POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Design of a Reinforcement Learning Framework to Automatically Interact with IoT Devices

Supervisors

Prof. Marco MELLIA

Prof. Luca VASSIO

Prof. Idilio DRAGO

Candidate

Giulia MILAN

December 2020

In IT systems, the presence of Internet of Things (IoT) devices is exponentially growing. Most of them are custom devices, and they rely on proprietary protocols, often closed or poorly documented. Here we want to interact with such devices, by learning their protocols in an autonomous manner.

Actually, a great heterogeneity on IoT devices and protocols exist. For example, different protocols can rely on different infrastructures, different physical, network and application layers. Moreover, each protocol has its own format and syntax. Also, for each protocol we can define multiple and different state-machines. We define the state-machine of a protocol as multiple series of states linked by one or more sequences of commands. These commands can be exchanged through that protocol to complete a predefined task. A task can be identified as a path inside the state-machine. The sequence of commands could change the state of the targeted IoT device, that is represented by some properties specific to that device, following a certain path - i.e., completing a task - inside the state-machine.

In this work, our goal is to learn the protocol of a generic IoT device while minimizing the interaction with the device. For our case, considering paths inside the state-machine of a protocol, we want to learn the most convenient paths, minimizing the number of interactions made with the remote peer in order to learn some useful commands of the protocol. This thesis aims to find a general approach to achieve this goal for multiple protocols using Reinforcement Learning (RL) techniques. Reinforcement Learning is a machine learning technique in which an agent takes actions in a particular environment to maximize a long-term reward.

The RL-based approach can be used with communication protocols in general. Among all protocols, we focus mainly on IoT protocols, as a great variety of them exist. We design a framework which uses RL to learn how to speak with unknown IoT devices automatically in the shortest possible time. The RL framework is designed to implement multiple RL algorithms and to try to automatize the interaction of the framework with the IoT devices present in a Local Area Network (LAN). We decide to support four RL algorithms, which are SARSA, Q-learning, SARSA(λ) and Q(λ), in order to prove the correct working of our approach and to give a comparison of the performance of these algorithms when used in a protocol-related context. To use these techniques, we give to RL algorithms an initial knowledge about the IoT protocols to target. We assume there exists a dataset with valid protocol messages of different IoT devices. However, we have no further knowledge on the semantics of such command messages, nor on whether particular devices would accept the commands. This dataset will be stored into a dictionary inside our framework, with the appropriate modifications.

In the future, if the framework is proved to efficiently work, it could be useful for monitoring purposes on attacks to IoT devices. In fact, most of these devices are not designed to implement security controls against cyberattacks [1]. This leads to possible security threats and abuses, as it happened in 2016 for the Mirai attack, in

which a botnet was built taking advantage of insecure IoT devices [2]. Indeed, this RL framework could be implemented inside a honeypot, which could monitor the activity of possible attackers to these IoT systems. Because of the heterogeneity of IoT protocols, this framework would help into the development of a honeypot which communicates with unknown attackers through a huge variety of protocols.

Therefore, we design our framework to be able to support multiple IoT protocols and multiple RL algorithms, in a way that should be flexible to the addition of new targeted IoT protocols. After this design process, we implement a first component of the RL framework, based on the Yeelight protocol. With this component we showed that RL techniques can actually be used to learn to interact with IoT protocols, learning to follow paths inside predefined state-machines defined for the protocols. In fact, after a learning process, our implemented framework is able to interact with Yeelight devices, learning useful actions and sending them some particular combinations of commands in order to change the status of the devices following the protocol-specific state-machine.

From our results, we found out that, using RL algorithms optimized with tuned parameters, our framework is able to learn something after sending about 500 commands to a single Yeelight device, for non-trivial state-machines. We showed that all implemented RL algorithms - SARSA, Q-learning, SARSA(λ) and Q(λ) - are suitable for our case scenarios. Also, we demonstrated that the performance of the algorithms strictly depend on the complexity of the state-machine chosen and on the length of the optimal path or other paths to be learned inside the state-machine. For longer paths and more complex state-machines, Q-learning performs slightly better. Moreover, since the Yeelight IoT protocol defines a maximum rate on the commands to be sent to devices, we found out that our implemented algorithms take at least 50 minutes to complete the entire learning process, after fixing the number of iterations - called episodes - for the RL algorithms.

Acknowledgements

Ringraziamenti

È giunto il momento dei ringraziamenti, ed in questo particolare momento storico e personale, questi si fanno più veri e necessari. Fino a qualche tempo fa, i ringraziamenti per questa tesi di laurea sarebbero stati completamente diversi. Eppure, forse è stato meglio così. Scrivo questi ringraziamenti in procinto di un nuovo lockdown, mentre sono in isolamento per un "indeterminato" come risultato, in un limbo perenne che sembra accompagnarci ed accompagnarmi dall'inizio di questa pandemia. La verità è che questo ultimo periodo mi ha fatto capire chi davvero ha reso il conseguimento di questo titolo un po' meno pesante, tra persone da poco entrate a far parte un po' di più della mia vita a persone che sono con me da fin troppo tempo.

