FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Using Deep Reinforcement Learning Techniques to Optimize the Throughput of Wi-Fi Links

Héber Miguel Severino Ribeiro

Mestrado em Engenharia Eletrotécnica e de Computadores

Supervisor: Rui Lopes Campos

July 1, 2022

Abstract

Ever since the introduction of the first IEEE 802.11 standard in 1997 (commercially known as Wi-Fi), there have been considerable advances in the development of this technology. Every new standard released has brought a myriad of improvements and new configurable parameters. These parameters make it possible to get an increasingly better link quality in Wi-Fi connections. However, the optimal configuration of these parameters becomes a complex task and the existing link adaptation algorithms are no longer suitable, opening up the opportunity to use Machine Learning techniques to address this challenge.

Over the last two decades, the Machine Learning field has seen considerable advances, mainly due to the increase in computational power. This has made it possible for new advanced techniques to be developed and used in a set of scientific and engineering areas. One powerful technique that has been on the rise in recent years is Reinforcement Learning (RL), which consists in training an agent that learns how to perform the best actions by observing and interacting with the environment it is designed for.

This dissertation had the goal of developing a link adaptation algorithm that dynamically adapts the Modulation and Coding Scheme (MCS), Channel Bandwidth (CB) and Guard Interval (GI) parameters, according to the current radio channel conditions, and is resilient to sudden changes in the link quality, in order to maximize the throughput. The algorithm was developed using Deep Reinforcement Learning (DRL), a fusion of Deep Learning (DL) and RL that allows for the use of RL with Deep Neural Networks (DNNs). It was then trained, tested and validated using the ns-3 software. The obtained simulation results show the proposed algorithm has significant gains when compared with its state of the art counterparts.

Resumo

Desde a introdução da primeira norma IEEE 802.11 em 1997 (comercialmente conhecida como Wi-Fi), foram feitos avanços significativos no desenvolvimento desta tecnologia. Todas as normas publicadas desde então trouxeram uma grande variedade de melhoramentos e novos parâmetros configuráveis. Estes parâmetros permitem obter uma qualidade de ligação cada vez melhor em ligações Wi-Fi. No entanto, a configuração ótima destes parâmetros torna-se uma tarefa complexa e os algoritmos de débito adaptativo existentes não são apropriados, abrindo caminho para a utilização de técnicas de *Machine Learning* para endereçar este desafio.

Ao longo das duas última décadas, a área de *Machine Learning* tem visto avanços consideráveis, principalmente devido ao aumento do poder computacional. Isto possibilitou que novas técnicas avançadas fossem desenvolvidas e utilizadas numa grande variedade de áreas científicas e de engenharia. Uma técnica poderosa que tem sido popularizada nos últimos anos é *Reinforcement Learning* (RL), que consiste em treinar um agente que aprende como executar as melhores ações através da observação e interação com o ambiente para o qual foi desenhado.

Esta dissertação tem como objetivo desenvolver um algoritmo de adaptação de ligação capaz de adaptar dinamicamente os parâmetros *Modulation and Coding Scheme* (MCS), *Channel Bandwidth* (CB) e *Guard Interval* (GI), de acordo com as condições atuais do canal rádio, e ser resiliente a mudanças bruscas na qualidade da conexão, de forma a maximizar o *throughput* obtido. O algoritmo foi desenvolvido utilizando *Deep Reinforcement Learning* (DRL), uma fusão de *Deep Learning* (DL) e RL que permite que RL seja utilizado juntamente com *Deep Neural Networks* (DNNs). O algoritmo foi posteriormente treinado, testado e validado utilizando o simulador de redes ns-3. Os resultados de simulação obtidos mostram que o algoritmo proposto tem ganhos significativos comparado a outros algoritmos do estado da arte.

Acknowledgements

First of all, I would like to thank my supervisors, Rui Lopes Campos, Rúben Miguel Rei Queirós and Helder Martins Fontes for all the support they have given me during the development of this dissertation. This work would not be possible without them.

I would like to thank my friends for always being there for me, both during the good and the less pleasant times. The laughs and moments we shared daily were precious and helped me to keep going.

Last but not least, I would like to thank my family for always supporting me and allowing me to become the person I am today. They mean the world to me and I owe them my life.

Héber Ribeiro

Contents

List of Figures

List of Tables

Abbreviations

ACK	Acknowledgement
AI	Artificial Intelligence
A-MPDU	Aggregate Medium Access Control Protocol Data Unit
AP	Access Point
BER	Bit Error Ratio
BSS	Basic Set Service
CB	Channel Bandwidth
CDF	Cumulative Distribution Function
DL	Deep Learning
DNN	Deep Neural Network
DQN	Deep Q-Network
DRL	Deep Reinforcement Learning
DRL-LA	Deep Reinforcement Learning Link Adaptation
DSSS	Direct Sequence Spread Spectrum
EDRA	Experience Driven Rate Adaptation
EWMA	Exponential Weighted Moving Average
FER	Frame Error Ratio
FHSS	Frequency Hopping Spread Spectrum
FLR	Frame Loss Ratio
FSR	Frame Success Ratio
GI	Guard Interval
HR-DSSS	High-Rate Direct Sequence Spread Spectrum
LFA	Level of Frame Aggregation
MAC	Medium Access Control
MCS	Modulation and Coding Schemes
MIMO	Multiple Input Multiple Output
ML	Machine Learning
NIC	Network Interface Controller
NSS	Number of Spatial Streams
OFDMA	Orthogonal Frequency-Division Multiple Access
OFDM	Orthogonal Frequency-Division Multiplexing
PB	Preamble Puncturing
PHY	Physical
PLR	Packet Loss Ratio
ReLU	Rectified Linear Unit
RL	Reinforcement Learning
RSSI	Received Signal Strength Indicator
SISO	Single Input Single Output

SL	Supervised Learning
SNR	Signal-to-Noise Ratio
SR	Spatial Reuse
STA	Station
STR	Service Time Ratio
UL	Unsupervised Learning
WLAN	Wireless Local Area Networks

Chapter 1

Introduction

1.1 Context

The usage of Wireless Local Area Networks (WLANs) is one of the most common practices today. These networks have been popularized in domestic, educational, commercial and corporative environments, allowing for multiple devices to communicate with each other and also access the Internet. They are implemented using the IEEE 802.11 set of technical standards, commercially known by the brand name Wi-Fi. The first IEEE 802.11 standard was released back in 1997 and since then, multiple standards have been introduced. By the time of writing, the latest standards released are the IEEE 802.11n (2009), IEEE 802.11ac (2013) and IEEE 802.11ax (2020). These standards were developed to meet the demand for high throughputs and network capacity. With each subsequent standard, new configurable parameters were introduced, both in the Phyiscal (PHY) and in the Medium Access Control (MAC) layers, aimed at improving the performance of connections inside a WLAN. These parameters need to be optimally configured depending on the current radio channel conditions, making it paramount to have a link adaptation algorithm capable of accomplishing this.

Over the years, various link adaptation algorithms have been developed to address the challenge presented above [1]. However, with the introduction of the latest IEEE 802.11 standards and their enhancements, these algorithms have shown to be outdated and unable to optimally adapt the link, as they do not fully utilize the capabilities of the most recent standards. This creates a demand for better solutions when it comes to link adaptation algorithms.

1.2 Motivation

Machine Learning (ML) is currently a very prominent field, as it is being actively researched and developed. This field spans a variety of techniques that are being increasingly applied to many scientific areas, including Wireless Communications.

Due to the constant advances in the ML field, the efficiency and versatility that ML techniques provide, and the promising results obtained in recent related works, it seems that the way forward

when it comes to developing advanced link adaptation algorithms, capable of fully utilizing the features of the most recent IEEE 802.11 standards, should be based on the use of ML techniques.

Recent works have shown that ML techniques can be successfully used in the creation of link adaptation algorithms. More specifically, the Reinforcement Learning (RL) technique has been utilized to developed dynamic link adaptation algorithms for the IEEE 802.11 standards in [2] and [3], which attempt to optimally configure various parameters based on the current radio channel conditions. These works present very promising results and have laid the foundation when it comes to applying RL for dynamic link adaptation. However, to the best of our knowledge, there are still few studies in this subject and there is much room for improvement, pointing to the potential of this dissertation.

1.3 Problem Definition

Recent IEEE 802.11 standards, like the IEEE 802.11n, IEEE 802.11ac and IEEE 802.11ax, allow for the creation of high throughput WLANs due to their many enhancements. However, the most popular rate and link adaptation algorithms are no longer suitable, as they do not explore the full potential of these standards. These algorithms can also present a myriad of problems like excessive overhead, increased delay and heavy processing. Furthermore, they fail to quickly adapt to unpredictable channel quality changes.

To reach the theoretical high throughputs that these standards offer, while also ensuring a quick adaptation when sudden radio channel quality changes occur, there is a need of novel link adaptation algorithms that optimally configure various parameters depending on the radio channel conditions. This dissertation addresses this problem by utilizing Deep Reinforcement Learning (DRL) as a basis for a new algorithm.

1.4 Objectives

This dissertation had the goal of developing and validating a new DRL-based Link Adaptation (DRL-LA) algorithm for Wi-Fi networks. This algorithm should be able to configure three parameters to adapt the link: Modulation and Coding Schemes (MCS), Guard Interval (GI) and Channel Bandwidth (CB). Though its development is focused on the IEEE 802.11ac standard, it can be easily adjusted to be compatible with previous standards. Furthermore, the algorithm should have a backup mechanism that is activated whenever the configurations chosen by the DRL-LA algorithm fail. To achieve this, the following specific objectives were defined:

- **Solution design** - Design and modelling of the DRL-based link adaptation algorithm;
- **Solution implementation** - Development of the various programs necessary to implement the DRL-LA algorithm. This includes the program that constructs the simulations, the program that implements the DRL agent and the program that interconnects those two components;

- **Performance Evaluation** - A wireless link adaptation algorithm must function correctly in different scenarios by adequately configuring the different parameters per the current radio channel conditions. The developed algorithm shall be tested in various scenarios in order to validate its performance and compare it to other competing algorithms.

1.5 Contributions

The main contribution of this dissertation is a novel link adaptation algorithm – the DRL-LA algorithm – focused on, but not limited to, the IEEE 802.11ac standard. The DRL-LA algorithm is capable of optimally configuring a set of parameters according to the radio channel conditions observed in real-time, and it is resilient to sudden radio channel condition variations. With the successful development of this algorithm, we show that ML-based, and more specifically, DRL-based approaches have a lot of potential for further research concerning the link adaptation problem in Wi-Fi networks.

1.6 Document Structure

The rest of this document is structured in five chapters. Chapter 2 focuses on the state of the art and the related work. Chapter 3 details the proposed solution to address the problem, explaining the plan designed for the implementation and the tools and technologies used to achieve the goal. Chapter 4 describes the actual implementation of the solution. Chapter 5 describes the network simulations designed for validation and the obtained results. Chapter 6 presents the main conclusions drawn from this dissertation and points out the future work on this subject.

Chapter 2

State of the Art

This chapter covers the study and analysis on the state of the art. It exposes the technologies and solutions related to the main topics of this dissertation, such as the IEEE 802.11 set of standards, ML and existing link adaptation algorithms. It is divided into four sections:

- IEEE 802.11 - A brief introduction to the IEEE 802.11 set of technical standards and the enhancements brought with each release;
- Link Adaptation Algorithms - An overview of existing link adaptation algorithms for the IEEE 802.11 standards;
- Machine Learning - A definition of Machine Learning, its three main paradigms, Deep Learning, Deep Reinforcement Learning and key concepts;
- Deep Reinforcement Learning Algorithms - A short survey on existing Deep Reinforcement Learning algorithms;
- Related Work - An overview of existing solutions for wireless link adaptation algorithms using RL.

2.1 IEEE 802.11

The IEEE 802.11 set of technical standards [4], commercially known by the brand name Wi-Fi, are used for implementing WLANs. Nowadays, the usage of WLANs based on these standards is widespread in domestic, educational, commercial, and corporative environments and has become a fundamental part of how people form local networks and access the Internet. The first IEEE 802.11 standard, known as legacy, was released in 1997. Since then, multiple consequent standards have been introduced, each bringing new configurable parameters, both in the PHY and MAC layers, aimed at improving the performance of connections inside a WLAN. Although dozens of 802.11 standards have been released over the years, the most relevant and groundbreaking are the following:

- **IEEE 802.11 (Legacy)** - The original IEEE 802.11 standard was released in 1997 and provided two raw data rates of 1 and 2 Mbps using three non-overlapping channels in the 2.4 GHz frequency band. It either used Frequency Hopping Spread Spectrum (FHSS) or Direct Sequence Spread Spectrum (DSSS);
- **IEEE 802.11a** - Released in 1999, two years after the original IEEE 802.11, it introduced new data rates with varying modulation types using the 5 GHz frequency band, going up to 54 Mbps. It also used Orthogonal Frequency-Division Multiplexing (OFDM) with 52 subcarrier channels;
- **IEEE 802.11b** - Also released in 1999, it introduced data rates with varying modulation types using the 2.4 GHz frequency band, managing to go as high as 11 Mbps. It used High-Rate Direct Sequence Spread Spectrum (HR-DSSS);
- **IEEE 802.11g** - This standard, released in 2003, introduced even more data rates with varying modulation types using the 2.4 GHz frequency band. The highest data rate is 54 Mbps, a considerable increase over the 11 Mbps provided by the 802.11b. Like IEEE 802.11a, it used OFDM;
- **IEEE 802.11n** - With this standard, connections became even faster and more reliable. Released in 2009, it added Multiple Input Multiple Output (MIMO), 40 MHz channels, and modulation types up to 64-QAM. This standard also allows up to 4 spatial streams with a maximum theoretical throughput of 600 Mbps;
- **IEEE 802.11ac** - Initially released in 2013, this standard managed to provide gigabit speeds by extending the IEEE 802.11n concepts. It included a wider bandwidth, going all the way to 160 MHz, up to 8 spatial streams, downlink multi-user MIMO, and a high-density modulation (256-QAM). However, this standard works exclusively in the 5 GHz band, which means dual-band access points and clients continue to use 802.11n at 2.4 GHz;
- **IEEE 802.11ax** - The sixth generation of Wi-Fi, released in 2020, supports both the 2.4 and 5 GHz bands and allows modulations up to 1024-QAM, has reduced subcarrier spacing, and used scheduled based resource allocation. It uses Orthogonal Frequency Division Multiple Access (OFDMA) and is capable of coexisting efficiently with 802.11a/g/n/ac clients, as it was designed for maximum compatibility.

A typical WLAN based on the IEEE 802.11 standards can function in two modes: infrastructure and ad-hoc. A WLAN operating in infrastructure mode has one or more Access Points (APs), and all the devices present in the network – smartphones, computers, etc – communicate with each other through an AP. These devices are also known as Stations (STAs). This means that even if STAs are physically next to each other, they do not communicate directly between them. Instead, they send their packets to the AP, which redirects them to their destination. An example of a WLAN operating in infrastructure mode is a domestic network, in which a router interconnects

the STAs inside the house and also provides access to the Internet. When considering an ad-hoc mode WLAN, the STAs communicate directly with each other without the need for an AP. One example of this is setting up two computers in ad-hoc mode.

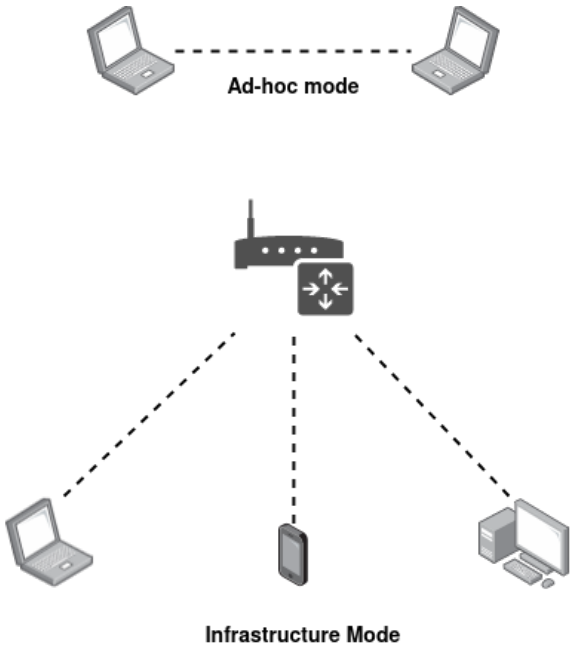

Figure 2.1: Comparison between the ad-hoc and infrastructure modes.

2.2 Link Adaptation Algorithms

A wireless network is a complex system with many variables. As such, the link quality can suffer abrupt or prolonged variations, affirming the necessity for link adaptation algorithms. An algorithm of this kind may only focus on optimally configuring the MCS parameter per the current radio channel conditions, also known as a rate adaptation algorithm. Other more complex algorithms may attempt to configure more parameters than just the MCS, such as the Number of Spatial Streams (NSS), GIs, and CB.

In general, a traditional link adaptation algorithm works in the following way: when the radio channel presents good conditions, and the signal is strong, the algorithm selects a high MCS value, which allows for a higher throughput. When the channel presents poor conditions, and the signal is weak, the algorithm selects a small MCS value, which lowers throughput, but ensures that packets are delivered. The aforementioned radio channel conditions depend on various factors like signal interference, node mobility and channel fading. Currently, there are many link adaptation algorithms available that use unique approaches to try to achieve the high throughputs supported by the latest IEEE 802.11 standards. The authors in [1] made a survey and evaluation on multiple adaptation algorithms. Iwl-mvm-rs [5] and Minstrel-HT [6] are probably the current most widely used adaptation algorithms, but there are other interesting and recent options like STRALE [7] and Damysus [8]. These four algorithms are further analyzed in the following subsections, as

Iwl-mvm-rs and Minstrel-HT are among the most popular options, while STRALE and Damysus are representative of two unique and recent approaches to the problem.

2.2.1 Iwl-mvm-rs

Iwlwifi is the default driver for Intel's wireless Network Interface Controllers (NICs) [5] and comes with the Iwl-mvm-rs link adaptation algorithm. This algorithm is compatible with standards up to IEEE 802.11ac and the most recent Intel NICs have versions that already support IEEE 802.11ax. The Iwl-mvm-rs algorithm decides whether to transmit in legacy mode (IEEE 802.11a or IEEE 802.11g) or non-legacy mode (IEEE 802.11n or IEEE 802.11ac). It also decides whether the transmission is made in Single Input Single Output (SISO) or MIMO mode and configures multiple parameters, such as the MCS, GI, and frame aggregation. It achieves this by having cycles that alternate between two phases: the MCS scaling phase and the Column scaling phase.

A cycle starts with the MCS scaling phase. During this phase, only the MCS value is changed in an attempt to maximize the throughput. It measures the throughput obtained by the current MCS value by multiplying the success ratio of up to the last 62 frame transmissions with the theoretical throughput that this MCS value can obtain and makes the following decision:

- If the success ratio is small or the measured throughput is zero, it decreases the MCS value;
- If the measured throughput with the higher adjacent MCS value is better, or unknown, increase the MCS value;
- If the measured throughput with the current MCS is higher than the throughput obtained with both adjacent MCS values (lower and higher), maintain the MCS value.

If the MCS value does not change, the MCS scaling phase ends, and the Column scaling phase begins. This phase tries to find a better column, which is a combination of the mode (SISO or MIMO), GI, and antenna configuration parameters. The algorithm has a table with all the different columns and theoretical throughputs associated with each of them and will try to use the columns that present a theoretical throughput higher than the currently measured throughput. If an attempted column achieves a higher throughput than the one previously measured, the algorithm keeps using that column; otherwise, it is avoided in future cycles, and the algorithm reverts to the column it was using before.

By doing multiple cycles composed of the two phases described, the algorithm converges towards the best combinations of the parameters mentioned above. However, this algorithm has some limitations, like not having a joint rate and bandwidth adaptation, lack of scalability, and no online learning capability [3].

