FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Analysis and Optimisation of Computational Delays in Reinforcement Learning-based Wi-Fi Rate Adaptation

Ricardo Jorge Espírito Santo Trancoso

Mestrado em Engenharia Eletrotécnica e de Computadores

Supervisor: Helder Martins Fontes

October 13, 2022

Abstract

Wi-Fi is a wireless communication technology working on the basis of radio waves, that has steadily become more common in day-to-day use. It is present in a wide variety of appliances, since it allows local connectivity between devices.

The *IEEE 802.11* standard was developed to implement Wi-Fi technology. Recent amendments to the standard introduced multiple configurable parameters. Among them, are several Modulation and Coding Schemes (MCS), which allow the network to better adapt to its environment. However, the environment is rarely ever static, so dynamic Rate Adaptation (RA) was developed. Dynamic RA is the act of dynamically adjusting the MCS depending on current wireless link quality. This can decrease repeated data transfers from errors, in turn resulting in a higher throughput.

Some algorithms have been proposed to allow a network to continuously adapt to its link quality. One option that has gathered a lot of interest recently, is Reinforcement Learning (RL). Although there is an increasing amount of research in RL applied to dynamic link quality problems, not a lot of it was conducted in experimental scenarios. Similarly, we could not find much information about the influence of computational delays on RL-based RA algorithms.

Computational delays not only slow down the algorithm's speed, but may also further impact them if it affects their responsiveness to changes in link quality. Outdated information of link quality can impact its choice of rates, resulting in degrading network throughput.

In this dissertation we analyse a preexisting dynamic RA algorithm implemented through RL called Data-driven Algorithm for Rate Adaptation (DARA). This algorithm faced some issues when deployed in an experimental scenario. We have made changes and new contributions to this algorithm that reduce its computational delays and enable its use in an experimental scenario. Our analysis revealed some insights on the causes of computational delays and ways to address them. Our implementation resulted in an algorithm around seven times faster than its predecessor.

Resumo

Wi-Fi é uma tecnologia de comunicação de redes sem fios que funciona à base de ondas rádio, cujo uso tem vindo a ser mais comum no dia-a-dia. Está presente numa grande variedade de dispositivos, pois permite uma ligação local entre eles.

A norma *IEEE 802.11* foi desenvolvida para implementar a tecnologia Wi-Fi. Alterações recentes à norma introduziram vários parâmetros configuráveis. Entre eles, estão os *Modulation and Coding Schemes* (MCS), que permitem à rede adaptar-se melhor ao seu ambiente. No entanto, esse ambiente raramente é estático. Por isso, desenvolveu-se *Rate Adaptation* (RA) dinâmica. RA dinâmica é o ato de ajustar dinamicamente o MCS dependendo da qualidade da ligação sem fios atual. Isto pode diminuir as retransmissões devido a erros, levando a um aumento do débito.

Alguns algoritmos foram propostos para que uma rede se adapte continuamente à sua qualidade de ligação. Uma opção que tem sido alvo de interesse recentemente, é *Reinforcement Learning* (RL). Ainda que a pesquisa de RL aplicada a problemas de qualidade dinâmica de ligação tenha vindo a aumentar, não encontramos muita investigação feita em ambientes experimentais. De forma semelhante, não conseguimos encontrar grande informação sobre a influência de atrasos computacionais em algoritmos de RA baseados em RL.

Atrasos computacionais não só afetam a velocidade do algoritmo, como podem também ter um maior impacto se afetarem a reatividade desses algoritmos a mudanças da qualidade da ligação. Dados desatualizados sobre a qualidade da ligação podem influenciar a sua escolha de MCS, o que resulta num decréscimo do débito da rede.

Nesta dissertação, analisaremos um algoritmo de RA dinâmica preexistente implementado através de RL chamado Data-driven Algorithm for Rate Adaptation (DARA). Este algoritmo apresentou dificuldades a ser aplicado a um cenário experimental. Fizemos mudanças e novas contribuições a este algoritmo que diminuem os seus atrasos computacionais e tornam possível o seu uso num cenário experimental. A nossa análise aumentou o nosso conhecimento sobre as causas desses atrasos computacionais, e soluções para esses atrasos. A nossa implementação resultou num algoritmo aproximadamente sete vezes mais rápido que o seu antecessor.

Acknowledgements

This dissertation would not have been possible without the assistance of several people, nor would it have been the same experience. For this reason, I would like to thank everyone who contributed, directly or indirectly, to the completion of this dissertation.

First and foremost, I would like to thank my supervisors, Helder Fontes, Ruben Queirós and Rui Campos, for presenting this opportunity to me. Their guidance and knowledge were indispensable to this project, and their patience and availability went above and beyond. It is not an understatement to say that this dissertation would not exist without their contributions.

I would also like to thank my family for always being at my side, encouraging my successes but also forgiving my mistakes. It is thanks to their unconditional support, be it emotional or financial, that I had the chance to be where I am today.

Finally, I would like to thank all of my friends who have accompanied me in both my times of need and of leisure. Not only did they help me reach my goal, but more importantly, have made the journey along the way worth it.

This work is a result of the project "DECARBONIZE – DEvelopment of strategies and policies based on energy and non-energy applications towards CARBON neutrality via digitalization for citIZEns and society" (NORTE-01-0145-FEDER-000065), supported by Norte Portugal Regional Operational Programme (NORTE 2020), under the PORTUGAL 2020 Partnership Agreement, through the European Regional Development Fund (ERDF).

Ricardo Jorge Trancoso

"It always seems impossible until it's done."

Nelson Mandela

Contents

1 Introduction **1**

1.1 Context . . . 1
1.2 Motivation . . . 2
1.3 Problem Definition . . . 3
1.4 Objectives . . . 4
1.5 Contributions . . . 4
1.6 Document Structure . . . 5

2 State of the Art **7**

2.1 Introduction . . . 7
2.2 IEEE 802.11 . . . 7
2.3 Rate Adaptation . . . 9
2.4 Reinforcement Learning . . . 11
2.5 Computational Delay . . . 13
2.6 Fed4FIRE+ . . . 14
2.7 Linux Tools . . . 15
2.7.1 mac80211 module . . . 15
2.7.2 proc filesystem . . . 15
2.8 Related Work . . . 16
2.8.1 DARA . . . 16
2.8.2 Other Works . . . 16

3 E-DARA Specification and Implementation **19**

3.1 Overview . . . 20
3.2 mac80211 Changes . . . 21
3.2.1 Procfs Initialisation . . . 21
3.2.2 File Operations . . . 22
3.2.3 Module Compilation . . . 25
3.3 Information Parsing . . . 26
3.4 Agent Implementation . . . 29
3.4.1 DQN Agent Implementation . . . 29
3.4.2 Q-learning Agent Implementation . . . 30
3.5 Environment Implementation . . . 32
3.5.1 Action Deployment . . . 33
3.5.2 Reward Calculation . . . 34
3.5.3 State Query . . . 36

4 Evaluation of the E-DARA Algorithm 39
4.1 Methodology 39
4.2 Preliminary Algorithm Evaluations 40
4.2.1 Information Processing 40
4.2.2 Comparison of the Environments 43
4.2.3 Comparison of the Agents 45
4.3 Final Evaluation and Comparison 49
4.3.1 Baseline DARA Evaluation 49
4.3.2 Final comparison 50

5 Conclusions and Future Work 53

References 55

List of Figures

1.1 Simplified overview of a Rate Adaptation implementation. 2

2.1 Reinforcement Learning interaction loop. 12

3.1 Basic overview of the algorithm loop. 20
3.2 Console output from reading /proc/tese/stats. 26
3.3 Content of the input file. 26
3.4 Shell command piping example. 27
3.5 Console output of running our Python parsing script. 28
3.6 Console output of running our Regex parsing script. 29
3.7 iperf example of switching the MCS rate. 34
3.8 Console output from reading /proc/net/wireless. 36

4.1 Results of the subprocess method for information processing. 41
4.2 Results of the Python method for information processing. 42
4.3 Results of the Regex method for information processing. 42
4.4 Box plots of *step* function time in milliseconds for each approach. 44
4.5 Plot of *step* execution time moving average over time. 45
4.6 Box plots of the agent side execution time in milliseconds for each NN with training. 46
4.7 Box plots of the agent side execution time in milliseconds for each NN without training. 47
4.8 Box plots of the agent side execution time in milliseconds of DQN and Q-learning. 48
4.9 Box plots of the environment side execution time in milliseconds of E-DARA and DARA. 51

List of Tables

2.1 Table of IEEE 802.11 standards. 9
2.2 Table of Minstrel's Multi-Rate Retry chain. 11
2.3 Specifications of w-iLab.2 nodes. 14

4.1 Average function execution time for environment alternatives in milliseconds. . . 43
4.2 Minimum function execution time for environment alternatives in milliseconds. . 44
4.3 Average time outside *step* function for different NNs. 47
4.4 Minimum time outside *step* function for different NNs. 47
4.5 Average and minimum time outside *step* function for the Q-learning algorithm. . 48
4.6 Average time of the environment and agent sides in our baseline DARA algorithm. 49
4.7 Minimum time of the environment and agent sides in our baseline DARA algorithm. 50

Abbreviations

CPU	Central Processing Unit
DARA	Data-driven Algorithm for Rate Adaptation
DQN	Deep Q-Network
DSSS	Direct Sequence Spread Spectrum
E-DARA	Enhanced Data-driven Algorithm for Rate Adaptation
FHSS	Frequency Hopping Spread Spectrum
GPU	Graphics Processing Unit
IEEE	Institute of Electrical and Electronics Engineers
INESC TEC	Instituto de Engenharia de Sistemas e Computadores, Tecnologia e Ciência
MAC	Medium Access Control
MCS	Modulation and Coding Scheme
MU-MIMO	Multi-User Multiple Input Multiple Output
NIC	Network Interface Controller
NN	Neural Network
ns-3	Network Simulator 3
OFDM	Orthogonal Frequency Division Multiplexing
OFDMA	Orthogonal Frequency Division Multiple Access
OS	Operating System
PHY	Physical Layer
RA	Rate Adaptation
Regex	Regular Expressions
RL	Reinforcement Learning
RSSI	Received Signal Strength Indicator
SNR	Signal-to-Noise Ratio
Wi-Fi	Wireless Fidelity
WiN	Wireless Networks Research Group
WLAN	Wireless Local Area Network

Chapter 1

Introduction

1.1 Context

The IEEE 802.11 standard, commonly known as *Wi-Fi*, is present in most household and office devices nowadays. It establishes the rules for the creation and use of Wireless Local Area Networks (WLAN), such as the frequencies that are used in communications, or which transmission techniques to use.

Due to advances in technology, the complexity of connected services has grown significantly over the last decade. Consequently, the demand for higher communication throughput has also increased across all forms of communication. Wi-Fi is no exception, and to meet those demands, several amendments were released over the years. One possibility introduced in those amendments, is adapting the Physical Layer (PHY) transfer rate to the current link quality. This is done by adjusting the Modulation and Coding Scheme (MCS). However, link quality is not static, and to fit the current channel conditions, the PHY transfer rates have to be changed dynamically. This mechanism is called dynamic Rate Adaptation (RA).

Algorithms for dynamic RA exist, such as *Minstrel* [1], the default RA algorithm of Linux kernels; and *Iwlwifi* [2], the RA algorithm used in Intel wireless chips. However, they have a few shortcomings [3]. They sample the network and compare different rates to choose the most appropriate one, but often take too long to converge to a fitting rate. They may even miss it entirely, because both algorithms tend to not work optimally under network conditions with very dynamic link quality. Dynamic RA algorithms based on Reinforcement Learning (RL) are now emerging on the state of the art as an alternative to these traditional static and heuristic-based algorithms.

The Wireless Networks Research Group (WiN), that belongs to the Centre for Telecommunications and Multimedia of INESC TEC, developed a Data-driven Algorithm for Rate Adaptation (DARA) [4], which is the basis of this dissertation. DARA is a standard compliant and RL-based RA algorithm that was initially implemented in Network Simulator 3 (*ns-3*) [5]. Afterwards, and previous to the work of this dissertation, there was an initial effort to migrate the implementation of DARA to an experimental environment. This preliminary implementation of DARA, for an

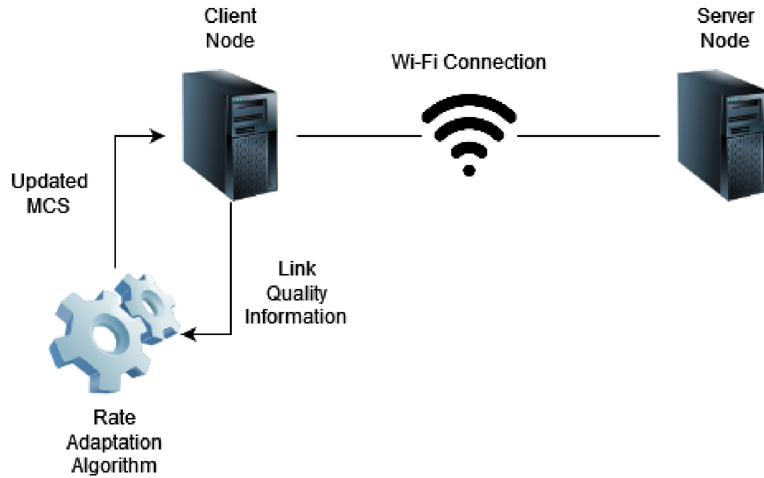

Figure 1.1: Simplified overview of a Rate Adaptation implementation.

experimental environment, was the focus of the work developed in this dissertation. However, this migration faced some issues.

1.2 Motivation

RL algorithms have been applied to the RA problem to increase the throughput of WLANs while remaining standard compliant. Instead of sampling the network as Minstrel and Iwlwifi do, these algorithms often require an exploratory training phase. It is possible to separate the training phase from the deployment of the algorithm. Training could be performed only once, reducing the computational delays of training during the deployment. Alternatively, the training can be done alongside deployment, which can accommodate online learning. Online learning would allow the algorithm to deal with unforeseen circumstances. Automatic solution finding and online learning are keys to improving Wi-Fi network speed and quality. Many encouraging results have been achieved with RL algorithms [3, 6, 7, 8], often better than the results of Minstrel and Iwlwifi.

Running an RL algorithm in a wireless device may affect its normal function. Computational delays can slow down optimal MCS selection for the current link quality, and overuse of system resources can even halt the process entirely. To the best of our knowledge, issues stemming from computational delays have not been analysed in detail. Reducing those delays should improve the effectiveness of RL algorithms. In turn, this should lead to an even higher throughput increase in Wi-Fi networks when using an RL algorithm for dynamic RA.

Most work of the state of the art on RL-based approaches for RA was validated in simulation. Despite the advantages of simulation, it is not always an accurate representation of real scenarios. Networks in real environments tend to be volatile and unpredictable. Furthermore, simulation can obscure the aforementioned problems stemming from computational delays. An algorithm that was only validated in simulation may have problems when implemented in an experimental scenario. This can happen since the simulation did not account for the computational delays of:

1) gathering and calculating the state information of the environment; 2) calculating the action to take based on that observation; and 3) implementing the appropriate action.

DARA was originally developed considering a real environment. However, it faced several issues that led to its development in simulation instead. Among these issues, is the impact of the algorithm's processing time. The algorithm took so much time to collect the observation, the reward, and to apply an action, that its decision was outdated by the time it was made. Analysing and reducing its delays may solve the problem of implementation in a real environment, which motivated the work developed on this dissertation.

