

Integration of AI and mobile gateways in LoRa Networks

Master Thesis



Approval

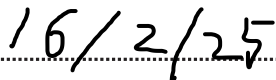
This thesis has been prepared over 5½ months at the Department of Applied Mathematics and Computer Science, at the Technical University of Denmark, DTU, in partial fulfilment for the degree Master of Science in Engineering, MSc Eng.

It is assumed that the reader has a basic knowledge in the areas of statistics and Computer Science.

Simon Kristoffer Janum - s194609

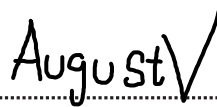


Signature

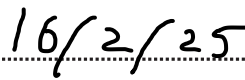


Date

August Falck Kreinert Valentin - s194802



Signature



Date

Abstract

The growing demand for Low-Power Wide Area Networks (LPWANs) in Internet of Things (IoT) applications, such as sensor nodes in agriculture or devices in smart cities, has highlighted scalability challenges for technologies like LoRa. Traditional solutions, such as stationary gateways, often face logistical and financial constraints. This thesis explores the use of Reinforcement Learning (RL) to improve the scalability and performance of LoRa networks using RL-driven mobile gateways.

Using Objective Modular Network Testbed in C++ (OMNeT++) for network simulation and Stable-Baselines3 (SB3) for custom RL environment training, we investigate the performance of an RL-driven mobile gateway compared to stationary gateways and simple mobility models. Packet Delivery Ratio (PDR) and Jain's Fairness Index are used as network metrics to quantify performance.

We show that a model trained in SB3 can be successfully transferred to OMNeT++, maintaining both its behavior and performance. Furthermore, evaluations highlight that, in cases with sufficient time between transmissions, a single RL-driven gateway provides performance comparable to that of four stationary gateways.

Our evaluations show that RL-driven mobility achieves a near-perfect PDR and fairness in near-ideal simulation environments, where node placement and transmission intervals allow sufficient time for adaptation, demonstrating advantages in adaptability over stationary solutions. Static mobility models, like circular motion, perform well in specific scenarios but are less adaptable compared to RL-driven gateways. However, challenges remain in complex decision-making, particularly with high transmission rates or faulty nodes, which can reduce performance.

This thesis demonstrates the potential of RL for scaling LoRa networks dynamically and suggests future work to refine decision-making strategies and improve robustness for real-world applications.

Acknowledgements

We would like to express our deepest gratitude to our supervisors, Assistant Professor Charalampos 'Haris' Orfanidis and Professor Xeno 'Fontas' Fafoutis, for their invaluable guidance, support, and encouragement throughout the course of this thesis. Their expertise and insights have not only shaped the direction of this work but also significantly enhanced our knowledge and understanding of the field and we are truly thankful for the opportunity to learn from them. We would also like to extend our sincere thanks to Fabian Fernando Jurado Lasso for his guidance and insights into Reinforcement Learning. His expertise in the area played a crucial role in driving this work forward and was instrumental in helping us reach this stage of the thesis.

We would also like to thank our respective families and girlfriends for their unwavering love, support, and encouragement. Thank you for believing in each of us and for always being there, whether it was through moments of doubt or triumph. Your support has been a constant source of strength, and we could not have completed this journey without each of you.

Contents

Preface	iii
Abstract	iv
Acknowledgements	v
1 Introduction	6
1.1 Motivation	6
1.2 Main Objective	6
1.3 Structure	7
2 Background	8
2.1 LoRa	8
2.2 LoRaWAN	10
2.3 OMNeT++	12
2.4 INET	12
2.5 FLoRa	12
2.6 Neural Networks	13
2.7 Reinforcement Learning	17
2.8 Stable-Baselines3 (SB3)	23
3 Related Works	25
3.1 LoRa Literature	25
3.2 Mobility for LoRa Literature	25
3.3 Reinforcement Learning Literature	26
3.4 Reward Shaping Literature	28
4 Methodology	29
4.1 Setup	29
4.2 Custom Environment in Stable-Baselines3	29
4.3 Policy	32
4.4 Credit Assignment	34
4.5 Integrating OMNeT++ for Model Evaluation	34
4.6 Experiments	37
4.7 Evaluation method	37
5 Evaluation	41
5.1 Scenario 1	43
5.2 Scenario 2	53
5.3 Scenario 3	56
5.4 Scenario 4	63
5.5 Scenario 5	71
6 Discussion and Feasibility	80

7 Conclusion	82
8 Future works	83
Bibliography	84
Appendices	87
Appendix A Direct Training from OMNeT++ simulations	88
Appendix B OMNeT++ custom component diagram	89
Appendix C Scenario 2 - Gateway Distances to Nodes	90
Appendix D Scenario 3.A - Gateway Distances to Nodes	92
Appendix E Validation Model Conversion Script	94

List of Figures

2.1	LoRa illustration of chirps [4]	8
2.2	Coding Rate 5/7 [7]	9
2.3	Spectrogram of different Spreading Factors [8]	10
2.4	Star-of-stars LoRa topology	11
2.5	Neural Network Visualization [11]	13
2.6	Residual learning: a building block ^[13]	16
2.7	Stable-Baselines3 policy network architecture [17]	24
4.1	Custom environment in SB3	30
4.2	Distribution of transmission interval at 1500 steps with a standard deviation of 5	31
4.3	Packet Reception Propability	32
5.1	S1 Fairness during training for final model	44
5.2	S1 PDR during training for final model	44
5.3	S1 Performance box plot for final model	45
5.4	S1 Performance bar plot for final model	45
5.5	S1.A Heatmap of episode	46
5.6	S1.A Fairness during training	47
5.7	S1.A PDR during training	47
5.8	S1.B Fairness during training	48
5.9	S1.B PDR during training	48
5.10	S1.B: Fairness with a 86400 normalization value for expected time	50
5.11	S1.B Fairness with residuals and a 86400 normalization value for expected time	50
5.12	S1.B PDR with a 86400 normalization value for expected time	51
5.13	S1.B PDR with residuals and a 86400 normalization value for expected time	51
5.14	S1.C Fairness and PDR during training using DQN	52
5.15	S2 Environments in SB3 and OMNeT++.	53
5.16	S2 Performance bar plot in SB3	54
5.17	S2 Performance box plot in SB3	54
5.18	S2 PDR and Fairness box plots in OMNeT++	55
5.19	S2 Performance bar plot in OMNeT++	56
5.20	S3 Performance bar plot (no faulty node)	57
5.21	S3.A Performance bar plot	58
5.22	S3.A Heatmap of gateway in an episode	59
5.23	S3.B Performance bar plot	60
5.24	S3.B Disance for node 3 over an episode	60
5.25	S3.B Heatmap for gateay in an episode	61
5.26	S3.C Performance bar plot	62
5.27	S3.C Heatmap of Gateway	62
5.28	S4.A Initial positions of LoRaWAN units.	64

5.29	S4.A PDR box plot	65
5.30	S4.A Fairness box plot	65
5.31	S4.A Performance bar plot	66
5.32	S4.B Initial positions of LoRaWAN units.	67
5.33	S4.B PDR and Fairness box plots	68
5.34	S4.B Performance bar plot	68
5.35	S4.C Initial positions of LoRaWAN units.	69
5.36	S4.C PDR and Fairness box plots	70
5.37	S4.C Performance bar plot	71
5.38	S5.A Initial positions of LoRaWAN units.	73
5.39	S5.A PDR and Fairness bar plots	73
5.40	S5.A Performance bar plot	74
5.41	S5.B Initial positions of LoRaWAN units.	75
5.42	S5.B PDR and Fairness box plots	75
5.43	S5.B Performance bar plot	76
5.44	S5.C Initial positions of LoRaWAN units.	78
5.45	S5.C PDR and Fairness box plots	78
5.46	S5.C Performance bar plot	79
A.1	System flow for training in OMNeT++ directly	88
B.1	Diagram of the relation between custom components in OMNeT++	89
C.1	S2 Gateways distance to node 0 over steps	90
C.2	S2 Gateways distance to node 1 over steps	90
C.3	S2 Gateways distance to node 2 over steps	91
C.4	S2 Gateways distance to node 3 over steps	91
D.1	S3.A Gateways distance to node 0 over steps	92
D.2	S3.A Gateways distance to node 1 over steps	92
D.3	S3.A Gateways distance to node 2 over steps	93
D.4	S3.A Gateways distance to node 3 over steps	93

List of Tables

2.1	SB3 PPO compatibility for different space types	24
2.2	SB3 DQN compatibility for different space types	24
5.1	Default simulation parameters for OMNeT++	42
5.2	Default simulation parameters for SB3	42
5.3	S1 Learning parameters	43
5.4	S2 SB3 scenario parameters	53
5.5	S2 OMNeT++ scenario parameters	53
5.6	S3 Scenario parameters	57
5.7	S4.A OMNeT++ Parameters	64
5.8	S4.B OMNeT++ parameters	67

5.9	S4.C OMNeT++ Parameters	69
5.10	S5.A OMNeT++ Parameters	72
5.11	S5.A Circular motion parameters	72
5.12	S5.B OMNeT++ Parameters	74
5.13	S5.B Circular motion parameters	74
5.14	S5.C OMNeT++ parameters	76
5.15	S5.C Circular motion parameters	77

Acronyms

A2C Advantage Actor Critic. 27

ACER Actor Critic and Experience Replay. 26

ACuTE Automatic Curriculum Transfer from Simple to Complex Environments. 27, 28

ADR Adaptive Data Rate. 11, 12

AI Artificial intelligence. 6, 29, 41

BW Bandwidth. 8, 10, 27

CR Coding Rate. 9, 10, 42

CSS Chirp Spread Spectrum. 8

CTDE Centralized Training, Decentralized Execution. 27

DQN Deep-Q-Network. 19–21, 23–27, 41, 43, 52, 82

FLoRa Framework for LoRa. 6, 12, 25, 34, 35

HF High-Fidelity. 53, 82

IoT Internet of Things. iv, 6, 10, 12

LF Low-Fidelity. 82

LiteRT Lite Runtime. 23

LoRa Long Range. iv, 6–10, 12, 25–27, 29, 34, 41, 80, 82, 83

LoRaWAN Long Range Wide Area Network. 9–12, 25, 42, 81

LPWANs Low-Power Wide Area Networks. iv, 6, 26

LSTM Long Short-Term Memory. 16

MAC Media Access Control. 10

MAPPO Multi-Agent Proximal Policy Optimization. 27

MDP Markov Decision Process. 18, 28

NED Network Description. 12

OMNeT++ Objective Modular Network Testbed in C++. iv, 6, 12, 29, 34, 35, 37, 39, 41–43, 49, 53, 55, 63, 71, 82, 88

PDR Packet Delivery Ratio. iv, 6, 37–40, 43, 45, 46, 49, 52, 55, 57, 58, 61, 73, 75, 76, 78

PoC Proof-of-Concept. 29

PPO Proximal Policy Optimization. 20, 21, 23–27, 41, 43, 52, 82

RL Reinforcement Learning. iv, v, 6, 7, 17, 19–21, 26–29, 41–43, 63, 71, 73–78, 80, 82, 83

RNNs Recurrent Neural Networks. 16

SB3 Stable-Baselines3. iv, 1, 6, 23, 24, 29, 30, 34, 35, 37–43, 45, 49, 53, 55–57, 82, 83

SF Spreading Factor. 8–10, 27, 81

TCP Transmission Control Protocol. 12

TF TensorFlow. 35

TFLite TensorFlow Lite. 35

TFLite Micro TensorFlow Lite Micro. 35

ToA Time on Air. 9, 42

TP Transmission Power. 8, 27, 81

UAV Unmanned Aerial Vehicle. 26, 27, 80

UDP User Datagram Protocol. 12

1 Introduction

1.1 Motivation

Low-Power Wide Area Networks (LPWANs) is a growing topic, as it is expanding throughout the field of Internet of Things (IoT), with countless applications, such as agriculture or smart cities. Scenarios include monitoring soil conditions [1], air pollution [2] and traffic management [3]. Since the emergence of LoRa in IoT, researchers has focused on investigating its scalability and other physical characteristics like the achievable range. Scalability is still an issue and researchers try to solve it with adding more LoRa gateways or making the gateway mobile.

Expanding networks with more gateways to cover the range of all clients consists of logistical and financial challenges. To combat these issues, a mobile gateway can serve a larger area by traversing between landmarks. However, the introduction of mobility leads to new research questions, such as the optimal paths and speed of the gateway, to optimize network metrics. These problems are however categorically difficult to solve optimally, as clients can be placed in various locations with individual range and transmission times. Therefore, studying the use of Reinforcement Learning (RL), to find suitable mobility configurations in real-time will provide useful insight into mobile gateways. To this a high-fidelity network simulator is used, which will be modified to support RL driven mobility. The project will also contain the use of separate learning environments for training, and have the model be transferred and used in the network simulator for evaluation.

1.2 Main Objective

The main goal of this thesis is to investigate the use of Reinforcement Learning (RL) for scaling LoRa networks. This has been done with the use of OMNeT++, a network simulator which includes the Framework for LoRa (FLoRa), and Stable-Baselines3 (SB3) for the Reinforcement Learning which will be transferred to OMNeT++. This thesis addresses the problem where the environment has distant nodes, e.g. in agriculture where sensor nodes can have large distances between each other. A common solution is to try and place a minimal amount of stationary gateways in the best possible positions to cover and serve all nodes. Instead we propose a mobile gateway, instead of adding new ones. More specifically we introduce a mobile gateway which operates under a Reinforcement Learning Policy, and explore how this will perform compared to both multiple stationary gateways and naive mobility implementations.

This however also comes with its own problems, as neither FLoRa or INET directly supports mobility driven by an AI model. To this we propose TFLite Micro together with custom modules, to perform inference during OMNeT++ simulations. This essentially consists of training a model in a custom Stable-Baselines3 (SB3) environment, to then export into OMNeT++ for evaluation. Therefore a rather large portion of the project also dealt with setting up the framework to make an AI model suitable and comparable with OMNeT++, and how to effectively train such a model. For evaluation, Packet Delivery Ratio (PDR) and Jain's Fairness Index are used as performance metrics to provide an objective comparison between the results.

All related code can be accessed through the project's GitHub¹ repository.

¹https://github.com/yconsj/AI_Lora_Mobility

1.3 Structure

This thesis is organized according to the work carried out during the MSc project in the winter semester of 2024-2025. The thesis focuses on the final selected frameworks and tools rather than on an exhaustive list of all the options tested and investigated.

Chapter 2 provides the necessary background information, covering the theoretical foundations of LoRa and Reinforcement Learning. Understanding these concepts is essential for interpreting the terminology and ideas presented in this thesis. Chapter 3 presents a review of related works, highlighting research on LoRa and Reinforcement Learning that is relevant to the context of this project.

Chapter 4 outlines the methodology, including the decisions made during the project, the setup of the framework for model training, and the experiments carried out. It also discusses the structure of the simulation environment and the evaluation metrics used.

Chapter 5 presents the results of the experiments described in Chapter 4, along with an evaluation of the results and the methodology used throughout the project.

Chapter 6 presents a discussion of the feasibility of the project and its potential real-world applications. This section focuses on how the project could be applied in practice and identifies any modifications or considerations that may be necessary for real-world implementation.

Finally, Chapter 7 provides a conclusion based on the results of the project, followed by a discussion of ideas for future work in the subsequent chapter.

2 Background

2.1 LoRa

LoRa, which stands for Long Range, is a physical radio communication technique. It was developed by Cycleo in 2014 and later acquired by Semtech, and is developed to enable low data rate communication over long distances. The technique to achieve this takes roots from Chirp Spread Spectrum (CSS) technology. LoRa transmits data through radio waves by generating frequency-modulated chirps and encoding them into symbols that represent information. Up-chirps refer to frequency changes from a low point to a high point, while down-chirps describe frequency changes from high to low. Figure 2.1 illustrates the different types of chirps used in LoRa. First, a series of identical preamble up-chirps alerts receivers. This is followed by two down-chirps, which are used to synchronize communication and allow the receiver to adjust its reception window in case of misalignment. After synchronization, data symbols are transmitted. Due to its use of CSS modulation, LoRa is highly robust against interference and resistant to Doppler effects, making it suitable for applications involving mobility.

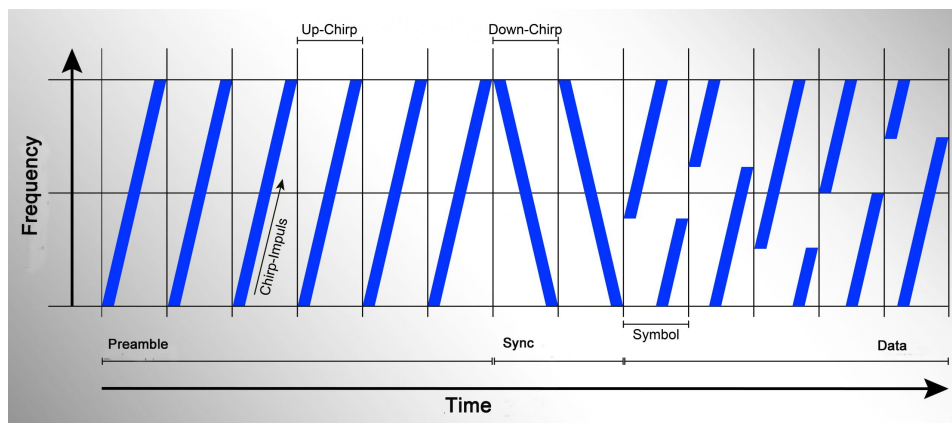


Figure 2.1: LoRa illustration of chirps [4]

There are 3 variables that can be tweaked for data rates: Spreading Factor (SF), Bandwidth (BW), Transmission Power (TP) [5].

Bandwidth (BW)

Bandwidth (BW) is the range of frequency in the transmission band, hence its the width of the transmission signal. A large bandwidth will result in higher data rate, but also makes the sensitivity lower which reduces range as the signal is harder to detect. LoRa can use channels with a bandwidth of 125kHz, 250kHz, or 500kHz and is constrained by the region and frequency plan.

Transmission Power

Transmission Power (TP) is the amount of power that the transmitter produces when outputting. An increase in transmission power will increase the signal strength for the receiver, making the success of the transmission more likely.

LoRa transmits on the license-free sub-GHz spectrum, i.e. 915 MHz, 868 MHz and 433 MHz. It is also documented that it can operate on the 2.4 GHz band, trading off range for a higher data rate [6]. The 868 MHz frequency band is used in Europe with a limit of 14 dBm as the max transmission power.

Time on Air (ToA) refers to total duration it takes for a device to transmit a complete packet over the air. The *duty cycle* is the fraction of time a device spends transmitting relative to a given period, typically expressed as a percentage:

$$\text{Duty Cycle} = \left(\frac{\text{ToA}}{\text{Total Time Period}} \right) \times 100 \quad (2.1)$$

LoRaWAN enforces a strict duty cycle constraint ranging from 0.1% to 1%, depending on the channel used. This means that with a 1% duty cycle, if a message takes 200 ms to transmit (Time on Air), the device must wait for:

$$\left(\frac{1}{1\%} - 1 \right) \times 200 \text{ ms} = 19.8 \text{ s} \quad (2.2)$$

before transmitting a new message. This requirement helps mitigate network congestion and inherently conserves energy.

Coding Rate (CR)

Coding Rate (CR) refers to the proportion of bits in the data stream that carry data, where the rest of the bits are used for Forward Error Correction. There are 4 code rates used for LoRaWAN; 4/5, 4/6, 5/7 or 4/8. e.g. a Coding Rate of 5/7, will have 5 bits for data, and 2 bits for error correction. Different Coding Rate exist to balance the data transmission efficiency and reliability. The choice of coding rate affects error correction, data throughput, and Time on Air.

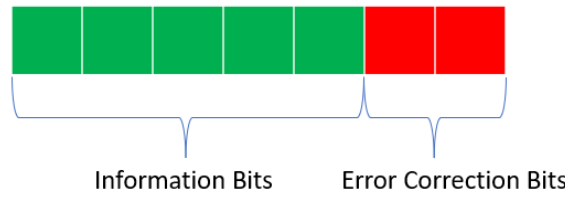


Figure 2.2: Coding Rate 5/7 [7]

Spreading Factor (SF):

The Spreading Factor (SF) controls the chirp rate, which directly affects the data transmission rate. A lower SF results in a higher chirp rate, leading to a higher data rate, and vice versa for higher SF. The chirp rate is inversely related to the transmission range: longer chirps are more resistant to noise and interference, making higher SF values preferable for long-range communication, although this comes at the cost of lower data throughput. LoRa supports six spreading factors, ranging from SF7 to SF12, where the number represents the number of bits sent per symbol. For example, SF7 corresponds to 7 bits per symbol. Each increase in SF doubles the symbol duration, which results in a lower data rate.

The *symbol period* T_s represents the time required to transmit a single symbol. It is given by:

$$T_s = \frac{2^{SF}}{BW} \quad (2.3)$$

where:

- SF is the spreading factor (7 to 12).
- BW is the bandwidth in Hz.

A larger spreading factor increases T_s , meaning each symbol takes longer to transmit, thereby reducing the data rate.

The *nominal bit rate* R_b , which represents the rate at which useful data bits are transmitted over the LoRa network, is given as:

$$R_b = SF \cdot \frac{CR}{T_s} \quad \text{bits/sec} \quad (2.4)$$

or

$$R_b = \frac{CR \cdot SF \cdot BW}{2^{SF}} \quad \text{bits/sec} \quad (2.5)$$

where:

- SF is the spreading factor.
- BW is the bandwidth in Hz.
- CR is the error correction coding rate.
- T_s is the *symbol duration*.

A higher spreading factor results in a lower bit rate, as each symbol takes longer to transmit. However, this trade-off increases resilience to interference and improves range. A spectrogram of different Spreading Factors can be seen in Figure 2.3.

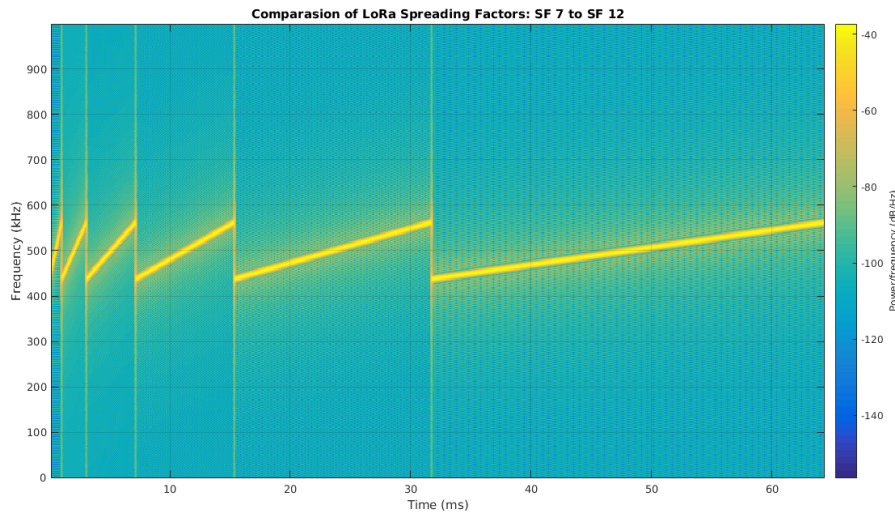


Figure 2.3: Spectrogram of different Spreading Factors [8]

2.2 LoRaWAN

Long Range Wide Area Network (LoRaWAN) is the Media Access Control (MAC) layer protocol, that is used on top of LoRa. It is developed by the LoRa Alliance and they allow other companies to create their own IoT networks that are based on their specifications.

This allows for quick setup of the networks anywhere, as long as the hardware and software requirements are met. The most common application of LoRaWAN is moving sensor-data from end-nodes to a database.

LoRaWAN networks are constructed in a way such that each end devices communicate with a gateway. Gateways are designated devices, equipped with hardware that makes them capable of reception and transmission to multiple frequencies at once. A single LoRaWAN network can have many gateways, where each gateway may service potentially hundreds of nodes. These gateways are then responsible for transmitting the data further up to a network server. This topology is referred to as "star-of-stars", with the network server being the primary star.

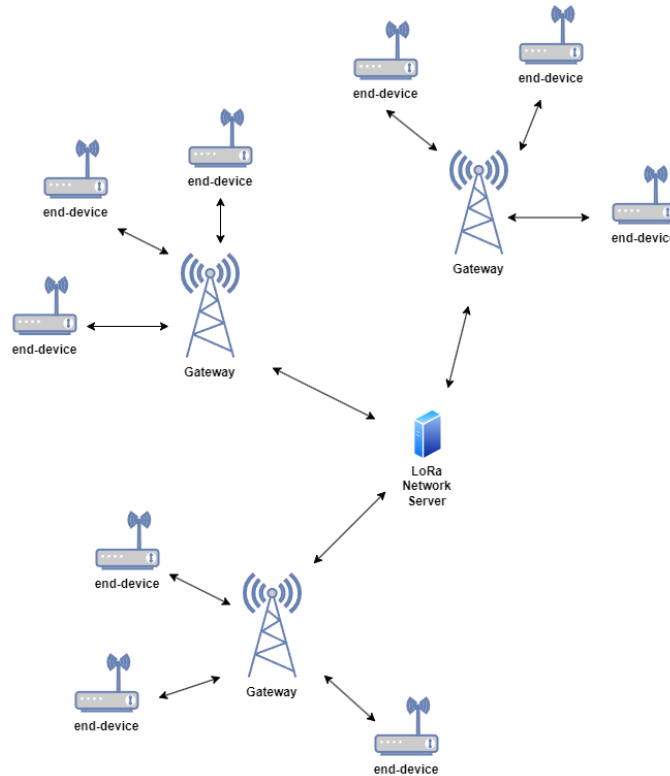


Figure 2.4: Star-of-stars LoRa topology

This type of wireless connection is highly valuable in today's world, where data collection and real-time information are crucial in many fields, such as agriculture and healthcare. LoRaWAN is ideal for many use cases due to its design, which emphasizes reliable, low data-rate transmissions over distances of up to 20km, and low power consumption.

LoRaWAN networks are structured similarly to cellular networks, with a mast-mounted LoRa antenna. End devices do not need to remain constantly active; they only need to power on when communication with the gateway is required. This approach significantly extends battery life while enabling the network to function at a large scale by minimizing interference.

Additionally, LoRaWAN networks can be configured to use Adaptive Data Rate (ADR). ADR is a mechanism managed by the network server, which dynamically adjusts the parameters of the LoRaWAN end-nodes to optimize transmission performance. Depending on the LoRaWAN implementation, the specifics of ADR may vary. For example, in the open-source LoRaWAN network stack "The Things Network," the ADR mechanism optimizes parameters such as spreading factor, bandwidth, and transmission power to balance data rates, airtime, and energy consumption [9].

LoRaWAN specifies 3 classes which a device can take when using LoRaWAN; class A, B and C. These classes have their own protocol for receive windows, where class A is mandatory and class B and C are optional additions to class A:

- Class A (All) is for battery powered devices, where each uplink to a gateway is followed by two short slots for downlinks from the gateway.
- Class B (Beacon) is like class A, but devices of this class periodically sends beacons to the gateway, enabling extra down link windows.
- Class C (Continuous) is like class A, but devices of this class have their radio constantly open to listen. Hence this class is the one that uses the most power and class C devices are often powered by mains.

2.3 OMNeT++

Objective Modular Network Testbed in C++ (OMNeT++) is a component-based C++ simulation library and framework, which uses discrete events. OMNeT++ is primarily used for building network simulations, and has a strong component architecture for models. This makes the framework great for creating specific simulations, as components can be modified, reused or build upon. OMNeT++ provides a high-level language (NED) for assembling components into models.

OMNeT++ also has extensive GUI support, and the simulation kernel is standard C++, which means it can run on pretty much all platforms with a modern C++ compiler.

2.4 INET

The INET Framework is an open-source model library for OMNeT++. INET provides protocols and other models for researchers and students working with communication networks. INET states it is especially useful when designing and validating new protocols or scenarios. INET, like OMNeT++, is built around the concept of modules, which then communicate through message passing, allowing for combinations of modules to create routers, servers and other networking devices.

INET contains models for the internet stack (TCP, UDP, IPv4, IPv6, etc.), wired and wireless link-layer protocols. As well as support for mobility, which allows for moving devices based on different mobility models, which is an important component for this thesis.

2.5 FLoRa

Framework for LoRa (FLoRa) created by Mariusz Slabicki and Gopika Premsankar, is a simulation framework which is based on OMNeT++ and uses components from INET. The simulation framework is used for end-to-end simulations for LoRa networks. This includes modules for LoRa nodes, gateways and network server. The FLoRa framework was developed as part of a research in adaptive configuration of LoRa networks for dense IoT deployments [10], the study also describes the development of FLoRa, and how it was validated against experimental results. The network server can be connected to application logic which is deployed as independent modules. Dynamic management of configuration parameters is supported for nodes and network server through Adaptive Data Rate (ADR), and energy consumption statistics are collected for all nodes. However, neither these energy consumption statistics nor Adaptive Data Rate (ADR) are used in this thesis.

2.6 Neural Networks

Neural networks are a technology within the field of machine learning, inspired by the structure and function of the human brain. These networks consist of layers of interconnected nodes, called neurons, which process input data to generate outputs. This allows them to recognize patterns, make decisions based on data, and generate predictions. Typically, a neural network consists of an input layer, one or more hidden layers, and an output layer, with the hidden layers serving as intermediate stages of data processing.

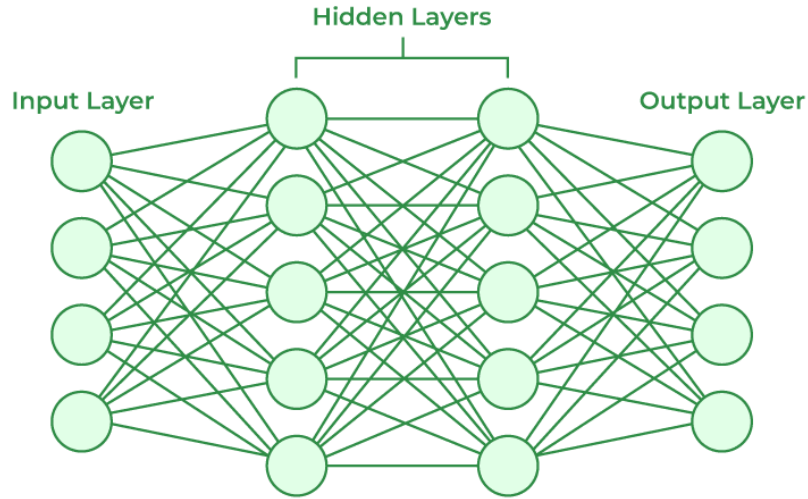


Figure 2.5: Neural Network Visualization [11]

2.6.1 Forward Propagation

Forward propagation is the process used to calculate the output value, or prediction, that a model produces when given an input. During forward propagation, the input is passed sequentially through each layer of the network.

Each neuron in a layer receives a partial input from the output of the previous neurons, multiplied by their respective weights. The total input to a neuron, often referred to as the net input or weighted sum, is the sum of all partial inputs plus a bias term. The final output of the neuron, also called its activation, is obtained by applying an activation function to the net input.

More formally, the activation of a neuron i 's in layer L is computed as follows:

$$a_i^{(L)} = f \left(b[i] + \sum_{j=0}^{|L-1|} a_j^{(L-1)} \cdot w_{ji} \right) \quad (2.6)$$

Where:

- $|L|$ and $|L - 1|$ are the sizes of layers L and $L - 1$, respectively.
- $a_j^{(L-1)}$ represents the activation of neuron j in layer $L - 1$.
- w_{ji} is the weight connecting neuron j in layer $L - 1$ to neuron i in layer L .
- $b[i]$ is the bias term for neuron i in layer L .

- f is the activation function applied to the net input.

This process allows the network to compute the prediction for the given input by propagating values forward through its structure as seen in Figure 2.5.

2.6.2 Activation functions

Many activation functions are used and evaluated in neural networks, each with its own characteristics and applications. Non-linearity is a desirable property in activation functions, as it enables the model to learn complex relationships between inputs and outputs. It is also common for different layers within a neural network to use different activation functions, depending on their role in the architecture. Some of the common activation functions and their characteristics are:

- ReLU:

$$f(x) = \begin{cases} x, & \text{if } x > 0 \\ 0, & \text{if } x \leq 0 \end{cases} \quad (2.7)$$

ReLU is one of the most widely used activation functions for training deep neural networks. It is simple to implement and computationally efficient, as negative inputs output zero. This makes gradient computation during backpropagation straightforward, with gradients being either 1 or 0. Additionally, ReLU helps mitigate the vanishing gradient problem, a common issue with other activation functions, where gradients become extremely small and effectively disappear, slowing down learning.

- Sigmoid:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \quad (2.8)$$

Sigmoid ranges between 0 and 1, making it normalized and creates non-linearity. This is also nice as it is bound between values, meaning the activation will not be able to output absurd large values. However it is prone to vanishing gradients as the range is small, it is also more computational heavy as it uses an exponential. Sigmoid is also not a zero centered function, which can make gradient updates biased to one direction and therefore make optimization harder.

- Tanh:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \quad (2.9)$$

Tanh ranges from -1 and 1, and is also non-linear and has many of the same properties as sigmoid, as it also normalizes, have bound values and also has a the problem of vanishing gradients. Tanh is however a zero-centered function, which makes it preferable over sigmoid in many cases.

- Linear:

$$f(x) = x \quad (2.10)$$

The Linear activation function is also known as the "no activation function", as the output is always identical to the input. It can be used for models that needs only to solve very simple tasks, but does not perform well for most tasks. It is commonly used on the last layer in a given model, even complex, as it serves as a "no activation function" to output the actual prediction of the neural network.

- Softmax:

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^N e^{z_j}} \quad (2.11)$$

Softmax is a function that calculates the distribution of probabilities for a vector of raw scores. This activation function is most commonly used in the last layer, as they are commonly used in multi-class clarification problems or other problems that has a discrete output space.