2.2.2 Minstrel-HT

Minstrel-HT is the default link adaptation algorithm in the Linux kernel for IEEE 802.11n and IEEE 802.11ac devices. It is the evolution of the popular Minstrel algorithm and is currently

one of the most used algorithms, being adopted by millions of devices. It first creates groups of rates based on the CB, GI and NSS parameters. Each of these groups has a set of eight different data rates that are represented by a MCS value. The groups that are created depend on what the transmitter device supports. For example, if the transmitter only supports 1 stream, regular GI and a CB of 20 MHz, it will only have available one single group that contains values for the MCS that range from 0 to 7. But if the transmitter supports 2 spatial streams, short GI and a CB of 20 MHz, it will have two groups available: the first group will have MCS values that range from 0 to 7 and the second group will have values that range from 8 to 15. In total, there can be up to 16 groups if the transmitter allows for it.

Like iwl-mvm.rs, this algorithm also functions in cycles of two phases: the sampling phase and the non-sampling phase. A cycle starts with the sampling phase, where a random data rate is picked from the available groups created. If this data rate achieves a higher throughput than the previously selected data rate, then it will be used for subsequent transmissions. If not, then the previous data rate is kept. The throughput measured is calculated using the Frame Loss Ratio (FLR), while also considering the Exponential Weighted Moving Average (EWMA). The sampling phase is an iterative process in which the algorithm tries multipe data rates, and by the end of this phase, it selects three: the one that obtained the best throughput, the one that obtained the second best throughput and the one that has the best probability of providing an acceptable throughput in case the first two fail.

The second phase, called the non-sampling phase, consists in making transmissions using the best data rate picked previously. If packets begin to be lost and the maximum number of retransmissions is reached, the algorithm switches to the second best data rate. Likewise, if this data rate leads to considerable packet loss, the best probability rate is finally used.

Even though this is currently one of the most used link adaptation algorithms, it presents various problems and limitations. It consistently fails to achieve optimal throughput values, especially in situations when the radio channel conditions suddenly change [9]. It also does not come close to utilizing the capabilities of recent IEEE 802.11 standards, like IEEE 802.11n and IEEE 802.11ac, as it only focuses on configuring the MCS value.

2.2.3 STRALE

STRALE is an algorithm that configures both a PHY parameter and a MAC parameter, the former being the MCS value and the latter being the Aggregate Medium Access Control Protocol Data Unit (A-MPDU) length. It tries to avoid decreasing the MCS value, preferring to reconfigure the A-MPDU length instead. At first, a transmission is made with some MCS value and A-MPDU length, and after receiving a block Acknowledgement (ACK), the algorithm then calculates the optimal A-MPDU length that would have obtained the highest throughput in the previous transmission. After that, it also calculates the A-MPDU length for the next transmissions using EWMA. The two values calculated are then subtracted and the result is compared to a threshold. If it is greater than the threshold, the algorithm has to decide if decreasing the MCS value while using the newly calculated A-MPDU length would be beneficial or not, avoiding unnecessary decreases.

Even though this algorithm presents promising results, its main drawback is that it does not consider the effects of interference.

2.2.4 Damysus

Damysus is a recent link adaptation algorithm focused on the IEEE 802.11ax standard. It attempts to optimally configure the MCS value, while also being resilient to the deterioration of the radio channel conditions and improving the network performance in the presence of Basic Set Service (BSS) Color and Preamble Puncturing (PB). It manages to select a new MCS value without introducing overhead and exploits the Spatial Reuse (SR) mechanism of IEEE 802.11ax to improve the throughput in dense networks. Damysus makes a statistical study during intervals of 100 miliseconds and during cycles of 1 second where it records the success and failure of packet transmissions to calculate the Packet Loss Ratio (PLR) and compares it against a packet loss threshold. It then decides how to make the parameter configurations based on that comparison.

The main drawback of this algorithm is that it depends on PLR thresholds, as they are hard to define and there is no single threshold that guarantees the achievement of the maximum throughput possible. Also, it is only compatible wth the IEEE 802.11ax standard.

2.3 Machine Learning

Machine Learning [10] is a branch of Artificial Intelligence (AI) that allows computer systems to learn and improve from experience without being explicitly programmed. It has seen a very rapid evolution in recent years due to the significant improvements in hardware technology. As such, it is currently a very researched topic, and it is being applied to a great variety of scientific fields. Due to its elevated complexity, it was divided into three main paradigms: supervised learning, unsupervised learning and reinforcement learning. These three paradigms will be further detailed in the subsections below, along with a brief exposition to Deep Learning and Deep Reinforcement Learning.

2.3.1 Supervised Learning

Supervised Learning (SL) [11] consists in having an algorithm that learns by using data sets where all the data is labeled. Labeled data basically means that we know what that data represents, like, for example, having a picture of an animal and knowing what kind of animal it is. In other words, with labeled data, we know what the outputs are for sets of inputs. The data is labeled by humans, hence why this paradigm is called "supervised".

With this paradigm, an algorithm is trained with a labeled dataset, called a training set. This training is done until the algorithm is able to detect the underlying patterns and relationships between the input data and the output labels. After completing the training phase, the algorithm should then be able to receive unlabeled data as an input and correctly label it.

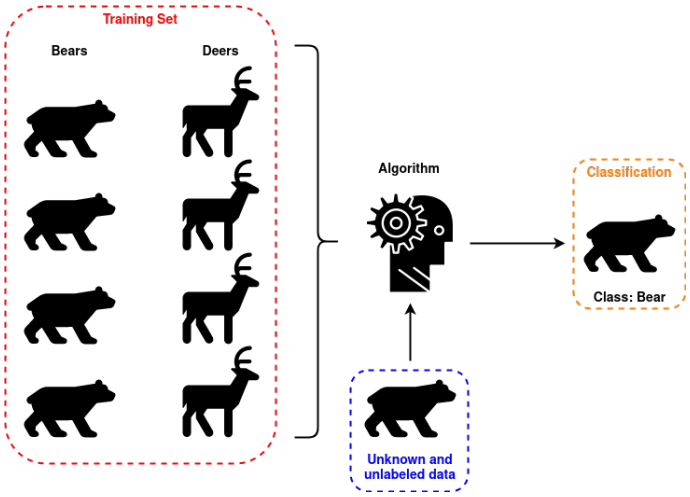

Figure 2.2: The Supervised Learning process.

2.3.2 Unsupervised Learning

Unlike SL, the Unsupervised Learning (UL) [12] paradigm consists in using an algorithm that analyzes and clusters unlabeled data sets. It works by discovering patterns in the data without human intervention, hence why it is called "unsupervised".

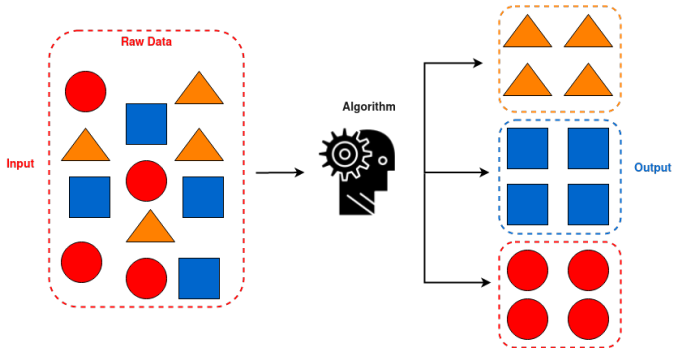

Figure 2.3: The Unsupervised Learning process.

2.3.3 Reinforcement Learning

Inspired by animal psychology, RL [13] consists in using an agent that is capable of learning, in an interactive environment, by trial and error. It accomplishes this by having a reward system that rewards the agent for taking suitable actions. Some rewards are higher than others, depending on the outcome of actions. Over time, the agent associates which actions give it a higher reward and which actions punish it, opting to choose the former.

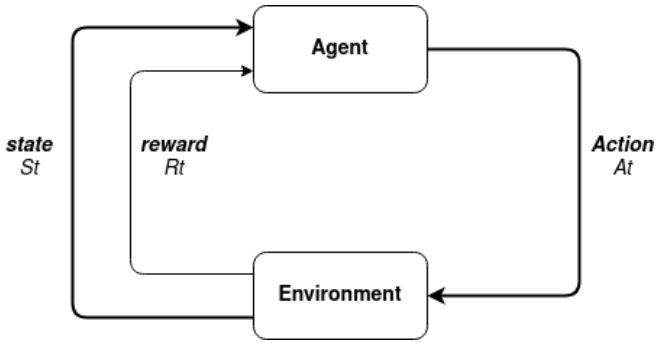

Figure 2.4: The Reinforcement Learning process.

Figure 2.4 represents the RL process, in which we have an agent and an environment. The agent interacts with the environment through actions. The environment returns to the agent a state and a reward associated with the previous action the agent took.

RL is similar to UL because it does not require labeled data to learn. However, their end goals are ultimately different, as UL strives to find similarities and patterns in data. In contrast, RL tries to find a suitable action model that maximizes the agent's cumulative reward.

2.3.4 Deep Learning

Deep Learning (DL) [14] is a branch of ML that uses multi-layered artificial neural networks to learn representations from complex and abstract data such as text, images, sound, or video. These multi-layered artificial neural networks are also known as Deep Neural Networks (DNNs) and they are characterized by having multiple layers between the input and output layers, called hidden layers.

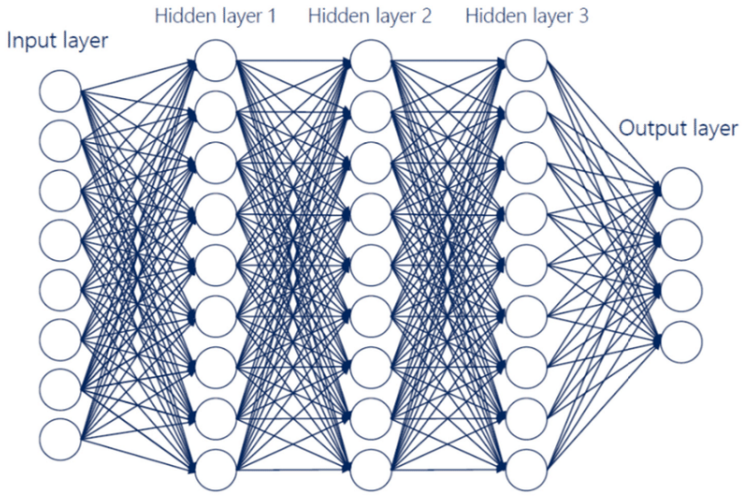

Figure 2.5: The Deep Learning process [15].

One of the major problems of DNNs is their explainability. Although they provide great value to the performance of a model, it is hard or even impossible to explain exactly how the model gets

the answer. The hidden layers are essentially a black box. Nonetheless, DL has revolutionized the field of ML, as it can be used to solve problems that would otherwise be unsolvable with just traditional ML techniques.

2.3.5 Deep Reinforcement Learning

DRL [16] is essentially the combination of DL with RL. DRL algorithms incorporate the process behind RL with DNNs and allow us to solve problems that are too complex for traditional RL algorithms. It is a relatively recent technique that is having a rise in popularity and is being actively researched and developed. It is currently being used in various areas, like for example:

- **Automotive** - The development of autonomous and self-driving vehicles is becoming very common;
- **Manufacturing** - The use of intelligent robots for various tasks, like sorting out products and helping in assembly lines;
- **Healthcare** - The use of algorithms that are capable of making a diagnosis and determining treatment plans;
- **Finances** - The use of algorithms that evaluate trading strategies and manage investments.

Recently, DRL algorithms have even been applied to the Wireless Communications field to tackle a wide variety of problems, like developing link adaptation algorithms for Wi-Fi networks.

2.3.6 Key Concepts

When it comes to ML, there are some important concepts that are transversal to all the paradigms previously mentioned: overfitting, underfitting, activation functions and hyperparameters. Overfitting happens when a ML model perfectly fits its training data set, or in the case of RL, its training environment. This means that, even though the model performs very well with the training data, it fails to generalize its learning to new, unseen data, negatively impacting its performance. Common solutions to overfitting are training with more data or stopping the training session when it is detected that the model is no longer learning, but is instead overfitting the training data and weakening its ability to generalize. Underfitting is exactly the opposite, as an underfitted model is unable to capture the relationship between the input and output variables. This means that an underfitted model will perform poorly on both the training data and on unseen data. A common solution to underfitting is to simply increase the duration of the training session, as short training sessions may not be enough for the agent to learn.

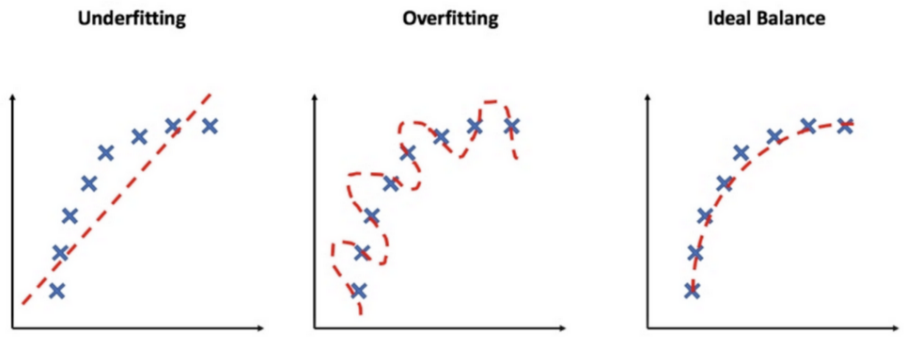

Figure 2.6: Illustration of underfitting, overfitting and ideal fitting. [17]

Activation functions are crucial components of neural networks that control which neurons are activated in the various layers. To achieve this, they use the weighted sum of inputs and biases from the network. They are useful because they add non-linearity into neural networks, allowing them to learn powerful operations. If there were no activation functions, the entire network would essentially be a linear operation and would no longer be capable of learning and performing complex tasks. Commonly used activation functions [18] are the Rectified Linear Unit (ReLU), tanh, sigmoid and Leaky ReLU.

Hyperparameters are variables whose values control the learning process and they are defined by the person creating the ML model. However, there are no exact best values for each hyperparameter, as that largely depends on the problem at hand. Determining the best value for a hyperparameter is usually done by trial and error, or in other words, by trying various values and observing which performs more efficiently. When it comes to reinforcement learning, the most commonly configured hyperparameters are:

- **Learning rate** - This hyperparameter, commonly denoted by the letter α, controls how quickly the model adapts to the problem. A smaller learning rate requires more training steps, as it makes smaller changes to the weights in the neural network, at each update. On the other hand, a larger learning rate results in more rapid changes, and therefore, requires less training steps. Picking a good learning rate is a precise task, as a value that is too large can cause the model to converge too quickly to a sub-optimal solution, whereas a value that is too small can cause the process to get stuck;

- **Replay buffer size** - The replay buffer, also known as memory, is a data structure where a RL agent stores its experiences. It enables the agent to memorize and reuse past experiences, similar to how humans do. When new experiences are added to this buffer and it becomes full, older experiences are forgotten. The size of the buffer is a hyperparameter, as small buffers are not capable of storing many experiences, whereas large buffers may occupy too much physical memory or introduce noise;

- **Batch size** - The batch size is the number of samples from the replay buffer that the agent uses at each time step. Since the replay buffer is quite large, it is not viable to use all the sample available in it. Therefore, the solution consists in only using some of the samples;
- **Discount factor** - This hyperparameter, commonly denoted by the letter γ, determines the agent's preference to obtain rewards sooner rather than later. It takes values between zero and one. If it is equal to zero, then the agent will only learn about actions that produce an immediate reward. If, on the other hand, it is equal to one, then the agent will evaluate each of the available actions based on the sum of all of its potential future rewards. This is a hyperparameter that is adjusted by taking into account if future rewards are more important than immediate rewards;
- **ε-greedy** - This hyperparameter controls how often the agent explores and exploits, or in other words, how often it takes random actions versus taking actions that it already learned. It takes a value between 0 and 1, and the higher it is, the more exploratory the agent behaves. In general, at the beginning of a training session, the agent starts off with a high ε-greedy value, which decays over time in order to make the agent exploit more and use the knowledge obtained.

2.4 Deep Reinforcement Learning Algorithms

Considering the solution proposed in this thesis consists in using DRL, it is important to present a brief survey on the most prominent DRL algorithms. These algorithms define computational procedures that are essentially strategies on how the agent interacts with an environment and obtains experience. They can be classified as model-based or model-free.

In a model-based algorithm, the agent has access to a model of the environment and can predict the reward for an action before actually performing it. In other words, the agent already has some knowledge of the environment and will always try to perform an action that obtains the maximum reward, no matter what consequence the action may cause. Good examples of model-based algorithms are the famous AlphaGo [19] and AlphaZero [20]. On the other hand, in a model-free algorithm, the agent does not have knowledge of the environment and needs to first perform the actions, learning from then and adjusting its policy to achieve optimal rewards. Furthermore, model-free algorithms can belong to two classes: value-based and policy-based. Value-based methods rely on a RL agent choosing the best action for each state, which means the agent must explore random actions and learn which ones return a better reward. Policy-based methods have the agent rely directly on a stochastic policy function to map states to actions.

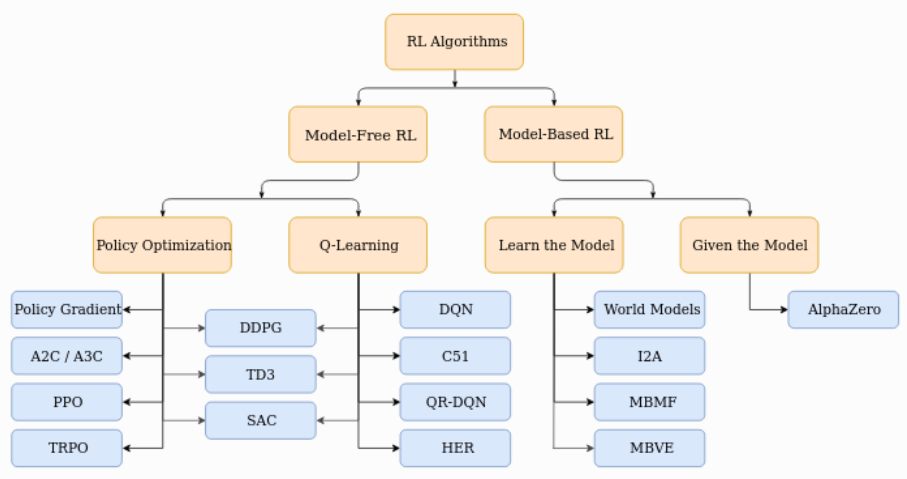

Figure 2.7: Types of reinforcement learning algorithms [21].

The authors in [22] proposed a value-based algorithm, named Deep Q-learning, that builds upon the ideas introduced by the classic Q-learning algorithm [23]. While Q-learning uses a Q-table to map state and action pairs to a Q-value, Deep Q-learning replaces that table with a deep neural network and maps input states to action and Q-value pairs.

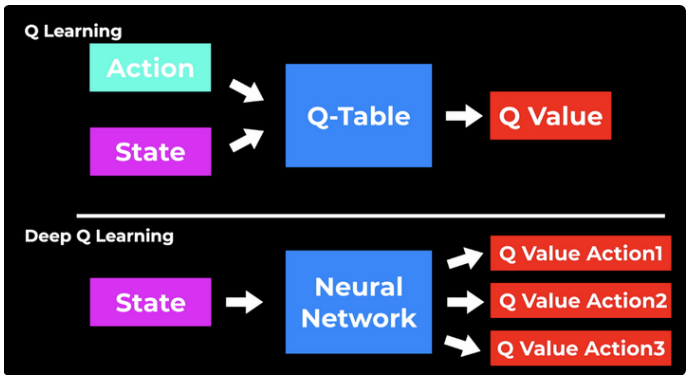

Figure 2.8: Q-learning vs. Deep-Q learning [24]

Through Deep Q-learning, a deep neural network gets a state as an input, and produces different Q-values for each action, choosing the action with the highest Q-value. Subsequently, the weights in the neural network are updated, achieving more efficient outputs. Unlike classic Q-learning, which can only be applied to simple problems with few states and actions, Deep Q-learning can be used in complex scenarios with dozen of possible states and actions, being very efficient for discrete action spaces.