A mia mamma Rita e mio papà Alex, i miei genitori, le persone che più di ogni altre hanno reso possibile questo piccolo sogno, questa piccola ambizione. Diciamo che ci sono un milione o un miliardo di vecchie lire di motivi per cui dovrei ringraziarli: sicuramente dall'avermi messa al mondo, dall'avermi dato tutto l'amore, l'educazione e l'istruzione che mi hanno portato fino a questo momento, per non avermi mai forzato nello scegliere la mia strada, per avermi posto le domande che mi hanno fatto riflettere su quali avrei voluto che fossero le risposte, per avermi proposto soluzioni nei momenti in cui non vedevo via d'uscita. Per non avermi mai messo fretta, per avermi dato tutto il tempo del mondo, per avermi tranquillizzata quando tutto sembrava andare in direzione opposta alla mia sanità mentale e alla fine di questo percorso. Ma forse, il motivo più importante per cui non basterà una vita per ringraziarli, è come abbiano sempre ininterrottamente creduto in me, più di chiunque altro e sicuramente più di me stessa, dandomi tutta la forza ed il supporto per fare del mio meglio, e mi sembra che ci siano riusciti. Non avrei potuto chiedere di meglio.

A mio fratello Marco, che mi ha dato un po' di vita con la rubrica "Chi ha preso il coronavirus oggi", che si è fatto ore di videochiamate durante la quarantena, che ha gioito delle mie gioie (lo spero) e che è stato anche un confidente ed un amico, più di quanto avrei potuto immaginare.

A Dani, una persona che si è intromessa nella mia vita senza che nemmeno me ne accorgessi e che ancora non ha intenzione di andarsene, perché nei momenti in cui non è rimasto più nessuno, ha sopportato egregiamente ogni lamentela ed anzi ha partecipato alle mie gare di lamentele, mi ha spronato a rimettermi al lavoro quando credevo di non riuscire a fare un passo in più, ha creduto in me e forse mi ci ha fatto credere un po' di più anche a me. C'è stato, e questo già sarebbe stato abbastanza.

A Stefi, che le voglio bene come se non fossi mai partita da Spilimbergo, che vorrei sempre portare con me in ogni posto nel mondo, che ogni volta che le parlo o che torno a casa lei c'è, che ascolta i miei drammi e che qualsiasi cosa faccia è orgogliosa di me, come se fossi davvero una sorella per lei.

A Matte, perché è stato con me per metà del percorso, perché so bene quanto mi abbia supportato e sopportato, quanto sia fiero di me, anche adesso, e perché occuperà sempre una parte del mio cuore.

A Enri, perché ha sempre reagito con entusiasmo alle mie notizie e perché in un mio momento di estrema fragilità mi ha donato della bellezza di un abbraccio che ha riempito il vuoto che altri avevano scavato.

A Sack, per essere stata la prima persona che ho conosciuto al Politecnico e per essere stata quella più gentile, più generosa e altruista che abbia mai conosciuto.

A Fra, perché non dimentico quanto il mio primo anno di magistrale sia stato duro, e a quanto sarebbe stato diverso senza il suo supporto, quasi continuo, dei primi mesi.

Ai tamburi di Spilimbergo e a quelli di Grugliasco, perché durante un periodo di esami, di stress, di ansie, non c'è niente di piu bello di spaccare i timpani degli altri tutti assieme.

Ai molesti, che mi hanno portato ad avere male agli addominali da quanto mi hanno fatto ridere durante la quarantena.

Ai "ragazzi" dell'ufficio, per avermi fatto sperimentare un ambiente di lavoro estremamente interessante e leggero, e per aver reso il lavoro decisamente piu divertente.

A Marco, Idilio e Luca che mi hanno dato l'opportunità di buttarmi a capofitto in qualcosa di cui non conoscevo nulla ed uscirne (spero) con un gran successo.

Ognuno di voi è stato parte di questo percorso, che è stato impegnativo ma senza alcun dubbio soddisfacente, e solo l'assenza di una delle persone che ho ringraziato avrebbe reso questo puzzle incompleto.

Per tutto questo e per molto altro, *grazie*.

Table of Contents

List of Tables

List of Figures

Acronyms

RL
Reinforcement Learning

IoT
Internet of Things

LAN
Local Area Network

MDP
Markov Decision Process

MC
Monte Carlo

TD
Temporal Difference

IRL
Inverse Reinforcement Learning

Chapter 1

Introduction

The Internet of Things (IoT) is a term that refers to the network of physical objects embedded with sensors, software and other technologies for the purpose of gathering information, exchanging and analyzing data with other IoT devices or systems over the Internet. This term was first coined in 1999 by Kevin Ashton, but, according to Cisco Systems, IoT was actually born between 2008 and 2009 [3]. Since then, the presence of IoT devices is not negligible: according to [4], by the end of 2020 there will be between 20.4 billion and 31 billion IoT devices connected to the Internet, while by 2025 the trends suggest that this number will rise to 75 billion of IoT devices. These devices have been introduced in many IT systems, from smart homes to drones and medical devices. This adoption is expected to keep growing even faster in the future thanks to the availability of higher bandwidths, smaller latency and new networking capabilities, due to new technologies as 5G mobile networks [5].