2.6.3 Backward Propagation

Backpropagation is a method used in neural networks to propagate errors backward from the predicted output, employing the chain rule to compute partial derivatives. The chain rule allows us to calculate the derivative of the composition of two differentiable functions as the product of their individual derivatives:

$$\frac{d}{dx}[f(g(x))] = \frac{df}{dg} \cdot \frac{dg}{dx}$$

Backpropagation is essential for calculating the gradients of neurons, weights, and biases based on the model's performance, making it a core component of the training process. It is commonly used with gradient descent to optimize weights and biases. The training process involves backpropagating errors over many iterations of data, making small adjustments to the weights and biases for each sample. This iterative process gradually "nudges" the network in the right direction, improving performance over time.

2.6.4 Other Neural Network mechanisms

Neural networks may also include other mechanisms. These mechanisms are employed either to enable solving certain kinds of learning problems, or to improve training efficiency by reducing the time required for the model to converge. Below is a non-exhaustive list of such mechanisms.

Dropout

Dropout is an effective regularization technique used to prevent neural networks from overfitting [12]. Overfitting occurs when a neural network learns the training data so thoroughly that it struggles to generalize to new, unseen data. Dropout addresses this by randomly deactivating (or "dropping out") a subset of neurons during each training iteration, effectively creating a reduced or so called *thinned* version of the network. For each training episode a new *thinned* network is then sampled and trained on a batch of data.

At test time, the full network (without any dropout) is used to make predictions. To ensure consistency between training and testing, the weights of the network are scaled down proportionally to the dropout probability during training. This scaling compensates for the missing neurons during training, ensuring that the expected outputs remain consistent.

By introducing randomness during training and reducing reliance on specific neurons, dropout significantly lowers the risk of overfitting. It achieves this by forcing the network to learn more robust and generalized features, ultimately decreasing the generalization error and improving performance on unseen data.

Residuals

Residuals are a technique designed to address the problem of vanishing gradients, which was discussed earlier. Residual networks (ResNets) were first introduced in the computer vision research paper titled "Deep Residual Learning for Image Recognition" [13]. This issue frequently arises in deeper neural networks, where increasing depth leads to accuracy saturation and eventual degradation. The technique involves adding a *shortcut* (or skip) connection that bypasses one or more layers by directly linking the input to the output, as illustrated in Figure 2.6.

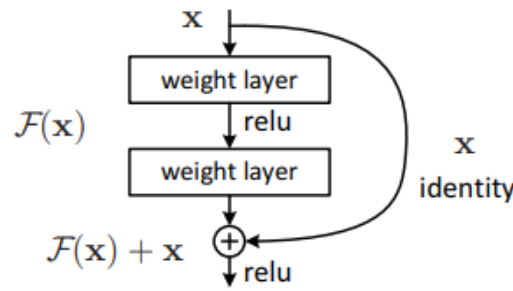


Figure 2.6: Residual learning: a building block^[13].

The key idea behind residual learning is to reformulate the learning objective of the neural network. Instead of learning the desired mapping $H(x)$ directly, the network learns a residual function $F(x) = H(x) - x$. This reformulation simplifies the learning process, particularly for deeper networks, as the residual $F(x)$, is often easier to optimize than the entire mapping $H(x)$. Once the residual function $F(x)$ is learned, the original mapping can be reconstructed as $H(x) = F(x) + x$. The shortcut connection ensures that the identity mapping x is readily available, allowing the network to propagate input information forward with minimal effort and thus avoid vanishing gradients, making residual learning highly effective for training deeper networks.

Memory

In many machine learning problems, the current state is often sufficient to make accurate predictions or take effective actions. For example, in Pac-Man, knowing the current positions of Pac-Man, the ghosts, and the pellets may be enough to compute an optimal move. However, there are scenarios where the immediate state alone does not capture the full context required for decision-making or predictions.

In such cases, memory mechanisms become essential. These mechanisms allow models to retain and utilize information from previous states - this is particularly relevant for sequential data involving temporal dependencies, such as speech recognition or time-series forecasting.

Recurrent Neural Networks (RNNs) address the issue of needing memory by introducing loops within the network, enabling information to persist over time. In this way, the RNNs maintains a hidden state which is used alongside the conventional input. The hidden state is then updated after each inference. However, traditional RNNs often suffer from issues such as vanishing and exploding gradients, which hinder their ability to retain long-term dependencies. To address these limitations, Long Short-Term Memory (LSTM) networks were introduced. LSTM is a specialized type of RNNs that incorporate 'gates': input, output, and forget gates. These gates are used to control the flow of information, by allowing the LSTM to selectively remember or forget information, making them highly effective for tasks requiring long-term memory. This means the LSTM can function, even given long delays between important events. An alternative to RNNs and LSTM is a technique known as frame stacking. Rather than introducing memory into the model, frame stacking transforms the input state by incorporating a sequence of consecutive inputs. For instance, a single input state x becomes $[x_1, x_2, \dots, x_n]$, where n is the number of stacked frames.

While frame stacking does not explicitly provide memory, it gives the model additional context by embedding temporal information into each input state. This method is espe-

cially useful in tasks like video recognition, where stacking frames allows the model to infer motion from consecutive images.

2.7 Reinforcement Learning

Reinforcement Learning (RL) is a machine learning category that is neither supervised or unsupervised, but is in its own category of learning. The three categories are distinct and each used for their own type of application.

2.7.1 Supervised learning

Supervised learning leverages labeled datasets to train models that can predict outcomes and recognize patterns. These datasets consist of input-output pairs, enabling the model to learn the correct mapping between inputs and outputs. The primary objective is for the model to generalize this learned mapping so that it can accurately predict outputs for new, unseen inputs. Common applications of supervised learning include classification tasks (e.g., identifying spam emails) and regression tasks (e.g., predicting house prices).

2.7.2 Unsupervised learning

Unsupervised learning does not use labeled data. Instead, the objective is to identify patterns, groupings, or underlying structures within the dataset. The model learns from the inherent structure of the data without guidance on any "right" answer. Common techniques in unsupervised learning include clustering (e.g., grouping similar customers for targeted marketing) or dimensionality reduction (e.g., simplifying data for visualization or processing).

2.7.3 Reinforcement learning

Reinforcement Learning (RL) involves an agent that learns by interacting with an environment. Instead of using preexisting datasets, the agent explores the environment and receives feedback in the form of rewards or penalties based on the actions it takes. The goal is for the agent to maximize its cumulative reward over time by learning an effective policy (often represented by a neural network) that dictates which actions to take in various situations. Reinforcement learning is used in applications such as robotics, game AI, or autonomous driving.

For this thesis, we will focus solely on reinforcement learning as is the most suited for this application, since there are no applicable datasets. Each episode corresponds to a simulation run, where the gateway gets to explore and and try and receive the most packets. There are many reinforcement learning algorithms, each with their own benefits and suitable application.

Training

Generally, an algorithm for training a model using RL will use the following procedure, split into a series of steps. Different RL algorithms may diverge in various ways from the process.:

- Collect data from an episode: Interact with the environment using the current policy to record states, actions, and rewards.
- Process rewards: Evaluate and, if necessary, transform the collected rewards to account for credit assignment or long-term objectives.
- Calculate the policy update: Compute how the policy should change to improve future performance.
- Update the policy: Adjust the policy parameters based on the calculated update.

These steps are repeated until the policy performs well enough in the environment or a predefined number of episodes are completed.

Collecting data for an episode

The agent collects data by interacting with the environment using its current policy. This includes tracking the sequence of states visited, actions taken, and rewards received.

Determine the rewards for the episode

During data collection, rewards are assigned based on the state and actions. However, at the end of an episode, these rewards are often processed using a discounting mechanism. This emphasizes long-term rewards by attributing future outcomes back to earlier decisions, enabling the policy to learn from delayed consequences.

Policy Update calculation

Policy updates depend on estimating how to adjust the policy to improve performance. Most algorithms calculate a gradient that reflects how the expected return changes with respect to the policy parameters. For example, the update direction often involves terms like the policy's sensitivity to actions ($\nabla_{\theta} \log \pi_{\theta}(a|s)$) and the associated reward signal. Monte Carlo methods or other sampling approaches are typically used to approximate these updates.

Update the policy

The policy parameters are updated using the calculated gradient or its equivalent. A common update rule is:

$$\theta_{t+1} = \theta_t + \alpha \nabla_{\theta} J(\theta) \quad (2.12)$$

Here, θ_t represents the current policy parameters, α is the learning rate, and $J(\theta)$ is the policy objective. Some algorithms instead minimize a loss function, often related to the expected reward, such as the negative advantage or value function error.

2.7.4 Types of RL Algorithms

Reinforcement Learning algorithms have 3 categories to distinguish them. They are model-based vs Model-free, On-policy vs off-policy, and On-line policy vs Off-line policy.

Model-based:

Model-based Reinforcement Learning trains a model that learns the transition probability of states. In other words, it builds a model of the environment by learning the transition dynamics and reward function. The environment is represented as a Markov Decision Process (MDP), defined by states S , actions A , transition function $P(s'|s, a)$, and reward function $R(s, a)$. By approximating these, the agent can simulate experiences and improve decision-making efficiently.

Example: A chess-playing AI can use a model-based approach by building an internal model of the game. It simulates possible future moves and their outcomes before selecting the best move, improving decision-making efficiency.

Model-free:

Model-Free Reinforcement Learning optimizes behavior through trial-and-error without learning transition dynamics. It includes value-based methods like Q-Learning and policy-based methods like Policy Gradient. These approaches are typically less sample efficient but excel in complex environments.

Example: Deep Q-Networks (DQN) learn to play Atari games by directly mapping pixels to actions without modeling the game environment.

On-line policy:

On-line means the algorithm continually gathers new data during training and uses it to learn. When the policy is updated, new data can be gathered and the changes can be

evaluated. If a policy algorithm is strictly online, then it only learns from each data point once before discarding it.

Example: A robot learning to walk updates its walking strategy in real-time based on sensor feedback.

Off-line policy:

Off-line policy means the algorithm uses pre-collected data from a fixed dataset to train the model. This means the policy does not interact with the environment during training. As such, off-line policy algorithms closely resembles supervised learning. If a policy algorithm is strictly off-line, then it must be re-run from start whenever the dataset is changed.

Example: Training a self-driving car policy using a large dataset of recorded driving scenarios without real-time interaction.

On-Policy:

On-Policy means the algorithm only has a single policy π at any time, and gathers only data using π that is being trained. This means the algorithm evaluates and improve the policy iteratively, so the data reflects the most current version of the policy.

Example: PPO updates its policy by collecting experiences, then adjusting the policy gradually using a clipping mechanism to ensure stable learning without large changes.

Off-Policy:

Off-policy means the algorithm trains using data gathered by a different model than the currently trained one. Commonly 2 policies are in use at a time, but sometimes more. These policies are called target policy π and the behavior policy. The behavior policy is used to generate the data, and the target policy is trained based on these observation. If there is too much deviation between the 2 policies, this will slow the learning.

This allows the algorithm to use mechanisms such as "experience replay". This allows the agent to store experiences (state, action, reward, next state) and sample from this memory to update its policy. This improves learning and avoid "catastrophic forgetting".

Example: Q-Learning learns an optimal policy while using an exploratory behavior policy to collect diverse experiences.

2.7.5 Deep Q-Network (DQN)

Deep-Q-Network (DQN) is a Reinforcement Learning algorithm that combines Q-learning with deep neural networks to handle high-dimensional state spaces, such as raw pixel data from games. Q-learning is a value-based algorithm that teaches an agent by assigning values to each action taken in a given state, ultimately learning a function $Q(s, a)$ (the Q-function), which estimates the cumulative reward for taking an action a in a state s .

DQN is a model-free and off-policy algorithm, meaning it does not require a model of the environment. The learning process does not directly take place from experiences as they occur. Instead, the agent stores experiences in an experience replay buffer and samples from it for learning. The Q-network is trained by minimizing a series of loss functions (as defined in Equation 2.13) [14].

$$L_i(\theta_i) = \mathbb{E}_{(s,a,r,s') \sim U(D)} \left[(y_i - Q(s, a; \theta_i))^2 \right] \quad (2.13)$$

where the target value y_i is now defined as:

$$y_i = r_t + \gamma \max_{a'} Q(s_{t+1}, a'; \theta_i^-) \quad (2.14)$$

In these equations:

- $\mathbb{E}_{(s,a,r,s') \sim U(D)}$ denotes the expectation over state-action-reward-next-state tuples (s, a, r, s') sampled uniformly from the experience replay buffer D .
- $Q(s, a; \theta_i)$ is the predicted Q-value for the state-action pair (s, a) , as calculated by the Q-network using the current network parameters θ_i .
- y_i represents the target value used in the loss function, which is computed using the Bellman backup equation.
- r_t is the immediate reward obtained at time step t .
- γ is the discount factor that determines the importance of future rewards.
- $\max_{a'} Q(s_{t+1}, a'; \theta_i^-)$ represents the maximum Q-value for the next state s_{t+1} , which is computed using the target network parameters θ_i^- .

A key limitation of DQN is that it requires a discrete action space, because:

- The Q-function must assign a value to each possible action.
- If the action space were continuous, there would be an *infinite* number of possible actions, making it impossible to evaluate all of them.

For continuous action spaces, *policy-based* methods such as **Proximal Policy Optimization (PPO)** are more suitable.

2.7.6 Proximal Policy Optimization (PPO)

Proximal Policy Optimization (PPO) is a model-free and on-policy RL algorithm. The algorithm is widely used for training agents in continuous and high-dimensional action spaces. Unlike DQN, PPO uses policy gradients to learn, instead of Q-learning. It does so by trying to optimize a policy function $\pi(a|s)$, which is the probability distribution over actions a for a state s , and aims to maximize the expected cumulative rewards. The goal is to maximize the expected cumulative rewards by learning a policy that maps states to actions, rather than estimating value functions as in Q-learning.

PPO achieves this by using a surrogate objective function, which depends on both the old and new policy parameters as well as the advantage estimate. The advantage estimate is typically derived from a *value network* (Another neural network) that predicts the expected return starting from a given state. This surrogate objective serves as the loss function and is minimized to optimize the policy. The function can be seen in equation 2.14 [15].

To ensure stable training, PPO also employs clipping, which restricts large updates to the policy. By preventing drastic policy changes, clipping helps maintain training stability and sample efficiency.

$$L^{\text{PPO}}(\theta) = \mathbb{E}_t \left[\min \left(r_t(\theta) \hat{A}_t, \text{clip} \left(r_t(\theta), 1 - \epsilon, 1 + \epsilon \right) \hat{A}_t \right) \right] \quad (2.14 \text{ [15]})$$

- \mathbb{E}_t refers to the expectation over time steps t in the trajectory. This expectation is taken over a distribution of states, actions, and rewards from the agent's experience.
- $\pi_\theta(a|s)$: The probability of taking action a in state s , according to the current policy with parameters θ .
- $\pi_{\theta_{\text{old}}}(a|s)$: The probability of taking action a in state s , according to the old policy with parameters θ_{old} (used for clipping).

- $r_t(\theta)$: The probability ratio between the new policy and the old policy, defined as:

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)} \quad (2.15)$$

- ϵ : The clipping parameter that controls how far the new policy is allowed to deviate from the old policy.
- \hat{A}_t : The advantage estimate, representing how much better an action a_t is compared to the expected action at state s_t . This is typically calculated as:

$$\hat{A}_t = \delta_t + \gamma\delta_{t+1} + \gamma^2\delta_{t+2} + \dots \quad (2.16)$$

where δ_t is the temporal-difference error, and the advantage is calculated using the *value network*.

PPO is categorized as an **actor-critic** algorithm. This means it consists of two main components:

- **Actor**: The actor is responsible for selecting actions based on the current policy. It is typically represented by a *policy network*, which outputs a probability distribution over actions given a state. This allows the agent to take actions according to its learned policy.
- **Critic**: The critic evaluates the actions taken by the actor. It provides feedback on the quality of the actions, usually in the form of value estimates. The critic can be represented by a *value function*, such as a state-value function $V(s)$, which estimates the expected return (future rewards) starting from a particular state.

In PPO, the **actor** is the *policy network*, which decides which actions to take based on the observed state, while the **critic** is typically a *value network* that estimates the value of the current state. The critic helps the actor by providing feedback that improves the policy over time.

A good analogy for DQN and PPO could be where the agent is a child on a playground, and the goal is to help the child figure out how to maximize their fun. In this analogy, DQN acts like a guide who helps the child build a list of possible actions, but instead of suggesting what to do, the guide gives the child the tools to figure out for themselves which activity is the best based on past experience. The child goes on the swing and the guide says, "You had this much fun," and then the child tries the slide, and the guide says, "This was better (or worse)."

The same "child on playground" analogy can be done with PPO. The PPO approach is like the coach refining the child's policy (deciding which activities to do based on the state of the playground) through feedback, ensuring that they have fun in a stable, efficient manner. The coach's feedback helps the child decide what to do next, but it allows room for a little randomness in their decisions, encouraging the child to try new things while gradually learning what activities are the most fun.

2.7.7 Credit Assignment Problem

One of the biggest challenges in Reinforcement Learning) is the *credit assignment problem*, which focuses on how to allocate rewards and penalties to an agent. The way these credits are assigned significantly impacts the efficiency and effectiveness of an agent's learning process. A poor credit structure can result in the agent learning undesirable behaviors or failing to achieve its objectives altogether.

Credits, which can be positive (rewards) or negative (penalties), need to be assigned thoughtfully. It's not just about rewarding the agent for correct behavior, but also about ensuring the reward magnitude is appropriate. A key example of this challenge can be seen in the game *Snake*, where the goal is for the snake to eat as many apples as possible. While it's effective to reward the agent immediately after it eats an apple, this alone can be insufficient. Without additional guidance, the agent may learn to wander aimlessly, waiting for apples to appear rather than optimizing its movement toward them.

To address this, one could provide feedback based on the snake's proximity to the next apple. The agent could be penalized for being far away or rewarded for getting closer. This intermediate feedback could steer the agent toward apples more effectively. However, there are potential pitfalls. For example:

- **Local Minima:** If the snake is overly punished for being too far from the apple, it may find that staying still or killing itself minimizes the penalty, which results in poor performance.
- **Overemphasis on Proximity:** On the other hand, if the reward for being near an apple is too large, it might cause the snake to focus too much on proximity and not enough on actually eating the apple. This could lead to the snake hovering around the apple endlessly without ever collecting it, which also hinders progress.

Finding the right balance between rewards and penalties is crucial. If penalties are too harsh, the agent may avoid taking risks and become stuck in suboptimal solutions. If rewards are too generous, the agent may fixate on specific behaviors (like proximity) at the cost of achieving the true goal, which is maximizing the score by eating apples.

This brings us to another challenge in reinforcement learning: balancing *exploration* (trying new behaviors) and *exploitation* (maximizing known good behaviors). If the agent faces too much penalty for exploring, it might miss out on discovering better strategies. Alternatively, if rewards for exploration are too strong, the agent might spend too much time experimenting and not enough time optimizing behavior.

Solving the credit assignment problem often requires careful reward design and experimentation. Techniques like *reward shaping*, where intermediate rewards are used to help gradually guide the agent through different stages of behavior. For instance, rewarding the agent for moving closer to an apple, then rewarding it for reaching the apple, and finally rewarding the successful act of eating the apple helps form a more complex, goal-oriented strategy.

Ultimately, effective credit assignment involves fine-tuning the reward and penalty structure, ensuring that the agent's behavior aligns with the desired outcomes. This thesis will present the credit assignment in our environment, as well as some of the reward structures that were ultimately not included.

2.7.8 Policy representation

The gateway needs to be given the policy after the model has been updated after a training run. There is multiple ways to represent the policy, using different data structures. The 'simplest' and most standalone is to implement the policy into a XML file, and have a custom mobility component which can run the policy.

Another option is to use an external library which can represent the policy as well as to execute it. Using external libraries will provide optimizations as they are specifically made to that purpose. Some libraries which looked suitable:

- **ONNX:** An open format built to represent machine learning models. ONNX defines a common set of operators - the building blocks of machine learning and deep learning models. And a common file format to use models with a variety of frameworks, tools, runtimes, and compilers.
- **libtorch:** is a PyTorch C++ frontend, which was designed with the idea that the Python frontend is great, and should be used when possible. The goal of the C++ frontend is to make low latency and high performance environments more feasible, while not sacrificing the user experience of the Python frontend.
- **Lite Runtime (LiteRT):** is formerly known as TensorFlow Lite, is Google's high-performance runtime for on-device AI. It has key features such as; Multi-platform support, Diverse language support (C++, Python and more) and High performance. This makes it ideal for lightweight, efficient and/or real-time rendering solutions.

2.8 Stable-Baselines3 (SB3)

Stable-Baselines3 (SB3) is an open-source framework for reinforcement learning, developed by the community and designed to implement a variety of model-free policy optimization algorithms [16]. The framework has undergone three versions, with the latest version utilizing PyTorch, and all versions being forks of OpenAI Baselines. Stable-Baselines3 is supported exclusively for Python. In addition, certain experimental or niche features are not included in the main framework but are available in the `sb3-contrib` repository.

To use SB3, the developer must follow the steps outlined below:

- Create a custom environment that defines both the agent's actions and how the environment is affected by these actions, thereby determining the environment's input state (observations).
- Choose a policy optimization algorithm that is compatible with the environment.
- Select a neural network architecture that is suitable for the chosen algorithm.

The environment is implemented as a class containing three core methods: `__init__()`, `step()`, and `reset()`.

- `__init__()` sets up the environment's properties, including the dimensionality and range of the observation space.
- `step(action)` defines the agent's response to the numeric value derived from the model's inference. This includes changes due to both the agent's actions and the natural progression of time, depending on how the environment is designed. Consequently, how much time that passes between each `step` depends on the design. As long as chronological events are executed for the time that occurs between actions.
- `reset()` is used to generate a fresh initial state. Reset is invoked to generate the first state, and also at the end of each episode to reset the environment.

One of the key advantages of SB3 is its separation of concerns between the environment, policy, and neural network. By designing only the environment, developers can integrate any policy algorithm and neural network architecture, as long as the input and output spaces are compatible. For example, the Proximal Policy Optimization and Deep-Q-Network algorithms are compatible with the following types of action- and observation-spaces, as summarized in Table 2.1 and 2.2 .

Space Type	Action	Observation
Discrete	✓	✓
Box	✓	✓
MultiDiscrete	✓	✓
MultiBinary	✓	✓
Dict	✗	✓

Table 2.1: SB3 PPO compatibility for different space types

Space Type	Action	Observation
Discrete	✓	✓
Box	✗	✓
MultiDiscrete	✗	✓
MultiBinary	✗	✓
Dict	✗	✓

Table 2.2: SB3 DQN compatibility for different space types

The policy networks in Stable-Baselines3 consist of two main components: a feature extractor and a fully connected network, as illustrated in Figure 2.7. The feature extractor processes the observations to extract relevant information. For example, in a video game, the observations could be raw pixel data. The feature extractor would then identify key elements within the scene, such as enemies, allies, points, or other important objects. This extracted information is then passed into a fully connected network, which uses it to predict optimal actions in the game.

By default the network architecture consists of 2 fully connected layers each with 64 neurons when using PPO or DQN. Each layer also uses the tanh activation function by default.

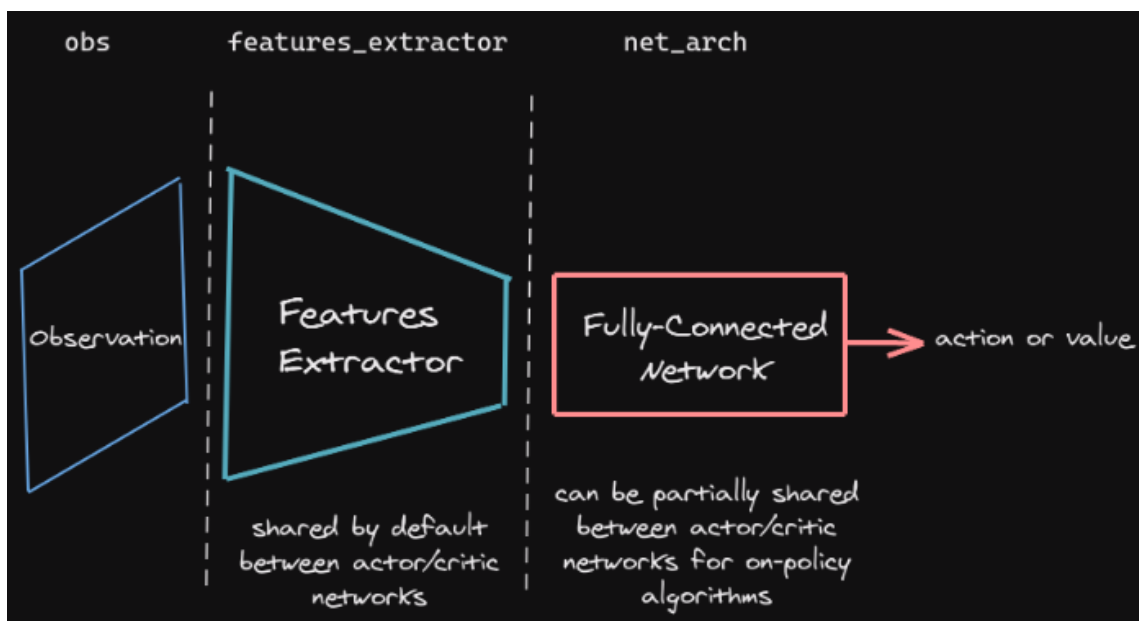


Figure 2.7: Stable-Baselines3 policy network architecture [17]

3 Related Works

This chapter reviews recent research and relevant literature related to this project, with a focus on LoRa and Reinforcement Learning. It will begin with an overview of studies and research on LoRa, particularly in the context of mobile gateways and scaling LoRa networks. Following that, the chapter shifts to Reinforcement Learning, highlighting key studies on the use of Deep-Q-Network (DQN) and Proximal Policy Optimization (PPO). It also examines the literature on reward shaping, emphasizing its role in improving the performance and learning process.

3.1 LoRa Literature

New literature and research for LoRa is continuously being published, as it is still an evolving technology with much potential. Researchers are therefore putting in effort and time into determining these possibilities of what LoRa can be used for, as well as possible limitations and how to overcome them. One study evaluates the use of LoRa in wireless sensor networks [18]. It presents real-world experiments assessing communication between LoRa nodes, focusing on LoRa wireless performance, LoRaWAN reliability, and overall network efficiency. The findings highlight that deploying multiple gateways significantly enhances coverage, particularly in challenging topographical areas where terrain impacts connectivity. Another study evaluated LoRa for the use in open field cultivation scenarios [19], which was simulated using FLoRa. The paper presents a series of different scenarios where antenna types, number of gateways and number of nodes is explored. Here it is discovered that increasing the number of gateways from 1 to 4, increases the performance by 28.6%, and increasing the number of nodes from 10 to 1000 will increase the percentage of collisions from 1.7% to 65.7%.

Addressing the problem of collisions when more nodes are introduced, Semtech LoRa documentation [5] highlights their enhanced network capacity which they achieve through their modulation technique. This is accomplished by employing orthogonal spreading factors, which enable multiple signals with different spreading factors to be transmitted simultaneously, even on the same channel. Signals modulated with other spreading factors appear as minimal noise to the target receiver, allowing for efficient utilization of the channel without significant degradation in receiver sensitivity.

Besides Semtechs documentation, there are several studies investigation collisions avoidance and detection for LoRa and LoRaWAN. One study investigates the use of supervised learning for collision detection in LoRaWAN [20]. By using *decision trees* and *random forest* algorithms to trained to predict whether collisions occurred. The research showed that *random forest* performed best with a precision of 0.95, and a accuracy of 0.96. They conclude that they plan to also explore the use of neural networks in the future, as they anticipate it could deliver great results in more complex settings.

3.2 Mobility for LoRa Literature

The literature addressing mobility in LoRa gateways is hard to come by, especially AI driven. However one study addresses the use of mobility in gateways for livestock smart monitoring [21]. The paper proposes the use of one mobile LoRa gateway for livestock monitoring. The experiments was done using a discrete-event based simulator called LoRaSim, which is a python-based environment, and these simulation tool was then used to simulate environments where static and mobile gateways was used. In the paper the

mobility presented is a straight forward mobility that moves the gateway back and forth, with additional stops. It was concluded that this type of mobility is best suited for vast wide livestock areas, and introducing multiple stops for the mobile gateway will not only improve energy consumption but also the Data Exchange Rate (DER).

A similar study was conducted for Simulation of Mobile LoRa Gateway for Smart Electricity Meter [22], The paper suggests the use of a mobile gateway on a motorcycle which moving in a residential area with a series of nodes to collect data from smart meters. The mobility of the gateway follows a simple straight vertical movement. To perform the study they, as the previous study, also used LoRaSim to test and experiment different scenarios. They also see a large amount of collisions for scenarios with large number of nodes, stating that the challenge is to choosing certain times to send where collisions are less likely to occur, however from the experiments it could also be concluded that using a mobile gateway can provide better coverage, minimizing how many gateways are needed. Furthermore, they also found that increasing amount of nodes will also increase the power consumption when they investigated the Network Energy Consumption.

One study explored the use of a mobile LoRa gateway to minimize power consumption in Low-Power Wide Area Networks [23]. The paper introduces LoRaDrone, a mobile system leveraging Unmanned Aerial Vehicle (UAV)s as gateways. By flying closer to IoT nodes, the UAVs reduce the required transmission distance, allowing devices to successfully transmit data using lower transmission power. Since energy consumption in wireless communication is directly linked to transmission power, this reduction significantly decreases the overall energy usage of IoT devices. To achieve this, the authors focus on two primary elements: a mobile approach using UAVs for proximity-based communication and a low-power communication mechanism designed to optimize data transmission efficiency. The evaluation of LoRaDrone is conducted through various simulation scenarios, where results show that, compared to a static gateway, the system improves energy efficiency by up to 70.37 times.

3.3 Reinforcement Learning Literature

The literature on Reinforcement Learning is vast and dense, as it applies to a wide range of problems. Reinforcement Learning encompasses various tools and techniques that can optimize or enhance model learning. Researchers continuously study and document how different methods perform across diverse problems and environments. One study compares Reinforcement Learning algorithms across different games [24]. The study focuses on three Reinforcement Learning algorithms: DQN, PPO, and Actor Critic and Experience Replay (ACER) which is evaluated over 19 games. The results show that DQN achieves the highest overall score compared to PPO and ACER, but the paper also highlights its drawbacks. While DQN achieves the best average performance, it also exhibits the highest variance, leading to unstable results. This issue is particularly evident in games that require a consistent score. To address this, the paper proposes adjusting the learning rate and discount factor to improve stability, though this may increase the time required for learning. Alternatively, the study suggests using other algorithms, such as PPO or ACER, to achieve more stable performance.

Another study, investigated the use of DQN and PPO for navigation in autonomous vehicles [25] The study used simulations to create environments with 3 different environments; urban, suburban and highway, and different traffic density and weather conditions. The study found that while both DQN and PPO demonstrated good performance and effectiveness in different self-driven scenarios, it was PPO that significantly outperformed DQN

across all scenarios. The study shows that PPO has a $\sim 4\text{-}7\%$ higher completion rate than DQN across all scenarios of different environments and conditions.

From the studies and research, it can be difficult to distinguish and find out when a certain algorithm would be most appropriate for a problem and environment. One study aims to compare these algorithms to find the characteristics of where the algorithms thrive [26]. The study compares the use of three algorithms; PPO, DQN and A2C, to solve a game called BreakOut Atari. The paper presents their main insights that were found between the use of the three different algorithms. One of the more relevant to this project is that DQN works well if the environment has a clear immediate form of rewards, where PPO and A2C perform better when there is more reward exploration and complex strategies are needed for the environment. The study also found that DQN is more resilient to change on hyperparameters, where PPO and A2C needs more fine tuning to get good results.

A survey on model-based deep Reinforcement Learning explores its advantages in high-dimensional problems [27]. It highlights that model-based approaches are more sample-efficient than model-free methods, a crucial factor in robotics, where data collection is costly. However, high-dimensional tasks require large neural networks, increasing data demands and reducing efficiency. The study emphasizes that learning transition models first allows for better generalization to unseen problems. To address the curse of dimensionality, it proposes latent models, which replace a single complex transition model with multiple specialized, compact models, improving efficiency. Additionally, the survey discusses curriculum learning in self-play, where gradually increasing difficulty helps policies learn progressively. Overall, the study concludes that no single Reinforcement Learning approach is universally best, but model-based methods hold promise for tackling complex problems more efficiently.

A study [28] that builds on similar ideas to LoRaDone [23] explores resource allocation for multiple UAV gateways using Multi-Agent Proximal Policy Optimization (MAPPO) to optimize energy efficiency (in bits per joule). The gateways remain stationary, while mobile nodes move in random directions, potentially switching between gateway coverage areas. Each gateway selects the appropriate SF, TP, and Bandwidth while only observing a subset of nodes, dynamically allocated for load balancing. This partial observability requires cooperation among gateways. The study employs Centralized Training, Decentralized Execution (CTDE), where the critic network has full state access, but individual agents act based only on local observations. Comparing their approach with other RL methods, the results show a 30.5% improvement in energy efficiency for 20 nodes, and 11% improvement for 60 nodes. This demonstrates both feasibility of RL in LoRa applications, and the benefits of coordinated decision-making in multi-agent RL.

Lastly, a study on transfer learning in RL [29] introduces Automatic Curriculum Transfer from Simple to Complex Environments (ACuTE), a framework designed to improve Reinforcement Learning by facilitating policy transfer. ACuTE starts by training in a simplified low-fidelity environment, where curriculum generation and experimentation are computationally efficient. This low-fidelity environment retains core structural similarities to the high-fidelity environment but reduces its complexity. An optimized curriculum is generated in the low-fidelity environment and then mapped to the high-fidelity environment. The curriculum is progressively learned before being transferred to the final target task. Finally, the learned policy is transferred to a physical robot. This approach addresses a common challenge in traditional Sim2Real methods, where curriculum generation often requires more time than directly learning the task from scratch. Experimental results demonstrate that ACuTE achieves faster convergence and improved jump-start perfor-

mance compared to baseline approaches. Its ability to generalize across fidelity levels, despite noisy mappings, highlights its potential for efficient and scalable transfer learning. The authors suggest extending ACuTE to multi-agent systems, enabling inter-agent curriculum transfer for more complex collaborative tasks.

