictive power on new data. In the context of robot navigation, this translates to a robot that performs well in controlled or familiar environments but fails to navigate effectively in new or varied settings. Overfitting can be attributed to several factors. A primary cause is an insufficiently diverse training dataset. If the data does not represent the variety of scenarios a robot might encounter, the model will struggle to generalize. Another contributing factor is excessive model complexity. A model with too many parameters relative to the amount of training data can "memorize" data details, including noise, rather than learning to generalize from broader patterns. The implications of overfitting in robot navigation are significant. A model that overfits may navigate flawlessly

in a simulated environment or one similar to its training data but can fail unpredictably in new or more complex real-world situations. This unpredictability is not acceptable in applications where robotic reliability and safety are paramount.

Detecting overfitting is a critical step in developing effective neural networks for robot navigation. The most straightforward indicator of overfitting is observing the model's performance on a separate validation dataset, not used during training. If the model shows high accuracy on the training data but performs poorly on the validation data, it is likely overfitting. Another sign is when training accuracy continues to improve, while validation accuracy starts to decline. This divergence suggests that the model is becoming too specialized to the training data. Additionally, plotting learning curves that show model performance over time on both training and validation sets can visually highlight overfitting occurrences.

In the context of robot navigation, overfitting might also be identified through practical trials. For instance, if a robot navigates effectively in a known environment but struggles in new or slightly altered settings, this practical inconsistency can indicate an overfitted model.

5.2.3.2 Solutions to overfitting

To avoid overfitting in neural network, several techniques can be employed: regularization, adding noise, dropout, early stopping, weight constraints, data augmentation, training with more data, feature selection, K-fold Cross-validation and reducing model complexity[129-131]. Below is a Table 5-9 summarizing the solutions for these techniques, along with their description, advantages, and limitations.

Table 5-9. The solutions for overfitting in neural networks for robot navigation, along with their description, advantages and limitations[129-131].

Solutions	Description	Advantages	Limitations	Reference
Data Augmentation	Increasing the dataset's size and variety by modifying existing data.	Enhances training data diversity; more robust model.	May not capture real-world complexity; can create unrealistic examples.	[132]

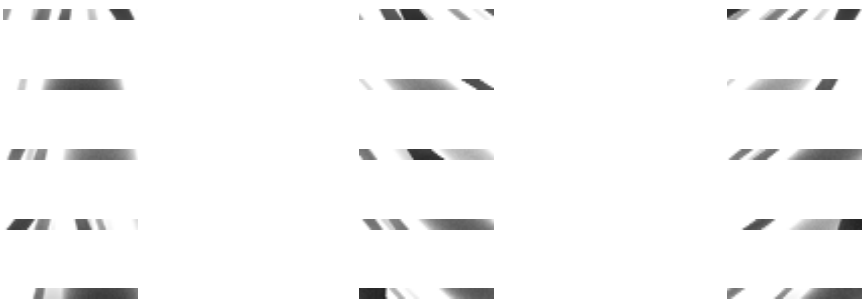
Regularization	Adding a penalty term (L1/L2) to the loss function to discourage complex models.	Simple, effective in reducing overfitting.	Proper regularization parameter tuning needed; can lead to underfitting.	[133]
Dropout	Randomly setting a fraction of input units to 0 at each update during training.	Reduces neuron co-adaptation; improves generalization.	Increases training time; dropout rate tuning needed.	[124]
Early Stopping	Stopping the training when the model's performance on a validation set starts to degrade.	Prevents overfitting, easy to implement.	Requires representative validation set; can stop training prematurely.	[134]
Cross-Validation	Dividing the dataset into multiple sets and rotating training on these subsets.	Robust model validation; reduces overfitting.	Computationally expensive, especially for large datasets.	[135]
Reducing Model Complexity	Using simpler models with fewer parameters or layers.	Easier to train, less likely to fit noise.	May not capture all complexities needed for accurate navigation.	[136]
Ensemble Methods	Combining multiple models for predictions.	Better performance and robustness than individual models.	Increases computational cost and complexity.	[137]
Transfer Learning	Utilizing a pre-trained model on a similar task and fine-tuning it	Reduces need for large training dataset, accelerates	Pre-trained model might not be fully suitable for the	[138]

	for the specific task.	training.	specific task.

5.2.3.3 Adjusting to input data

An input data is obtained by scanning 5 times with the same decision, in this case, in the test simulation part, starting with decision = 1, the trained network will always think that the decision = 1 is the best match. Because the deviation in other decision picture is obvious. (Previously the data was sloping according to direction which resulted in the same decision being made each time) The program for updating the input data is attached in Appendix E, the update data mostly all tends to point upwards towards the centre, which is the same for straight, left, and right turn. All the data looks a similar shape, and the network will be able to pull out the moving objects and make the correct decision. The input data (5 images each decision) of original and update group is shown in Table 5-10.

Table 5-10. The input data (5 images each decision) of original and update group.

	Decision 1	**Decision 2**	**Decision 3**
Original input data			
Update input data			

Then use the update input data to train the neural network, the result is shown in figure. The validation accuracy this time is 93.74% which looks much better than the previous training. After that, implement the trained network to navigate the robot in the test simulation part. Finally, organize the collected data in the test part, process and calculate the obstacle avoidance ability using the same method previously used in the static environment. The result is shown in Table 5-11.

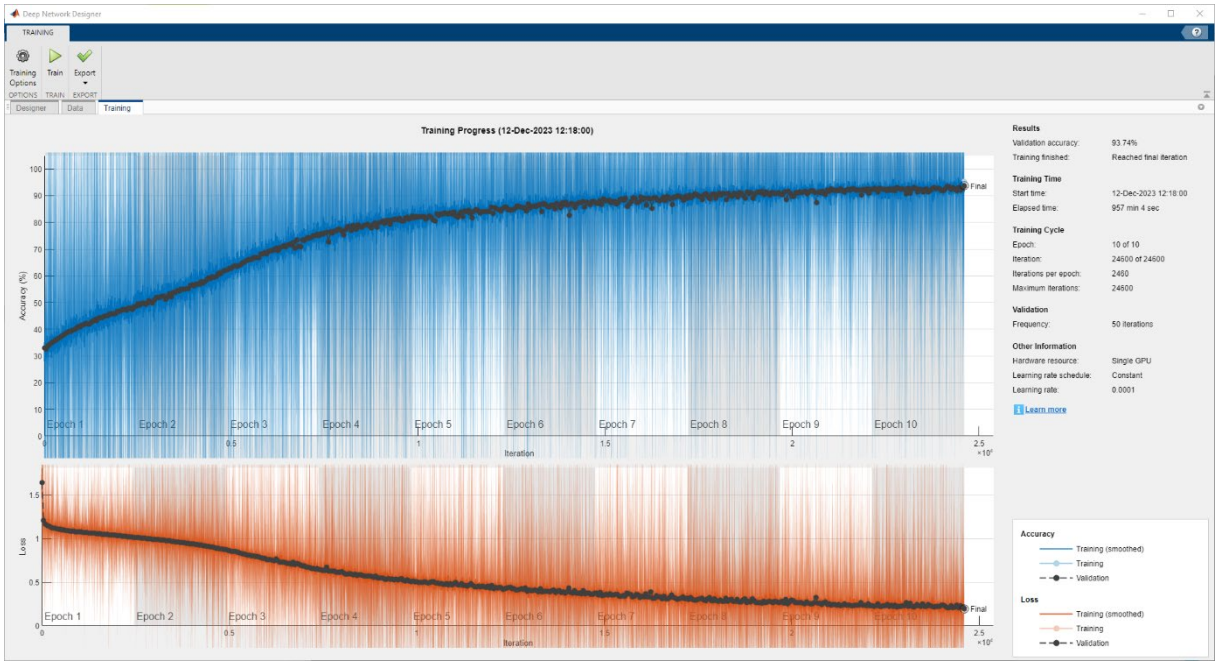


Figure 5-8. The training result for the update input data in dynamic case with the improved algorithm B.

Table 5-11. The test result for the trained network with update input data.

	Percentage of north travel	**Collision Rate**
Update algorithm	-3.22%	6.78%

5.3 Conclusion in dynamic neural network

This study aimed to develop and assess a dynamic neural network approach for enhancing the efficiency and accuracy or robot navigation. The implemented model and algorithms demonstrated significant improvements in navigational precision and adaptability in varied environments, as evidenced by the collision rate reduced from 17.21% to 6.78%. This finding represents a significant contribution to the field of autonomous robotics, illustrating the potential of dynamic neural network in controlled, yet complex and virtual environments.

It is, however, crucial to acknowledge the limitations faced during the study. While the model showed promising results in the Webots simulation, its performance and applicability in real-world scenarios remain to be validated. Additionally, the computational demands of the model pose a challenge for its deployment in less controlled environments.

Future research could focus on bridging the gap between simulated and real-world applications. Enhancing the model's ability to process real-world sensory data and improving its computational efficiency would be key steps in this direction. Further exploration into the scalability of the model to more complex and unstructured environments would also be a valuable area of study.

Chapter 6. Physical implementation

6.1 Introduction

The transition from simulation to real-world implementation is a crucial step in validating the performance of any neural network model developed for robotics. In this chapter, we focus on testing the neural network, which was trained in the Webots simulation environment, on a physical robot in a controlled indoor lab. This test aims to evaluate the network's navigation and obstacle avoidance capabilities in a static environment similar to the one used in simulation, but with the added complexities and unpredictability inherent in real-world conditions.

To ensure accurate positioning and navigation during the tests, the robot was equipped with super beacons to track its position and (Popolu Minimum-9 v5) an IMU to provide real-time compass heading data, replacing the simulated compass readings used during training. These additional sensors allow the robot to gather real-time spatial and orientation data, feeding this information into the neural network for decision-making.

This chapter will outline the steps taken to implement the neural network on the real robot, describe the hardware setup used for position tracking and heading detection, and provide detailed results from the testing phase. The performance of the network in the real-world environment will be compared to its results in simulation, offering insights into how well the neural network transfers from a simulated environment to a physical one and identifying any potential discrepancies or challenges faced during the implementation.

6.2 Hardware Setup Overview

To ensure that the neural network operates as effectively in the real world as it did in the simulation, the robot was outfitted with two key hardware components: super beacons for tracking its position and an IMU for capturing its compass heading. These components provide the critical spatial and orientation data needed for the robot to navigate accurately.

6.2.1 Super Beacons for Position Tracking

Super beacons play a pivotal role in translating the precision of the simulation environment into real-world testing. While the robot in Webots can continuously access its position programmatically, such ready-made positional data does not exist outside of the virtual space. Consequently, an external positioning system becomes necessary to track the robot's progress and ensure it remains aligned with the path that the neural network defines. By placing fixed beacons around the perimeter of the lab, the setup effectively recreates the constant position feedback available in the simulation, but now grounded in physical reality.

The underlying principle of the super beacon system relies on signal-based triangulation. Each beacon emits signals picked up by a transceiver mounted on the robot; by accurately measuring the signal travel times between the beacons and the receiver, the system can calculate the robot's exact position within the environment. These calculations occur continuously and at low latency, allowing the neural network to receive near real-time positional updates. This frequent refresh of location data means that the robot can respond dynamically to changes in its surroundings, especially when it detects potential obstacles via LiDAR.

A key benefit of using super beacons is their centimetre-level precision. In many real-world navigation applications, even small discrepancies in position tracking can accumulate over time, causing the robot to deviate from its planned route. By maintaining a high degree of accuracy, the super beacons help the robot replicate the precise manoeuvring it performed in simulation, whether traveling in a straight line northward or making tight turns around obstacles. Another advantage arises from the wide coverage that multiple beacons provide. Placing them strategically around the test area ensures the robot can be tracked consistently regardless of its location, which prevents signal loss or patchy tracking that might otherwise disrupt navigation.

Another essential property is the system's low latency. In realistic scenarios where a robot must make split-second decisions, slow position updates could mean the difference between navigating smoothly around an obstacle and colliding with it. With the super beacon system in place, the neural network can access the robot's position quickly enough to keep pace

with its forward movement, applying slight course corrections whenever sensor data indicates a looming collision or a drifting heading.

Overall, the super beacon system offers a solution for bridging the gap between simulation and physical deployment. By delivering precise, continuous, and low-latency location data, these beacons preserve the performance advantages the neural network achieved in simulation. Without such accuracy, the robot could experience significant challenges—either wasting time on inefficient routes or risking collisions if it drifts off its planned path. In this sense, the super beacon system forms a cornerstone of the successful transfer of CNN-based navigation from virtual tests to real-world applications.

6.2.2 IMU for Compass Heading

To complement the position tracking provided by the super beacons, IMU was added to the robot to provide real-time compass heading information. In the Webots simulation, the robot's orientation (compass heading) was known at all times. In the physical implementation, the IMU replicates this functionality, providing the heading data needed for the robot to navigate effectively towards the north while avoiding obstacles.

The IMU used in this implementation contains accelerometers, gyroscopes, and magnetometers to track the robot's orientation relative to the environment. The IMU's compass feature is especially important for the northward navigation task that the neural network was trained to perform in the simulation.

The IMU provides accurate, real-time heading information that ensures the robot maintains directional awareness and can quickly adjust its path thanks to minimal data delay; furthermore, its seamless integration with the neural network guarantees that the robot's decision-making process aligns with its training conditions. By offering timely feedback, the IMU plays a crucial role in orienting the robot correctly during navigation, particularly when moving northward while avoiding obstacles. The continuous stream of reliable heading data allows the neural network to make informed decisions about the robot's path in the same way it did during the simulation phase.

6.2.3 Integration of Super Beacons and IMU with the Neural Network

The integration of the super beacons and IMU with the neural network was a key step in ensuring that the robot received the same type of data in the real-world environment as it did during simulation. This involved synchronizing the position data from the super beacons and the heading data from the IMU to feed into the neural network in real-time.

Both the position and heading data are pre-processed and synchronized before being fed into the neural network. This ensures that the neural network receives real-time data in the correct format, allowing it to make accurate decisions based on the robot's current location and orientation.

The pre-processing ensures that the sensor data (super beacons and IMU) is formatted in a way that is compatible with the input layer of the neural network, allowing for a seamless transition from simulated data to real-world sensor data.

This integration is critical for testing the neural network's generalization capabilities in real-world scenarios. By providing the same type of data (position and heading) to the neural network in the real-world environment, we ensure that the neural network is tested under conditions as similar as possible to those in the simulation, enabling a fair assessment of its performance.

6.3 Integration of the Trained Neural Network

In this section, we focus on the steps involved in deploying the trained neural network from the Webots simulation environment onto the physical robot. The neural network, which was trained and validated using MATLAB's Deep Learning Toolbox, must be transferred, and integrated with the real-time sensor data from the super beacons and IMU to enable the robot to navigate the indoor lab environment autonomously.

6.3.1 Transferring the Neural Network

The neural network trained in the Webots simulation is initially developed and refined in MATLAB, and for real-world deployment, it must be exported from MATLAB's environment

and then uploaded onto the robot's onboard computational system. The first step involves exporting the trained CNN from MATLAB as a Series Network object, preserving the entire architecture—including layers, weights, biases, and trained parameters—and saving it in a format compatible with the robot's onboard processor so that the model structure remains intact during transfer. Next, the robot's onboard processor, typically a microcontroller or embedded computer, is configured to handle incoming sensor data in real time by installing the necessary software packages to run the neural network model, such as MATLAB Runtime or lightweight libraries like TensorFlow Lite, which can interpret the trained model and execute it efficiently. Once this configuration is complete, the neural network model is loaded into the robot's processor and linked to the real-time sensor data. The robot's control system is then designed to handle the inference from the neural network, converting sensor inputs from sources like LiDAR, IMU, and super beacons into actionable commands such as moving forward or turning left or right.

6.3.2 Real-Time Data Handling

Sensor data management and processing form the backbone of the robot's ability to navigate effectively. Before the neural network can make any decisions, the raw sensor output must be converted into a format that the network can interpret. By carefully pre-processing and organizing the data from various sensors, it becomes possible to deliver reliable inputs to the neural network and obtain timely, accurate responses.

The LiDAR sensor operates by continuously scanning the robot's surroundings, detecting obstacles, and calculating distances to nearby objects. This sensor essentially provides a real-time "map" of the environment, which is updated as the robot moves. However, the unprocessed LiDAR data may contain noise or extraneous information, so a filtering step is essential. During pre-processing, the LiDAR readings are arranged in a manner consistent with the neural network's expected input structure. If the network was originally trained with LiDAR readings as a one-dimensional image, the same dimensionality (for example, $1 \times 67 \times 1$) is maintained. This ensures that the network does not become confused by differences in input shape and can immediately interpret the distances as it was trained to

do. Once the LiDAR data has been filtered and converted, it is ready to be sent into the neural network for real-time analysis.

The IMU's heading data, often expressed in degrees, is similarly refined to mirror the angle or vector format the neural network anticipates. This heading value is a key indicator of the robot's orientation, enabling the network to assess whether the robot is currently aligned with the direction it intends to move. By transforming the IMU's raw angular data to match the network's training inputs, the robot can consistently use the same reference points for direction in both simulation and physical tests. Thanks to these conversions, the IMU data becomes fully compatible with the neural network and can be integrated alongside the LiDAR readings without generating errors or confusion.

In contrast to the LiDAR and IMU inputs, the position data derived from super beacons is not directly fed into the neural network. The super beacons determine the robot's location through triangulation, providing an external reference point for navigation. While this position data does not factor into the neural network's real-time decision-making, it serves an important function by offering a means to verify that the robot is moving as expected. If the robot's actual location begins to diverge from the intended path, the navigation system can identify the discrepancy and intervene if necessary.

Once the sensor data has been properly formatted, the robot's control unit sends it to the neural network on a continuous basis. This real-time feed allows the network to interpret the robot's immediate surroundings and orientation. Through a series of internal layers, each trained to recognize and respond to specific patterns, the neural network determines the robot's next action. It might, for instance, decide whether the robot should maintain its current course, execute a left turn, or take a right turn in response to an obstacle or in pursuit of its target heading. The capacity for immediate feedback ensures that the network can adapt quickly to any changes in the robot's environment.

The speed at which the neural network processes incoming data is critical for successful obstacle avoidance. Even a short delay can cause the robot to react too slowly, leading to collisions or unintentional detours. To address this challenge, optimization strategies for the neural network's calculations are put in place. These might involve efficient hardware

configurations, streamlined code, or specialized libraries that can handle matrix operations at high speed. By reducing processing lag to a minimum, the robot is capable of making prompt and effective decisions based on up-to-the-second sensor inputs.

In the final step, once the network arrives at a decision, it sends instructions directly to the motor control system. If, for instance, the network has detected an obstacle in front of the robot and determined that a left turn is the best action, the control system will instantaneously adjust the motor speeds on the relevant wheels. This alignment of sensors, processing, and actuation forms an unbroken chain of data and commands, enabling the robot to navigate its environment with a high level of autonomy and responsiveness.

6.3.3 Challenges and Adjustments

During the integration process, one of the most significant hurdles involved dealing with sensor data noise. In a virtual environment like Webots, the data is generally clean, consistent, and perfectly aligned with the simulated conditions. However, once the robot operates in the real world, the LiDAR sensor can occasionally pick up false readings triggered by reflections or interference from shiny surfaces and complex environmental shapes. These extraneous signals, if unaddressed, can lead the neural network to make erroneous decisions. To mitigate the impact of this noise, filtering techniques were introduced, smoothing the incoming data before it reached the neural network. This additional pre-processing step acted as a buffer against the fluctuations inherent in raw sensor outputs, ensuring that the robot based its decisions on more reliable information.

Another challenge that emerged was the timing and synchronization of sensor inputs. While the neural network requires simultaneous readings from the LiDAR, IMU, and super beacons for accurate inference, real-world devices often gather and relay data at slightly different intervals. Any lag or mismatch in these streams can cause the neural network to interpret outdated information and respond incorrectly. To address this, developers carefully managed the polling intervals of each sensor, ensuring that data streams were aligned as precisely as possible before being passed through the network. This approach allowed the robot's software to generate a coherent snapshot of its environment at every decision cycle,

preventing the system from blending new IMU data with older LiDAR information, or vice versa.

Processing speed posed yet another obstacle for successful deployment. Since the robot must move and react in real time, the neural network's inference engine needs to deliver outputs at a pace that aligns with the robot's motor control loop. Onboard computing hardware, although capable, might become a bottleneck if the model is excessively large or computationally intensive. Various optimization strategies were therefore employed to balance performance with accuracy. In some instances, the complexity of certain operations or the depth of specific network layers was reduced, allowing faster calculations without overly compromising the model's ability to generalize from its training. Profiling tools played a pivotal role in this stage, monitoring how long each inference step took and highlighting areas where further optimization could trim down processing times.