1.3 Problem Definition

On a device running an RL algorithm for RA, there are many factors that affect the algorithm's performance. To the best of our knowledge, factors such as the computational delays of the algorithm have not been detailed in the state of the art. These delays will be our focal point, and can exist for various reasons:

- **Algorithm Delays –** Inefficiency or unnecessary halting from the algorithm. A task that the algorithm executes may be too slow for the algorithm's requirements. In such situations, it may be necessary to find alternatives to how it executes the task.

- **OS Overheads –** Overheads introduced by the Operating System (OS) of the device. The OS manages the system resources and services. It can be seen as an interface between the device's hardware and software, thus affecting our algorithm.

- **Hardware Limitations –** Device hardware specifications may limit the algorithm's processing. For example, not enough CPU (Central Processing Unit) or GPU (Graphics Processing Unit) power can delay the RA action.

Delays in the algorithm's processing impair its performance in RA. If the algorithm takes too long to choose an MCS, the Wi-Fi node might spend too much time using a sub-optimal MCS for the current link quality in the scenario. It might also take so long that when it does change, the chosen MCS is no longer appropriate for the current link quality, since it is constantly changing. In the end, this may result in a lower network throughput. If we minimise those delays, we can avoid situations where we end up with a sub-optimal throughput due to using an outdated MCS selection for the current link quality.

As previously stated, DARA has faced issues in an experimental scenario. Some of these issues are due to computational delays. We intend to verify this, and solve these issues if possible.

We will take as a basis DARA [4], in which we found room for improvement. We will not develop a new algorithm from scratch for this dissertation. This approach allows us to compare the original version of the algorithm to our Enhanced DARA (E-DARA), and evaluate the difference of the impact of computational delays between both.

1.4 Objectives

As far as we know, there is limited research on the state of the art addressing the computational delays of RL-based solutions. More specifically, we did not find any work focusing on the impact of using RL algorithms for RA in experimental scenarios. Therefore, the main objective of this dissertation is to characterise and quantify the computational delay of each main function of the algorithm, evaluating the feasibility of using RL algorithms for RA in experimental scenarios. More specifically, to enhance DARA's execution speed (i.e. minimise DARA's computational delays).

To achieve this, we intend to:

- **Improve our understanding of the computational delays in RL algorithms –** There are few works addressing this problem in the state of the art. Often, problems of this type are not the focus of these works, so the solution to the delays is not documented in detail. In this work, we will address this problem and document the steps we take.

- **Develop and implement the solution in an experimental scenario –** Experimental scenarios represent real environments more accurately than simulation. We will document the hardware specifications, the OS of the device, and other relevant information.

- **Analyse the computational delays –** This involves the use of profiling tools that assess the processing time, and the use of system resources when running the RL algorithm. The goal is to identify which issues arise from the processing time. This is an important step to decide how to approach the problem.

- **Devise a solution for the identified issues –** We implemented an enhanced version of DARA which aims to solve the identified problems. This implementation is documented in Chapter 3.

- **Validate our solution –** We compare the obtained results using our enhanced version of DARA to the baseline, the original DARA algorithm. This comparison is documented in Chapter 4.

1.5 Contributions

In this dissertation we have made several contributions. These contributions have resulted in our E-DARA implementation being faster than the original DARA. These were our main contributions:

- **Multiple alternative implementations –** We implemented and tested multiple ways to parse and input data in the algorithm. These implementations are detailed in Chapter 3. We documented the thought process behind these alternative implementations, and clarify some of their advantages and disadvantages.

- **Improvements to data acessibility –** Our device used a version of the Linux kernel. The kernel separates data into kernel space and user space. Accessing data in kernel space is not trivial. Some of the necessary data for our algorithm was updated to user space every 100 milliseconds. Our modifications to a kernel module enable us to access it without the 100 millisecond limitation.
- **Evaluation and validation of implementations –** We collected and analysed statistics on the aforementioned implementation versions. These analyses, displayed in Chapter 4, allowed us to make informed decisions on which of the implementations was probably the best for the combination of hardware and software used in the experimental setup of this dissertation.

1.6 Document Structure

This document is organised as follows: in Chapter 2, we introduce the state of the art and relevant background of our work. It includes information on the IEEE 802.11 standard, Reinforcement Learning, the DARA algorithm, relevant tools we use, and related work that was done in the field. In Chapter 3, we have our E-DARA specification and implementation. Here, we talk about how the algorithm works as a whole, which options we had to specify and implement, and the thought process behind our choices. In Chapter 4, we detail the evaluation and validation of our solution. We share our results, with accompanying plots, as well as compare our E-DARA implementation to the original DARA. In Chapter 5, we present the conclusions we derived from our results, final thoughts, and future work that can possibly improve our implementation even further.

Chapter 2

State of the Art

2.1 Introduction

In this dissertation, we analyse the computational delays in Reinforcement Learning algorithms used in an experimental environment. This involves the use of an RL algorithm in order to improve IEEE 802.11 network quality of service – e.g. increasing throughput and decreasing packet loss – and the analysis of the processing time of such a system. We will be covering the following topics in this chapter:

- IEEE 802.11 – Brief description of what it is, and the most relevant revisions to it;
- Rate Adaptation – Motivation behind RA, and how it works;
- Reinforcement Learning – How RL works and can be applied, and a brief description of relevant algorithms;
- Computational Delay – How it can affect the performance of RL algorithms, and possible ways to address the issue;
- Fed4Fire+ – The experimental testbed that was used in our analysis;
- Linux Tools – Linux components that were a part of our implementation;
- Related Work – Work that has contributed to the topic we discuss.

2.2 IEEE 802.11

IEEE 802.11 [9] is a standard that was developed to implement WLAN communications, and is commonly referred to as *Wi-Fi*. It specifies Medium Access Control (MAC) and physical layer (PHY) protocols. Although first developed in 1997, it has since seen many amendments and improvements that extend the technology's capabilities, keeping it up to date. It has become a popular technology due to its ease of use; adding new devices is fast and simple. When used to

connect to a modem, it permeates Internet access to multiple simultaneously connected devices wirelessly.

IEEE 802.11-1997 The first official protocol of the IEEE 802.11 standard is called *IEEE 802.11-1997*, also known as IEEE 802.11 legacy mode. It operated on the 2.4 to 2.4835 GHz band, and had a transmission rate of 1 or 2 megabits per second (Mbit/s). It implemented Frequency Hopping Spread Spectrum (FHSS), and Direct Sequence Spread Spectrum (DSSS).

IEEE 802.11b In 1999, *IEEE 802.11b-1999*, known as just IEEE 802.11b, was created. It extends transmission rate possibilities by allowing 5.5 and 11 Mbit/s connections, in addition to the original legacy rates. They also share the same frequency bands of 2.4 to 2.4835 GHz. The new rates were only implemented with DSSS because FHSS did not follow the Federal Communications Commission's instructions on operating with rates superior to 2 Mbit/s. IEEE 802.11b achieved theoretical ranges of 400 m in open spaces, and was the first widely adopted standard.

IEEE 802.11a On the other hand, the IEEE 802.11a standard – conceived around the same time as IEEE 802.11b – could employ 6, 9, 12, 18, 24, 36, 48 and 54 Mbit/s connections. However, it operated in the 5 GHz frequencies, with 20 MHz channels. This amendment introduced a technique known as Orthogonal Frequency Divison Multiplexing (OFDM), instead of using DSSS or FHSS. OFDM enables data to be sent in smaller sets simultaneously in different frequencies, but only when they would not interfere with each other. The 5 GHz band reduces the possibility of interference, since this band is less used. These two points improved the IEEE 802.11a reliability. However, it did not work with devices that only implemented the IEEE 802.11b and legacy modes, and the signal quality could be worse when there were obstacles present.

IEEE 802.11g In 2003, IEEE 802.11g (or *IEEE 802.11g-2003*) introduced IEEE 802.11a rates of up to 54 Mbit/s, in the IEEE 802.11b band of 2.4 GHz. Since it uses the same band, it shares the same expected network area coverage. An IEEE 802.11g device is compatible and can communicate with IEEE 802.11b devices. Furthermore, IEEE 802.11g also implements OFDM. When communicating with an IEEE 802.11b device, it reverts back to DSSS instead [10].

IEEE 802.11n Afterwards, in 2009, came the IEEE 802.11n amendment, also known as Wi-Fi 4. Among the biggest changes brought by this amendment is the introduction of configurable parameters. Namely, Short Guard Interval , Multiple Input Multiple Output, several Modulation and Coding Schemes (MCS), and Frame Aggregation. It can reach theoretical rates of 450 Mbit/s.

IEEE 802.11ac The IEEE 802.11ac amendment – or Wi-Fi 5 – is the successor to IEEE 802.11n, and was finalized in 2013. A lot of the related work was done in this version of the standard. It further improves IEEE 802.11n features by introducing Multi-User Multiple Input Multiple Output (MU-MIMO), which enables simultaneous transmission between more than just one device, using

multiple antennae; and *256-QAM* modulation which increases the amount of data that can be transferred at once.

IEEE 802.11ax Finally, the IEEE 802.11ax amendment was approved in 2021. It is commonly known as Wi-Fi 6. It introduces further improvements such as *1024-QAM* modulation and Orthogonal Frequency Division Multiple Access (OFDMA). OFDMA is an upgraded version of OFDM that also distributes packets in frequency divisions, but for multiple users at once. This version of Wi-Fi works in both 2.4 GHz and 5 GHz frequencies, unlike previous versions. It can reach theoretical rates of 9.6 Gb/s.

Table 2.1: Table of IEEE 802.11 standards.

Standard	Theoretical Rate	Frequencies	Techniques
802.11-1997	2 Mbit/s	2.4 GHz	FHSS; DSSS
802.11b	11 Mbit/s	2.4 GHz	DSSS
802.11a	54 Mbit/s	5 GHz	OFDM
802.11g	54 Mbit/s	2.4 GHz	OFDM; DSSS
802.11n	450 Mbit/s	2.4 GHz	OFDM; MIMO
802.11ac	1.7 Gb/s	5 GHz	OFDM; MU-MIMO
802.11ax	9.6 Gb/s	2.4, 5 and 6 GHz	OFDMA; MU-MIMO

In Table 2.1, we can see a summary of the discussed versions of the standard. Most of the related work was done in the IEEE 802.11ac standard, since IEEE 802.11ax is still recent. In this dissertation our focus will be the IEEE 802.11n standard due to limitations of the tools we will be using. However, the biggest change that impacts our work between IEEE 802.11n and IEEE 802.11ac is the number of MCS indexes. Therefore, extending our results to more recent versions of the standard should be straightforward.

2.3 Rate Adaptation

The objective of RA is to find the optimal data transmission rate for a given link quality. However, this is not an easy task. Just as link quality constantly changes, so do the optimal rates. To properly adjust the rate, the link quality should be monitored. The objective of RA is to improve network performance (e.g. throughput), despite changes in link quality.

For example, in a situation with poor link quality, it may be worthwhile to reduce the transmission rate. A lower transmission rate should reduce the error rate, which in turn lowers the number of retransmissions. By avoiding wasteful retransmissions, a lower rate can increase the useful throughput.

One way to monitor link quality, is to probe the Signal-to-Noise Ratio (SNR) of the data that is being transmitted. The SNR is the ratio of power of a meaningful signal over the power of background noise. A higher SNR means the signal is clear, and often indicates a higher quality

communication. Alternatively, we can calculate the frame loss ratio. The frame loss ratio is the ratio between the number frames that are not delivered, and the total number of frames sent. High frame loss ratios may indicate poor link quality, although this is not always the case. High frame loss ratios may also exist due to frame collisions from simultaneous transmissions.

Another possibility is probing different rates. Instead of only making decisions based on passively monitored link quality, it is possible to switch to different rates, and evaluate their success. One disadvantage of this method is that while probing unsuitable rates, the throughput may decline. This method is how some existing RA algorithms work, such as Minstrel [1] – the default algorithm of Linux kernels – and Iwlwifi [2] – the RA algorithm used in Intel wireless chips.

Minstrel Minstrel collects statistics to calculate the probability of success of a frame transmission for a given rate. For each rate, Minstrel starts by calculating the probability of success of a frame transmission periodically, using an Exponential Weighted Moving Average, as follows:

$$P_{success} = (1-\alpha) * P_{current} + \alpha * P_{previous} \tag{2.1}$$

where $P_{current}$ represents the ratio of successful transmissions over the total number of attempts of the current period, and $P_{previous}$ represents the weighted moving average of the previous period. $P_{success}$ is the probability that is used for the calculation of the maximum achievable throughput – which Minstrel does for each rate, according to the following equation [11]

$$MaximumThroughput = P_{success} * N_{packets} * FrameSize \tag{2.2}$$

where $P_{success}$ is the previously calculated weighted moving average of success of a frame transmission, and $N_{packets}$ is the number of packets that can be transmitted in one second under perfect link quality.

Minstrel uses the Multi-Rate Retry chain. In summary, it takes 4 candidate rates (r_0, r_1, r_2 and r_3), and a corresponding number of frame transmission retries (c_0, c_1, c_2 and c_3) specified by the driver. Rate r_0 will be used until the retries exceed c_0. Then, r_1 is used instead. This continues down to r_3 if necessary. Since Minstrel dedicates 10% of its transmissions to probe randomly chosen rates, only 90% of its transmissions are normal packets. It populates the 4 rates differently, depending on whether it is a random lookaround rate, or a normal rate, as follows:

As shown in Table 2.2, for normal traffic, the rates are: the best throughput rate, the next best throughput rate, the best probability rate, and the base rate. For the lookaround rate, if the randomly selected rate is lower than the current best throughput rate, the rates are: the best throughput rate, the random rate, the best probability rate, and the base rate. If the randomly selected rate turns out to be higher, then the random rate is first, and the best throughput rate is second. Despite switching between the rates in its Multi-Rate Retry chain based only on the number of frame transmission retries, the rates that constitute the chain are only updated once every 100 ms.

Table 2.2: Table of Minstrel's Multi-Rate Retry chain.

Attempt	Lookaround Rate		Normal Rate
	Random < Best	Random > Best	
r_0	Best Rate	Random Rate	Best Rate
r_1	Random Rate	Best Rate	Next Best Rate
r_2	Best Prob	Best Prob	Best Prob
r_3	Base Rate	Base Rate	Base Rate

The rates are switched by adjusting one of IEEE 802.11's configurable parameters, the MCS. Each MCS index is a combination of a modulation type and a coding rate. Lower index MCS have lower theoretical throughputs, but should be more reliable in terms of error rates.

There is a distinction to make between RA methods that are standard-compliant, and those that are non-compliant. Non-compliant methods make use of features and information that may not be easily accessed in most common Network Interface Controllers (NICs). Accessing those features requires modifying the NIC, or using feedback mechanisms (e.g. sending receiver's SNR back to the transmitter with out-of-band mechanisms). Those methods can only be implemented on devices with the same modifications afterwards. On the other hand, compliant methods only use features that are available by default on NICs that implement the IEEE 802.11 standard. Not only would this most likely mean an easier implementation, it should also be easier to extend it to most NICs.

RL-based algorithms applied to RA can improve network performance, by learning to relate link quality data – such as the SNR – to the most appropriate rate to use in that case. There are several solutions [3, 7, 12, 13] in the state of the art achieving this.

2.4 Reinforcement Learning

Reinforcement Learning is a subset of Machine Learning with the goal of automatically learning a solution to a problem. It works by having an agent that receives an observation from an environment. Then, the agent executes the action it deems best, considering the observation. This action interacts with the environment, which returns a new observation, and a reward associated with the action. This process can loop indefinitely.