3.4 Reward Shaping Literature

Reward shaping is a technique in Reinforcement Learning that helps guide an agent's learning by modifying the reward structure to make the desired behavior easier to discover. It addresses the credit assignment problem by providing intermediate rewards that highlight steps toward the goal. A critical aspect of reward shaping is policy invariance, which ensures that changes to the reward structure do not alter the optimal policy. When done correctly, reward shaping accelerates learning while preserving the agent's ability to converge to the optimal solution.

One study investigates how changes to the reward function affect the optimal policy for a reinforcement learning model [30]. The paper explores the conditions under which modifications to the reward function in a Markov Decision Process (MDP) preserve the optimal policy. The authors demonstrate that, in addition to the well-known positive linear transformation from utility theory, it is also possible to add a reward for transitions between states, provided that the reward is expressible as the difference in the value of an arbitrary potential function applied to those states. Moreover, this transformation is shown to be a necessary condition for invariance, meaning that any other reward transformation could result in suboptimal policies unless additional assumptions about the underlying MDP are made. The paper also highlights some common issues ("bugs") that arise in reward shaping procedures, particularly when non-potential-based rewards are used. It also proposes methods for constructing shaping potentials that correspond to distance-based and subgoal-based heuristics. The authors show that these potentials can lead to significant reductions in learning time, improving the efficiency of the agent's training process.

Another paper [31] argues that policy invariance becomes particularly relevant when unobserved variables are present. Through simulations, the authors verify their theoretical findings, demonstrating that if all relevant covariates are observed, causality and invariance are not necessary for obtaining distributionally robust policies. However, these factors become significant when some variables remain unobserved. To address such cases, the authors adapt ideas from causal inference and introduce the notion of invariant policies.

Their theoretical results show that, under certain assumptions, an invariant policy that is optimal in the training environments also remains optimal in unseen environments, ensuring it is distributionally robust. Additionally, they propose a method for discovering invariant policies through an off-policy invariance test, which can be combined with any existing policy optimization algorithm to learn the optimal invariant policy.

4 Methodology

This chapter outlines the implementation and design of the project, covering both the configuration of OMNeT++ and the setup of the custom environment in SB3. It explains the design choices for key components, including how the environment is structured, how rewards are shaped, and how training is conducted. By breaking down these elements, the chapter provides a clear understanding of the methodologies adopted to integrate mobility in OMNeT++ with reinforcement learning and training in SB3.

4.1 Setup

The project consists of two separate environments: OMNeT++ and SB3. SB3 is used for model training in a simplified environment, where the final trained model can be exported and deployed in OMNeT++ for inference. While experiments with training directly in OMNeT++ were conducted (as visualized in Appendix A), this approach proved too time-consuming to be practical for this project. Therefore, the training process was conducted in SB3, where mobility, LoRa, and lower-level components were either abstracted away or simplified to speed up training while preserving essential decision-making elements.

To determine whether AI could be effectively integrated within OMNeT++, an initial Proof-of-Concept (PoC) was carried out. Alongside this, a literature review was conducted to establish a foundational understanding of OMNeT++, LoRa, and Reinforcement Learning, ensuring familiarity with the fundamental theories and existing research relevant to the project.

Following the PoC and literature review, a selection process was undertaken to identify suitable frameworks and tools for enabling AI-driven mobility. Since no "out-of-the-box" solution existed, various tools and frameworks were tested to assess their capabilities and limitations. The main requirements for selection were the ability to integrate with OMNeT++ while also supporting efficient model training. The final choice was made based on compatibility, adaptability, and computational efficiency.

Once the appropriate frameworks were identified, a low-fidelity simulation environment was developed to facilitate training. Initially, it consisted of two stationary nodes and a mobile gateway moving along a single axis. After verifying that a mobility model could adequately handle this scenario, the simulation was expanded to a two-dimensional space. Subsequently, additional complexity was introduced by incorporating four nodes with randomized positions and duty cycles. This incremental scaling ensured that the model could generalize to more complex network conditions while maintaining reliable performance.

4.2 Custom Environment in Stable-Baselines3

A custom environment has been created for this thesis. The objective is to design a minimal environment compared to OMNeT++, which significantly accelerates the learning process while allowing easier and more flexible modifications and implementations. Our custom 2D environment consists of several key elements, which will be discussed. First, the visual aspects of the environment are described, followed by the technical details, such as how nodes are represented and how transmissions are performed.

To support development, a rendering of the environment was created. This rendering was instrumental in verifying the components during implementation and observing the

agent's behavior after training. The rendering is shown in Figure 4.1. The environment is depicted as the area within the red box, while additional information is displayed below.

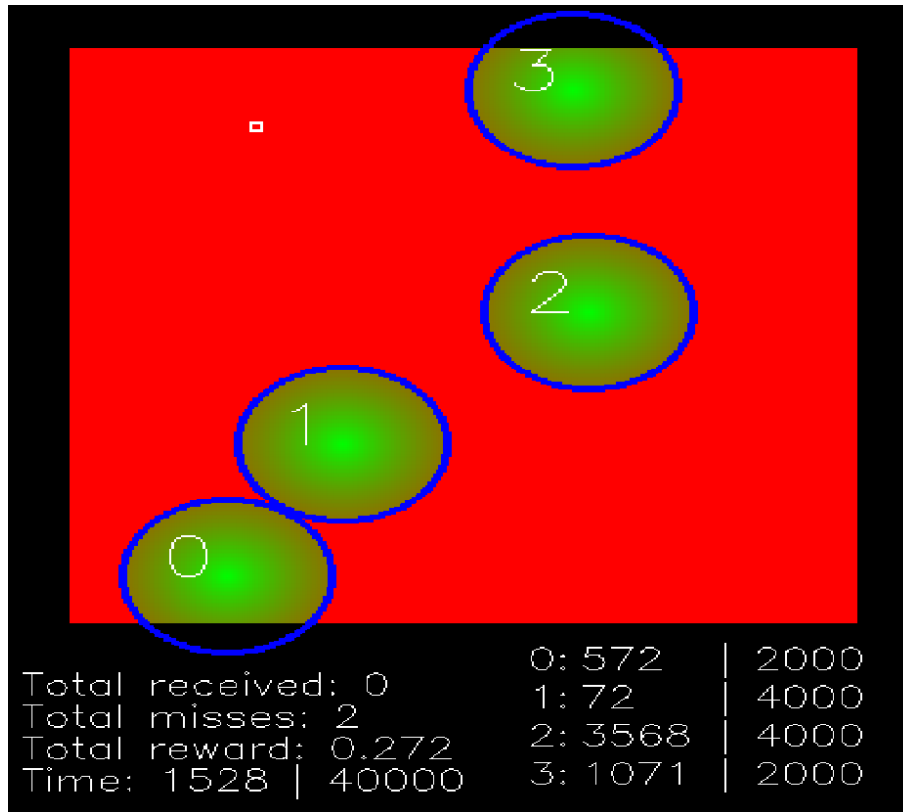


Figure 4.1: Custom environment in SB3

Some key elements can be seen in the rendering:

- A discrete state space as a grid, visualized as a red box.
- The gateway (agent), visualized as a white rectangle.
- Four nodes, visualized with numbering and blue circles filled with green.

The gateway is not implemented as a class directly. Instead, it is represented by a position that, at each step, "listens" for incoming packets. In other words, it checks during each step if any packets have been received from the nodes. The nodes are individual classes that control when they transmit and the extent of their transmission radius. Consequently, the gateway does not have its own "radius"; it relies solely on the transmission radius of the individual nodes. This abstraction simplifies the process of verifying whether the gateway is within range in our custom environment.

Nodes and Transmissions

In our custom environment, each node is defined by several key attributes: *position*, *time of first packet*, *transmission interval*, and *transmission model*.

- The ***time of first packet*** specifies the step at which the node transmits its first packet. For example, a value of 150 means that the first packet is transmitting at step 150.
- The ***transmission interval*** defines the frequency with which a node transmits packets, measured in steps. The ***transmission spacing*** refers to the time (in steps)

between any two consecutive transmitted packets in the environment.

Time in the environment corresponds directly to discrete steps, where each step increments the time by 1. This can also be interpreted as seconds in the environment, as the agent moves at a fixed speed (meters per step, which represents meters per second). However, for consistency, we refer to this progression as steps throughout this thesis.

After the first packet is transmitted at the designated time, the subsequent transmission times are computed based on the transmission interval, with some added random variation. For this a truncated normalization function is used which takes the transmission interval and a standard deviation of 5 steps. A distribution for a transmission interval of 1500 can be seen in Figure 4.2. This variation is introduced using a small distribution, ensuring that the intervals between packet transmissions are not fixed but have a degree of randomness to mimic real-world transmission conditions. The resulting transmission schedule allows for a more dynamic and realistic representation of packets transmitted in the environment.

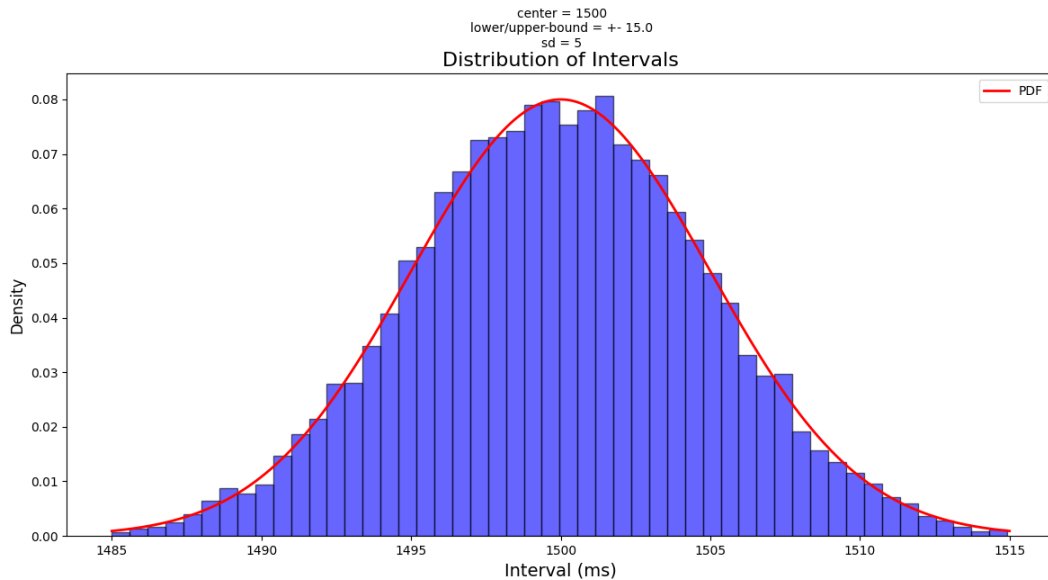


Figure 4.2: Distribution of transmission interval at 1500 steps with a standard deviation of 5

Resetting or creating the environment randomizes the positions of both the nodes and gateways, as well as the transmission intervals for the nodes. This ensures that the environment changes with each episode, aiming to prevent overfitting to specific times and positions. Positions of nodes are also prevented from having overlapping transmission radius when placed, which makes the environments less likely to be "clumped" together and force the nodes to be spread out. The objective is to train a model capable of observing the given positions and estimating packet times, enabling it to efficiently navigate between the nodes.

The node also includes a *Transmission model*, which is a class that determines whether a packet is successfully received. While packets may be sent, there is no guarantee they will be received. This depends on several factors. First, if the gateway is outside the transmission radius of a node, the packet will always be missed. Second, even within the transmission radius, the likelihood of receiving a packet decreases with increasing dis-

tance from the transmitting node. In other words, a gateway further away will experience a higher probability of packet loss. This approach is designed to provide a simplified representation of real-world behavior. The probability function used can be seen in Equation 4.1.

$$P(\text{Packet Reception}) = e^{-\frac{\text{distance}}{\text{ploss scale}}} \quad (4.1)$$

Here, *ploss scale* is a configurable parameter that adjusts the probability scaling. This value can be set based on the transmission radius, and in our experiments its set at 50, with the transmission radius of each node being set to 40. This means that at there is 44.9% probability that a packet will be received if the gateway is on the edge of the transmission radius, Figure 4.3 shows a graph of the probability.

The value of *ploss scale* was chosen as this value, as it provided an aggressive loss in probability further away, with the goal of making this behavior more comprehensible for the gateway during training. The same reasoning applies for the hard cutoff outside the transmission range, where packets cannot be received, when in reality there would still be a chance to receive such packets.

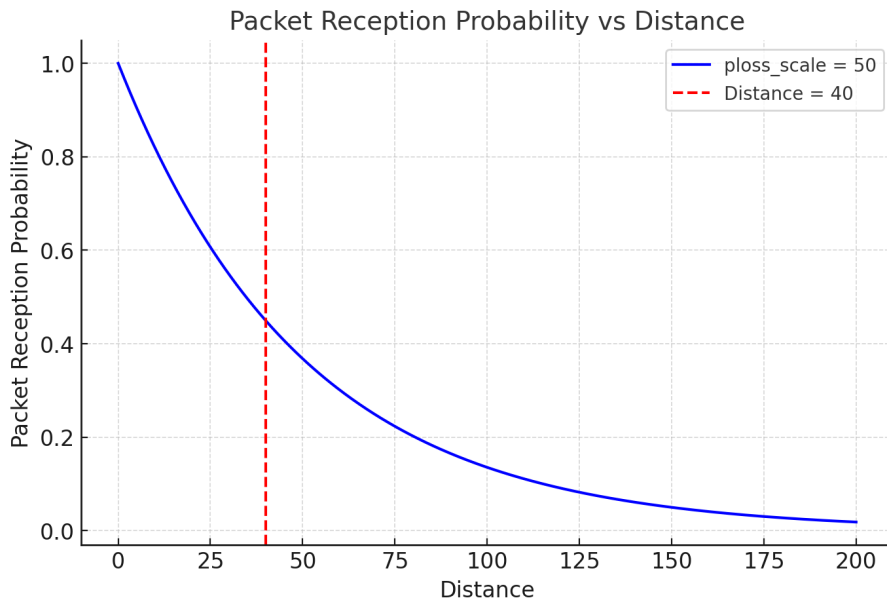


Figure 4.3: Packet Reception Probability

4.3 Policy

The policy represents the neural network that the gateway (agent) uses to make movement decisions based on its observations. Defining these observations is crucial, as they determine what the gateway "sees" and "knows." The information provided should be both meaningful and realistically obtainable.

For this, the **default PPO policy network architecture** is used, with the exception that it consists of three fully connected layers for the *actor* and *critic*, as this improved learning efficiency.

The observation space is defined as:

$$\text{obs}(\Delta t, \theta, \Delta d) \quad (4.2)$$

where:

- Δt : The expected time until each node transmits a packet.
- θ : The direction from the gateway to each node, measured in degrees.
- Δd : The distance between the gateway and each node.

In an environment with 4 nodes, the observation space consists of 12 values, as each input to the state is proportional to the number of nodes. These observations are normalized to bring all values into a similar scale (e.g., between 0 and 1), allowing the agent to treat all inputs equally and preventing certain features from overshadowing others.

To achieve this normalization:

- The distance (Δd) is normalized by dividing it by the maximum distance in the environment.
- The direction (θ) is normalized by dividing it by 360.
- The expected time (Δt) is normalized by dividing it by the largest expected time interval in the environment.

This normalization ensures that all features contribute equally to the learning process, allowing the agent to generalize better across different situations.

From the observations the agent has 5 discrete actions to choose from: *Up*, *Down*, *Left*, *Right* and *Standing still*. During testing, it was observed that the agent learned faster and more effectively when each action had a greater impact. Instead of taking an action at every step, the agent was set to take one action every 10 steps. This approach increased the significance of each action, encouraging the agent to better understand the environment and make good or bad actions more noticeable.

For successful packet reception, the gateway must reach a position with a reliable connection. Since the environment lacks obstacles that could impede signal propagation, the only factor affecting reception probability is the distance between the mobile gateway and the transmitting node. To achieve optimal performance (receiving all packets) the gateway must maintain a short distance from the transmitting node at each transmission time. Thus, for the gateway to receive all packets in a given scenario, the distance between sequentially transmitting nodes must be small enough to allow the gateway to move into favorable positions in time.

From this, we derive:

$$\Delta t_{T1,T2} = \frac{\text{distance}(\text{node}_{\text{transmitting},T1}, \text{node}_{\text{transmitting},T2})}{v_{\text{gateway}}} \quad (4.3)$$

where:

- $\Delta t_{T1,T2}$ is the transmission spacing, the time required for the gateway to move between two sequentially transmitting nodes.
- $\text{distance}(\text{node}_{\text{transmitting},T1}, \text{node}_{\text{transmitting},T2})$ is the Manhattan distance between the two transmitting nodes.
- v_{gateway} is the speed of the mobile gateway.

Since the gateway moves only along the x- or y-axis, the Manhattan distance is used.

4.4 Credit Assignment

Credit assignment for our agent in the environment has been tested and investigated thoroughly, with many different iterations to find what worked best. Initially, rewards were given for successfully receiving packets, which is one of the more obvious rewards, as the agent's main goal is to receive the most packets. Additionally, the agent is punished for missing a packet, as this indicates that the agent was not in the correct position to receive it due to incorrect actions taken.

It was discovered that varying the punishment for a missed packet depending on how far away it was from the node transmitting it greatly improved learning. This scaling of the penalty reduces the negative impact if the missed packet was closer to the agent, making the feedback more nuanced.

However, these two credits rewards for receiving packets and penalties for missed packets, were not sufficient for the agent to effectively learn the environment. Therefore, it was decided to further guide the agent towards the node that was likely to transmit a packet soonest. This was achieved by providing a small reward based on how close the agent was to the node that would transmit next.

To summarize, the reward structure used in the final design consisted of:

- Reward for successfully receiving a packet.
- Penalty for losing a packet (scaled based on distance).
- Small reward or penalty, based on distance to the nodes that will transmit next.
- Small reward for taking an action in the direction of the node that is about to transmit a packet

Additionally, several other reward shapes were tested but ultimately not included in the final structure:

- Bonus reward for consecutive packet receptions without missing any.
- A declining reward if consecutive packets are received in a row from the same node.
- Bonus reward for packet reception based on fairness.
- Fixed reward given after X actions following the reception of a packet.

These additional reward structures were designed to try and encourage the agent to explore and prevent it from getting stuck serving only a subset of nodes. However, after finding an effective reward structure and appropriate hyperparameters, it was determined that these additional rewards were unnecessary for our final model.

4.5 Integrating OMNeT++ for Model Evaluation

To evaluate models trained in the simplified environment of Stable-Baselines3, we utilize OMNeT++ (section 2.3), a highly refined network simulation tool. By integrating the FLoRa framework (section 2.5), we can accurately emulate the unique physical signal properties of LoRa technology. This approach allows us to transition a trained model from low-fidelity environment to a high-fidelity one, ensuring its effectiveness in more realistic simulations.

Implementing this simulation was a crucial aspect of the project. The configuration integrates multiple frameworks within OMNeT++, including INET and FLoRa, alongside TF-lite-micro. To ensure integration and compatibility, it was essential to establish clear

relationships between the components across these frameworks. In Appendix B, a diagram giving an overview of the relation between components can be seen.

4.5.1 Inference Library

Each framework mentioned in section 2.7.8 was evaluated for compatibility with OMNeT++. However, only a variant of TensorFlow Lite (TFLite), specifically TensorFlow Lite Micro (TFLite Micro), was successfully adapted. TFLite Micro A key compatibility issue arose due to OMNeT++'s toolchain, which does not support libraries requiring the "flat-buffers" dependency. To work around this, a prebuilt version of TFLite Micro, originally designed for the Arduino system, was used¹. This version of TFLite Micro was already compiled into standalone C++ source and header files, making integration straightforward. Adapting it for OMNeT++ required only minor modifications: references to Arduino peripherals and compiler directives (`#ifdef`) were removed, along with redundant Arduino example code. However, it is worth noting that this version of TFLite Micro is outdated compared to the latest upstream release, now also referred to as LiteRT Micro².

Coincidentally, TFLite Micro is a lightweight version of TensorFlow Lite designed for DSPs, microcontrollers, and other low-memory devices. This not only makes it compatible with OMNeT++ but also highly relevant for real-world deployment.

SB3 provides guidelines for exporting models, including support for TFLite, though it requires multiple conversion steps. For our case, modifications were necessary to the export process. Since SB3 is based on PyTorch, converting a model to TFLite requires an intermediate step through TensorFlow (TF). To achieve this, an equivalent neural network must be constructed using the TF API, after which the weights and biases from the original model are copied over. The final TF model is then converted to a TFLite file using TensorFlow's built-in `TFLiteConverter`. At runtime, this TFLite file is used for inference in OMNeT++.

To ensure correctness, both the TF and original models were tested on the same randomly generated dataset, verifying that their outputs align and confirming the validity of the conversion process.

4.5.2 File structure

Configuration of the environment follows common practice for (relatively) small projects in OMNeT++. As FLoRa is the top-level framework in the project, it is where the component attribute values are specified, in a `omnetpp.ini` file. The composite modules such as nodes, gateways and Radio mediums are described in a network in a `.NED` file. Then the modules are contained in their respective projects; Custom LoRa gateway, nodes and apps in FLoRa, custom mobility and inference application in INET, and Tensorflow-Lite runtime in the TF-lite-micro project.

4.5.3 INET Mobility System

The INET framework in OMNeT++ includes built-in mobility logic, with pre-existing mobility modules such as circular movement and back-and-forth linear "tractor" movement. However, these modules define their movement patterns during the simulation's initialization phase, making it impossible to modify mobility dynamically during runtime. While OMNeT++ supports XML formats, which could theoretically allow for a custom inference component to dynamically adjust mobility using an XML-based model, this possibility has not been explored in this thesis. Instead a custom module is developed to support the our modified inference library.

¹<https://github.com/tensorflow/tflite-micro-arduino-examples>

²<https://ai.google.dev/edge/litert/microcontrollers/overview>

4.5.4 Custom Mobility Module

The custom mobility model, `AdvancedRLMobilityModule`, that has been developed, inherits from `INET_MOBILITY_BASE`. The mobility model is configured as a submodule of the gateway. During the initialization stage of the simulation, the position, direction and speed properties are configured based on parameters in the `omnetpp.ini` file.

For our custom mobility model, we define a submodule called `AdvancedLearningModel`, which is responsible for executing inference based on the input variables (eq. (4.2)).

The mobility model invokes the `AdvancedLearningModel` periodically, as configured by `mobility.modelUpdateInterval`. Based on the output of the invocation, the mobile gateway updates its direction. The mobility module moves in this direction until next invocation.

The information required for the input variables of the `AdvancedLearningModel` inference is gathered as follows, for each node j :

- During initialization, the position of node j , position_j , is fetched.
- The expected transmission time, $T_{\text{expected},j}$, is initialized to the time of node j 's first packet, $T_{\text{first},j}$.
- The average duty cycle, $t_{\text{duty cycle},j}$, is stored as a constant.

During runtime, the position of the gateway, $\text{position}_{\text{gateway}}$, is fetched from the mobility module and used to calculate the distance and direction to each node j . Additionally, the expected transmission time $T_{\text{expected},j}$ is dynamically updated based on packet reception or elapsed time. Specifically, when either:

1. A packet P_j is received from node j , or
2. The current time T_{current} is equal to the expected time $T_{\text{expected},j}$,

then the expected transmission time $T_{\text{expected},j}$ is incremented by the transmission interval $t_{\text{transmission interval},j}$. However, due to the mobility module only periodically polling and updating its state (and the practical difficulties of comparing floating-point values), it is not feasible to update the expected time at the exact moment of $T_{\text{expected},j}$. Therefore, compensation is made for the time elapsed since passing $T_{\text{expected},j}$.

As such, the behavior can be mathematically expressed as the following piecewise function:

$$T_{\text{expected},j} = \begin{cases} T_{\text{current}} + t_{\text{transmission interval},j} & \text{if } P_j \text{ is received,} \\ T_{\text{expected},j} + t_{\text{transmission interval},j} - (T_{\text{current}} - T_{\text{expected},j}) & \text{if } T_{\text{current}} > T_{\text{expected},j}, \\ T_{\text{expected},j} & \text{otherwise.} \end{cases}$$

An alternative to this compensation of $T_{\text{expected},j}$, is to instead make use of periodic self-messages in the module, but solving the issue arithmetically is sufficient.

4.5.5 OMNeT++ Data Collection

To enable custom data collection within the simulation, a component called `StateLogger` has been created. It functions as a singleton, with nodes and other components signaling the `StateLogger` through function calls to report new information, such as when a packet is transmitted. After the mobility module invokes `AdvancedLearningModel`, the `StateLogger` module is triggered to record relevant data for plotting and evaluation, such as the number of packets sent and received for performance metrics (section 4.7.1). This is done by appending an entry to a log data list.

Once the simulation terminates, the `StateLogger` module writes the collected data to a file. This data can then be used for plotting and evaluating different scenarios discussed in chapter 5.

4.6 Experiments

Each experiment has been performed after the model has been trained in SB3. Experiments in SB3 consist of loading in the model and executed it in the SB3 environment. The environment is before-hand modified to certain scenarios, ensuring the are consistent. This means that the investigation and experiments for our reward structure and training will not be presented, all experiments are based on the final trained model.

Experiments in OMNeT++ is performed by exporting the model, and modify settings in the main .ini file. This includes things such as position of nodes, send intervals, logging. Plots are then generated from logs for both SB3 and OMNeT++ experiments.

4.7 Evaluation method

Our custom SB3 environment is a minimal and down-scaled environment compared to OMNeT++. Likewise, it is therefore important that variables are scaled accordingly, such that the evaluation environment is comparable to the training environment. This is also important across individual experiments solely in OMNeT++. For both SB3 and OMNeT++, we will evaluate the performance of each experiment in 100 episodes. This means the use of identical seeds in OMNeT++ should be used across experiments to ensure validity in our results. It also means that the environments should be correctly scaled when comparing the results from SB3 and OMNeT++, such as positions, transmit intervals, transmission radius or other relevant variables.

4.7.1 Evaluation Metrics

This section presents two key metrics, the Packet Delivery Ratio (PDR) and Jain's fairness index, which will be used to validate the results in both OMNeT++ and SB3. These metrics serve as benchmarks to assess the performance and learning progress of the gateway. By analyzing these metrics, we can evaluate how well the agent performs in the environment and compare the performance across different scenarios.

Tracking these metrics in both OMNeT++ and SB3 allows us to validate the consistency and effectiveness of the trained policy, ensuring that the gateway efficiently serves all nodes in the environment.

Packet Delivery Ratio (PDR)

Packet Delivery Ratio (PDR) is a commonly used metric for evaluating the performance and reliability of wireless communication networks. It represents the ratio of successfully delivered packets to the total number of packets transmitted in the network 4.4. A high PDR indicates that the network is efficiently delivering most of the transmitted packets, demonstrating strong reliability and performance. Conversely, a low PDR suggests potential issues such as interference, congestion, or other factors causing packet loss. For this reason, PDR is selected as a key metric for evaluating our experiments and drawing valid conclusions.

$$PDR = \frac{Recieved\ Packets}{Generated\ Packets} \cdot 100 \quad (4.4)$$

Jain's Fairness Index

Jain's Fairness Index evaluates how evenly resources are distributed among users—in this case, how evenly packets are received by each node. The fairness index is calculated

using 4.5, where x_i represents the value associated with the i -th node (or user), and n is the total number of nodes.

The index ranges from 0 to 1, where 1 indicates perfect fairness. A high fairness index suggests that packets are distributed evenly across all nodes, whereas a low fairness index implies that some nodes receive significantly fewer packets, potentially due to factors such as poor connectivity or network interference.

$$f(x_1, \dots, x_n) = \frac{(\sum_{i=1}^n x_i)^2}{n \sum_{i=1}^n x_i^2} \quad (4.5)$$

As can be derived from eq. (4.5), the lowest value is actually $1/n$, when resources are allocated to a single node out of n nodes. Thus, a fairness of 0 represents a special case where there is no throughput at all.

PDR and Jain's Fairness Index measure different aspects of network performance, and a high value in one does not always correspond to a high value in the other. For instance, a high PDR can be achieved even if one node experiences significant packet loss, which would be reflected in a low Jain's Fairness Index. Conversely, strong fairness does not necessarily indicate a high PDR; all nodes could experience the same level of packet loss, resulting in a fair distribution of packets but an overall poor network performance since it is "equally bad" for all nodes.

For these reasons, we chose to use both metrics, as together they provide a comprehensive indication of the gateway's performance.

4.7.2 Evaluation consistency

Ensuring consistency in evaluations is critical for the validity and reliability of our thesis experiment. By maintaining a stable environment and controlling randomness, we ensure that variations in outcomes are due to the actual factors being studied rather than external noise or unpredictable influences. This is particularly important when comparing different scenarios, models, or approaches, as it allows for a fair and unbiased assessment.

To achieve this, we use seeding, which ensures that any randomness introduced during the experiment, such as randomized initial conditions, transmission deviations, or packet loss. Seeding allows us to reproduce results, making our findings more credible and scientifically rigorous. Without it, variations in experimental runs could lead to misleading conclusions, reducing the reliability of our comparisons.

Additionally, we conduct 100 simulations for data collection to obtain statistically significant and stable performance metrics. Running multiple episodes helps mitigate the effects of outliers or anomalies that may arise in a single trial. By averaging results over 100 episodes, we achieve a more accurate representation of the underlying trends and overall performance. However, some of the presented data will be from a single episode, as it provides a more detailed example of specific behaviors and decision-making processes within the scenario.

Overall, by enforcing consistency through controlled randomness and a sufficiently large number of episodes, we enhance the credibility, reproducibility, and robustness of our findings, ensuring that our conclusions are based on reliable and meaningful data.

Seeds for Stable-Baselines3

Seeding in SB3 is done by setting two seeds before any of the 100 episodes are executed. The first seed is for Python libraries and randomness, ensuring that any random numbers generated (e.g., random sampling, shuffling) produce consistent results across different runs.

The second seed must be set for the model itself. In SB3, the `model.set_random_seed` function is used to set the random seed for the model's internal randomness. This ensures that the behavior of the model, including training results, exploration, and environment interaction, remains reproducible across different runs.

Seeds for OMNeT++

In OMNeT++, the `seed-set` configuration variable is used to determine which set of seeds will be selected for each simulation run, allowing for different runs to use unique random number sequences. This prevents overlap between runs while ensuring that each one gets distinct seeds.

The `seed-set` affects which set of seeds is used for a particular simulation. The simulation kernel can automatically generate seeds, and each simulation run may use a different seed set. The first simulation may start with seed set 0, then seed set 1 for the second, and so on. OMNeT++ ensures that all automatically generated seeds are spaced far apart, preventing overlap of random sequences between different simulation runs.

Setting `seed-set = ${repetition}` ensures that all runs with the same repetition number use the same seed set. This is useful when we want each repetition of an experiment to follow the exact same sequence of random numbers, allowing for direct comparisons between different scenarios.

In our case, for a simulation run series, seeds will range from 0 to 99, ensuring that all 100 simulations use a different seed. To achieve this, we set:

```
repeat = 100
```

This causes every combination of iteration variables to be repeated 100 times, and the predefined OMNeT++ variable `$repetition` holds the loop counter which is equivalent to `${repetition = 0..99}`.

4.7.3 Data Collection & Presentation

Both metrics can be meaningfully evaluated at any point in an episode, calculated based on the number of packets transmitted and received. The source data includes the number of packets transmitted per node, the number of packets received by the gateway for each node, and auxiliary data such as gateway positions. The sampling frequency differs between SB3 and OMNeT++: in SB3, data is collected at every time step, corresponding to every simulated second, while in OMNeT++, data is recorded whenever the model is polled, which occurs every 10th simulated second. Since an evaluation episode lasts for a full day, the resulting dataset consists of 86,400 data points per episode in SB3 and 8,640 in OMNeT++. Given the large volume of data, visualization techniques are essential for extracting meaningful insights. To quantify model performance, three types of plots are employed.

Training Data Plots These plots evaluate different training methods, with the horizontal axis representing the number of simulation steps. The key metrics include:

- **PDR:** Sampled at the end of an episode.
- **Fairness:** Sampled at the end of an episode.

Per Episode Plots These plots capture data for a single episode to analyze specific behavior:

- **Distance to Nodes:** Tracks the distance between the mobile gateway and each node. Transmission times are plotted for reference, helping to identify cyclic movement patterns and their relation to transmissions.
- **Heatmap (SB3 Exclusive):** This plot visualizes the time spent at each position by transforming mobile gateway position data into a more intelligible format:
 - Count occurrence of gateway at each position.
 - Add a small positive value (less than 1) to ensure all values remain positive.
 - Apply a logarithmic transformation to compress the data range and thus emphasize rarely visited positions.
 - Normalize the range to $[0, 1]$ for better visualization (optional).

Episode Batch Plots These plots summarize performance across multiple episodes to mitigate the effects of stochastic variations. Each episode batch includes 100 simulation runs, from which summary statistics for performance metrics are visualized.

- **PDR Per Node (Bar Plot):** Presents the mean PDR per node with standard deviation error bars.
 - In some cases, the sum of the mean PDR and standard deviation exceeds 100%, even though packet delivery rate cannot logically surpass this limit. This typically occurs due to large variations in packet reception across episodes.
- **PDR Distribution (Box Plot):** A standard box plot illustrating the PDR distribution across episodes using median, quartiles, and outliers.
- **Fairness Distribution (Box Plot):** Similar to the PDR box plot but for fairness.