The authors in [25] proposed an algorithm that combines both a value-based and a policy-based approach, named Soft Actor-Critic. It is based on the maximum entropy reinforcement learning framework, in which the actor attempts to maximize the reward while also maximizing the entropy. With this framework, increased entropy results in more exploration, accelerating the learning and preventing the policy from prematurely converging to a bad local optimum. Overall,

the expected cumulative long term reward is maximized and the entropy balances the exploitation and exploration of the environment. This algorithm uses continuous action spaces and has been efficiently applied in robotics.

C51 [26] is an algorithm based on Deep Q-learning, designed for discrete action spaces. The main difference between C51 and Deep Q-learning is that C51 predicts a histogram model for the probability distribution of the Q-value. The advantage of doing this is that, by learning the distribution rather than simply the expected value, the algorithm becomes more stable during training, leading to improved results. Although the algorithm has shown to be more efficient than Deep Q-learning when used appropriately, it needs to perform complex distributional computations and is more difficult to set up.

TRPO [27] is a policy-based algorithm that alternates between sampling data through environmental interaction and updating the policy parameters by solving a constrained optimization problem. By doing this, it prevents significant performance drops by keeping the updated policy within a trust region close to the current policy. It has shown to be more efficient than other popular policy-based algorithms and it works with both discrete and continuous action spaces. However, it is computationally expensive and loses efficiency if the observation space is large.

Though there are various other algorithms besides the ones detailed above, as seen in Figure 2.7, they possess a complexity that is beyond what is required for this dissertation. However, it is important to retain the idea that there are multiple choices available, each with their strengths and weaknesses.

2.5 Related Work

There are many existing link adaptation algorithms. In this chapter, we analyze four of them, with two of them being among the most widely used. This chapter also exposes what kind of problems and limitations are inherent to those algorithms, one of the most common being only adjusting the MCS parameter and not utilizing more capabilities offered by the latest IEEE 802.11 standards, such as IEEE 802.11n and IEEE 802.11ac. With the rise of ML, new approaches are being taken when it comes to developing new link adaptation algorithms. This section is dedicated to the analysis of two particular and very relevant works that managed to create new algorithms using RL: SmartLA [2] and EDRA [3].

2.5.1 SmartLA

SmartLA is a link adaptation algorithm that uses RL. It attempts to optimally configure PHY and MAC parameters to adapt the link, the parameters being MCS, GI, CB, and Level of Frame Aggregation (LFA). It actively searches for the best combination of these parameters and uses the Frame Error Ratio (FER) and Bit Error Ratio (BER) as observation metrics.

In this work, the authors made the following associations to the state, action and reward:

- **State**: A state is represented by a tuple that contains the previous values configured for the MCS, CB, GI and LFA. The tuple is therefore a unique combination of these values;

- **Action**: An action is performed by the agent to change the state. Its purpose is to configure new values for the MCS, CB, GI and LFA, but it does not mean that it always changes these four values. For example, an action may consist of only changing the MCS and the CB values, or even just the MCS value. To better illustrate this, suppose that the tuple for the current state, S_1, contain the values MCS_1, CB_1, GI_1 and LFA_1. If the agent performs an action that changes the MCS and GI values to MCS_2 and GI_2, respectively, then the system will move to state S_2, represented by a tuple that now contains the values MCS_2, CB_1, GI_2 and LFA_1;

- **Reward**: After an action is taken by the agent and the system moves on to a new state, data transmissions are made using the values in the state tuple for the PHY and MAC parameters. The system then measures the BER and uses it to calculate a reward for the agent. Therefore, the lower the measured BER, the higher the reward the agent gets, positively reinforcing the combination of values that the agent chose.

The BER and FER measurements are also used to obtain the Q-value. This value represents how useful a given action is in gaining some future reward. A high Q-value indicates either a high BER, high FER, or even both, and the goal is to decrease the Q-value as much as possible.

The authors of this work tested their solution against other existing algorithms like SampleLite, Minstrel, and Minstrel-HT. They have shown that SmartLA significantly outperforms the aforementioned algorithms, proving the potential of ML-based (and more specifically, RL-based) link adaptation algorithms.

2.5.2 EDRA

The authors of this work identified the main limitations of current adaptation algorithms, which are no joint rate and bandwidth adaptation, lack of scalability and no online learning capability. To address these limitations, they designed an algorithm named EDRA (Experience Driven Rate Adaptation) using DRL, which attempts to optimally configure the MCS, CB and NSS parameters. To accomplish this, the sub-frame loss, Received Signal Strength Indicator (RSSI), and the bandwidth's Service Time Ratio (STR) are used as observation metrics. In this work, the state, action, and reward were defined as follows:

- **State**: A state contains observations of the current radio channel conditions and the values used to configure the parameters. It therefore includes the MCS, CB and NSS values, as well as the measured sub-frame loss, RSSI and STR. The sub-frame loss and RSSI represent the current link quality, while the STR indicates the congestion level of the bandwidth being used by the current MCS value;

- **Action**: An action in this model is in fact a set of six actions, A_1 to A_6, where each action represents upward and downward moving directions of CB, MIMO mode and MCS value in sequence;
- **Reward**: The reward in this model is calculated based on the obtained goodput. In short, the goodput values measured are scaled so that they are relative to the highest goodput in the same environment.

This solution was tested against the Iwlwifi driver, that contains the Iwl-mvm-rs adaptation algorithm, and Minstrel. Through this comparison, they have shown that EDRA is able to outperform Iwl-mvm-rs and Minstrel by up to 821.4% and 242.8%, respectively, proving the effectiveness of using DRL techniques for wireless link adaptation.

2.6 Summary

This chapter gave an introduction to the IEEE 802.11 set of technical standards, an overview of four different link adaptation algorithms, a description of ML, its paradigms and DL, a short survey on popular DRL algorithms and, finally, an analysis of two related works, SmartLA and EDRA.

Through the study of the evolution of the IEEE 802.11 standards and state of the art link adaptation algorithms, it was possible to conclude about the limitations of the existing algorithms and that the capabilities of the latest standards are not being fully exploited. With the recent fast development of ML, ML techniques are being increasingly applied to many areas, including Wireless Communications. The works analyzed in Section 2.5 have shown the effectiveness of applying RL and DRL models to the problem of wireless link adaptation, as the algorithms developed in those works greatly outperform state of the art algorithms such as Minstrel-HT and Iwl-mvm-rs. However, and in spite of these promising results, there is still little research done in this area and much room for improvement. One particular problem that, to the best of our knowledge, has not been approached, is the resiliency of an algorithm when it comes to sudden changes in the radio channel conditions. Even if an algorithm manages to optimally configure various parameters and maximize the obtained throughput, its efficiency will suffer if it is unable to handle a sudden deterioration in the radio link quality. This is the subject of this dissertation.

Chapter 3

DRL-LA Algorithm

The main goal of this dissertation was to develop a link adaptation algorithm, focused on, but not limited to, the IEEE 802.11ac standard, which is capable of optimally configuring a set of parameters according to the radio channel conditions observed in real-time. The algorithm must also be resilient to sudden radio channel condition changes, a factor that has been overlooked in the state of the art.

This chapter details the DRL-LA algorithm, the required setup and software tools used, and the work structure followed to develop the algorithm.

3.1 Proposed Solution

Having analysed the existing paradigms and techniques of ML, it seems that RL makes the most sense for this challenge, as the agent in a RL model adapts to a dynamic environment, and unlike other forms of ML, there is no need for a data set. Considering that the goal of the link adaptation algorithm is to adapt a set of parameters to maximize the throughput in a wireless connection with varying radio channel conditions, the process behind RL is very in tune with this.

To further take advantage of the capabilities of modern computing systems, DL can be used along with RL, obtaining DRL, that was already briefly introduced in Chapter 2. Using DRL instead of RL implies using DNNs that allow the algorithm to learn faster and provide higher quality results, although it introduces trade offs like a much higher complexity and the need for powerful hardware.

The algorithm developed uses the Signal-to-Noise Ratio (SNR) for determining the radio channel quality and configures three parameters accordingly: MCS, CB and GI. We've seen before that the RL loop consists in cycles that have states, actions and rewards, which for this solution, are defined as follows:

- **State** - A state consists of the observation made on the SNR value, and the previous values used to configure the three aforementioned parameters. It can be represented as a list, like for example, $S_n = \{MCS_{n-1}, CB_{n-1}, GI_{n-1}, SNR_n\}$, where n is the current cycle;

- **Action** - Actions taken by the DRL agent are what actually configure the three parameters at every cycle. Therefore, an action consists of a MCS value, a GI value and a CB value. They can also be represented as a list, like for example, $A_n = \{MCS_n, CB_n, GI_n\}$. Only discrete values can be chosen, which means that the action space (the space that includes every possible action) is discrete;

- **Reward** - The reward returned to the agent after it takes an action is computed using the following function:

$$\textbf{Reward} = FSR \times \frac{(x \times avgMCS + y \times avgCB + \frac{1}{z \times avgGI})}{\text{Number of frames sent by AP}}$$

  where:

  – **FSR** is the Frame Success Ratio and is equal to the number of frames sent by the AP divided by the number of frames received by a STA, in a cycle:

$$\textbf{FSR} = \frac{\text{Number of frames sent by AP}}{\text{Number of frames received by STA}}$$

  – **avgMCS** is the normalized MCS rate chosen by the agent to transmit frames, during a cycle:

$$\textbf{avgMCS} = \frac{\text{Chosen MCS rate}}{\text{Maximum MCS rate available}}$$

  – **avgCB** is the normalized CB value chosen by the agent to transmit frames, during a cycle:

$$\textbf{avgCB} = \frac{\text{Chosen CB value}}{\text{Maximum CB value available}}$$

  – **avgGI** is the normalized GI value chosen by the agent to transmit frames, during a cycle, divided by the minimum GI available, as smaller GIs are preferred, if the radio channel conditions allow for it:

$$\textbf{avgGI} = \frac{\text{Chosen GI value}}{\text{Minimum GI value available}}$$

  – **x**, **y** and **z** are variables that define how much weight each parameter has over the reward value. These variables had to be optimally tuned in order to achieve an efficient reward function.

The reason for normalizing the MCS, CB and GI values is to guarantee that the value outputted by the reward function is between 0 and 1. This ensures that the agent is able to distinguish good and bad actions more effectively and converges more quickly, as the range of rewards is smaller. Furthermore, normalizing these values also ensures that they all have an impact on the reward. For example, without normalization, the MCS and CB values would be much larger than the GI value, as the former are in the order of mega (10^6) while the latter is in the order of nano (10^{-9}).

Each cycle is set to have a duration of 100 milliseconds, as this is a long enough period for the agent to decide on an action, while also guaranteeing that not too much time passes until the agent takes another action, ensuring a constant good configuration of the link. The agent picks an action at the beginning of the cycle and the three parameters are configured according to that action. Then, during that time period, every frame is sent by the AP using the configurations made. For each frame received by the STA, the SNR value is observed and the STA sends to the AP an acknowledgment frame. If the AP does not receive the acknowledgement, then it considers that the frame was lost. Finally, at the end of each cycle, the reward is calculated as described above.

3.1.1 Action Space

We mentioned before that an action taken by the agent configures the three parameters simultaneously. An alternative approach would be the agent taking three actions at once, one for each parameter. However, creating an agent that can produce multi-action outputs is a lot more complex than a simple single-action agent, and it was for this reason that the latter approach was chosen. To understand how a single action can configure three parameters, we first need to define what values each parameter can take.

802.11ac - VHT MCS, SNR and RSSI

VHT MCS	Modulation	Coding	20MHz				40MHz				80MHz				160MHz			
			Data Rate		Min. SNR	RSSI	Data Rate		Min. SNR	RSSI	Data Rate		Min. SNR	RSSI	Data Rate		Min. SNR	RSSI
			800ns	400ns			800ns	400ns			800ns	400ns			800ns	400ns		
1 Spatial Stream																		
0	BPSK	1/2	6.5	7.2	2	-82	13.5	15	5	-79	29.3	32.5	8	-76	58.5	65	11	-73
1	QPSK	1/2	13	14.4	5	-79	27	30	8	-76	58.5	65	11	-73	117	130	14	-70
2	QPSK	3/4	19.5	21.7	9	-77	40.5	45	12	-74	87.8	97.5	15	-71	175.5	195	18	-68
3	16-QAM	1/2	26	28.9	11	-74	54	60	14	-71	117	130	17	-68	234	260	20	-65
4	16-QAM	3/4	39	43.3	15	-70	81	90	18	-67	175.5	195	21	-64	351	390	24	-61
5	64-QAM	2/3	52	57.8	18	-66	108	120	21	-63	234	260	24	-60	468	520	27	-57
6	64-QAM	3/4	58.5	65	20	-65	121.5	135	23	-62	263.3	292.5	26	-59	526.5	585	29	-56
7	64-QAM	5/6	65	72.2	25	-64	135	150	28	-61	292.5	325	31	-58	585	650	34	-55
8	256-QAM	3/4	78	86.7	29	-59	162	180	32	-56	351	390	35	-53	702	780	38	-50
9	256-QAM	5/6			31	-57	180	200	34	-54	390	433.3	37	-51	780	866.7	40	-48
2 Spatial Streams																		
0	BPSK	1/2	13	14.4	2	-82	27	30	5	-79	58.5	65	8	-76	117	130	11	-73
1	QPSK	1/2	26	28.9	5	-79	54	60	8	-76	117	130	11	-73	234	260	14	-70
2	QPSK	3/4	39	43.3	9	-77	81	90	12	-74	175.5	195	15	-71	351	390	18	-68
3	16-QAM	1/2	52	57.8	11	-74	108	120	14	-71	234	260	17	-68	468	520	20	-65
4	16-QAM	3/4	78	86.7	15	-70	162	180	18	-67	351	390	21	-64	702	780	24	-61
5	64-QAM	2/3	104	115.6	18	-66	216	240	21	-63	468	520	24	-60	936	1040	27	-57
6	64-QAM	3/4	117	130.3	20	-65	243	270	23	-62	526.5	585	26	-59	1053	1170	29	-56
7	64-QAM	5/6	130	144.4	25	-64	270	300	28	-61	585	650	31	-58	1170	1300	34	-55
8	256-QAM	3/4	156	173.3	29	-59	324	360	32	-56	702	780	35	-53	1404	1560	38	-50
9	256-QAM	5/6			31	-57	360	400	34	-54	780	866.7	37	-51	1560	1733.3	40	-48
3 Spatial Streams																		
0	BPSK	1/2	19.5	21.7	2	-82	40.5	45	5	-79	87.8	97.5	8	-76	175.5	195	11	-73
1	QPSK	1/2	39	43.3	5	-79	81	90	8	-76	175.5	195	11	-73	351	390	14	-70
2	QPSK	3/4	58.5	65	9	-77	121.5	135	12	-74	263.3	292.5	15	-71	526.5	585	18	-68
3	16-QAM	1/2	78	86.7	11	-74	162	180	14	-71	351	390	17	-68	702	780	20	-65
4	16-QAM	3/4	117	130	15	-70	243	270	18	-67	526.5	585	21	-64	1053	1170	24	-61
5	64-QAM	2/3	156	173.3	18	-66	324	360	21	-63	702	780	24	-60	1404	1560	27	-57
6	64-QAM	3/4	175.5	195	20	-65	364.5	405	23	-62			26	-59	1579.5	1755	29	-56
7	64-QAM	5/6	195	216.7	25	-64	405	450	28	-61	877.5	975	31	-58	1755	1950	34	-55
8	256-QAM	3/4	234	260	29	-59	486	540	32	-56	1053	1170	35	-53	2106	2340	38	-50
9	256-QAM	5/6	260	288.9	31	-57	540	600	34	-54	1170	1300	37	-51			40	-48

Figure 3.1: Theoretical throughputs table for IEEE 802.11ac [28].

Figure 3.1 represents every theoretical maximum throughput for each combination of the MCS, GI, CB and NSS parameters in the IEEE 802.11ac standard. Due to the limited time available to

develop this dissertation, only SISO mode was explored, which means the NSS is always 1. This being said, the following decisions were made for each parameter:

- **MCS** - This parameter can take all the available values, which range from 0 to 9. If the CB is equal to 20 MHz, then the available values range from 0 to 8;
- **GI** - The GI parameter can also take all available values, which are only two: 800 and 400 nanoseconds;
- **CB** - Unlike the previous parameters, the CB was limited, as it can only be configured to 20, 40 and 80 MHz, with 160 MHz being excluded.

Taking these decisions into account, we can crop Figure 3.1 to only show the combinations available to the agent.

VHT MCS	Modulation	Coding	20MHz				40MHz				80MHz			
			Data Rate		Min. SNR	RSSI	Data Rate		Min. SNR	RSSI	Data Rate		Min. SNR	RSSI
			800ns	400ns			800ns	400ns			800ns	400ns		
1 Spatial Stream														
0	BPSK	1/2	6.5	7.2	2	-82	13.5	15	5	-79	29.3	32.5	8	-76
1	QPSK	1/2	13	14.4	5	-79	27	30	8	-76	58.5	65	11	-73
2	QPSK	3/4	19.5	21.7	9	-77	40.5	45	12	-74	87.8	97.5	15	-71
3	16-QAM	1/2	26	28.9	11	-74	54	60	14	-71	117	130	17	-68
4	16-QAM	3/4	39	43.3	15	-70	81	90	18	-67	175.5	195	21	-64
5	64-QAM	2/3	52	57.8	18	-66	108	120	21	-63	234	260	24	-60
6	64-QAM	3/4	58.5	65	20	-65	121.5	135	23	-62	263.3	292.5	26	-59
7	64-QAM	5/6	65	72.2	25	-64	135	150	28	-61	292.5	325	31	-58
8	256-QAM	3/4	78	86.7	29	-59	162	180	32	-56	351	390	35	-53
9	256-QAM	5/6			31	-57	180	200	34	-54	390	433.3	37	-51

Figure 3.2: Cropped theoretical throughput table to only show the combinations available to the agent.

By observing Figure 3.2, we can see that there are 58 combinations in total. Although this is not a large action space by any means, it is big enough so that the agent needs several hours, or even days, to properly train and learn, depending on how powerful the computer being used is. This was the main reason why the 160 MHz value for the CB parameter was cut, as that would bring the action space size to a total of 78 actions, which already makes a huge difference when it comes to the time and resources required for training. This is also why the NSS was not considered as a parameter for the agent to configure, as that would increase the action space even more. For example, if the agent could configure the NSS to be 2, that would bring the action space size to a total of 156 actions. Furthermore, this would also require the use of sophisticated and more complex DRL algorithms and neural networks, as the agent might not be able to converge if the algorithm being used is incapable of handling such a large number of actions.

3.1.2 Backup Mechanism

Like previously mentioned, DRL-LA has a backup mechanism that is activated whenever the parameter configurations chosen by the agent fail to achieve acceptable results. This may happen

when, for example, the radio channel quality suddenly deteriorates due to an obstacle, causing the SNR value to drop considerably. This is a problem because the DRL agent will only take this SNR variation into account in the next cycle, which means that the parameter configurations remain inadequate for the rest of the current cycle, causing frames to be lost and the throughput to greatly drop. In cases like this, the backup mechanism ensures that a quick reconfiguration of the parameters is made to avoid this situation. The backup mechanism works as follows:

1. Starting from the beginning of the cycle, after the agent configures the parameters, a counter is initialized that counts how many frames are lost. A frame is considered to be lost when the AP does not receive an acknowledgement packet from the STA, which means that there is no confirmation that the frame was received;

2. If a certain number of frames n is lost, then the first phase of the backup mechanism is activated. In this phase, the CB value is reduced to a lower, adjacent value. For example, if the current CB value is 80 MHz, then it will be reduced to 40 MHz. However, if the current CB value is already the lowest possible – 20 MHz – then the backup mechanism immediately activates the second phase. This first phase is an attempt to solve the situation without making an aggressive reconfiguration right away, aiming at maintaining the highest throughput possible;

3. If, after activating the first phase, frames are still being lost, then the second phase is activated. In this phase, the tolerance for the frames that can be lost is reduced in order to make the mechanism react faster. For as long as a small number of frames keep being consecutively lost, the mechanism keeps reducing the MCS, until it eventually reaches the optimal value for this parameter that assures that the frames will not be lost. If the backup mechanism is forced to decrease the MCS value all the way down to 0, then the CB value will also be set to 20 MHz, if it was not already set with that value.