Not only a huge amount of devices exists, but they also belong to many different brands. Devices belonging to different brands could rely on different protocols and different technologies. There could be multiple ways to differentiate among different IoT protocols. For example, IoT protocols usually rely on Bluetooth, ZigBee or Wi-Fi. Moreover, they could also rely on different application protocols, such as MQTT, CoAP and XMPP. Since these are only a limited number of possibilities among all available possible customizations, there exists a lot of different combinations between technologies, network protocols, application protocols and semantics. Finally, most of the devices rely on proprietary protocols, which are often closed and poorly documented.

For these reasons, we want to find an automatic approach to communicate with unknown IoT devices, learning their protocols. Our approach consists in using Reinforcement Learning (RL) techniques to learn the protocol of a generic IoT device while minimizing the interaction with the device.

The heterogeneity of IoT devices and protocols leads to have for each IoT

protocol different message formats and different syntax for fields in messages. It also leads to have for each device different possible states, upon which we can build a state-machine for a certain device and protocol.

To define what we mean by state-machine we need to define the state for an IoT device. We consider the state of a device as the set of all properties that the IoT device can have, which are used to control the behaviour and settings of the device. For example, in case of a smart light bulb, the state can be defined by the values of its power status, its color and its brightness. From this, we define the state-machine of a protocol as a model containing multiple series of states linked by sequences of commands that can be exchanged through that protocol. A collection of ordered states, linked by commands, is defined as "path" or "task". Commands change the status of the IoT device, and therefore they change the current state of the state-machine. One possible path inside the state-machine defined for a light bulb could be followed using the following sequence of commands: switch the light on, perform some operations to change the color and the brightness of the bulb and then switch the bulb off again.

In our work, considering paths inside the state-machine of a protocol, we want to learn the most convenient paths, minimizing the number of interactions made with the remote peer in order to learn some useful commands of the protocol. Reinforcement Learning (RL) will be used to achieve this goal for multiple protocols, that is a machine learning technique in which an agent takes actions in a certain environment to maximize a long-term reward. To use these techniques, we give to RL algorithms an initial knowledge about the IoT protocols to target. We assume there exists a dataset with valid protocol messages of different IoT devices. However, we have no further knowledge on the semantics of such command messages, nor on whether particular devices would accept the commands.

This task of learning automatically an unknown IoT protocol could then help to support security activities. For example, implementing our approach inside a honeypot would help a honeypot to attract attackers targeting IoT devices. That would be interesting because, often, IoT devices have few or no security controls against cyberattacks, as pointed out by [1], and on the other side IoT devices are attractive targets for attackers, both because of the user's dependence on their devices and because of the power of decision that we give to these devices - e.g. IoT devices could also be used in an autonomous vehicle, which drives for us [6]. According to [5], in 2020 IoT devices make up about 33% of infected devices, while in 2019 this number was only about 16%[1]. This scenario leads to the need of protecting IoT devices, but because of their heterogeneity, effective mechanisms

[1]These statistics have been computed aggregating data from monitoring network traffic on more than 150 million devices globally.

of defense become complicated and, for this reason, our solution could be used to support these security mechanisms.

Coming back to our work, since we assume we have a dataset with valid protocol messages, this means we assume we already know the format of messages and the syntax of IoT protocols. From this initial knowledge, which is stored inside a dictionary, we design a RL framework that learns best paths inside the state-machines of the protocols, using RL algorithms. Our framework should perform a learning process in the shortest possible time, minimizing the number of interactions with the remote IoT device. Therefore in this context, we actually mimic the behaviour of an attacker, which tries to explore the state-machine of the IoT device it is trying to communicate with.

To prove the correct working of our approach, we start implementing our framework targeting a single IoT protocol: the Yeelight protocol. Moreover, in order to evaluate multiple techniques and compare their performance, we implement four RL algorithms: SARSA, Q-learning, SARSA(λ) and Q(λ). The framework we implemented is able to learn a simple IoT protocol, given possible protocol messages, with generally a limited number of interactions with devices in the local network. After performing an optimization phase on the RL algorithms, our RL framework is able to learn after sending from 200 to about 500 commands to a single Yeelight device, depending on the complexity of the state-machine the RL algorithm should explore. We showed that, for our scenario, all four implemented RL algorithms are suitable. Also, we noted that the performance of the algorithms strictly depend on the complexity of the state-machine chosen and on the length of the optimal path or other paths to be learned inside the state-machine. For longer paths and more complex state-machines, Q-learning performs slightly better, but this does not happen for all cases. Practically, since we focus on the Yeelight protocol, which defines a maximum rate on the commands to be sent to Yeelight devices, our framework takes at least 50 minutes to complete the entire learning process, after fixing the number of iterations - or episodes - for the learning process.