5 Evaluation

This chapter evaluates AI-driven mobility for LoRa gateways serving multiple nodes using Reinforcement Learning. It explores various scenarios, each addressing different aspects of the project, with sub-cases included to examine the general scenario in different conditions.

The first scenario examines training performance under different reward structures and Reinforcement Learning algorithms (PPO and DQN). Since the choice of algorithm and reward design significantly impacts learning efficiency and final policy behavior, this scenario aims to determine the most effective combination for optimizing network performance through gateway movement.

The second scenario evaluates the transfer of an SB3-trained model into OMNeT++. Ensuring correct model export is crucial for real-world deployment, as discrepancies between training and execution environments can degrade performance. This evaluation also measures how well the model retains its learned behaviors and adapts within OMNeT++, using our performance metrics.

The third scenario investigates the model's behavior when a faulty node is introduced into the environment. In real-world applications, sensor nodes may fail or transmit incorrect data, potentially misleading the Reinforcement Learning model. This scenario provides insight into how the model processes observations and how these inputs influence decision-making. By analyzing its response to faulty nodes, we can determine how observations contribute to either optimal or suboptimal behaviors under different assumptions and data conditions. This analysis is essential for improving the model's reliability in dynamic and unpredictable environments while also exploring ways to leverage these insights for more robust decision-making.

The fourth scenario compares an RL-driven gateway to multiple stationary gateways. While Reinforcement Learning enables adaptive mobility, stationary gateways are often used in practical deployments. This scenario evaluates whether mobility provides an advantage in terms of coverage and overall performance under different node placement configurations.

The final scenario assesses the effectiveness of an RL-driven gateway versus a static mobility model. Static mobility models follow predefined movement patterns that may not always be optimal. By comparing the two approaches in environments where static mobility is both well-suited and poorly suited, this scenario highlights the potential benefits and limitations of using reinforcement learning for gateway control.

Tables 5.1 and 5.2 outline the default parameters used for simulations in both OMNeT++ and SB3. In both frameworks, the simulations run for an equivalent duration of 86,400 seconds (1 day), with SB3 measuring time in steps. To ensure consistency, SB3 is scaled in distance, speed, and transmission range to match OMNeT++, with both area and gateway speed set to one-tenth of their OMNeT++ values. The node transmission radius cutoff is set to 40 meters in SB3, and reception probability as seen in Figure 4.3. 40 meters will correspond to 400 meters in OMNeT++, which is a relatively short range. This value was selected for training as it provided a clearer reward signal and is maintained for evaluations.

Transmission intervals are identical in magnitude in both SB3 and OMNeT++, ranging from 1000 ms to 2000 ms. The chosen interval duration ensures that a sufficient number of transmission events occur during each episode, allowing the model to learn effectively.

In this project, transmissions are based on uplink join-requests. According to the LoRaWAN v1.1 specifications [32], when accounting for a Coding Rate of $\frac{4}{8}$, the payload size of each message is 36 bytes. Omitting preambles and other overhead data, we assume that the ToA is equivalent to the time required to transmit all the data symbols. This results in a Time on Air (ToA), based on eq. (2.5) and eq. (2.3), given by:

$$ToA = n_{symbols} \cdot T_s = \frac{8 \cdot 36}{12} \cdot \frac{2^{12}}{125 \text{ kHz}} = 786.432 \text{ ms} \quad (5.1)$$

Considering the duty cycle restriction of 0.1% and using eq. (2.2), the minimum required transmission interval is:

$$T_{\text{minimum}} = \left(\frac{1}{0.1\%} - 1 \right) \cdot ToA = 999 \cdot 786 \text{ ms} \approx 785 \text{ s} \quad (5.2)$$

Hence the transmission intervals satisfies the transmission constraints set by LoRaWAN.

Parameter	Value
Area	3000m x 3000m
Simulation time	86400s (1 day)
Model inference interval	10s
Number of RL gateways	1
RL gateway speed	11 m/s
Number of nodes	4
Node transmission power	11 dBm
Node Spreading Factor	SF12
Node bandwidth	125 kHz
Node coding rate	4/8

Table 5.1: Default simulation parameters for OMNeT++

Parameter	Value
Area	300m x 300m
Simulation time	86400s (1 day)
Model inference interval	10s
Number of RL gateways	1
RL gateway speed	1.1 ms
Number of nodes	4
Node transmission radius	40m

Table 5.2: Default simulation parameters for SB3

5.1 Scenario 1

The first scenario was designed to evaluate the performance of our environment and the training setup in SB3. The experiment focuses primarily on the configuration of Reinforcement Learning, assessing the efficiency of the training process under various reward structures and algorithms, specifically PPO and DQN. It compares different reward shaping approaches, highlighting the impact of positional rewards and fairness rewards on training outcomes. The goal is to identify the optimal reward structure and algorithm to be used to train the final model implemented in OMNeT++. To this there are 3 cases which will be compared to our bare reward structure under PPO, which is the one used for the final model.

- **No reward shaping:** Where the only rewards or penalties given are when receiving or missing packets
- **Residuals:** A test case where Residual Connections are used for each layer in the network's feature extractor.
- **DQN:** Same final reward structure, but uses DQN algorithm instead of PPO for training.

The learning process for all cases is capped at 10 million steps. Each episode spans a quarter of a day (21600 seconds) with a time skip of 10, meaning the agent takes 2160 actions per episode. Evaluations occur every second episode.

Learning parameter	Value
Total timesteps	10 million
Episode length	21600 steps
Time skip	10
Evaluation frequency	43200 steps

Table 5.3: S1 Learning parameters

Figures 5.1 and 5.2 illustrate the fairness and PDR progression throughout training. These figures highlight how the model improves over time. By 3 million steps, the agent has developed a good understanding of the environment and how to serve nodes effectively, achieving an average PDR of 80% and a fairness metric approaching 1.

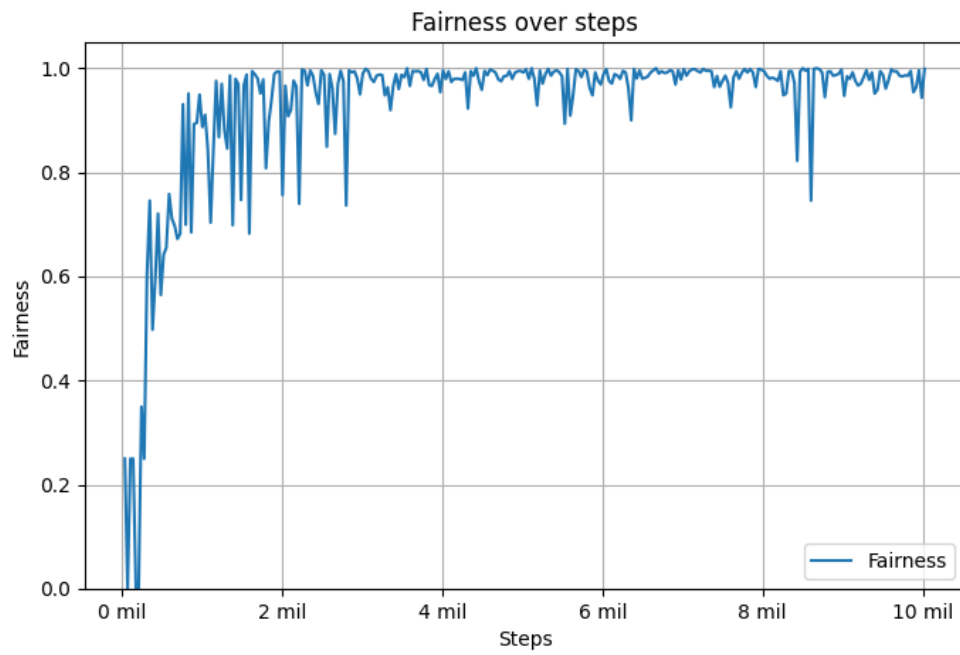


Figure 5.1: S1 Fairness during training for final model

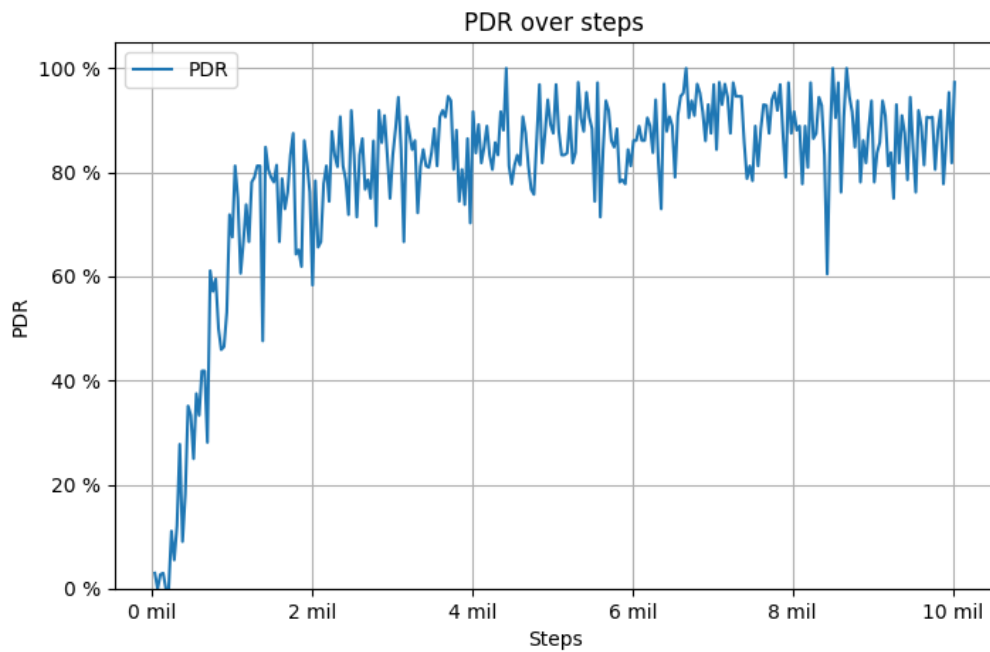


Figure 5.2: S1 PDR during training for final model

After training, see Figure 5.1 and 5.2, the model is then evaluated over 100 simulations, where the episode length is equivalent to one day. Figure 5.4 and 5.3, shows the averaged performance over all 100 simulations of the final model in SB3. Figure 5.3 shows a boxplot for fairness and PDR for all nodes over 100 simulations. It can be seen that the agent performs very well, achieving a great PDR and almost perfect fairness. Figure 5.4 shows the PDR and Fairness for each individual node over the 100 simulations. This shows that every node get an average PDR above 89%, with an overall average PDR of 89.54% and a 0.99 fairness for all the nodes.

Figure 5.3 also shows that there is some outliers and deviation to both PDR and fairness. This could be the result of certain node positions or transmission intervals that make it harder for the agent to serve all the nodes over long periods.

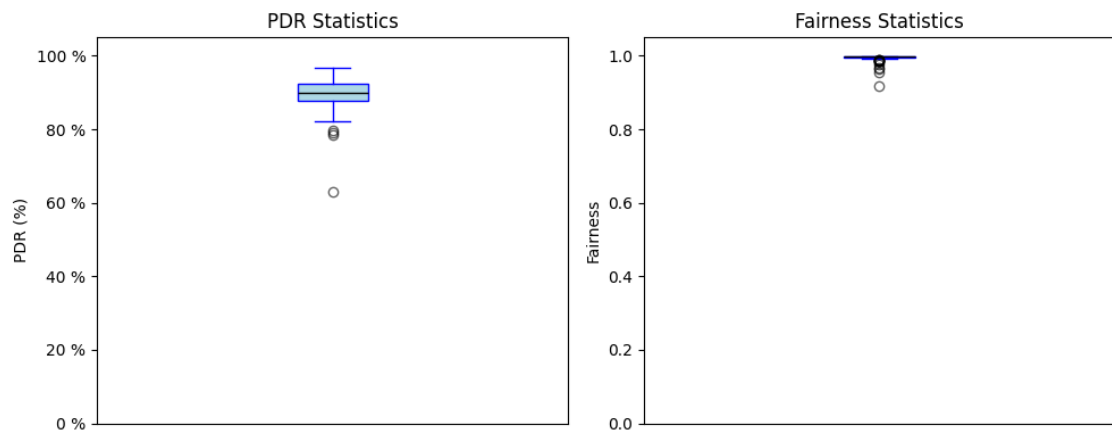


Figure 5.3: S1 Performance box plot for final model

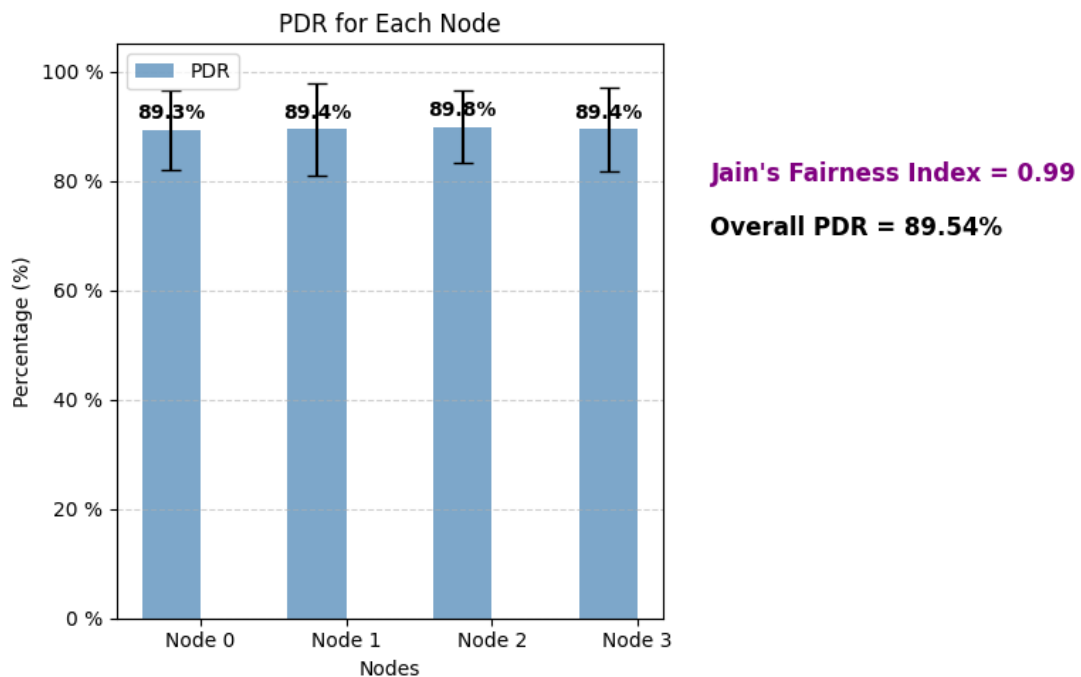


Figure 5.4: S1 Performance bar plot for final model

5.1.1 Case A: Training with No Reward Shaping

Training without reward shaping means the gateway only receives environmental signals in two cases: (1) a reward upon successfully receiving a packet and (2) a penalty for missing a packet. As a result, the gateway spends most of the time exploring blindly, receiving minimal feedback between packet events. The training process, illustrated in Figures 5.6 and 5.7, demonstrates the limited learning progress under these conditions. The gateway appears to converge to an average PDR of approximately 50% and 0.6 fairness with large deviations, likely because it falls into a local minimum where it prioritizes serving only a subset of nodes. Figure 5.5 is a heatmap of the gateways visited positions of a simulation run after training which shows this behavior of only serving a subset of nodes. The general behavior of this trained model is however rather arbitrary and cannot be concluded to anything specific, as no clear behavior of how or which nodes the gateway would serve was discovered. This highlights the crucial role of reward shaping in guiding the gateway's learning process.

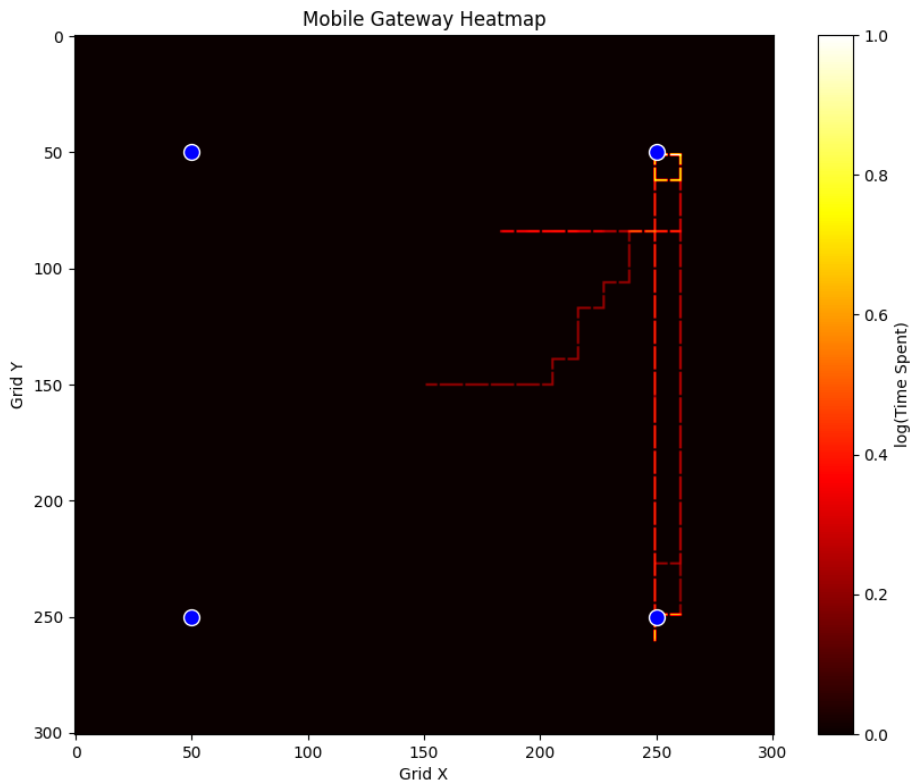


Figure 5.5: S1.A Heatmap of episode

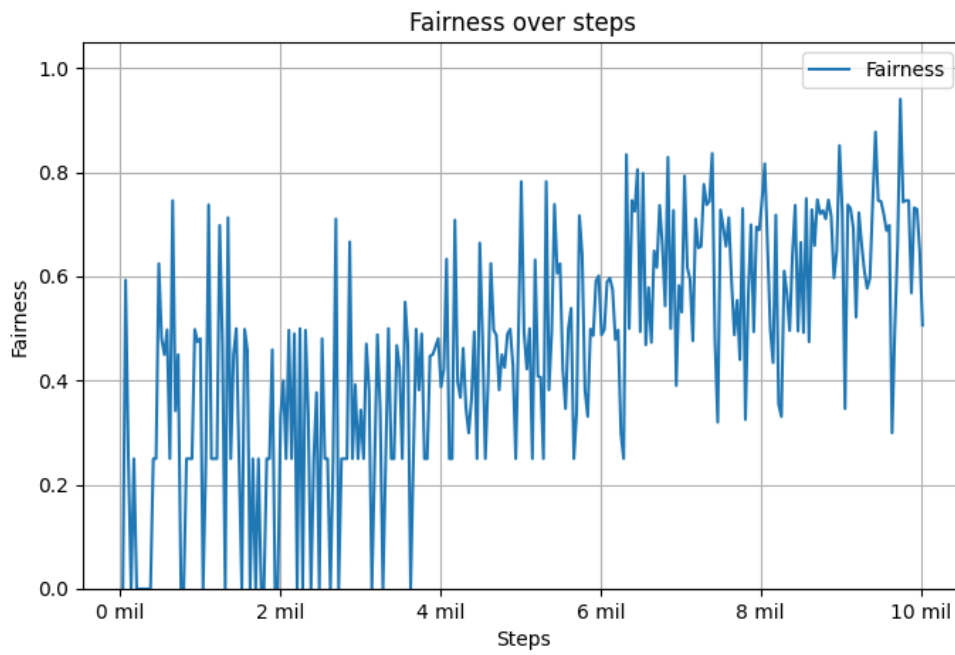


Figure 5.6: S1.A Fairness during training

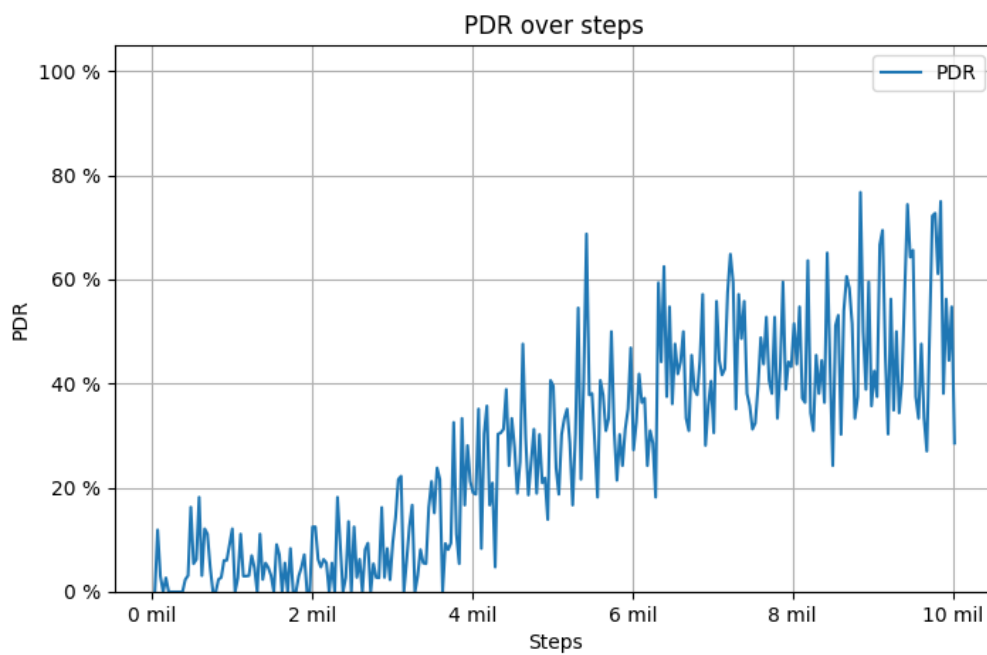


Figure 5.7: S1.A PDR during training

5.1.2 Case B: Training with Residuals

The impact of incorporating residuals, as shown in Figure 2.6, is evident in Figures 5.8 and 5.9. These figures illustrate how residuals improve the learning efficiency of the gateway. With residuals, the network reaches a stable convergence point in approximately 1 million steps, whereas training without residuals required around a little under 2 million steps to converge.

The effect of residuals should be more pronounced when observations are normalized with a larger value. For instance, if the expected packet transmission time is normalized using an entire day rather than the largest expected time in the environment, the resulting normalized values become much smaller. Consequently, during backpropagation, the gradients may also become very small, potentially leading to the vanishing gradient problem, which hinders effective learning.

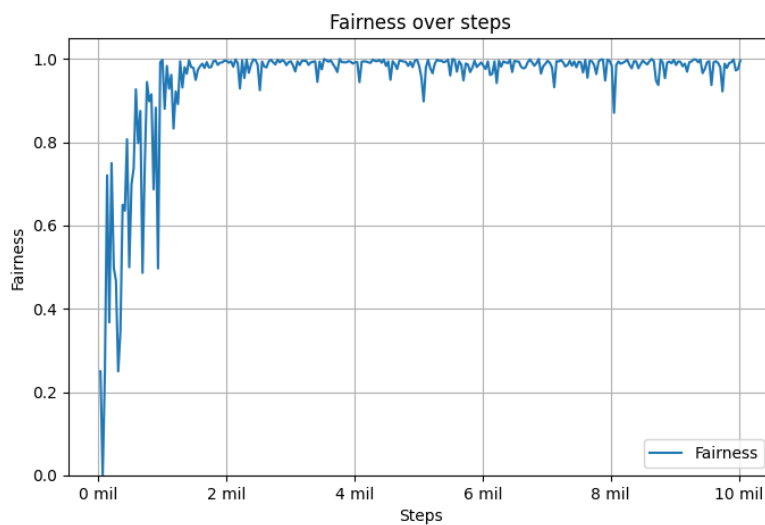


Figure 5.8: S1.B Fairness during training

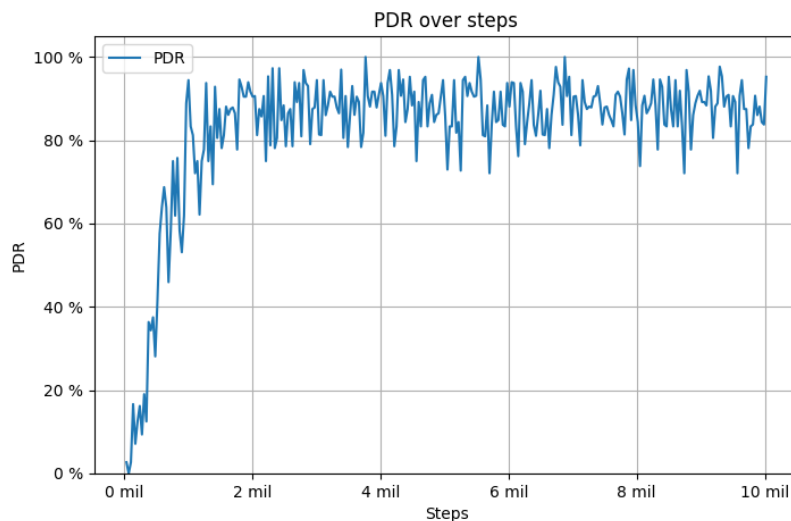


Figure 5.9: S1.B PDR during training

To validate this, a larger normalization value of 86,400 seconds (equivalent to one day) has been used for the expected packet transmission time. Figures 5.10 and 5.11 illustrate fairness during the learning process with and without residuals, while Figures 5.12 and 5.13 show the PDR. The results indicate that without residuals, the agent takes significantly longer to learn the correct behavior compared to Figures 5.1 and 5.2. Additionally, the introduction of residuals leads to a more noticeable improvement than previously observed.

This improvement is largely due to residual connections, which help prevent the vanishing gradient problem, which arises from small reward signals influenced by node positions and actions, as well as the larger normalization values used. Residual connections mitigate this by allowing gradients to flow more effectively during training. Instead of requiring each layer to learn a full transformation from input to output, residual connections enable layers to focus on learning the difference (or residual) between the input and the desired transformation, simplifying the optimization process. This not only helps prevent gradient vanishing but it also accelerates the learning process.

It should be noted the Residual Connection is a special network structure, which relies on identity mappings that bypass a layer. This structure introduces a computational graph that TensorFlow's Sequential API does not support, as it is limited to strictly linear layer stacks. As a result, despite of the benefits of residual connections, it could not be converted from PyTorch (SB3) to the TensorFlow Lite model used in OMNeT++. This also means the Final Model used for validation in SB3 did not include Residual Connections either.

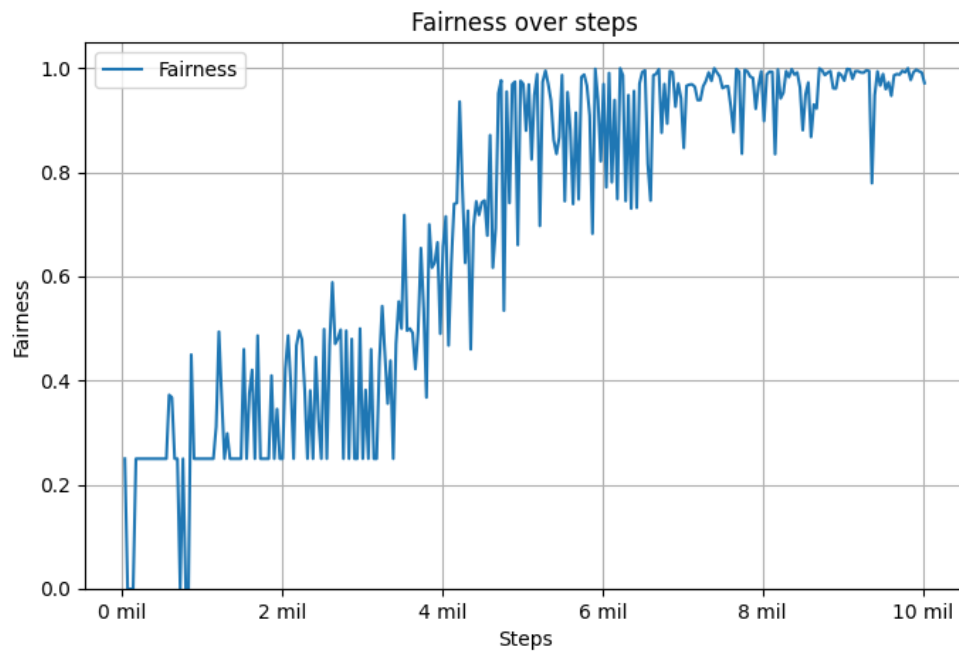


Figure 5.10: S1.B: Fairness with a 86400 normalization value for expected time

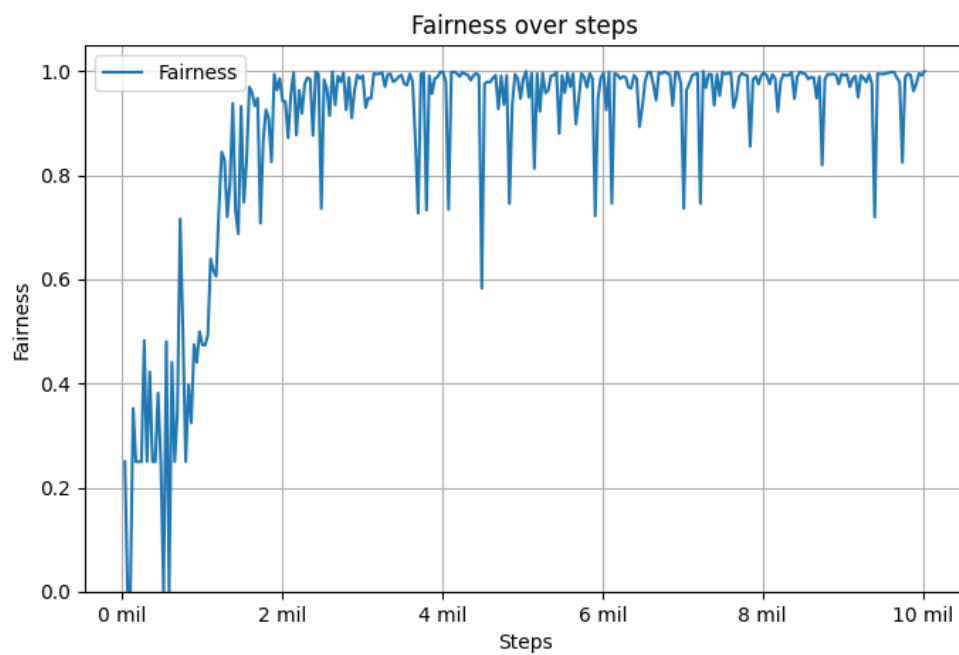


Figure 5.11: S1.B Fairness with residuals and a 86400 normalization value for expected time

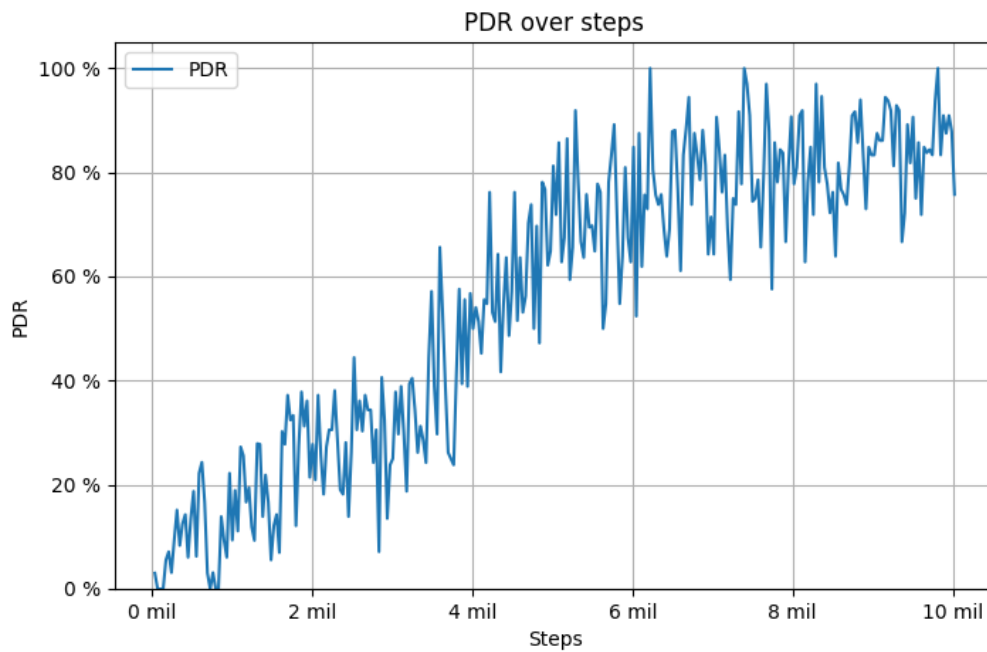


Figure 5.12: S1.B PDR with a 86400 normalization value for expected time

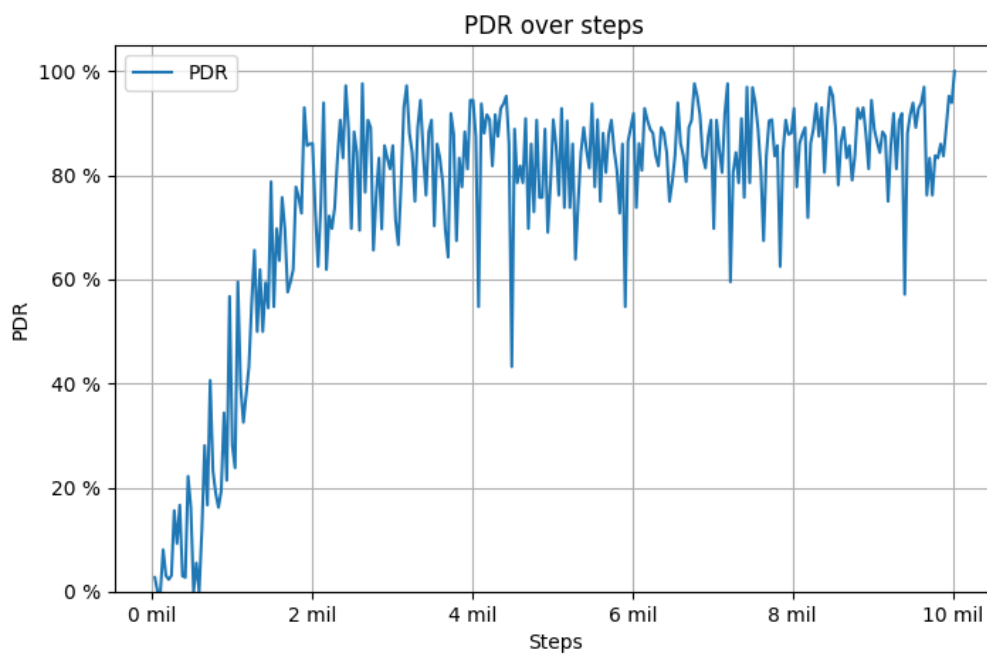
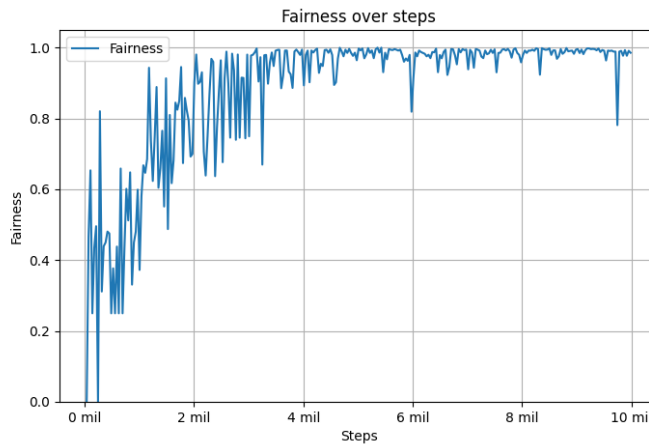


Figure 5.13: S1.B PDR with residuals and a 86400 normalization value for expected time

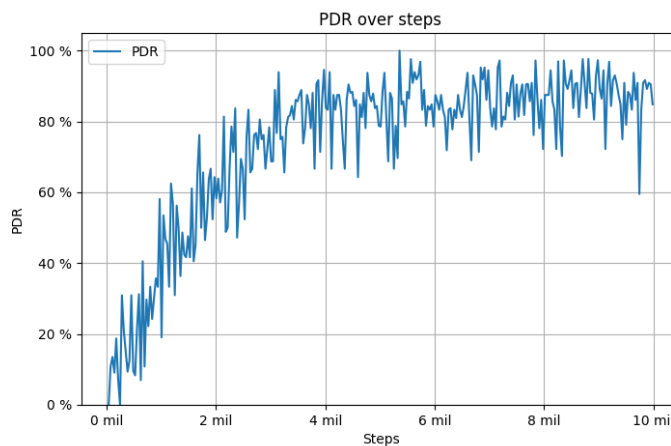
5.1.3 Case C: Training with DQN

The results of training with DQN instead of PPO can be seen in Figure 5.14a and Figure 5.14b. DQN demonstrates roughly the same convergence as our base PPO, with PDR and fairness stabilizing after approximately 4 million steps. This shows that DQN is also a good option to use for training in the current environment. DQN could therefore be a good option for training as it is typically less computationally demanding than PPO, making it more suitable for resource-constrained environments or scenarios where training efficiency is a priority.