The following figure helps visualize the type of situation that this mechanism aims to prevent:

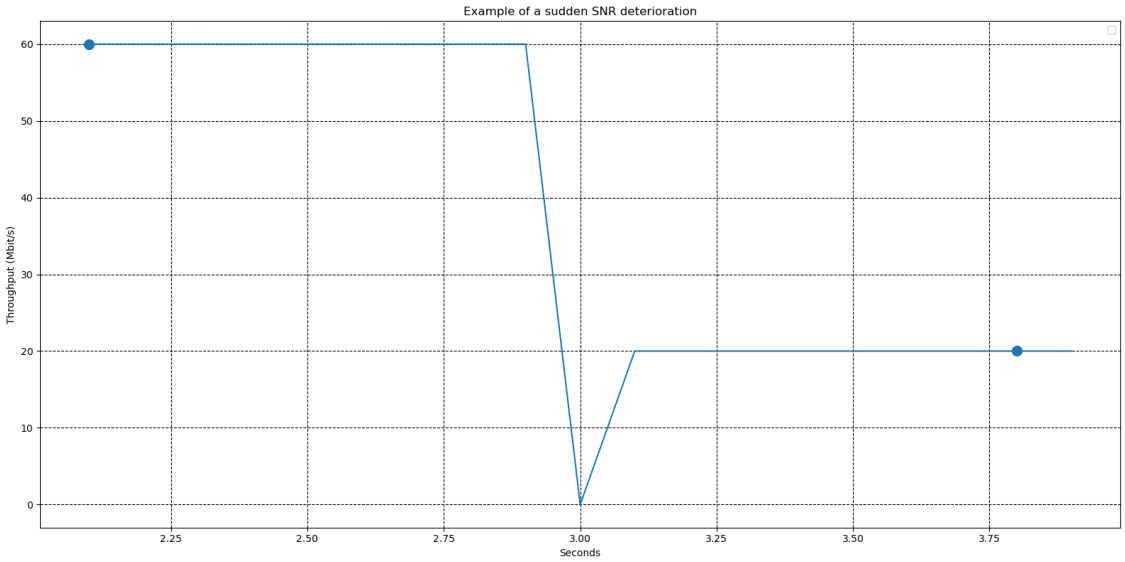

Figure 3.3: Example of a sudden SNR deterioration.

In the figure presented, it is possible to observe that the wireless link was initially stable, obtaining a constant throughput of 60 Mbits/s. However, after 3 seconds, the SNR value suddenly dropped, causing the throughput to reach 0 Mbits/s. The link was properly readjusted only after 100 milliseconds, because the agent needed a new cycle to make an observation on the SNR value and reconfigure the parameters appropriately. With the described backup mechanism, this situation would not occur. The mechanism probably will not reconfigure the parameters to optimal values, but will definitely prevent the throughput from reaching 0 Mbits/s until the agent has a chance to take another action. To, once again, facilitate the understanding of this whole process, Figure 3.4 is a flowchart that represents a cycle.

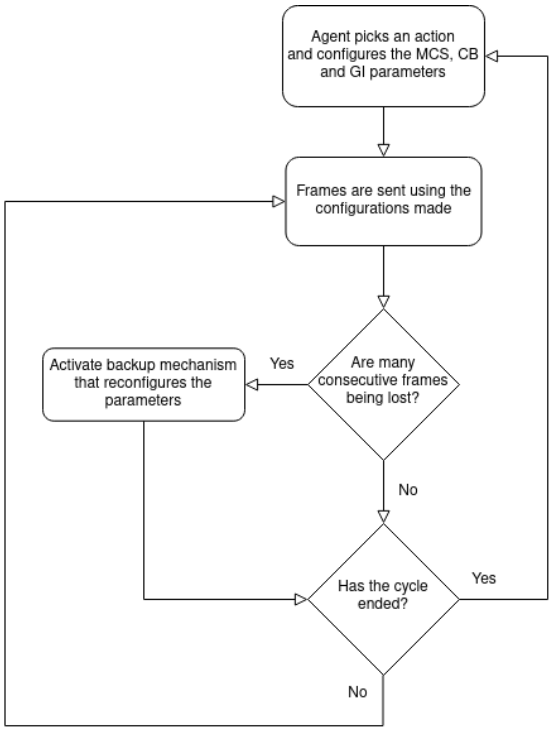

Figure 3.4: Flowchart representation of a cycle.

3.2 Software Tools and Setup

In order to implement the solution described, a number of software tools had to be used. The selection of these tools was made with different aspects in mind, like the compatibility between them, the easiness to set them up and their efficiency. When it comes to developing the DRL model, it would be a lengthy and difficult process to do it from scratch. To this end, there are several ML frameworks that assist with the implementation of models, the most popular being Tensorflow [29], PyTorch [30] and Shogun [31]. For the development of this algorithm, Tensorflow was chosen, as it is currently the most used framework, having a lot of material available, active support and is fairly easy to set up. Furthermore, it is compatible with TF-Agents [32] and Keras-rl2 [33], two well-known libraries that implement various RL and DRL algorithms. Although TF-Agents is more recent, has more support and implements more algorithms than Keras-rl2, the latter is easier to configure and use, being sufficient for the task at hand. Therefore, Keras-rl2 was chosen as the library to implement the DRL algorithm.

The DRL model built using the aforementioned tools needs an environment for the agent to learn and operate in. Considering that the purpose of the algorithm is to dynamically adapt wireless link connections, the environment is therefore a WLAN with atleast one AP and one STA. The agent configures the MCS, GI and CB parameters used for connections between AP and STAs, and observes the SNR values. Although the ultimate goal is to have an algorithm capable of operating

in real life scenarios, with real NICs, a simulation environment was chosen instead for the elaboration of this dissertation. The reason for this is that implementing a DRL algorithm in a NIC and making it learn and operate in a real environment is far too complex for the time that was available to work on this dissertation. To construct the simulation environment, ns-3 [34] was chosen due to being a fairly intuitive network simulator to work with, having plenty of documentation available and being the most used network simulator in the scientific community. Furthermore, it has the ns3-gym toolkit [35] available, which integrates both OpenAI Gym [36] and ns-3. Essentially, ns3-gym provides an API for communication between ns-3 and the DRL algorithm built with Tensorflow and Keras-rl2, allowing the agent to take actions and make observations in the simulation.

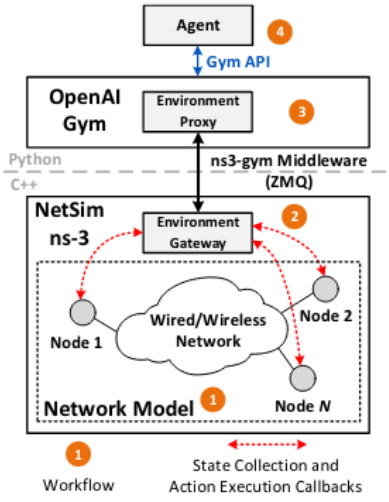

Figure 3.5: Architecture of the ns3-gym framework [37].

As we can see in Figure 4.2, the usage of two different languages was required, Python and C++. The former was used to build the DRL agent, while the latter was used to make the ns-3 simulations program and the ns3-gym interface program. To actually run simulations and train the DRL agent, two machines were used. One was a home desktop with a capable NVIDIA GPU, allowing for the use of the CUDA Toolkit [38], which greatly accelerates the training sessions. The other was a virtual machine provided by INESC TEC, that did not have a GPU available, but did have a decent CPU and RAM size, although the training sessions were slower due to not being GPU accelerated. With these two powerful machines at our disposal, it was possible to run multiple concurrent training sessions on each machine. This was particularly useful to test different hyperparameter configurations at the same time. For example, the most used configuration was four simultaneous training sessions on each machine, giving eight in total. This was possible because the ns-3 software only uses one CPU core per simulation. Considering that both machines have eight core CPUs, we can use each core for a different simulation, although we never went above four simulations per machine, as not to overload them.

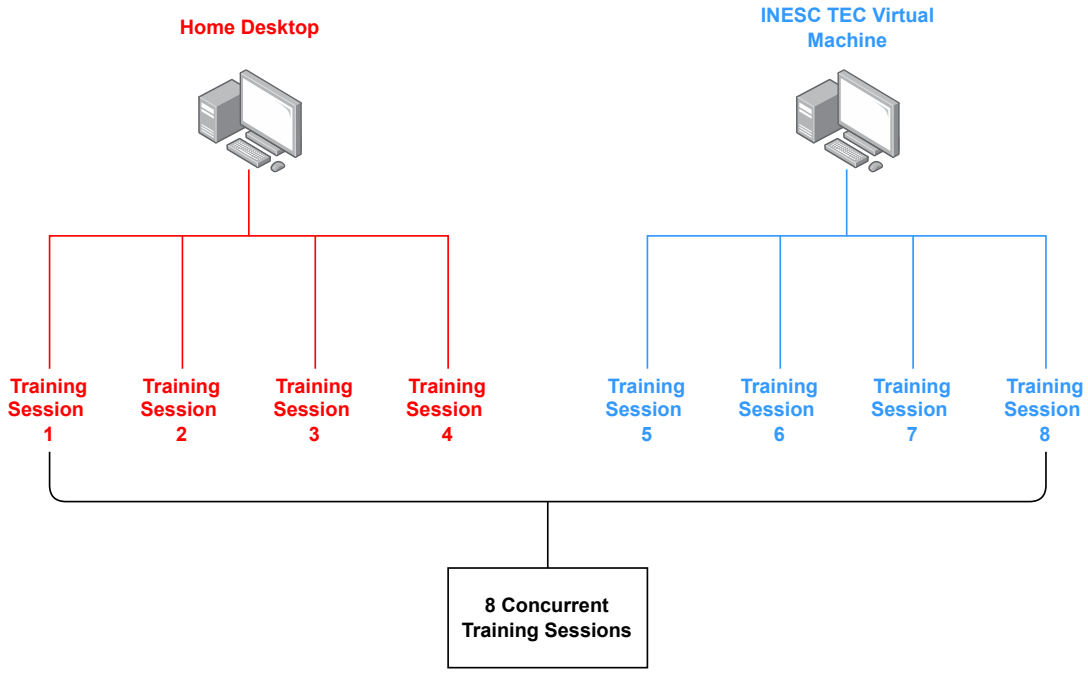

Figure 3.6: Concurrent training sessions on the two machines.

Chapter 4

Implementation

The implementation of the solution consisted in programming three different components: the DRL agent, the ns3-gym interface and the ns-3 simulations. As mentioned previously, the DRL agent was coded in Python, using Tensorflow and Keras-rl2, while the other two were coded in C++.

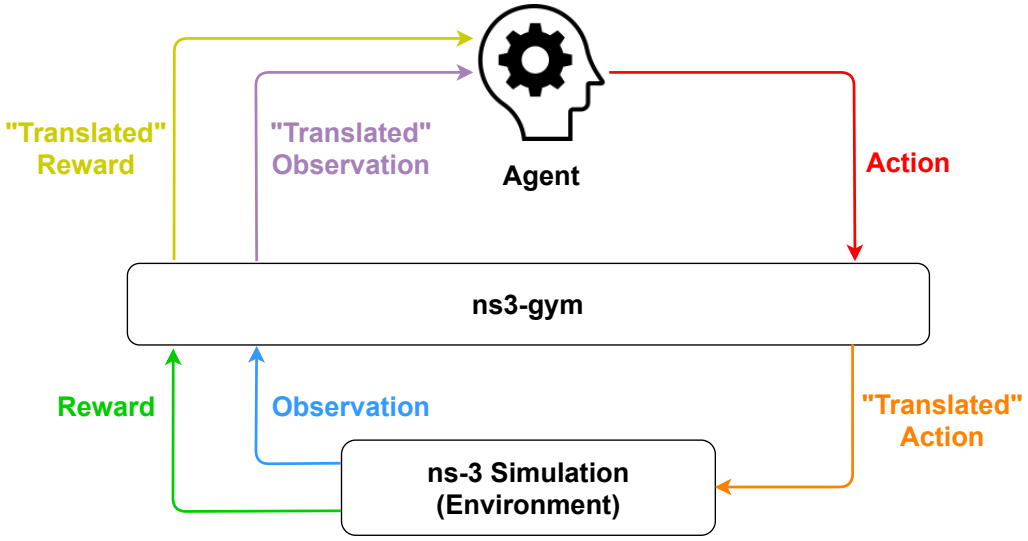

Figure 4.1: RL loop represented with ns3-gym and the ns-3 simulation.

To properly detail the whole process, this chapter is divided into four sections:

- Ns-3 Simulations - The ns-3 software allows us to create highly customizable network simulations to train and validate the algorithm. The simulations are programmed in C++ using the libraries provided by ns-3;
- Ns-3 Gym Implementation - The ns-3 gym toolkit was used to create a custom OpenAI Gym environment and allow the communications between the DRL agent and the ns-3 simulations. To this end, a program was developed in C++ that uses the functionalities provided by this toolkit to implement the Gym environment. It is in this program that we defined things

such as the reward function, the observations that are returned to the agent, the actions that the agent takes, etc;

- DRL Agent - This section explains how the DRL agent was implemented: which DRL algorithm was used, the neural networks built and the tuning of hyperparameters;
- Reward Function Tuning - The tuning of the reward function consisted in determining how much each of the configurable parameters should weigh on the reward calculation.

4.1 Ns-3 Simulations

Before developing the DRL Agent or programming the ns-3 gym interface, a C++ program was written using ns-3 libraries in order to create the various simulations in which the agent would train and be evaluated. This is a general purpose program, as it contains all the simulation scenarios planned in the context of this dissertation. By using appropriate arguments when invoking the program, we can control which of the available simulation scenarios to use, for how long the simulation should run and if it is going to be used to train the DRL agent or to test its performance. The arguments that can be used to invoke the program are:

- **Port** - Used to open a connection between the DRL agent, the ns-3 gym interface and the ns-3 simulation. If the user does not define it, it takes a default value of 5555. It is because of this argument that multiple simulations can run simultaneously, as it allows the user to open various connections by using different ports for each one;
- **Algorithm** - Defines which algorithm will be used for the simulation. There are three algorithms available: our DRL-LA algorithm, the Minstrel-HT algorithm detailed in Chapter 2 and the Ideal algorithm, an algorithm provided by ns-3 that is able to adapt the link ideally by prioritizing the BER [39];
- **Simulation Type** - Defines if the simulation will be used for training the DRL agent or for evaluating the performance of the algorithm chosen with the previous argument;
- **Mobility Model** - Chooses the mobility model. Each simulation scenario has a different mobility model. Thus, each model defines the way the network nodes move throughout the simulation;
- **Duration** - Defines the duration of the simulation, in seconds.

This program is invoked by the DRL agent program and the user can define the arguments there.

Every simulation scenario had one thing in common: there were only two network nodes, an AP and a STA. A large part of this program consists in setting up these two nodes, which requires creating the network they are in, setting the protocol that they use to communicate between each other, and finally, defining the transport layer payload size of the packets that they send. These configurations were also common to every simulation scenario, and it was defined that the nodes

communicate using the UDP protocol and the payload size is equal to 1472 bytes. Besides setting up the network nodes, the program also defines some crucial parameters, such as:

- **Signal Transmission Power** - This parameter was set to 20 dBm, which is the default value for many home routers. It indicates how powerful the wireless signal transmission is;
- **Antenna Gain** - This parameter, for simplicity, was set to 0 dBi. It determines the direction in which the antenna radiates the signal. An omni-directional antenna is an antenna that has no directivity, or in other words, an antenna that radiates energy equally in all directions. Antennas of this kind have a gain of 0 dBi;
- **Center Frequency** - This parameter was set to 5210 MHz. It defines the center frequency of the frequency band being used. Considering that the simulations were run using the IEEE 802.11ac standard, which only supports the 5 GHz frequency band, the center frequency needed to be included in this band, while also supporting channel bandwidths up to 80 MHz. For these reasons, the aforementioned value was chosen;
- **Propagation Loss Model** - The model used was always the Friis Free Space Propagation Model, which models a line-of-sight path loss in a free space environment without any interfering objects. The Propagation Loss Model characterizes how the radio waves propagate;
- **Error Rate Model** - The model chosen for this parameter was the Nist Error Model [40], as it models an error rate for different modulation and coding schemes. Ns-3 implements error models for Wi-Fi networks to simulate errors in frames.

When it comes to the simulation scenarios, each of them had an individual function that contained the code for creating the scenario. When the program is invoked, the arguments passed by the user are parsed, and the argument that controls which simulation scenario will run – the Mobiliy Model argument – is used to call the corresponding function. There were three scenarios considered: *waypoint*; *teleporting node*; and *waypoint with teleports*. The *waypoint* scenario consists in having the STA starting positioned right next to the AP and then moving away from the AP at a constant speed until it completely loses signal, which is around the 1300 meter mark. After losing signal, it returns to the AP at the same constant speed. The STA node takes the same amount of time in both components of the movement – moving away and returning to the initial position – as they last half the total simulation time each. For example, if the simulation lasts 100 seconds in total, each component will last 50 seconds.

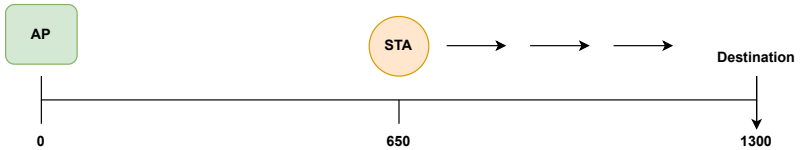

Figure 4.2: *Waypoint* scenario when the STA node is moving away from the AP node, with the STA currently at the halfway point.

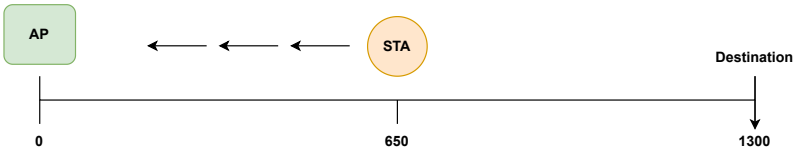

Figure 4.3: *Waypoint* scenario when the STA node is moving back towards the AP node, with the STA currently at the halfway point.

Figure 4.2 and Figure 4.3 represent the described scenario. This scenario is relevant because the SNR value will slowly decrease the further away the STA node gets from the AP, without any sudden changes, and then slowly increase again as the STA comes back to the AP. For this reason, this was the only scenario used to train the DRL agent, as it gives the agent an opportunity to observe a complete range of different SNR values and learn the best actions for each of them. A proper training session with a well built model will result in an agent that can optimally adapt the MCS, GI and CB values for pratically any SNR value. This means that, after training, the DRL agent should also perform very well in any other scenario, even though it only trained with the *waypoint* simulation, assuming that overfitting did not occur.

In the *teleporting node* scenario, the AP continuously teleports between 30 meters and 400 meters away from the AP. Additionally, the STA stays still for two seconds at each of these positions. Although this is not a realistic scenario, its purpose is to test how the DRL agent performs in a scenario where the SNR value changes abruptly. Forcing sudden extreme changes in the distance between the STA and the AP is a simple way to cause the intended SNR variations.

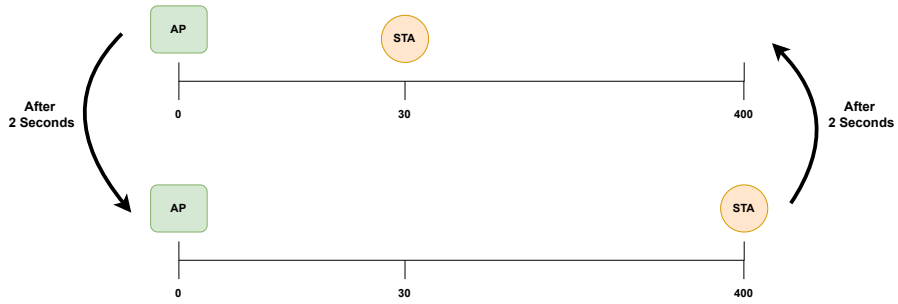

Figure 4.4: *Teleporting node* scenario.