Thanks to these targeted adjustments—ranging from advanced noise filtering to careful sensor synchronization and computational streamlining—the neural network ultimately achieved a suitable real-time performance. This integration allowed the robot to move and adapt within a complex environment, drawing on its sensory inputs to make split-second decisions. With the operational system in place, attention could then shift to testing the robot's navigation capabilities in actual real-world scenarios. The following section details how the experiments were conducted and outlines the metrics used to evaluate the robot's overall effectiveness, paving the way for a comprehensive analysis of the results.

6.4 Experimental Procedure

Once the neural network was integrated into the robot and the necessary hardware components (super beacons and IMU) were fully functional, the next step was to conduct real-world tests to evaluate the network's performance. This section outlines the experimental procedure used to test the robot's navigation capabilities in a controlled indoor environment, mirroring the static environment used during the simulation phase. The

objective of these tests is to assess how well the neural network, trained in a simulation, generalizes to real-world conditions.

6.4.1 Indoor Lab Environment Setup

The real-world tests were conducted in an indoor lab that closely resembles the static environment used during the Webots simulation. The environment consisted of predefined obstacles placed at fixed locations, and the robot's task was to navigate northward while avoiding these obstacles, just as it did during the simulation.

Figure 6-1. Implementation of the CNN on a lab robot. The lab setup is shown with beacon positions marked by red circles, with Beacon 5 located at the northernmost point. The robot is highlighted by dotted red circle.

An essential factor in the testing environment was the presence of static obstacles, which took the form of walls and boxes placed in predictable positions throughout the lab. Just like in the simulation, these obstacles were arranged in a fixed configuration that remained unchanged over the course of the tests. This consistency allowed the researchers to pinpoint how effectively the robot could detect and navigate around objects that it had previously encountered during its training phase, removing any ambiguity about whether unexpected variations in obstacle placement were influencing the robot's performance.

In addition to the arrangement of obstacles, the surface of the test area was deliberately kept flat and level. By ensuring that the floor offered a smooth, even terrain, the research

team minimized external variables that could otherwise distort the results. This approach helped isolate the robot's decision-making processes, making certain that any difficulties in movement arose from the robot's sensor readings or neural network-driven choices, rather than physical factors like uneven ground or inclines.

Accurate position tracking of the robot was another priority, and to achieve this, super beacons were strategically installed around the perimeter of the lab. These beacons provided precise signals that enabled the robot to determine its location at all times. The position data, although not directly integrated into the neural network's decision-making flow, proved essential for verifying whether the robot was moving as anticipated. By correlating the robot's actual trajectory with its expected path, the researchers gained valuable insight into how well the neural network, combined with other sensors, was guiding the robot through its environment.

Throughout these tests, a controlled lighting setup ensured that both the LiDAR sensor and IMU functioned under stable conditions. Any fluctuations in illumination could have introduced additional noise or measurement errors, especially for the LiDAR, which relies on light signals to gauge distances. By standardizing the lab's lighting, the team reduced the chance of sensor interference and allowed the robot's performance to be evaluated on the merits of its software and hardware design rather than on inconsistencies in the environment.

6.4.2 Testing Protocol

The testing protocol was designed to replicate realistic navigation conditions while staying true to the simulation parameters. The robot's overall performance was judged on its capacity to avoid obstacles, move accurately toward its intended destination, and respond promptly to sensor inputs. By doing so, the tests provided a clear view of whether the robot's real-world behavior matched its behavior in the virtual environment, ensuring that the transition from simulation to reality did not compromise its capabilities.

Central to the protocol was the requirement that the robot head north, reflecting the primary objective it had been trained on in simulation. To achieve this, the robot had to rely

on its LiDAR sensor for detection of obstacles in its path, with the neural network interpreting the sensor data in real time. The network decided whether the robot should turn left, turn right, or continue moving straight ahead whenever it encountered an obstacle. This emphasis on obstacle avoidance directly tested how well the network had generalized from its training data to actual conditions. Equally important was the continuous logging of all sensor readings—from the LiDAR, IMU, and super beacons—so that the research team could later assess precisely how the robot perceived its surroundings and made its decisions.

Before each test began, the robot was placed in a predefined location in the southern part of the environment, but oriented in a random heading. The choice of a random direction forced the robot to adjust its compass heading before it could begin traveling north. This setup effectively tested the robot's reliance on the IMU and the neural network's ability to recognize that it was not yet aiming in the correct direction. Once the test officially started, the robot was allowed to operate entirely under its own control. No manual input or intervention was permitted, ensuring that any successful navigation was due solely to the robot's sensory systems and the learned behaviours embedded in its neural network.

Throughout each test run, extensive data was collected. Position information from the super beacons allowed the team to see how closely the robot's path matched the intended route, and heading information from the IMU made it clear whether the robot was maintaining proper orientation. Meanwhile, the LiDAR data captured every instance in which the robot detected an obstacle. By comparing these logs against the robot's subsequent movements, the team could determine how well the neural network translated each sensor reading into a prompt decision. Delays or inaccuracies in responding to obstacles became evident when reviewing this collected information, making it easier to track the system's strengths or identify any potential weaknesses.

After a predefined amount of time—or once the robot reached a specified point along the northern boundary of the environment—each test concluded. If the robot successfully arrived at the northern target, the run was deemed a success, and the data was later analysed to understand precisely how the robot navigated. In cases where it could not reach the northern boundary before time expired, the logs and final positions still provided

valuable insights into what might have gone wrong, whether it involved sensor noise, incorrect heading adjustments, or a breakdown in obstacle recognition and avoidance. These outcomes laid the groundwork for refining the system and understanding how well the transition from simulation-based training to real-world deployment had been achieved.

6.4.3 Performance Metrics

One of the primary evaluation points involved navigation accuracy, which centred on how faithfully the robot adhered to its intended path toward the north. To obtain this measure, researchers gathered position data from the super beacons during each test. This positional information was then compared against the expected trajectory, making it easy to spot any deviations or course corrections the robot had made. In cases where the robot drifted significantly off the intended path, it became clear that the neural network or sensor suite was not providing sufficient information—or interpreting information correctly—about the robot's environment and heading.

Another important indicator was the obstacle avoidance success rate, which captured how many obstacles the robot successfully detected and circumvented without incident. The LiDAR sensor data, alongside the robot's actual manoeuvres, formed the basis for calculating this value. Every time the robot steered clear of an obstacle, the avoidance counts increased, while each unsuccessful attempt served as a reminder that the robot's LiDAR interpretation or the neural network's decision-making might need further refinement. When viewed collectively over multiple trials, the obstacle avoidance success rate gave a clear picture of whether the system could reliably interpret and act upon real-world sensor data to prevent collisions.

Although the obstacle avoidance success rate shed light on positive outcomes, the collision rate offered a direct measure of failure points within the navigation process. Expressed as a percentage of tests in which the robot collided with an obstacle, this metric was particularly revealing in highlighting situations where sensor noise or unexpected environmental factors led to mistakes. Because real-world conditions can differ drastically from the controlled nature of a simulation, even a robustly trained neural network might falter if the sensory input is distorted or delayed. The collision rate thus helped pinpoint areas where

improvements in sensor filtering, network architecture, or synchronization mechanisms might be needed.

A further layer of scrutiny came through heading adjustment accuracy, which examined how effectively the robot changed or maintained its direction based on the IMU data. Since the entire experiment revolved around having the robot move northward, any consistent misalignment in the heading could lead to an increased risk of collision, decreased navigation accuracy, and a longer time to reach the destination—if it reached the destination at all. By continuously tracking how well the robot stayed oriented toward the north, researchers could assess whether the IMU's compass data was being integrated properly into the neural network's decisions. Any recurrent deviations indicated gaps in the training data, inconsistencies in the real-time sensor feed, or limitations in the neural network's capacity to handle deviations from a straightforward heading.

6.4.4 Test Variations

One variation involved altering the robot's initial heading at the start of each test. Rather than always beginning in a uniform orientation, the robot was placed facing a different direction, forcing it to rely on its IMU data to identify that it was not pointing north. This alteration allowed researchers to gauge how quickly and accurately the system recognized its off-axis positioning and recalibrated its direction. If the robot corrected its heading efficiently across multiple trials and orientations, it indicated that the neural network was adept at integrating the IMU data into its decision-making process, regardless of how the robot began its journey.

A second type of variation adjusted the density of obstacles in the environment. In some tests, the lab space contained only a few widely spaced obstacles, while in others, the robot faced a labyrinth of closely packed barriers. Because the neural network had to interpret the robot's LiDAR data in real time, denser obstacle arrangements naturally provided a stricter test of its navigation and avoidance algorithms. High-density scenarios in particular challenged the robot's ability to pick a route that avoided collisions while still maintaining a generally northward trajectory. By comparing performance across environments of varying

complexity, researchers could see whether the system's training and sensor-processing capabilities held up under more demanding conditions.

A third category involved deliberately injecting artificial noise into the sensor feeds, predominantly the LiDAR and IMU data streams. In an idealized setting, sensors might provide stable and precise information, but real-world conditions often involve reflections, interference, or other inaccuracies. By emulating these disruptions, the research team could test the neural network's robustness in the face of flawed or incomplete data. If the robot continued to demonstrate reliable obstacle avoidance and heading correction despite higher levels of noise, it suggested that the network had effectively learned to recognize and compensate for sensor variability. Conversely, if performance dropped sharply, it signalled the need for further refinement in either the network's architecture or in the filtering and synchronization mechanisms that shaped the final data inputs.

6.4.5 Data Collection and Analysis

Throughout the testing process, an extensive record of sensor data—including LiDAR readings, IMU readings, and super beacon positions—was maintained alongside logs of the robot's real-time decisions and physical movements. This comprehensive dataset provided a valuable basis for understanding how effectively the neural network performed outside the controlled conditions of the simulation. By examining every step, the robot took and every decision the neural network made, researchers could gauge how closely the robot's behavior matched the predicted outcomes from the simulation phase.

One key area of analysis involved comparing the robot's actual path with the path it was intended to follow. Since the primary objective was to move north while avoiding obstacles, each test generated a trajectory that could be measured against the "ideal" route for that particular run. Deviations from the intended course helped pinpoint where the neural network might have struggled—whether due to sensor noise, an inadequate understanding of the environment, or a slower-than-expected reaction to changes in the robot's surroundings. The robot's ability to accurately maintain a northward heading, and the

degree to which it could navigate around obstacles without veering too far from its goal, revealed much about the network's real-world adaptability.

Researchers also devoted significant attention to the decision log produced during each test. Every time the neural network determined whether the robot should move forward, turn left, or turn right, these decisions were recorded and correlated with the sensor readings from the same moment in time. This correlation allowed the team to see precisely how the robot's environment—particularly the distribution of obstacles or changes in its heading—triggered specific outputs. If the robot continuously made decisions that corresponded well to the sensor data, it suggested that the network was both interpreting and reacting to real-world conditions in a manner consistent with its training. Conversely, any recurring mismatches or inexplicable actions hinted at gaps in the network's learning or limitations in how sensor data was filtered and fed into the network.

Another dimension of the analysis involved comparing performance metrics across various test variations. Because the tests included scenarios with different initial headings, varying obstacle densities, and artificially injected sensor noise, researchers could observe whether the neural network's robustness held up under increasingly difficult conditions. Runs featuring higher obstacle densities, for instance, revealed how well the network could cope with more frequent course corrections. Tests with altered heading positions indicated the network's capacity to recalibrate the robot's orientation quickly. Meanwhile, artificially noisy sensor data offered insights into how tolerant the system was to less-than-ideal inputs—an important consideration for real-world usage where no sensor is perfectly reliable.

By drawing all these strands together—actual paths taken, decision logs, and performance across different environmental and sensor configurations—the research team gained a holistic view of the neural network's strengths and weaknesses. The ensuing results section will delve into these findings in detail, highlighting where the network excelled, where improvements are needed, and how closely its performance in the lab environment resembled the simulations that came before. This comparison between real-world data and simulated data ultimately sets the stage for refining the neural network architecture and

improving integration with the physical robot, ensuring an even more capable navigation system in future iterations.

6.5 Results and Analysis

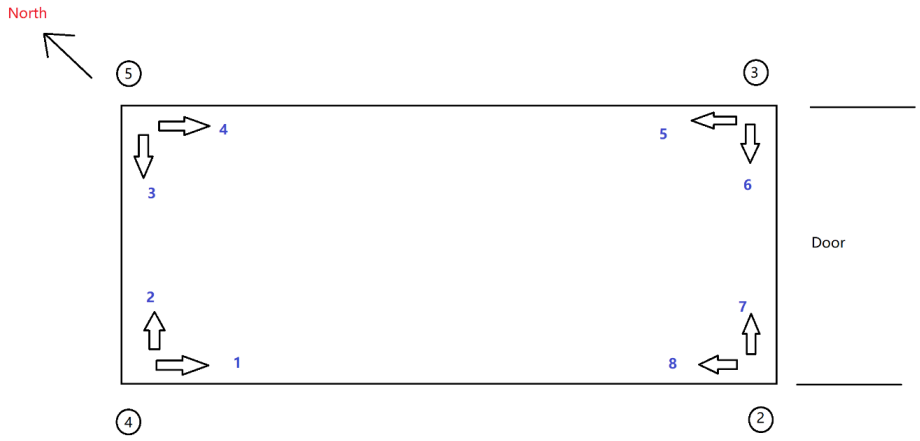


Figure 6-2. The experiment structure in the lab. The numbers in the circle 2,3,4,5 represent the four stational beacons. Beacon 5 is the most north part of this lab. Beacon 4 is the location of the laptop that controls the movement of the mobile robot and receives and sends signals. There are eight different start positions (Blue numbers) to test the trained neural network.

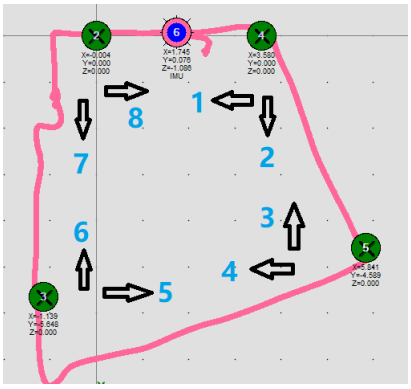


Figure 6-3. The figure obtained in the latest SW pack. Using mobile beacon (number 6) to record the map around the edge (wall). These positions are automatically generated by this software SW pack, refer to the previous figure 7-1 and adjust the correct order of numbers.

There are eight groups (different start positions) for this experiment work and test the trained neural network for LiDAR data only and LiDAR data with compass heading. Running the experiment three times for each group to avoid unnecessary errors and analyse the problem. The group A stands for LiDAR data only while the group B stands for LiDAR data with compass heading and the trajectories of the robot movement are shown in Figure 7-4, Figure 7-5. Figure 7-6 and Figure 7-7 below. The MATLAB code for the running experiment is shown in Appendix E. The loop time for each group is 50.

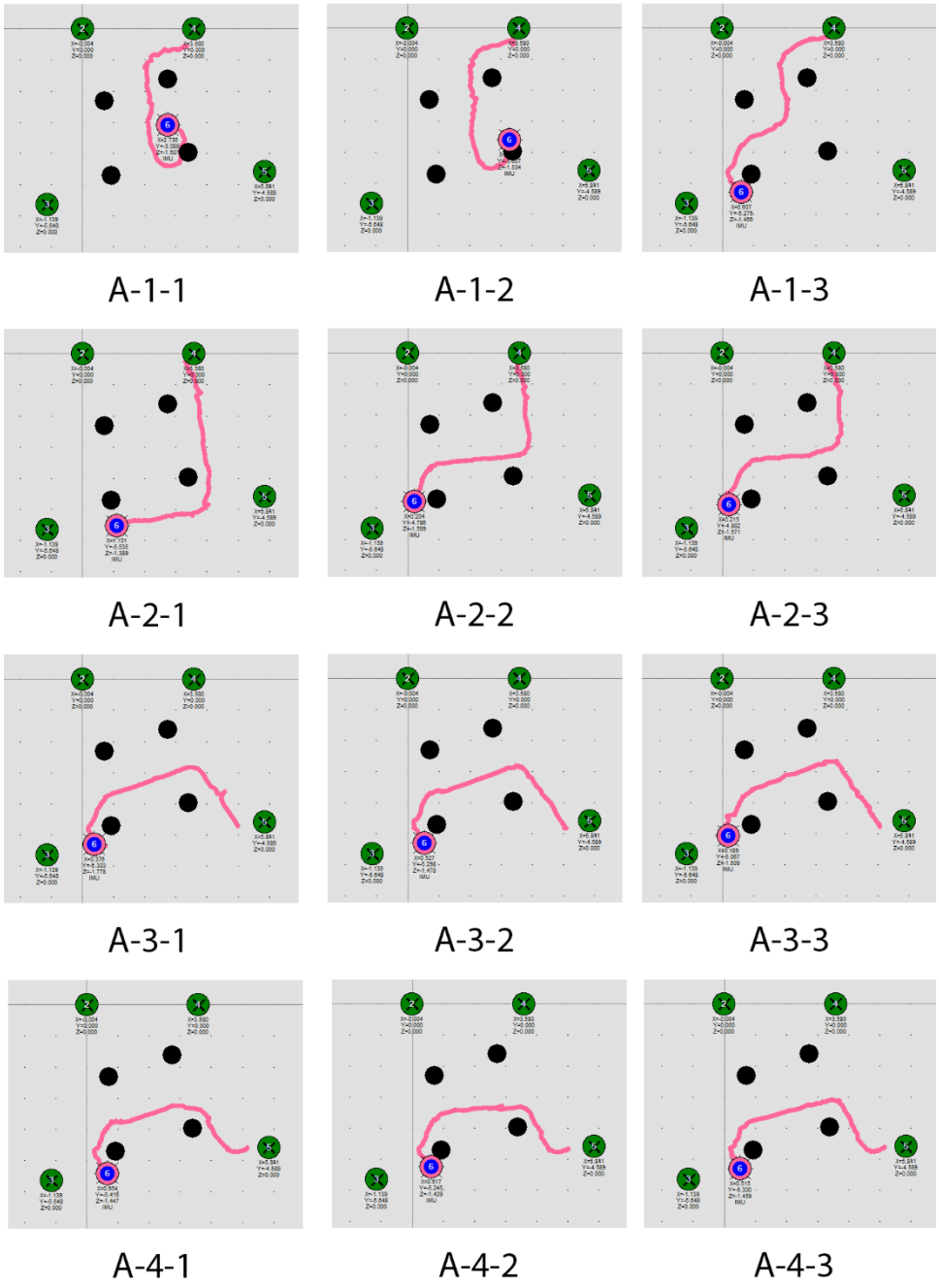


Figure 6-4. Group A 1-4 experiment results. Using trained neural network with LiDAR data only to navigate the robot in the lab.