The final model is trained using PPO instead of DQN due to DQN's inherent limitations in more complex environments. As the environment increases in complexity, the gateway must learn more intricate behaviors, leading to a larger observation space. PPO is better suited for handling this increased complexity. Additionally, if the gateway's actions were expanded to include continuous control over both direction and speed—requiring a continuous action space that cannot be effectively represented using discrete actions, then DQN would be unsuitable. Therefore, PPO was chosen not only for its ability to handle the problem's complexity but also because it supports potential future enhancements.



(a) S1.C Fairness during training using DQN



(b) S1.C PDR during training using DQN

Figure 5.14: S1.C Fairness and PDR during training using DQN

5.2 Scenario 2

The purpose of the second experiment is to validate that a model trained in SB3 can be successfully transferred to OMNeT++ and perform effectively. It will illustrate whether the simplified environment of SB3 emulates the High-Fidelity environment of OMNeT++, such that the training is meaningful. In this experiment, we compare the model's performance in SB3 and OMNeT++ to evaluate transferability. To achieve this, our custom environment has been scaled to closely resemble the OMNeT++ environment. The goal is to determine whether the trained model achieves comparable performance in both SB3 and OMNeT++ under normal conditions and assumptions.

	Position (x,y)	time of first packet	send interval
node 0	(50,50)	400	truncnorm(1600,5)
node 1	(50,250)	800	truncnorm(1600,5)
node 2	(250,250)	1200	truncnorm(1600,5)
node 3	(250,50)	1600	truncnorm(1600,5)
gateway	(150,150)	∞	∞

Table 5.4: S2 SB3 scenario parameters

	Position (x,y) (m)	time of first packet (s)	send interval (s)
node 0	(500,500)	400	truncnorm(1600,5)
node 1	(500,2500)	800	truncnorm(1600,5)
node 2	(2500,2500)	1200	truncnorm(1600,5)
node 3	(2500,500)	1600	truncnorm(1600,5)
gateway	(1500,1500)	∞	∞

Table 5.5: S2 OMNeT++ scenario parameters

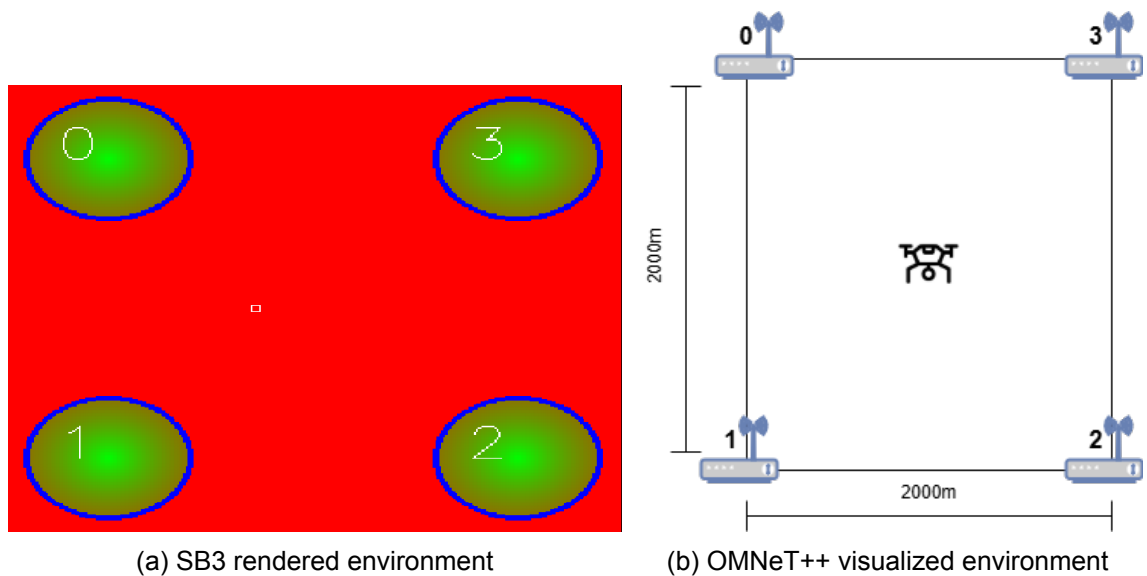


Figure 5.15: S2 Environments in SB3 and OMNeT++.

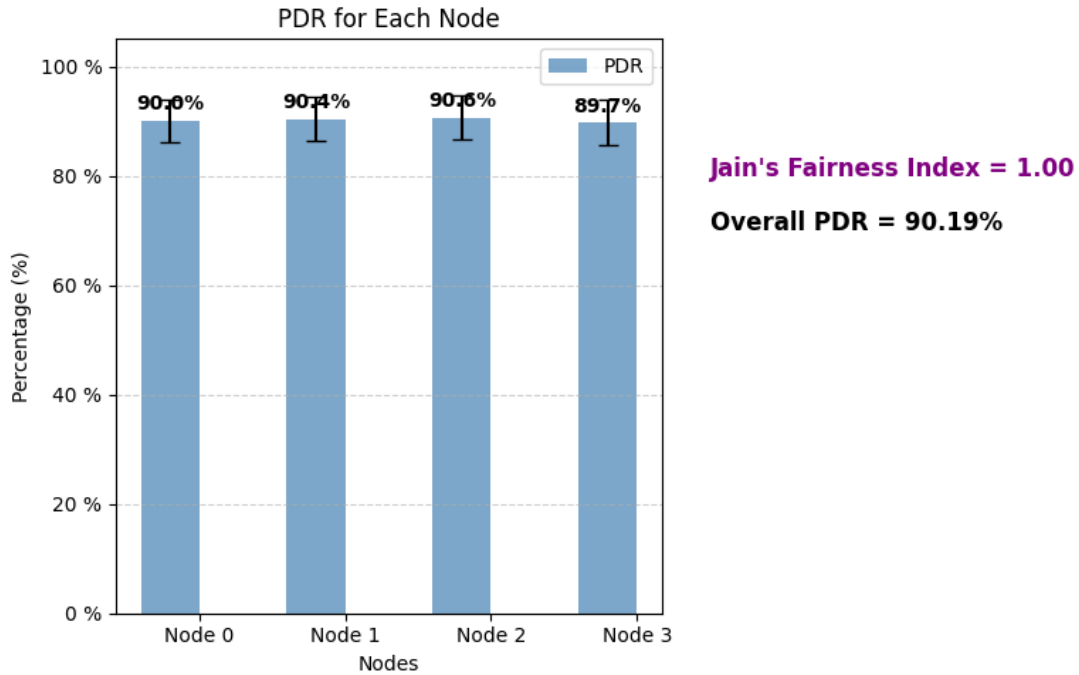


Figure 5.16: S2 Performance bar plot in SB3

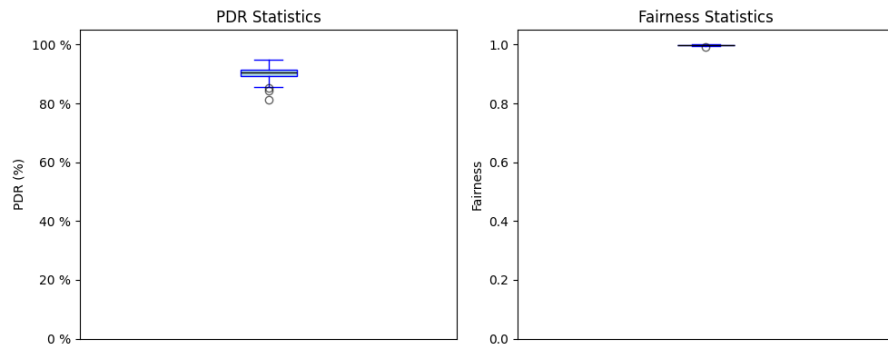


Figure 5.17: S2 Performance box plot in SB3

To validate that the model is correctly exported from SB3 to TensorFlow, a random set of input-output pairs is generated before export and then compared against the exported model's outputs to ensure consistency. This is done with a function that performs the following checks:

- Ensures that the input dimensions of both models match.
- Generates 10,000 random input samples within the expected input space.
- Computes the output probabilities from both the SB3 model and the converted TensorFlow model.
- Compares the outputs using the `numpy.allclose` function with a predefined absolute and relative tolerance.
- Reports any mismatches detected during validation.

The function can be seen in Appendix E, which gives an output for the final model:

```
Mean Absolute Difference: 7.510244870445604e-08
Mean Relative Difference: 1.3930197155787027e-06
```

These values indicate an extremely low mean difference, which is likely due to numerical precision variations during export rather than a faulty conversion.

The exported model is then integrated into OMNeT++ to verify that it remains transferable and behaves consistently with its original SB3 counterpart. The movement pattern expressed by the gateway for an episode is included in the Appendix C (fig. C.1, fig. C.2, fig. C.3, fig. C.4). The movement follows a cyclic pattern, with the gateway reducing its distance to each node before packet transmission. This demonstrates that the learned behavior is successfully transferred across neural network frameworks and simulation environments.

The performance seen in fig. 5.18a, fig. 5.18b and fig. 5.19 show that the model performs near-perfectly in OMNeT++, achieving a PDR of 99.94%. Additionally, none of the episodes resulted in a low data rate, as evidenced by the absence of meaningful outliers. In this scenario, the agent is able to receive nearly all packets transmitted by the four nodes. Since the delivery rate of each node is approximately equivalent, the Fairness metric is also near-perfect, resulting in Fairness of 1.00 due to rounding. Comparing the model's performance in OMNeT++ and SB3 reveals that OMNeT++ achieves both higher PDR and Fairness.

These results confirm the successful transfer of the model, demonstrating its ability to generalize from SB3 to OMNeT++. Furthermore, they suggest that the SB3 environment provides a conservative estimate of performance, acting as a lower-bound approximation of OMNeT++. In other words, SB3 serves as a pessimistic approximation of the real-world conditions simulated in OMNeT++.

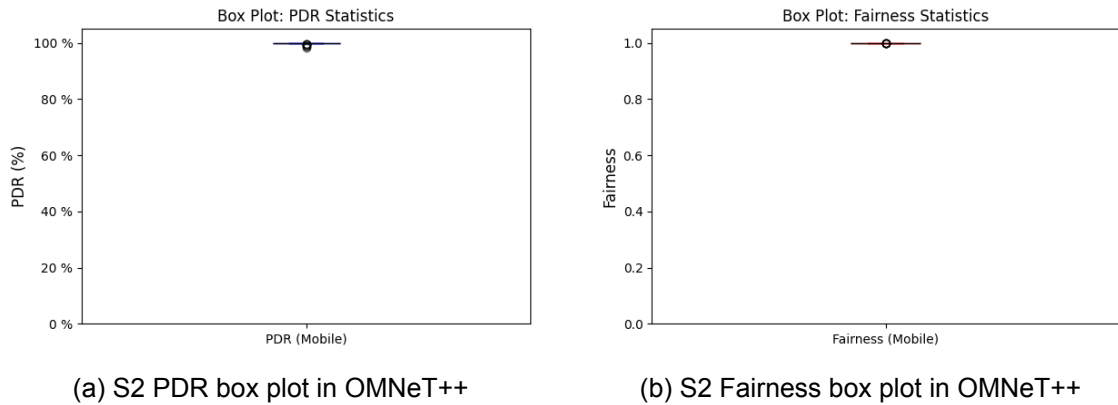


Figure 5.18: S2 PDR and Fairness box plots in OMNeT++

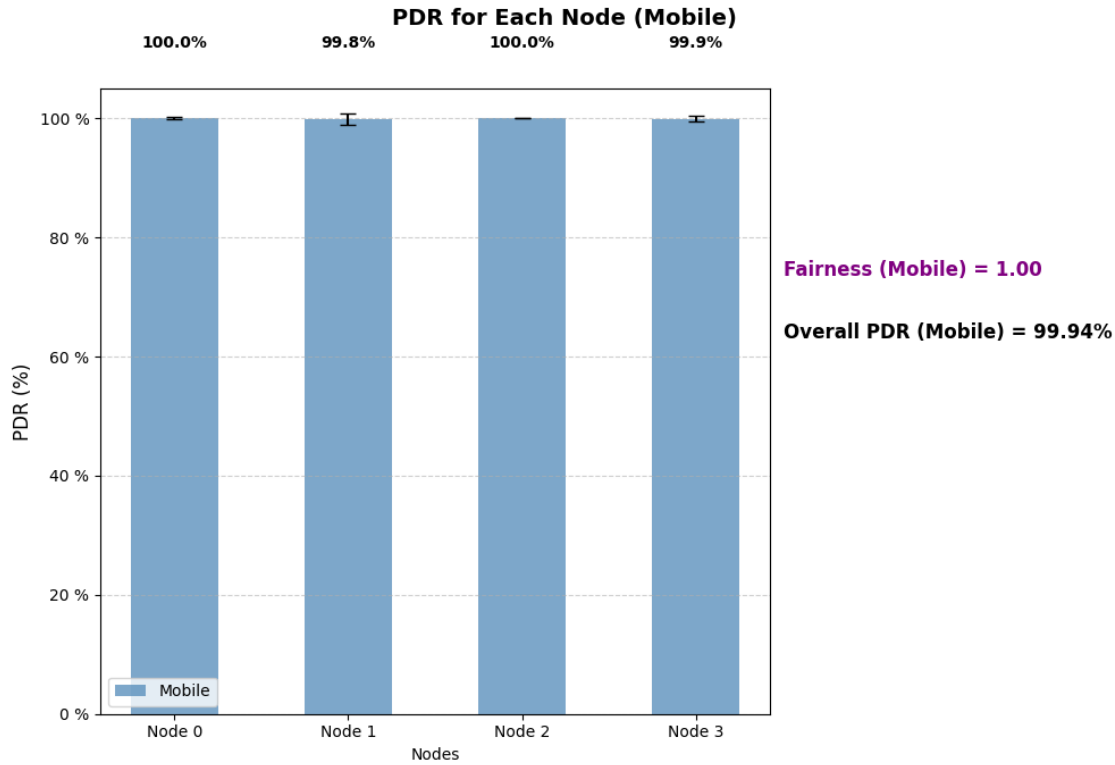


Figure 5.19: S2 Performance bar plot in OMNeT++

5.3 Scenario 3

The third scenario examines a critical aspect of our trained model in SB3: how it handles packet losses. These experiments are designed to observe the gateway's behavior when encountering a packet loss. The objective is not to assess the model's overall performance but to analyze its response to such situations. For this a node is modified to be *faulty*, such that it always produce a packet that will be missed.

In this context, three approaches to handling a faulty node are considered:

1. **Assuming Lost Packet:** When the expected time to the next packet from the affected node reaches zero, it assumes a packet loss has occurred. The expected time is then updated to reflect the arrival of the subsequent packet.
2. **Immediate Retry Assumption:** The expected time assumes the lost packet, is in fact not lost, and should be received imminently, making the expected packet time remain at zero indefinitely.
3. **Known Faulty Node:** The node is identified as faulty. In contrast to the immediate retry assumption, the expected time is set to a high value to encourage the gateway to avoid serving the faulty node.

This scenario highlights the potential challenges a gateway may face when faulty nodes are present. It also demonstrates effective methods for mitigating these issues by strategically adjusting expected packet times, ensuring the gateway continues to operate efficiently and prioritize active nodes.

For all three cases the same same parameters and setup used Scenario 2 with a modified faulty node, see Figure 5.15a and Table 5.6.

	Position (x,y)	time of first packet	transmission interval	faulty
node 0	(50,50)	400	truncnorm(1600,5)	\times
node 1	(50,250)	800	truncnorm(1600,5)	\times
node 2	(250,250)	1200	truncnorm(1600,5)	\times
node 3	(250,50)	1600	truncnorm(1600,5)	\checkmark
gateway	(150,150)	\times	\times	\times

Table 5.6: S3 Scenario parameters

To ensure a fair comparison, a base case without any faulty nodes is also tested, which corresponds to the SB3 results from Scenario 2. This serves as a benchmark for performance in an ideal environment. As shown in Figure 5.20, the agents achieve perfect fairness and a PDR of 90.19% over 100 evaluations. This highlights how well-suited the scenario is for the agent, providing optimal conditions for reaching all nodes efficiently. Establishing this ideal baseline makes it easier to assess the impact when introducing a faulty node.

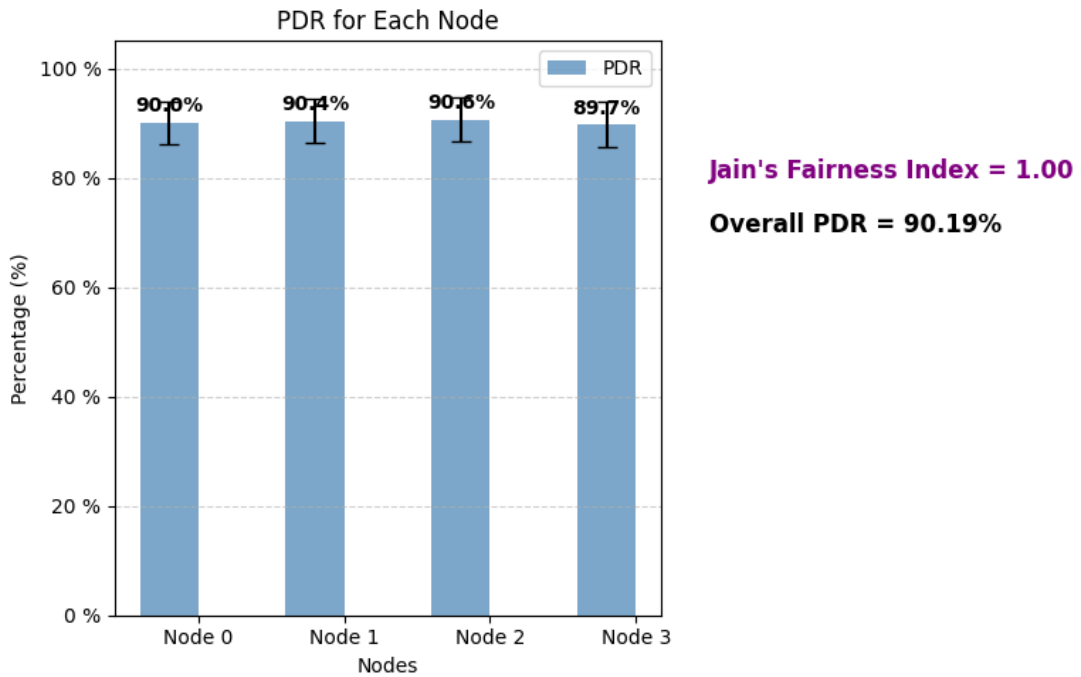


Figure 5.20: S3 Performance bar plot (no faulty node)

5.3.1 Case A: Adjusted Expected Time

The results indicate that even when a node is faulty, the agent continues to provide service to the other nodes by relying on expected time observations, as shown in Figure 5.21. Despite this, the remaining nodes maintain a relatively high PDR, though it is lower than before. Notably, node 0 exhibits the worst performance compared to Figure 5.20.

To understand this behavior, Appendix D (fig. D.1, fig. D.2, fig. D.3, fig. D.4) and Figure 5.22 shows that the gateway moves fairly between nodes, as no node is assumed to be faulty. Node 0, also has a drop in PDR of roughly $\sim 5\%$, this is likely because it is scheduled to transmit immediately after the faulty node. In this case, the gateway waits for a packet that never arrives. Although the expected time mechanism accounts for potential packet loss, the agent continues to wait as long as possible, anticipating the arrival of a packet just before the expected time elapses. Over time this will cause the gateway to not be able to reach node 0 in time, as the actual time the packet is transmitted and the expected transmission time deviates from each other slightly.

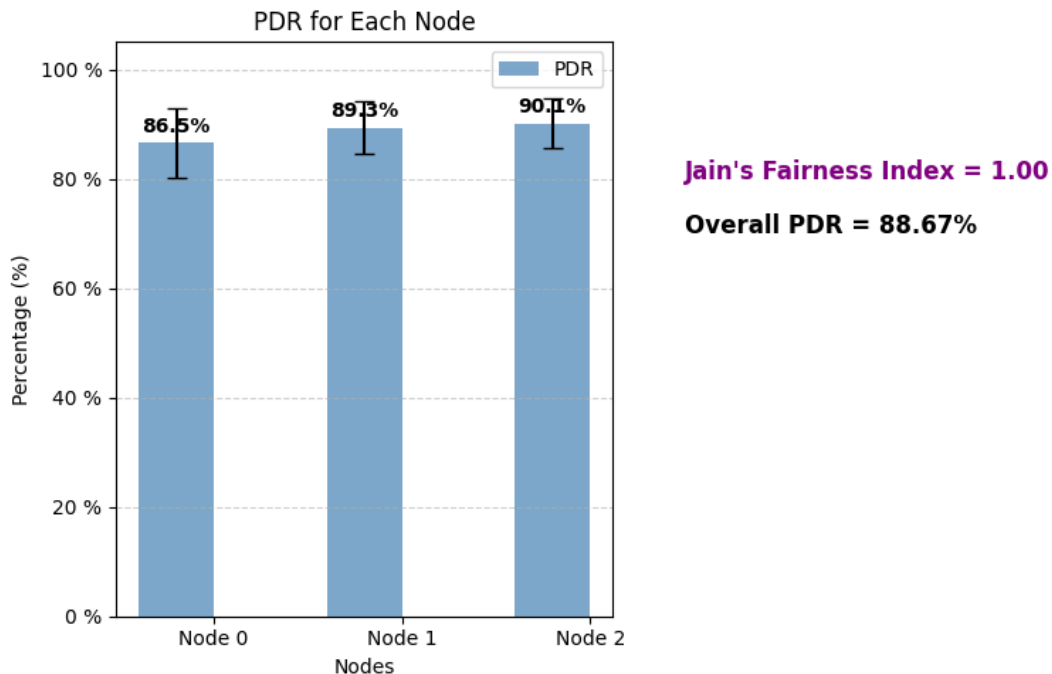


Figure 5.21: S3.A Performance bar plot

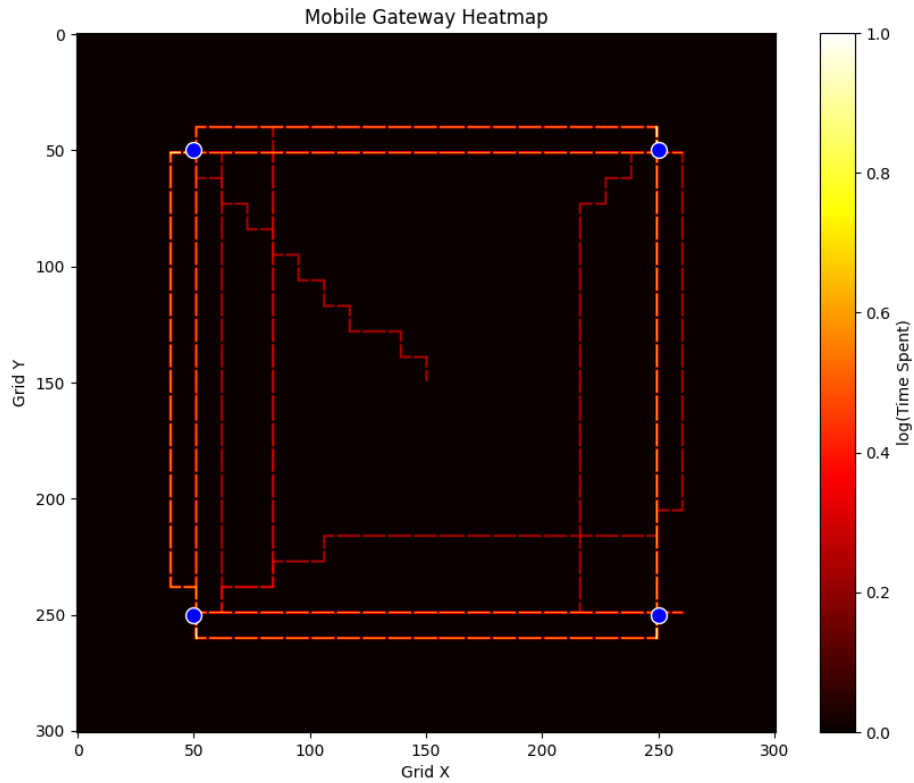


Figure 5.22: S3.A Heatmap of gateway in an episode

5.3.2 Case B: Immediate Retry Assumption

The results of this case highlight the importance of correctly assigning expected packet times to influence the behavior of the model. Figure 5.23 shows that the gateway fails to receive any packets from the nodes. This issue arises because the gateway remains close to node 3, as the faulty node continuously signals an expected packet arrival. Since the agent believes a packet is always imminent, it stays near node 3 indefinitely. This behavior is further illustrated in Figure 5.24, where the distance to node 3 remains close to zero, indicating that the gateway never moves away from the node. This is further validated from Figure 5.25, which shows a heatmap over an episode. Here it can be seen that the gateway visits all 4 nodes, but stops at node 3 as it indefinitely tries to wait for the packet it believes will be revived in any moment.

This behavior is clearly undesirable, as the gateway is unable to recognize that the node is faulty if the expected packet time does not reflect it. Therefore, it is crucial that the method used to calculate the expected packet time accounts for this issue, whether through an algorithm or another approach.

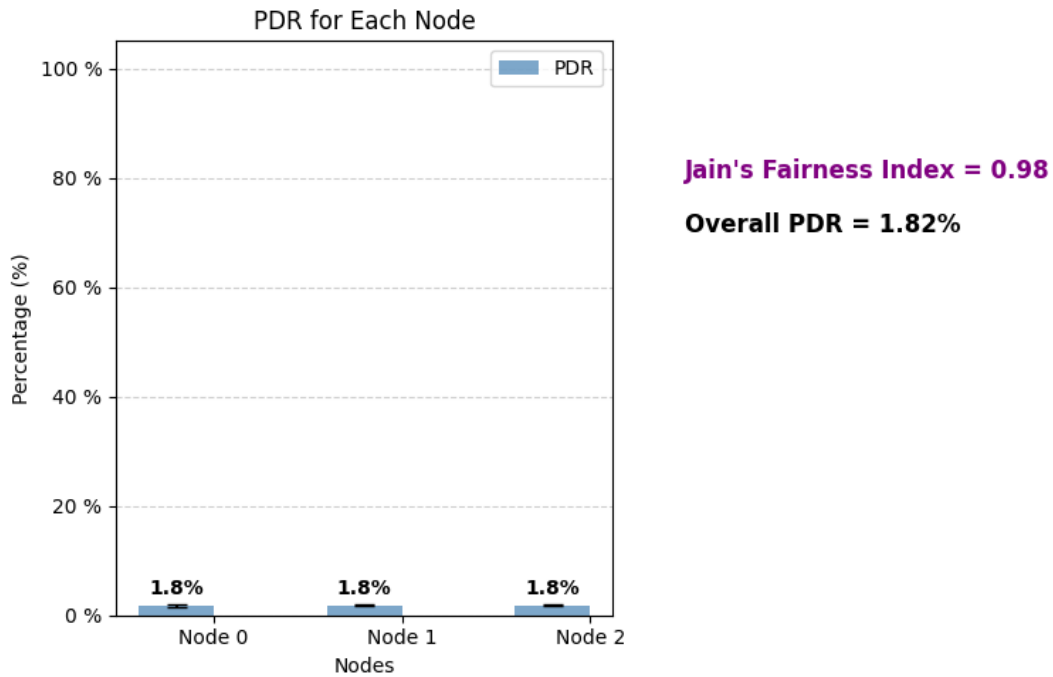


Figure 5.23: S3.B Performance bar plot

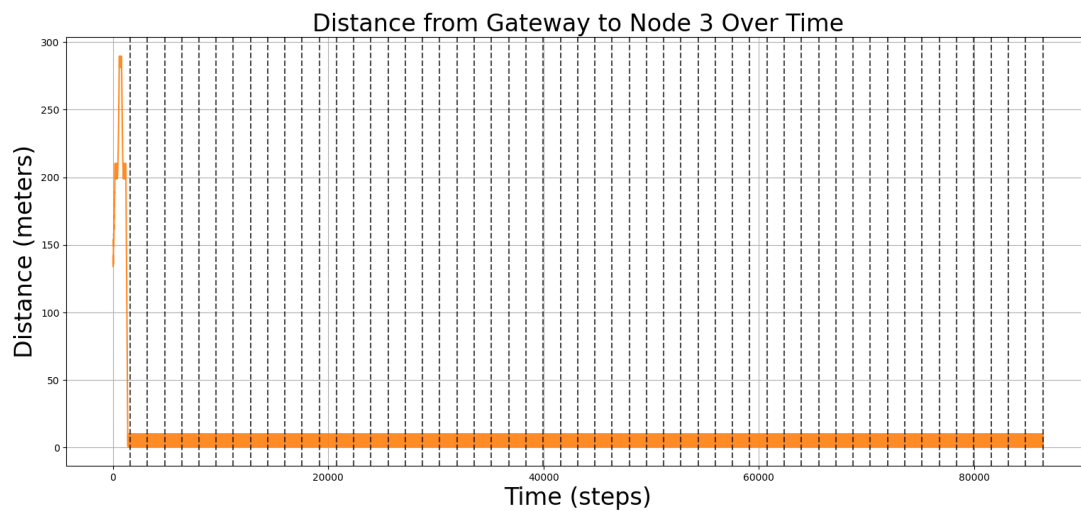


Figure 5.24: S3.B Disance for node 3 over an episode

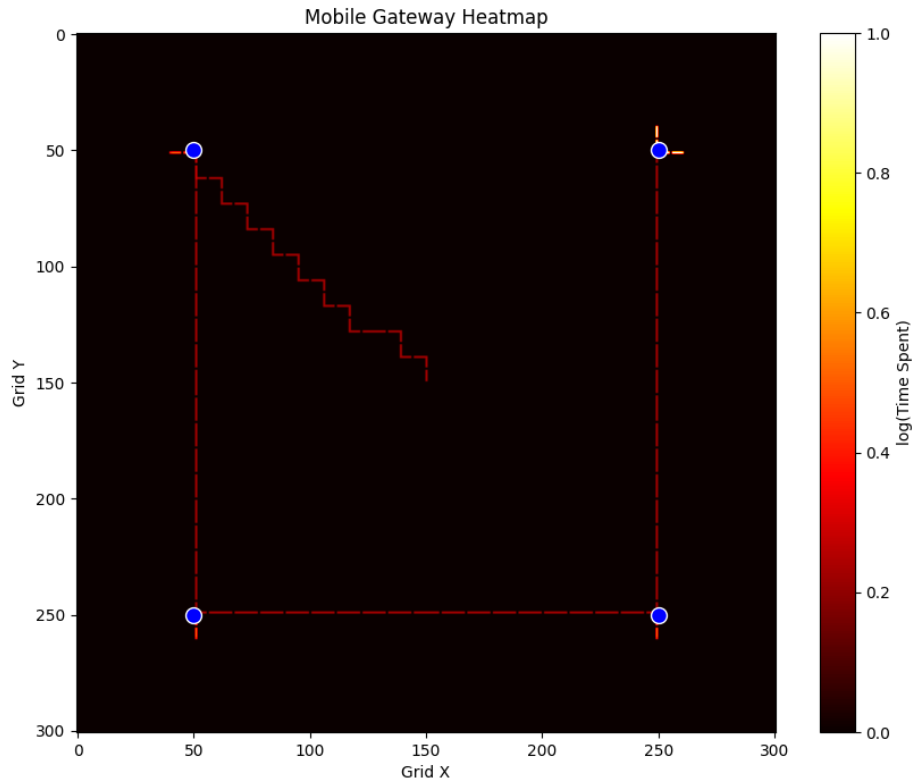


Figure 5.25: S3.B Heatmap for gateway in an episode

5.3.3 Case C: Known faulty node

This scenario explores how the gateway can be guided to avoid wasting time on a known faulty node. One effective approach is to manipulate the expected packet time, influencing the gateway's decision-making process. By assigning a distant expected packet time to the faulty node, the gateway deprioritizes it, allowing the system to function more efficiently.