Figure 4.4 illustrates the *teleporting node* scenario. Initially, the STA node was standing still at the 30 meter mark. After two seconds, it suddenly teleported to the 400 meter mark, causing the SNR value to suddenly decrease considerably. Two seconds after that, the STA node returned to the 30 meter mark, causing the SNR to suddenly increase to its previous value.

The *waypoint with teleports* scenario is very similar to the regular *waypoint* described before. The major difference is that, while the STA moves, it also teleports to random positions for brief periods of time, always returning to its last position before the teleport. Furthermore, the STA only reaches a maximum distance of 400 meters before returning back to the AP. The reason for this is so that the STA can teleport to distances longer than 400 meters in order to generate the

drastic SNR variations. The purpose of this scenario is to combine sudden SNR variations with the otherwise stable environment we had using *waypoint*. This mimics real life situations, where, for example, a person using a smartphone connected to the local Wi-Fi starts moving and encounters obstacles along the way that deteriorate the signal reception. Furthermore, when compared to the *teleporting node* scenario, the *waypoint with teleports* is a way more complex environment, as the STA only teleported between the same two positions in the former.

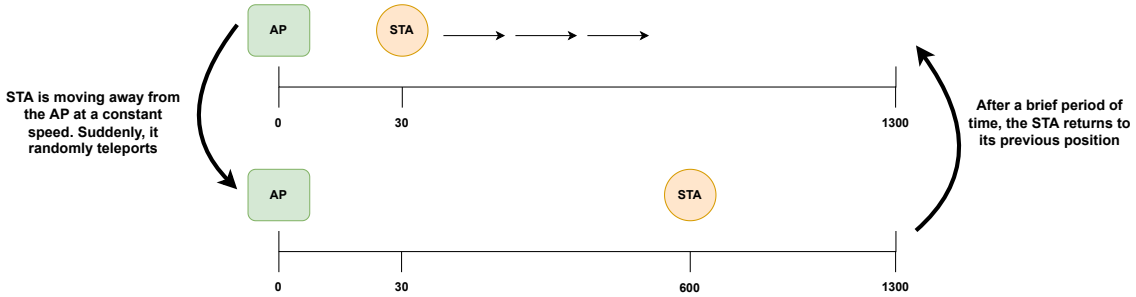

Figure 4.5: *Waypoint teleport* scenario.

Figure 4.5 illustrates an example of this scenario, in which the STA was moving away from the AP at a constant speed. When it reached the 30 meter mark, it suddenly teleported to the 600 meter mark, and as a consequence, the SNR drastically dropped. After two seconds, the STA returned to the 30 meter mark. In a real life scenario, these two seconds where the SNR hit low values could be equivalent to a situation where the person using a smartphone passed very quickly by an obstacle that caused the signal reception to deteriorate for a brief period.

Finally, one other important thing that this general purpose program does is connecting to the ns-3 gym interface. This is achieved by instantiating a class created in the ns3-gym program and then using functions provided by ns3-gym libraries with either the port defined by the user or the default port of 5555.

4.2 Ns3-gym

Creating a ns3-gym interface required once again to write a program in C++. In this program, we have a class named MyGymEnv. This class has a constructor that accepts as arguments a flag, that indicates if we're using DRL-LA or another algorithm, and an integer that defines the total time that the simulation should last.

```
MyGymEnv(bool mode, int simDuration);
```

Listing 4.1: MyGymEnv constructor.

This is the class instantiated in the ns-3 simulations program mentioned previously. The instructions provided by the ns3-gym developers state that the following functions must be implemented:

```
1 Ptr<OpenGymSpace> GetObservationSpace();
2 Ptr<OpenGymSpace> GetActionSpace();
3 Ptr<OpenGymDataContainer> GetObservation();
4 float GetReward();
5 bool GetGameOver();
6 std::string GetExtraInfo();
7 bool ExecuteActions(Ptr<OpenGymDataContainer> action);
```

Listing 4.2: Functions that must be defined for using ns3-gym.

Function 1 in Listing 4.2 controls how the observation space looks like. In this case, the DRL agent is supposed to only observe the non-discrete SNR value at each cycle. Function 2 has the same purpose of the previous function, but for the action space. As mentioned before, the action space contains 58 discrete value actions. Function 3 is used to obtain an observation from the simulation – the SNR value – and return it to the DRL agent. Function 4 is used to calculate a reward value and return it to the agent. This is where the reward function, detailed in Chapter 3, is implemented. Function 5 checks a predefined condition that ends the training episode if it is true, like if the total number of steps has been reached. Function 6 is meant to return extra information back to the agent, however, it was not properly defined and only returns a template string, as it had no relevant use for this case. Finally, function 7 is responsible for actually executing the action chosen by the DRL agent.

Besides the functions mentioned, other custom functions were implemented to monitor events in the network, such as the reception of ACKS, block ACKs or timeouts for the frames sent, in order to calculate the FSR value used in the reward function. It is also through a custom function in this program that the backup mechanism, detailed in Section 3.1.2, is implemented. The backup mechanism function takes advantage of the network monitoring done in the other custom functions to know when to become active and reconfigure the parameters. The optimal number of frames that must be lost for the mechanism to activate was determined empirically, through the observation of the results of simulations after the whole system was implemented and the DRL agent was trained. In the end, it was defined that 40 consecutive frames must be lost in order to activate the first phase of the backup mechanism, and from there, the tolerance reduces to 20 consecutive frames lost for activating the second phase and to keep reducing the MCS value while in the second phase.

4.3 DRL Agent

The DRL agent was implemented with a program coded in Python. The implementation of the DRL agent was relatively straightforward because of Tensorflow and Keras-rl2. These Python frameworks allowed for an easy and quick creation of a neural network, the size and activation functions of each layer, the DRL algorithm to be used and the configuration of hyperparameters. While the DRL algorithm used remained the same for every iteration of the implementation

process, the structure of the neural network and the values used for the configuration of the hyper-parameters changed.

4.3.1 Environment Creation

The first thing that the program does is connect to ns3-gym and ns-3 to instantiate an environment. It is with this instance that the agent interacts with. To do this, a Python function provided by the ns3-gym module is used.

```
ns3env.Ns3Env(port, algorithm, simType, mobilityModel, duration);
```

Listing 4.3: Environment instantiating.

The arguments used to invoke this function are the ones already listed in Section 4.1. When the environment is instantiated with these arguments, they are then passed to the ns-3 simulations program to be parsed and used. The user defines these arguments when invoking the Python program. This means that the Python program itself requires these arguments in other to run. Furthermore, by properly running the Python program, this function automatically starts the ns3-gym and ns-3 programs in order to instantiate the environment and initiate the simulation, so there is no need to run those programs manually. To run all these components together, the user needs to access a terminal window, change the directory to the folder that contains the DRL agent's Python program, the ns3-gym program and the ns-3 simulations program. This folder is itself situated inside the ns-3 installation directory. Finally, the user executes the Python program through the terminal, with appropriate arguments.

```
(base) heber@heber-HP-EliteBook-840-G1:~$ cd Documents/INESC-TEC/Tese/Código/ns-allinone-3.35/ns-3.35/scratch/opengym
(base) heber@heber-HP-EliteBook-840-G1:~/Documents/INESC-TEC/Tese/Código/ns-allinone-3.35/ns-3.35/scratch/opengym$ python agent.py drl train waypoint 300 5554
```

Figure 4.6: Example of how to run the DRL agent Python program on the terminal with the DRL-LA algorithm.

Figure 4.6 gives an example on how to run the DRL agent's Python program, which is called *agent.py*. In this example, the program was invoked with arguments that define that the algorithm to use is DR-LA, the simulation type is a training session, the simulation scenario – or mobility model – is the *waypoint* scenario, the duration per simulation is 300 seconds and the port to use is 5554. All these arguments will be passed to the ns-3 program, as mentioned earlier. Furthermore, even if the algorithm to use was not DRL-LA, we would still use the DRL agent's Python program to launch everything, even though the agent would become passive and not configure anything. The reason for doing this is that we still want to to be able to use the custom functions implemented in ns3-gym, in order to monitor events in the network and gather metrics to evaluate the performance of other algorithms.

```
(base) heber@heber-HP-EliteBook-840-G1:~$ cd Documents/INESC-TEC/Tese/Código/ns-allinone-3.35/ns-3.35/scratch/opengym
(base) heber@heber-HP-EliteBook-840-G1:~/Documents/INESC-TEC/Tese/Código/ns-allinone-3.35/ns-3.35/scratch/opengym$ python agent.py minstrel eval waypoint 300 5554
```

Figure 4.7: Example of how to run the DRL agent Python program on the terminal, but with the Minstrel-HT algorithm.

Figure 4.7 is an example of how run a simulation with the Minstrel-HT algorithm. In this example, a simulation of 300 seconds would be initiated to evaluate the performance of the Minstrel-HT algorithm.

4.3.2 DRL Algorithm

When it comes to the DRL algorithm, there were plenty of options to choose from, such as the ones mentioned in Chapter 2. To address the challenge at hand, the Deep-Q Learning algorithm was chosen, as it is one of the most prominent DRL algorithms, being very easy to build and train, as well as being highly effective for applications with relatively small discrete action spaces, suiting this challenge very well. This algorithm is already implemented by the Keras-rl2 library, which means using it is as simple as invoking the function provided.

```
dqn_agent = DQNAgent(model, nb_actions, memory,
target_model_update, policy, batch_size, gamma);
```

Listing 4.4: Function provided by Keras-rl2 to create a Deep-Q Learning agent.

Listing 4.4 shows the function used to instantiate what is referred to as a Deep-Q Network (DQN) agent, an agent that uses the Deep-Q Learning algorithm. The function is invoked using the following arguments:

- **model** - A Keras model, that defines the structure of the neural network. This model is created beforehand using another function provided by Keras, and defines parameters such as the number of hidden layers, the number of neurons per hidden layer and how the neurons are connected to each other, as well as the activation functions used;
- **nb_actions** - The number of possible actions that the agent can take, which is 58;
- **memory** - The memory argument refers to the replay buffer. It is also defined beforehand using a function provided by Keras-rl2;
- **target_model_update** - The Q function is recursive and when the agent updates its network for a given state and action, that update also impacts the prediction it will make for other states and actions. This can make for a very unstable network. This limitation is addressed by using a target network, which is a copy of the DQN that is not trained, but rather replaced with a fresh copy every so often. Therefore, this parameter controls how frequently this happens;
- **policy** - The policy is essentially the strategy that an agent uses to take actions. It defines how the agent behaves when given a state. For this reason, the policy is related to the ε-greedy parameter. The way the ε-greedy decays is determined by the policy and the policy itself evolves during the training. Similarly to some of the previous parameters, it is defined using a function provided by Keras-rl2;

- **batch_size** - This argument defines the batch size hyperparameter;
- **gamma** - This argument defines the discount factor hyperparameter.

After invoking this function, the DQN agent is created and all that is left to do is compile it and begin the training session. This is done by invoking the following functions:

```
dqn_agent.compile(optimizer);
dqn_agent.fit(environment, nb_steps, nb_max_episode_steps);
```

Listing 4.5: Functions provided by Keras-rl2 to compile the created agent and begin the training session.

The first function shown in Listing 4.5 is responsible for compiling the DQN agent. It takes as an argument an optimizer. Keras-rl2 has a small list of optimizers available. The Adam optimizer [41] was chosen, as it is one of the most popular and efficient gradient descent optimizers available. The second function is responsible for actually initiating the training session, and it takes the following arguments:

- **environment** - The environment that the agent interacts with. We use the environment instantiated with the function presented in Section 4.3.1;
- **nb_steps** - The total number of steps that will occur during a training session. A step is equivalent to one cycle of the RL loop, or in other words, 100 milliseconds of the ns-3 simulation, like previously explained in Chapter 3. The higher the number of steps, the longer the training session will be. The number of steps does not depend on the duration of the simulation. For example, if we define a small number of steps, but a very long simulation duration, the simulation will end when the number of steps ends and not the other way around. Therefore, it is important to pick values for the number of steps and for the simulation duration that make sense together;
- **nb_max_episode_steps** - Defines the maximum number of steps per episode. For example, if nb_max_episode_steps is set to 100 and nb_steps is set to 500, then the training session will last for five episodes. This parameter is useful to define how long an episode should be. As another example, if we want an episode to be a whole simulation, from beginning to end, and the simulation requires 3000 steps to run, then nb_max_episode_steps should be set to 3000. If we set this parameter to a value that is inferior to the number of steps that the simulation requires to finish, then the episode will end before the simulation has a chance to conclude, and a new episode and simulation will begin.

4.3.3 Neural network and hyperparameter tuning

The configuration of the neural network structure and the tuning of hyperparameters was one of the most time consuming tasks in this dissertation. The reason for this is that, in order to pinpoint

which neural network and hyperparameter combinations were the most efficient for this work, all of the planned options had to be trained and compared against each other. Considering that each training session lasts for several hours, the whole process ended up taking hundreds of hours to finish.

The amount of time required to test the different combinations was taken into account from the very beginning. Since there is an infinite number of possible combinations of neural network structures and hyperparameter configurations, and the time available to develop this thesis was quite limited, it was not feasible to attempt to optimally tune every hyperparameter available and to try many different neural network structures. Therefore, it was decided that the only hyperparameter to tune was the learning rate, with the others remaining fixed. The following list details every hyperparameter configured and their respective values:

- **Learning rate** - This hyperparameter was configured with three values: 1×10^{-2}; 1×10^{-3} and 5.5×10^{-3};
- **Discount factor** - This hyperparameter had a fixed value of 0.6. This value was picked as it is important that the agent has a slight preference for future rewards, but not in excess;
- **Replay buffer size** - The size of the replay buffer had a fixed value of 100000. This value could probably be smaller as to use less memory, but since it was supposed to remain fixed, a safer, larger value was instead chosen;
- **Batch size** - The batch size value was fixed at 64 samples, a commonly used value for this hyperparameter;
- ε**-greedy** - In the training sessions performed, this hyperparameter always started out with the value of 1 and linearly decayed all the way down to 0.1 by the end of the training session.

Having gone over the five hyperparameters considered, the next step was to define a series of neural network configurations and train each one of them. Considering that the learning rate hyperparameter took three different values, every neural network configuration was trained three times, one training session per learning rate value. A total of 11 neural networks were trained, which translates into 33 training sessions. The input and output layers were the same for every neural network, as they depend on the number of observations and the number of possible actions, respectively. For this reason, the input layer had one neuron, as the agent only makes one observation per cycle – the SNR – while the output layer had 58 neurons, as the action space has a total of 58 possible actions. It were the hidden layers that varied between the different neural networks. For simplicity, each neural network combination is identified as $n_1;n_2;...;n_i$, where n_X is the number of neurons at layer X. For example, 64;32;32 represents a neural network with three hidden layers. The first layer has 64 neurons, while the second and the third layers have 32 neurons each. This being said, the neural network configurations considered were the following: 32, 8;8, 16;16, 8;8;8, 16;16;16, 32;32, 64, 64;32, 64;64, 64;64;32 and 64;64;64. Implementing all these different networks was done by using functions provided by Keras.

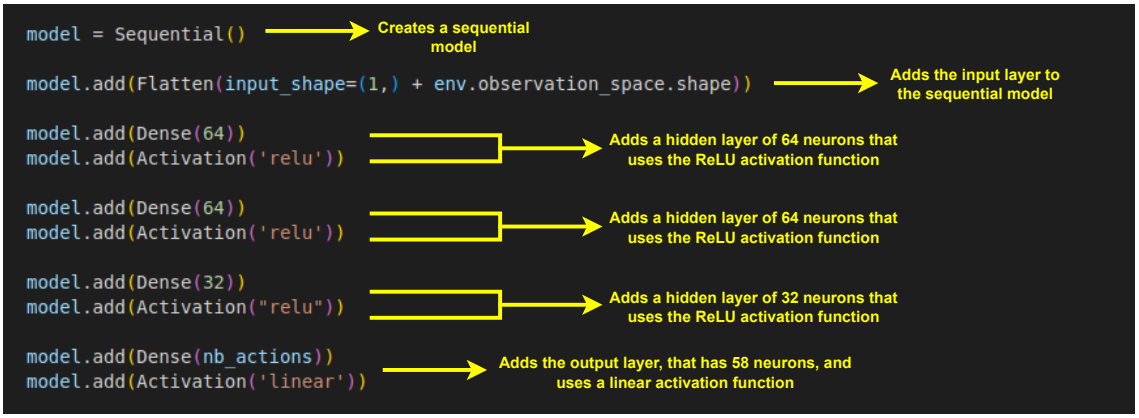

Figure 4.8: Example of a neural network creation with Keras.

Figure 4.8 is an example of a neural network creation with Keras. First, a Sequential model was instantiated. This is the most simple type of model that Keras provides, and allows us to create a straightforward network, layer by layer. Then, a network was built on top of this model. The network created has three hidden layers, two with 64 neurons and one with 32 neurons. Other than that, it has the input layer and the output layer. By using the *Dense()* function, we indicate that the layers are densely connected. This means that each neuron from a layer is connected to every neuron from the next layer. The hidden layers use the ReLU activation function. The reason for picking this function is because it is widely used in the DRL community, being highly recommended for most situations. Moreover, due to time limitations, we could not test the performance of other activation functions. The output layer uses the Linear activation function, as it is the most used output activation function for non-classification problems, such as this one. All the neural network combinations considered used the Sequential model, the ReLU activation function for the hidden layers and the Linear activation function for the output layer. The only thing that changed between combinations is the number of neurons and number of layers.

In order to evaluate how the different combinations of neural network configurations and learning rate values perform, two approaches were taken. The first approach consisted in saving the cumulative rewards that the agent obtained for each training episode and plotting graphs with these rewards, allowing us to get a visual representation on how effectively the agent was at learning with each configuration. The second approach consisted in testing the trained agents with the ε-greedy configured to zero. This means that the agent only take actions that it has learned, and no exploratory, random actions are taken. Both approaches used the *waypoint* scenario for the simulations. This means that all 33 training and testing sessions used this scenario.

4.3.3.1 First Approach - Cumulative Rewards Plots

For this first approach, three different graphs were plotted, one per learning rate value. Therefore, each graph has 11 cumulative reward plots, one for each neural network configuration. The rationale is to allow us to compare the learning performance of each neural network for a single learning rate, instead of trying to compare them all at once, which would be very confusing as

the graph would have 33 plots. It is also worth noting that each episode lasted for 300 simulation seconds.

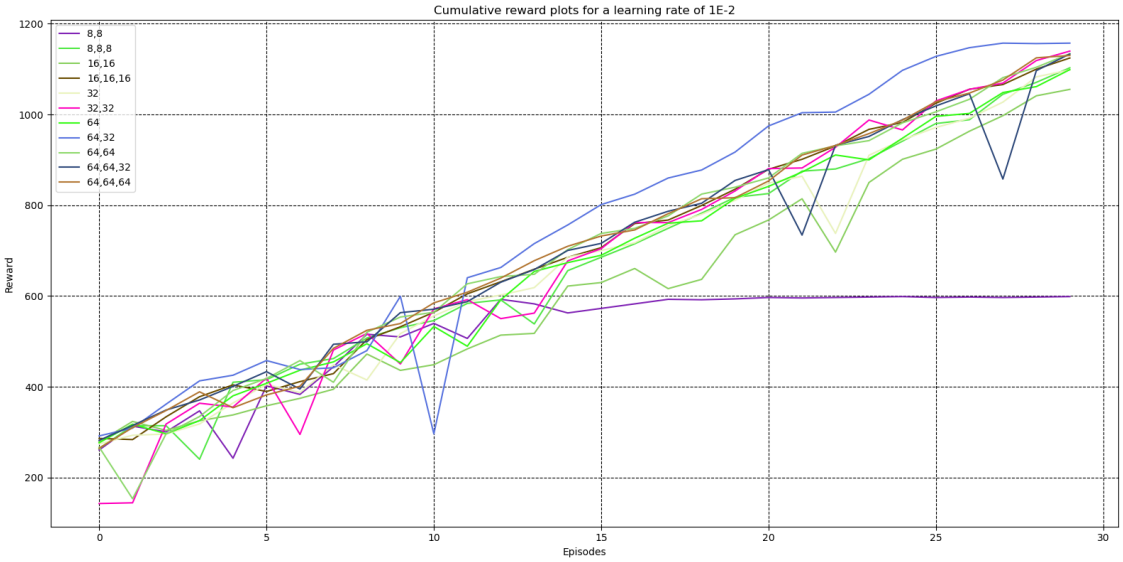

Figure 4.9: Cumulative reward plots for a learning rate of 1×10^{-2}.