In Figure 2.1 we can see a summary of RL interactions. The objective is to determine which actions give the highest reward in each state, without optimal examples of how to achieve the goal. This process is similar to how humans and animals learn. We do not require optimal examples of how to perform something in order to learn. Instead, we commonly learn by trying out things – resembling the actions taken by the agent – and then considering whether what we did held good results – resembling the reward. Afterwards, favourable actions are repeated, while different actions are attempted if they were not successful before.

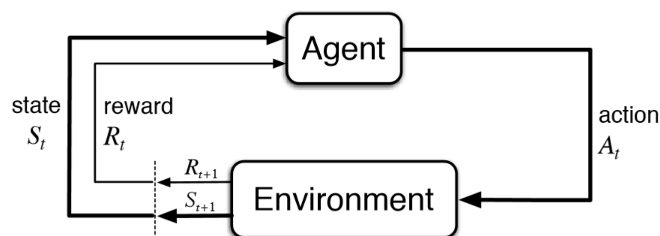

Figure 2.1: Reinforcement Learning interaction loop.

When making a decision on an action, we have two choices: *exploration* and *exploitation*. Exploration is choosing random actions to see how they perform. This allows the agent to gain more knowledge about the environment. Exploitation repeats actions that worked favourably before, in an attempt to maximise the reward. Although exploitation seems strictly better, balancing both is important [14]. For example, in a system that changes over time, exploration is key to adapting to new circumstances and achieving a higher cumulative reward.

However, maximising a total reward only coincides with achieving a goal if we pick an appropriate reward. For instance, if we are training an RL algorithm to play a video game where the goal is to survive as long as possible, considering the score as the reward is probably not the best course of action. Even if we had a great RL agent capable of maximising the reward — in this case, the score – this would not necessarily align with our goal, which is surviving for as long as possible. A more fitting choice might be to increment a reward for each time step we survive. While RL is a very goal-oriented form of Machine Learning, the task of picking the reward it should maximise in order to achieve that goal, is still ours.

In this dissertation, we will be evaluating the processing time of two different kinds of RL agents: a Deep Q-Network (DQN) agent and a Q-learning agent.

Q-learning This is an RL algorithm based on a lookup table, called a Q-table. A Q-table is a table with a row for each possible state, and a column for each possible action. In its cells, the corresponding state-action value is stored. This state-action value is the expected return of picking an action when presented with a certain state, and is more often called a Q-value. Q-learning converges the Q-table values to the optimal Q-values, eventually estimating the best action to pick for each state.

It learns by updating the Q-table after each action, through the Bellman equation:

$$q(s,a) = q(s,a) + \alpha(R + \gamma \max_{a'} q(s',a') - q(s,a)) \tag{2.3}$$

where $q(s,a)$ is the Q-value for a state s and action a; $q(s',a')$ is the Q-value of the next state s', in which we would pick the action that would result in the highest Q-value of that next state; α is the learning rate, which determines how quickly the agent abandons its previous Q-values in favour of its most recent results; γ is the discount rate, which determines the weight of future

rewards – a lower number would indicate a preference towards immediate rewards; and R is the numerical reward that resulted from our action.

DQN A Deep Q-Network is an RL algorithm based on the simpler Q-learning algorithm. Instead of having a Q-table that maps states and actions to their expected Q-values, it relies on a Q-function $f(s,a)$ that does the same. It leverages the power of Neural Networks (NNs) to approximate that Q-function. DQN refers to the learning method, but the NN itself can have any structure. In situations with many states and actions, DQN is preferred over Q-learning. A Q-table has to map every state to every action, so it can be computationally intensive to utilise it in such situations.

A Neural Network is a concept in Machine Learning, more relevant to deep learning algorithms, such as a DQN algorithm. It is a structure composed of multiple layers of nodes called neurons. These neurons connect to other neurons. There are multiple types of neurons, but we only focus on the simplest type, which has an associated weight and threshold. If the output of a neuron is above the threshold, it passes that output to the connected neurons depending on the weight of the connection. The first layer of an NN is an input layer that takes data from the outside. The last layer is an output layer which returns the result of the NN. The layers in between are called hidden layers and are usually the core of the NN's functioning.

An NN is often composed of multiple layers with many neurons. This results in a structure capable of identifying patterns in data. When coupled with iterative updates of the weights and thresholds of the neurons, the NN can improve its accuracy over time. This pattern can range from identifying an object in an image, to approximating a function.

The process through which a DQN agent learns, is analogous to Q-learning. Instead of updating Q-values in a Q-table towards the optimal Q-values, we update our NN to better approximate the Q-function. This is usually done with two nearly identical NN, one computing our predicted Q-value, and the other computing a target Q-value. Afterwards, we calculate the Mean Square Error of the loss, using the difference between those two Q-values. Then, we update the weights of the NN by back-propagating the loss using gradient descent. With a good approximation of the Q-function, the DQN agent knows the actions with the highest return for each state, and picks them.

2.5 Computational Delay

Computations are calculations done by a computer. They are not limited to arithmetic steps, and include executing instructions. Since algorithms are sets of instructions and arithmetic steps, devices running algorithms perform computations.

Despite individual computations usually being fast, they are not instant. Certain complex tasks can be very demanding in terms of computations. The speed of an algorithm depends on how quick these computations are executed. Since algorithms can be used to solve time-critical problems, faster computations are often desired. However, many factors can affect the end speed of an algorithm negatively. We call them computational delays.

One common example of a computational delay, is an overhead introduced by the OS of a device. Our algorithm will be a process in a device that is managed by an OS, and will not be the only process on the device. The OS is responsible for managing the resources and processes of the device, and thus, its multi-tasking. Multi-tasking is achieved by allocating time slots to its various tasks. During a task's time slot, the CPU processes that task. When the time slot is over, the task's processing is interrupted, and it works on a different task instead. These extra computations do not progress the algorithm's function, and are perceived as a delay from the algorithm's point of view.

We can also consider computational delays from the algorithm itself. Certain tasks can be slow to process, either due to their complexity, or due to their implementation. An implementation might be executing redundant or inefficient computations. It can also be an implementation that prioritises preserving memory over speed. Sometimes, a task is too complex, and we must explore alternative ways of achieving the desired output.

The physical components of the device also impact the overall performance. Although they do not directly cause a delay, the overall speed is dependant on the characteristics of the components. For instance, newer CPUs can perform the same instructions in a fraction of the time it would take older CPUs.

2.6 Fed4FIRE+

Fed4FIRE+ is a project under the European Union's Programme Horizon 2020. This project aims to facilitate research and innovation in the area of the Internet. They provide multiple Next Generation Internet testbeds for free experimentation [15].

Our testbed of interest will be *w-iLab.2* [16]. This testbed is located in Zwijnaarde, Belgium. It is composed of nodes scattered throughout 100 spots. There are 4 types of nodes, 2 of which are relevant to us. These are 48 ZOTAC nodes, 40 APU nodes, 10 DSS nodes and 15 mobile nodes, but we only used ZOTAC and DSS nodes. Each node has an embedded computer with 2 Atheros based Wi-Fi interfaces, supporting standards IEEE 802.11a to 801.11ac. The available computers can be accessed and controlled freely. In the nodes we used, we ran a Linux kernel, version 3.13.0, using an Ubuntu 14.04 distribution. A summary of the node specifications can be seen in Table 2.3.

Table 2.3: Specifications of w-iLab.2 nodes [16].

Feature	ZOTAC	APU	DSS/Mobile
CPU type	Intel Atom D525 (2cores, 1.8 GHz)	AMD G series T40E APU (2cores, 1GHz)	Intel core i5
RAM (GB)	4GB DDR2 800MHz PC2-6400 CL6	4GB DDR3 1066MHz	4GB DDR2 800MHz PC2-6400 CL6
Hard disk	160GB (2.5", SATA, 7200RPM, 16MB)	32GB (SSD,mSATA)	60GB (2.5", SATA, SSD)
Wi-Fi	2x802.11abgn	2x802.11ac	1x802.11abgn, 1x802.11ac

We use a Fed4FIRE+ testbed as it provides a real environment on which we can implement and evaluate our work. We test in an experimental setting as opposed to a simulated one due to the

unpredictability of real world wireless networks, and also because physical hardware is the most adequate equipment since the focus of our work is the processing time.

2.7 Linux Tools

Our version of the algorithm was implemented in the Linux OS, and a big part of the implementation depends on it. In this section we describe two Linux components that are relevant to our implementation. Namely, the mac80211 module [17], which we modified as part of our work, and the proc filesystem, that we used in these modifications.

2.7.1 mac80211 module

The Linux kernel is responsible for many different tasks. It is responsible for memory management, executing processes, controlling hardware, etc. Kernel modules are pieces of code which extend kernel functionality and enable it to accomplish some of these tasks.

Mac80211 [17] is a module for managing wireless devices. More specifically, it is a framework to create drivers for those wireless devices. For example, ath9k, the wireless device our wilab2 nodes (discussed in Section 2.6) use a device driver that depends on mac80211. It is similar to an interface between the linux kernel and the wireless device.

Mac80211 also implements rate control algorithms. One of those rate control algorithms – and its default – is Minstrel. Minstrel needs to calculate the probability of successful transmission. To calculate it, it needs information on frame successes and attempts. The mac80211 module stores this information in its variables. Furthermore, it computes and stores the Minstrel table of rates. The table of rates DARA reads comes from mac80211's implementation of Minstrel.

2.7.2 proc filesystem

The proc filesystem, also called procfs, is a virtual filesystem. It serves as an interface to data structures in the Linux kernel, and enables communication between kernel space and user space. Procfs often serves to provide information or statistics about the system, or a process. For example, each Linux process has a subfolder in the system, with information such as the virtual memory the process uses, or what command started the process.

It works by creating virtual files. Those virtual files do not have content that can be read and written to like a normal file. Instead, when read or written, they execute a function. These functions commonly read internal variables and output them, similar to a regular read. But they are not limited to this, and can have more complex behaviour. These internal variables, if part of a kernel module, would usually be restricted to kernel space. However, through procfs, the virtual file is in user space, despite being able to output data from the kernel space. This is what enables communication between the two spaces.

2.8 Related Work

In this section we discuss related work, such as the original DARA implementation this dissertation is based on. We also briefly talk about other RL implementations developed for RA, and possible gaps in the literature.

2.8.1 DARA

DARA[4] is an RA algorithm that employs a Deep RL method. More specifically, it uses a DQN. It takes the SNR of the environment as an input to its NN. The output is the MCS to adjust the rate of the link. To evaluate its performance, it uses a method similar to Minstrel's of evaluating the frame error rate. The version of DARA we had access to during the course of this dissertation was built for an experimental scenario.

It can be divided into 2 main components, the agent side and the environment side. The agent side is the agent itself and the NN. It is responsible for deciding the action as well as training. It was originally implemented with *tf-agents* [18], which works on the basis of *Tensorflow* [19]. These are Python modules created for implementing Machine Learning solutions.

The environment side serves as the interface which provides the input data to the algorithm, the SNR; is responsible for taking the action and applying it, i.e., actually switching the MCS that the wireless device uses; gathering information on frame attempts and successes, and calculating the reward. It makes use of the Python *gym* module, and a bash script created by the author that is ran through the Python *subprocess* module [20].

The bash script has several commands. This is because of the multiple different functions of the environment. The commands are used to access files, and often that information is parsed through piping other commands. For example, the information for calculating the reward is obtained through mac80211's Minstrel implementation, by reading a file with the tables Minstrel uses. One thing to note is that the tables are only updated every 100 ms. This means DARA is limited to calculating one reward once every 100 ms.

2.8.2 Other Works

There is a lot of work done in the topic of using ML-based approaches for the task of RA [3, 6, 7, 8, 21, 22]. A lot of these approaches have provided results superior to those of Minstrel and Iwlwifi. However, our focus in this dissertation is the computational delays of RL algorithms, a subset of ML.

The existing research on computational delays on the topic of RL algorithms used for RA is very limited. We believe that when computational delays are detected in other works, they are seen as problems in the implementation that are not a focus of the work. Therefore, contributions are not often made to the general understanding of computational delays. However, the authors in [3] mention the delay between querying the agent and receiving a candidate for switching rates. It was around 1.3 ms to 3.7 ms on average (tested in two different devices). When compared to common

probing intervals which tend to be in the order of tens of milliseconds, this is an improvement. This was achieved through the use of an asynchronous framework, which coordinates the RL agent and the kernel driver, so that it does not need to halt for the agent's training and inference.

Despite encouraging results of RL, they were mostly achieved in simulated scenarios, or in very specific experimental ones, with very powerful computational resources. As is often admitted by the authors, more rigorous testing in real environments is necessary before this technology can be deployed. To the best of our knowledge, there are no other works that evaluate and characterise computational and processing delays such as hardware limitations, and overheads that come from the OS. As such, our work may provide insight into causes of computational delays in these algorithms, as well as possible solutions. Such insights could possibly enhance preexisting algorithms, or facilitate the development of new ones.

Chapter 3

E-DARA Specification and Implementation

In this chapter, we will discuss our implementation of the DARA algorithm, which we call Enhanced DARA (E-DARA). The goal of E-DARA is to increase throughput in a Wi-Fi connection through Rate Adaptation, through a similar process to the original DARA. However, our implementation is different so as to decrease the computational delays of the algorithm, thus minimising the associated impact on performance. Note that the focus of this solution are the computational delays, rather than its training and evaluation as a RA algorithm.

The action of the algorithm corresponds to an MCS rate that the wireless device switches to. The state of the algorithm comes from a parameter called signal level. The signal level is a measurement of the strength of the connection's signal, similar to the Received Signal Strength Indicator (RSSI). The reward function that the algorithm uses to learn, differentiating good actions from bad actions, comes from Equation 3.1, which is similar to the one Minstrel uses.

$$\text{Reward} = \frac{\text{Successes}}{\text{Attempts}} \times \frac{\text{Current Rate}}{\text{Max Rate}} \tag{3.1}$$

The equation depends on the frame success probability and the theoretical throughput of the rate used, in comparison to the maximum rate.

We could not implement the agent side through *tf-agents* [18] such as in DARA. We believe this was due to the use of an older version of *tf-agents* in DARA, coupled with the use of a different Linux version with different libraries installed. Instead, we used *Keras* [23], a Python application programming interface for implementing NN models, and agents. The agent side's implementation is conditioned by the library used. The two main changes we apply to this side are choosing a different agent (between DQN and Q-learning), and different NN model sizes (in the case of a DQN agent). Hence, most of our implementation focuses on changes to the environment side. However, as we will see in the following chapter, this is the main bottleneck of the algorithm, so our focus on this side is not misdirected.

The key parts of our implementation are: 1) changes to the accessibility of data from the

kernel, which was done in the C programming language; 2) changes to the way data is collected and input into the algorithm; 3) changes to how the data is parsed; and 4) variations to the agent, such as Q-learning and DQN, or changing the NN that is used. Parts 2 to 4 have been implemented in both Python and Rust.

3.1 Overview

As previously stated, the functioning of the algorithm can be divided into two parts: the agent side, and the environment side. The agent side comprises the training of the agent and the action decision. We do not have a lot of control over this side. On the other hand, the environment side handles the action deployment, the reward calculation and the state query. This side is implemented by a function created by us, and that we have full control over. This function runs at every step.