Before exploring the design and implementation of the framework, in chapter 2 we completely define our scenario for this project, and present some related work. Next, we focus on the background and the methodology followed in our work, in chapter 3. For this part, we introduce Reinforcement Learning, and present the RL algorithms that we used: SARSA, Q-learning, SARSA(λ) and Q(λ). After, in chapter 4 we present a toy-case in order to explain how RL can be used in a general way in protocols. For this section, we focus on the TCP protocol. Then, we will actually present the details of our framework, describing the design choices. For usability purposes, we decided to build the framework upon multiple modules, each in charge of realizing some specific functionalities. In chapter 5, we describe these modules and put the basis for a framework extensible for multiple IoT protocols. After this design process, in chapter 6 we show the first implementation of this

framework, focused on a single IoT protocol: the Yeelight protocol, which will be described in details. We choose this protocol because it is well documented and it has a clear and strict structure. Using the algorithms already cited, we test the RL agent with some colored Yeelight bulbs, all connected to the same local network. In order to evaluate the efficacy and efficiency of the framework, we will eventually present the results achieved by the RL framework for the Yeelight protocol. In chapter 7, we will compare the performance of multiple algorithms, multiple parameters configurations and tasks for different state-machines of the protocol. We will also evaluate the time necessary to perform the learning process and the traffic generated by our RL algorithms. Finally, we will suggest some enhancements and some future work.

Chapter 2

Problem statement and related work

In this work we are interested in finding a general approach for interacting with devices, by learning their protocols. Since this is such a great and inclusive task, we decided to shrink this task into targeting solely IoT devices, thus IoT protocols. Here we define in details the goal of our project and the scenario in which we will design our solution afterwards. Finally, in this chapter, we will discuss some related work.

2.1 Goal and scenario

Our goal is to learn to interact with unknown devices in an autonomous way, shrinking our target to IoT devices. This learning task should be applicable to multiple IoT protocols. In this way we would have a system capable of speaking different IoT protocols, learning them in an automatic manner.

One of the problems related to this task is that multiple IoT devices can have all different purposes and can rely on different IoT protocols. Furthermore, different protocols rely on different technologies, network and application protocols. Also, messages for different protocols have different semantics, different formats and syntax. Moreover, we can define one or more possible state-machines for each protocol and its relative IoT devices. Indeed, a great heterogeneity among IoT protocols exist.

In mathematical terms, a state-machine is a behavior model, that consists of a finite number of states. A machine performs transitions between states, producing a certain output, based on the current state and on a input given to the machine [7]. Here we will refer to this structure to define state-machines for protocols and devices. We define the state of an IoT device as the set of all properties values that

the device has. For example, in case of a smart light bulb, the state can be defined by the values of its power status, its color and its brightness. In our context, the state should comprehend all values necessary and sufficient to describe the current configuration of the device and its past configurations. Given the definition of a state, we define the state-machine of a protocol as multiple series of states, linked by sequences of ordered commands that can be exchanged through that protocol in a correct way. This sequence of commands allows to change states inside the state-machine, identifying a path inside the state machine. These commands are the possible inputs that can be given to a state, that could change the state of the targeted IoT device, performing a transition between the current state and the next state. The next state represents the output produced by the state-machine. The commands for a protocol should be messages which have a correct format and syntax, and which are valid for the specific IoT device that we are targeting, in order to correctly move inside the state-machine.

Because of the diversity of IoT protocols, being able to deal with a huge variety of them is a challenging task. Therefore, in this work we assume there exists a dataset with valid protocol messages of different IoT devices. However, we have no further knowledge on the semantics of such command messages, nor on whether particular devices would accept the commands. Giving this knowledge and assuming we now the low-level infrastructure of the protocol - e.g., if it is based on Wi-Fi - we want to learn paths inside the state-machine of that protocol. This goal should be reached focusing on these key points:

- The number of interactions made with the remote peer to learn some useful messages should be minimized;
- The time needed for learning useful messages should be reasonable;
- Once some useful messages are learned, the "learner" should be able to send a correct sequence of messages, that follows a predefined path - e.g., completing its task - inside the state-machine defined for the specific protocol.

To achieve this goal, this thesis wants to find a general approach for multiple protocols using Reinforcement Learning (RL) techniques.