Figure 4.9 contains the cumulative reward plots for each considered neural network after they were trained for 30 episodes with a learning rate of 1×10^{-2}. The reason for training for 30 episodes is that we empirically determined this to be a very good option, as less episodes were not enough for the agent to converge and more episodes did not prompt the agent to learn more than it did with just 30. By analyzing the different plots, we can see that they all start with low cumulative episode rewards. Overtime, the cumulative rewards keeps increasing. This is to be expected, because the agent keeps learning and the ε-greedy keeps decaying. In other words, the agent becomes more capable of adapting the link due to the knowledge gained and also takes less random actions because of the ε-greedy decay, choosing to use the actions it already learned. This leads to better rewards throughout later episodes and higher cumulative reward values. We can also tell that most neural network configurations performed very closely to each other and converged to similar values at the end of the training session, with some configurations being visibly slightly worse than others. Two very clear exceptions are the 8;8 and the 64;32 configurations. The former performed very poorly and converged very early to a cumulative reward of around 600 after the fifteenth episode, which means it became incapable of learning more after that episode. The latter performed much better than every other network, as it clearly learned more quickly than others throughout the whole process, as evidenced by the higher cumulative rewards for each episode. On top of that, it also converged on a higher value than the other networks. We can, therefore, conclude that the 64;32 network was the most efficient at training, for this particular learning rate.

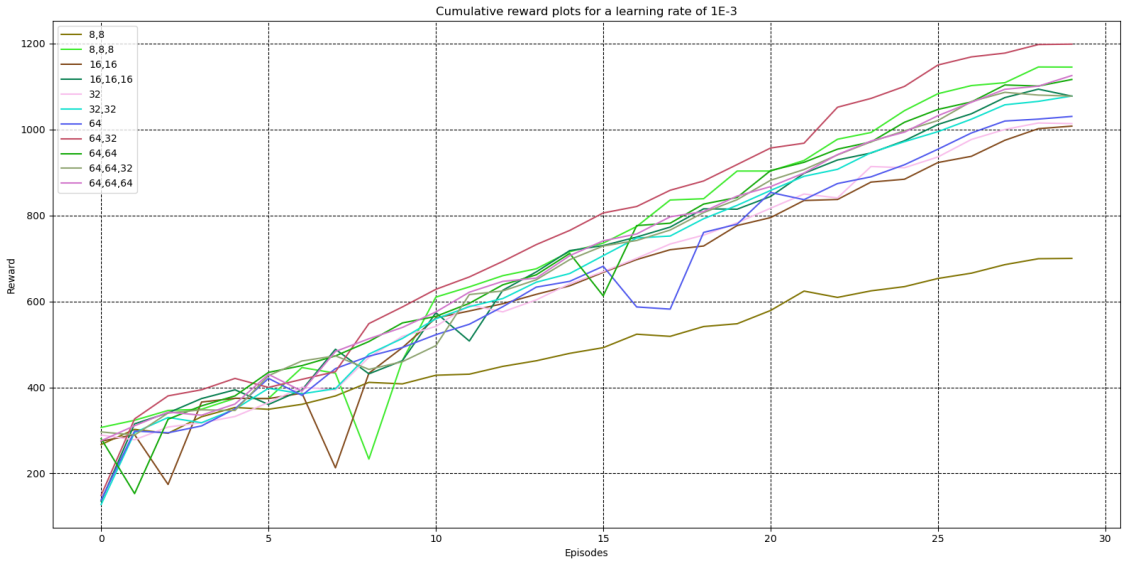

Figure 4.10: Cumulative reward plots for a learning rate of 1×10^{-3}.

Figure 4.10 contains the cumulative reward plots for each considered neural network after they were trained for 30 episodes with a learning rate of 1×10^{-3}. Right away, we can see that, with this learning rate, the different neural network combinations had more varied results. Unlike the previous case where a learning rate of 1×10^{-2} was used and most of the neural network combinations had similar performances, in this case, the various combinations had a more diverging learning experience, as the plot lines are more separated in general. Another noticeable difference is that most of the combinations end up converging at different end values, whereas they converged at very similar values for the learning rate of 1×10^{-2}. We can also see that the network of 8;8 is still the combination that performs the worse, as it converged to a value much smaller than the other networks. The main difference in relation to the previous graph, however, is that this network did not converge until the very end of the training session. Furthermore, the 64;32 combination is still the neural network that performed the best, converging to an end cumulative reward value of 1200 and having the overall best training performance.

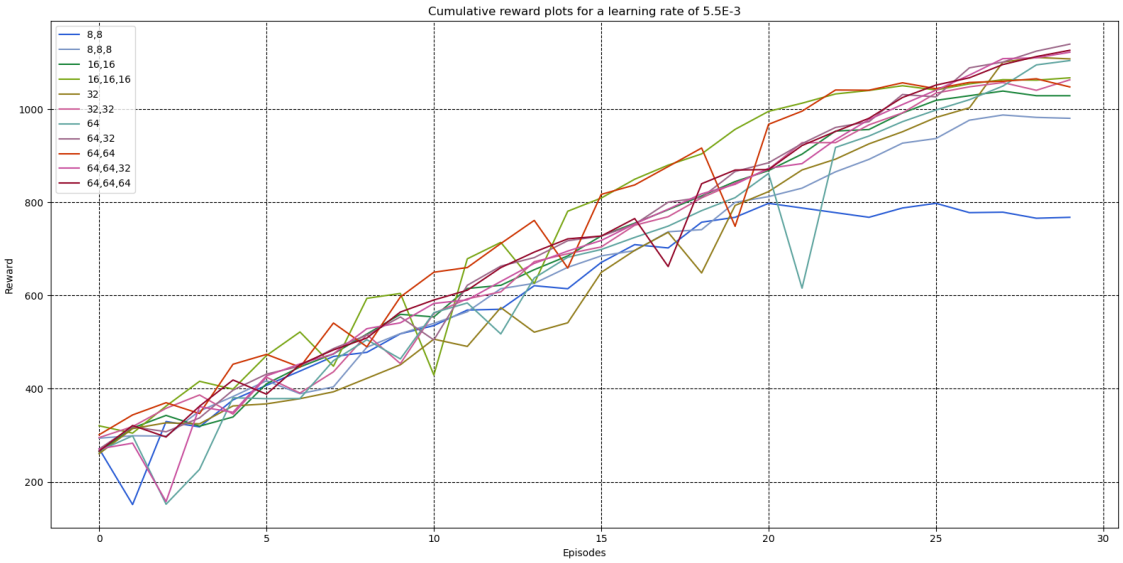

Figure 4.11: Cumulative reward plots for a learning rate of 5.5×10^{-3}.

Figure 4.11 contains the cumulative reward plots for each considered neural network after they were trained for 30 episodes with a learning rate of 5.5×10^{-3}. It seems that this learning rate caused the worst results so far, as most neural network combinations had a very unstable learning process, evidenced by the various downward peaks present in the plots. Furthermore, while for the other learning rates most neural networks converged to end values well above 1000, for this learning rate many of them converged to values just slightly above 1000. Not one combination came close to converging to an end value of 1200, something that was achieved for the learning rate of 1×10^{-3} and nearly achieved for the learning rate of 1×10^{-2}. The one thing that this learning rate had in common with the others is that the 8,8 neural network is still the worst performing one, although it performed better than in the other two cases, converging to an end value of 800, while the others converged at 600 and around 700.

4.3.3.2 Second Approach - Testing The Trained Agents

With all the 11 neural network combinations fully trained 3 times, one per learning rate value, we can technically say that we have 33 different DRL trained agents at our disposal. By analyzing the cumulative reward plots in the previous approach, we managed to get a good idea on which networks performed best at training. However, this is not enough, and the most conclusive evidence we can get of which neural network combination is best is to actually evaluate the trained agents to see how they perform with the knowledge they've gained. To this end, the *waypoint* scenario was used once again, instead this time, the ε-greedy parameter is set to zero from the very beginning, which means the agent will fully exploit, never taking random actions. Due to time constraints, these performance evaluations only ran for 5 episodes, or in other words, 5 simulations, with each simulation lasting for 300 seconds, just as before. This means that each neural network has 5

cumulative reward values per learning rate in the end. Instead of plotting lines with just 5 values, it was instead decided to calculate the mean value.

	1×10^{-2}	1×10^{-3}	5.5×10^{-3}
8;8	558	689	769
16;16	1049	1020	1076
8;8;8	1115	1120	998
16;16;16	1139	1092	1067
32	1110	1021	1158
32;32	1171	1095	1074
64	1083	1016	1132
64;32	1179	1204	1103
64;64	1101	1112	1100
64;64;32	1097	1098	1115
64;64;64	1101	1098	1109

Table 4.1: Evaluation mean cumulative reward values for the different neural network and learning rate combinations.

Table 4.1 contains the mean cumulative reward values for each combination of neural network and learning rate, after evaluation. This table allows us to observe the combinations that performed the best by pinpointing the ones that achieved a higher mean cumulative reward value. We can see that the absolute best combination is the 64;32 neural network with a learning rate of 1×10^{-3}, which obtained a mean cumulative reward of 1204.

4.3.3.3 Conclusions

After analyzing the results obtained in both approaches, we can draw some conclusions. First of all, having many hidden layers did not prove to be an important factor, as it was possible to obtain good results with networks that had just one or two hidden layers. This is further supported by the fact that the best performing neural network only has two hidden layers, even though there were networks with three hidden layers. However, if we are using only one or two hidden layers, then the number of neurons needs to be high, otherwise the agent will not be able to learn much. As an example of this, we have the 8;8 network, which was consistently the worst performing network for every learning rate. A small number of neurons can still work if there are at least three hidden layers, as evidenced by the 8;8;8 network. Furthermore, a high number of neurons combined with two or more hidden layers seems to be counterproductive, as evidenced by the 64;64, 64;64;32 and 64;64;64 networks, which only obtained average results. This leads us to conclude that, in the context of the work developed for this dissertation, neural networks with a higher complexity do not perform better, possibly due to overfitting, and there is no advantage in using them. This also works well in our favour, as denser networks are more computationally expensive. When it comes to the learning rate value, the difference in results was not as diverse as we initially expected. In fact, the results obtained with each learning rate were very similar for most neural networks, leading us to the conclusion that this hyperparameter could be further varied with a wider range of

values. To conclude, the 64;32 network with a learning rate of 1×10^{-3} got the best results, both during training and during evaluation. For this reason, this combination was selected.

4.4 Reward Function Tuning

The final step of the implementation process consisted in determining the ideal weight that each parameter should have on the reward function. As previously mentioned in Section 4.2, the reward function is defined in the C++ program that implements the ns3-gym interface.

```
if(this->nSuccess == 0) {
    reward=0;
}

else {
    float avgRate = (this->sumDataRate/1000000)/(this->nAttempts*maxRate);
    float avgCB = this->sumCB/(this->nAttempts*maxCB);
    float avgGI = 1/(this->sumGI/(this->nAttempts*minGI));
    float fsr = this->nSuccess/this->nAttempts;
    float sumRatio = (0.55*avgRate) + (0.35*avgCB) + (0.1*avgGI);
    reward = sumRatio*fsr;
```

Figure 4.12: Code snippet of the reward function.

Figure 4.12 contains a code snippet of the reward function implemented in the ns3-gym interface. This code implements the logic detailed in Section 3.1. The line of code inside the yellow box is where the weight of each parameter on the reward value is set. In this particular example, the chosen MCS has a weight of 55% on the calculated reward, while the CB and GI have weights of 35% and 10%, respectively. These percentages are important, as they control how much priority the agent gives to each parameter. A 55% weight for the MCS parameter means that the MCS is the parameter with the highest priority, as the agent will get larger rewards for choosing a good MCS value, even if the CB and GI values are not optimal. Likewise, a 35% weight for the CB means that it is the parameter with the second highest priority, and finally, the GI is the lowest priority parameter. Furthermore, these were the percentages used for every training session detailed in Section 4.3.3.

In order to determine the optimal percentages for each parameter, the exact same approaches detailed in Section 4.3.3 were used. This time, however, the neural network structure and learning rate remained fixed at 64;32 and 1×10^{-3}, as mentioned in Section 4.3.3.3. Only the weights were varied, with the exception of the GI weight, which remained fixed at 10%, as using a short guard interval has an impact of about that percentage on the throughput [42]. For simplicity, and considering that the GI weight is fixed, the weight combinations trained and evaluated are represented as $MCS_{percentage}$;$CB_{percentage}$. For example, 55;35 means that the weights are 55% for the MCS, 35% for the CB and, of course, 10% for the GI. The combinations defined were 50;40, 51;39, 52;38, 53;37, 54;36, 55;35, 56;34, 57;33, 58;32, 59;31 and 60:30.

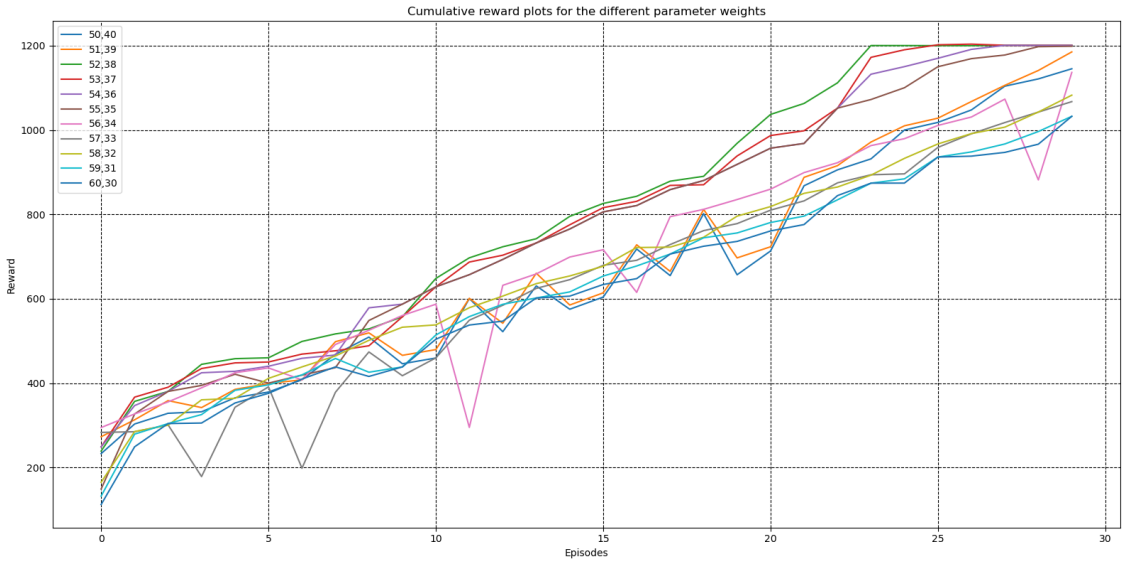

Figure 4.13: Cumulative reward plots for the different parameter weights.

Figure 4.13 contains the cumulative reward plots for each combination of MCS and CB weights. By observing the plots, we can see that the best combinations, in terms of learning performance, were 55;35, 54;36, 53;37 and 52;38, with the others performing considerably worse. The overall best was the 52;38 combination, or in other words, 52% for the MCS, 38% for the CB and 10% for the GI.

Having finished the training sessions, we then proceeded to evaluate the trained agents – again, just like in Section 4.3.3 – and register the mean cumulative reward values for each weight combination:

	Mean Cumulative Reward Value
50;40	1115
51;39	1179
52;38	1203
53;37	1198
54;36	1201
55;35	1199
56;34	1105
57;33	1092
58;32	1086
59;31	1006
60;30	1004

Table 4.2: Evaluation mean cumulative reward values for the different weight combinations.

By analyzing Table 4.2, we can see that the weight combinations that performed the best in evaluation were 52;38, 53;37, 54;36 and 55;35. These results are in tune with the plots observed in Figure 4.13, as these were the only combinations that managed to converge towards the 1200 cumulative reward value.

After finishing this final part of the implementation process and observing the results, we can conclude that tuning the weights of the parameters in the reward function did not provide a superior overall performance, as the agent seems to converge towards a maximum cumulative reward value of 1200, being unable to learn any more than that. The advantage that tuning the parameter weights brought was accelerating the learning process, or in other words, making the agent converge more quickly. This is evidenced by the 52;38 combination, which managed to converge considerably faster than the other combinations that also converged on the 1200 value. Taking this into account, we can also conclude that the agent learns more rapidly if the CB parameter has a bigger impact on the reward calculation, to a certain limit. It seems that the limit is 38%, as the 51;39 combination proved to already be considerably worse.

Chapter 5

Performance Evaluation

This chapter presents the simulation results obtained when evaluating the final version of the DRL-LA algorithm. DRL-LA was evaluated using the three scenarios described in Section 4.1, and for each simulation scenario, four main results were obtained: a graph that plotted the throughputs achieved throughout the simulation, the registered FSR values, a graph plotting the Cumulative Distribution Function (CDF), and finally, the average throughput values. Along with DRL-LA, these scenarios were also used to evaluate two other algorithms, Minstrel-HT and Ideal, as a way to compare the performances obtained by each. The chapter is divided into four sections, one section per simulation scenario and a section for conclusions.

5.1 *Waypoint* Scenario

For the *waypoint* scenario, two different simulations were executed. The first simulation is exactly like the one described in Section 4.1. Both the DRL-LA and the Minstrel-HT algorithms were evaluated using this simulation. However, due to an unidentified bug in ns-3, the Ideal algorithm crashes when running this simulation, as it cannot handle distances larger than 650 meters between the AP and the STA. For this reason, a second simulation was executed, which consists in the AP only moving until it reaches the 650 meter mark, instead of the original 1300 meters like in the first simulation. Therefore, the first simulation was used to compare the performance of DRL-LA algorithm against the performance of Minstrel-HT, while the second simulation was used to compare the performance of all three algorithms.

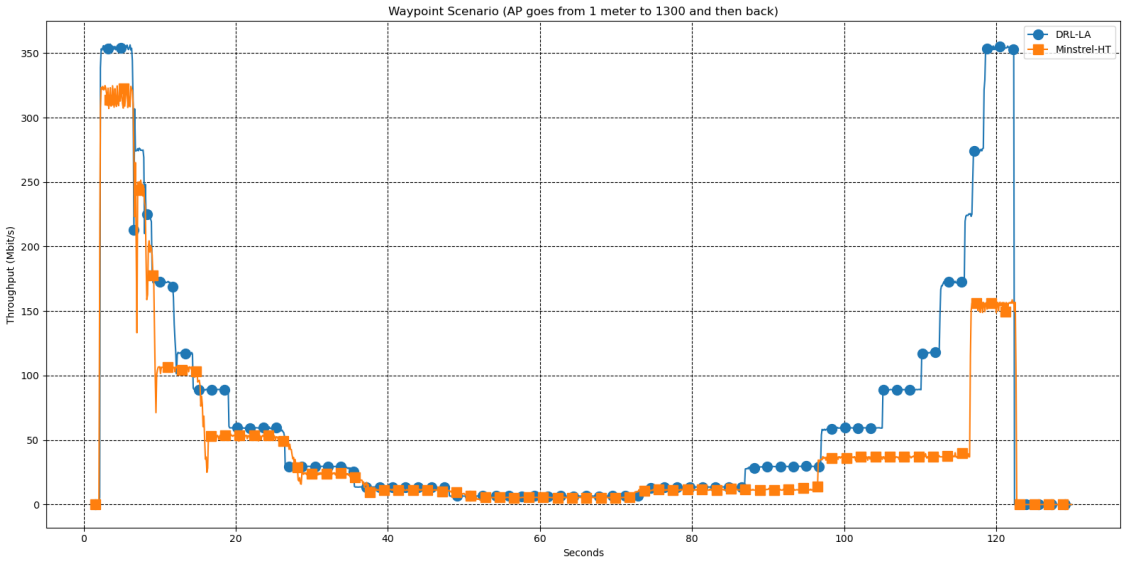

Figure 5.1: Throughputs for the first *waypoint* simulation, in which the AP goes from 1 to 1300 meters and back.