The results confirm that this strategy is highly effective. As shown in Figure 5.21, Node 3 experiences a better PDR since the gateway no longer wastes time waiting for packets from the faulty node. This behavior is further validated by the heatmap in Figure 5.27, which visualizes the gateway's movement during an episode. The heatmap clearly shows that the gateway never moves towards the faulty node, instead focusing on serving the remaining active nodes. Beyond handling faulty nodes, this strategy also enables the gateway to dynamically adjust the number of nodes it serves. Nodes effectively "absent" from the environment can be assigned a permanent distant expected packet time, ensuring optimal resource allocation. In this case, a constant expected packet time equal to the transmission interval (1600) is used to achieve this effect.

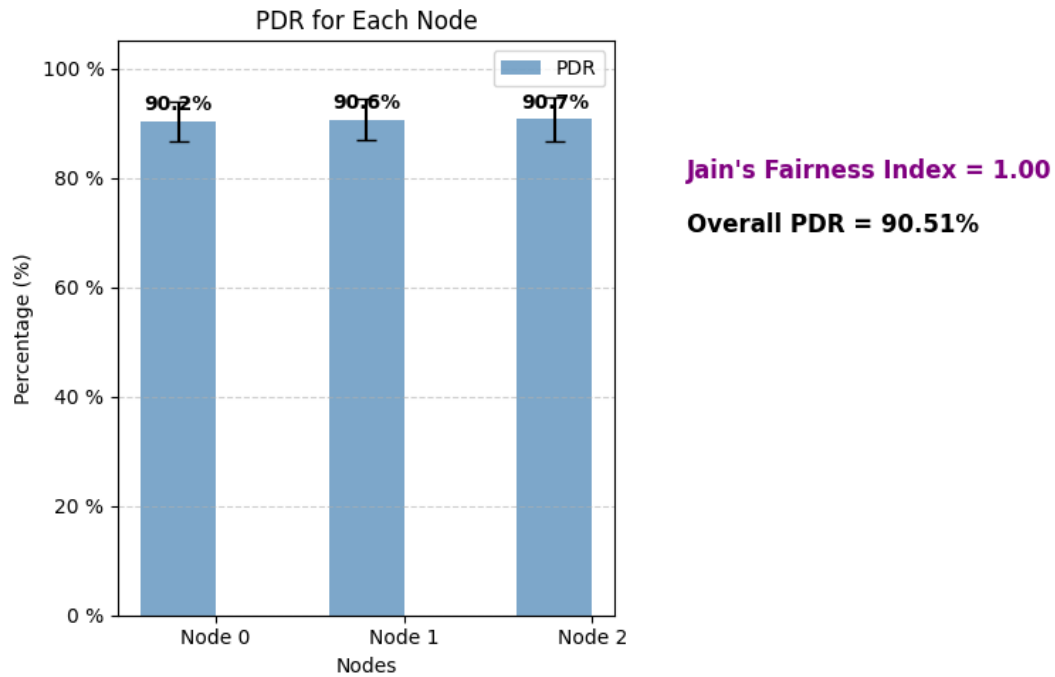


Figure 5.26: S3.C Performance bar plot

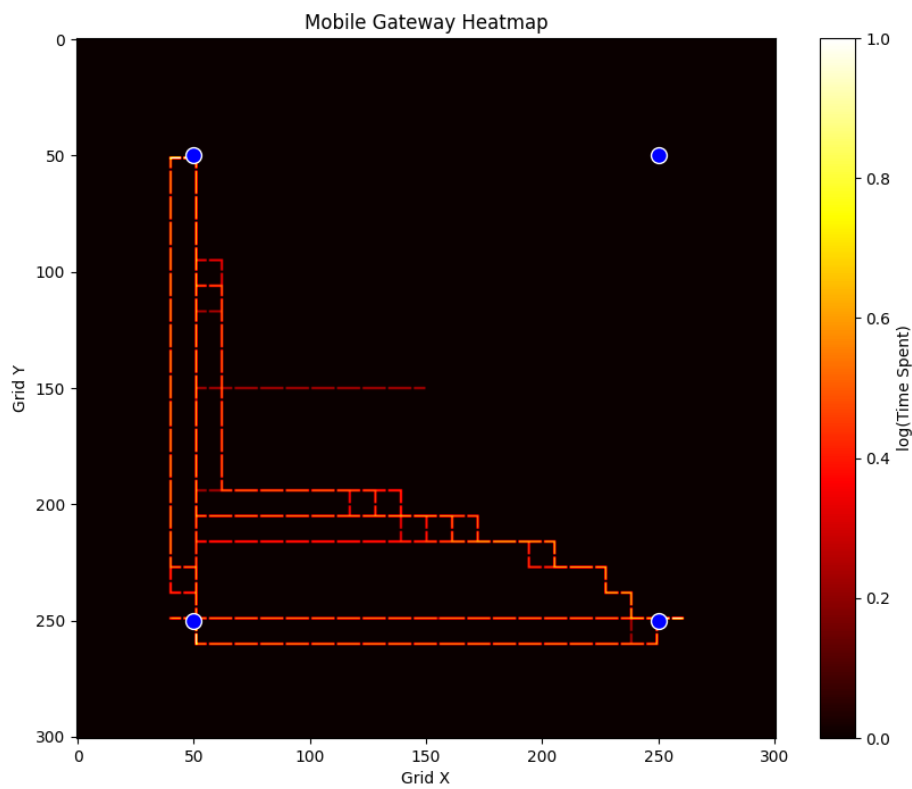


Figure 5.27: S3.C Heatmap of Gateway

5.4 Scenario 4

The fourth scenario compares the performance of a reinforcement learning-based mobile gateway to multiple static gateways in OMNeT++. The experiment evaluates various node placement configurations to assess how effectively the mobile gateway performs relative to stationary gateways, which are expected to provide consistent full coverage. By analyzing differences in performance, this scenario aims to determine the minimum number of stationary gateways a single mobile gateway can replace.

As observed in section 5.2, the mobile gateway performed optimally, successfully receiving nearly all packets. In this scenario, the goal is to compare the novel RL-trained mobile gateway with traditional stationary gateways. Since an ideal scenario with abundant resources offers little insight, this experiment introduces a more challenging environment by reducing the transmission interval. Under these conditions, the mobile gateway must consistently make optimal decisions to receive packets successfully, increasing the difficulty of the task.

From the results of this scenario, we will observe that stationary gateways, even when positioned relatively close to the target nodes (within 500 m), do not achieve perfect performance. Additionally, the mobile gateway demonstrates its ability to effectively substitute stationary gateways. Specifically, we will see that the performance of the mobile gateway is comparable to four stationary gateways. However, this advantage depends on the duration between transmissions being long enough for the mobile gateway to cover the required physical distance. These findings will highlight the mobile gateway's potential as an efficient solution in dynamic environments.

5.4.1 Case A: 4 Stationary Gateways

In this case, the goal is to compare the performance of an RL-trained mobile gateway with that of four stationary gateways. This scenario serves as a baseline for the remaining cases, representing the performance of stationary gateways under ideal conditions. Given that there are four nodes, the optimal placement for the stationary gateways is adjacent to each node to ensure efficient transmissions. Therefore, each stationary gateway is positioned 100 meters away from a node along the x-axis.

To decide on the transmission intervals for the nodes, we evaluate the distance between each node. In the case that nodes are located in a grid-layout with 2000m distance, such that sequentially transmitting nodes are adjacent, the time between each transmissions equals

$$2000m / 11 \frac{m}{s} = 181 s \quad (5.3)$$

This is the minimum amount of time required between transmission for the mobile gateway to reach each node and maximize the reception probability. Though, this requires the mobile gateway to move ideally - however, it is not able to change its direction at any point in time; only every 10 seconds by inference of the neural network. This can make it overshoot in directions, which necessitate backtracking. Additionally, the gateway only has access to an estimate of transmission times. As such, the time between transmissions for the scenario is padded, to a value of 250 s. This means, every node will have an individual send interval of $n_{nodes} \cdot \Delta T_{transmissions} \Rightarrow 4 \cdot 250 s = 1000 s$.

	Position (x,y) (m)	time of first packet (s)	send interval (s)
node 0	(500,500)	400	truncnorm(1000,5)
node 1	(500,2500)	650	truncnorm(1000,5)
node 2	(2500,2500)	900	truncnorm(1000,5)
node 3	(2500,500)	1150	truncnorm(1000,5)
mobile gateway	(1500,1500)	X	X
station. gateway 0	(400,500)	X	X
station. gateway 1	(400,2500)	X	X
station. gateway 2	(2600,2500)	X	X
station. gateway 3	(2600,500)	X	X

Table 5.7: S4.A OMNeT++ Parameters

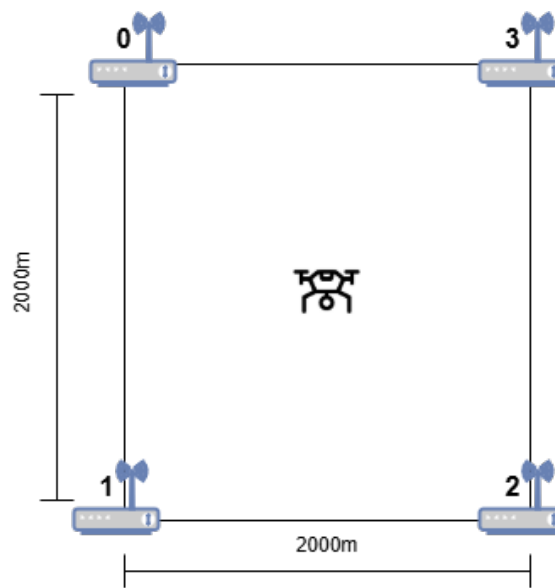


Figure 5.28: S4.A Initial positions of LoRaWAN units.

As can be seen in fig. 5.29, the PDR of the stationary gateways is perfect, receiving all transmitted packets. This is as expected, since each stationary gateway is allocated to service only a single node, which is spatially adjacent. Once more, an ideal fairness (fig. 5.30) is also generated by this configuration, as each node has 100% throughput with the stationary gateways (fig. 5.31).

On the other hand, the constraint of time between transmissions affects the performance of mobile gateway somewhat. The mobile gateway achieves an average PDR of 96.70%, and fairness of 0.99. The performance is not completely consistent, as can be seen from the box plot (fig. 5.29). The minimum and maximum PDR of the 100 episodes, excluding outliers, is approximately 90% and 100%, respectively. The average performance is still remarkable, but the distribution does show that the model is not completely robust. In section 5.2, the mobile gateway had a perfect performance of approximately 100% PDR. This case thus validates that the amount of time between transmissions are a critical aspect of the mobile gateway's performance.

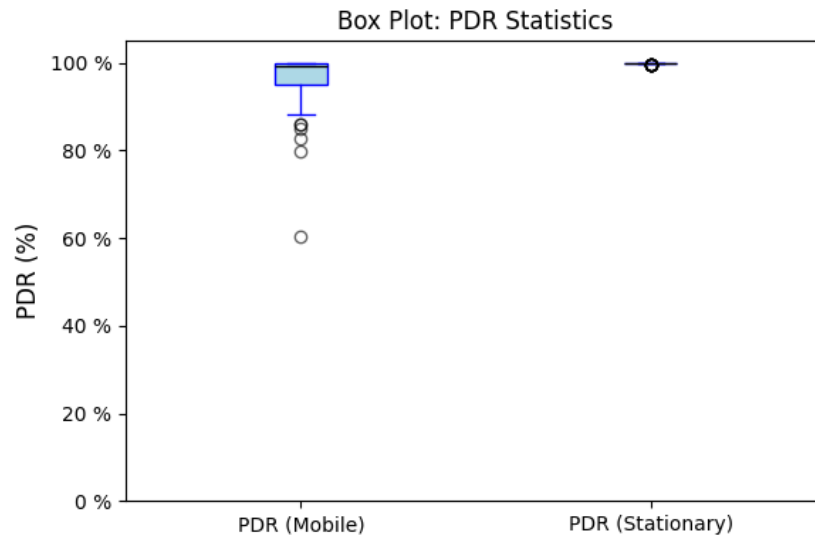


Figure 5.29: S4.A PDR box plot

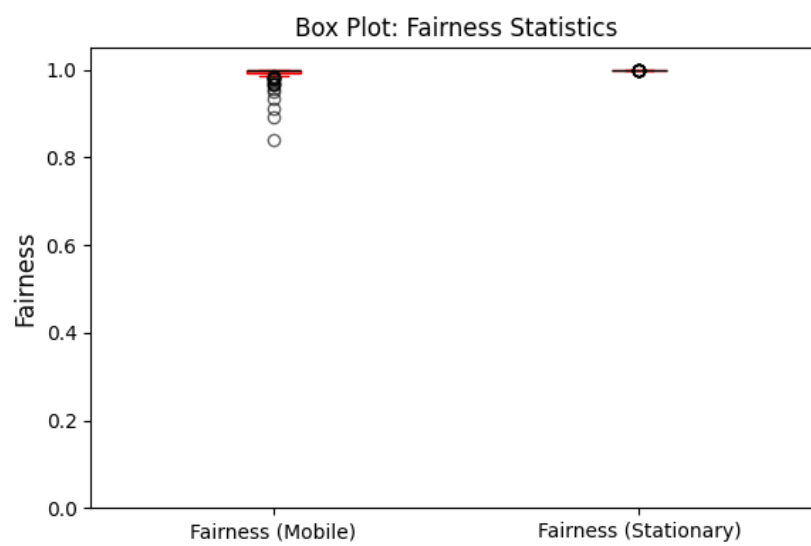


Figure 5.30: S4.A Fairness box plot

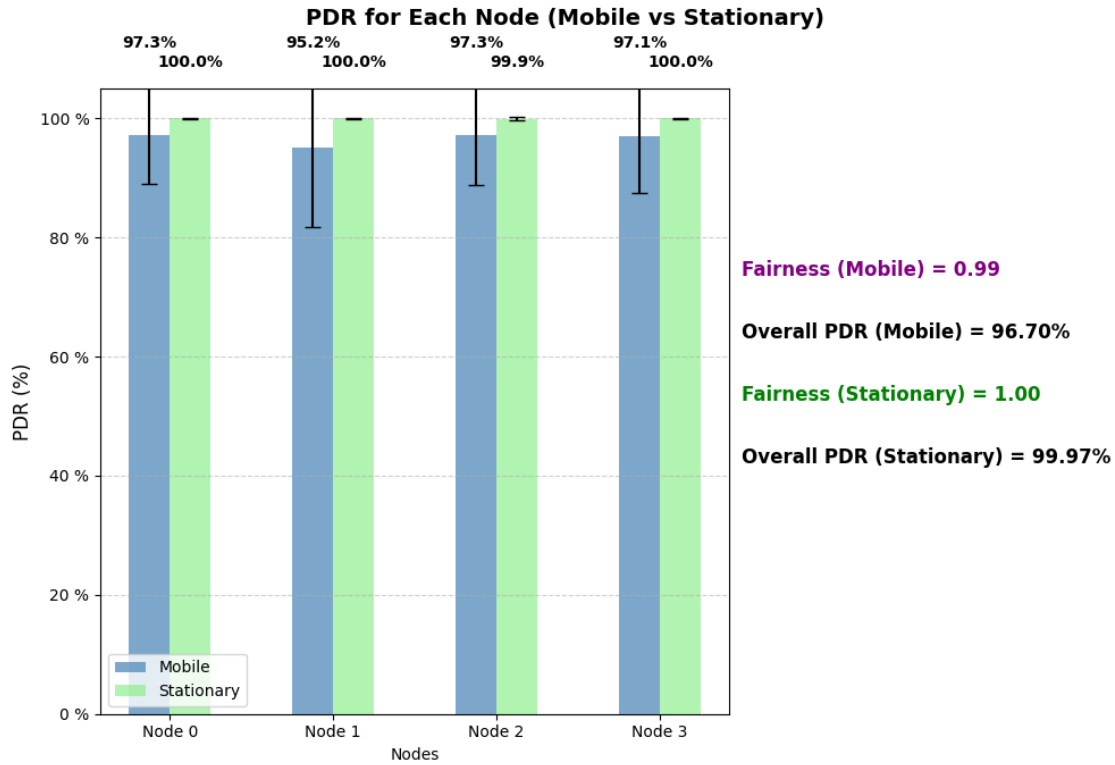


Figure 5.31: S4.A Performance bar plot

5.4.2 Case B: 3 Stationary Gateways

In this scenario case, only three stationary gateways are deployed to service four nodes.

In the previous case, the four nodes were positioned too far apart for a single gateway to cover multiple nodes simultaneously. If one stationary gateway were simply removed from that setup, the expected result would be a packet delivery ratio (PDR) of 75%, with fairness following a similar trend.

To create a more balanced scenario, node 0 and node 1 retain their original positions, while node 2 and node 3 are moved closer together. However, this adjustment also impacts the mobile gateway, as the distance between the two node pairs (0,1) and (2,3) increases. To compensate for this, the node pairs are positioned closer to each other.

Additionally, the placement of stationary gateways is adjusted to better suit the new configuration. Instead of being positioned adjacent to each node, two stationary gateways are placed to jointly service two nodes, while the remaining two nodes share a single gateway.

From the perspective of the stationary gateways, two nodes are expected to receive strong service, while the other two will likely experience moderate coverage, leading to a lower fairness metric.

	Position (x,y) (m)	time of first packet (s)	send interval (s)
node 0	(500,500)	400	truncnorm(1000,5)
node 1	(500,2500)	650	truncnorm(1000,5)
node 2	(2000,2000)	900	truncnorm(1000,5)
node 3	(2000,1000)	1150	truncnorm(1000,5)
mobile gateway	(1500,1500)	X	X
station. gateway 0	(600,500)	X	X
station. gateway 1	(600,2500)	X	X
station. gateway 2	(2000,1500)	X	X

Table 5.8: S4.B OMNeT++ parameters

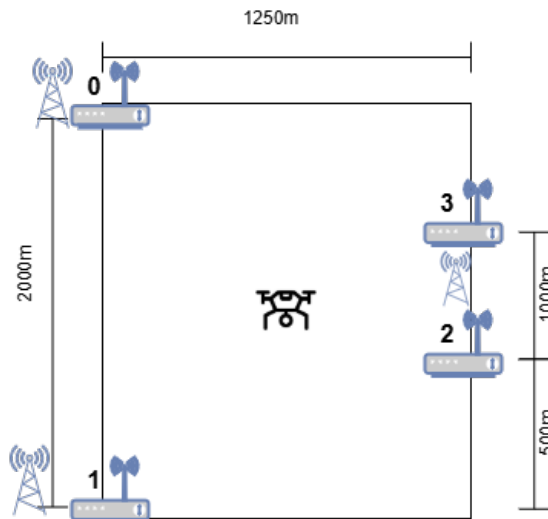


Figure 5.32: S4.B Initial positions of LoRaWAN units.

The performance of mobile and stationary gateways is highly comparable in this case. The mobile gateway achieves both a higher packet delivery ratio (PDR) and fairness; however, the stationary gateways exhibit much greater consistency, with minimal variability in performance across episodes.

The mobile gateway provides slightly worse service to node 1, while nodes 2 and 3 are poorly serviced by the stationary gateways. The performance of the stationary gateways aligns with expectations, but the behavior of the mobile gateway was unexpected. One might have anticipated a lower overall PDR for the mobile gateway, with performance being evenly distributed between the (1,2) and (2,3) node pairs, as these pairs share the same inter-node distance.

In this case, the mobile gateway proves to be a more-than-adequate substitute for the stationary gateways.

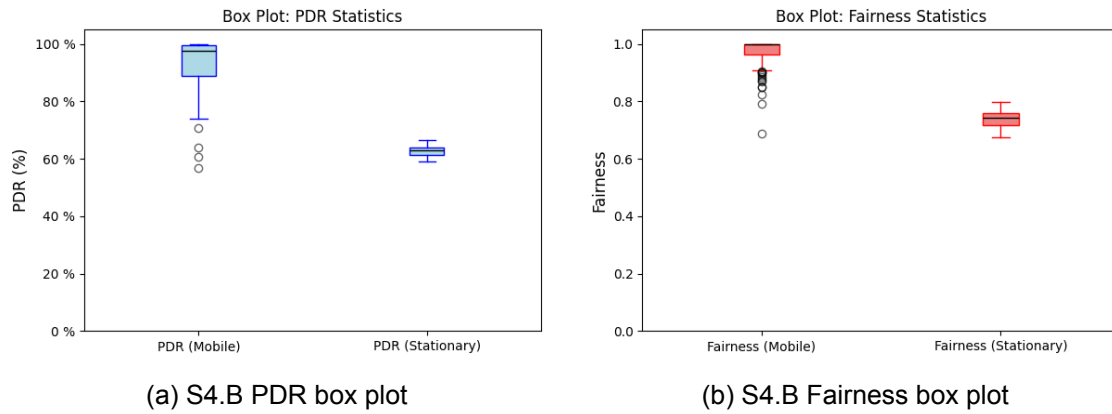


Figure 5.33: S4.B PDR and Fairness box plots

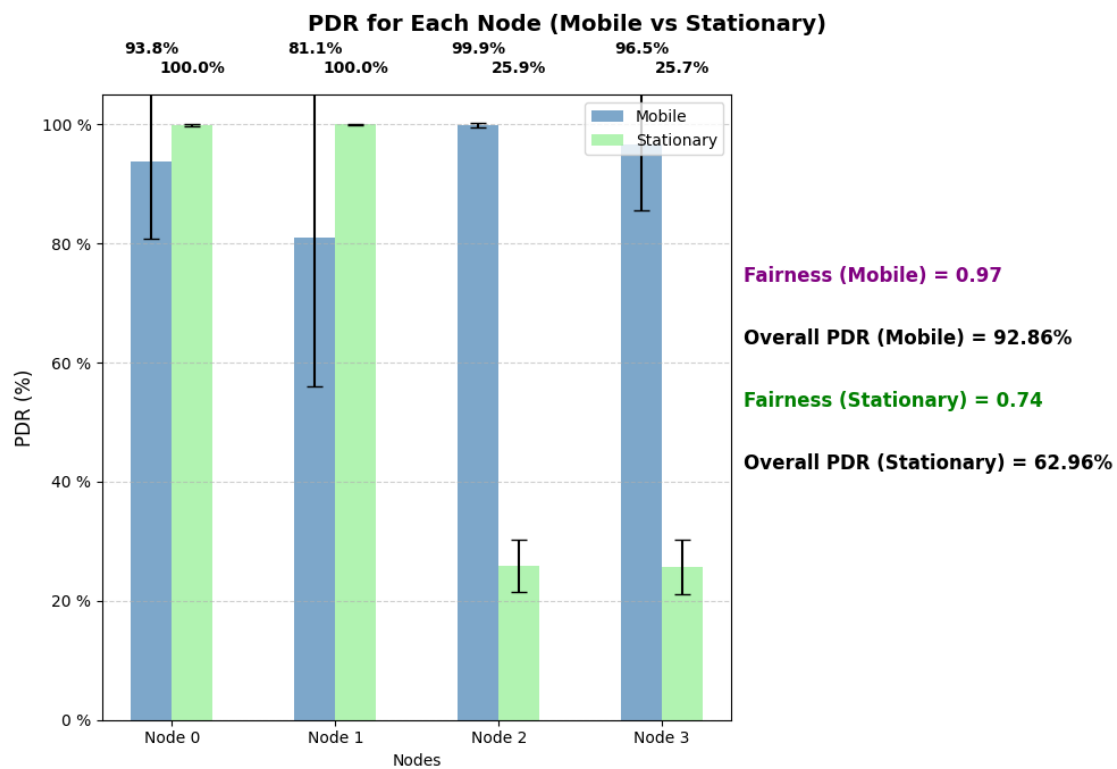


Figure 5.34: S4.B Performance bar plot

5.4.3 Case C: 2 Stationary Gateways

In this case the mobile gateway is compared against 2 stationary gateways, all-together servicing 4 nodes. 2 sets of nodes are placed in opposite corners of a grid, and stationary gateways are placed in the midpoint between them

The four nodes can be grouped in 2 clusters of (0,3) & (1,2) by their spatial proximity, of which a stationary gateway is placed at the midpoint of each. This means the stationary gateways have an euclidean distance to their respective nodes of 495m.

Furthermore, nodes 2 & 3 are configured with a send interval of 2000 s, while nodes 0 & 1 have send intervals of 1000 s.

This gives a repeating pattern for transmissions as follows:

$$0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 0 \rightarrow 1$$

Therefore, this scenario represents a mobility pattern where the mobile gateway must always move between the clusters, but periodically spend more time at each.

	Position (x,y) (m)	time of first packet (s)	Transmission interval (s)
node 0	(500,2000)	400	truncnorm(1000,5)
node 1	(1500,1000)	650	truncnorm(1000,5)
node 2	(1850,1350)	900	truncnorm(2000,5)
node 3	(850,2350)	1150	truncnorm(2000,5)
mobile gateway	(1500,1500)	X	X
station. gateway 0	(675,2175)	X	X
station. gateway 1	(1675,1175)	X	X

Table 5.9: S4.C OMNeT++ Parameters

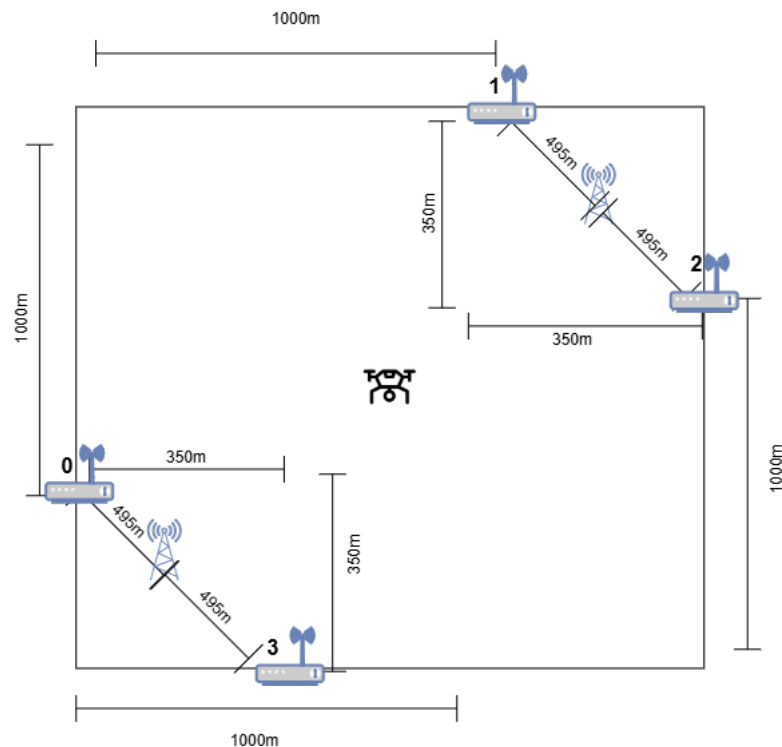


Figure 5.35: S4.C Initial positions of LoRaWAN units.

In this case, it is expected that mobile gateway exhibits lower PDR for the nodes that have a distance of 2000 m to the prior transmitting node (1 & 3), compared to nodes that are only 750 m away from the prior transmitting node (0 & 2). But since nodes 2 & 3 have lower send frequencies, their respective packet delivery rate might be higher, as the gateway has longer time to prepare for the occasional transmission. Therefore, node 1 is expected to have the lowest PDR.

For the stationary gateway, we expected equal performance across all nodes, since the conditions are identical - rate of transmission should not affect the PDR for stationary gateways.

In this case, consulting the results of (fig. 5.36a, fig. 5.36b, fig. 5.37) the mobile gateway is able to reach PDR of almost 100%, consistently across the batch of episodes. Against expectations, the Mobile gateway has equal performance across nodes. Though it is apparent that node 1 experiences a greater standard deviation of PDR, the fairness is still rounded to 1.00.

On the other hand, the 2 stationary gateways reach a PDR of 87% consistently. As such, the mobile gateway has a stable advantage over the stationary gateways for this scenario.

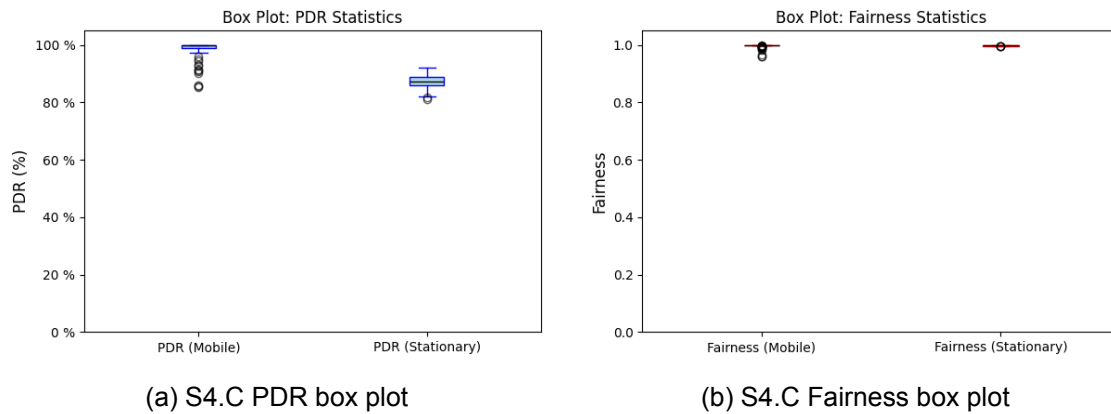


Figure 5.36: S4.C PDR and Fairness box plots

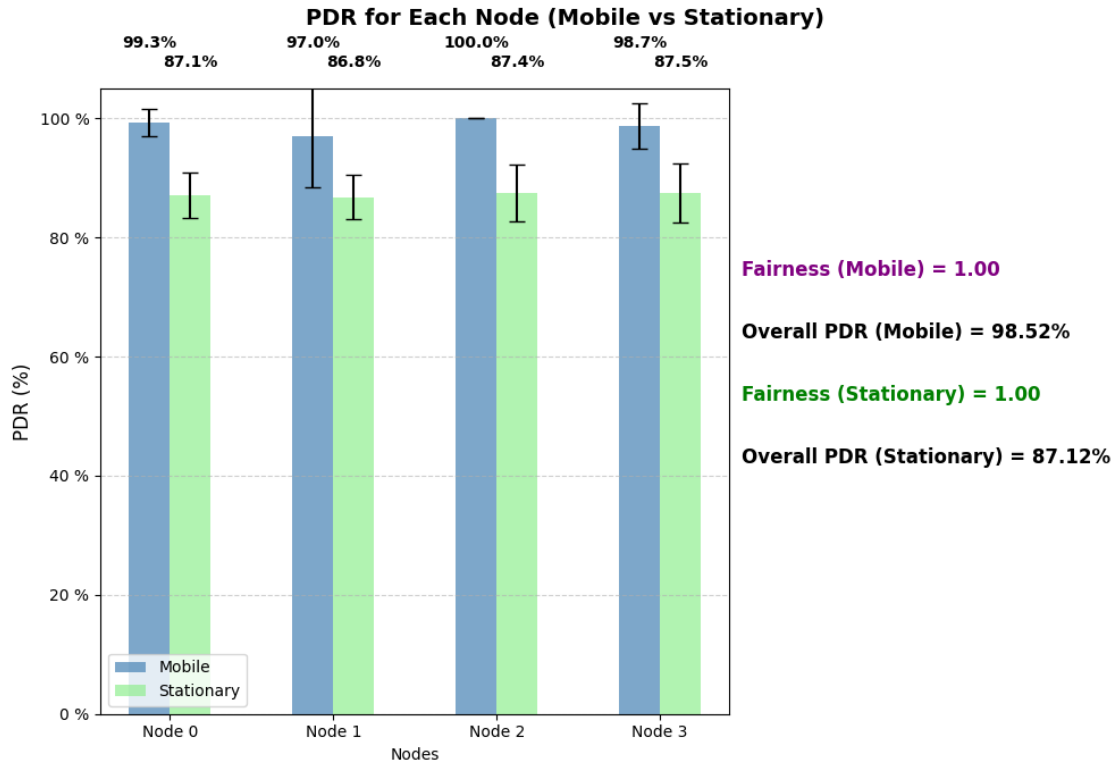


Figure 5.37: S4.C Performance bar plot

5.5 Scenario 5

The fifth scenario compares the performance of a mobile gateway utilizing reinforcement learning to a gateway with static mobility in OMNeT++.