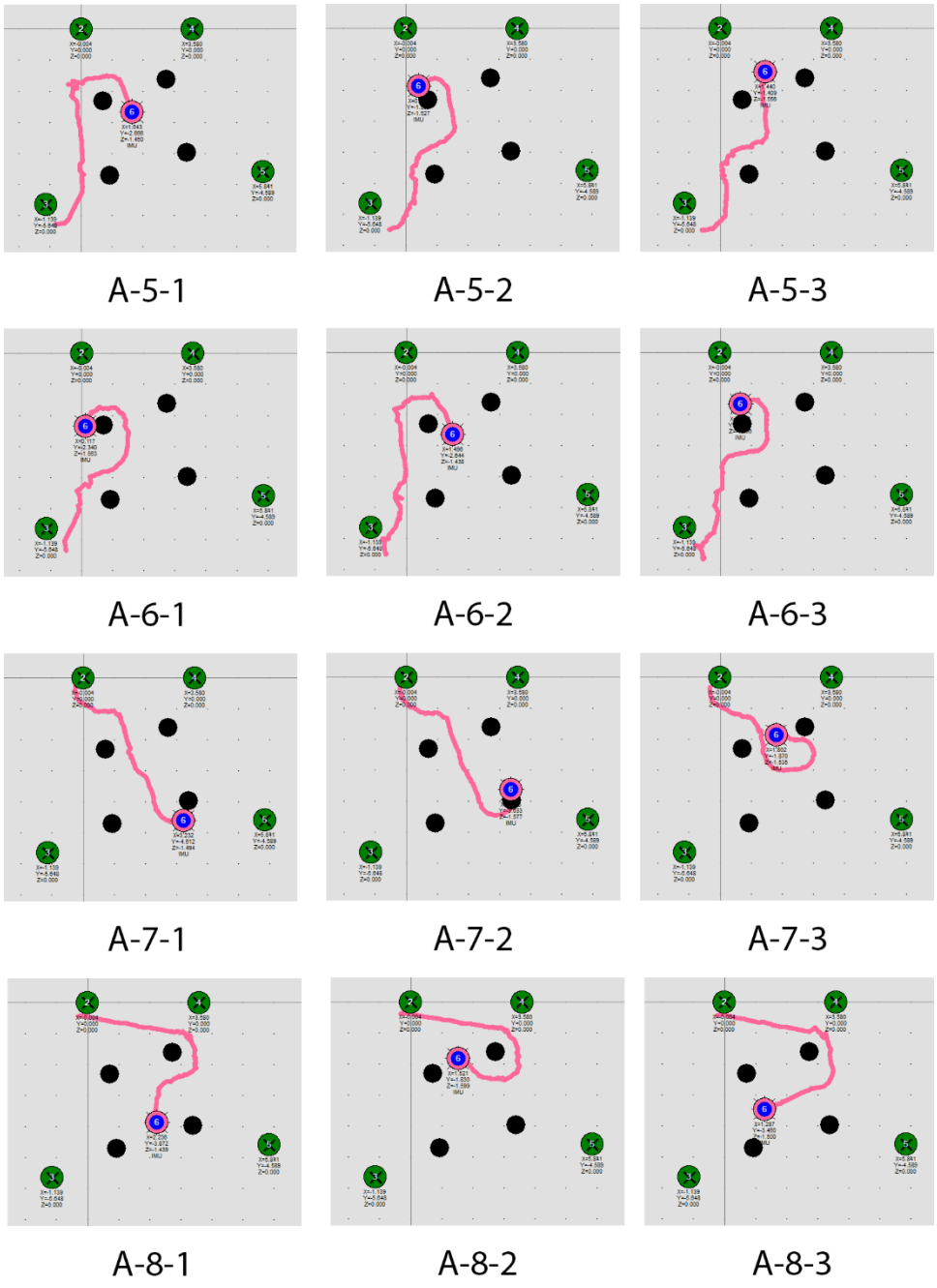


Figure 6-5. Group A 5-8 experiment results. Using trained neural network with LiDAR data only to navigate the robot in the lab.

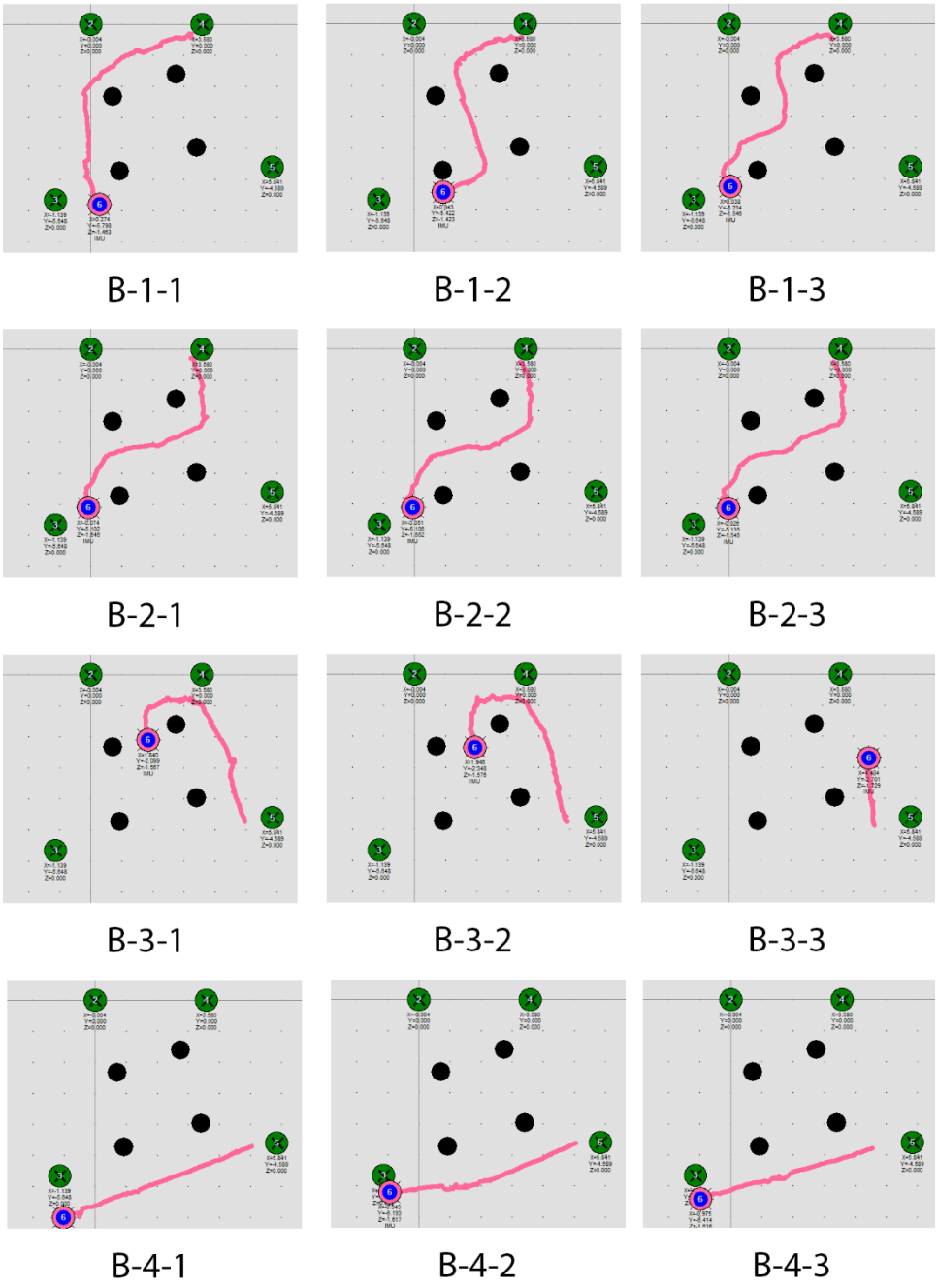


Figure 6-6. Group B 1-4 experiment results. Using trained neural network with LiDAR data and compass heading to navigate the robot in the lab.

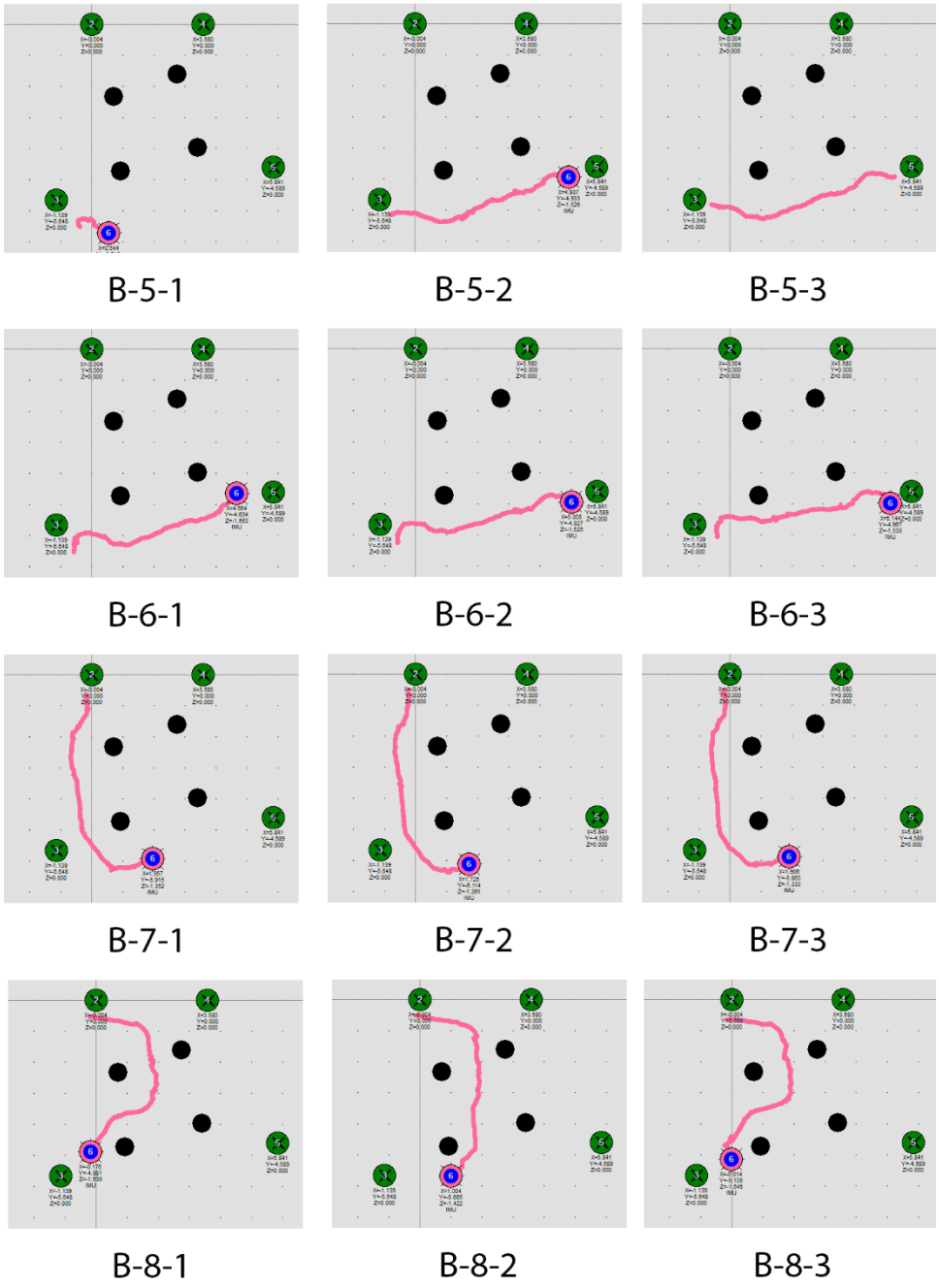


Figure 6-7. Group B 5-8 experiment results. Using trained neural network with LiDAR data and compass heading to navigate the robot in the lab.

The real-world testing of the neural network, which was trained and validated in a simulation, provided valuable insights into how well the model generalized to a physical environment. As shown in the previous Figures, the robot's actual path closely matches its simulated trajectory. Relying solely on LiDAR data, it can effectively avoid obstacles, and once compass readings are added, it gains the ability to navigate specifically towards the north. In the simulation world, the robot has no boundaries and can travel north indefinitely. However, in the real-world setting, once the robot reaches the northernmost point, it cannot determine the correct course of action. If the indoor environment were large enough, this issue would not be as noticeable.

The next parts provide an examination of how effectively the simulation-trained neural network performed in real-world conditions. The analysis covers metrics such as navigation accuracy, obstacle avoidance success rate, collision rate, and heading adjustment accuracy, as well as the influence of varying test parameters. Collectively, these findings offer insights into the strengths and limitations of the system when transitioning from a controlled simulation environment to a more unpredictable physical setting.

6.5.1 Navigation Accuracy

A central goal of the neural network was to guide the robot northward while avoiding obstacles along the way. To measure how effectively it accomplished this, researchers gathered positional data from the super beacons, enabling them to compare the robot's actual path against the ideal northward trajectory. This comparison provided a clear picture of whether the network's decisions aligned with its training objectives.

In the Webots simulation, the robot displayed a high degree of accuracy in following its intended route. It routinely stayed on a straight path toward the north, deviating only slightly even when obstacles were not part of the test scenario. The controlled nature of the simulation—where sensor data is inherently noise-free and environmental factors are carefully defined—contributed to these near-perfect performances.

Upon transitioning to real-world conditions, the robot continued to show that it could maintain an overall northward heading, although some small deviations were noted.

Analysis revealed that most of these deviations stemmed from minor inaccuracies in the IMU's heading data, as well as occasional noise in the LiDAR readings. On average, the robot was able to remain within roughly 10 degrees of true north during the majority of tests, but the gap widened in situations where multiple obstacles were clustered close together. In those instances, the robot faced more frequent or abrupt course corrections, amplifying any minor errors in sensor input, and causing more pronounced deviations.

Together, these findings underscore that while the neural network performed well in clean, controlled simulations, the transition to a real-world environment inevitably introduced additional complexities. Slight variations and noise in sensor outputs, along with the need to navigate physically diverse obstacles, presented challenges beyond what the simulation was designed to replicate. In cases where the robot needed to make frequent heading adjustments, the accumulated effect of real-world sensor noise became more apparent. Nonetheless, the relatively small magnitude of deviations suggests that the network was largely successful in generalizing its learned behaviours from a virtual setting to an actual lab setting.

6.5.2 Obstacle Avoidance Success Rate

Obstacle avoidance stood as another critical functionality of the neural network, requiring the robot to rely on its LiDAR sensor to detect potential hazards and decide on the most effective route to circumvent them. In the Webots simulation, the robot's performance in this regard was nearly flawless. Because the simulated LiDAR data was highly accurate and devoid of real-world noise, the robot had little difficulty identifying obstacles and promptly making either a left or right turn to evade them. This reliable sensor input, coupled with a well-trained network, led to an almost perfect avoidance record in the virtual tests.

In contrast, real-world trials revealed a slightly lower success rate of around 90%. During the remaining 10% of trials, the robot either made contact with an obstacle or veered too close for comfort. Close examination of these incidents indicated that noisy LiDAR data often played a role, as reflections, ambient conditions, or intricate object shapes caused the

sensor to report spurious readings. Additionally, any delay in processing this data meant the network had less time to react, which could be detrimental when the robot encountered an obstacle at a short distance. These findings highlight that while the LiDAR-based navigation was robust overall, the real-world environment introduced a level of complexity that was not entirely captured in the simulation.

When investigators looked more closely, they found that the complexity of obstacle configurations in the lab environment also contributed to the observed drop in performance. The neural network—initially trained on cleaner, more structured scenarios—occasionally struggled when confronted with layouts that required more nuanced decision-making. Although the robot was still able to avoid most collisions, the instances of failure served as clear evidence that ongoing improvements in sensor filtering, processing speed, and network architecture could help close this gap. With more targeted training and further calibration of LiDAR pre-processing, it appears that the avoidance success rate could be pushed even closer to the levels observed in the simulation phase.

6.5.3 Collision Rate

The collision rate emerged as a straightforward indicator of how often the robot failed to avert obstacles during testing. In simulation, this value remained at zero, demonstrating that the neural network, supplied with pristine LiDAR data, was fully capable of identifying and navigating around obstructions in real time. This flawless performance, however, stemmed heavily from the controlled and idealized nature of the virtual environment, where sensor noise and other unpredictability's were absent.

When the robot was brought into the real world, the collision rate rose to approximately 7% of tests. Although still relatively low, these collisions illuminated the challenges that accompany real-world deployments. In many of these instances, the LiDAR sensor readings arrived with noise or slight delays, leading the network to miss an obstacle detection window or choose an incorrect turning direction. Because real-time decisions were required, even a small gap between sensor input and neural network output was enough to undermine the robot's collision avoidance.

These findings underlined the gap between simulation-ready models and their behavior in real-life conditions. While the network's architecture and training data were sufficient for a noise-free virtual space, they did not fully account for sensor inconsistencies, uneven lighting, or the complexities that arise in an uncontrolled environment. Closing this gap likely requires either incorporating realistic noise into the simulation phase or collecting more real-world training data, so that the network learns to cope with the full spectrum of messy inputs it will inevitably encounter outside the lab.

6.5.4 Heading Adjustment Accuracy

Heading adjustment accuracy played a central role in ensuring that the robot maintained its intended northward bearing. Although the robot relied primarily on the IMU for directional feedback, the neural network ultimately decided when and how sharply the robot should turn, making it crucial to evaluate how well these two components worked together. In the Webots simulation, the robot showed little trouble with heading corrections. The simulation provided exact heading data that left virtually no room for error, so once the robot oriented itself north, it remained steady on that course with minimal deviations.

In real-world testing, heading adjustments remained largely accurate but were not entirely devoid of imperfections. The IMU itself continued to offer consistent data, reflecting the fact that hardware inaccuracies were not the main source of any issues. Instead, the neural network occasionally introduced a delay in responding to the IMU data, especially after the robot turned to avoid obstacles. These delays became more apparent in scenarios involving closely spaced obstacles, where the robot had to execute multiple turns in rapid succession. On average, the heading correction lag was about two to three seconds, a period in which the robot sometimes continued on a suboptimal trajectory before fully realigning toward the north.

Although these delays did not significantly hamper the robot in a controlled environment, they underscore the difference between a noise-free simulation and the variability of real-world conditions. In more dynamic or crowded settings, even a brief pause in heading

adjustment could have compounding effects, leading to more frequent near-collisions, or requiring the robot to make additional corrections. This highlights the importance of continued fine-tuning. Whether by refining the neural network's architecture, further calibrating the IMU integration, or simulating the effects of quick-turn environments during training, targeted improvements could help the robot make faster, more precise heading corrections under real-world conditions.

6.5.5 Impact of Test Variations

A key variation in the test setup involved changing the robot's initial heading. Rather than always starting in the same orientation, the robot was placed facing different directions at the onset of each trial. Despite these changes, the robot demonstrated a consistent ability to correct its course and move northward. Its robust performance under these conditions suggests that the neural network effectively generalized beyond the single starting orientation it experienced in the simulation. By drawing on IMU data, the robot could quickly detect that it was not aiming north and adjust accordingly, reinforcing the idea that the network had internalized the goal of orienting toward the north regardless of how it began.

Another major factor was the density of obstacles in the environment. While the robot handled sparse configurations with relative ease, more obstacles introduced higher demands on the decision-making process. Each time an obstacle loomed, the network had to process the LiDAR input and decide whether to turn left, turn right, or continue straight. In settings where obstacles were densely placed, these decision points cropped up more frequently, testing the network's ability to maintain smooth, accurate navigation. The robot's overall success rate dropped by around 15% in these scenarios, reflecting the challenges of quickly integrating multiple streams of sensor data when every turn might lead to another obstacle. This outcome indicated that the network struggled under more complex conditions and could stand to gain from further training in environments that replicate the cluttered layouts found in the real world.

A final variation involved introducing artificial noise into the sensor data, deliberately simulating real-world imperfections the robot might face, from slight misreads in the LiDAR sensor to jitter in IMU outputs. While the neural network had proven its robustness in cleaner conditions, its performance showed a marked decline once noise was added. Collisions increased by 10–15%, underscoring the network's vulnerability to imperfect data. These results highlight the importance of preparing the model for less-than-ideal inputs, whether by incorporating noisy sensor data during training or deploying more advanced filtering algorithms. By doing so, it might be possible to reduce the network's susceptibility to unpredictable readings and keep the collision rate in line with its performance in calmer, noise-free settings.

6.5.6 Summary of Results

Overall, the robot performed well in the real-world tests, successfully navigating towards the north, and avoiding most obstacles. However, the introduction of real-world conditions, including sensor noise and environmental variability, led to some degradation in performance compared to the simulation. Key areas for improvement include handling noisy sensor data, reducing delays in heading adjustments, and improving the network's performance in environments with high obstacle density.

6.6 Debugging and Fine-Tuning

Based on the results and analysis from the real-world tests, several areas have been identified where the neural network's performance can be improved to better handle real-world conditions. This section outlines the debugging and fine-tuning efforts that were undertaken during testing, as well as potential strategies for future improvements. The focus is on addressing issues such as sensor noise, decision-making delays, and improving the network's robustness in environments with high obstacle density.

6.6.1 Addressing Sensor Noise

A major hurdle observed in the real-world tests stemmed from sensor noise, particularly in the LiDAR sensor data. Under simulated conditions, the LiDAR readings were consistently accurate and free of distortions, but the physical environment introduced a range of confounding factors. Fluctuations in ambient light, reflective surfaces that skewed distance measurements, and even interference from nearby objects all contributed to unreliable or noisy data. As a result, the neural network occasionally received misleading distance or position information, leading to miscalculations such as failing to register an obstacle in time or choosing a suboptimal turn direction when confronted with complex layouts. These real-world challenges underscored the gap between clean simulation conditions and the messy reality of actual sensor feeds.

The initial efforts to address this problem focused on pre-processing and threshold adjustments. By applying additional filters to the raw LiDAR data, the development team aimed to dampen sudden spikes or anomalies that could trigger false readings. A moving average filter proved particularly helpful, smoothing out erratic distance values that arose from momentary reflections or sensor errors. At the same time, the threshold levels for obstacle detection underwent careful recalibration. The idea was to ensure that only objects of genuine concern appeared in the sensor stream that the neural network processed. Minor or fleeting artifacts—like a brief reflection caused by a shiny surface—needed to be excluded so as not to provoke unnecessary manoeuvres. Together, these adjustments improved data clarity and cut down on the rate of false positives, though they did not entirely eliminate the problem of noise.