To start, the agent and the environment are initialised. After initialising, the environment queries the state, which will be used for the first action decision. Then, the algorithm enters a loop. We can identify five main steps which are the biggest bottlenecks of the operation, and whose analysis is the focus of this dissertation. Figure 3.1 is an overview of the loop, with these main steps in orange (note that the reward statistics are queried twice, but this is considered the same step).

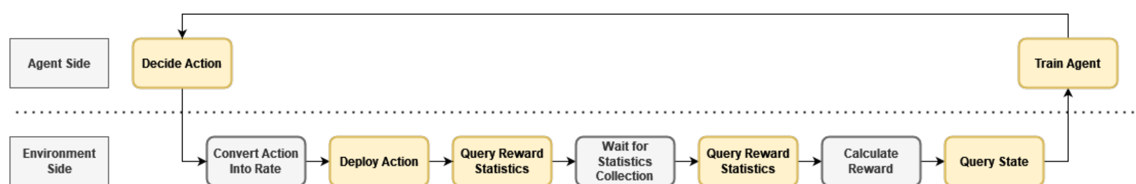

Figure 3.1: Basic overview of the algorithm loop.

The following is a quick summary of the algorithm loop. The loop begins when the agent decides an action, and we enter the environment side's step function. This function starts by deploying the chosen action. Then, it queries the total frame successes and attempts. The algorithm waits a configurable amount of time (we chose 50 ms), and queries the total frame statistics again. After, it calculates the difference of the previous query to obtain the reward according to Equation 3.1. The pause between queries gives time to collect information on frame statistics. Then, the algorithm queries the signal level, which will be the next step's state. Subsequently, the algorithm leaves the environment side's step function. Using the reward, the agent is trained, and the loop starts over.

The alternatives we explored are two different agents (DQN and Q-learning), different NN models in the case of DQN, and then three ways to implement the three main environment side steps. These alternatives are discussed in further detail in the following sections.

3.2 mac80211 Changes

Before implementing the algorithm, we wanted to avoid the 100 ms limitation in calculating the reward that affected DARA. This limitation came from Minstrel updating its table only once every 100 ms. To achieve this, we went to the source of the Minstrel data. Our goal was to find an alternative way to obtain information on frame attempts and successes.

The mac80211 module implements a version of Minstrel. The module keeps track of frame successes and attempts. However, this data usually remains in kernel space. To address this, we modified the mac80211 module's implementation of Minstrel. In our modification, we created a procfs system that can access the same frame successes and attempts variables Minstrel uses to create its tables. We can access the data without needing to wait 100 ms for Minstrel to update the table. Instead, we read the file at any time to obtain this data. By accessing the data more frequently we can reduce the time of each step. This may not leave us enough time to gather information since the last query, and lead to a premature conclusion. The trade-off between acessing the data more frequently or giving it more time to gather information has to be considered when choosing the time between queries.

In this section we will detail our implementation, dividing it into three steps: initialisation of the procfs system, defining file operations to implement the features we need, and compilation as well as installation of the modified module.

3.2.1 Procfs Initialisation

We start with the source code of mac80211. This is a folder with many C files and headers. The version of the module's source code must match the Linux kernel version in use. In our case, we were running Linux kernel version 3.13.0. Our changes are limited to the file *rc80211_minstrel.c*, which contains mac80211's implementation of Minstrel.

We start by adding the following to the list of includes in the file:

```
1 #include <linux/printk.h>
2 #include <linux/proc_fs.h>
```

Listing 3.1: Includes added to *rc80211_minstrel.c*.

<linux/proc_fs.h> is the header file for the procfs library. This library is necessary to implement a procfs system. *<linux/printk.h>* is not mandatory, but it enables the use of the *printk* function to print kernel messages for debugging.

Then, we need to initialise the procfs system. We do this on minstrel's initialisation function, called *rc80211_minstrel_init*. This will create the procfs files whenever mac80211 initialises minstrel. The following is how our *rc80211_minstrel_init* function looked after our changes:

```
1 int __init
2 rc80211_minstrel_init(void)
3 {
4     proc_folder = proc_mkdir("tese", NULL);
5     if (proc_folder == NULL) {
```

```
 6             printk("Tese␣-␣Error␣creating␣/proc/tese\n");
 7             return -ENOMEM;
 8         }
 9
10         proc_file = proc_create("stats", 0666, proc_folder, &fops);
11         if (proc_file == NULL) {
12             printk("Tese␣-␣Error␣creating␣/proc/tese/stats\n");
13             proc_remove(proc_folder);
14             return -ENOMEM;
15         }
16
17         return ieee80211_rate_control_register(&mac80211_minstrel);
18     }
```

Listing 3.2: Minstrel initialisation function *rc80211_minstrel_init* after our changes.

Our procfs file is called */proc/tese/stats*. This means we create a subdirectory *tese* under the *proc* directory. This is done in lines 4-8. Within the *tese* subdircetory, we create the file we will use, called *stats*. This is done in lines 10-15.

Line 4 in the initialisation function creates the *tese* subdirectory. The *proc_mkdir* function in line 4 creates a new procfs directory. It takes two arguments. The first argument is a string with the name of the new directory. The second argument is a pointer to the parent directory. It returns a pointer to the newly created directory. When it is *NULL*, such as in our case, it uses the standard *proc* directory as its parent.

Line 10 creates the *stats* file. The *proc_create* function creates a procfs file on which file operations (such as read and write) can be defined. It takes four arguments. The first argument is a string with the name of the new file. The second argument is the access rights of the file. *0666* gives access to the user without any privileges. The third argument is a pointer to the parent directory. We used the previously created *proc/tese* directory's pointer. The fourth argument is a pointer to a structure with file operations applicable to this file. This function returns a pointer to the newly created procfs file.

Lines 5-8 and 11-15 are there for error handling purposes. In case of an error creating any procfs file or directory, the error is logged, pointers are cleared, and the initialisation of the module is aborted. Line 17 is part of minstrel's original initialisation and was left unchanged.

3.2.2 File Operations

In the previous subsection, we created a procfs file through the *proc_create* function. This function took a pointer to a structure with file operations as an argument. This structure simply maps certain file operations to a function. When a file operation is executed on the procfs file, the mapped function is executed. The mapped functions usually follow conventions depending on the file operation. For example, when reading a file, it is expected that the return value of the function is the number of bytes that were successfully read, and that certain arguments are passed, such as the number of bytes to read.

Our file operations structure only uses two file operations: read and write. When reading from the file, it executes a function called *pread*. While writing to the file, it executes a function called *pwrite*. Both functions are implemented by us. The code for the structure we used, which we called *fops*, is the following:

```
1 static struct file_operations fops = {
2     .read = pread,
3     .write = pwrite,
4 };
```

Listing 3.3: Procfs file operations definition.

As stated before, procfs is a virtual file system. Its files do not have any content. The same holds for our *stats* file. When the file is read, our *pread* function is executed. It is up to the function to implement the functionality we want. In our case, we want it to be similar to a regular file read, except it prints variables directly from the code. In Listing 3.4, we have our *pread* function.

```
1  static ssize_t pread(struct file *File, char *user_buffer, size_t count,
       loff_t *offs) {
2      static char buffer[PROCFS_MAX_SIZE];
3      static unsigned long buffer_size;
4      static int finished = 0;
5
6      if (finished) {
7          finished = 0;
8          return 0;
9      }
10     finished = 1;
11
12     buffer_size = (unsigned long) sprintf(buffer, "%d,%d\n", g_successes,
           g_attempts);
13
14     if (copy_to_user(user_buffer, buffer, buffer_size)) {
15         return -EFAULT;
16     }
17
18     return buffer_size;
19 }
```

Listing 3.4: *pread* function that is executed when our file is read.

In a regular read, the function returns the number of bytes read. Additionally, it requires arguments such as the pointer to the file that will be read; a pointer to the buffer on which the contents that were read are copied to; the number of bytes to read; the offset that determines the position at which reading will start. Our *pread* function follows that convention.

Line 12 places the frame successes variable and the frame attempts variable to a temporary buffer, separated by a comma through the *sprintf* function. This temporary buffer is what the output from "reading" the file will be. The return of the *sprintf* function is the number of bytes that were printed, which tells us the size of the buffer. Line 14 uses the *copy_to_user* function,

which is the proper way to copy contents from kernel space to user space. It copies from the temporary buffer to the buffer that was passed as a function argument. Finally, the number of bytes that were passed to the user buffer is returned.

When a file is too large, it is not read in a single read function. Instead, the read function is executed several times, until the return is zero, meaning there is nothing left to read. We had an issue when reading our file. Rather than executing the *pread* function once, it would execute it in a loop. This happened because our return value was never zero. For this reason, we added lines 4-10. This lets our file be read normally once, then return zero at the second execution. This should not be a problem because the printed variables are small enough that they can be read in a single function call.

The following is our *pwrite* function.

```
1  static ssize_t pwrite(struct file *File, const char *user_buffer, size_t
       len, loff_t *offs) {
2      int command;
3
4      if (kstrtoint_from_user(user_buffer, 1, 0, &command)) {
5          return -EFAULT;
6      }
7
8      if (command == 1) {
9          g_successes = 0;
10         g_attempts = 0;
11     } else {
12         printk("Tese - Unknown command - doing nothing\n");
13     }
14
15         return 1;
16 }
```

Listing 3.5: *pwrite* function that is executed when our file is written to.

Analogous to the *pread* function, our *pwrite* function definition is similar to a regular write function. It takes as arguments a pointer to the file; a pointer to the buffer from which contents would be written to the file; the amount of bytes that would be written; and the offset, which is the position in the file after which writing would begin. The return value of the function is the number of written bytes.

Despite the similarities to a regular write function, we have no intention of writing any content to the file. Instead, this is a way to issue commands to mac80211. Commands can be executed by attempting to write an integer – each integer corresponding to a certain command – to the file. We only implemented one command: resetting the successes and attempts variables to zero. This happens when we write a *1* to the file.

Line 4 uses the function *kstrtoint_from_user*. This function is the proper way to pass a user space string to a kernel space integer. The first argument is the user buffer from which the conversion will happen. The second argument is the number of bytes to be converted. We only have one

command, of one digit, so we only consider one byte. The third argument is the base the number is in. Passing a zero detects the base automatically, and in this case uses a decimal base. The fourth argument is a pointer to the variable on which the integer will be stored. Since commands are only one byte, the function ends when it returns one.

3.2.3 Module Compilation

After making changes to the *rc80211_minstrel.c* file, all that is left is to compile and install the modified module. To do this, we use the Makefile within the mac80211 source code directory, with a few changes. A Makefile contains tasks to be executed. Makefiles are often used to compile executable files from the source code.

The following is a snippet of what we changed in the Makefile:

```
1  all:
2          make -C /lib/modules/3.13.0/build modules M=/groups/ilabt-imec-be/
               inesctec-win/ricardojt/mac80211
3
4  clean:
5          make -C /lib/modules/3.13.0/build clean M=/groups/ilabt-imec-be/
               inesctec-win/ricardojt/mac80211
6
7  insmod:
8          cp /groups/ilabt-imec-be/inesctec-win/ricardojt/mac80211/mac80211.
               ko /lib/modules/3.13.0/kernel/net/mac80211/mac80211.ko
9          modprobe -r ath9k
10         modprobe -r ath10k_pci
11         modprobe -r ath10k_core
12         modprobe -r mac80211
13         modprobe mac80211
14         modprobe ath9k
```

Listing 3.6: Mac80211 module Makefile.

Lines 1 and 2 define the default task of the Makefile, which is to compile the module. Compiling Linux kernel modules requires specialised tools, provided by Linux. In our case, those tools were in the */lib/modules/3.13.0/build* directory. In line 2, we call another Makefile in that directory with access to those tools. We pass *modules* as an argument because we are building a module, and then the directory with the module source code. Our directory is */groups/ilabt-imec-be/inesctec-win/ricardot/mac80211*.

In line 4, we define a task to clean our directory. In line 5, we call that same Makefile, but with *clean* as an argument. Cleaning the directory deletes all compiled files, but leaves source code files.

The final task, *insmod*, copies the compiled kernel module file – called *mac80211.ko* because it is a kernel object. It places the kernel object file in the location of the original mac80211 module, replacing it. After replacing the module files, our changes will be applied after restarting the module. We do this with the *modprobe* utility. *modprobe -r* stops a module. First, we stop

the modules *ath9k*, *ath10k_pci* and *ath10k_core*. They depend on mac80211, and the mac80211 module cannot be shut down before stopping its dependencies. Then, we stop and restart the mac80211 module. This will apply the changes. Finally, we also restart the *ath9k* module, since it is the driver for our wireless device.

After these steps, our changes to mac80211 should be applied. A file in */proc/tese/stats* should exist. When this file is read, either through a terminal command such as *cat*, or through a Python function such as *read*, it should output the frame successes and attempts, separated by a comma. Additionally, writing a *1* to the file should reset the frame successes and attempts.

```
root@node1:/groups/ilabt-imec-be/inesctec-win/ricardojt# cat /proc/tese/stats
4902,6538
```

Figure 3.2: Console output from reading /proc/tese/stats.

Figure 3.2 is an example of reading the file, using the Linux terminal and the *cat* command.

3.3 Information Parsing

Many of the processes of our algorithm involve reading files. For example, querying the state requires reading from a file in */proc/net/wireless*, and calculating the reward requires reading the file */proc/tese/stats*. However, these files can contain information other than what we need. Additionally, the contents are read as a string, and may not be formatted in a way that the algorithm can process.

We came up with three methods to parse the information to extract data that the algorithm can process. Those methods are: a) shell command piping; b) python string functions; and c) regular expressions (Regex). We ran some preliminary tests to ascertain which was the fastest option. In Chapter 4, we detail these tests. In the following paragraphs, we have a description of each method as well as an example of its usage.

The input file is a copy of */proc/net/wireless* called *input*. The */proc/net/wireless* file is the one we read from when querying the state. We use a copy because the file is constantly updating. This ensures the example input file stays the same in the following examples.

Figure 3.3 shows the content of the input file after reading it.

```
root@node1:/groups/ilabt-imec-be/inesctec-win/ricardojt# cat input
Inter-| sta-|   Quality        |   Discarded packets               | Missed | WE
 face | tus | link level noise |  nwid  crypt   frag  retry   misc | beacon | 22
 wlan0: 0000   38.  -72.  -256        0      0      0     11     23         0
```

Figure 3.3: Content of the input file.

Shell Command Piping Shell command piping is the use of a shell utility called piping. Piping is the act of carrying over the output from one command to another. Usually, piping is a way to

filter some output. For example, reading from a file in the shell, and passing the output through a filtering command such as *grep* or *awk*. This was used in the original DARA implementation, and required the use of the subprocess module.

As an example, we will pipe the contents of `cat input`, which reads the input file, to the *awk* command. Our goal is to retrieve only the signal level, which in this example is -72. The *awk* command accepts arguments that define how it filters what it receives as an input. The full *awk* command we will use in this example is `awk 'NR==3 {print substr($4, 1, length($4)-1)}'`. The `NR==3` leaves only the third line in the file, which is the one that contains the values. The value we are looking for is the fourth field of the line, which can be printed with `$4`. However, this would also print the period in the field, which we do not want.

To avoid this, we use the *substr* function, which only prints a subset of the original string. This function takes as the first argument the input field; as the second parameter the start position, 1, which means the start of the field; and the third parameter is the length of the subset of the string, which in our case is one less than the whole length of the field, removing the final period.

To apply these filters to the content of the input file, we pipe the output from reading it with *cat*, to our *awk* command, hence `cat input | awk 'NR==3 {print substr($4, 1, length($4)-1)}'`.