If the solution works, this approach could be used to implement some mechanisms of defence in order to prevent and analyze possible attackers targeting IoT devices. A possible defense mechanism is to perform a monitoring activity on attackers, exploiting honeypots. Honeypots are decoys that mimic a target for hackers and they can exploit intrusion attempts to gain information about the attackers, learning about their intentions, their actions and learning about new kinds of attacks [8]. A honeypot for IoT devices should implement some techniques to be able to engage attackers as long as possible, pretending to be the targeted IoT device, responding to the attacker with proper responses as the attacker expects the IoT device to

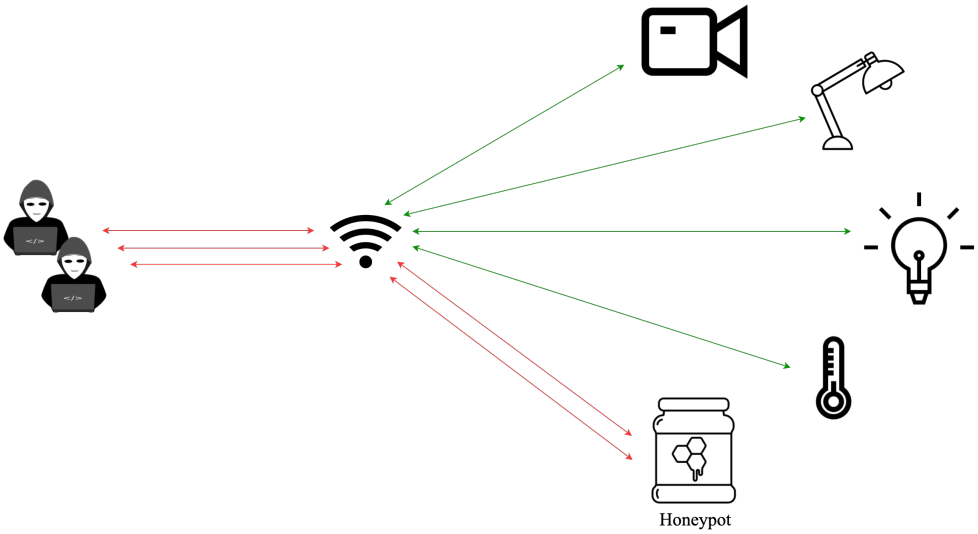


Figure 2.1: General scenario with IoT devices and a honeypot connected to a LAN. Attackers target the honeypot without knowing it is a honeypot.

respond, with the same protocol that the attacker is targeting. An example of this situation can be found in Figure 2.1, showing a Local Area Network (LAN) to which multiple IoT devices are connected. These devices have different purposes. As showed in the figure, they could be smart lamps, smart cameras, etc. These devices could also belong to different brands, hence they would communicate with different protocols, therefore our solution could be adapted inside the honeypot.

In our scenario, instead, we are actually playing the role of an attacker, wanting to target multiple IoT devices and exploiting RL techniques. To use these techniques, which need a learning process, it is necessary to communicate in real-time with IoT devices, sending commands inside the LAN and collecting feedback directly from the devices. Since our solution aims to learn and explore the state-machine for different protocols, the LAN can contain multiple devices belonging to different IoT protocols.

2.2 Related work

To the best of our knowledge we found a limited number of research studies which focus on the usage of Reinforcement Learning techniques for the IoT context. Most of the studies concerning these topics focus on the development of IoT honeypots using RL methods.

One of the first works about RL and IoT honeypots is presented in [9]. It

focuses on the adaptation of honeypots for improving the security of IoT devices. In this study, they argue that because of the heterogeneity of IoT devices, both low-interaction and high-interaction honeypots are not suitable for the IoT context. Hence, their solution is to use machine learning technologies to automatically learn knowledge about the behaviour of IoT devices and build an "intelligent-interaction" honeypot. Among the machine learning techniques explored, some RL techniques are also used.

In 2020, a new research study has been presented, regarding IoT attacks and relative mechanisms of defense [10]. This work states that the IoT attack detection for IoT devices is more challenging than the traditional attack detection, because of multiple factors. First, multiple vulnerabilities are exposed because of the diversity of platforms, protocols, software, and hardware of the IoT devices. Second, low-rate attacks are often used by IoT attackers and these are more challenging to detect than high-rate attacks. The third factor considers the evolution of attackers, that are becoming more intelligent, in the sense that they can dynamically change their attack strategies based on the feedback of the environment, in order to avoid being detected by defense mechanisms. Because of this behaviour, it is more challenging for the defender to discover a consistent pattern that can be used to identify the attack. In order to adapt to these unique characteristics of IoT attacks, the research study in [10] proposes an attack detection model, based on Reinforcement Learning techniques, which is in charge of automatically learning and recognizing the transformations of the attack pattern.

With respect to these studies, we are instead mostly interested in proving the correct working of our approach, evaluating multiple alternatives for RL techniques, performing some comparisons among results obtained with all of these. Using these techniques, we then want to deal with different IoT protocols and we are concerned in learning multiple paths inside non-trivial state-machines. In some of the cited works, as [9], the goal is to engage attackers for a limited number of steps, like one or few. This makes their honeypot learn very short paths, while RL methods can also be used for more complex situations, in which longer paths are involved.

Concerning the vastness of the IoT context, in [11] the authors propose an approach to facilitate the interoperability between heterogeneous IoT devices, exploiting the Q-learning RL algorithm.