Looking forward, there are multiple avenues for strengthening the network's resilience to imperfect sensor data. One promising strategy involves incorporating noisy data directly into the training phase. By injecting artificial noise or simulated sensor distortions into the dataset, the neural network can learn to differentiate between reliable signals and spurious readings. This enhanced training regime helps the model internalize robust decision-making, even when the sensor stream is cluttered with inaccuracies. The concept is that repeated

exposure to flawed inputs teaches the network to discount anomalies and place greater weight on data points that are consistent and meaningful.

In tandem with improved training methods, more advanced filtering algorithms hold the potential to further refine the information fed into the neural network. Techniques like the Kalman filter or other adaptive noise reduction methods can dynamically track and predict sensor states, then use those predictions to filter out erratic data points. This approach is especially advantageous if the robot is operating in an environment where sensor readings fluctuate unpredictably, as a well-tuned filter can help the robot recognize the difference between typical sensor variability and genuine obstacles. By combining these advanced filtering solutions with targeted training strategies, the system can continue to move closer to the high reliability seen in simulation, making the robot more adept at navigating complex, noisy real-world scenarios.

6.6.2 Improving Decision-Making Speed

A noticeable issue that arose during testing was the occasional delay in the robot's decision-making process, especially in situations where the robot came upon multiple obstacles in rapid succession. Even a slight delay could have serious consequences, as the robot might not turn or stop in time to avoid a collision or inefficiency. These moments made it clear that while the neural network was accurate enough to recognize obstacles and plan paths, its speed of execution sometimes lagged behind the demands of the real-world environment.

One of the initial steps taken to address these delays involved optimizing the neural network itself. By reducing the complexity of certain computationally heavy layers—particularly in the convolutional architecture—developers were able to lighten the network's workload. For instance, cutting down the number of filters in key convolutional layers helped the network process incoming data more quickly, without causing a dramatic loss in the accuracy of obstacle detection or heading management. In parallel, the onboard system's software stack was refactored to allow sensor data processing to occur simultaneously where possible. This

parallelization prevented any single thread of execution from becoming a bottleneck, ensuring that LiDAR readings, IMU updates, and the neural network's inference were handled in tandem whenever feasible.

Looking ahead, further efficiency gains can be achieved by pruning the neural network. Pruning involves systematically removing neurons or even entire layers that have little to no impact on the final outcome. With fewer redundant computations, the model can run in less time and still maintain most of its predictive power. Another method for accelerating inference is model quantization, whereby floating-point weights are converted to fixed-point representations. Quantized models typically consume less memory and can execute faster on many hardware platforms, because smaller, fixed-point arithmetic operations require fewer resources. By combining pruning and quantization, the system could become leaner and more responsive, allowing the robot to make swift decisions even in obstacle-dense scenarios.

6.6.3 Enhancing Performance in High Obstacle Density Environments

One notable finding from the real-world tests was a clear decline in the robot's performance whenever it encountered environments filled with closely spaced obstacles. Because each new obstacle demanded a near-immediate decision—whether to turn, slow down, or keep moving—the neural network sometimes failed to respond swiftly enough, resulting in increased collisions or less efficient paths. This shortfall became especially pronounced in dense layouts, where rapid, consecutive manoeuvres left little margin for error.

During the debugging phase, one of the principal interventions involved expanding the training data. Rather than relying solely on the simulation scenarios with moderate obstacle spacing, additional obstacles were added to the Webots environment to create more congested spaces. By exposing the neural network to these busier settings, it learned to recognize and react to scenarios that demanded more frequent turning and more nuanced obstacle avoidance. At the same time, the team fine-tuned several hyperparameters,

including the learning rate and batch size. Lowering the learning rate slowed down the training process, giving the network more time to internalize complex patterns and ensuring it could capture subtler cues about where and when to turn in high-density environments.

Looking toward future improvements, one potential avenue lies in giving the network greater contextual awareness. Incorporating memory-based architectures, such as LSTM layers, could enable the neural model to remember past decisions and sensor readings. This temporal memory would be particularly beneficial in crowded conditions, where the decision to turn left or right might depend not just on the immediate obstacle, but also on the sequence of obstacles encountered moments before. Another strategy for bolstering the robot's obstacle-handling abilities is to augment its sensory inputs. By integrating additional data streams—possibly from stereo cameras or depth sensors—the robot would gain richer information about the environment, potentially spotting objects at different angles or further distances. Such comprehensive input would allow the network to plan a path that anticipates future manoeuvres rather than making each turn in isolation, ultimately enhancing both responsiveness and collision avoidance in tight quarters.

6.6.4 Adjusting Heading Control Mechanisms

Occasional delays in the robot's heading adjustments were evident during the real-world tests, especially when the robot encountered obstacles in rapid succession. Although the robot generally maintained its heading toward the north, these short lags occasionally left it traveling briefly off-course before realigning.

Early debugging efforts focused on recalibrating the IMU to ensure the device consistently provided accurate heading information. Even small errors in orientation data could compound over time, causing the robot to misjudge how much it needed to turn in order to return to a northward path. In addition, the team refined the robot's heading control parameters, allowing it to make quicker, more fluid course corrections. This adjustment helped reduce the period the robot spent off its target heading following an obstacle-avoidance manoeuvre.

To further address this issue, future iterations could incorporate a predictive heading model, one that estimates the robot's upcoming orientation based on both current speed and the proximity of obstacles. Such a model would enable the neural network to proactively schedule heading corrections, cutting down the delay between obstacle avoidance and realignment. Beyond predictive strategies, adopting adaptive control algorithms could allow the robot to dynamically modify its turning rate based on how crowded the environment is. In high-density situations, a faster turn rate would help the robot steer clear of obstacles more quickly and regain its northward heading without undue delay.

6.6.5 Future Directions and Long-Term Improvements

A promising direction for future development lies in the incorporation of unsupervised learning mechanisms, which would allow the robot to adapt to unfamiliar environments without the need for manually labelled training data. In this approach, the robot would explore its surroundings independently and gather insights from its own experiences, learning to identify new patterns or anomalies as it navigates. By actively refining its understanding of the environment through direct interaction, the robot could become more versatile, continually adjusting its behavior when it encounters unexpected obstacles or layout changes that were never explicitly presented during training. This ability to learn "on the fly" could prove indispensable in real-world settings, where no two environments are exactly the same and conditions can change rapidly.

Another avenue for enhancing the robot's navigation capabilities involves the use of transfer learning, where a pre-trained model—originally trained on simulation or other datasets—is fine-tuned using real-world data. Because much of the network's foundational knowledge remains intact, the time and computational resources required for additional training can be significantly reduced. Moreover, this approach helps align the model's parameters with the specific characteristics of the real-world environment, such as minor sensor noise, variable lighting, or unique obstacle arrangements. By periodically collecting new data and retraining parts of the network, the robot's ability to make accurate decisions can steadily improve,

ultimately narrowing the gap between simulation-based performance and real-world effectiveness.

Looking to the horizon, incorporating dynamic elements into the testing environment would also represent a major step forward. Rather than navigating static layouts, the robot would encounter moving obstacles or changing terrain, capturing a degree of realism that is often absent in controlled laboratory tests. This shift would not only challenge the robot's current navigation strategies but also illuminate how well the neural network handles scenarios that require real-time adjustments under pressure. Over time, continued exposure to such dynamic conditions would guide refinements to both the training pipeline and the network architecture, ensuring the robot remains robust and adaptable when confronted with new and evolving challenges in the field.

Through debugging and fine-tuning, significant improvements were made to the neural network's real-world performance. Adjustments to sensor data processing, decision-making speed, and handling complex environments all contributed to more reliable navigation and obstacle avoidance. However, future work remains to optimize the network further and enhance its capabilities for even more challenging real-world applications.

6.7 Conclusion and Future Work

This chapter has described the transition of a neural network—initially trained in a simulated environment—to a functioning real-world application, using a physical robot outfitted with additional hardware like super beacons and an IMU. The primary purpose of this implementation was to assess how effectively the network could adapt to a physical, static environment after being refined in Webots, focusing on tasks such as maintaining a northward heading and avoiding obstacles. In large part, the tests revealed that the neural network met these objectives, demonstrating a reliable capacity for autonomous navigation even when faced with the natural variability of real-world conditions. Still, the chapter also spotlighted a range of challenges, including sensor noise, delayed decision-making, and difficulties in dense obstacle layouts, all of which pinpoint areas where further enhancements are necessary.

An overview of key findings centres on how the network generalized from simulation to real-world conditions. Although the robot was introduced to additional complexities like noisy sensor data and unpredictable surroundings, it navigated effectively, maintaining its heading and steering clear of obstacles. In terms of performance metrics, the robot generally stayed on course, diverging only slightly when sensor inaccuracies came into play or when the environment presented frequent obstructions. Its 90% success rate in obstacle avoidance suggested that the LiDAR data, though imperfect in the real world, was still interpreted well enough to prevent the majority of collisions. The collision rate itself, roughly 7%, served as a reminder that simulation training cannot capture every real-world variable, and heading adjustment data showed that while the robot usually oriented itself northward promptly, certain scenarios—especially when obstacles were tightly packed—led to brief lags in correcting its course.

Despite the successful aspects, the real-world tests also laid bare some difficulties. Sensor noise, a constant factor in any real-world scenario, posed a real threat to the robot's accuracy, especially when LiDAR readings arrived late or were distorted by reflections. Occasional delays in decision-making, particularly in the presence of multiple obstacles, sometimes caused the robot to make last-minute course corrections or even collide with objects. Further, the challenges in densely cluttered environments illustrated the network's need for faster and more nuanced decision-making. These observations make it clear that additional refinement, both in how sensor data is pre-processed and how decisions are made, will be vital for the next stage of development.

Looking ahead, the chapter suggests multiple long-term improvement strategies to bolster the robot's capabilities. One of the principal approaches involves enhancing how sensor data is processed. Advanced filtering techniques, ranging from Kalman filters to machine learning-based noise reduction, could diminish the impact of inconsistent LiDAR and IMU readings. Another opportunity lies in training with real-world data. By using transfer learning to fine-tune the model on physical-world inputs, the network gains a better foothold on conditions it did not experience in simulation. This expansion into dynamic environments—introducing moving obstacles or shifting terrain—would also push the network's adaptability, as would

the integration of unsupervised learning methods that allow the robot to explore unfamiliar settings without extensive labelled data.

Further refinement of the neural network's architecture can accelerate decision-making and improve performance in high-stress contexts. Techniques like pruning and quantization can streamline the model, making it leaner and faster to run, while memory-based layers like LSTMs might help the system maintain context over time, which is especially valuable in dynamic or obstacle-heavy scenarios. Ultimately, testing in more elaborate settings—those with greater complexity, tighter spacing, or moving elements—will provide new insights into how the network manages under real-world pressures, paving the way for more advanced and resilient autonomous robotics applications.

To sum up, while this physical deployment of the neural network demonstrated the approach's viability and underscored the significant improvements possible, it also exposed the critical challenges engineers must overcome when transferring simulation-trained models into tangible, real-world implementations. The collective lessons from these experiments offer a strong platform for continued research and development, guiding the design of even more capable neural networks and robotic control systems, and edging closer to truly robust and efficient autonomy.

Chapter 7. First-Year SLAM Development: Foundations for Real-World Robotics

7.1 Robot design

In previous work, a two-wheel centred differential robot was build and equipped with a LiDAR sensor to test SLAM process in a static environment. The whole system diagram is shown below, Xbee modules are used for wireless communication between robot and PC. The robot chassis was rapid prototyped within the Stephenson Building the final mobile robot has a diameter of 30cm and height of 26cm.

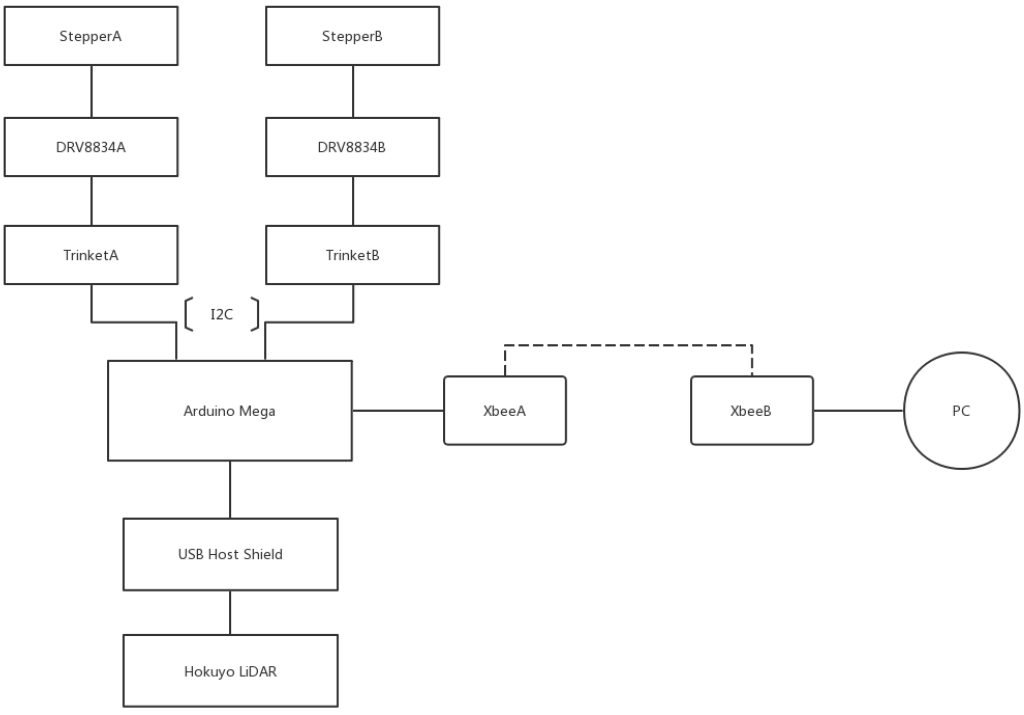


Figure 7-1. System block diagram.

7.2 Methodology

This project will use a scan-matching method called normal distribution transform (NDT) and wheel odometry to get a precious estimated pose, and then build a grid-based map. The grid map can recognize the static and dynamic objects then detect the dynamic item.

1. Normal Distribution Transform (NDT)

The goal of NDT is to find the pose of the second scan that maximises the likelihood that the points of the second scan lie on the first scan surface. The process of NDT can be divided into several steps:

1) Subdivide the space occupied by the reference scan into grid cells
2) Put all the points into the box and each cell should contain at least three points
3) Calculate the probability density function (PDF) of each cell and the PDFs describe the surface shape of the cell. Assuming a D-dimensional normal random process generate the locations of the reference scan points, the likelihood of having measured $\vec{x}$ is:

$$\mathrm{p}(\vec{x}) = \frac{1}{(2\pi)^{\frac{D}{2}}\sqrt{|\Sigma|}} \exp\left(-\frac{(\vec{x} - \vec{\mu})^T \Sigma^{-1} (\vec{x} - \vec{\mu})}{2}\right) \tag{12}$$

Where $\vec{\mu}$ and Σ are the mean vector and covariance matrix of the reference scan surface points. The factor $((2\pi)^{D/2}\sqrt{|\Sigma|})^{-1}$ is the scale of the function and maybe replaced by a constant c_0 for some practical purposes. And the mean and covariance matrix are calculated as:

$$\vec{\mu} = \frac{1}{m}\sum_{k=1}^{m} \vec{y}_k \tag{13}$$

$$\Sigma = \frac{1}{m-1}\sum_{k=1}^{m}(\vec{y}_k - \vec{\mu})(\vec{y}_k - \vec{\mu})^T \quad (14)$$

Where $\vec{y}_{k=1,\ldots,m}$ are the reference scan points in the cell.

4) Initialize the optimised parameters $\vec{p}$ by zero or by using wheel odometry data:

$$\vec{p} = \left(t_x, t_y, \phi\right)^T \quad (15)$$

5) Scan alignment by using the spatial mapping T between two scans which is given by:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} cos\phi & -sin\phi \\ sin\phi & cos\phi \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} t_x \\ t_y \end{pmatrix} \quad (16)$$

6) Map the reconstructed point into the coordinate frame of the first scan based on the optimised parameters and decide the corresponding normal distributions of each mapped point. Then calculate the score of the parameter by evaluating the distribution for each point and summing the results. It is defined as:

$$score(p) = \sum_i \exp\left(\frac{-(x_i' - q_i)^T \sum_i^{-1}(x_i' - q_i)}{2}\right) \quad (17)$$

Where:

x_i : The reconstructed point of sample i of the second scan.

x_i' : The point x_i mapped into the first scan coordinate frame based on the parameter p, it is $x_i' = T(x_i, p)$

$q_i, \sum_i$: The mean and covariance matrix of the corresponding normal distribution of point x_i'.

7) The final step is optimizing the score to get a new parameter estimate by using Newton's Algorithm. Repeat step 6 until a convergence criterion is met.

The goal of Newton's Algorithm in this step is to minimize– $score$. It iteratively uses the parameters $p = (p_i)^T$ to minimize a function f and solves the equation:

$$H\,\Delta p = -g \tag{18}$$

Where g is the gradient vector of f with entries

$$g_i = \frac{\partial f}{\partial p_i} \tag{19}$$

Moreover, H is the Hessian matrix of f with entries

$$H_{ij} = \frac{\partial f}{\partial p_i \partial p_j} \tag{20}$$

The increment Δp is added to the current estimate:

$$p \leftarrow p + \Delta p \tag{21}$$

Apply this algorithm to the function– $score$, the Hessian and gradient are the summands of partial derivatives in the score equation in step 6. We can rewrite:

$$q = x_i' - q_i \tag{22}$$

So one summand s of – $score$ can be given by:

$$s = -exp\frac{-q^T \sum^{-1} q}{2} \tag{23}$$

The entries g_i of the gradient can be written:

$$g_i = -\frac{\partial s}{\partial p_i} = -\frac{\partial s}{\partial q}\frac{\partial q}{\partial p_i} = q^T \sum^{-1} \frac{\partial q}{\partial p_i}\, exp\, \frac{-q^T \sum^{-1} q}{2} \tag{24}$$

In this equation, $\partial q / \partial p_i$ is given by the Jacobian matrix J_T of T:

$$J_T = \begin{bmatrix} 1 & 0 & -xsin\emptyset - ycos\emptyset \\ 0 & 1 & xcos\emptyset - ysin\emptyset \end{bmatrix} \quad (25)$$

The entries H_{ij} of the Hessian matrix H are given by:

$$H_{ij} = -\frac{\partial s}{\partial p_i \partial p_j} =$$

$$-exp\frac{-q^T\sum^{-1}q}{2}\left((-q^T\sum^{-1}\frac{\partial q}{\partial p_i})(-q^T\sum^{-1}\frac{\partial q}{\partial p_j}) + (-q^T\sum^{-1}\frac{\partial^2 q}{\partial p_i \partial p_j}) + (-\frac{\partial q^T}{\partial p_j}\sum^{-1}\frac{\partial q}{\partial p_i})\right) (26)$$

addition, the second-order derivatives used in this equation are given:

$$\frac{\partial^2 q}{\partial p_i \partial p_j} = \begin{cases} \begin{bmatrix} -xcos\emptyset + ysin\emptyset \\ -xsin\emptyset - ycos\emptyset \end{bmatrix} & if\ i = j = 3 \\ \begin{bmatrix} 0 \\ 0 \end{bmatrix} & otherwise \end{cases} \quad (27)$$

The most important part is the grid parameters. A large grid may result in low precision while a small grid needs high memory and long computation time. The initial value has a little correlation with the result and even the initial value error is large, it can still be corrected [139, 140].