Figure 5.1 contains plots of the throughputs obtained by the DRL-LA and Minstrel-HT algorithms throughout the original *waypoint* scenario simulation. It is clear to observe that DRL-LA performed better than Minstrel-HT, as it was able to generate higher throughputs and adapt more efficiently to the SNR variations due to the increase and decrease in the distance between the AP and the STA.

When it comes to the FSR that each algorithm obtained for this simulation, DRL-LA registered a value of 99.7%, while Minstrel-HT registered a value of 93.14%. This essentially means that less frames were lost when using DRL-LA than when using Minstrel-HT.

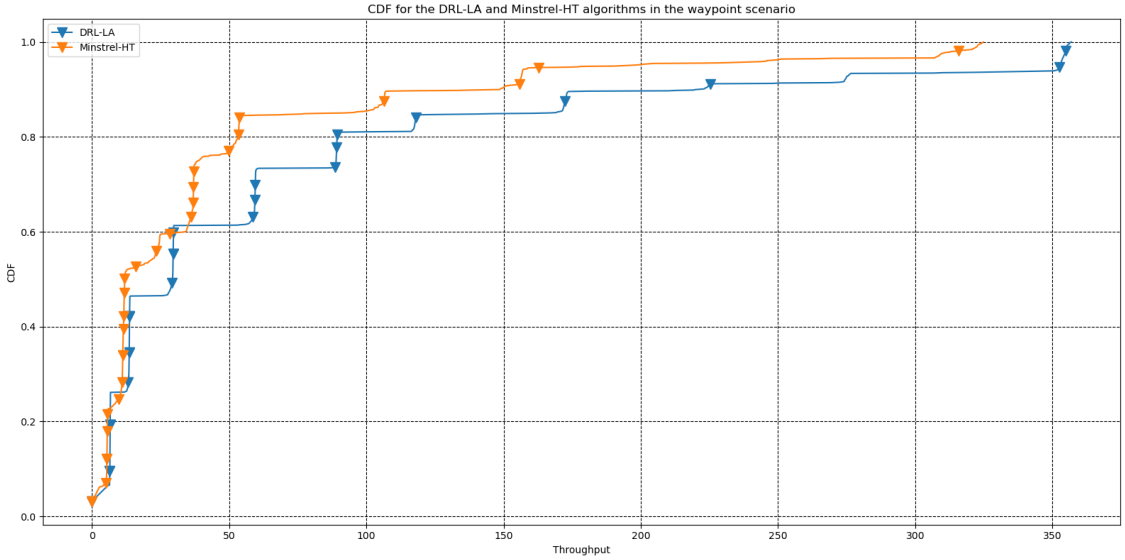

Figure 5.2: CDF for the DRL-LA and Minstrel-HT algorithms for the original *waypoint* scenario.

Figure 5.2 represents the CDF plots for both the DRL-LA and Minstrel-HT algorithms for the original waypoint simulation. By observing the figure, we can clearly say that DRL-LA outperformed Minstrel-HT. We can especially see that, at the 90th percentile, there is a significant difference of around 115 Mbit/s between both algorithms. Furthermore, the average throughput obtained by each algorithm throughout the whole simulation was 67.8 Mbit/s for DRL-LA and 44.4 Mbit/s for Minstrel-HT, a gain of more than 50%.

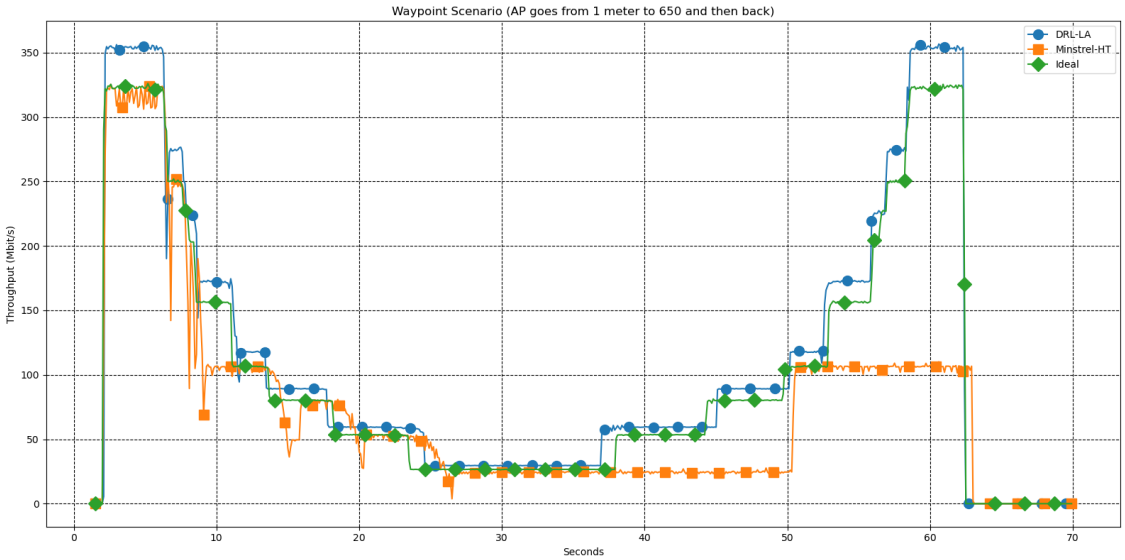

Figure 5.3: Throughputs for the second *waypoint* simulation, in which the AP goes from 1 to 650 meters and back.

Figure 5.3 contains plots of the throughputs obtained by the DRL-LA, Minstrel-HT and Ideal algorithms throughout the alternative *waypoint* scenario simulation, in which the AP only reaches a maximum distance of 650 meters. With this graph, we can compare the performances of the DRL-LA and Ideal algorithms. Furthermore, the Minstrel-HT algorithm was also included in order to be able to compare all three algorithms at the same time. We can conclude that DRL-LA was still the overall best performing algorithm, reaching the highest throughputs and being able to adapt efficiently.

When it comes to the FSR that each algorithm obtained for this simulation, DRL-LA registered a value of 99.8%, while Minstrel-HT and Ideal registered values of 94.4% and 99.9%, respectively. This means that, even though DRL-LA managed to get the highest throughputs, the Ideal algorithm was still slightly better when it comes to avoiding frame losses, being able to quickly adapt almost perfectly to SNR variations.

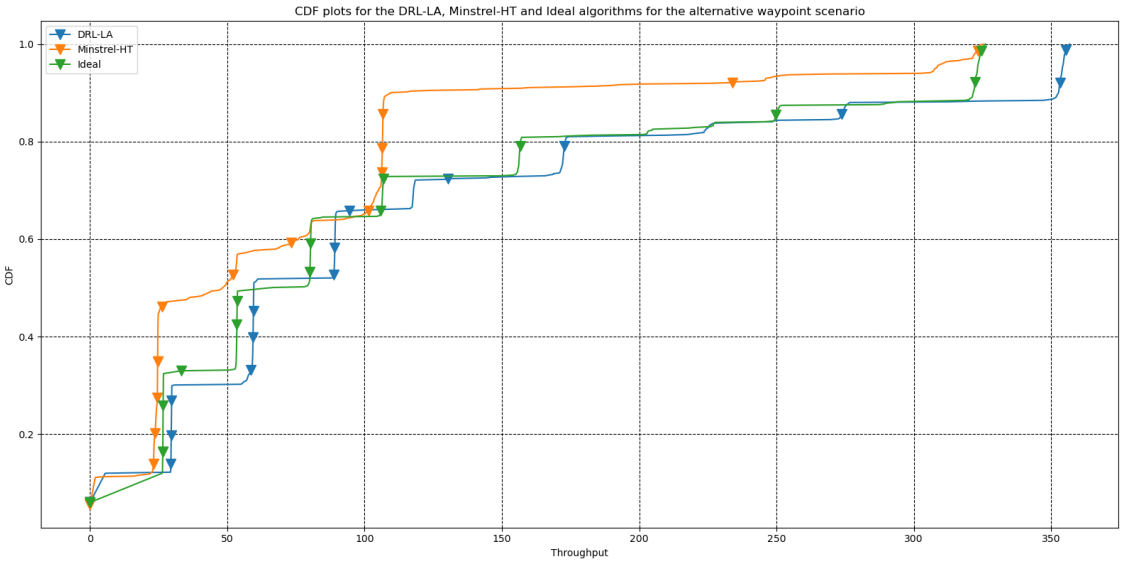

Figure 5.4: CDF for the DRL-LA, Ideal and Minstrel-HT algorithms for the alternative *waypoint* scenario.

Figure 5.4 represents the CDF plots for the DRL-LA, Ideal and Minstrel-HT algorithms for the alternative *waypoint* simulation. Once again, we can observe that, in general, DRL-LA outperformed the other two. At the 90th percentile, DRL-LA had a massive difference of 250 Mbit/s in relation to Minstrel-HT, and a difference of 35 Mbit/s in relation to Ideal. Furthermore, the average throughput obtained by each algorithm throughout the whole simulation was 113 Mbit/s for DRL-LA, 103.2 Mbit/s for Ideal and 73.46 Mbit/s for Minstrel-HT.

5.2 *Teleporting Node* Scenario

For the *teleporting node*, only the original scenario described in Section 4.1 was necessary, as the Ideal algorithm does not crash with this scenario, unlike the previous case with the *waypoint* scenario. However, two simulations were still executed. The reason for this is that, while the first simulation aims to compare the DRL-LA, Ideal and Minstrel-HT algorithms, the purpose of the second simulation is to compare DRL-LA with the backup mechanism enabled versus DRL-LA with the backup mechanism disabled. This allows us to observe the effectiveness of the backup mechanism and the gains it introduced to the overall solution.

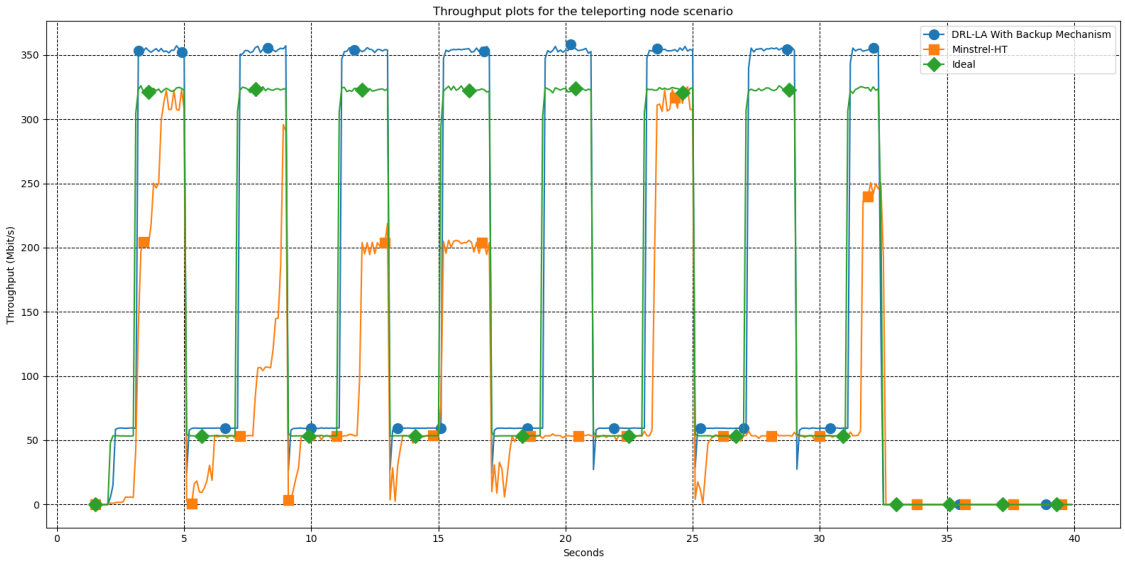

Figure 5.5: Throughputs for the *teleporting node* scenario.

Figure 5.5 contains plots of the throughputs obtained by the DRL-LA, Ideal and Minstrel-HT algorithms throughout the *teleporting node* scenario simulation. We can observe that, as expected, DRL-LA managed to achieve the highest throughputs. However, the most interesting aspect to observe with this scenario is the capability that the algorithm has to adapt to sudden SNR variations. We can observe that Minstrel-HT performed the worst, never being able to quickly and fully adapt to these sudden variations, even reaching a momentary throughput of 0 Mbit/s at some points. DRL-LA, on the other hand, performed very efficiently, adapting quickly to the SNR variations and avoiding low, or even null, throughput values thanks to the backup mechanism implemented. At last, the Ideal algorithm had a basically flawless performance when it comes to adapting to the variations. However, it is interesting to note that, excluding the slight downward peaks in the throughput that occur when the SNR value suddenly changes, DRL-LA was extremely similar to the Ideal algorithm, in terms of performance, when adapting to the new SNR value.

For this simulation, DRL-LA registered a FSR of 99.8%, while the Ideal and Minstrel-HT registered FSR values of 99.9% and 89%, respectively. Once again, DRL-LA managed to have an FSR considerably higher than Minstrel-HT, with Ideal still coming out on top by 0.01%.

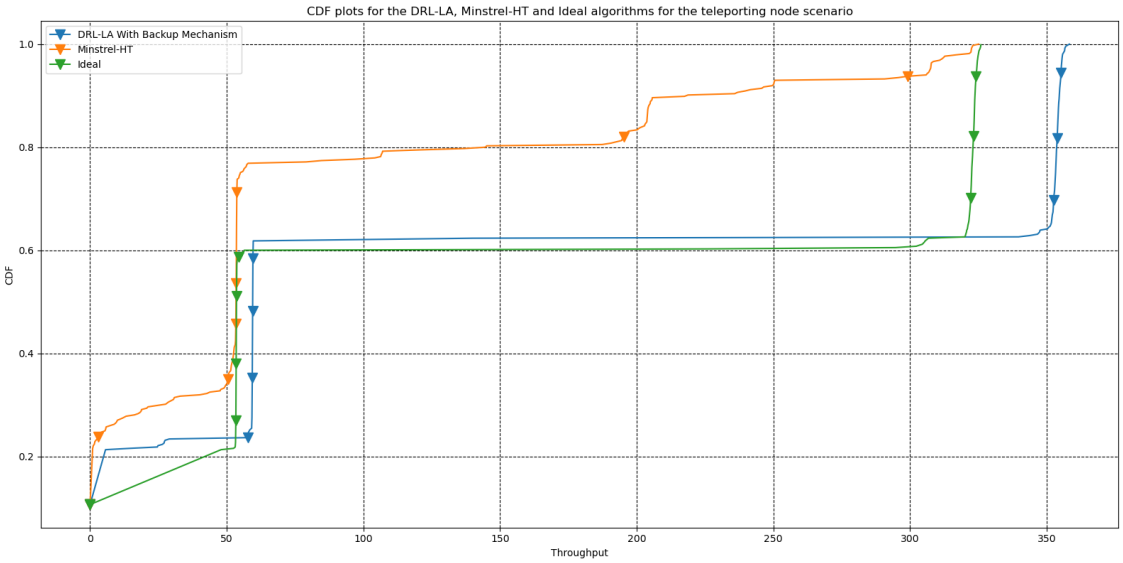

Figure 5.6: CDF for the DRL-LA, Ideal and Minstrel-HT algorithms for the *teleporting node* scenario.

Figure 5.6 represents the CDF plots for the DRL-LA, Ideal and Minstrel-HT algorithms for the *teleporting node* scenario. We can observe that Minstrel-HT performed very poorly in comparison to DRL-LA and Ideal, which performed very similarly for the most part, with DRL-LA being able to reach higher throughputs. At the 90th percentile, DRL-LA had a difference of 150 Mbit/s in relation to Minstrel-HT, and a difference of 35 Mbit/s in relation to Ideal. The average throughput obtained by each algorithm throughout the whole simulation was 157.4 Mbit/s for DRL-LA, 149.4 Mbit/s for Ideal, and 77.8 Mbit/s for Minstrel-HT.

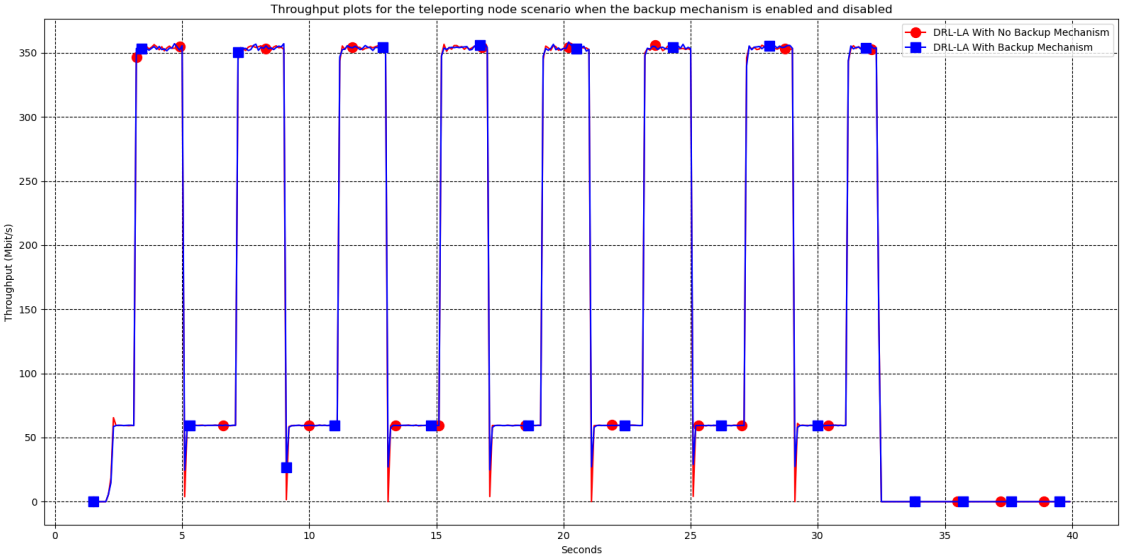

Figure 5.7: Comparison of the throughput plots obtained using DRL-LA with the backup mechanism enabled and disabled.

Figure 5.7 contains plots of the throughputs obtained by DRL-LA throughout the *teleporting node* scenario simulation, with and without the backup mechanism enabled. As expected, the overall performance was basically the same and the plots overlap. The main differences between the two versions of DRL-LA can be observed at the instants where the SNR changes drastically, as this is the type of situation where the backup mechanism is used. We can clearly observe that the version without the backup mechanism reaches a throughput of 0 Mbit/s during the transitions, before the agent has a chance to get another observation and reconfigure the parameters accordingly. The version with the backup mechanism manages to avoid this situation and the throughput never reaches zero, even though it still drops quite a bit in the transitions. To further compare the two, we can look at the FSR values that each version of DRL-LA registered. The version with the backup mechanism registered a value of 99.8%, while the version without the backup mechanism registered a value of 96.6%.

5.3 *Waypoint With Teleports* Scenario

The final scenario simulated is the *waypoint with teleports* scenario, also described in Chapter 4.1. This scenario had the same problem as the *waypoint* scenario when it comes to simulating it with Ideal, as this algorithm crashes. However, no additional alternative simulation was elaborated, as this time, the problem could not be solved by simply not allowing the AP node to move farther away than 650 meters, for unknown reasons. Considering that coming up with a solution for this problem was taking too long, it was decided that DRL-LA would only be compared to Minstrel-HT for this scenario. Furthermore, and just like in the *teleporting node* scenario, the two versions of DRL-LA – backup mechanism enabled and backup mechanism disabled – were once again compared against each other.