Without focusing on the IoT world, multiple studies about developing honeypots exploiting Reinforcement Learning techniques exist. In 2014, a self-adaptive honeypot was presented in [12]. The honeypot the authors developed is capable of learning from the interaction with attackers. Their honeypot emulates a SSH server, while exploiting one RL algorithm to interact with attackers, the SARSA algorithm. Upon this work, later in 2018, they proposed an improved self-adaptive SSH honeypot [13], which uses a different RL algorithm, Deep Q-learning. This

algorithm embeds the standard RL algorithm, the Q-learning, with neural networks. In these two works, the reward function for the RL algorithms implemented was purely subjective. By this assumption, they showed that the performance of the honeypot is strictly related on how the reward function was modeled. As a consequence of this conclusion, in 2019 a new research study [14] showed how the problem of the definition of the reward function could be overcome with the additional use of a technique called Inverse Reinforcement Learning (IRL). This technique modifies the reward, trying to determine the underlying structure of the reward, given the optimal policy.

Another related work is [15], in which the authors design an adaptive honeypot. They modeled the attacker as a Semi-Markov Decision Process (SMDP) and they applied Reinforcement Learning to learn the optimal policy and value for the SMDP.

Moreover, another study of 2018 suggests the deployment of a honeypot that uses Reinforcement Learning in order to hide the honeypot functionality to attackers [16]. This adaptive honeypot learns the best responses to overcome initial attempts of the attackers who try to detect the honeypot. Following this work, in 2020 the same research group proposed a new framework [17] for the development of agile and adaptive honeypots against evolving malware. They stated that malware-propagation and compromise methods are in most cases automated and repetitious, and this repetitive characteristic can be exploited by using Reinforcement Learning algorithms, such as SARSA and Q-learning, embedded in honeypots. Concerning the RL part of this work, the RL-based honeypots have some points in common to our RL framework, since they provide the support for the same RL algorithms as us. However, their work is not focused on IoT protocols.

Finally, studies related to developing IoT honeypots exist, which are not bound with RL techniques. For what concerns these IoT-specific honeypots, the goal is to realize honeypots able to manage the diversity of IoT protocols. For example, in [18] they propose an IoT honeypot to attract and analyze Telnet-based attacks against various IoT devices. Since multiple attempts to realize IoT honeypots exist [18][19], the study in [6] aims to analyze current solutions specific to IoT honeypots and other solutions that can be adapted to the IoT context, in order to set the basis for a methodology that allows a broader-scale deployment of IoT honeypots.

Chapter 3

Background and methodology

After having explained the goal and the scenario for this project, we present the methodology we follow for solving that goal.

In section 3.1, we first provide a background on Reinforcement Learning (RL) in general and describe the main RL techniques. Since our framework uses multiple RL algorithms, we present them in this chapter, highlighting the differences among them. Moreover, we define some metrics for evaluating the results we will present later in chapter 7.

3.1 Reinforcement Learning

Reinforcement Learning (RL) is a machine learning technique in which an agent takes actions in a particular environment to maximize a long-term reward. The environment defines the state in which the agent is and the new state in which the agent ends up after performing an action. In RL, the learning process is goal-directed [20], in the sense that the agent aims to learn how to achieve a final goal. The goal is expressed in terms of reward, that the agent obtains interacting with the environment, reaching a certain state or following a certain sequence of states inside the environment.

A fundamental concept in RL is that the RL agent learns from its past experience [21]: the agent learns from the consequences of its actions, instead that being explicitly taught, and it can select its actions in two ways. The first way is based on exploitation, in which the agent selects an action on the basis of its past experiences. The second way instead is based on exploration, in which new actions are selected. This learning strategy can be identified as trial and error learning. As a consequence of an action or of multiple actions, the agent receives a feedback

from the environment in the form of a numerical reward. This value encodes the success of an action's outcome, and the agent tries to learn to select actions that maximize the accumulated reward over time [21].

A common scenario in which RL can be used is a program that plays chess. In this game, the RL agent is represented by a player of the game. The states of the RL problem to solve are identified by the positions of the chess pieces, while the actions that the RL agent can perform are the possible moves that each of the pieces of the agent can do. For example, the possible moves for a pawn are: perform one pass forward one square, perform two passes forward two squares on its first move and move diagonally forward only when capturing an opponent's chess piece[1]. This program interacts with its environment, and receives a reward depending on how it performs at each play. For example, it could receive a positive reward if it wins a game. Conversely, the agent could receive a negative reward - a penalty - for being checkmated.

To make an analogy, we can say that this machine learning technique is inspired by behavioral psychology, because it emulates a learning process similar to how children learn to perform a new task, without giving them explicit information on how to perform that task. The RL agent - as the children in the previous example - has to work through the problem on its own and this approach makes Reinforcement Learning in contrast to other Machine Learning techniques. These techniques instead are not emulating the behaviour of humans, but trying to mimic the working of the brain neural system, as spiking neural networks do. These networks copy the concepts of neurons and synaptic states from the brain, simulating the way in which neurons transmit information among each others [22].