2. Occupancy Grid Mapping

Occupancy grid mapping is a popular algorithm in the robot SLAM process. It discretizes the environment into several cells, which are binary random variables that represent occupied and free space. Then the probability distribution of the map is given by the cells, given the sensor data $z_{1:t}$ and the poses $x_{1:t}$ of the sensor, the estimated map is[141, 142]:

$$p(m \mid z_{1:t}, x_{1:t}) = \prod_i p(m_i \mid z_{1:t}, x_{1:t}) \quad (28)$$

Then use Bayes rule to obtain:

$$p(m_i \mid z_{1:t}, x_{1:t}) = \frac{p(z_t \mid m_i, z_{1:t-1}, x_{1:t}) p(m_i \mid z_{1:t-1}, x_{1:t})}{p(z_t \mid z_{1:t-1}, x_{1:t})} \quad (29)$$

Make Markov assumption that z_t is independent from $x_{1:t-1}$ and $z_{1:t-1}$, this leads to

$$p(m_i \mid z_{1:t}, x_{1:t}) = \frac{p(z_t \mid m_i,\ x_t)\ p(m_i \mid z_{1:t-1}, x_{1:t-1})}{p(z_t \mid z_{1:t-1}, x_{1:t})} \tag{30}$$

Apply Bayes rule again for $p(z_t \mid m_i, x_t)$ and get

$$p(z_t \mid m_i, x_t) = \frac{p(m_i \mid z_t, x_t)\ p(z_t \mid x_t)}{p(m_i \mid x_t)} \tag{31}$$

Combine the two equations above and furthermore assume that x_t is not relative with m_i if there is no observationz_t. In addition, get:

$$p(m_i \mid z_{1:t}, x_{1:t}) = \frac{p(m_i \mid z_t, x_t)p(m_i \mid z_{1:t-1}, x_{1:t-1})}{p(m_i)p(z_t \mid z_{1:t-1}, x_{1:t})} \tag{32}$$

Each cell is defined as a binary variable and use $\neg m_i$ to represent the opposite event:

$$p(\neg m_i \mid z_{1:t}, x_{1:t}) = \frac{p(\neg m_i \mid z_t, x_t)p(\neg m_i \mid z_{1:t-1}, x_{1:t-1})}{p(\neg m_i)p(z_t \mid z_{1:t-1}, x_{1:t})} \tag{33}$$

By computing the ratio of both probabilities:

$$\frac{p(m_i \mid z_{1:t}, x_{1:t})}{p(\neg m_i \mid z_{1:t}, x_{1:t})} = \frac{p(m_i \mid z_t, x_t)p(m_i \mid z_{1:t-1}, x_{1:t-1})p(\neg m_i)}{p(\neg m_i \mid z_t, x_t)p(\neg m_i \mid z_{1:t-1}, x_{1:t-1})p(m_i)} \tag{34}$$

Finally, add $p(\neg m_i) = 1 - p(m_i)$ in and get:

$$\frac{p(m_i \mid z_{1:t}, x_{1:t})}{1 - p(m_i \mid z_{1:t}, x_{1:t})} = \frac{p(m_i \mid z_t, x_t)}{1 - p(m_i \mid z_t, x_t)} \frac{1 - p(m_i)}{p(m_i)} \frac{p(m_i \mid z_{1:t-1}, x_{1:t-1})}{1 - p(m_i \mid z_{1:t-1}, x_{1:t-1})} \tag{35}$$

The occupancy grid-mapping algorithm uses a log odds ratio to represent the occupancy by $l_{t,i}$:

$$l_{t,i} = \log \frac{p(m_i \mid z_{1:t}, x_{1:t})}{1 - p(m_i \mid z_{1:t}, x_{1:t})} \tag{36}$$

The log odds representation can avoid the numerical instabilities for profanities around one or zero. Use this method at other parts and get

$$l_{inv,i} = log \frac{p(m_i \mid z_t, x_t)}{1 - p(m_i \mid z_t, x_t)} \tag{37}$$

$$l_0 = log \frac{p(m_i)}{1 - p(m_i)} \tag{38}$$

the equation XX becomes:

$$l_{t,i} = l_{t-1,i} + l_{inv,i} - l_0 \tag{39}$$

Moreover, the occupancy can be recovered from the log odds ratio by:

After that, the problem of updating the map is becoming an addition and subtraction question that can be simply solved in the program.

7.3 SLAM in MATLAB

The raw LiDAR data is decoded into distance & range values in MATLAB, then transferred into point clouds and put into a 200×200 matrix. Each cloud point may increase the grid values up to a maximum value to 1. At the same time, set the grids that between occupancy grid and robot position as free grids which decrease the value and set the minimum value to

-1. At this point, it is easy to get the current map with scan data. The whole map can be the sum of each current map with correct position. The normal way is using wheel odometry to get the robot position and it is often effective in short distance. However, because of the noise including encoder errors and wheel slip, the cumulative error may directly affect the localization process as well as the whole map build. A correlation matrix with two continuously maps is proposed to reduce the estimated position error. The correlation matrix is plotted in a 4D graph with X, Y, theta and score value. X, Y, Theta are the position of the robot while the score is the sum of each grid in the current map multiplied by the same grid in the global map. The score values are linearly mapped to the colours indicated in a colour map matrix. An example of the correlation matrix 4D diagram is shown in figure 6-2. The correlation matrix can show the best-estimated position of current map. The red points indicate the best match and are concentrated near the position estimated from wheel odometry.

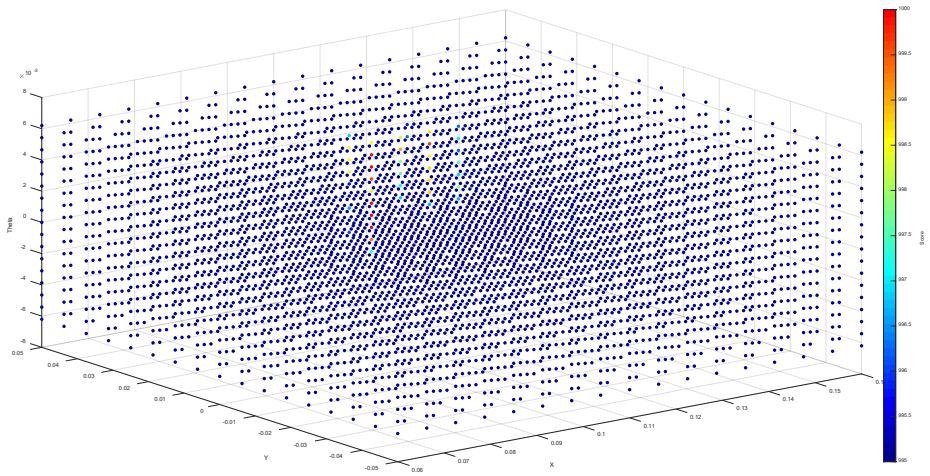


Figure 7-2. Diagram of the 4D data in correlation matrix. There are four values: x,y,z and scores.

The experiment was tested in the school office in a $7m \times 3.5m$ room and several wood boards are used to half the room to reduce the complexity of the environment. The mobile robot movement instruction is controlled under a MATLAB GUI with move and rotate control buttons. For example, W button means the robot will move forward for 2 seconds with $5.5cm/s$ speed and A button will make the robot rotate in left hand side for 2 seconds at

$0.353 rad/s$ speed. An example of the data collected is shown in figure 6-3. After each motion, the LiDAR sensor will scan for once then build the maps. Wheel odometry and built in MATLAB scan matching function are used as reference groups for comparison with this approach.

Fields	map	range	wopose	smpose	cmpose	cmap	cmmap
1	200x200 double	1x682 double	[0,0,0]	[0,0,0]	[0,0,0]	200x200 double	200x200 double
2	200x200 double	1x682 double	[0.4400,0,0]	[0.4309,0.0187,-0.0132]	[0.4233,0.0167,0.0062]	200x200 double	200x200 double
3	200x200 double	1x682 double	[0.8800,0,0]	[0.8721,0.0233,-0.0080]	[0.8800,0.0500,0.0083]	200x200 double	200x200 double
4	200x200 double	1x682 double	[1.3200,0,0]	[1.3380,0.0075,0.0279]	[1.3367,0.0667,0.0103]	200x200 double	200x200 double
5	200x200 double	1x682 double	[1.7600,0,0]	[1.3557,-0.4250,-0.1979]	[1.7600,0.1000,0.0083]	200x200 double	200x200 double
6	200x200 double	1x682 double	[1.7600,0,-1.0600]	[1.7705,0.0023,-1.0638]	[1.7600,0.1000,-1.0476]	200x200 double	200x200 double
7	200x200 double	1x682 double	[1.7600,0,-2.1200]	[1.7532,0.0118,-2.1104]	[1.7600,0.1000,-2.1159]	200x200 double	200x200 double
8	200x200 double	1x682 double	[1.7600,0,-3.1800]	[1.7477,-0.0170,3.1015]	[1.7600,0.0833,-3.1924]	200x200 double	200x200 double
9	200x200 double	1x682 double	[1.7600,0,-4.2400]	[1.7600,0,2.0432]	[1.7433,0.1000,-4.2379]	200x200 double	200x200 double
10	200x200 double	1x682 double	[1.7600,0,-3.1800]	[1.7568,0.0059,3.1003]	[1.7600,0.0833,-3.1924]	200x200 double	200x200 double

Figure 7-3. Experiment data includes different estimated position values and the map.

The positions from wheel odometry, scan matching, and correlation matrix are given in the data matrix. Finally, the global maps are built with these estimated robot position. Figure 6-4 shows the grid maps with three different methods. Although the wheel odometry can give the basic profile of the environment, the cumulative errors make the walls thicker. Theoretically, the scan-matching function in MATLAB should give more precise result than wheel odometry. However, the matching process may have bad matches when the score is quite low. Despite the proposed correlation matrix costs more time than the scan matching function in MATLAB, it has better performance due to every step matching. The correlation matrix can have better results with smaller step size but will cost more calculation time.

(a)

(b)

(c)

Figure 7-4. Three grid maps with different type position. (a): wheel odometry (b): MATLAB scan matching function (c): correlation matrix (this work).

With the information of this static environment and the robot position, robot navigation problem can be basically solved in a traditional algorithm approached way. The information of the objects is displayed in the map, and it is easy for the robot to avoid it.

7.4 Determination of moving objects

Dynamic objects can be tracked and represented by using several continuous scan figures. One test in a small environment, the robot stands still and scan for several times. In the scan process, put one object into the environment and then move it once. Figure 6-5 shows the 12 scans in this test and the dynamic object first appears in the third picture with grey colour. Then the grids that represent this object become white after two scans. From picture 6 to 8, the object moves to another position and the previous position grids colour change to grey and then turn black. The new position grids become grey and then turn white. Picture 8 to 12 indicate that the object is removed from the environment. Comparison of the global map between frames indicate which objects are static and which are dynamic. This means that is it easy to build up a map of the static objects in the environment. The problem then becomes predicting where the dynamic objects will be moving to.

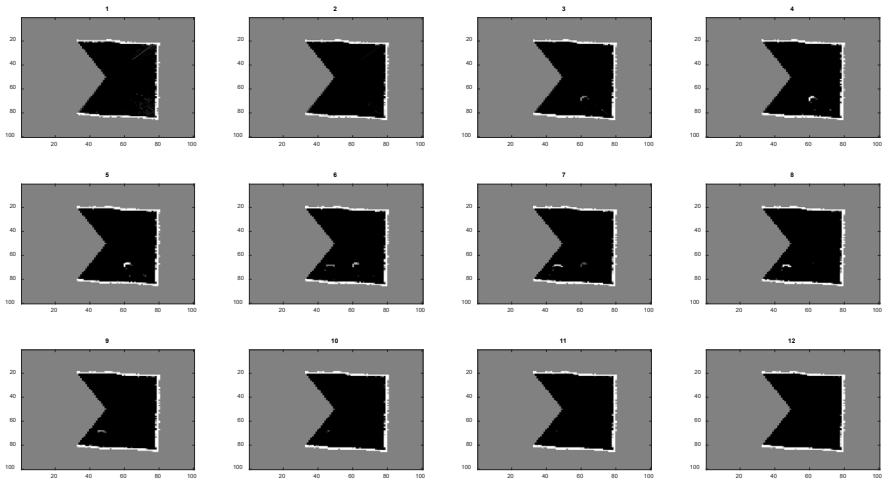


Figure 7-5. Twelve scans for detecting dynamic object.

Chapter 8. Conclusion and future work

8.1 Summary of Findings

The primary objective of this research was to develop, train, and implement a CNN capable of navigating a robot through a static environment while avoiding obstacles. Initially, the neural network was trained using data from simulations conducted in the Webots environment, employing MATLAB's Deep Learning Toolbox. This training process leveraged sensor data such as LiDAR readings and compass heading information to create a model capable of processing real-time inputs and guiding the robot autonomously.

In the later stages of this research, the trained network was deployed onto a real robot for testing in a physical environment. The robot was equipped with additional hardware, including super beacons for accurate positional tracking and an IMU for obtaining real-time compass heading. These components were critical in ensuring the robot could receive real-time data similar to what was used during the simulation phase, thus enabling a reliable comparison between simulated and real-world performance.

The results showed that the neural network was effective at navigating towards a predefined direction (north) and avoiding static obstacles in a controlled lab environment. While the overall performance was successful, the tests also highlighted specific challenges—such as sensor noise, decision-making delays, and increased difficulty in environments with higher obstacle density—that need to be addressed in future iterations. Despite these challenges, the project succeeded in demonstrating the viability of transferring a neural network trained in a simulated environment to a real-world robotic platform.

8.2 Contribution to Knowledge

This research makes pivotal contributions to the fields of robotics, machine learning, and autonomous navigation by examining how a neural network, initially trained in a virtual

environment, can be effectively deployed in real-world conditions. In doing so, it addresses the complexities that arise when moving from a controlled simulation to scenarios where noise, variability, and physical limitations inevitably come into play. By demonstrating a successful transfer of a CNN from simulation to a physical robot, the study offers insights that can considerably streamline future development and reduce the resource-intensive process of on-site training. Just as importantly, it highlights both the benefits and potential pitfalls of using convolutional neural networks for tasks that require swift and accurate decision-making.

One of the central findings involves simulation-to-reality transfer, an area critical to modern robotics development. Through rigorous testing, this research establishes that a CNN, trained solely in a simulation platform like Webots, can still perform at a high level in the real world. This is an important step forward because training a robot purely in a physical setting can be prohibitively expensive, time-consuming, and occasionally risky for the hardware. The ability to replicate or approximate physical conditions in simulation—and then carry over a trained model with minimal adjustments—signals that future projects may be able to expedite their own development pipelines. This efficiency, in turn, enables quicker iteration cycles and lower overhead, as the bulk of training can be safely conducted in a virtual space before any physical deployment. Just as crucially, the data gathered here supports the notion that simulated models can handle genuine sensor information, including noisy LiDAR readings and drifting IMU orientations, without wholesale retraining or extensive redesign.

Alongside this fundamental simulation-to-reality insight, the project furthers knowledge on autonomous navigation using convolutional neural networks. Traditionally, CNNs are known for their efficacy in processing images; however, this work shows that the same architectures can adapt to other data formats like one-dimensional LiDAR output. By feeding real-time sensor data into the CNN, the robot was able to learn decision patterns for obstacle avoidance and directional maintenance, effectively deciding when to turn or move forward. The rapid inference required for such decision-making underscores how CNNs, particularly those optimized for speed, can hold their own in applications beyond static image classification. This points to vast potential for CNN-driven solutions in scenarios

demanding quick reflexes and complex navigational logic, especially given the network's ability to capture nuanced patterns within the sensor data.

A further contribution lies in how the robot integrated multiple types of real-time data. The combination of LiDAR, super beacons, and an IMU provided a panoramic view of the robot's environment. LiDAR readings gave precise distance measurements, the beacons supplied positional tracking, and the IMU indicated heading, each contributing its own layer of information. The study details how these data sources were synchronized and processed in real time so that the CNN could leverage an up-to-date view of the robot's surroundings. This is significant because many robotic systems rely on just one or two sensors; the success of a more multi-faceted approach shows the benefits of fusing different data streams to enhance situational awareness. By weaving together these complementary sensor feeds, the robot achieved greater accuracy in collision avoidance, smoother route planning, and an overall higher level of autonomy.

Inevitable real-world challenges also emerged, most notably sensor noise, timing bottlenecks, and environmental volatility. The presence of reflective surfaces, interference from other objects, and even slight variations in lighting can taint LiDAR or IMU data. In this research, such noise often led to minor but potentially impactful inaccuracies, prompting the team to implement refined pre-processing methods and noise reduction strategies. In tandem, the neural network architecture was optimized to shrink decision-making latency, ensuring the robot made timely turns in environments featuring frequent and dense obstacles. By documenting these trials and their fixes, the work serves as a candid account of the obstacles that arise once a system leaves the relative safety of a simulation. In doing so, it provides a reliable resource for others looking to replicate or extend these findings without having to rediscover solutions through trial and error.

Finally, the research puts forward a forward-looking roadmap for continued exploration and improvement. By presenting specific ways the system could be evolved—such as introducing advanced filtering algorithms like Kalman filters, training on real-world data through transfer learning, or enhancing adaptability with unsupervised learning—it offers a springboard for tackling even more demanding environments. The approach could be extended to dynamic

settings where obstacles move, terrains change, or additional sensors such as cameras or depth sensors come into play. Through recommended methods like network pruning, quantization, or the inclusion of memory-based layers like LSTMs, the robot could become both faster and more adaptable in real-time operation.

Taken together, these elements form a comprehensive framework that future researchers and practitioners can draw upon. The successful deployment of a CNN-trained robot underscores how simulation can be harnessed to efficiently develop and test autonomous systems. It also underscores the intricacies of real-world deployment, where sensor noise, rapid decision-making requirements, and environmental variability present formidable yet surmountable hurdles. By mapping out these challenges and providing tangible strategies for overcoming them, this research lights a path toward increasingly robust and flexible autonomous robots, ready to tackle complex navigational tasks in both controlled and unpredictable domains.

8.3 Meeting the Research Objectives

The first objective of this research involved the development and training of a CNN that could handle sensor data inputs and produce real-time navigation decisions. Achieving this required gathering a substantial volume of data from the Webots simulation, which served as a controlled environment for generating realistic LiDAR scans, compass headings, and various obstacle configurations. In this phase, the Deep Learning Toolbox in MATLAB played a pivotal role, allowing the researchers to experiment with different architectures and hyperparameters. Iterations focused on ensuring that the network was neither too large to process data quickly nor too small to capture the complexities of navigating toward a specific direction—north—while detecting and avoiding obstacles. By the conclusion of this stage, the CNN successfully demonstrated its capacity to map sensor inputs to effective movement commands, establishing a strong basis for the next phases of testing and validation.

Once the CNN had been thoroughly trained, the second key objective centred on validating it within the Webots simulation environment. This step allowed for systematic testing under carefully controlled scenarios that replicated various obstacle densities and placements.

During these simulated trials, the network showed impressive accuracy, maintaining a consistent northward trajectory and reliably bypassing obstructions in real time. Because the simulation could be manipulated to introduce new challenges—such as sudden changes in obstacle layout or variations in the robot's starting position—the research team gained valuable insights into both the strengths and potential vulnerabilities of the CNN before taking on the complexities of the physical world. By refining the network's parameters and confirming its readiness in simulation, the team ensured that any significant issues could be caught early, thereby easing the eventual transition to real-world tests.

The third key objective involved deploying the trained CNN onto a physical robot in a real-world setting to assess its navigation performance under actual operating conditions. Here, the robot was equipped with a LiDAR sensor, an IMU, and super beacons, all of which fed the network with continuous streams of data. These tests revealed that the CNN could indeed interpret real-time sensor information, make quick navigation decisions, and maintain its northward heading—capabilities initially demonstrated only in simulation. Although real-world experiments introduced complications like sensor noise, unpredictable reflections, and slight delays in processing, the robot's overall performance remained robust. Minor deviations from the planned path highlighted areas needing further refinement, such as filtering out LiDAR inaccuracies and reducing decision lag. Nevertheless, the core achievement of autonomous navigation and obstacle avoidance in a static, physical environment underscored the network's capacity to bridge the gap between controlled simulation and tangible applications.

Finally, an overall assessment of the research objectives confirmed that they were successfully met. By showcasing strong performance first in the Webots simulation and then in physical trials, the CNN demonstrated its ability to function as an effective navigation controller. It combined multi-sensor data—LiDAR readings for obstacle detection, IMU inputs for heading information, and beacon-derived positional data—to produce stable, real-time decisions. The precise alignment of simulation results with real-world outcomes highlighted the quality of the training process, as well as the adaptability of the chosen CNN architecture. While the challenges uncovered in real-world testing, such as sensor noise and occasional processing delays, underscore the necessity of ongoing refinements, the

overarching conclusion is that the objectives outlined at the start—training a CNN for autonomous navigation and successfully deploying it in a physical environment—were realized.

8.4 Conclusion and Future Directions

This research underscores the substantial promise of CNNs for real-world robotic navigation, adding to an expanding body of knowledge in autonomous systems. By progressing from a purely simulation-based environment to actual physical tests, the study has revealed both the strengths and the remaining obstacles in deploying neural networks for genuine applications. Such insights are crucial for building robots that can operate safely and effectively in varied, real-world settings where sensor noise, environmental unpredictability, and hardware limitations become pressing concerns.

Looking ahead, one avenue for further advancement is the integration of additional sensory inputs. While the current system already leverages LiDAR and IMU data, richer modalities such as camera feeds or depth sensors could enhance the robot's environmental perception. For example, visual data might allow the robot to recognize and categorize different types of obstacles instead of treating all obstructions as equivalent, leading to more strategic and context-aware navigation. Depth sensors could provide layered, volumetric representations of the surrounding space, guiding more complex manoeuvres in cluttered or multi-tiered environments. By combining various data streams, future systems could achieve greater reliability and adaptability, especially when confronted with unexpected or intricate terrains.