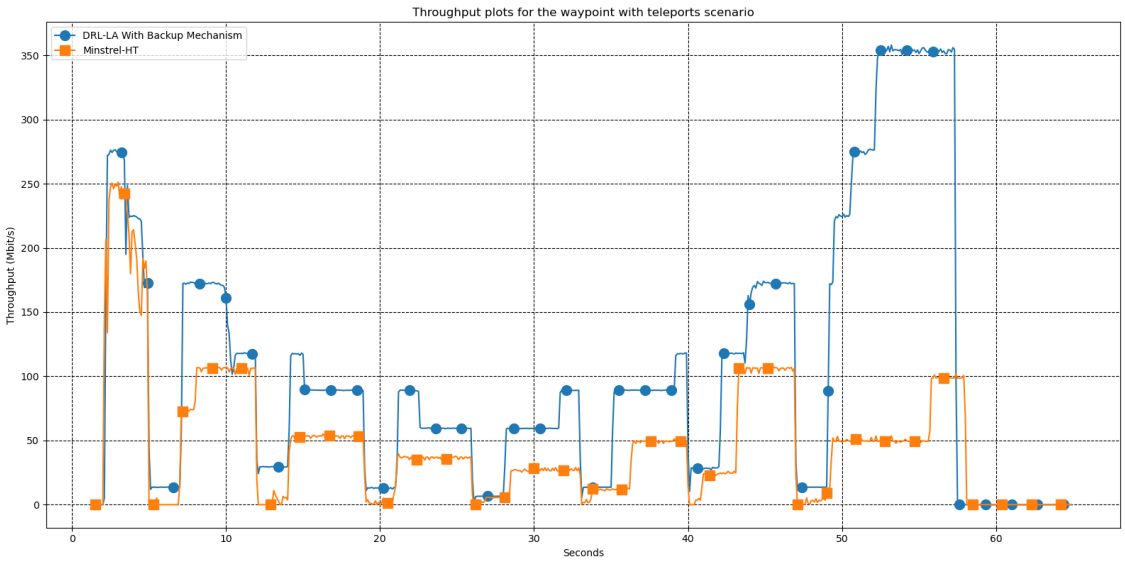

Figure 5.8: Throughputs for the *waypoint with teleports* scenario.

Figure 5.8 contains the plots of the throughputs obtained by the DRL-LA and Minstrel-HT algorithms throughout the *waypoint with teleports* scenario simulation. Once again, DRL-LA managed to reach higher throughputs. Furthermore, due to the backup mechanism, it managed to avoid null throughputs during the sudden transitions and adapt quickly, unlike Minstrel-HT. For this simulation, DRL-LA registered a FSR of 99.6%, while Minstrel-HT registered a FSR of 92.1%.

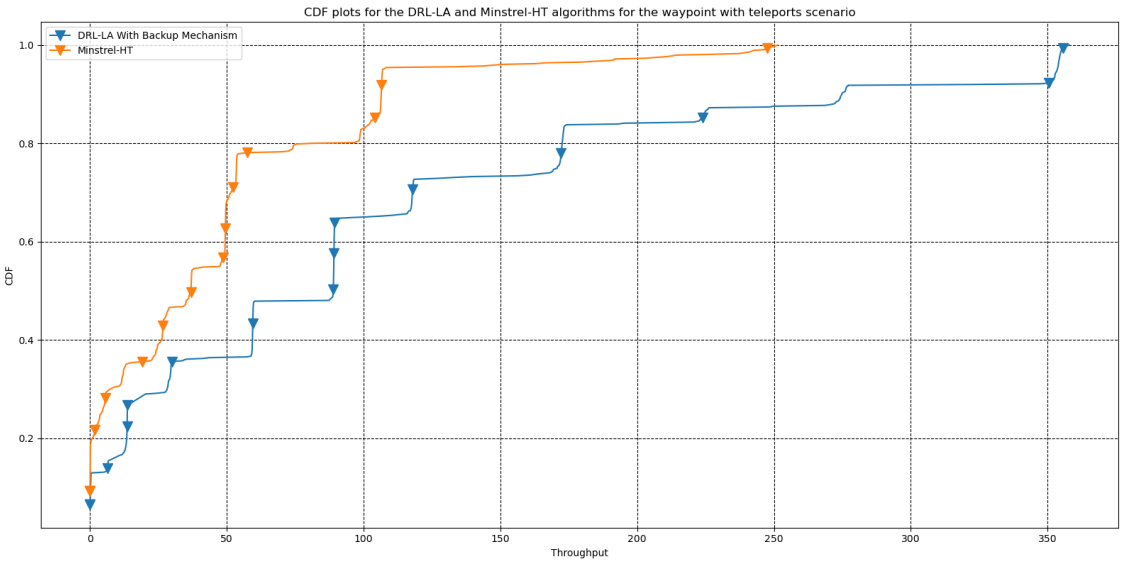

Figure 5.9: CDF for the DRL-LA and Minstrel-HT algorithms for the *waypoint with teleports* scenario.

Figure 5.9 represents the CDF plots for the DRL-LA and Minstrel-HT algorithms for the *waypoint with teleports* scenario. It is clear to see that DRL-LA largely outperforms Minstrel-HT. At the 90th percentile, there is a difference of 235 Mbit/s. Furthermore, the average throughput obtained by each algorithm throughout the whole simulation was 103.1 Mbit/s for DRL-LA and 45.9 Mbit/s for Minstrel-HT, a gain of more than double.

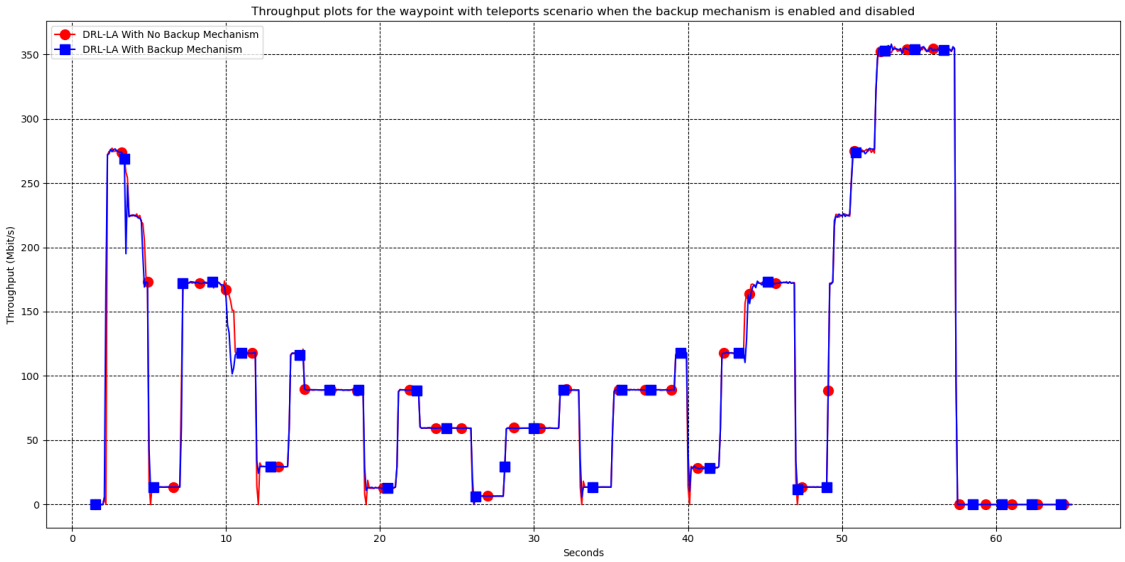

Figure 5.10: Comparison of the throughputs obtained using DRL-LA with the backup mechanism enabled and disabled in the *waypoint with teleports* scenario.

Figure 5.10 contains the plots of the throughputs obtained by DRL-LA throughout the *waypoint with teleports* simulation, with and without the backup mechanism enabled. Just like in the *teleporting node* scenario, both versions of DRL-LA performed basically the same, with the exception of the instants when the SNR value changed suddenly and drastically. We can see that the version with the backup mechanism disabled obtains null throughputs in these instants, whereas the version with the backup mechanism enabled manages to avoid reaching null values. Furthermore, the version with the backup mechanism enabled registered a FSR value of 99,6%, while the version with the backup mechanism disabled registered a FSR value of 97.9%.

5.4 Conclusions

After running every planned simulation and observing the results, we can conclude that DRL-LA greatly outperforms the Minstrel-HT algorithm, both in terms of the throughputs that are obtained and in terms of how fast it can adapt to SNR variations. Furthermore, it even manages to outperform the Ideal algorithm in terms of the throughput obtained, even though Ideal is slightly better at adapting to SNR variations, being able to do it instantly. However, the fact that the Ideal algorithm performed a bit better in this aspect is not a significant result, as this algorithm, like the name suggests, attempts to ideally adapt the link by using mechanisms that could never be implemented in real life. Nonetheless, it is an interesting comparison to make. Finally, the simulations made with the backup mechanism enabled and disabled allowed us to more clearly visualize its impact when it comes to adapting the link. Even though using the mechanism did not translate into considerable gains on the registered FSR values, it is still a useful feature to prevent the throughput from plunging to null values during sudden transitions.

Chapter 6

Conclusion and Future Work

The usage of WLANs is one of the most common practices today. These networks have been popularized in domestic, educational, commercial and corporative environments, allowing for multiple devices to communicate with each other and also access the Internet. They are implemented using the IEEE 802.11 set of technical standards, commercially known by the brand name Wi-Fi. Over the years, various link adaptation algorithms have been developed with the objective of optimally adapting Wi-Fi connections, in order to provide users with high throughputs. However, with the introduction of the latest IEEE 802.11 standards and their enhancements, these algorithms have shown to be outdated and unable to properly adapt the wireless links, as they do not fully utilize the capabilities of the most recent standards. This creates a demand for better solutions when it comes to link adaptation algorithms.

This thesis had the goal of developing a novel link adaptation algorithm using DRL. The objectives defined for this work were successfully accomplished, as we were able to create a capable algorithm that met the validation criteria. The simulations executed have shown that the DRL-LA algorithm is able to properly adapt the link by optimally configuring the MCS, GI and CB parameters. Not only is the algorithm able to reach very high throughputs when the radio channel conditions allow for it, but it is also very effective in adapting to a poor signal quality. Furthermore, the backup mechanism implemented provides DRL-LA with an additional resilience to sudden variations in the radio channel conditions, guaranteeing that the the throughput never drops to a null value.

Even though the objectives defined for this dissertation were accomplished, there are still a myriad of opportunities that can be explored. First of all, the DRL-LA algorithm was tested with very simple scenarios, in which the STA node only moved in one dimension. Furthermore, only two network nodes were present for every scenario, the AP and the STA. This means that we did not test how the algorithm performs when there are factors like interference between nodes. Also, the Friis Free Space Propagation Model used for the simulations is a very simplistic, line-of-sight path loss model, meaning that other, more complex models should be used for testing; nevertheless, we do not envision a significant impact on the algorithm performance as it showed good performance in unstable scenarios.

Future work would first involve creating scenarios that are a lot more complex and close to real life situations. These scenarios would require a much deeper comprehension of the ns-3 software in order to utilize all of its capabilities to produce these simulations. By testing the DRL-LA algorithm with realistic scenarios, we could then pinpoint its major weaknesses and investigate ways to overcome them, obtaining an even more robust algorithm. The ultimate goal after extensively validating the algorithm in simulation would be to port it to real nodes and test it in real world scenarios, making the necessary adjustments.

References

[1] Ibrahim Sammour and Gerard Chalhoub. Evaluation of Rate Adaptation Algorithms in IEEE 802.11 Networks. *Electronics*, 9(9):1436, September 2020. doi:10.3390/electronics9091436.

[2] Raja Karmakar, Samiran Chattopadhyay, and Sandip Chakraborty. SmartLA: Reinforcement learning-based link adaptation for high throughput wireless access networks. *Computer Communications*, 110:1–25, September 2017. doi:10.1016/j.comcom.2017.05.017.

[3] Syuan-Cheng Chen, Chi-Yu Li, and Chui-Hao Chiu. An Experience Driven Design for IEEE 802.11ac Rate Adaptation based on Reinforcement Learning. In *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications*, pages 1–10, Vancouver, BC, Canada, May 2021. IEEE. doi:10.1109/INFOCOM42981.2021.9488876.

[4] IEEE 802.11, The Working Group Setting the Standards for Wireless LANs. [Online; accessed 15. Nov. 2021]. Available from: https://www.ieee802.org/11/.

[5] Intel. (2020) iwlwifi linux wireless. [Online; accessed 13. Mar. 2022]. Available from: https://wireless.wiki.kernel.org/en/users/drivers/iwlwifi.

[6] Minstrel-ht: New rate control module for 802.11n. [Online; accessed 28. Oct. 2021]. Available from: https://lwn.net/Articles/376765/.

[7] Seongho Byeon, Kangjin Yoon, Changmok Yang, and Sunghyun Choi. STRALE: Mobility-aware PHY rate and frame aggregation length adaptation in WLANs. In *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*, pages 1–9, Atlanta, GA, USA, May 2017. IEEE. doi:10.1109/INFOCOM.2017.8056965.

[8] Ioannis Selinis, KoFnstantinos Katsaros, Seiamak Vahid, and Rahim Tafazolli. Damysus: A Practical IEEE 802.11ax BSS Color Aware Rate Control Algorithm. *International Journal of Wireless Information Networks*, 26(4):285–307, December 2019. doi:10.1007/s10776-019-00439-6.

[9] Teuku Yuliar Arif. Evaluation of the Minstrel-HT Rate Adaptation Algorithm in IEEE 802.11n WLANs. *International journal of simulation: systems, science & technology*, March 2017. doi:10.5013/IJSSST.a.18.01.11.

[10] Issam El Naqa and Martin J. Murphy. What Is Machine Learning? In Issam El Naqa, Ruijiang Li, and Martin J. Murphy, editors, *Machine Learning in Radiation Oncology*, pages 3–11. Springer International Publishing, Cham, 2015. doi:10.1007/978-3-319-18305-3_1.

[11] Pádraig Cunningham, Matthieu Cord, and Sarah Jane Delany. Supervised learning. In Matthieu Cord and Pádraig Cunningham, editors, *Machine Learning Techniques for Multimedia: Case Studies on Organization and Retrieval*, pages 21–49. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. doi:10.1007/978-3-540-75171-7âĆĆ.

[12] Zoubin Ghahramani. Unsupervised learning. In Olivier Bousquet, Ulrike von Luxburg, and Gunnar Rätsch, editors, *Advanced Lectures on Machine Learning: ML Summer Schools 2003, Canberra, Australia, February 2 - 14, 2003, Tübingen, Germany, August 4 - 16, 2003, Revised Lectures*, pages 72–112. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. doi:10.1007/978-3-540-28650-9âĆĚ.

[13] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Adaptive Computation and Machine Learning Series. The MIT Press, Cambridge, Massachusetts, second edition edition, 2018.

[14] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, May 2015. doi:10.1038/nature14539.

[15] Massimo Merenda, Carlo Porcaro, and Demetrio Iero. Edge Machine Learning for AI-Enabled IoT Devices: A Review. *Sensors*, 20(9):2533, April 2020. doi:10.3390/s20092533.

[16] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. Deep Reinforcement Learning: A Brief Survey. *IEEE Signal Processing Magazine*, 34(6):26–38, November 2017. doi:10.1109/MSP.2017.2743240.

[17] Arun Addagatla. Investigating Underfitting and Overfitting. [Online. accessed 13. Dec. 2021]. Available from https://medium.com/geekculture/investigating-underfitting-and-overfitting-70382835e45c, April 2021.

[18] Siddharth Sharma, Simone Sharma, and Anidhya Athaiya. ACTIVATION FUNCTIONS IN NEURAL NETWORKS. [Online. accessed 14. Mar. 2022]. *International Journal of Engineering Applied Sciences and Technology*, 04(12):310–316, May 2020. URL: https://www.ijeast.com/papers/310-316,Tesma412,IJEAST.pdf, doi:10.33564/IJEAST.2020.v04i12.054.

[19] AlphaGo. [Online. accessed 18. Dec. 2021]. Available from: https://www.deepmind.com/research/highlighted-research/alphago.

[20] AlphaZero: Shedding new light on chess, shogi, and Go. [Online. accessed 18. Dec. 2021]. Available from: https://www.deepmind.com/blog/alphazero-shedding-new-light-on-chess-shogi-and-go.

[21] Part 2: Kinds of RL Algorithms — Spinning Up documentation. [Online. accessed 22. Mar. 2022]. Available from: https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html.

[22] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through

deep reinforcement learning. *Nature*, 518(7540):529–533, February 2015. doi:10.1038/nature14236.

[23] Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3-4):279–292, May 1992. doi:10.1007/BF00992698.

[24] Reinforcement Learning With (Deep) Q-Learning Explained. [Online. accessed 19. Mar. 2022]. URL: https://www.assemblyai.com/blog/reinforcement-learning-with-deep-q-learning-explained/.

[25] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. [Online. accessed 10. Dec. 2021]. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 1861–1870. PMLR, 10–15 Jul 2018. URL: https://proceedings.mlr.press/v80/haarnoja18b.html.

[26] Marc G. Bellemare, Will Dabney, and Rémi Munos. A distributional perspective on reinforcement learning. [Online. accessed 16. Mar 2022]. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 449–458. PMLR, 06–11 Aug 2017. URL: https://proceedings.mlr.press/v70/bellemare17a.html.

[27] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. [Online. accessed 16. Mar. 2022]. In Francis Bach and David Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 1889–1897, Lille, France, 07–09 Jul 2015. PMLR. URL: https://proceedings.mlr.press/v37/schulman15.html.

[28] MCS Table and How To Use it. [Online. accessed 17. Mar. 2022]. Available from https://wlanprofessionals.com/mcs-table-and-how-to-use-it/.

[29] TensorFlow. [Online. accessed 18. Dec. 2021]. Available from: https://www.tensorflow.org/.

[30] Keras: The Python deep learning API. [Online. accessed 18. Dec. 2021]. Available from: https://keras.io/.

[31] Shogun Machine Learning - Home. [Online. accessed 17. Nov. 2021]. Available from: https://www.shogun-toolbox.org/.

[32] TensorFlow Agents. [Online. accessed 17. Nov. 2021]. Available from: https://www.tensorflow.org/agents.

[33] Taylor McNally. taylormcnally/keras-rl2. [Online. accessed 17. Nov. 2021]. Available from: https://github.com/taylormcnally/keras-rl2, May 2022. original-date: 2019-05-21T06:23:09Z.

[34] nsnam. Ns-3. [Online. accessed 18. Dec. 2021]. Available from: https://www.nsnam.org/.

[35] Ns-3 App Store - ns3-gym: OpenAI Gym integration. [Online. accessed 18. Dec. 2021]. Available from: https://apps.nsnam.org/app/ns3-gym/.

[36] Gym Documentation. [Online. accessed 17. Nov. 2021]. Available from: https://www.gymlibrary.ml/.

[37] Piotr Gawłowicz and Anatolij Zubow. ns-3 meets OpenAI Gym: The Playground for Machine Learning in Networking Research. In *Proceedings of the 22nd International ACM Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems - MSWIM '19*, pages 113–120, Miami Beach, FL, USA, 2019. ACM Press. URL: http://dl.acm.org/citation.cfm?doid=3345768.3355908, doi:10.1145/3345768.3355908.

[38] CUDA Toolkit - Free Tools and Training. [Online. accessed 17. Nov. 2021]. Available from: https://developer.nvidia.com/cuda-toolkit, July 2013.

[39] ns-3: ns3::IdealWifiManager Class Reference. [Online. accessed 13. Apr. 2022]. Available from https://www.nsnam.org/doxygen/classns3_1_1_ideal_wifi_manager.html#details.

[40] Guangyu Pei and Thomas R Henderson. Validation of OFDM model in ns-3. page 5.

[41] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization, January 2017. Number: arXiv:1412.6980 arXiv:1412.6980 [cs]. URL: http://arxiv.org/abs/1412.6980.

[42] Wireless basic configuration: Short Guard Interval and multipath effect FAQ. [Online. accessed 23. Oct. 2021]. Available from https://www.sonicwall.com/support/knowledge-base/wireless-basic-configuration-\protect\@normalcr\relax-short-guard-interval-and-multipath-effect-faq/170504672960493.