Figure 3.4 shows an example of the full piping process, correctly outputting only the value we are looking for.

```
root@node1:/groups/ilabt-imec-be/inesctec-win/ricardojt# cat input | awk 'NR==3 {print
 substr($4, 1, length($4)-1)}'
-72
```

Figure 3.4: Shell command piping example.

Python The Python approach consists of using built-in Python functions that modify strings. Some of the functions relevant to this approach are the *find* and *split* functions. This has the advantage of not requiring any additional modules. Note that in this approach we used preexisting string operation functions, rather than creating original ones specifically tailored to our purposes.

For the Python example, we created a small script. This script reads the file, filters the information we need and turns it into an integer, as we would in the algorithm. Additionally, it prints that integer so that we can see the result.

The script code is the following:

```
1 with open('input', 'r') as f:
2     content = f.read()
3
4 result = content.split('\n')[2]
5 result = result.split()[3]
6 result = result[:-1]
7 result = int(result)
8
```

```
9 print(result)
```

Listing 3.7: Python script with built-in functions to process the input file.

Lines 1-2 read the file called *input*. Line 4 splits the content by lines, and takes the third line (Python is zero-indexed). Line 5 splits the line by space-separated fields, and takes the fourth. Line 6 excludes the final character of the field (the trailing period). Line 7 converts the result from a string to an integer. In line 9 the result is printed.

Figure 3.5 shows the output of running our Python parsing script.

```
root@node1:/groups/ilabt-imec-be/inesctec-win/ricardojt# python3 pythonparse.py
-72
```

Figure 3.5: Console output of running our Python parsing script.

Regex Regex is a way to specify text patterns, and enables searching for these patterns. The advantage of Regex is the ability to create complex search patterns without needing dedicated search functions. In theory, string operations should be faster. However, we did not create Python functions optimised for our purposes, while the Regex patterns we use are tailored to our intents.

We use a Python built-in module called *re* [24] to implement Regex pattern matching. Therefore, in this example we still use a Python script, but the implementation is different.

This script's code is the following:

```
1  import re
2
3  with open('input', 'r') as f:
4      content = f.read()
5
6  pattern = re.compile(r'(?:wlan0[^-]*)(-\d+)')
7  result = pattern.findall(content)[0]
8  result = int(result)
9
10 print(result)
```

Listing 3.8: Python script with Regex to process the input file.

The main differences between this script, and the previous are lines 6 and 7. Line 6 compiles our Regex pattern into a variable. In line 7 we find all occurrences of the pattern in the content of the file. The first occurrence is our intended output.

The Regex pattern we use is `(?:wlan0[^-]*)(-\d+)`. This pattern searches for a line starting with *wlan0*. This corresponds to our intended line, the third one. Then, it saves the first number that is a negative value. This corresponds to our signal level. It also does not save the trailing period because that is not a digit.

Figure 3.6 shows the output of running our Python parsing script.

```
root@node1:/groups/ilabt-imec-be/inesctec-win/ricardojt# python3 regexparse.py
-72
```

Figure 3.6: Console output of running our Regex parsing script.

3.4 Agent Implementation

After making changes to the mac80211 module, and finding out the best way to process information, all that is left is the implementation of the RL algorithm. This is also done in Python. We use two scripts, one that implements the agent side, building and running the agent, and another that implements environment side.

This section focuses on the agent side script. In the agent side, the agent training and action decision are the two main steps. We only experiment with parameters such as the type of agent, which can be a DQN or use Q-learning; and in the case of a DQN agent, the NN architecture. This is because there are not many alternatives in this method, since we use the *tensorflow* and *keras* modules to create the DQN agent; and our Q-learning implementation follows the mathematical definition closely.

We implemented a DQN and a Q-learning agent. The DQN agent was considered because it is the agent that was used in the original DARA implementation. Q-learning was considered because it is a simpler algorithm, and should be faster. In theory, Q-learning should be sufficient for the task, because the number of states and actions is small enough that having a Q-table should not be too computationally intensive. We implemented a working Q-learning agent that can learn and improve, but did not evaluate its performance in order to validate its effectiveness, since that was not our focus.

3.4.1 DQN Agent Implementation

We use the *tensorflow* and *keras* modules, as they provide tools to create a DQN agent. The DQN agent, after being built with these modules, has functions to train and evaluate the agent.

The following is the Python function we use to create the agent.

```
1 def build_agent():
2     actions = env.action_space.n
3     model = build_model(1, actions)
4     policy = BoltzmannQPolicy()
5     memory = SequentialMemory(limit=50000, window_length=1)
6     dqn = DQNAgent(model=model, memory=memory, policy=policy,
7         nb_actions=actions, nb_steps_warmup=10, target_model_update=1e-2)
8     dqn.compile(Adam(lr=1e-2), metrics=['mae'])
9     return dqn
```

Listing 3.9: *build_agent* function definition.

The *actions* variable has the number of possible actions, also known as action space. In our case there are eight MCS rates, so our action space is eight. The *build_model* function creates

the NN. The policy, memory buffer and compilation parameters used were taken from a Github repository [25]. Optimising these parameters was beyond the scope of this dissertation.

The *build_model* function that creates the NN changes depending on our choice of NN. The following is an example of the function to build an NN with two hidden layers of 32 neurons.

```
1 def build_model(states, actions):
2     model = Sequential()
3     model.add(InputLayer(input_shape=(states,)))
4     model.add(Dense(32, activation='relu'))
5     model.add(Dense(32, activation='relu'))
6     model.add(Dense(actions, activation='linear'))
7     return model
```

Listing 3.10: *build_model* function definition.

Line 2 means that the layers will be connected sequentially. Line 3 adds an input layer with as many neurons as there are possible states. In our case, we consider 100 discrete states. Lines 4 and 5 add the two hidden layers. Line 5 adds a final output layer with as many neurons as there are actions.

To modify the NN structure we only change lines 4 and 5. For example, if we wanted one layer with 64 neurons, we would replace both lines with a single line with the function `model.add(Dense(64, activation='relu'))` instead.

With these two functions, we can create a DQN object with two methods, *fit* and *test*. We use these methods to train and evaluate the agent, respectively. After training the agent, we can save the model's weights. By saving and loading the model's weights, we can resume training or evaluate the model.

3.4.2 Q-learning Agent Implementation

The Q-learning agent was implemented in Python by us. It uses a matrix initialised with zeroes as the Q-table. The Q-table is updated through the Bellman optimality equation. It uses an epsilon-greedy policy. We used the *numpy* module [26], to create and utilise the Q-table matrix; and the *gym* module, to interface with the environment.

The following code block (Listing 3.11) is the setup of the algorithm, where we initialise some of the variables that we will use.

```
1  lr = 0.1
2  gamma = 0.8
3  epsilon = 1.0
4  max_epsilon = 1.0
5  min_epsilon = 0.1
6  decay_rate = 0.1
7
8  qtable = np.zeros((env.observation_space.n,env.action_space.n))
9
10 state = env.reset()
```

```
11  total_rewards = 0
```

Listing 3.11: Q-learning algorithm setup.

The variable *lr* is the learning rate. Variable *gamma* is the discount rate. Variable *epsilon* is the current epsilon. In a greedy-epsilon policy, the epsilon decreases in order to progressively exploit more often. In this case, our epsilon varies decays from 1 to 0.1. The variable *decay_rate* dictates how fast the epsilon value declines. Line 8 creates the Q-table. Lines 10 and 11 reset the environment and the rewards.

After initialisation, the algorithm enters a loop. Each loop cycle is a time step. The loop continues until our target number of steps is achieved. Listing 3.12 is the code for the loop.

```
1  for step in range(max_steps):
2      tradeoff = random.uniform(0, 1)
3
4      if tradeoff > epsilon:
5          action = np.argmax(qtable[state,:])
6      else:
7          action = env.action_space.sample()
8
9      new_state, reward, _, _ = env.step(action)
10
11     qtable[state, action] = qtable[state, action] + lr * (reward + gamma *
           np.max(qtable[new_state,:]) - qtable[state, action])
12
13     total_rewards += reward
14     state = new_state
15     epsilon = min_epsilon + (max_epsilon - min_epsilon)*np.exp(-decay_rate*
           step)
```

Listing 3.12: Q-learning algorithm implementation with training.

The *tradeoff* variable is a random number between 0 and 1. It is compared to the epsilon value in lines 4-7 to see if the algorithm will exploit or explore. If an exploitation step is chosen, the algorithm picks the action it deems best (line 5). Otherwise, in an exploration step, it samples a random action instead (line 7).

In line 9, the algorithm runs the *step* function. The *step* function belongs to the environment side of the algorithm and will be explored in further detail later. It applies the action, returns the reward of that action and the following state.

In line 11, the algorithm applies the Bellman optimality equation and updates the Q-table. In line 15, the new decayed epsilon is calculated.

These two code blocks detail our full Q-learning algorithm implementation. This implementation both chooses actions and trains the algorithm. If instead we had a trained Q-table and wanted to only exploit it without further training, we can instead use a more concise version. Listing 3.13 shows a full exploitation-only implementation of our Q-learning algorithm.

```
1  state = env.reset()
2  total_rewards = 0
```

```
3 for step in range(max_steps):
4     action = np.argmax(qtable[state ,:])
5
6     new_state , reward , _, _ = env.step(action)
7
8     total_rewards += reward
9     state = new_state
```

Listing 3.13: Exploitation-only Q-learning implementation.

3.5 Environment Implementation

The environment side is the interface between the agent and the device. This side is where the majority of our focus was directed to, and where we have the most control over, since it was implemented entirely by us. As will be seen in Chapter 4, this is the biggest bottleneck of the algorithm's execution, which means our focus here was warranted.

Figure 3.1 once again has an overview of our algorithm implementation. In the figure, we can see the steps associated with the environment side. The main steps are coloured orange. Note that querying the reward happens twice, but that is only an implementation detail. We consider it a single task.

The environment side extends the Python *gym* module in order to create an interface with the environment. This side is mostly comprised by a Python function that is executed at every time step. This function is called *step*. It is responsible for querying the state, calculating the reward, and deploying the action. The following code block (Listing 3.14) is our *step* function implementation.

```
1 def step(self, action):
2     rate = self.actionlist[action]
3     rateint = self.rateintlist[action]
4     self.set_action(self.wiface,rate)
5     old_stats = self.get_reward()
6
7     time.sleep(0.05)
8
9     stats = self.get_reward()
10    successes, attempts = stats[0] - old_stats[0], stats[1] - old_stats[1]
11
12    if attempts == 0:
13        reward = 0
14    else:
15        reward = rateint/MAX_RATE * (successes / attempts)
16
17    self.iteration += 1
18
19    if self.iteration >= self.max_steps:
20        done = True
```

```
21        else:
22            done = False
23
24        self.state = _convert_signal(self.get_state())
25
26        return self.state, reward, done, {}
```

Listing 3.14: Implementation of the *step* function.

Note the *set_action*, *get_reward* and *get_state* functions. Each of these sub-functions inside the main *step* function represents one of the main tasks of the environment side. We explored three alternative implementations for each of these functions. Those alternatives were implemented and evaluated in order to reduce the computational delay of the algorithm. There is a sub-section for each of these tasks that details their functioning, as well as the three alternative implementations.

The three alternatives we explored are: 1) a pure Python approach; 2) an approach that uses a Python module called *subprocess*; and 3) to extend Python with Rust.

The subprocess approach is the most similar approach to the original DARA implementation, which is why we implemented it. In this approach, we make use of a Python module called *subprocess*. This module enables the use of shell commands from within Python. Because DARA made extensive use of a bash script, it used the *subprocess* module often.

In the Python approach, we attempted to use idiomatic Python for the task functions. For accessing the files to query the state and to calculate the reward, we used the regular Python built-in function *read*. The only exception is on action deployment. In this case, the implementation is the same as in the subprocess approach, because deploying an action requires the use of a shell command.

Rust is a compiled programming language whose execution speed is comparable to C. This means that the execution speed is much faster than Python. For this reason, we attempted to implement the three aforementioned tasks as Rust functions, that are called from within Python.

3.5.1 Action Deployment

Deploying an action is separate from the agent deciding it. When the agent decides on an action, it outputs an integer that corresponds to a certain MCS rate. The integer is an index to the list of MCS rates we use, which are of 6, 9, 12, 18, 24, 36, 48 and 54 Mbit/s. These are the rates of IEEE 802.11g, but it would be easy to adjust the possible actions to a different version of the standard.

Deploying an action refers to getting the wireless device to start transmitting with the new rate. To do that, we use a Linux terminal application called *iwconfig*. With this application we can change the MCS rate at will using the following command: `iwconfig [device name] rate [rate] fixed`. The device name is the name by which the computer recognises the wireless device, in case it has multiple devices. In our case, the device name is *wlan0*.

In Figure 3.7 we can see the console output of an *iperf* connection. *iperf* is a tool for measuring connection bandwidth. Before the measurement, the highest MCS rate of 54 Mbit/s was set,

```
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0- 1.0 sec  2.56 MBytes  21.5 Mbits/sec
[  3]  1.0- 2.0 sec  1.04 MBytes  8.75 Mbits/sec
[  3]  2.0- 3.0 sec   504 KBytes  4.13 Mbits/sec
```

Figure 3.7: iperf example of switching the MCS rate.

by using the command `iwconfig wlan0 rate 54M fixed`. In the 1-2 second interval, the following command was executed: `iwconfig wlan0 rate 6M fixed`. This switched the MCS rate to the lowest of 6 Mbit/s. This switch causes the decrease in bandwidth seen in the last two seconds.

In our algorithm, the action deployment is handled by the *set_action* function. This function is responsible for running the previous console command. For this reason, both the subprocess and Python versions of this function are the same. We did not find a reliable alternative to run a console command in Python.

Listing 3.15 defines the subprocess version of the *set_action* function. The function body of the Python version is the same.

```
1  def subprocess_set_action(self, wiface, rate):
2      p = subprocess.run(['iwconfig', self.wiface, 'rate', rate, 'fixed'])
```

Listing 3.15: Subprocess version of set_action.

The Rust version of the *set_action* function follows a similar thought process. It invokes the same *iwconfig* shell command as well. Listing 3.16 shows the Rust implementation of the *set_action* function.

```
1  fn set_action(_py: Python, wiface: &str, rate: &str) -> PyResult<String> {
2      let mut cmd = "iwconfig ".to_owned();
3      cmd.push_str(wiface);
4      cmd.push_str(" rate ");
5      cmd.push_str(&rate);
6      cmd.push_str(" fixed");
7      Command::new("sh").arg("-c").arg(&cmd).output().expect("Error changing
           action");
8      Ok(cmd)
9  }
```

Listing 3.16: Rust version of set_action.

3.5.2 Reward Calculation

To calculate the reward, we have a function that obtains the total frame successes and attempts. The function achieves this by reading from our procfs virtual file that has access to the mac80211 internal variables.

The internal variables correspond to the total frame successes and attempts rather than those over a certain timeframe. Thus, we run the function once after deploying the action, wait for a configurable amount of time – we chose 50 ms for our tests – and then run the function again. By calculating the difference between the two results, we get the frame successes and attempts in those 50 ms, with the chosen action. The configurable pause gives the algorithm time to collect enough frame statistics with the new rate.

To finalise, we calculate the reward according to Equation 3.1:

$$\text{Reward} = \frac{\text{Successes}}{\text{Attempts}} \times \frac{\text{Current Rate}}{\text{Max Rate}} \tag{3.1}$$

In this equation, we divide the frame successes by the attempts to obtain the frame success ratio. Then, we divide the current rate's theoretical maximum throughput, by the theoretical maximum throughput of the highest rate (54 Mb/s). In the end, the reward should be a number between 0 and 1, where 1 corresponds to the theoretical maximum achievable throughput.