A simple mobility model may effectively solve many scenarios without requiring complex neural network training. The term "static mobility" contrasts with mobility based on machine learning (previously referred to simply as the "mobile gateway"), which dynamically adapts to circumstances. In contrast, static mobility follows a predetermined movement pattern defined solely by the scenario configuration. To distinguish between the two approaches, we use the terms **RL mobility** and **static mobility**.

In this scenario, we evaluate a static mobility gateway following a circular motion pattern. This motion is defined by its center position, radius, linear velocity, and initial angular position.

The experiments in this scenario aim to assess how effectively the RL mobile gateway performs compared to the static mobility gateway under various setups, including configurations that are both favorable and unfavorable for static mobility.

The results will show that static mobility performs well only in ideal cases, when it is specifically configured. In contrast, the performance of the RL-based mobile gateway is influenced by the distance between nodes; while it adapts to different situations, its performance lacks robustness.

5.5.1 Case A: Evenly positioned nodes on circle

In this case, we investigate whether a well-tuned static mobility gateway can achieve high performance in an environment specifically suited for it. The four nodes are positioned equidistantly on a larger circle, each transmitting at harmonic frequencies (a consistent transmission spacing) with equal offsets in transmission times. By carefully tuning the constant angular velocity of the circular motion along with its initial angular position, the static gateway can arrive at the exact location of a node at its transmission time.

The nodes are placed on a circle with a radius of 1000m, resulting in a Manhattan distance of 2000m between sequentially transmitting nodes. To ensure equal spacing, the nodes are positioned at angular intervals of $2\pi/4 = \pi/2$ radians.

	Position (x,y) (m)	time of first packet (s)	Transmission interval (s)
node 0	(1500,1500)	400	truncnorm(1000,5)
node 1	(2500,500)	650	truncnorm(1000,5)
node 2	(3500,1500)	900	truncnorm(2000,5)
node 3	(2500,2500)	1150	truncnorm(2000,5)
RL mobile gateway	(2500,1500)	χ	χ

Table 5.10: S5.A OMNeT++ Parameters

The same transmission timings as in section 5.4.3 is used. This means there is 250 seconds between transmissions occurring. It is desired to tune the circular motion to ideal values for this scenario. Therefore the static mobility gateway should have an angular velocity of

$$\omega = \frac{\frac{\pi}{2}}{250 \text{ s}} = \frac{\pi}{500} \frac{\text{rad}}{\text{s}} \quad (5.4)$$

Therefore, the linear velocity will be

$$\left(\frac{\pi}{500} \frac{\text{rad}}{\text{s}}\right) \cdot (1000\text{m}) = 2\pi \frac{\text{m}}{\text{s}} \approx 6.28 \frac{\text{m}}{\text{s}} \quad (5.5)$$

The initial angular position of the static gateway is determined by the position of the first transmitting node. Node 0 transmits the first packet at $t_{\text{node0},0} = 400\text{s}$. Based on the Cartesian coordinates of node 0, it has an angular position of $\pi \text{ rad}$ on the circle centered at (2500,1500). As the circle mobility has an angular velocity of $\pi/2$, it can be calculated that

$$\theta_{\text{initial}} + t_{\text{node0},0} \cdot \omega = \theta_{\text{node0}} \Rightarrow \theta_{\text{initial}} = \theta_{\text{node0}} - t_{\text{node0},0} \cdot \omega = \pi - 400\text{s} \cdot \frac{\pi}{500} \frac{\text{rad}}{\text{s}} = \frac{\pi}{5} \quad (5.6)$$

	Initial angle (rad)	Center Position (x,y) [m]	Radius [m]	Linear velocity [m/s]
Circle mobility gateway	0.628	(2500,1500)	1000	6.28

Table 5.11: S5.A Circular motion parameters

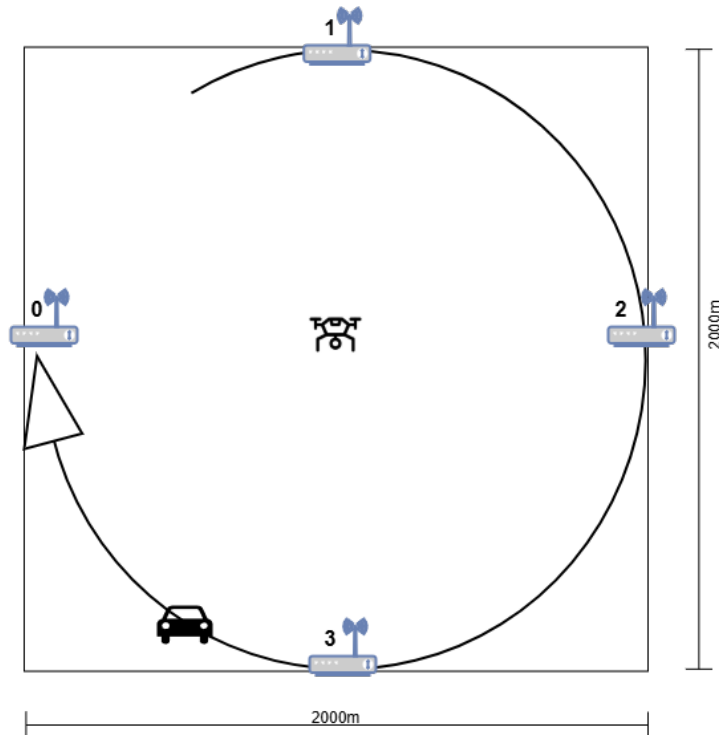
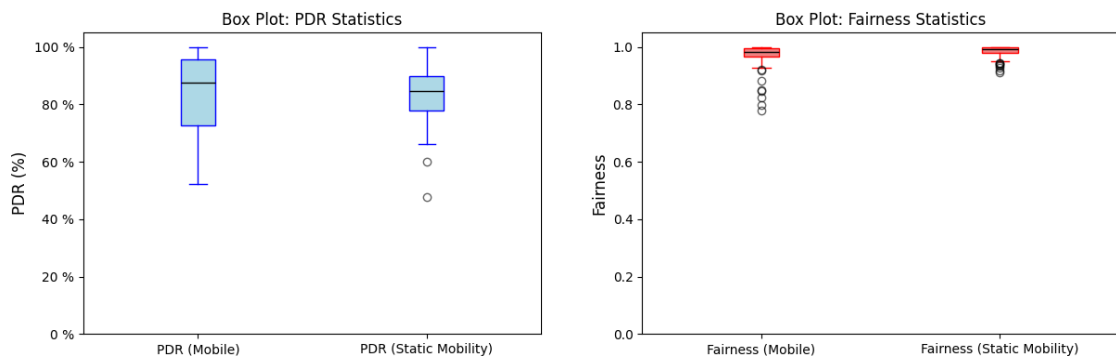


Figure 5.38: S5.A Initial positions of LoRaWAN units.

Based on the results of fig. 5.39a, fig. 5.39b, fig. 5.40, the performance of RL mobility and ideally configured static mobility can be considered equivalent. The expectation of the static mobility is near-perfect PDR, but it is only able to achieve 85%. This must be due to the stochastic deviation in transmission times, that the static mobility is not able to adjust to. But even though the average performance of both mobility is equivalent, the range of PDR for RL Mobility has a larger distribution. This indicates that it is less stable across episodes. Furthermore, RL Mobility has a high fairness, yet it is still apparent that node 1 is not serviced well, as it has a lower average PDR and a high deviation.

To conclude, an ideally configured circular motion attains both high PDR and fairness, but is not perfect. The RL mobility can reach the same average performance, but is not as reliable within the temporal-spatial constraints of this scenario.



(a) S5.A PDR bar plot

(b) S5.A Fairness bar plot

Figure 5.39: S5.A PDR and Fairness bar plots

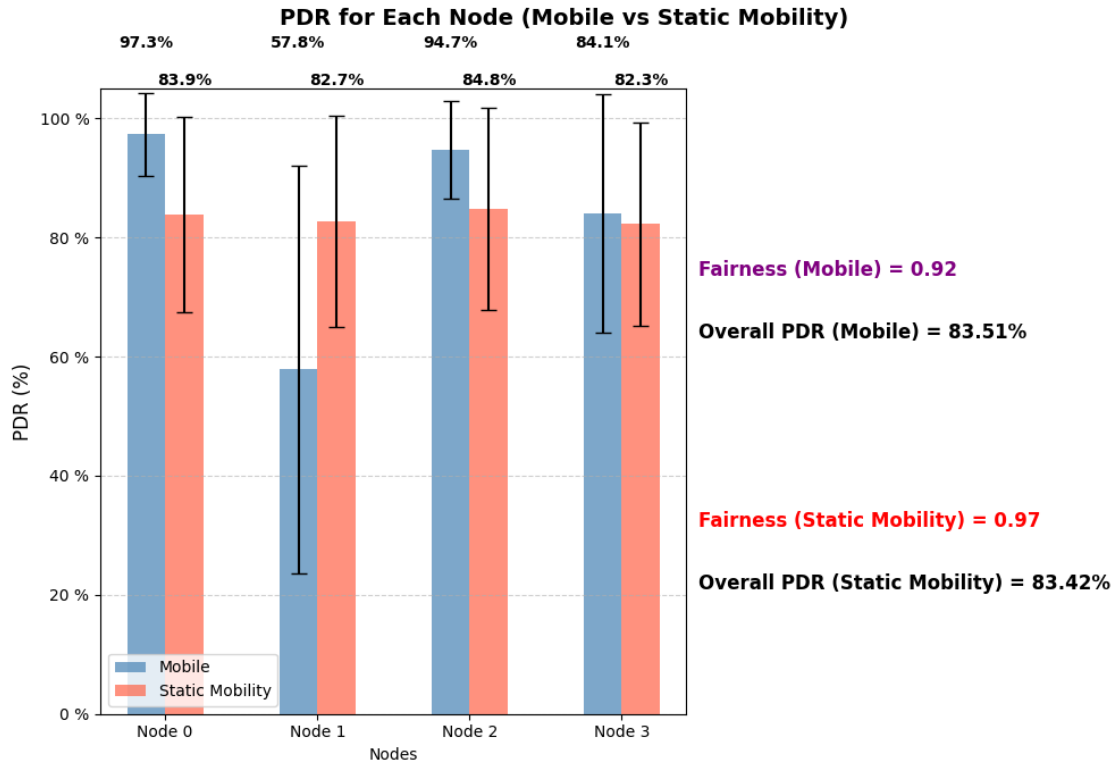


Figure 5.40: S5.A Performance bar plot

5.5.2 Case B: Dispersed nodes approximating circle

In most real-world scenarios, nodes are not perfectly arranged in a circular or any other ideal formation. This investigation aims to provide insights into how our RL-based mobility-driven gateway performs in comparison to a static-mobility-driven one. The case is based on the scenario outlined in section 5.5.1, with a slight modification to the node positions. The nodes are now shifted in a Manhattan grid with distances of 300m, 500m, 350m, and 300m, respectively.

	Position (x,y) (m)	Time of first packet (s)	Transmission interval (s)
node 0	(1700,1400)	400	truncnorm(1000,5)
node 1	(2700,800)	650	truncnorm(1000,5)
node 2	(3800,1450)	900	truncnorm(2000,5)
node 3	(2400,2300)	1150	truncnorm(2000,5)
RL mobile gateway	(2500,1500)	χ	χ

Table 5.12: S5.B OMNeT++ Parameters

	Initial angle (rad)	Center Position (x,y) [m]	Radius [m]	Linear velocity [m/s]
Circle mobility gateway	0.628	(2500,1500)	1000	6.28

Table 5.13: S5.B Circular motion parameters

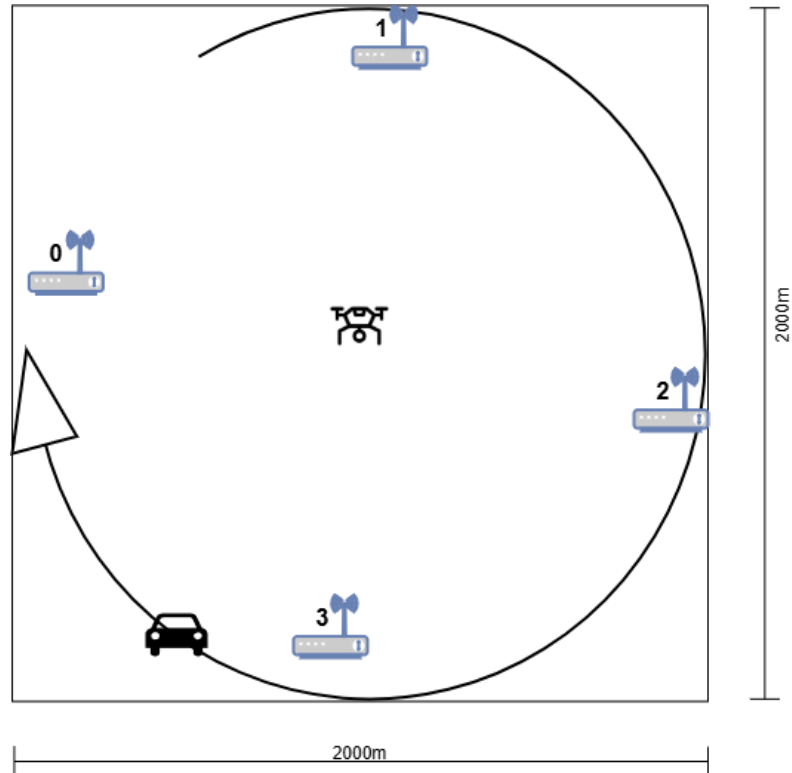
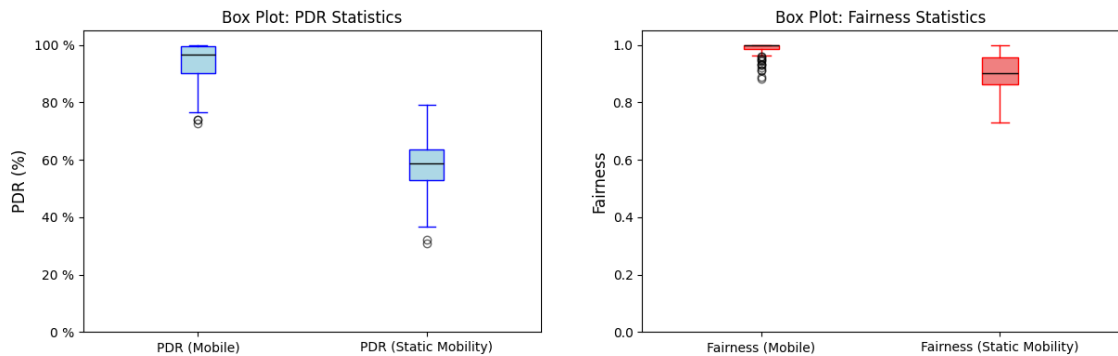


Figure 5.41: S5.B Initial positions of LoRaWAN units.

Looking at (fig. 5.42a,fig. 5.42b,fig. 5.43) The performance of the static gateway sharply decreases by the displacement of nodes from the circle. The performance is no longer fair, and much lower PDR. Particularly, fewer packets from node 1 are received by the static gateway. This can be rationalized as being due to the larger deviation from the circle than other nodes. On the other hand, RL-mobile gateway actually achieves even better performance than in section 5.5.1. This is likely attributed to the fact that distances between each node is decreased, reducing the need for exact, optimal decisions.

In this case, its apparent that RL mobility handles deviation from the circular distribution of nodes, much better than the static mobility gateway.



(a) S5.B PDR box plot

(b) S5.B Fairness box plot

Figure 5.42: S5.B PDR and Fairness box plots

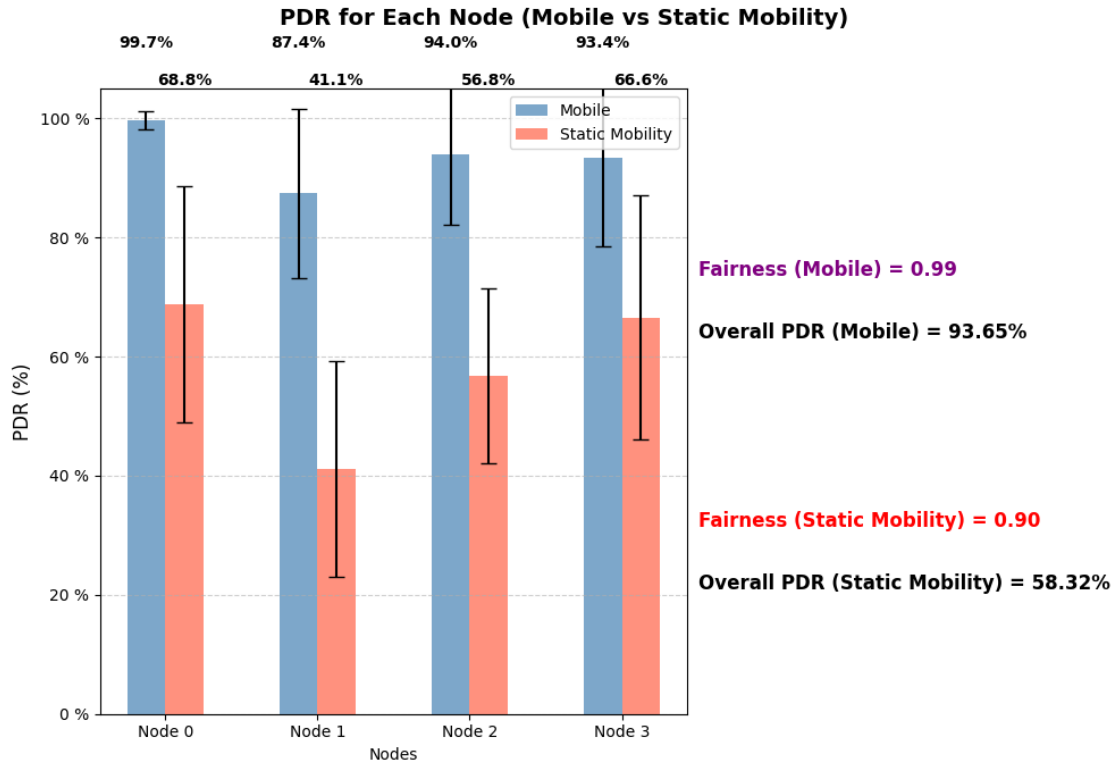


Figure 5.43: S5.B Performance bar plot

5.5.3 Case C: Evenly positioned nodes on a larger grid

In this case, nodes are once again positioned on a grid. However, the sequentially transmitting nodes are now positioned at a distance from each other exactly equal to:

$$\text{Distance}_{T1,T2} = v_{RL_gateway} \cdot \Delta T_{transmissions} \quad (5.7)$$

Substituting the given values:

$$\text{Distance}_{T1,T2} = 11 \frac{\text{m}}{\text{s}} \cdot 250 \text{ s} = 2750 \text{ m} \quad (5.8)$$

This means that any suboptimal actions chosen by RL mobility might lead to a lasting degradation in service across an episode. Therefore, it can be expected that both PDR and fairness must be lower than in scenarios with ample time between transmissions, such as section 5.2.

	Position (x,y) (m)	Time of first packet (s)	Transmission interval (s)
node 0	(500,500)	400	truncnorm(1000,5)
node 1	(500,3250)	650	truncnorm(1000,5)
node 2	(3250,3250)	900	truncnorm(1000,5)
node 3	(3250,500)	1150	truncnorm(1000,5)
RL mobile gateway	(500,800)	\times	\times

Table 5.14: S5.C OMNeT++ parameters

On the other hand, the circular motion of the Static Mobility gateway is configured such that its linear velocity equals that of the RL Mobile gateway, i.e., $v_{\text{static_gateway}} = 11 \frac{\text{m}}{\text{s}}$. The center of circular motion is placed at the midpoint of all nodes, enabling it to ideally provide fair performance.

We maintain that the static mobility gateway performs a full rotation in the same time as the transmission intervals of the nodes, i.e., 1000 s. Thus, the static mobility gateway has an angular velocity of:

$$\omega_{\text{static_gateway}} = \frac{2\pi}{1000 \text{ s}} \quad (5.9)$$

The radius of the circle is derived from the linear velocity, such that:

$$v_{\text{static_gateway}} = \text{radius}_{\text{static_gateway}} \cdot \omega_{\text{static_gateway}} \Rightarrow \text{radius}_{\text{static_gateway}} = \frac{11 \frac{\text{m}}{\text{s}}}{\frac{2\pi}{1000 \text{ s}}} \approx 1751 \text{ m} \quad (5.10)$$

The radius of this circle exceeds half the length of the sides formed by the node grid but does not exceed the distance from the center of the circle to the nodes. Thus, the circle intersects the grid at two points per side, near the corners. This results in the motion being a circle intersecting the square formed by the node positions.

The minimum distance between the static mobility gateway and a node at transmission time is given by:

$$\left| \sqrt{1375^2 + 1375^2} - 1375 \right| \approx 194 \text{ m} \quad (5.11)$$

The initial angle is configured such that the distance to the nodes is minimized at their time of transmission, similarly to section 5.5.1.

	Initial angle (rad)	Center Position (x,y) [m]	Radius [m]	Linear velocity [m/s]
Circle mobility gateway	0.628	(1875,1875)	1375	8.64

Table 5.15: S5.C Circular motion parameters

The performance of static mobility is expected to be fair, just as section 5.5.1 exhibited, but lower RL since there will be some distance away from the transmitting node, albeit short distance.

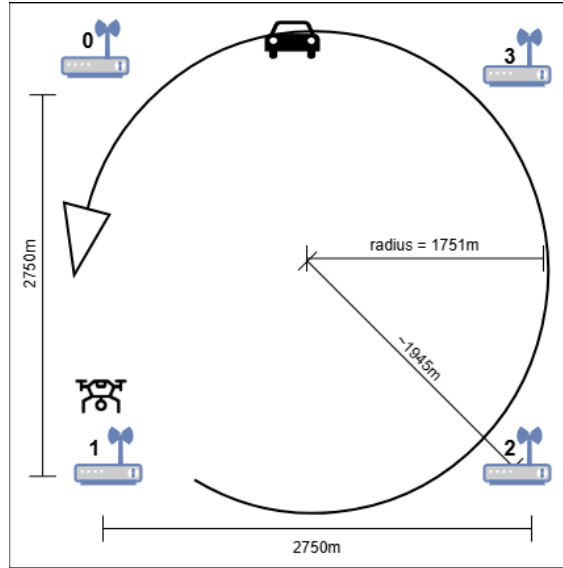
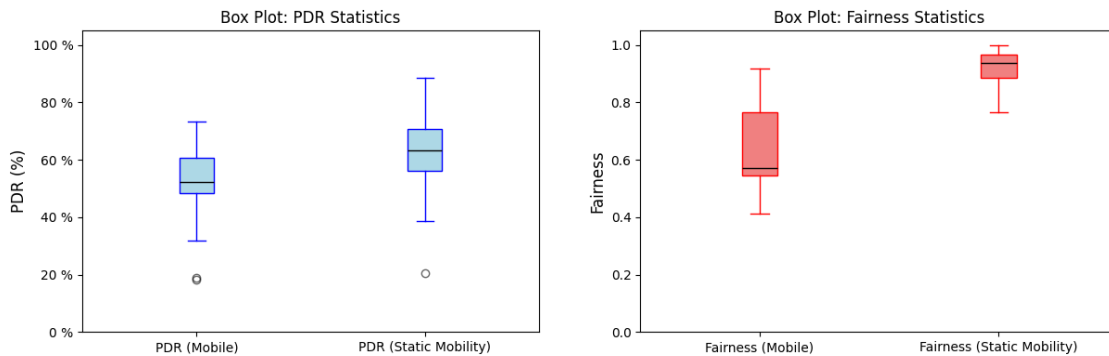


Figure 5.44: S5.C Initial positions of LoRaWAN units.

In this case, it is apparent that RL Mobility does not serve the nodes fairly. Specifically, node 0 is neglected, while node 2 seems to be prioritized. Additionally, the average PDR achieved by RL mobility is relatively low, and the deviation of both the PDR and fairness metrics is wide. This demonstrates that having ample time to traverse between nodes and receive packets is crucial, as simply having the exact amount of time is insufficient for optimal performance.

The static mobility gateway, on the other hand, performed as expected, exhibiting completely fair behavior. However, the reduction in average PDR compared to section 5.5.1 was significant, with a decrease of approximately 20%—from 83% to 63%. This drop is surprising and cannot be attributed solely to the increased minimum distance (from 0m to 195m). It is more likely a result of the static mobility gateway following a larger circle with a higher linear velocity. As a result, the distance between the gateway and nodes at average transmission times is also increased. Consequently, any delays or early transmissions could lead to a greater distance between the gateway and the transmitting node, further impacting the PDR.



(a) S5.C PDR box plot

(b) S5.C Fairness box plot

Figure 5.45: S5.C PDR and Fairness box plots

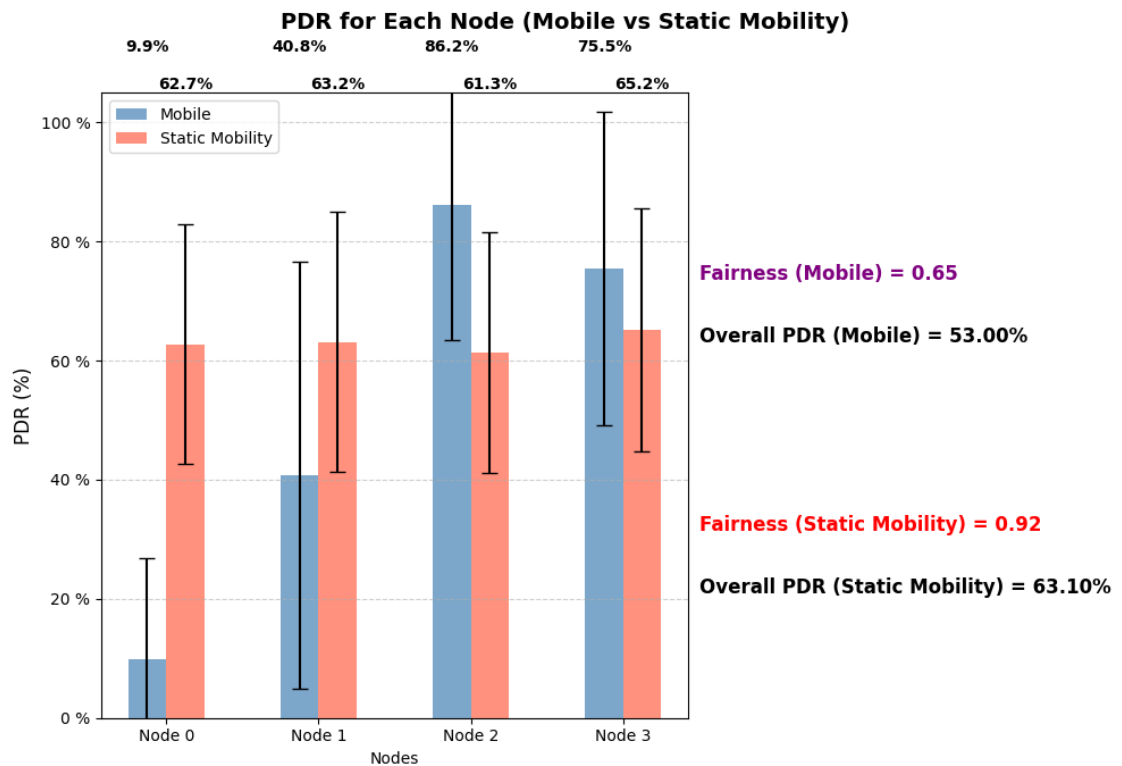


Figure 5.46: S5.C Performance bar plot

6 Discussion and Feasibility

From our results, a mobile gateway trained using RL shows promising performance in both the training and evaluation environments. However, determining its real-world applicability requires an investigation into the feasibility of deployment in practical use cases. This feasibility depends on multiple factors, including vehicle constraints, computational resources, physical movement limitations, data availability, and network scalability.

The type of vehicle used as the mobile gateway was left unspecified in this project, with speed parameters based on regulations for delivery UAVs. Real-world implementation depends on constraints imposed by the vehicle type, whether aerial, aquatic, or terrestrial. Each presents unique mobility challenges: terrestrial vehicles are more prone to encountering obstacles, while aerial and aquatic vehicles are affected by external forces such as wind and waves. Furthermore, the simulation did not account for realistic physical movement acceleration was assumed to be instant, allowing the vehicle to change direction or speed instantaneously. In a real-world scenario, inertia, drag, and acceleration limits must be considered. A practical deployment would likely involve a robotics controller to interpret high-level movement decisions from the neural network and translate them into feasible physical actions.

Another critical factor is the computational resources required for model inference. A realistic baseline for embedded deployment is the Raspberry Pi 4 Model B (4GB)¹, which features 4GB SDRAM, WiFi, Bluetooth, and multiple USB ports which could allow for connecting a LoRa module. It also has a MicroSD card slot for ample storage and operates at approximately 15W (5V, 3A). While the Raspberry Pi is more of a miniature PC than a microcontroller, it remains relevant for IoT applications due to its balance of performance and form factor. However, its power consumption is relatively high compared to typical embedded devices, which may be a limiting factor for battery-powered applications.

The final trained model consists of 12 input nodes and 6 hidden layers, amounting to 444 nodes and 18304 weights. Of these, the critic network contains 4 layer, 218 nodes, and 9024 weights, meaning that only 4 layers, 226 nodes, and 9280 weights are necessary for inference. The exported TFLite model file has a flash size of 40 kiloByte, which is minimal compared to the available memory on most embedded platforms. Based on results from the study "Quantization and Deployment of Deep Neural Networks on Microcontrollers" [33], where a model of approximately 410 kiloByte was deployed on the SparkFun Edge microcontroller using TFLite Micro, it had an inference time of approximately 2000ms per input, a linear scaling assumption suggests that our model's inference time would be approximately

$$2000 \text{ ms} \times \frac{40 \text{ kiloByte}}{410 \text{ kiloByte}} = 0.195 \text{ s.} \quad (6.1)$$

On a Raspberry Pi 4, with significantly greater computational resources, the inference time would be expected to decrease substantially. In this project, inference was performed every 10 seconds, meaning the proportion of time spent on inference is approximately

$$\frac{0.195 \text{ s}}{10 \text{ s}} = 1.95\%. \quad (6.2)$$

¹<https://www.raspberrypi.com/products/raspberry-pi-4-model-b/specifications/>

In the simulation framework, inference was treated as instantaneous, but in real-world applications, execution time becomes a critical factor. If inference latency is too high, the gateway's movement decisions may be based on outdated information, reducing overall performance. Furthermore, energy consumption is also a key concern. The study also reported that a 410 kiloByte model consumed $1.56 \mu\text{Wh}$ per inference. Extrapolating for our smaller model size, the estimated energy consumption per inference is

$$1.56 \mu\text{Wh} \times \frac{40 \text{ kiloByte}}{410 \text{ kiloByte}} = 0.1523 \mu\text{Wh} = 548.28 \mu\text{J}. \quad (6.3)$$

Over a 10-second period, this results in an average power consumption of

$$\frac{548.28 \mu\text{J}}{10 \text{ s}} = 54.83 \mu\text{W}. \quad (6.4)$$

These values remain well within the power budget of a Raspberry Pi 4. However, since these estimates are based on a different hardware platform, they should be viewed as rough approximations. Nevertheless, they indicate that exporting the model to an embedded system is feasible. Beyond computational and energy constraints, the deployment of a mobile gateway must consider the availability of required input data. The model assumes knowledge of end-node positions, the gateway's own position (e.g., via GPS), the mean send interval of nodes, and a method for synchronizing with transmission schedules, such as tracking the time of the first packet received. While this information is easily accessible in a simulation environment, obtaining it in a real-world deployment may introduce additional challenges.

This thesis has been conducted without considering LoRaWAN device classes, as nodes only transmit join messages, which are uplink transmissions. However, many practical applications rely on both uplink and downlink communication, making device class an important factor in feasibility. For instance, Class B devices periodically open receive windows, requiring gateways to remain nearby longer to support potential downlink transmissions. Class C devices, which are almost always available for downlink reception, demand careful coordination to avoid interference with uplink transmissions.

A major limitation of the current model is its ability to consider only exactly 4 nodes as input at any given time. In real-world LoRaWAN networks, which may consist of hundreds or thousands of nodes, this is an evident scalability issue. One possible approach is to dynamically select a subset of nodes as input, changing throughout an episode based on relevance. Alternatively, a model capable of handling a larger number of nodes simultaneously could be explored, although this would likely require a larger neural network with higher computational demands. A hybrid approach, where a preprocessing mechanism filters and prioritizes the most relevant nodes before passing them to the main model, could provide a balance between efficiency and scalability. Furthermore, the evaluations assumed uniform signal propagation characteristics among nodes, including Transmission Power and Spreading Factor. In practice, LoRaWAN networks can consist of heterogeneous device types with varying transmission power, antenna strengths, and communication frequencies. Some nodes may transmit less frequently, while others may have weaker signals due to hardware limitations or environmental interference. These factors could alter the optimal movement strategy for the gateway, requiring it to adapt dynamically, such as by prioritizing proximity to nodes with weaker signals to improve communication reliability.

7 Conclusion

This thesis explored the use of mobility in LoRa gateways driven by Reinforcement Learning. Our research demonstrates that Reinforcement Learning-based mobility can be effectively implemented in OMNeT++, and that models trained in a Low-Fidelity custom environment like Stable-Baselines3 (SB3) can be successfully transferred to OMNeT++ while maintaining strong performance. This approach enables rapid development and experimentation with custom environments and learning processes, as different frameworks and tools can be easily tested. Throughout this study, several challenges emerged. Developing a framework to support RL models in OMNeT++ was a significant task, with much of the complexity stemming from designing a custom environment and training models in SB3.