Another significant direction involves training strategies that move beyond the present reliance on simulation. Although this research successfully demonstrated that models trained solely in a simulated environment could transfer to real-world conditions, further refinements—such as transfer learning—may improve performance under challenging or

unpredictable scenarios. By collecting new data directly from physical tests and using it to fine-tune the CNN, developers can bridge any remaining gap between simulated and authentic conditions more effectively. This real-world training could be conducted iteratively, where the model refines itself after each round of physical testing, thereby enhancing its resilience to sensor noise, environmental changes, and diverse obstacle layouts.

Additionally, broadening the scope of testing to include dynamic elements stands out as a clear next step. So far, trials have largely focused on static obstacles in relatively controlled environments. However, many real-world applications—ranging from autonomous delivery robots navigating city streets to drones flying through crowded spaces—involve moving obstacles, rapidly changing terrain, and other mutable factors. By incorporating dynamic features into test environments, future research can determine whether CNNs can adapt in real-time to changes such as shifting pedestrian traffic, vehicles in motion, or unpredictable weather. These tests would offer a more realistic lens through which to evaluate the neural network's robustness, providing valuable feedback on whether it can predict, track, and respond to moving threats and opportunities.

Altogether, this research has served as a strong foundational demonstration of how neural networks can be trained in simulation yet still excel in tangible robotics applications. Such successes emphasize the growing feasibility of deploying machine learning solutions directly in the field without prohibitive costs or risks during the training phase. As sensor options continue to multiply, and as researchers refine network architectures and training methodologies, the potential for more sophisticated, flexible, and robust autonomous systems becomes increasingly apparent. In this sense, the project not only showcases immediate benefits but also lays the groundwork for exciting future innovations in autonomous navigation.

Reference

[1] N. Zherdev, M. Kurenkov, K. Belikova, and D. Tsetserukou, "SwipeBot: DNN-based Autonomous Robot Navigation among Movable Obstacles in Cluttered Environments," in *2023 IEEE 97th Vehicular Technology Conference (VTC2023-Spring)*, 20-23 June 2023 2023, pp. 1-5, doi: 10.1109/VTC2023-Spring57618.2023.10200601.

[2] Z. Wang, C. He, and Z. Miao, "Navigation control of mobile robot based on fuzzy neural network," in *2023 3rd Asia-Pacific Conference on Communications Technology and Computer Science (ACCTCS)*, 25-27 Feb. 2023 2023, pp. 98-102, doi: 10.1109/ACCTCS58815.2023.00070.

[3] A. Rocchi, Z. Wang, and Y. J. Pan, "A Practical Vision-Aided Multi-Robot Autonomous Navigation using Convolutional Neural Network," in *2023 IEEE 6th International Conference on Industrial Cyber-Physical Systems (ICPS)*, 8-11 May 2023 2023, pp. 1-6, doi: 10.1109/ICPS58381.2023.10128041.

[4] H. Zheng, Q. Wang, and J. Ji, "Navigation line extraction based on image processing for weeding robot," in *IECON 2022 – 48th Annual Conference of the IEEE Industrial Electronics Society*, 17-20 Oct. 2022 2022, pp. 1-5, doi: 10.1109/IECON49645.2022.9968644.

[5] X. Ruan, C. Lin, J. Huang, and Y. Li, "Obstacle avoidance navigation method for robot based on deep reinforcement learning," in *2022 IEEE 6th Information Technology and Mechatronics Engineering Conference (ITOEC)*, 4-6 March 2022 2022, vol. 6, pp. 1633-1637, doi: 10.1109/ITOEC53115.2022.9734337.

[6] B. Kaleci, K. Turgut, and H. Dutagaci, "2DLaserNet: A deep learning architecture on 2D laser scans for semantic classification of mobile robot locations," *Engineering Science and Technology, an International Journal,* vol. 28, p. 101027, 2022/04/01/ 2022, doi: https://doi.org/10.1016/j.jestch.2021.06.007.

[7] J. Cai *et al.*, "An Initial Alignment Method of Mobile Robot Rotational Strapdown Inertial Navigation System Based on RWNN Observer," in *2022 34th Chinese Control and Decision Conference (CCDC)*, 15-17 Aug. 2022 2022, pp. 5583-5587, doi: 10.1109/CCDC55256.2022.10034085.

[8] A. Burguera, "Robust Underwater Visual Graph SLAM using a Simanese Neural Network and Robust Image Matching," in *VISIGRAPP (4: VISAPP)*, 2022, pp. 591-598.

[9] K. Yang *et al.*, "Research on UWB/IMU location fusion algorithm based on GA-BP neural network," in *2021 40th Chinese Control Conference (CCC)*, 26-28 July 2021 2021, pp. 8111-8116, doi: 10.23919/CCC52363.2021.9549463.

[10] N. D. Toan and K. G. Woo, "Mapless Navigation with Deep Reinforcement Learning based on The Convolutional Proximal Policy Optimization Network," in *2021 IEEE International Conference on Big Data and Smart Computing (BigComp)*, 17-20 Jan. 2021 2021, pp. 298-301, doi: 10.1109/BigComp51126.2021.00063.

[11] N. D. Toan and K. Gon-Woo, "Environment Exploration for Mapless Navigation based on Deep Reinforcement Learning," in *2021 21st International Conference on Control, Automation and Systems (ICCAS)*, 12-15 Oct. 2021 2021, pp. 17-20, doi: 10.23919/ICCAS52745.2021.9649893.

[12] P. H. M. Piratelo *et al.*, "Convolutional neural network applied for object recognition in a warehouse of an electric company," in *2021 14th IEEE International Conference on Industry Applications (INDUSCON)*, 2021: IEEE, pp. 293-299.

[13] N. A. Pham, L. A. Nguyen, and X. T. Truong, "Socially aware robot navigation framework: Social activities recognition using deep learning techniques," in *2021 8th NAFOSTED Conference on Information and Computer Science (NICS)*, 21-22 Dec. 2021 2021, pp. 381-385, doi: 10.1109/NICS54270.2021.9701551.

[14] R. S. Kulikov, N. V. Masalkova, O. V. Glukhov, N. I. Petukhov, and A. A. Chugunov, "Using Neural Networks to Navigate Robots Among Obstacles," in *2021 Systems of Signals Generating and Processing in the Field of on Board Communications*, 16-18 March 2021 2021, pp. 01-05, doi: 10.1109/IEEECONF51389.2021.9416023.

[15] M. Medvedev, A. K. Farhood, and D. Brosalin, "Development of the Neural-Based Navigation System for a Ground-Based Mobile Robot," in *2021 7th International Conference on Mechatronics and Robotics Engineering (ICMRE)*, 3-5 Feb. 2021 2021, pp. 35-40, doi: 10.1109/ICMRE51691.2021.9384825.

[16] G. Guo, Z. Ding, Q. Zhang, Y. Zhang, Y. Hu, and Z. Tao, "Trajectory tracking of tracked underwater dredging robot based on adaptive neural network," in *2021 International Conference on Security, Pattern Analysis, and Cybernetics （SPAC)*, 18-20 June 2021 2021, pp. 347-352, doi: 10.1109/SPAC53836.2021.9540007.

[17] S. A. K. Diane and E. A. Lesiv, "Neural-based Visual Odometry Trained in a Virtual Enviroment for a Mobile Robot Navigation," in *2021 14th International Conference Management of large-scale system development (MLSD)*, 27-29 Sept. 2021 2021, pp. 1-5, doi: 10.1109/MLSD52249.2021.9600249.

[18] C. Cheng and Y. Chen, "A Neural Network based Mobile Robot Navigation Approach using Reinforcement Learning Parameter Tuning Mechanism," in *2021 China Automation Congress (CAC)*, 22-24 Oct. 2021 2021, pp. 2600-2605, doi: 10.1109/CAC53003.2021.9728061.

[19] R. Kulikov, N. Masalkova, O. Glukhov, A. Chugunov, and V. Semenov, "An Ultrasonic Vision based on a Neural Network for Navigating Robots Through Obstacles," in *2021 3rd International Youth Conference on Radio Electronics, Electrical and Power Engineering (REEPE)*, 11-13 March 2021 2021, pp. 1-5, doi: 10.1109/REEPE51337.2021.9387993.

[20] K. Chaudhary, G. Lal, A. Prasad, V. Chand, S. Sharma, and A. Lal, "Obstacle Avoidance of a Point-Mass Robot using Feedforward Neural Network," in *2021 3rd Novel Intelligent and Leading Emerging Sciences Conference (NILES)*, 23-25 Oct. 2021 2021, pp. 210-215, doi: 10.1109/NILES53778.2021.9600550.

[21] E. Nwaonumah and B. Samanta, "Deep Reinforcement Learning For Visual Navigation of Wheeled Mobile Robots," in *2020 SoutheastCon*, 28-29 March 2020 2020, pp. 1-8, doi: 10.1109/SoutheastCon44009.2020.9249654.

[22] X. Li, K. Guo, T. Jia, and X. Zhang, "Visual perception and navigation of security robot based on deep learning," in *2020 IEEE International Conference on Mechatronics and Automation (ICMA)*, 13-16 Oct. 2020 2020, pp. 1216-1221, doi: 10.1109/ICMA49215.2020.9233649.

[23] H. Li, Y. Mao, W. You, B. Ye, and X. Zhou, "A neural network approach to indoor mobile robot localization," in *2020 19th International Symposium on Distributed Computing and Applications for Business Engineering and Science (DCABES)*, 16-19 Oct. 2020 2020, pp. 66-69, doi: 10.1109/DCABES50732.2020.00026.

[24] R. V. Hoa, T. D. Chuyen, N. T. Lam, T. N. Son, N. D. Dien, and V. T. T. Linh, "Reinforcement Learning based Method for Autonomous Navigation of Mobile Robots in Unknown Environments," in *2020 International Conference on Advanced Mechatronic Systems (ICAMechS)*, 10-13 Dec. 2020 2020, pp. 266-269, doi: 10.1109/ICAMechS49982.2020.9310129.

[25] T. D. Dung and G. Capi, "Neural Network based 3D Mapping Using Depth Image Camera," in *2020 International Conference on Image Processing and Robotics (ICIP)*, 6-8 March 2020 2020, pp. 1-5, doi: 10.1109/ICIP48927.2020.9367338.

[26] H. M. Yudha, T. Dewi, N. Hasana, P. Risma, Y. Oktarini, and S. Kartini, "Performance Comparison of Fuzzy Logic and Neural Network Design for Mobile Robot Navigation," in *2019 International Conference on Electrical Engineering and Computer Science (ICECOS)*, 2-3 Oct. 2019 2019, pp. 79-84, doi: 10.1109/ICECOS47637.2019.8984577.

[27] W. Yuan, S. Zhang, T. Wu, and B. Dai, "Moving Direction Predicting with Deep Neural Networks for Mobile Robots," in *2019 11th International Conference on Intelligent Human-Machine Systems and Cybernetics (IHMSC)*, 24-25 Aug. 2019 2019, vol. 2, pp. 312-317, doi: 10.1109/IHMSC.2019.10167.

[28] S. Thrun, "Probabilistic robotics," *Communications of the ACM,* vol. 45, no. 3, pp. 52-57, 2002.

[29] O. J. Woodman, "An introduction to inertial navigation," University of Cambridge, Computer Laboratory, 2007.

[30] A. Elfes, "Using occupancy grids for mobile robot perception and navigation," *Computer,* vol. 22, no. 6, pp. 46-57, 1989.

[31] J. Fuentes-Pacheco, J. Ruiz-Ascencio, and J. M. Rendón-Mancha, "Visual simultaneous localization and mapping: a survey," *Artificial intelligence review,* vol. 43, pp. 55-81, 2015.

[32] N. Sünderhauf, "Robust optimization for simultaneous localization and mapping," Technischen Universitat Chemnitz, 2012.

[33] T. Bailey and H. Durrant-Whyte, "Simultaneous localization and mapping (SLAM): part II," *IEEE robotics & automation magazine,* vol. 13, no. 3, pp. 108-117, 2006, doi: 10.1109/MRA.2006.1678144.

[34] H. Durrant-Whyte and T. Bailey, "Simultaneous localization and mapping: part I," *IEEE Robotics & Automation Magazine,* vol. 13, no. 2, pp. 99-110, 2006, doi: 10.1109/mra.2006.1638022.

[35] M. Montemerlo, S. Thrun, D. Koller, and B. Wegbreit, "FastSLAM: A factored solution to the simultaneous localization and mapping problem," *Aaai/iaai,* vol. 593598, 2002.

[36] G. Grisetti, C. Stachniss, and W. Burgard, "Improved techniques for grid mapping with rao-blackwellized particle filters," *IEEE transactions on Robotics,* vol. 23, no. 1, pp. 34-46, 2007.

[37] G. Grisettiyz, C. Stachniss, and W. Burgard, "Improving grid-based slam with rao-blackwellized particle filters by adaptive proposals and selective resampling," in *Proceedings of the 2005 IEEE international conference on robotics and automation*, 2005: IEEE, pp. 2432-2437.

[38] G. Grisetti, R. Kümmerle, C. Stachniss, and W. Burgard, "A tutorial on graph-based SLAM," *IEEE Intelligent Transportation Systems Magazine,* vol. 2, no. 4, pp. 31-43, 2010.

[39] R. Siegwart, I. R. Nourbakhsh, and D. Scaramuzza, *Introduction to autonomous mobile robots*. MIT press, 2011.

[40] M. Montemerlo, S. Thrun, D. Koller, and B. Wegbreit, "FastSLAM 2.0: An improved particle filtering algorithm for simultaneous localization and mapping that provably converges," in *IJCAI*, 2003, vol. 3, pp. 1151-1156.

[41] J. M. Santos, D. Portugal, and R. P. Rocha, "An evaluation of 2D SLAM techniques available in robot operating system," in *2013 IEEE international symposium on safety, security, and rescue robotics (SSRR)*, 2013: IEEE, pp. 1-6.

[42] S. Kohlbrecher, O. Von Stryk, J. Meyer, and U. Klingauf, "A flexible and scalable SLAM system with full 3D motion estimation," in *2011 IEEE international symposium on safety, security, and rescue robotics*, 2011: IEEE, pp. 155-160.

[43] W. Hess, D. Kohler, H. Rapp, and D. Andor, "Real-time loop closure in 2D LIDAR SLAM," in *2016 IEEE international conference on robotics and automation (ICRA)*, 2016: IEEE, pp. 1271-1278.

[44] Ó. M. Mozos and â. M. Mozos, *Semantic labeling of places with mobile robots*. Springer, 2010.

[45] L. Yuan, K. C. Chan, and C. G. Lee, "Robust semantic place recognition with vocabulary tree and landmark detection," in *Workshop of IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2011.

[46] E. Fazl-Ersi and J. K. Tsotsos, "Energy minimization via graph cuts for semantic place labeling," in *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2010: IEEE, pp. 5947-5952.

[47] A. Pronobis, O. Martinez Mozos, B. Caputo, and P. Jensfelt, "Multi-modal semantic place classification," *The International Journal of Robotics Research,* vol. 29, no. 2-3, pp. 298-320, 2010.

[48] A. Rottmann, O. M. Mozos, C. Stachniss, and W. Burgard, "Semantic place classification of indoor environments with mobile robots using boosting," in *AAAI*, 2005, vol. 5, pp. 1306-1311.

[49] O. M. Mozos, C. Stachniss, and W. Burgard, "Supervised learning of places from range data using adaboost," in *Proceedings of the 2005 IEEE international conference on robotics and automation*, 2005: IEEE, pp. 1730-1735.

[50] L. Shi and S. Kodagoda, "Towards generalization of semi-supervised place classification over generalized Voronoi graph," *Robotics and Autonomous Systems,* vol. 61, no. 8, pp. 785-796, 2013.

[51] B. Kaleci, Ç. M. Şenler, H. Dutağacı, and O. Parlaktuna, "A probabilistic approach for semantic classification using laser range data in indoor environments," in *2015 international conference on advanced robotics (ICAR)*, 2015: IEEE, pp. 375-381.

[52] P. Newman and J. Leonard, "Pure range-only sub-sea SLAM," in *2003 IEEE International Conference on Robotics and Automation (Cat. No. 03CH37422)*, 2003, vol. 2: Ieee, pp. 1921-1926.

[53] R. Kümmerle, G. Grisetti, H. Strasdat, K. Konolige, and W. Burgard, "g 2 o: A general framework for graph optimization," in *2011 IEEE international conference on robotics and automation*, 2011: IEEE, pp. 3607-3613.

[54] M. Kaess, A. Ranganathan, and F. Dellaert, "iSAM: Incremental smoothing and mapping," *IEEE Transactions on Robotics,* vol. 24, no. 6, pp. 1365-1378, 2008.

[55] J. McCormac, A. Handa, A. Davison, and S. Leutenegger, "Semanticfusion: Dense 3d semantic mapping with convolutional neural networks," in *2017 IEEE International Conference on Robotics and automation (ICRA)*, 2017: IEEE, pp. 4628-4635.

[56] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature,* vol. 521, no. 7553, pp. 436-444, 2015.

[57] K. Alex, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional networks," in *volume-1; pages-1097–1105; NIPS'12 Proceedings of the 25th International Conference on Neural Information Processing System*.

[58] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 779-788.

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

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

[61] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *nature,* vol. 323, no. 6088, pp. 533-536, 1986.

[62] A. Graves, A.-r. Mohamed, and G. Hinton, "Speech recognition with deep recurrent neural networks," in *2013 IEEE international conference on acoustics, speech and signal processing*, 2013: Ieee, pp. 6645-6649.

[63] Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *IEEE transactions on neural networks,* vol. 5, no. 2, pp. 157-166, 1994.

[64] T. Mikolov, M. Karafiát, L. Burget, J. Cernocký, and S. Khudanpur, "Recurrent neural network based language model," in *Interspeech*, 2010, vol. 2, no. 3: Makuhari, pp. 1045-1048.

[65] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation,* vol. 9, no. 8, pp. 1735-1780, 1997.

[66] H. Sak, A. W. Senior, and F. Beaufays, "Long short-term memory recurrent neural network architectures for large scale acoustic modeling," 2014.

[67] F. A. Gers, J. Schmidhuber, and F. Cummins, "Learning to forget: Continual prediction with LSTM," *Neural computation,* vol. 12, no. 10, pp. 2451-2471, 2000.

[68] Y. Gal and Z. Ghahramani, "A theoretically grounded application of dropout in recurrent neural networks," *Advances in neural information processing systems,* vol. 29, 2016.

[69] J. Kober, J. A. Bagnell, and J. Peters, "Reinforcement learning in robotics: A survey," *The International Journal of Robotics Research,* vol. 32, no. 11, pp. 1238-1274, 2013.

[70] L.-C. Chen, G. Papandreou, I. Kokkinos, K. Murphy, and A. L. Yuille, "Semantic image segmentation with deep convolutional nets and fully connected crfs," *arXiv preprint arXiv:1412.7062,* 2014.

[71] A. Giusti *et al.*, "A machine learning approach to visual perception of forest trails for mobile robots," *IEEE Robotics and Automation Letters,* vol. 1, no. 2, pp. 661-667, 2015.

[72] R. A. Newcombe *et al.*, "Kinectfusion: Real-time dense surface mapping and tracking," in *2011 10th IEEE international symposium on mixed and augmented reality*, 2011: Ieee, pp. 127-136.

[73] V. Mnih *et al.*, "Human-level control through deep reinforcement learning," *nature,* vol. 518, no. 7540, pp. 529-533, 2015.

[74] T. Schaul, "Prioritized Experience Replay," *arXiv preprint arXiv:1511.05952,* 2015.

[75] D. Silver *et al.*, "Mastering the game of Go with deep neural networks and tree search," *nature,* vol. 529, no. 7587, pp. 484-489, 2016.

[76] V. Mnih, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602,* 2013.

[77] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347,* 2017.

[78] R. Lowe, Y. I. Wu, A. Tamar, J. Harb, O. Pieter Abbeel, and I. Mordatch, "Multi-agent actor-critic for mixed cooperative-competitive environments," *Advances in neural information processing systems,* vol. 30, 2017.

[79] A. Kendall *et al.*, "End-to-end learning of geometry and context for deep stereo regression," in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 66-75.

[80] M. Bojarski, "End to end learning for self-driving cars," *arXiv preprint arXiv:1604.07316,* 2016.

[81] P. Dayan and G. E. Hinton, "Feudal reinforcement learning," *Advances in neural information processing systems,* vol. 5, 1992.