The core of this step, and the biggest bottleneck, is reading the procfs file. In comparison, the calculations that are performed, are negligible. Those calculations are done in the *step* function, and the access to the procfs file is done by the *get_reward* function.

In our *get_reward* function alternatives, the main difference is how we access the procfs file. As seen in Figure 3.2, the content is two values separated by a comma. After accessing the file, we separate the values by the comma, and turn them to integers.

Listings 3.17, 3.18 and 3.19 are our three implementations of the *get_reward* function.

```
1 def subprocess_get_reward(self):
2     p = subprocess.run(['cat','/proc/tese/stats'],capture_output=True,
          universal_newlines=True)
3     out = p.stdout
4     stats = [int(x) for x in out.split(',')]
5     return stats
```

Listing 3.17: Subprocess version of get_reward.

```
1 def python_get_reward(self):
2     with open('/proc/tese/stats', 'r') as f:
3         out = f.read()
4     stats = [int(x) for x in out.split(',')]
5     return stats
```

Listing 3.18: Python version of get_reward.

```
1 fn get_reward(_py: Python) -> PyResult<(i64, i64)> {
2     let mut content = read_to_string("/proc/tese/stats").unwrap();
3     content.pop();
4     let v: Vec<&str> = content.split(',').collect();
5     let successes: i64 = v[0].parse().unwrap();
6     let attempts: i64 = v[1].parse().unwrap();
7     Ok((successes, attempts))
```

```
8 }
```

Listing 3.19: Rust version of get_reward.

3.5.3 State Query

For the agent to decide on the best action, it needs information about the environment. This information is the state. In our case, the state should convey information of how good the link connection is. To do this, we read from a file in the directory */proc/net/wireless* This file is a procfs file, present in Linux in the distribution we used, Ubuntu 14.04. In Figure 3.8 we can see the output of the file. The file contains statistics about network interfaces. Among these statistics is the one we use, signal level. In the case of the figure, the signal level is -69.

```
root@node1:/groups/ilabt-imec-be/inesctec-win/ricardojt# cat /proc/net/wireless
Inter-| sta-|   Quality        |   Discarded packets               | Missed | WE
 face | tus | link level noise |  nwid  crypt   frag  retry   misc | beacon | 22
 wlan0: 0000   41.  -69.  -256        0      0      0      0     23        0
```

Figure 3.8: Console output from reading /proc/net/wireless.

The signal level is a measure of the signal's strength, akin to the RSSI. This measure is not guaranteed to represent the RSSI accurately, but it should be consistent between measurements.

Our agent takes in discrete states, ranging from 0 to 99. Because the signal level is a negative value, we apply some calculations after being queried. We add 99 to the signal level, and set it to 99 in case it exceeds that value, or to 0 if it is lower than that. This ensures the value stays between 0 and 99. However, we have not seen any occurrence of the signal level exceeding those bounds after the operation. Taking the example of Figure 3.8, the final state according to the algorithm would be 30.

This conversion to a state between 0 and 99 is done by an auxiliary function that is executed after all three versions of the *get_state* function. Listing 3.20 shows the code of that auxiliary function.

```
1 def _convert_signal(signal):
2     signal += 99
3     if signal < 0:
4         signal = 0
5     elif signal > 99:
6         signal = 99
7     return signal
```

Listing 3.20: State conversion function.

The three alternatives of the *get_state* function differ in how they read the */proc/net/wireless* file. After reading the file, we use a Regex pattern to extract the signal level. In the python and subprocess versions, we compile the Regex pattern beforehand as a global variable with `state_regex = re.compile(r'(?:wlan0[^-]*)(-\d+)')`. After extracting the signal

level, we transform it into an integer and convert it with the *convert_signal* function. Listings 3.21, 3.22 and 3.23 are the three alternatives of the *get_state* function.

```
1 def subprocess_get_state(self):
2     p = subprocess.run(['cat','/proc/net/wireless'], capture_output=True,
        universal_newlines=True)
3     out = p.stdout
4     signal = int(state_regex.findall(out)[0])
5     return signal
```

Listing 3.21: Subprocess version of get_state.

```
1 def python_get_state(self):
2     with open('/proc/net/wireless', 'r') as f:
3         out = f.read()
4     signal = int(state_regex.findall(out)[0])
5     return signal
```

Listing 3.22: Python version of get_state.

```
1 fn get_state(_py: Python) -> PyResult<i64> {
2     let content = read_to_string("/proc/net/wireless").unwrap();
3     let re = Regex::new(r"(?:wlan0[^-]*)(-\d+)").unwrap();
4     let cap = re.captures(&content).unwrap().get(1).unwrap().as_str();
5     Ok(cap.parse().unwrap())
6 }
```

Listing 3.23: Rust version of get_state.

Chapter 4

Evaluation of the E-DARA Algorithm

4.1 Methodology

All of our tests were done in an experimental setting, in nodes from the W-iLab.2 testbed [16] acessible through Fed4Fire+. They consist of a client node and a server node communicating wirelessly. The client is running our E-DARA implementation, and adjusts the MCS dynamically. Our tests are conducted in the client node.

We used 2 nodes: a ZOTAC node and a DSS node [1]. The ZOTAC node is the server, while the DSS node is the client. Since the algorithm runs on the client, it uses a DSS node because of its better CPU. The ZOTAC has an Intel Atom D525 1.8 GHz 2-core as its CPU, and a 160 GB HDD. The DSS node has an Intel Core i5 2.6 GHz 4-core as its CPU and a 60 GB SSD. Both nodes have a 4 GB DDR2 800 MHz PC2-6400 CL6 RAM, and use a Ubuntu 14.04 Linux Distribution, version 3.13.0 of the Linux kernel.

To begin, we load our modified version of the mac80211 module, and initialise E-DARA. After making sure both devices can communicate with each other, we start a UDP transmission through `iperf`. Our algorithm can log each action it takes, the state that lead to that action, as well as the resulting reward.

Our tests are timing measurements of Python functions in the algorithm. These Python functions usually represent a certain task of the algorithm. By selecting which function to time, we can measure the time of specific tasks separately. For example, we can know the time it took to deploy an action, or the time it took to query the state. Timing introduces a slight overhead to the regular functioning of the algorithm. This overhead should be small enough to be negligible. To avoid this overhead as best we can, we do not time all the functions at once. For example, when measuring the time that it takes the algorithm to process information, we do not measure the time of tasks such as calculating the reward.

As stated in Chapter 3, we often came up with multiple alternatives for each challenge we faced. We use the previously mentioned timing measurements to determine the best alternative. We will measure our final version of E-DARA – containing the best alternatives we considered

[1] The hardware specifications of the nodes can be found at https://doc.ilabt.imec.be/ilabt/wilab/hardware.html#id2

– and a version of DARA as close to the original as we could get. In the end, we compare and analyse these measurements.

4.2 Preliminary Algorithm Evaluations

In the implementation of our algorithm, we had challenges such as how to gather information from the device, how to process it, or which NN model to use. For each of these problems, we considered multiple alternatives. In this section, we run preliminary tests for each of the alternatives. This allows us to find the best option for each challenge.

4.2.1 Information Processing

As stated before, we process information from files frequently. This information cannot be directly used by the algorithm. First, we have to filter the information from those files, as well as format it in the appropriate way. To process information, we considered three alternatives: 1) Regex; 2) Python functions; and 3) bash commands, through the subprocess module.

We tested the three alternatives in two different scenarios to evaluate them, through a Python script using the *timeit* module [27]. The two scenarios considered are when reading from the */proc/net/wireless* file, which has a complex structure; and our */proc/tese/stats* file which only has two values separated by a comma. We called these scenarios *wireless* and *stats* respectively due to the name of the files. For our tests, we read each of the two files only once. We placed the contents of each file in a string in our script, so that the input is always the same.

For the *wireless* file scenario, our objective is a negative integer that corresponds to the signal level of the connection. The input is the same as in Figure 3.3, so the result should be -72. The implementations are also the same as in Section 3.3.

For the *stats* scenario, our objective is a list with two integers, one for the number of successful frame transmissions and another for the total frame transmission attempts. In this scenario, the input is the same as in Figure 3.2, so the result should be 4902 successes and 6538 attempts.

We use the *timeit* module to measure the time each method takes to execute ten thousand times. Then, we repeat that measurement five times, and consider the fastest measurement. Finally, we divide that number by the number of times the function was executed to find out the average. The time to load the required modules is not measured, since this is a one-time delay at the algorithm's initialisation. Additionally, after creating a regex pattern, the pattern needs to be compiled before use. Since this can also be done at initialisation, and only needs to be done once, it is not measured either.

Subprocess Using a bash script through subprocess to process information is how DARA handles this challenge. For this reason, we decided to evaluate this method.

We implemented this by having the subprocess module execute a bash script that pipes our predetermined input to a *awk* command that filters the information we need. Since each command returns one value, for the *stats* scenario we needed to execute two bash scripts, one for each value.

The alternative would be to return these values in a certain format, like in two different lines, or comma-separated. However, the data already comes separated by a comma. If it returned one of those formats, it would require further processing through another method, so this approach would be redundant.

We can see this implementation in Listing 4.1. The *subprocess_stats1.sh* script pipes the input to the command `awk -F',' '{print $1}'`. This command separates the contents by a comma and returns the first field. The *subprocess_stats2.sh* script is very similar, but has `{print $2}`. This returns the second field instead.

```
1 def subprocess_stats():
2     p = subprocess.run(["testes/subprocess_stats1.sh"], capture_output=True
          , universal_newlines=True)
3     successes = int(p.stdout)
4     p = subprocess.run(["testes/subprocess_stats2.sh"], capture_output=True
          , universal_newlines=True)
5     attempts = int(p.stdout)
6     return [successes,attempts]
```

Listing 4.1: Subprocess method that calls bash scripts to process */proc/tese/stats* data.

In Figure 4.1 we can see the results of this approach.

```
>>> time_subprocess_wireless()
subprocess_wireless average: 5.031795710200095 ms
>>> time_subprocess_stats()
subprocess_stats average: 9.979209684202215 ms
```

Figure 4.1: Results of the subprocess method for information processing.

Python In this approach, we use generic Python built-in functions, namely *split*. It should be noted that when reading the */proc/net/wireless* file, we often copied a single value from the list that the function returns, and pasted that value to a variable that we modify further. This operation also takes time. A Python approach developed very specifically for our requirements, as opposed to using generic functions, might be faster.

The Python approach for processing */proc/tese/stats* is the following.

```
1 def python_stats():
2     return [int(x) for x in content_stats.split(',')]
```

Listing 4.2: Python method to process */proc/tese/stats* data.

In Figure 4.2 we can see the results of this approach.

```
>>> time_python_wireless()
python_wireless average: 0.0017245488101616502 ms
>>> time_python_stats()
python_stats average: 0.0011701262198039331 ms
```

Figure 4.2: Results of the Python method for information processing.

Regex In theory, processing the Regex pattern should introduce an overhead, and a pure Python string function approach should be faster. However, complex patterns can be created with Regex. These complex patterns often rule out potential candidate matches very fast. This reduces the amount of time spent on something that will not match. As we see in the results, for a more complex scenario, Regex pattern matching is faster than using Python's generic functions.

The Regex approach for processing */proc/tese/stats* is the following. The Regex pattern used is `(\d+)`, which returns any numbers it finds.

```
1 def regex_stats():
2     return [int(x) for x in re_stats.findall(content_stats)]
```

Listing 4.3: Regex method to process */proc/tese/stats* data.

In Figure 4.3 we can see the results of this approach.

```
>>> time_regex_wireless()
regex_wireless average: 0.0013517389001208358 ms
>>> time_regex_stats()
regex_stats average: 0.0018107501199119724 ms
```

Figure 4.3: Results of the Regex method for information processing.

In conclusion, Regex was the fastest method for the *proc/net/wireless* file, with the Python approach coming in second. For the */proc/tese/stats* file, Python was the fastest, while Regex came second. Using Bash scripts through subprocess was by far the slowest method.

The discrepancy in the fastest method for the two files comes from their complexity. The */proc/tese/stats* file only requires separating the values by a comma. This is trivial to do in Python, with a single use of the *split* function. On the other hand, processing the */proc/net/wireless* file with the Python approach requires using the *split* function twice, picking the correct index from their result, and then removing the trailing period from that result. With the Regex approach, this is a matter of matching a single pattern.

The subprocess approach was the slowest because every time a command is executed, a shell instance needs to be created. Furthermore, the subprocess module needs to be compatible with multiple OSes. In order to have this compatibility, an overhead is introduced.

With these results, we used the Regex approach for processing the */proc/net/wireless* file in the *get_state* function, and the Python approach for processing the */proc/tese/stats* file in the *get_reward* function.

4.2.2 Comparison of the Environments

In Section 3.5, we detailed our environment implementation. We had three alternatives for the tasks of the environment side. These tasks are deploying the action, calculating the reward, and querying the state. The alternatives are using the subprocess module, using an idiomatic Python approach, and using a Rust approach.

To compare these three different alternatives we ran two tests. One test was to measure the *step* function execution time. This effectively measures the environment side's execution time as a whole. The other test was to measure each of the task sub-functions separately. Note that the *get_reward* function is called twice at each time step. Because the implementations were already detailed, we only discuss the results of these measurements in this section.

The measurements to the functions were done by measuring the clock time before the function call, and after it. Afterwards, the difference between the clock time after the function call, and before it, is the time spent in the function execution. This clock time is gathered with the *time* module's *perf_counter_ns* function. Additionally, we can also measure the time spent outside the function by taking the difference between the clock time at function exit, and the clock time at the following function call.

Measurements can be taken when the algorithm is training, and when it is only being exploited. This should not affect our measurements when we are evaluating the environment side.

For the sake of comparing only the different environment alternatives, we kept all other parameters the same. This means our measurements involve a DQN agent training for 10,000 steps, with an NN model with two hidden layers of 32 neurons. Measuring the execution time of function calls introduces an overhead, albeit small. Because of the overhead, we completed two measurements for each environment alternative, one for the *step* function, and another for the task sub-functions.

Figure 4.4 is a box plot of the execution time of the *step* function in milliseconds for each approach. Table 4.1 contains the average execution time of the *step*, *set_action*, *get_reward* and *get_state* functions in milliseconds. Table 4.2 contains the minimum execution time of the *step*, *set_action*, *get_reward* and *get_state* functions in milliseconds. We considered the minimum to be relevant in delay analysis because delays are always positive, so it serves as an approximation of a best case scenario. Note that these measurements include the 50 ms waiting period for data gathering that the *step* function has.

Table 4.1: Average function execution time for environment alternatives in milliseconds.