From a training perspective, experiments with both PPO and DQN revealed that while DQN converges similarly, studies show that PPO is better suited for complex environments requiring continuous control. This is especially important if the observation space grows, by adding more nodes. Additionally, the use of residual connections was shown to accelerate convergence. Unfortunately, due to conversion limitations between PyTorch and TensorFlow Lite, the final deployed model did not incorporate residual connections, which may guide the next steps in research.

Our approach achieved a near-perfect packet delivery ratio (PDR) in High-Fidelity OMNeT++ simulations. Compared to the conventional approach of using stationary gateways, the RL-driven gateway performed favorably, proving capable of substituting them. However, evaluation of the final model revealed that it lacks robustness in general scenarios and cannot provide performance guarantees without prior evaluation.

In comparison to other methods of controlling a mobile gateway, such as "static mobility," circular motion was evaluated and shown to perform well in specific scenarios. However, its effectiveness is highly scenario-dependent and requires specific tuning of the motion, whereas the RL-driven gateway is more adaptable and performs better when static mobility does not precisely align with the environment. Notably, the RL-driven gateway exhibits a broader performance distribution across episodes, exactly because of its dynamic nature, as its behavior is affected by stochastic events.

The study also examined the gateway's behavior in the presence of faulty nodes. In scenarios where a node fails, the agent continues to serve the remaining nodes, albeit sometimes at the cost of waiting for an expected packet from the faulty node. Adjusting the expected packet times proved to be an effective strategy for mitigating this issue, highlighting the critical role of adjusting input variables in guiding the gateway's decision-making.

The RL-driven gateway demonstrated potential but struggled with complex decision-making, particularly when serving nodes transmitting in close proximity or in the presence of faulty nodes, highlighting the need for a more advanced strategy to enhance robustness. Although the final trained model does not yet exhibit perfect behavior and is not immediately suitable for real-world deployment, the framework developed in this project provides a strong foundation for further research. By refining the model to account for factors such as vehicle dynamics, computational constraints, real-world data availability, and network scalability, this work has the potential to contribute to more robust industrial IoT applications, extending beyond theoretical simulations.

8 Future works

Future research could focus on training a gateway to operate effectively in more challenging environments. This includes scenarios where the gateway must make more complex decisions, such as prioritizing which nodes to serve when not all are reachable at all times or detecting faulty nodes. Additionally, teaching the gateway to explore new nodes while efficiently handling a larger number of them would be valuable.

Another interesting direction is clustering multiple closely positioned nodes into "super nodes," framing the problem as a clustering task to improve scalability. Furthermore, incorporating multiple mobile gateways working together to dynamically service stationary nodes introduces a multi-agent Reinforcement Learning (RL) challenge with significant potential.

Reframing the problem into a multi-objective RL framework could also be beneficial. One objective could focus on selecting target positions, while another handles path planning, especially in environments with obstacles where the most direct path is not always feasible.

Lastly, improving the realism of LoRa emulation in SB3 training could be crucial. As the number of nodes scales up, network collisions may become more frequent, requiring a more accurate model of these interactions.

In conclusion, there is vast potential for future exploration in this domain, with numerous directions that could enhance both the efficiency and adaptability of RL-driven gateways.

Bibliography

- [1] S. R. Jino Ramson et al. "A Self-Powered, Real-Time, LoRaWAN IoT-Based Soil Health Monitoring System". In: *IEEE Internet of Things Journal* 8.11 (June 2021), pp. 9278–9293. ISSN: 2327-4662. DOI: [10.1109/JIOT.2021.3056586](https://doi.org/10.1109/JIOT.2021.3056586).
- [2] Sarun Duangsuwan et al. "A Study of Air Pollution Smart Sensors LPWAN via NB-IoT for Thailand Smart Cities 4.0". In: *2018 10th International Conference on Knowledge and Smart Technology (KST)*. Jan. 2018, pp. 206–209. DOI: [10.1109/KST.2018.8426195](https://doi.org/10.1109/KST.2018.8426195).
- [3] Fatima Salahdine, Shobhit Aggarwal, and Asis Nasipuri. "Short-Term Traffic Congestion Prediction with Deep Learning for LoRa Networks". In: *SoutheastCon 2022*. Mar. 2022, pp. 261–268. DOI: [10.1109/SoutheastCon48659.2022.9763927](https://doi.org/10.1109/SoutheastCon48659.2022.9763927).
- [4] *MOKO Smart: LoRa illustration of chirps*. URL: <https://www.mokosmart.com/lora-frequency/>.
- [5] *AN1200.22 LoRa Modulation Basics*. URL: <https://semtech.my.salesforce.com/sfc/p/#E0000000JelG/a/2R0000001OJk/yDEcfAkD9qEz6oG3PJryoHKas3UMsMDa3TFqz1UQOkM>.
- [6] Reyhane Falanji, Martin Heusse, and Andrzej Duda. "Range and Capacity of LoRa 2.4 GHz". In: *Mobile and Ubiquitous Systems: Computing, Networking and Services*. Ed. by Shangguan Longfei and Priyantha Bodhi. Cham: Springer Nature Switzerland, 2023, pp. 403–421. ISBN: 978-3-031-34776-4.
- [7] *The Network Things: Visualization example of code rate*. URL: <https://www.thethingsnetwork.org/docs/lorawan/fec-and-code-rate/>.
- [8] *All About LoRa and LoRaWAN*. URL: <https://www.sghosly.com/p/lor-is-chirp-spread-spectrum.html>.
- [9] *The Things Network LoRaWAN Adaptive Data Rate*. URL: <https://www.thethingsnetwork.org/docs/lorawan/adaptive-data-rate/>.
- [10] Mariusz Slabicki, Gopika Premasankar, and Mario Di Francesco. "Adaptive configuration of lora networks for dense IoT deployments". In: *NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium*. Apr. 2018, pp. 1–9. DOI: [10.1109/NOMS.2018.8406255](https://doi.org/10.1109/NOMS.2018.8406255).
- [11] *GeeksforGeeks: Visualization of neural network*. URL: <https://www.geeksforgeeks.org/neural-networks-a-beginners-guide/>.
- [12] Nitish Srivastava et al. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". In: *Journal of Machine Learning Research* 15.56 (2014), pp. 1929–1958. URL: <http://jmlr.org/papers/v15/srivastava14a.html>.
- [13] Kaiming He et al. *Deep Residual Learning for Image Recognition*. 2015. arXiv: [1512.03385](https://arxiv.org/abs/1512.03385) [cs.CV]. URL: <https://arxiv.org/abs/1512.03385>.
- [14] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, et al. "Human-level control through deep reinforcement learning". In: *Nature* 518.7540 (2015), pp. 529–533. DOI: [10.1038/nature14236](https://doi.org/10.1038/nature14236).
- [15] John Schulman et al. *Proximal Policy Optimization Algorithms*. 2017. arXiv: [1707.06347](https://arxiv.org/abs/1707.06347) [cs.LG]. URL: <https://arxiv.org/abs/1707.06347>.
- [16] Antonin Raffin et al. "Stable-Baselines3: Reliable Reinforcement Learning Implementations". In: *Journal of Machine Learning Research* 22.268 (2021), pp. 1–8. URL: <http://jmlr.org/papers/v22/20-1364.html>.
- [17] *Stable-Baselines3: Visualization of network architecture*. URL: https://stable-baselines3.readthedocs.io/en/master/guide/custom_policy.html.

- [18] Andrew J Wixted et al. "Evaluation of LoRa and LoRaWAN for wireless sensor networks". In: *2016 IEEE SENSORS*. Oct. 2016, pp. 1–3. DOI: [10.1109/ICSENS.2016.7808712](https://doi.org/10.1109/ICSENS.2016.7808712).
- [19] "Performance Evaluation of LoRa Networks in an Open Field Cultivation Scenario". In: DOI: [10.1109/MOCAS52088.2021.9493416](https://doi.org/10.1109/MOCAS52088.2021.9493416).
- [20] Nabil Abdoun et al. "Collision Detection in LoRaWAN Using Machine Learning". In: *2024 IEEE 22nd Jubilee International Symposium on Intelligent Systems and Informatics (SISY)*. Sept. 2024, pp. 1–6. DOI: [10.1109/SISY62279.2024.10737620](https://doi.org/10.1109/SISY62279.2024.10737620).
- [21] Mukhammad Gufron Ikhsan et al. "Mobile LoRa Gateway for Smart Livestock Monitoring System". In: *2018 IEEE International Conference on Internet of Things and Intelligence System (IOTAIS)*. Nov. 2018, pp. 46–51. DOI: [10.1109/IOTAIS.2018.8600842](https://doi.org/10.1109/IOTAIS.2018.8600842).
- [22] Sugianto Sugianto et al. "Simulation of Mobile LoRa Gateway for Smart Electricity Meter". In: *2018 5th International Conference on Electrical Engineering, Computer Science and Informatics (EECSI)*. Oct. 2018, pp. 292–297. DOI: [10.1109/EECSI.2018.8752818](https://doi.org/10.1109/EECSI.2018.8752818).
- [23] Ciyuan Chen et al. "LoRaDrone: Enabling Low-Power LoRa Data Transmission via a Mobile Approach". In: *2022 18th International Conference on Mobility, Sensing and Networking (MSN)*. 2022, pp. 239–246. DOI: [10.1109/MSN57253.2022.00050](https://doi.org/10.1109/MSN57253.2022.00050).
- [24] Jintaro Nogae et al. "Comparison of reinforcement learning in game AI". In: *2022 23rd ACIS International Summer Virtual Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD-Summer)*. July 2022, pp. 82–86. DOI: [10.1109/SNPD-Summer57817.2022.00022](https://doi.org/10.1109/SNPD-Summer57817.2022.00022).
- [25] Rishabh Sharma and Prateek Garg. "Optimizing Autonomous Vehicle Navigation with DQN and PPO: A Reinforcement Learning Approach". In: *2024 Asia Pacific Conference on Innovation in Technology (APCIT)*. July 2024. DOI: [10.1109/APCIT62007.2024.10673440](https://doi.org/10.1109/APCIT62007.2024.10673440).
- [26] Neil De La Fuente and Daniel A. Vidal Guerra. *A Comparative Study of Deep Reinforcement Learning Models: DQN vs PPO vs A2C*. 2024. arXiv: [2407.14151](https://arxiv.org/abs/2407.14151) [cs.LG]. URL: <https://arxiv.org/abs/2407.14151>.
- [27] Aske Plaat, Walter Kusters, and Mike Preuss. *Deep Model-Based Reinforcement Learning for High-Dimensional Problems, a Survey*. 2020. arXiv: [2008.05598](https://arxiv.org/abs/2008.05598) [cs.LG]. URL: <https://arxiv.org/abs/2008.05598>.
- [28] Abdullahi Isa Ahmed and El Mehdi Amhoud. *Energy-Efficient Flying LoRa Gateways: A Multi-Agent Reinforcement Learning Approach*. 2025. arXiv: [2502.03377](https://arxiv.org/abs/2502.03377) [cs.NI]. URL: <https://arxiv.org/abs/2502.03377>.
- [29] Yash Shukla et al. *ACuTE: Automatic Curriculum Transfer from Simple to Complex Environments*. 2022. arXiv: [2204.04823](https://arxiv.org/abs/2204.04823) [cs.R0]. URL: <https://arxiv.org/abs/2204.04823>.
- [30] Andrew Y. Ng, Daishi Harada, and Stuart J. Russell. "Policy Invariance Under Reward Transformations: Theory and Application to Reward Shaping". In: *Proceedings of the Sixteenth International Conference on Machine Learning*. ICML '99. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999, pp. 278–287. ISBN: 1558606122.
- [31] Sorawit Saengkyongam et al. "Invariant Policy Learning: A Causal Perspective". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 45.7 (July 2023), pp. 8606–8620. ISSN: 1939-3539. DOI: [10.1109/TPAMI.2022.3232363](https://doi.org/10.1109/TPAMI.2022.3232363).
- [32] LoRa Alliance. *LoRaWAN Specification v1.1*. Accessed: 2025-02-16. 2017. URL: <https://resources.lora-alliance.org/technical-specifications/lorawan-specification-v1-1>.

- [33] Pierre-Emmanuel Novac et al. "Quantization and Deployment of Deep Neural Networks on Microcontrollers". In: *Sensors (Basel, Switzerland)* 21 (Apr. 2021). DOI: [10.3390/s21092984](https://doi.org/10.3390/s21092984).

Appendices

Appendix A Direct Training from OMNeT++ simulations

It was found that directly training from OMNeT++ was too slow; however, it may still be valuable for fine-tuning the training process.

In the setup, the environment is OMNeT++, with the mobile gateway acting as the agent. The reinforcement learning process takes place in a Python script, which is also responsible for creating episodes by setting up the OMNeT++ environment and executing simulations. OMNeT++ and Python communicate through shared data, such as logging states from OMNeT++ after each episode, which the Python script processes.

A visualization of how the different components interact is shown in Figure A.1.

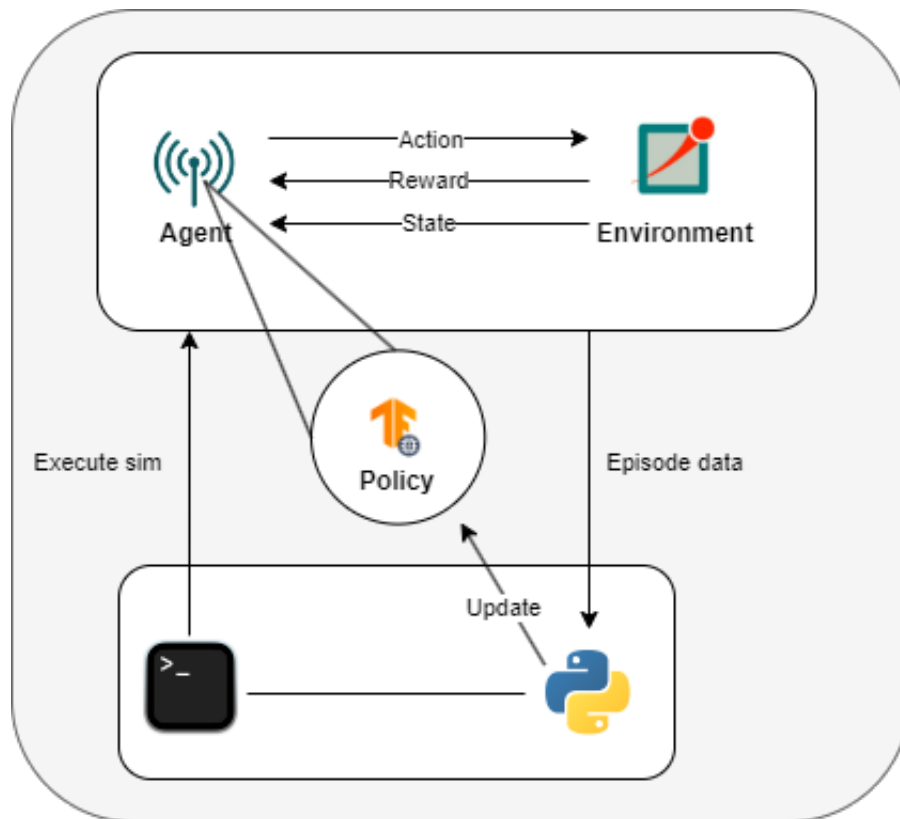


Figure A.1: System flow for training in OMNeT++ directly

Appendix B OMNeT++ custom component diagram

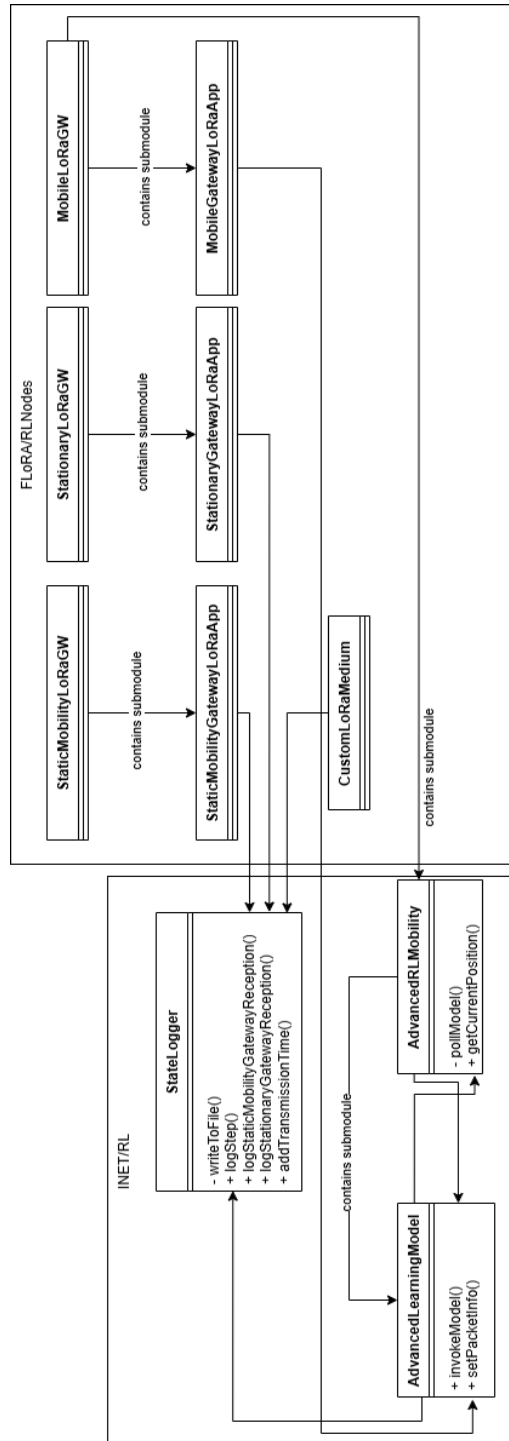


Figure B.1: Diagram of the relation between custom components in OMNeT++

Appendix C Scenario 2 - Gateway Distances to Nodes

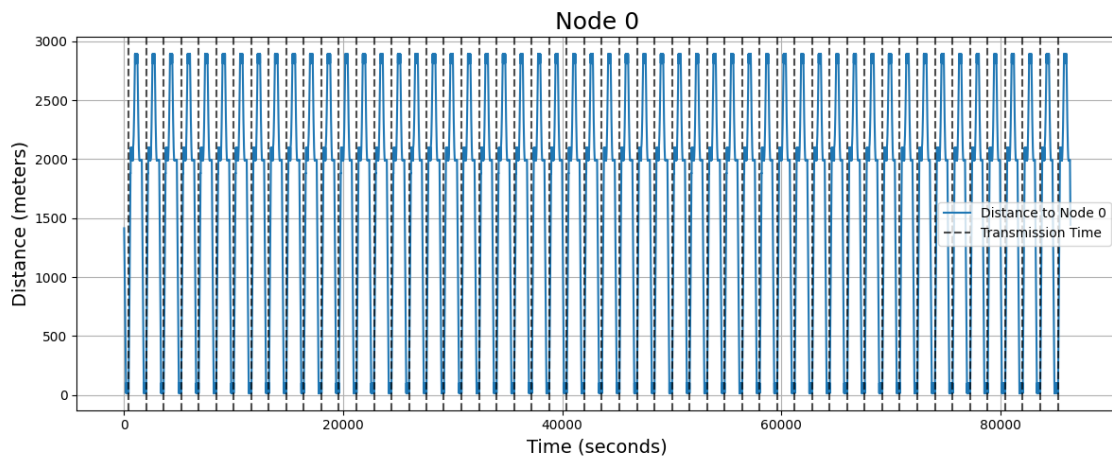


Figure C.1: S2 Gateways distance to node 0 over steps

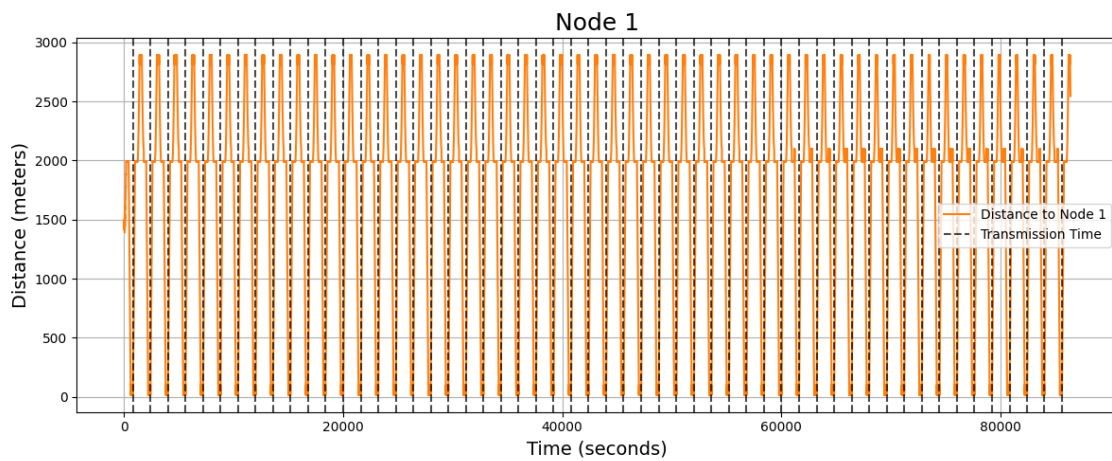


Figure C.2: S2 Gateways distance to node 1 over steps

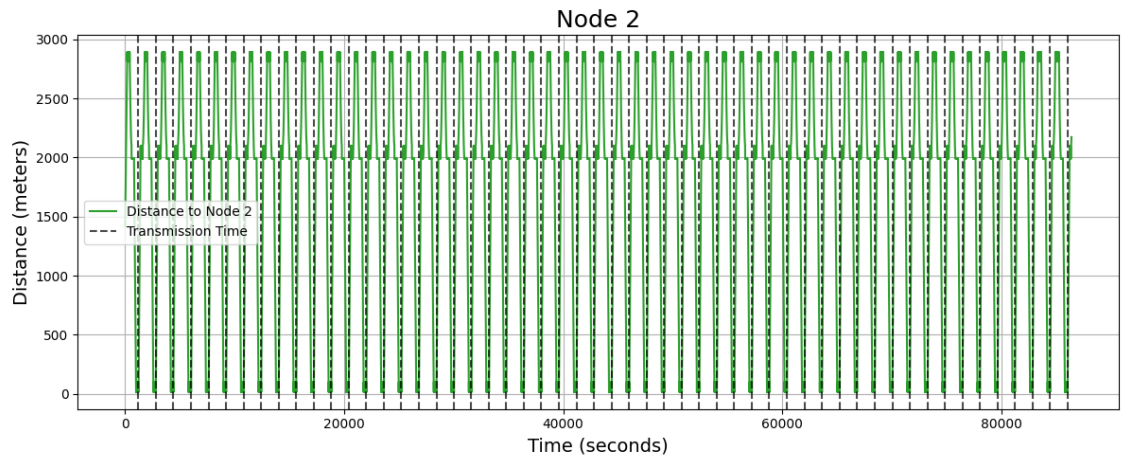


Figure C.3: S2 Gateways distance to node 2 over steps

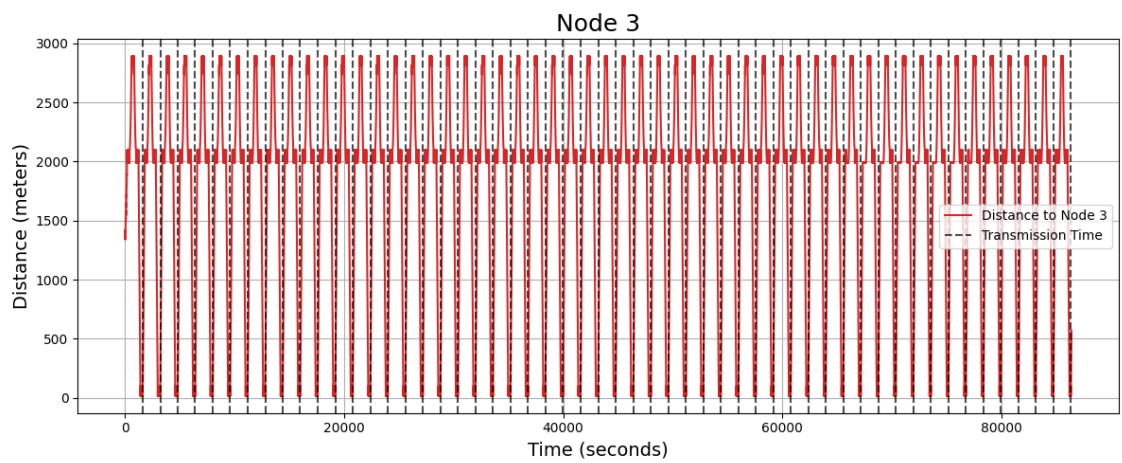


Figure C.4: S2 Gateways distance to node 3 over steps

Appendix D Scenario 3.A - Gateway Distances to Nodes

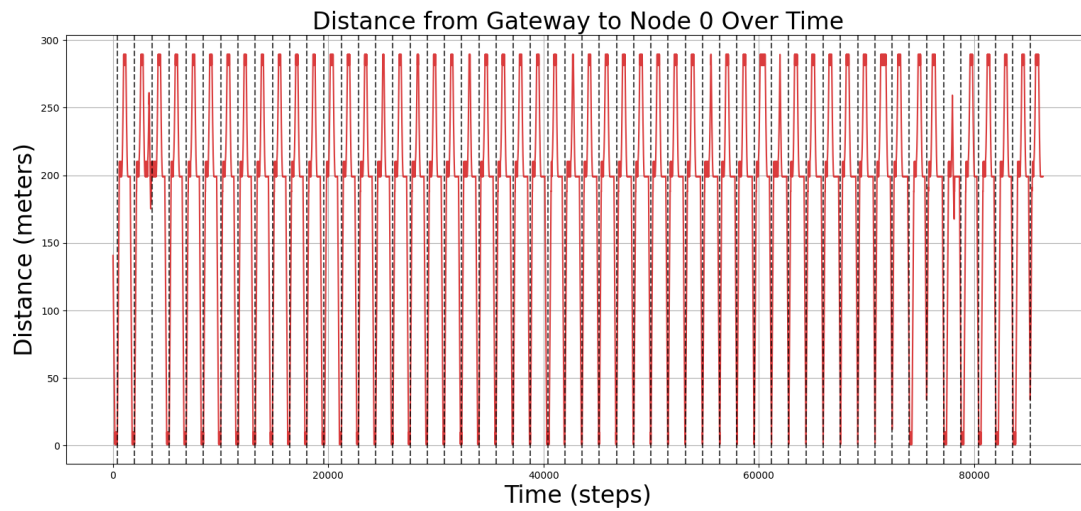


Figure D.1: S3.A Gateways distance to node 0 over steps

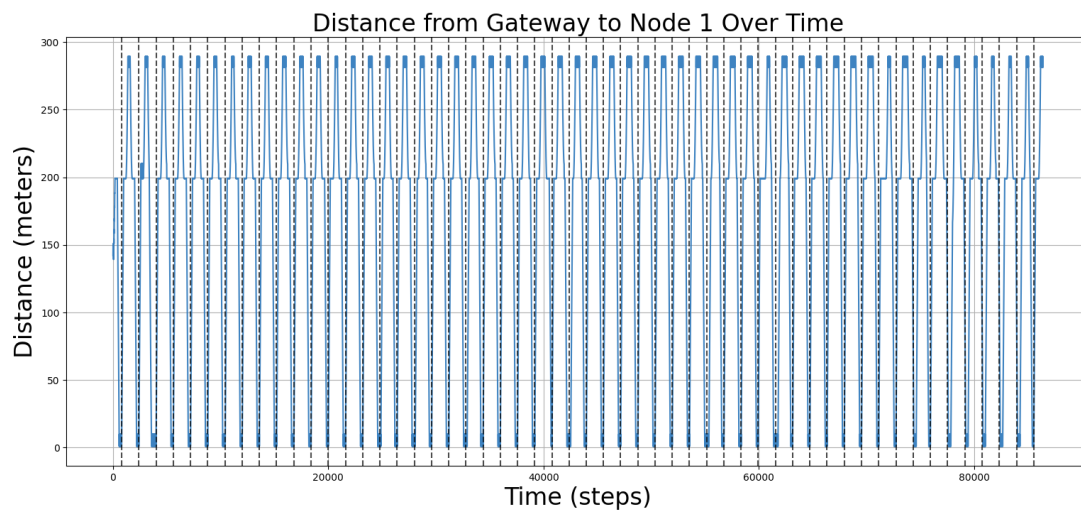


Figure D.2: S3.A Gateways distance to node 1 over steps

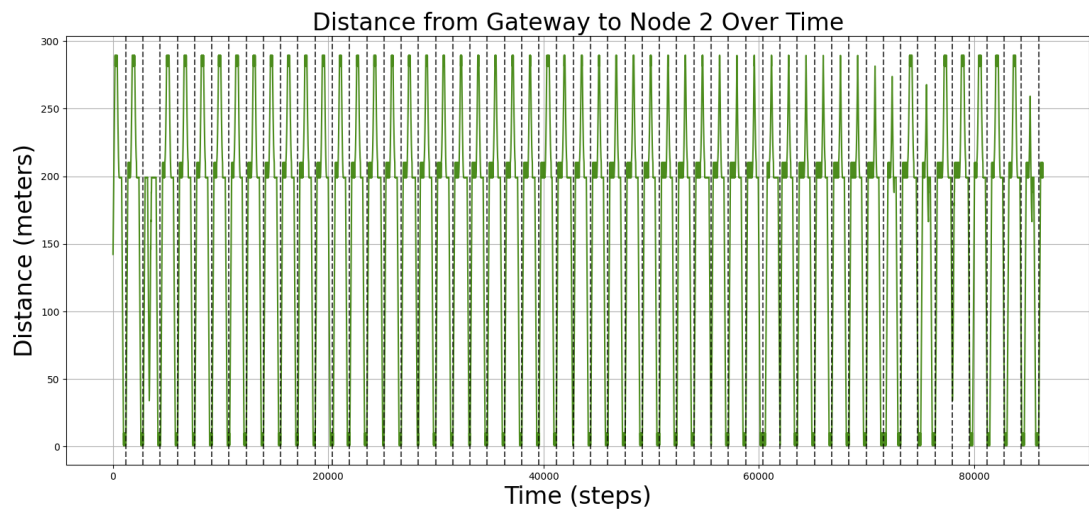


Figure D.3: S3.A Gateways distance to node 2 over steps

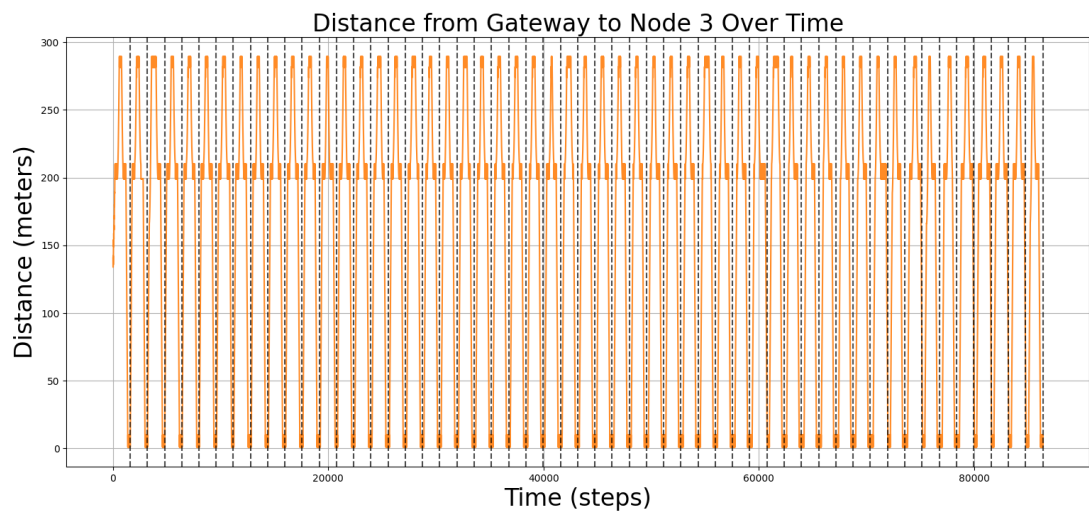


Figure D.4: S3.A Gateways distance to node 3 over steps

Appendix E Validation Model Conversion Script

```
1 def test_sb3_tf_model_conversion(sb3_model, tf_model: TFPolicy):
2     tolerance = {"abs": 2e-6, "rel": 2e-5}
3
4     sb3_input_dim = sb3_model.observation_space.shape[0]
5     tf_input_dim = tf_model.input_dim
6
7     assert sb3_input_dim == tf_input_dim, f"Input dimensions must match. {
8     sb3_input_dim = } || {tf_input_dim = } "
9     abs_diffs = []
10    rel_diffs = []
11    for _ in range(1000):
12        random_input = np.random.random((1, sb3_input_dim)).astype(np.float32)
13        sb3_output = sb3_get_action_probabilities(random_input.flatten(),
14        sb3_model).flatten()
15        tf_output = tf_model.call(tf.convert_to_tensor(random_input)).numpy().
16        flatten()
17
18        abs_diff = np.abs(sb3_output - tf_output)
19        rel_diff = np.abs(abs_diff / (np.abs(sb3_output) + 1e-8)) # Avoid
20        division by zero
21
22        abs_diffs.append(abs_diff)
23        rel_diffs.append(rel_diff)
24
25    if not np.allclose(sb3_output, tf_output, atol=tolerance["abs"], rtol=
26    tolerance["rel"]):
27        print(f"Mismatch detected!\nSB3: {sb3_output}\nTF: {tf_output}")
28    mean_abs_diff = np.mean(abs_diffs)
29    mean_rel_diff = np.mean(rel_diffs)
30    print(f"Mean Absolute Difference: {mean_abs_diff:.6f}")
31    print(f"Mean Relative Difference: {mean_rel_diff:.6f}")
32    print("Completed test")
```

Listing Appendix E.1: model conversion test

Technical
University of
Denmark

Richard Petersens Plads, Building 324
2800 Kgs. Lyngby
Tlf. 4525 1700