In supervised learning, for example, the learning process learns from a training set of labeled records provided by a knowledgeable external supervisor [20]. Each record in the training process is a description of a certain situation, together with a label of the correct action the system should take in that situation. Typically, the correct action is intended as an identifier of the category to which the record belongs. Thus, in this kind of learning, the goal of the system is to extrapolate or generalize its responses, in a way that is acts correctly in situations that are not present in the training set [20]. In RL, instead, the environment does not provide to the learning agent any information about which is the correct action that the system should take in that situation, but it returns only a numerical value, the reward. To discover what is the best action to take in that particular situation, the agent should hypothetically try all possible series of all available actions.

Unsupervised learning is also different from Reinforcement Learning. Most of the times the goal of unsupervised learning is to find hidden structures in collections

[1]In this example we exclude other moves of the pawn, as the en passant move.

of unlabeled data, while Reinforcement Learning focuses on maximizing a reward, not looking for hidden structures.

Furthermore, differently from other kinds of learning, RL is an "interactive" problem in which it is often impractical to obtain complete information about the desired behavior that agent should follow [20]. This happens because this information cannot be both correct and representative of all the situations in which the agent has to act, since the RL agent acts in uncharted territory. For this reason, an agent must be able to learn from its own experience. Hence, in order to build an experience, the RL agent has to deal with the trade-off between exploration and exploitation: to obtain a big amount of reward, a RL agent must prefer actions that it has already tried in the past and that it found to be effective in producing a substantial reward; to discover such actions, instead, it has to perform actions that it has not selected before. Therefore, the agent should *exploit* what it has already experienced in order to obtain a great reward and try to improve the current best solution, but it also has to *explore* in order to be able to select actions better in the future, finding new and better paths. We can deduce that neither exploration nor exploitation can be pursued exclusively without failing at performing the task that the agent wants to learn. The agent could fail multiple times while trying a variety of actions and progressively favoring those actions that appear to be the best. Moreover, in order to obtain a reliable estimate of the reward obtained by choosing a specific action, each action must be tried many times.

3.1.1 Main concepts

In this section, after providing some knowledge about the birth of Reinforcement Learning, we want to go into the details of this technique, providing some definitions for the terms that we will use from now on to describe our work.

Historically, two main disciplines contributed to RL [21]: optimal control theory and the animal learning by trial-and-error strategy. Optimal control problems have been addressed by methods of dynamic programming in later 1950s, and they belong to a large specific area on their own. Trial-and-error learning instead has roots in Psychology, especially in the branches of Classical Conditioning and Instrumental Conditioning. The first discipline - optimal control theory - comprehends mathematical approaches, which combined with the animal trial-and-error learning ended up in 1977 in a new computational method, the Temporal Difference (TD) Learning. TD learning is the first developed RL technique, which we will further describe in subsection 3.1.2. After TD learning, other different techniques have been developed, creating this new kind of learning approach: Reinforcement Learning.

In RL, as explained previously, the algorithm or agent learns by experience, interacting with an environment and receiving a reward based on the actions it

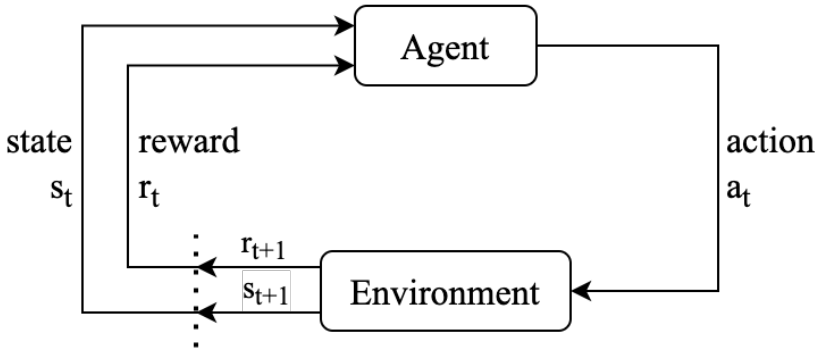


Figure 3.1: Basic schema of the Reinforcement Learning problem [20].

performed. A basic schema of RL is provided in Figure 3.1, in which the environment defines the current state s_t at time t in which the agent is. In this figure, the time is discrete, and at each time step t the RL agent performs an action a_t. After it, the environment returns to the agent the information about the new state s_{t+1} at time $t + 1$ and the reward r_{t+1}.

The ideal case study for RL problems is when the environment can be formulated as a Markov Decision Process (MDP) [20]. A Markov decision process (MDP) is a probabilistic model of a sequential decision problem [23], where states can be fully perceived, and the current state and the action selected determine a probability distribution on future states. This means that the result of applying an action from a certain state depends only on the current state and action, and does not depend on preceding visited states or performed actions [20]. If the probability distribution of future states of the environment depends only upon the current state and not on the sequence of events that preceded it, we say that the environment is an MDP.