[82] K. Cho, "On the Properties of Neural Machine Translation: Encoder-decoder Approaches," *arXiv preprint arXiv:1409.1259,* 2014.

[83] S. Shalev-Shwartz, S. Shammah, and A. Shashua, "On a formal model of safe and scalable self-driving cars," *arXiv preprint arXiv:1708.06374,* 2017.

[84] J. Yosinski, J. Clune, Y. Bengio, and H. Lipson, "How transferable are features in deep neural networks?," *Advances in neural information processing systems,* vol. 27, 2014.

[85] M. Satyanarayanan, "The emergence of edge computing," *Computer,* vol. 50, no. 1, pp. 30-39, 2017.

[86] R. Mur-Artal and J. D. Tardós, "Orb-slam2: An open-source slam system for monocular, stereo, and rgb-d cameras," *IEEE transactions on robotics,* vol. 33, no. 5, pp. 1255-1262, 2017.

[87] J. Engel, T. Schöps, and D. Cremers, "LSD-SLAM: Large-scale direct monocular SLAM," in *European conference on computer vision*, 2014: Springer, pp. 834-849.

[88] D. Fox, W. Burgard, and S. Thrun, "Markov localization for mobile robots in dynamic environments," *Journal of artificial intelligence research,* vol. 11, pp. 391-427, 1999.

[89] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770-778.

[90] M. Bojarski *et al.*, "End to end learning for self-driving cars," *arXiv preprint arXiv:1604.07316,* 2016.

[91] C. Cadena *et al.*, "Past, present, and future of simultaneous localization and mapping: Toward the robust-perception age," *IEEE Transactions on robotics,* vol. 32, no. 6, pp. 1309-1332, 2016.

[92] J. Redmon, "Yolov3: An incremental improvement," *arXiv preprint arXiv:1804.02767,* 2018.

[93] J. Preskill, "Quantum computing in the NISQ era and beyond," *Quantum,* vol. 2, p. 79, 2018.

[94] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE,* vol. 105, no. 12, pp. 2295-2329, 2017.

[95] S. Ross, G. Gordon, and D. Bagnell, "A reduction of imitation learning and structured prediction to no-regret online learning," in *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, 2011: JMLR Workshop and Conference Proceedings, pp. 627-635.

[96] C. Finn, P. Abbeel, and S. Levine, "Model-agnostic meta-learning for fast adaptation of deep networks," in *International conference on machine learning*, 2017: PMLR, pp. 1126-1135.

[97] R. R. Murphy, "Human-robot interaction in rescue robotics," *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews),* vol. 34, no. 2, pp. 138-153, 2004.

[98] M. Armbrust *et al.*, "A view of cloud computing," *Communications of the ACM,* vol. 53, no. 4, pp. 50-58, 2010.

[99] H.-K. Lo, M. Curty, and K. Tamaki, "Secure quantum key distribution," *Nature Photonics,* vol. 8, no. 8, pp. 595-604, 2014.

[100] S. Lloyd, "Universal quantum simulators," *Science,* vol. 273, no. 5278, pp. 1073-1078, 1996.

[101] H. Wang, S. Yu, and H. Xiang, "Efficient quantum algorithm for solving structured problems via multistep quantum computation," *Physical Review Research,* vol. 5, no. 1, p. L012004, 2023.

[102] R. Calo, "Robotics and the Lessons of Cyberlaw," *Calif. L. Rev.,* vol. 103, p. 513, 2015.

[103] J. Dai, S. Fazelpour, and Z. Lipton, "Fair machine learning under partial compliance," in *Proceedings of the 2021 AAAI/ACM Conference on AI, Ethics, and Society*, 2021, pp. 55-65.

[104] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems,* vol. 25, 2012.

[105] O. Michel, "Cyberbotics Ltd. Webots™: Professional Mobile Robot Simulation," *International Journal of Advanced Robotic Systems,* vol. 1, no. 1, p. 5, 2004, doi: 10.5772/5618.

[106] "MathWorks(2021). MATLAB - The Language of Technical Computing. [Online]." https://www.mathworks.com/products/MATLAB.html (accessed 2023).

[107] "Webots Reference Manual." https://cyberbotics.com/doc/reference/lidar#wb_lidar_get_range_image (accessed 2022).

[108] G. Klancar, A. Zdesar, S. Blazic, and I. Skrjanc, *Wheeled mobile robotics: from fundamentals towards autonomous systems*. Butterworth-Heinemann, 2017.

[109] B. Siciliano, O. Khatib, and T. Kröger, *Springer handbook of robotics*. Springer, 2008.

[110] A. Rodríguez-Molina, A. Herroz-Herrera, M. Aldape-Pérez, G. Flores-Caballero, and J. A. Antón-Vargas, "Dynamic Path Planning for the Differential Drive Mobile Robot Based on Online Metaheuristic Optimization," *Mathematics,* vol. 10, no. 21, p. 3990, 2022.

[111] R. Mathew and S. S. Hiremath, "Trajectory tracking and control of differential drive robot for predefined regular geometrical path," *Procedia Technology,* vol. 25, pp. 1273-1280, 2016.

[112] N.-T. Tran, T.-D. Ngo, D.-K. Nguyen, P. X. Son, and N. H. Thai, "Mapping and Path Planning for the Differential Drive Wheeled Mobile Robot in Unknown Indoor Environments Using the

Rapidly Exploring Random Tree Method," in *Regional Conference in Mechanical Manufacturing Engineering*, 2021: Springer, pp. 516-527.

[113] T. T. Hoang, N. N. A. Quan, V. C. Thanh, and T. L. T. Dong, "Navigation for Two-Wheeled Differential Mobile Robot in the Special Environment," in *International Conference on Industrial Networks and Intelligent Systems*, 2022: Springer, pp. 167-183.

[114] G. Indiveri, A. Nuchter, and K. Lingemann, "High speed differential drive mobile robot path following control with bounded wheel speed commands," in *Proceedings 2007 IEEE International Conference on Robotics and Automation*, 2007: IEEE, pp. 2202-2207.

[115] F. Arvin, K. Samsudin, and M. A. Nasseri, "Design of a differential-drive wheeled robot controller with pulse-width modulation," in *2009 Innovative Technologies in Intelligent Systems and Industrial Applications*, 2009: IEEE, pp. 143-147.

[116] Y. Chung, C. Park, and F. Harashima, "A position control differential drive wheeled mobile robot," *IEEE Transactions on Industrial Electronics,* vol. 48, no. 4, pp. 853-863, 2001.

[117] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The bulletin of mathematical biophysics,* vol. 5, pp. 115-133, 1943.

[118] M. A. Nielsen, *Neural networks and deep learning*. Determination press San Francisco, CA, USA, 2015.

[119] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE,* vol. 86, no. 11, pp. 2278-2324, 1998.

[120] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 1026-1034.

[121] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Bidirectional encoder representations from transformers," 2016.

[122] L. B. Cesar, M.-Á. Manso-Callejo, and C.-I. Cira, "BERT (Bidirectional Encoder Representations from Transformers) for Missing Data Imputation in Solar Irradiance Time Series," *Engineering Proceedings,* vol. 39, no. 1, p. 26, 2023.

[123] MathWorks(2023). Deep Learning Toolbox (R2023b). [Online] Available: https://uk.mathworks.com/help/deeplearning/index.html

[124] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *The journal of machine learning research,* vol. 15, no. 1, pp. 1929-1958, 2014.

[125] S. Koenig and R. G. Simmons, "Unsupervised learning of probabilistic models for robot navigation," in *Proceedings of IEEE International Conference on Robotics and Automation*, 1996, vol. 3: IEEE, pp. 2301-2308.

[126] X. Gu, J. Luo, and Y. Ma, "Modeling Spatial-Temporal Interactions for Robot Crowd Navigation," in *2022 China Automation Congress (CAC)*, 25-27 Nov. 2022 2022, pp. 3512-3517, doi: 10.1109/CAC57257.2022.10055051.

[127] I. K. Ibraheem and F. H. Ajeil, "Multi-objective path planning of an autonomous mobile robot in static and dynamic environments using a hybrid PSO-MFB optimisation algorithm," *arXiv preprint arXiv:1805.00224,* 2018.

[128] M. M. Bejani and M. Ghatee, "A systematic review on overfitting control in shallow and deep neural networks," *Artificial Intelligence Review,* pp. 1-48, 2021.

[129] P. Baheti. "What is Overfitting in Deep Learning [+10 Ways to Avoid It]." https://www.v7labs.com/blog/overfitting (accessed December 1, 2021).

[130] J. Brownlee. "How to Avoid Overfitting in Deep Learning Neural Networks." https://machinelearningmastery.com/introduction-to-regularization-to-reduce-overfitting-and-improve-generalization-error/ (accessed August 6th, 2019).

[131] C. GOYAL. "Complete Guide to Prevent Overfitting in Neural Networks (Part-1)." https://www.analyticsvidhya.com/blog/2021/06/complete-guide-to-prevent-overfitting-in-neural-networks-part-1/ (accessed June 12th, 2021).

[132] C. Shorten and T. M. Khoshgoftaar, "A survey on image data augmentation for deep learning," *Journal of big data,* vol. 6, no. 1, pp. 1-48, 2019.

[133] A. Y. Ng, "Feature selection, L 1 vs. L 2 regularization, and rotational invariance," in *Proceedings of the twenty-first international conference on Machine learning*, 2004, p. 78.

[134] L. Prechelt, "Early stopping-but when?," in *Neural Networks: Tricks of the trade*: Springer, 2002, pp. 55-69.

[135] R. Kohavi, "A study of cross-validation and bootstrap for accuracy estimation and model selection," in *Ijcai*, 1995, vol. 14, no. 2: Montreal, Canada, pp. 1137-1145.

[136] C. Bishop, "Pattern recognition and machine learning," *Springer google schola,* vol. 2, pp. 35-42, 2006.

[137] T. G. Dietterich, "Ensemble methods in machine learning," in *International workshop on multiple classifier systems*, 2000: Springer, pp. 1-15.

[138] S. J. Pan and Q. Yang, "A survey on transfer learning," *IEEE Transactions on knowledge and data engineering,* vol. 22, no. 10, pp. 1345-1359, 2009.

[139] P. Biber and W. Strasser, "The normal distributions transform: a new approach to laser scan matching," vol. 3, ed, 2003, pp. 2743-2748 vol.3.

[140] M. Magnusson, "The three-dimensional normal-distributions transform: an efficient representation for registration, surface analysis, and loop detection," Örebro universitet, 2009.

[141] S. Thrun, W. Burgard, and D. Fox, *Probabilistic robotics*. Cambridge, Mass.; London: The MIT Press (in English), 2006.

[142] C. Stachniss, *Robotic mapping and exploration*. Berlin: Berlin : Springer, 2009.

[143] "Create Simple Deep Learning Network for Classification." https://uk.mathworks.com/help/deeplearning/ug/create-simple-deep-learning-network-for-classification.html (accessed 2018).

Appendix A. MATLAB code for neural network

- **LiDAR data only**

```
layers = [
   imageInputLayer([1 67 1],"Name","imageinput")
    convolution2dLayer([2 4],8,"Name","conv_1","Padding","same")
   batchNormalizationLayer("Name","batchnorm_1")
   reluLayer("Name","relu_1")
   maxPooling2dLayer([2 2],"Name","maxpool_1","Padding","same","Stride",[2 2])
      convolution2dLayer([2 4],16,"Name","conv_2","Padding","same")
   batchNormalizationLayer("Name","batchnorm_2")
   reluLayer("Name","relu_2")
   maxPooling2dLayer([2 2],"Name","maxpool_2","Padding","same","Stride",[2 2])
    convolution2dLayer([2 4],32,"Name","conv_3","Padding","same")
   batchNormalizationLayer("Name","batchnorm_3")
   reluLayer("Name","relu_3")
      fullyConnectedLayer(3,"Name","fc")
   softmaxLayer("Name","softmax")
   classificationLayer("Name","classoutput")];
plot(layerGraph(layers));
```

- **LiDAR data and compass heading**

```
layers = [
   imageInputLayer([2 67 1],"Name","imageinput")
    convolution2dLayer([2 4],8,"Name","conv_1","Padding","same")
   batchNormalizationLayer("Name","batchnorm_1")
   reluLayer("Name","relu_1")
   maxPooling2dLayer([1 2],"Name","maxpool_1","Padding","same","Stride",[2 2])
      convolution2dLayer([2 4],16,"Name","conv_2","Padding","same")
   batchNormalizationLayer("Name","batchnorm_2")
   reluLayer("Name","relu_2")
   maxPooling2dLayer([1 2],"Name","maxpool_2","Padding","same","Stride",[2 2])
  convolution2dLayer([2 4],32,"Name","conv_3","Padding","same")
```

```
    batchNormalizationLayer("Name","batchnorm_3")
    reluLayer("Name","relu_3")
   fullyConnectedLayer(3,"Name","fc")
    softmaxLayer("Name","softmax")
    classificationLayer("Name","classoutput")];
plot(layerGraph(layers));
```

Appendix B. MATLAB code for data classification

1. Transfer version_2 for Lidar data and compass heading

```
lear;
count = zeros(12,1);      nsquadval = 20;        folder_path = 'D:\Quan\Data\All data for code\Supervisored\All';
chdir(folder_path);
file_info = dir(fullfile(folder_path,'*.mat'));
n_files = length(file_info);
or floop = 1:1:12
   foldername = "F" + floop;
   mkdir(folder_path,foldername)
end
for loop = 1:1:n_files
   if(rem(loop,1000)==0)  % Print loop count to screen
      loop
   end
      filename = "MATLAB_Data_" + loop + ".mat";
   astruct = load(filename);
   bcell = struct2cell(astruct);


   data = bcell[106];
   px = data(1);
   py = data(2);
   pz = data(3);
    cx = data(4);
   cy = data(5);
   cz = data(6);
   rad = atan2(-cx,cy);
   bearing = (rad/3.14159) * 180.0;
   if(bearing<-180)
      bearing = bearing+360;
   elseif(bearing>180)
```

```
        bearing = bearing-360;
    end
        decision = data(7);
    if(bearing>=-nsquadval && bearing<=nsquadval)
        nesw = 1;
    elseif(bearing>nsquadval && bearing<(180-nsquadval))
        nesw = 2;
    elseif(bearing>(-180+nsquadval) && bearing<-nsquadval)
        nesw = 4;
    else
        nesw = 3;
    end
    foldernum = (decision-1)*4+nesw;
        angle_d = linspace(bearing-120,bearing+120,667);
    for aloop = 1:1:667
        if(angle_d(aloop)>180)
            angle_d(aloop) = angle_d(aloop) - 360;
        elseif(angle_d(aloop)<-180)
            angle_d(aloop) = angle_d(aloop) + 360;
        end
    end
    angle = uint8(255 - (abs(angle_d)*255/180)); % White for north, darker either side
    lidar_m = data(8:end);
    lidar = 0 * ([1:667]);
    for dloop = 1:1:667
        if(lidar_m(dloop)<5.6)
            lidar(dloop) = uint8(lidar_m(dloop)*255/5.6);
        else
            lidar(dloop) = uint8(255);
        end
    end
    tempimage = [angle;lidar];
    image = uint8(imresize(tempimage,[2 67]));
    count(foldernum) = count(foldernum) + 1;
    outfilename = "F" + foldernum + "\Data_" + foldernum + "_" + count(foldernum) + ".png";
    imwrite(image,outfilename,'png');
end
```

```
ount = count'
ext program needs to select certain number
of files from each folder at random
mkdir(folder_path,"1")
mkdir(folder_path,"2")
mkdir(folder_path,"3")
 Select required number of data files from each folder
n_datafiles = [500 500 500 500 500 500 500 500 500 500 500 500]
 for loop1 = 1:1:4
  for loop2 = 1:1:3
  F_num = (loop2-1)*4 + loop1;
  filenum = randperm(count(F_num),n_datafiles(F_num));
    for loop3 = 1:1:n_datafiles(F_num)
      filename = "Data_" + F_num + "_" + filenum(loop3) + ".png";
      source = fullfile("F"+F_num,filename);
      destination = fullfile(""+loop2,filename);
      copyfile(source,destination)
    end
  end
end
```

2. Transfer version_3 for Lidar data only

```
clear;
count = zeros(12,1);      % Count for each folder
nsquadval = 20;          % Quadrant size / 2 (N&S)
folder_path = 'D:\Quan\Data\All data for code\Supervisored\All';
chdir(folder_path);
file_info = dir(fullfile(folder_path,'*.mat'));
n_files = length(file_info);
or floop = 1:1:3
  foldername = "F" + floop;
  mkdir(folder_path,foldername)
end
 for loop = 1:1:n_files
  if(rem(loop,1000)==0)  % Print loop count to screen
     loop
  end
```

```
filename = "MATLAB_Data_" + loop + ".mat";
 astruct = load(filename);
 bcell = struct2cell(astruct);
  data = bcell[143];
 px = data();
 py = data(2);
 pz = data(3);
  cx = data(4);
 cy = data(5);
 cz = data(6);
 rad = atan2(-cx,cy);
 bearing = (rad/3.14159) * 180.0;
 if(bearing<-180)
   bearing = bearing+360;
 elseif(bearing>180)
   bearing = bearing-360;
 end
  decision = data(7);
  if(bearing>=-nsquadval && bearing<=nsquadval)
   nesw = 1;
 elseif(bearing>nsquadval && bearing<(180-nsquadval))
   nesw = 2;
 elseif(bearing>(-180+nsquadval) && bearing<-nsquadval)
   nesw = 4;
 else
   nesw = 3;
 end
 foldernum = decision;
  angle_d = linspace(bearing-120,bearing+120,667);
 for aloop = 1:1:667
   if(angle_d(aloop)>180)
     angle_d(aloop) = angle_d(aloop) - 360;
   elseif(angle_d(aloop)<-180)
     angle_d(aloop) = angle_d(aloop) + 360;
   end
 end
 angle = uint8(255 - (abs(angle_d)*255/180)); % White for north, darker either side
```

```
   lidar_m = data(8:end);
  lidar = 0 * ([1:667]);
  for dloop = 1:1:667
   if(lidar_m(dloop)<5.6)
    lidar(dloop) = lidar_m(dloop)*255/5.6;
   else
    lidar(dloop) = 255;
   end
  end
 tempimage = [lidar];
  image = uint8(imresize(tempimage,[1 67]));
   count(foldernum) = count(foldernum) + 1;
  outfilename = "F" + foldernum + "\Data_" + foldernum + "_" + count(foldernum) + ".png";
  imwrite(image,outfilename,'png');
end
ount = count'
 mkdir(folder_path,"1")
mkdir(folder_path,"2")
mkdir(folder_path,"3")
 n_datafiles = [30000 30000 30000]
 for loop1 = 1:1:1
  for loop2 = 1:1:3
  F_num = (loop2-1)*4 + loop1;
  filenum = randperm(count(F_num),n_datafiles(F_num));
   for loop3 = 1:1:n_datafiles(F_num)
    filename = "Data_" + F_num + "_" + filenum(loop3) + ".png";
    source = fullfile("F"+F_num,filename);
    destination = fullfile(""+loop2,filename);
    copyfile(source,destination)
   end
  end
end
```