Environment	Subprocess	Python	Rust
step average (ms)	99.637	62.805	65.107
set_action average (ms)	11.535	15.105	14.981
get_reward average (ms)	13.226	0.246	0.085
get_state average (ms)	13.468	0.299	1.012

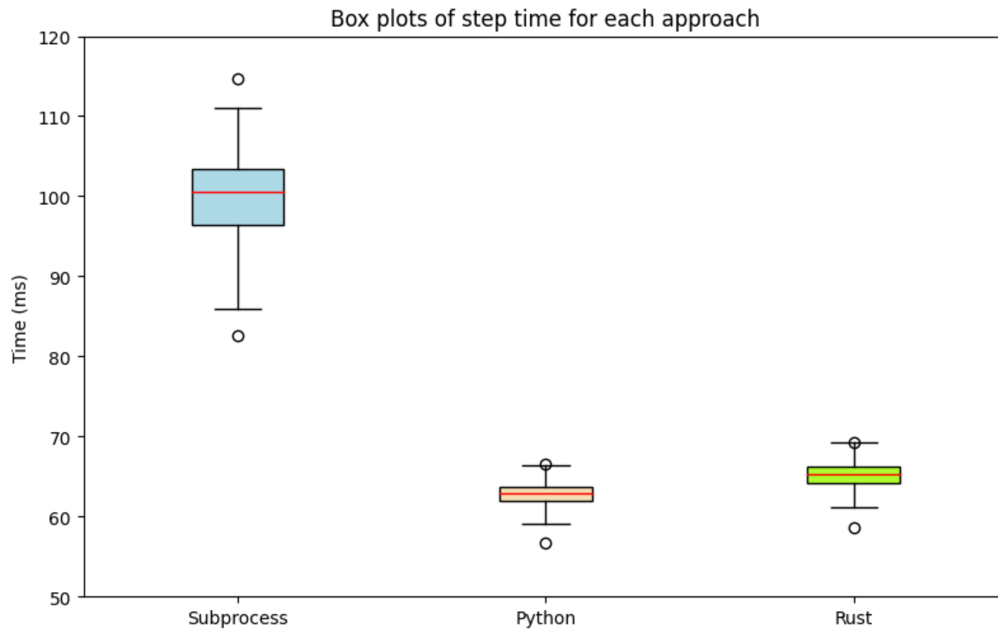

Figure 4.4: Box plots of *step* function time in milliseconds for each approach.

Table 4.2: Minimum function execution time for environment alternatives in milliseconds.

Environment	Subprocess	Python	Rust
step minimum (ms)	78.421	56.701	58.539
set_action minimum (ms)	7.369	10.646	9.949
get_reward minimum (ms)	7.827	0.113	0.034
get_state minimum (ms)	8.640	0.125	0.571

From these results, we conclude the idiomatic Python approach was the fastest, and Rust was a close second. The execution time of Rust might be faster, but every time a Rust function is called from within Python, a conversion to the passed and returned variables must be computed. This conversion introduces an overhead to the use of Rust functions which might explain why it is slower in the long run.

The subprocess version was the slowest. The subprocess module has a lot of overhead due to cross-platform support, that is not useful in our case.

We decided to use the Python alternative because overall, it was the fastest. It may be worth to pursue a mixed approach that uses the best approach for each task. Otherwise, for improvements on the environment side of the algorithm, the focus should be on action deployment. Action deployment is slower than the other two tasks by at least a whole order of magnitude.

During testing, we had concerns with the fluctuation of execution time over the 10,000 steps of training and/or exploitation. If this execution time varied drastically over time, this could affect our measurements if they were done sequentially. For this reason, we also evaluated the execution

time of the *step* function over time. Figure 4.5 is a plot of those results. A moving average of window size 16 was applied in order to smooth out the results, because our intent was to find variations over time, and not occasional irregularities.

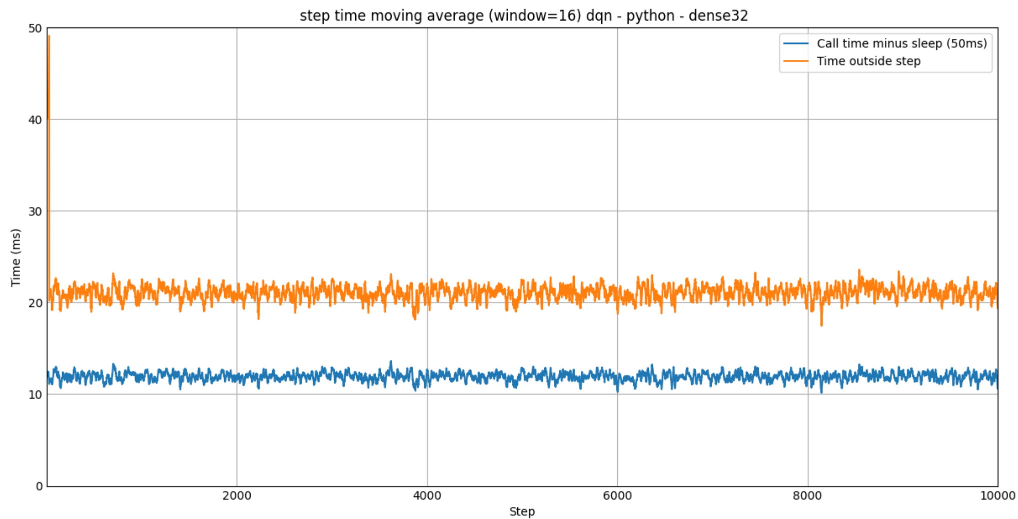

Figure 4.5: Plot of *step* execution time moving average over time.

As we can see from Figure 4.5 , with the exception of the very first time outside the *step* function, the execution time stays within certain bounds, and a systemic increase or decrease is not observable. The spike in the first time outside the *step* function may be a delay stemming from the initialisation of the agent side.

4.2.3 Comparison of the Agents

In the agent side, we do not have the same level of control as in the environment side. Instead of evaluating conceptually different implementations, we instead have certain choices to make. The choices we have are the type of agent used – DQN and Q-learning – and in the case of a DQN agent, the NN architecture.

Because of our lesser control over this side, implementing a way of measuring the time spent training and exploiting the algorithm is not trivial. To solve this issue, we instead perform the same measurements on the *step* function as before. Then, we consider the time spent between exiting the *step* function, and calling it again. This measures the time spent outside the *step* function, which should be the agent side's execution time.

One disadvantage of this approach is that we do not have direct and separate measurements of the time it takes to train the algorithm, and the time it takes to decide the action. For this reason, we measure and compare the same agents twice. Once training and exploiting, and then again exploiting only.

Training the algorithm is required at first, but after having a sufficiently trained algorithm, further training may no longer be necessary. This is why we decided to prepare these two measurements.

In our implementation, the NN is configurable, and we have several choices. We decided to evaluate four. Those are a model with one layer of 24 neurons; two layers of 32 neurons each; a layer of 1024 neurons; and with four layers of 256 neurons. The environment implementation we chose is the Python approach.

The reasoning to evaluate the first model, is to test a relatively minimalist model with few neurons. The second model is to evaluate what we considered to be a standard model. The final two models intend to test bigger models, one with emphasis on a single layer with many neurons, another with the same number of neurons but more layers.

Figure 4.6 is a box plot of the agent side execution time for each NN when both training and exploiting. Figure 4.7 is a box plot of the agent side execution time for each NN when only exploiting. Table 4.3 contains the average time spent outside the *step* function for each of the network models, when training and exploiting the agent as well as when exploiting only. Table 4.4 contains the minimum time spent outside the *step* function for the same parameters. The measurements were taken over 10,000 steps in both the training and the exploitation-only scenario.

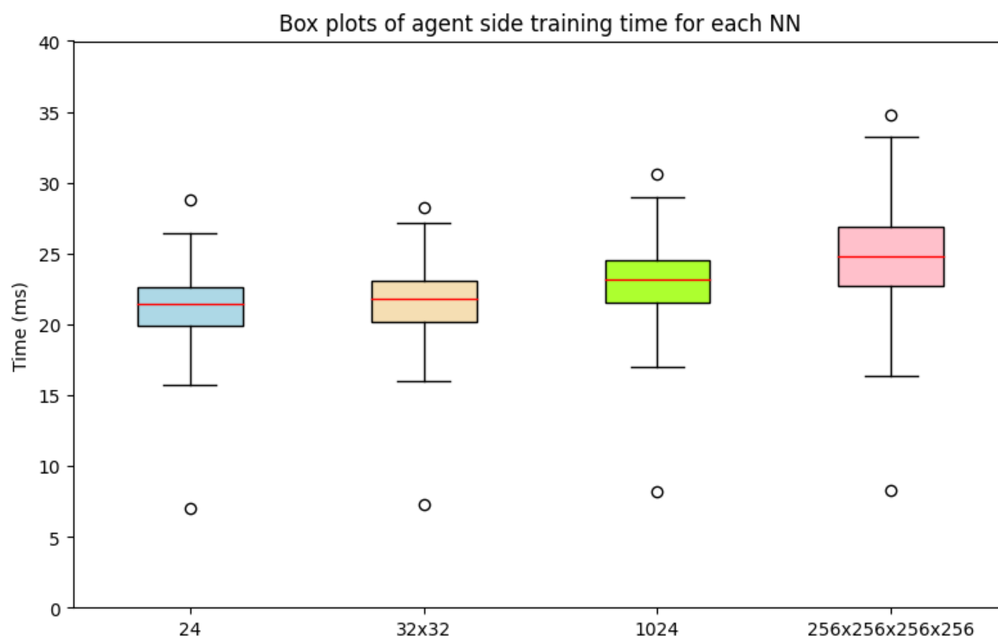

Figure 4.6: Box plots of the agent side execution time in milliseconds for each NN with training.

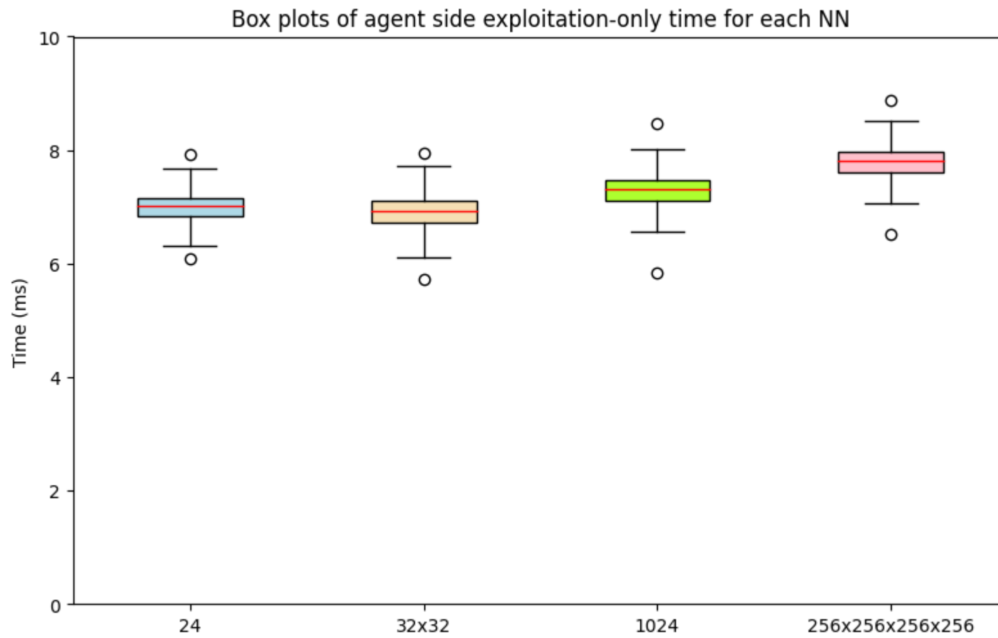

Figure 4.7: Box plots of the agent side execution time in milliseconds for each NN without training.

Table 4.3: Average time outside *step* function for different NNs.

Neural Network	Training & Exploitation (ms)	Exploitation-Only (ms)
24	21.349	6.967
32x32	21.606	6.915
1024	23.022	7.263
256x256x256x256	24.912	7.761

Table 4.4: Minimum time outside *step* function for different NNs.

Neural Network	Training & Exploitation (ms)	Exploitation-Only (ms)
24	6.973	4.829
32x32	7.209	4.946
1024	7.522	4.907
256x256x256x256	7.997	5.618

From these results, we gather that model size does slightly impact execution time. Bigger models are slightly slower, but the difference is very small. For this reason, the model should be chosen in accordance with the algorithm's performance. However, this is beyond the scope of this dissertation. Because we could not ascertain an objectively better NN model, we decided to use the NN composed of two layers with 32 neurons on future evaluations.

We also performed similar measurements with the Q-learning algorithm implementation. The environment side approach we chose is the Python approach. Our measurements were taken over 10,000 steps including training. However, we only measured a Q-learning algorithm with training because we realised it was still faster than the DQN approach on exploitation-only scenarios.

Figure 4.8 is a box plot of the agent side execution time in milliseconds of a DQN and Q-learning agent. It has a plot of a DQN agent with a training phase, and with exploitation-only. The Q-learning plot both trains and exploits. Table 4.5 contains the average and the minimum time spent outside the *step* function for the Q-learning algorithm.

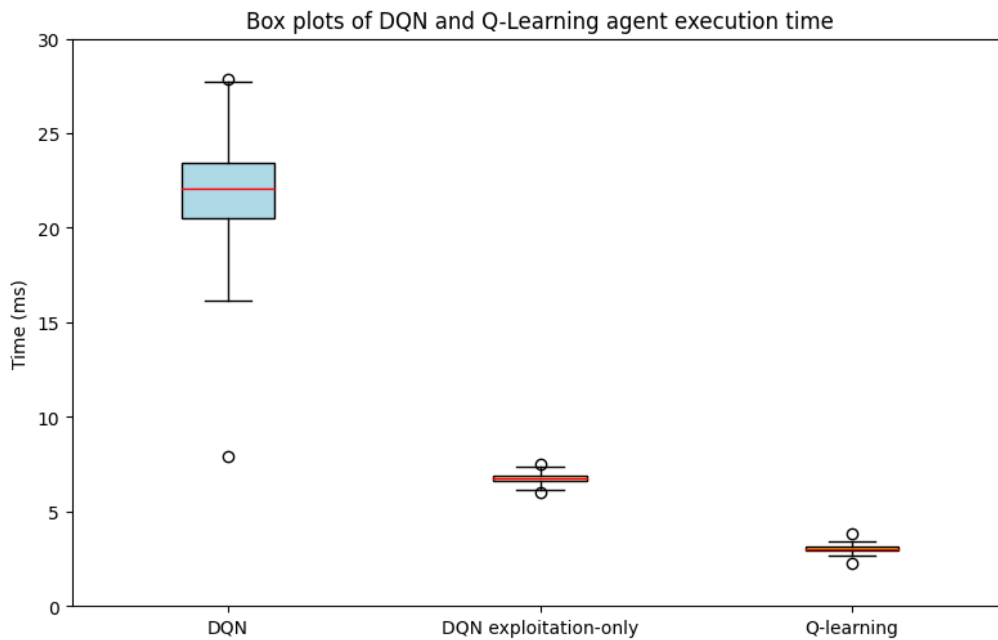

Figure 4.8: Box plots of the agent side execution time in milliseconds of DQN and Q-learning.

Table 4.5: Average and minimum time outside *step* function for the Q-learning algorithm.

Q-learning	Training & Exploitation
Average time (ms)	3.062
Minimum time (ms)	1.472

By looking at Figure 4.5 and comparing Tables 4.3 and 4.4 to Table 4.5, we can conclude that a Q-learning algorithm does indeed train faster than a DQN. In fact, a Q-learning training step is faster than a DQN exploitation-only step.

Our Q-learning algorithm did have an increasing average reward over its training period, suggesting it is a feasible option. However, we did not rigorously evaluate its performance in comparison to the DQN apart from execution time.

4.3 Final Evaluation and Comparison

The previous preliminary evaluations enabled us to make informed decisions about our choices for our final algorithm implementation. In the end, our choice was a DQN agent, similar to DARA's. We used a 32 by 32 NN model for the DQN. We used the Python version of the environment side's *step* function because it was the fastest overall.