3. Transfer version_4 for dynamic environment

```
lear;
count = zeros(12,1);     % Count for each folder
nsquadval = 20;          % Quadrant size / 2 (N&S)
```

```
folder_path = 'D:\Quan\Webots\April\March\controllers\Random-Motion\Collection\All';
chdir(folder_path);
file_info = dir(fullfile(folder_path,'*.mat'));
n_files = length(file_info);
 for floop = 1:1:12
    foldername = "F" + floop;
    mkdir(folder_path,foldername)
end
     for loop = 1:1:n_files
    if(rem(loop,1000)==0)  % Print loop count to screen
        loop
    end
   filename = "MATLAB_Data_" + loop + ".mat";
    astruct = load(filename);
    bcell = struct2cell(astruct);
        data = bcell[143];
    px = data(1);
    py = data(2);
    pz = data(3);
        cx = data(4);
    cy = data(5);
    cz = data(6);
    rad = atan2(-cx,cy);
    bearing = (rad/3.14159) * 180.0;
    if(bearing<-180)
        bearing = bearing+360;
    elseif(bearing>180)
        bearing = bearing-360;
    end
        decision = data(7);
        if(bearing>=-nsquadval && bearing<=nsquadval)
        nesw = 1;
    elseif(bearing>nsquadval && bearing<(180-nsquadval))
        nesw = 2;
    elseif(bearing>(-180+nsquadval) && bearing<-nsquadval)
        nesw = 4;
    else
```

```
    nesw = 3;
end
foldernum = (decision-1)*4+nesw;
    angle_d = linspace(bearing-120,bearing+120,667);
for aloop = 1:1:667
    if(angle_d(aloop)>180)
        angle_d(aloop) = angle_d(aloop) - 360;
    elseif(angle_d(aloop)<-180)
        angle_d(aloop) = angle_d(aloop) + 360;
    end
end
angle = uint8(255 - (abs(angle_d)*255/180)); % White for north, darker either side
 for loop = 1:1:5
    start = 8 + 666 * (loop - 1);
    endt = 8 + 666 * loop;
                lidar_m = data(start:endt);
    lidar = 0 * ([1:667]);
    for dloop = 1:1:667
        if(lidar_m(dloop)<5.6)
            lidar(dloop) = uint8(lidar_m(dloop)*255/5.6);
        else
            lidar(dloop) = uint8(255);
        end
    end
            tempimage = [lidar];
    image = uint8(imresize(tempimage,[1 67]));
     eval(['image',num2str(loop),'=','image',';']);
        end
        count(foldernum) = count(foldernum) + 1;
outfilename = "F" + foldernum + "\Data_" + foldernum + "_" + count(foldernum) + ".gif";


imwrite(image1,outfilename,'gif','LoopCount',1,'DelayTime',0.2);
imwrite(image2,outfilename,'gif','WriteMode','append','DelayTime',0.2);
imwrite(image3,outfilename,'gif','WriteMode','append','DelayTime',0.2);
imwrite(image4,outfilename,'gif','WriteMode','append','DelayTime',0.2);
imwrite(image5,outfilename,'gif','WriteMode','append','DelayTime',0.2);
```

```
nd
count = count'
```

4. Extra_1 additional- random

```
clear;
count = zeros(12,1);     % Count for each folder
nsquadval = 20;          % Quadrant size / 2 (N&S)
folder_path = 'D:\Quan\Data\All data for code\Supervisored\All';
chdir(folder_path);
file_info = dir(fullfile(folder_path,'*.mat'));
n_files = length(file_info);
for floop = 1:1:12
   foldername = "F" + floop;
   mkdir(folder_path,foldername)
end
for loop = 1:1:n_files
   if(rem(loop,1000)==0)  % Print loop count to screen
      loop
   end
    filename = "MATLAB_Data_" + loop + ".mat";
   astruct = load(filename);
   bcell = struct2cell(astruct);

   data = bcell[143];
   px = data(1);
   py = data(2);
   pz = data(3);
      cx = data(4);
   cy = data(5);
   cz = data(6);
   rad = atan2(-cx,cy);
   bearing = (rad/3.14159) * 180.0;
   if(bearing<-180)
      bearing = bearing+360;
   elseif(bearing>180)
      bearing = bearing-360;
   end
```

```
    decision = data(7);
    if(bearing>=-nsquadval && bearing<=nsquadval)
    nesw = 1;
elseif(bearing>nsquadval && bearing<(180-nsquadval))
    nesw = 2;
elseif(bearing>(-180+nsquadval) && bearing<-nsquadval)
    nesw = 4;
else
    nesw = 3;
end
foldernum = (decision-1)*4+nesw;
    angle_d = linspace(bearing-120,bearing+120,667);
for aloop = 1:1:667
    if(angle_d(aloop)>180)
        angle_d(aloop) = angle_d(aloop) - 360;
    elseif(angle_d(aloop)<-180)
        angle_d(aloop) = angle_d(aloop) + 360;
    end
end
angle = uint8(255 - (abs(angle_d)*255/180)); % White for north, darker either side
  lidar_m = data(8:end);
lidar = 0 * ([1:667]);
for dloop = 1:1:667
    if(lidar_m(dloop)<5.6)
        lidar(dloop) = uint8(lidar_m(dloop)*255/5.6);
    else
        lidar(dloop) = uint8(255);
    end
end
for extraloop = 1:1:10
    tempimage = [angle;lidar];
    image = uint8(imresize(tempimage,[2 67]));
    for dloop = 1:1:67
        if(extraloop>1)
            if(image(2,dloop)<200)
                image(2,dloop) = image(2,dloop)+uint8(9*rand);
            end
```

```
        end
      end
        count(foldernum) = count(foldernum) + 1;
      outfilename = "F" + foldernum + "\Data_" + foldernum + "_" + count(foldernum) + ".png";
      imwrite(image,outfilename,'png');
   end
end
ount = count'
```

5. Extra_2 additional-systematic

```
clear;
count = zeros(12,1);     % Count for each folder
nsquadval = 20;          % Quadrant size / 2 (N&S)
folder_path = 'D:\Quan\Data\All data for code\Supervisored\All';
chdir(folder_path);
file_info = dir(fullfile(folder_path,'*.mat'));
n_files = length(file_info);
 for floop = 1:1:12
   foldername = "F" + floop;
   mkdir(folder_path,foldername)
end
for loop = 1:1:n_files
   if(rem(loop,1000)==0)  % Print loop count to screen
      loop
   end
    filename = "MATLAB_Data_" + loop + ".mat";
   astruct = load(filename);
   bcell = struct2cell(astruct);
      data = bcell[143];
   px = data(1);
   py = data(2);
   pz = data(3);
      cx = data(4);
   cy = data(5);
   cz = data(6);
   rad = atan2(-cx,cy);
   bearing = (rad/3.14159) * 180.0;
```

```
if(bearing<-180)
   bearing = bearing+360;
elseif(bearing>180)
   bearing = bearing-360;
end
   decision = data(7);
   if(bearing>=-nsquadval && bearing<=nsquadval)
   nesw = 1;
elseif(bearing>nsquadval && bearing<(180-nsquadval))
   nesw = 2;
elseif(bearing>(-180+nsquadval) && bearing<-nsquadval)
   nesw = 4;
else
   nesw = 3;
end
foldernum = (decision-1)*4+nesw;
   angle_d = linspace(bearing-120,bearing+120,667);
for aloop = 1:1:667
   if(angle_d(aloop)>180)
      angle_d(aloop) = angle_d(aloop) - 360;
   elseif(angle_d(aloop)<-180)
      angle_d(aloop) = angle_d(aloop) + 360;
   end
end
angle = uint8(255 - (abs(angle_d)*255/180)); % White for north, darker either side
   lidar_m = data(8:end);
lidar = 0 * ([1:667]);
for dloop = 1:1:667
   if(lidar_m(dloop)<5.6)
      lidar(dloop) = uint8(lidar_m(dloop)*255/5.6);
   else
      lidar(dloop) = uint8(255);
   end
end
  for extraloop = 1:1:10
   tempimage = [angle;lidar];
   image = uint8(imresize(tempimage,[2 67]));
```

```
    for dloop = 1:1:67
      %if(extraloop>1)
        if(image(2,dloop)<200)
          image(2,dloop) = image(2,dloop)+extraloop-1;
        end
      %end
    end
      count(foldernum) = count(foldernum) + 1;
    outfilename = "F" + foldernum + "\Data_" + foldernum + "_" + count(foldernum) + ".png";
    imwrite(image,outfilename,'png');
  end
end
count = count'
```

Appendix C. Webots main code

```
TIME_STEP = 64;
left_motor = wb_robot_get_device(convertStringsToChars("wheel_left"));
right_motor = wb_robot_get_device(convertStringsToChars("wheel_right"));
wb_motor_set_position(left_motor,inf);
wb_motor_set_position(right_motor,inf);
wb_motor_set_velocity(left_motor,0.0);
wb_motor_set_velocity(right_motor,0.0);
urg041x = wb_robot_get_device(convertStringsToChars("Hokuyo URG-04LX-UG01"));
wb_lidar_enable(urg041x, TIME_STEP);
wb_lidar_enable_point_cloud(urg041x);
bumper = wb_robot_get_device(convertStringsToChars("bumper"));
wb_touch_sensor_enable(bumper, TIME_STEP);
compass = wb_robot_get_device(convertStringsToChars("compass"));
wb_compass_enable(compass,TIME_STEP);
arrow = wb_robot_get_device(convertStringsToChars("arrow"));
wb_keyboard_enable(TIME_STEP);
decision = 0;
speed = 1.0;
LSpeed = 0;
RSpeed = 0;
start = 6;
decision_1 = 0;
decision_2 = 0;
decision_3 = 0;
decision_1_all = 0;
decision_2_all = 0;
decision_3_all = 0;decision_once = 0;
movement_counter = 0;    %movement counter for reverse when collision
movement_time = 0;      %movement counter in one decision
bumper_times = 0;
good_times = 0;Times_Random = 0;
Times_Retrain = 0;
random = 0;
result = 0;
```

```
time = 0;
retraintimes = 1;
reset = 0;
remove = 0;
Stop = 0;
Allretrain = 0;
previous_acc = 0;
loop_times = 0;
 Data_1 =[];
Data_2 =[];
Data_3 =[];
Data_4 =[];
Data_5 =[];
Data_6 =[];
Data_7 =[];
Data_8 =[];
Data_9 =[];
Data_10 =[];
 position_1 = [];
position_2 = [];
position_3 = [];
position_4 = [];
position_5 = [];
position_6 = [];
position_7 = [];
position_8 = [];
position_9 = [];
position_10 = [];
pototaion_1 = [];
rototaion_2 = [];
rototaion_3 = [];
rototaion_4 = [];
rototaion_5 = [];
rototaion_6 = [];
rototaion_7 = [];
rototaion_8 = [];
rototaion_9 = [];
```

```
rototaion_10 = [];
robot_node = wb_supervisor_node_get_from_def('TING');
trans_field = wb_supervisor_node_get_field(robot_node, 'translation');
rotoa_field = wb_supervisor_node_get_field(robot_node, 'rotation');
 random_position_1 = [5.5 -20 0.55];
random_position_2 = [5.5 -13 0.55];
random_position_3 = [-5 -20 0.55];
random_position_4 = [-5 -15 0.55];
random_position_5 = [0.63 -17.7 0.55];
random_position_6 = [-6.50 -8.72 0.55];
random_position_7 = [-1.63 -8.72 0.55];
random_position_8 = [6.11 -8.72 0.55];
random_position_9 = [3.40 -1.82 0.55];
random_position_10 = [-2.16 -1.77 0.55];
 collision = 0;
decision_bad = 0;
 load decision_all.mat;
  load 'network1-unsupervisoed_20000_good80%_20epoch_changedmaxpooling.mat';
net1 = trainedNetwork_2;
while wb_robot_step(TIME_STEP) ~= -1
      values = wb_supervisor_field_get_sf_vec3f(trans_field);
   rotos = wb_supervisor_field_get_sf_rotation(rotoa_field);
     save ('decision_all','decision_all');
       if  movement_time == 1 && movement_counter == 0 && values(2) > 25
            random_number = (round(rand(1,1)*9))+1 ;
            if random_number == 1
          INITIAL = random_position_1;
       elseif random_number == 2
          INITIAL = random_position_2;
       elseif random_number == 3
          INITIAL = random_position_3;
       elseif random_number == 4
          INITIAL = random_position_4;
       elseif random_number == 5
          INITIAL = random_position_5;
       elseif random_number == 6
          INITIAL = random_position_6;
```

```
    elseif random_number == 7
       INITIAL = random_position_7;
    elseif random_number == 8
       INITIAL = random_position_8;
    elseif random_number == 9
       INITIAL = random_position_9;
    elseif random_number == 10
       INITIAL = random_position_10;
    end
          var = 0.4;
    b = [(var*rand(1,2))-(var/2),0];
    INITIAL = INITIAL + b;
          wb_supervisor_field_set_sf_vec3f(trans_field, INITIAL);
          random_rotation = (3.14159*3/4) + ((rand(1,1)*3.14159/2));
    Rotation = [0,0,1,random_rotation];
    wb_supervisor_field_set_sf_rotation(rotoa_field, Rotation);
      end
  Com = wb_compass_get_values(compass);
    key = wb_keyboard_get_key();
    if key == WB_KEYBOARD_UP
    start = 1;
     end


  if start == 1
                 if wb_touch_sensor_get_value(bumper) == 0  && result == 1
       if movement_time > 0
          movement_time = movement_time - 1;
       end
              if movement_time == 0
          if decision ~= 0
                        decision_once = decision_once + 1;
                          decision_all = decision_all + 1;
                outfilename = ['D:\Quan\Webots\Nov\controllers\test_network\Data for unsuperviosred
20000(80%)_20epoch_changedmaxpooling\','MATLAB_Data_',num2str(decision_all)];
              save(outfilename,'Data_1');


                        result = 0;
```

```
    end
    end
end
    if movement_time == 0  && result == 0
        Com = wb_compass_get_values(compass);
    urg041x_values = wb_lidar_get_range_image(urg041x);
        cx = Com(1);
    cy = Com(2);
    cz = Com(3);
    rad = atan2(-cx,cy);
    bearing = (rad/3.14159) * 180.0;
    if(bearing<-180)
      bearing = bearing+360;
    elseif(bearing>180)
      bearing = bearing-360;
    end
        angle_d = linspace(bearing-120,bearing+120,667);
    for aloop = 1:1:667
      if(angle_d(aloop)>180)
        angle_d(aloop) = angle_d(aloop) - 360;
      elseif(angle_d(aloop)<-180)
        angle_d(aloop) = angle_d(aloop) + 360;
      end
    end
        lidar_m = urg041x_values;
    lidar = 0 * ([1:667]);
    for dloop = 1:1:667
      if(lidar_m(dloop)<5.6)
        lidar(dloop) = uint8(lidar_m(dloop)*255/5.6);
      else
        lidar(dloop) = uint8(255);
      end
    end
      angle = uint8(255 - (abs(angle_d)*255/180));
        position_1 = values;
    rototaion_1 = rotos;
    random = round(rand(1,1)*2);
```

```
decision = random + 1;
        lidar = uint8(urg041x_values*255/5.6);
          tempimage = [angle;lidar];
image = uint8(imresize(tempimage,[2 67]));
save('image','image');
          data = image;
         [XPred1,probs] = classify(net1,data);
 [prob,decision] = max(probs);
     Data_1 = [collision,values,Com,decision,urg041x_values];
Data_1 = [Data_1 probs];
              result = 1;
          if decision == 1
   LSpeed = 1;
   RSpeed = 1;
elseif decision == 2
   LSpeed = 0.5;
   RSpeed = 1;
elseif decision == 3
   LSpeed = 1;
   RSpeed = 0.5;
elseif decision == 0
   LSpeed = 0;
   RSpeed = 0;
end
          movement_time = 100;
       end
   if wb_touch_sensor_get_value(bumper) > 0
result = 0;
                Data_1(1) = 1;
decision_bad = decision_bad + 1;
decision_all = decision_all + 1;
          outfilename = ['D:\Quan\Webots\Nov\controllers\test_network\Data for unsuperviosred
20000(80%)_20epoch_changedmaxpooling\','MATLAB_Data_',num2str(decision_all)];
          save('decision_bad','decision_bad');
save(outfilename,'Data_1');
          random_number = (round(rand(1,1)*9))+1 ;
if random_number == 1
```

```
        INITIAL = random_position_1;
    elseif random_number == 2
        INITIAL = random_position_2;
    elseif random_number == 3
        INITIAL = random_position_3;
    elseif random_number == 4
        INITIAL = random_position_4;
    elseif random_number == 5
        INITIAL = random_position_5;
    elseif random_number == 6
        INITIAL = random_position_6;
    elseif random_number == 7
        INITIAL = random_position_7;
    elseif random_number == 8
        INITIAL = random_position_8;
    elseif random_number == 9
        INITIAL = random_position_9;
    elseif random_number == 10
        INITIAL = random_position_10;
    end
            wb_supervisor_simulation_reset_physics()
    wb_supervisor_field_set_sf_vec3f(trans_field, INITIAL);
            random_rotation =  ((rand(1,1)*3.14159*2));
    Rotation = [0,0,1,random_rotation];
    wb_supervisor_field_set_sf_rotation(rotoa_field, Rotation);
            Data_2 =[];
    Data_3 =[];
    Data_4 =[];
    Data_5 =[];
            decision_once = 0;
    movement_time = 0;
    start = 1;
    LSpeed = 0;
    RSpeed = 0;
end
```

```
end
   wb_motor_set_velocity(left_motor, LSpeed);
wb_motor_set_velocity(right_motor, RSpeed);
   drawnow;
   end
```

Appendix D. Network for dynamic case

```
digitDatasetPath = fullfile(' D:\Quan\Webots\April\March\controllers\Random-
Motion\Collection\lidaronly\All\');
Timds = imageDatastore(digitDatasetPath,'IncludeSubfolders',true,'LabelSource','foldernames');
Timds.ReadFcn = @customreader;
labelCount = countEachLabel(Timds);
Timg = readimage(Timds,1);
size(Timg);
numTrainFiles = 37500;
[TimdsTrain,TimdsValidation] = splitEachLabel(Timds,numTrainFiles,'randomize');
 layers_dynamic  = [
    image3dInputLayer([1 67 5 1],"Name","imageinput")
    convolution3dLayer([3 3 3],8,'Padding','same')
    batchNormalizationLayer
    reluLayer
    maxPooling3dLayer([2 2 2],'Padding','same')
    convolution3dLayer([3 3 3],16,'Padding','same')
    batchNormalizationLayer
    reluLayer
    maxPooling3dLayer([2 2 2],'Padding','same')
    convolution3dLayer([3 3 3],32,'Padding','same')
    batchNormalizationLayer
    reluLayer
    fullyConnectedLayer(3)
    softmaxLayer
    classificationLayer];
  options = trainingOptions('sgdm', 'InitialLearnRate',0.0001, 'MaxEpochs',5, 'Shuffle','every-epoch', ...
    'ValidationData',TimdsValidation, 'ValidationFrequency',5, 'Verbose',false, 'Plots','training-progress');
 net = trainNetwork(TimdsTrain,layers,options); %%%
YPred = classify(net,TimdsValidation);
 YValidation = TimdsValidation.Labels;
accuracy = sum(YPred == YValidation)/numel(YValidation);
 function data = customreader(filename)
    tempdata = imread(filename,'Frames','all');
    data = permute(tempdata,[1 2 4 3]);
```

Appendix E. Matlab code for testing the trained CNN in a real robot

```
%This programme is used for testing the trained network from Webots
%simulation into our real robot.
% The input data is LiDAR data & Compass heading(IMU).
clear all;
delete(instrfindall);


load 'all.mat'
%This network contains LiDAR data & Compass heading
load 'network1-supervisored';
net1 = trainedNetwork_1;

port = 'COM3';
baudrate = 9600;
s = serialport(port,baudrate);
set(s,'Timeout',10);
configureTerminator(s,'LF');
set(s,'InputBufferSize',512);

fopen(s);
pause(0.1);

fprintf(s,'B')
pause(0.1);

        % % Read ASCII-terminated string from the serial port.
        % data = readline(s);
for loop=1:1:3

    out=fscanf(s);

end

for loop_all = 1:1:50

stepsize = (pi*(240/180))/681;
for loop = 1:1:682
    range(loop) = 0;
    angle(loop) =  -(pi*(120/180))+stepsize*(loop-1);
end

range=LIDARScan681(s);

bearing = Getheading(s);

% Calculate angles for each lidar reading
angle_d = linspace(bearing-120,bearing+120,682);
for aloop = 1:1:682
```

```
    if(angle_d(aloop)>180)
        angle_d(aloop) = angle_d(aloop) - 360;
    elseif(angle_d(aloop)<-180)
        angle_d(aloop) = angle_d(aloop) + 360;
    end
end

angle_robot = uint8(255 - (abs(angle_d)*255/180)); % White for north, darker
either side
% angle = uint8((angle_d+180)*255/360);       % Black to white from S-W-N-E-W

stepsize = (pi*(240/180))/681;
for loop = 1:1:682
    range(loop) = 0;
    angle(loop) =  -(pi*(120/180))+stepsize*(loop-1);
end

range=LIDARScan681(s)/1000;

% piclidar(dloop) = uint8(urg041x_values(dloop)*255/5.6);
range = range * 255/5.6;
lidar = flip(range);

% plot(image)
index = find(lidar == 0);
lidar(index) = 255;

% plot(lidar)
% angle_robot = flip(angle_robot);

tempimage = [lidar];
image = uint8(imresize(tempimage,[1 67]));
save('image','image');

 [XPred1,probs] = classify(net1,image);
 [prob,decision] = max(probs);

  loop_all

 decision

all(1,loop_all) = decision;
all(2,loop_all) = bearing;
save('all_V2','all');

% Navigate with the network decision
if decision == 1
    fprintf(s,'W')
elseif decision == 2
    fprintf(s,'R')
elseif decision == 3
    fprintf(s,'L')
end

pause(1);

fprintf(s,'S');
```

```
pause(0.1);

end

% net = trainedNetwork_2
% exportNetworkToTensorFlow(net,"myModel")
```