In this section, we compare that final version of our algorithm, to a baseline as close as we can get to DARA.

4.3.1 Baseline DARA Evaluation

The original DARA implementation used the *tf-agents* module to create its agent. Just like in our own implementation, we could not get this to work. It was also implemented in a different Linux version than the one we used. This Linux version had a different version of mac80211's Minstrel implementation. DARA's implementation depended on a file created by mac80211 that did not exist in our devices. For these reasons, we had to make some changes to the DARA implementation

The first was to implement a *tensorflow* and *keras* DQN agent, similar to our own. This meant that agent side comparisons between E-DARA and DARA are not meaningful because we do not have a fair representation of DARA's original agent side. In fact, the agent side we used in DARA for it to work, is the same as our E-DARA. We used a 32 by 32 NN model for DARA as well. We were not discouraged by this, since the main bottleneck of the algorithm is the environment side, so that should be where our focus is.

The second was changes to the environment, because of the file that did not exist in our devices. Instead, a different version of the file existed. We changed the environment side to be able to apply the original logic to the new file as closely as possible. Our changes to the environment side should have a negligible affect on the results, especially considering the discrepancy between the time E-DARA and the original DARA take.

Table 4.6 contains the average time spent on the agent and environment sides of our baseline DARA implementation. Table 4.7 contains the minimum time spent on the agent and environent sides of our baseline DARA implementation. The measurements were taken over 2,000 steps in both the training and the exploitation-only scenario. The lower step count in comparison to other measurements is due to how much slower this implementation is.

Table 4.6: Average time of the environment and agent sides in our baseline DARA algorithm.

DARA	Training & Exploitation (ms)	Exploitation-only (ms)
Environment Side	578.770	578.878
Agent Side	20.670	5.603

Table 4.7: Minimum time of the environment and agent sides in our baseline DARA algorithm.

DARA	Training & Exploitation (ms)	Exploitation-only (ms)
Environment Side	556.642	563.651
Agent Side	5.384	3.974

4.3.2 Final comparison

In Figure 4.9 we can see a box plot comparing the environment side execution time of our final E-DARA implementation to DARA's. The scale of the y-axis is logarithmic.

From Figure 4.9 and Tables 4.6 and 4.7, we can see DARA is exceptionally slow when compared to any of our results. In fact, it is around seven times slower when we compare a full step time of both agent and environment side. We attribute this to the frequent use of bash scripts in DARA, which tend to be slow. The way the bash scripts are called through the subprocess module also introduces further overheads. Furthermore, execution time was not a priority when developing DARA.

When we compare even the best case scenario of minimum time of DARA's implementation, to the average *step* function time of a Python environment in Table 4.1, we obtain very promising results. Namely, that our implementation of the environment side algorithm is around eight to nine times faster than the original DARA algorithm.

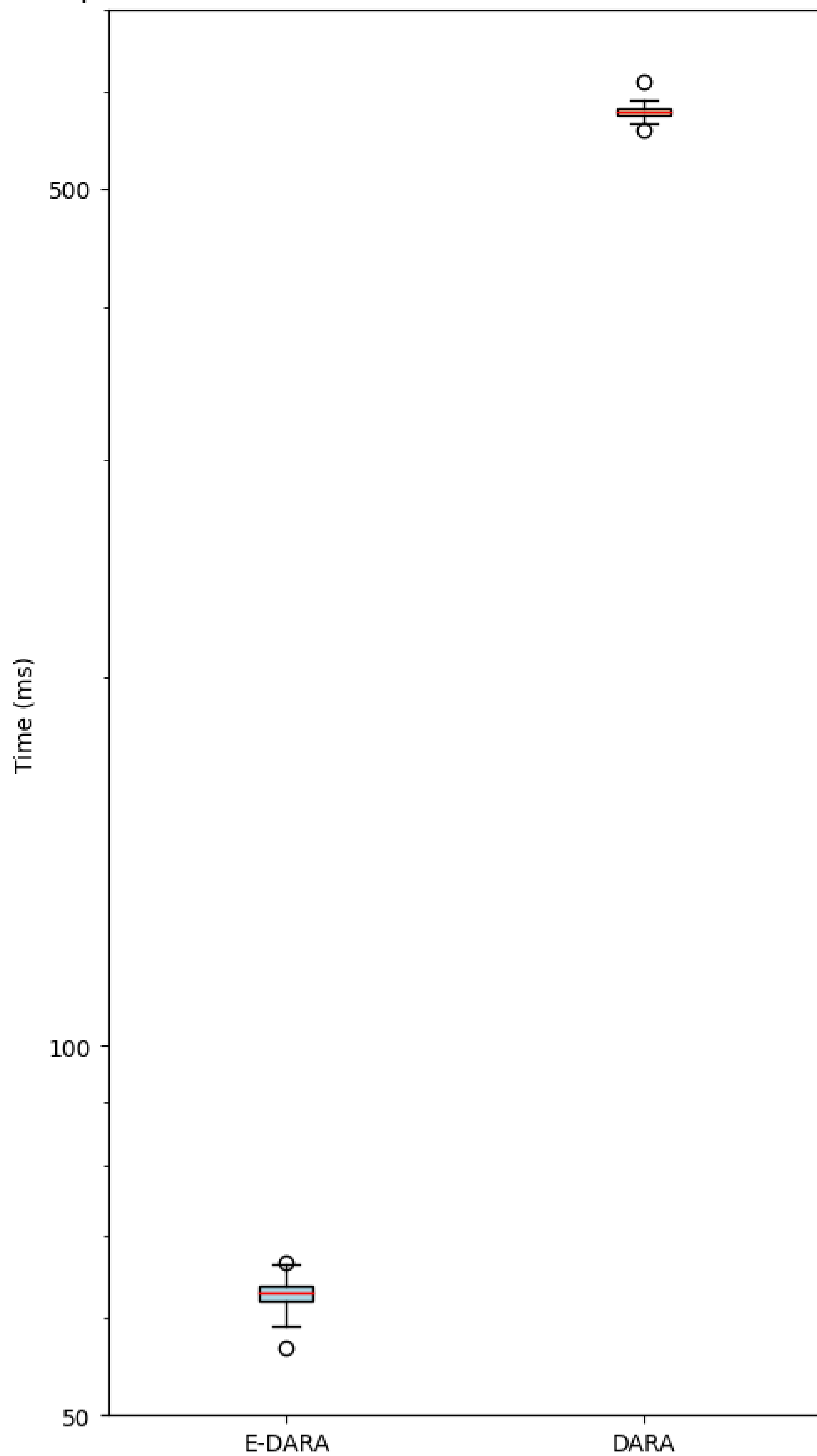

Figure 4.9: Box plots of the environment side execution time in milliseconds of E-DARA and DARA.

Chapter 5

Conclusions and Future Work

The original DARA implementation was intended to be tested in experimental scenarios. However, DARA faced challenges that impacted its performance, and the focus was turned to simulation scenarios instead. One possible reason for those challenges may have been computational delays. From our results, we concluded that the two main origins of computational delays in the DARA implementation are the frequent use of bash commands, and the time Minstrel takes to update its tables.

Bash commands are not very fast. Furthermore, they require the initialisation of a shell instance, which introduces an overhead. In DARA's Python script, it runs these bash commands through the subprocess module. This module introduces further overheads due to its code for cross-platform compatibility.

DARA got the data to calculate its reward from Minstrel update tables. This became another limitation because those tables only updated once every 100 ms. We succeeded in overcoming this limitation by modifying a Linux kernel module. This enabled access to that same data whenever we needed it.

Through these contributions, we managed to create our own implementation, E-DARA. E-DARA is faster than DARA by a factor of around seven times. If computational delays were the issue preventing DARA from being applied to an experimental scenario successfully, we believe our improvements solve that issue. The time our algorithm takes on each complete time step (agent and environment side), which is around 60 to 80 ms, is faster than the minimum time Minstrel takes on each table update (only relevant for the calculation of the reward), which is 100 ms.

Despite these satisfactory results, we have identified several shortcomings and further improvements to our work. For example, even though our algorithm is faster than Minstrel, it does not implement backup rates. Despite the higher frequency of updates from our algorithm, there may be situations where Minstrel seemingly reacts faster by switching to one of its backup rates.

Also, because the performance of the algorithm was beyond the scope of this work, we could not reach conclusive judgements on certain challenges. In particular, the challenge of picking the best NN architecture in a DQN agent, or the time the algorithm spends collecting data for the calculation of the reward.

Considering the NN influence on computational delays is very small, our suggestion is that the performance of the algorithm should be the prime factor in this decision. As for the choice of data collection time, we suggest the lowest time that still allows the algorithm to perform robustly.

We have implemented a Q-learning agent that should in theory have a similar performance to the DQN while being faster. Future work involves validating this hypothesis rigorously, which we did not do apart from the execution time comparison. If the Q-learning agent can be suitably applied to this scenario, it may be worthwhile to implement it in a programming language with faster execution speed, such as C or Rust. This should not be too hard to achieve, because the Q-learning algorithm is not very complex.

Other possible rooms for improvement we have identified are the use of a Rust extension to replace the environment side; and further modifications to the mac80211 module. The reason why Rust was not faster than the Python implementation has to do with overheads from converting variables to and from Rust-compatible functions. This happens on each Rust function call and return. If instead we implemented the whole environment side in Rust – as opposed to particular functions for each task that are called multiple times – we could possibly have an even faster environment side that is not dragged down by the aforementioned conversion overheads.

Additional modifications to the mac80211 module – particularly to enable direct action deployment – may also prove to be relevant future work. In the environment side, action deployment was the slowest task. Thus, it has the biggest room for improvement. Direct action deployment through procfs, for example, could result in bash command overheads to be eliminated from the algorithm.

We believe our advancements in this area can improve or help the performance of RL algorithms for RA. In turn, this can bring improvements to network throughput in a standard-compliant way. Deployment of such RA solutions could enhance Wi-Fi technologies without significant changes to Wi-Fi's implementation. In the future, very specific or complicated technical adjustments may not be necessary to use these solutions. Using such solutions could be as simple as installing an algorithm. This could result in a higher Wi-Fi quality of service while keeping in line with its ease of use.

References

[1] The minstrel rate control algorithm for mac80211. Accessed on 02/01/2022. URL: https://wireless.wiki.kernel.org/en/developers/documentation/mac80211/ratecontrol/minstrel.

[2] Rémy Grünblatt, Isabelle Guérin-Lassous, Isabelle Guérin-Lassous, Isabelle Guérin-Lassous, Olivier Simonin, and Olivier Simonin. Simulation and performance evaluation of the intel rate adaptation algorithm. *MSWiM*, 2019. doi:10.1145/3345768.3355921.

[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. *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications*, 2021. doi:10.1109/infocom42981.2021.9488876.

[4] Ruben Queiros, Eduardo Nuno Almeida, Helder Fontes, Jose Ruela, and Rui Campos. Wi-fi rate adaptation using a simple deep reinforcement learning approach, 2022. arXiv:2202.03997.

[5] Homepage of ns-3. Accessed on 21/08/2022. URL: https://www.nsnam.org/.

[6] Raja Karmakar, Samiran Chattopadhyay, and Sandip Chakraborty. Intelligent MU-MIMO User Selection With Dynamic Link Adaptation in IEEE 802.11ax. *IEEE Transactions on Wireless Communications*, 2019. doi:10.1109/twc.2018.2890219.

[7] Raja Karmakar, Samiran Chattopadhyay, and Sandip Chakraborty. An online learning approach for auto link-Configuration in IEEE 802.11ac wireless networks. *Computer Networks*, 2020. doi:10.1016/j.comnet.2020.107426.

[8] Raja Karmakar, Samiran Chattopadhyay, and Sandip Chakraborty. Dynamic link adaptation in IEEE 802.11ac: A distributed learning based approach. *null*, 2016. doi:10.1109/lcn.2016.20.

[9] 802.11. P802.11ax. *IEEE Draft Standard for Information Technology — Telecommunications and Information Exchange Between Systems Local and Metropolitan Area Networks — Specific Requirements Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications Amendment Enhancements for High Efficiency WLAN*, 2020.

[10] Emerson Alecrim. O que é wi-fi? (conceito e versões), March 2008. Updated in 22/05/2021. Accessed on 16/11/2021. URL: https://www.infowester.com/wifi.php/.

[11] Wei Yin, Peizhao Hu, Jadwiga Indulska, Marius Portmann, Ying Mao, Ying Mao, Ying Mao, and Ying Mao. MAC-layer rate control for 802.11 networks: A survey. *Wireless Networks*, 2020. doi:10.1007/s11276-020-02295-2.

[12] Raja Karmakar, Samiran Chattopadhyay, and Sandip Chakraborty. IEEE 802.11ac Link Adaptation Under Mobility. *2017 IEEE 42nd Conference on Local Computer Networks (LCN)*, 2017. doi:10.1109/lcn.2017.90.

[13] Giovanni Peserico, Tommaso Fedullo, Alberto Morato, Stefano Vitturi, and Federico Tramarin. Rate Adaptation by Reinforcement Learning for Wi-Fi Industrial Networks. *2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2020. doi:10.1109/etfa46521.2020.9212060.

[14] Matej Črepinšek, Shih-Hsi Liu, and Marjan Mernik. Exploration and exploitation in evolutionary algorithms: A survey. *ACM Computing Surveys*, 2013. doi:10.1145/2480741.2480752.

[15] Fed4fire+ website. Accessed on 02/12/2021. URL: https://www.fed4fire.eu/the-project/.

[16] w-ilab.2 hardware and inventory. Accessed on 26/01/2022. URL: https://doc.ilabt.imec.be/ilabt/wilab/hardware.html.

[17] Documentation of mac80211 module. Accessed on 21/08/2022. URL: https://wireless.wiki.kernel.org/en/developers/documentation/mac80211.

[18] Homepage of tf-agents. Accessed on 21/08/2022. URL: https://www.tensorflow.org/agents.

[19] Homepage of tensorflow. Accessed on 21/08/2022. URL: https://www.tensorflow.org/.

[20] Documentation of the subprocess module. Accessed on 21/08/2022. URL: https://docs.python.org/3.8/library/subprocess.html.

[21] Chi-Yu Li, Syuan-Cheng Chen, Chien-Ting Kuo, and Chui-Hao Chiu. Practical Machine Learning-Based Rate Adaptation Solution for Wi-Fi NICs: IEEE 802.11ac as a Case Study. *IEEE Transactions on Vehicular Technology*, 2020. doi:10.1109/tvt.2020.3004471.

[22] Raouf Boutaba, Mohammad A. Salahuddin, Noura Limam, Sara Ayoubi, Nashid Shahriar, Felipe Estrada-Solano, Oscar M. Caicedo, and Oscar Mauricio Caicedo. A comprehensive survey on machine learning for networking: Evolution, applications and research opportunities. *Journal of Internet Services and Applications*, 2018. doi:10.1186/s13174-018-0087-2.

[23] Homepage of keras. Accessed on 21/08/2022. URL: https://keras.io/.

[24] Documentation of the re module. Accessed on 21/08/2022. URL: https://docs.python.org/3.8/library/re.html.

[25] Nicholas Renotte. A tutorial for keras-rl with openai gym. Accessed on 28/07/2022. URL: https://github.com/nicknochnack/TensorflowKeras-ReinforcementLearning.

[26] Homepage of numpy. Accessed on 21/08/2022. URL: https://numpy.org/.

[27] Documentation of the timeit module. Accessed on 21/08/2022. URL: https://docs.python.org/3.8/library/timeit.html.