Now we can give a more precise definition for the main components characterizing a RL problem, presented in Figure 3.1. The protagonist of RL is the *agent*, which is the learner and the decision maker. The *environment* is where the agent learns and decides what actions to perform. An *action* defines what the agent can actually do and the *state* is defined as the state of the agent inside the environment. For the learning process to work, the agent must be able to sense the state of its environment to some extent and must be able to perform actions that affect the state. Finally, the *reward* is (usually) a scalar value provided by the environment to the agent for each time step, that is equivalent to each action the agent has selected. The sign of the reward is more important than its value, since a positive reward could represent a good event - like pleasure in biological systems - while a negative reward could represent a bad event - like pain or punishments in biological systems. The reward signal defines the goal of a RL problem, since the only objective of the

agent is to maximize the total reward it receives over the long run [20].

Beyond these main concepts, other important elements of a RL system exists, like the policy, the value function and, in some cases, the model of the environment.

A *policy* defines the behaviour of the RL agent at a given time. Indeed, actions are selected by the agent according to a policy, which could also change over time [21]. Practically, a policy can be represented in some cases either as a simple function or a lookup table, while in other cases it may involve extensive computation, such as a search process [20]. For example, in the case in which we want to use RL to control the movements of the arms of a robot, the states are represented by all possible positions in the space of its arms. These positions are identified by continuous values, and these would lead to a continuous state space. In this case a lookup table is infeasible to represent the policy for each possible state, but a continuous function is necessary. The policy can be viewed as the core of a RL agent, in the sense that it alone is sufficient to determine its behavior.

Given the definition of policy, we can introduce the notion of value function. A *value function* is a mapping from states to real numbers, where the value of each state represents the estimation of the expected long-term reward - also called return - achieved starting from that state and then executing a particular policy.

The policy of an agent can be modified depending on the reward signal and on the value function [20]. The reward signal is the primary way for altering the policy: for example, an action followed by low reward may be replaced in the policy with another action, which will be selected when that same situation occurs in the future. The reward signal indicates what is good immediately while the value function describes what is good in the long run: the value of a state is computed as the total amount of reward an RL agent can expect to accumulate over the future starting from that state, and not only as the single reward obtained after performing one action. To make this distinction clearer, we present an example: there might be situations in which a state obtains a low immediate reward, while still having a high value. This happens when that state is followed by other states that obtain high rewards.

Looking at this example, since the RL agent aims to maximize the expected cumulative reward [21], it is desirable to select actions based on the value function. Unfortunately, it is much harder to compute values than it is to determine rewards: whereas rewards are given immediately by the environment at every step t, values must be estimated multiple times from the observations an agent makes over its entire lifetime. For example, in some RL problems the task that the RL agent wants to learn is defined as an episodic task, that means that the task has a terminal state. In these cases, the agent estimates values for states while learning the task over different episodes. In fact, the most important component of many RL algorithms is a method for efficiently estimating value functions.

The last concept to introduce is the *model* of the environment. The model is

something that mimics the behaviour of the environment from the agent's point of view. The model maps state-action pairs to probability distributions over states and, for example, given a state and an action, the model could provide the resultant next state and the next reward, without changing the environment state [20]. Actually, not every RL method needs a model of its environment to work.

Finally, note that most real-case environment examples are not strictly MDP. Fortunately, most of them can be approximated as MDP environments, being able to successfully apply the same concepts and RL methods we will describe in the following section.

3.1.2 RL problems and methods

There exist multiple methods to solve Reinforcement Learning problems. In general, these methods are employed to address different RL types of problems: the planning, the prediction and the control problem, that we will explain soon.

Among possible methods, we first make a distinction between model-based methods and model-free methods.

In model-based methods a model of the environment is available to the RL agent. These methods are mainly used for solving the *planning* problem, that is when an agent that knows the model considers every possible future situation while selecting an action, before any situation is actually experienced [20].

Main model-based RL algorithms are value iteration and policy iteration algorithms, each of which has origins in the field of Dynamic Programming [21]. These algorithms are based on a model of the environment, which practically is a transition table. This table contains all the knowledge necessary to the agent to achieve the goal of the specific RL problem. Indeed, these algorithms can be used to retrieve the optimal value function or the optimal policy, or both. However, they have a big limitation: in most cases, in which the environment is vast and mostly unknown, it is impossible to build a transition table to represent it. For this reason, as opposed to these methods, model-free methods have been introduced. These algorithms are the most used in practical scenarios [21].

Model-free methods are methods actually based on the trial-and-error learning process [20]: since a model of the environment is not available, the only way for the agent to collect information about the environment is through interaction. If properly used, these methods can be used for solving both the prediction and the control problems.

In the *prediction* problem, the RL algorithm is used to learn the value function for the policy the agent is following. At the end of the learning process, the value function describes, for every state visited while following the policy, the amount of future reward we can expect when performing actions starting from this state [21].

The *control* problem, instead, consists in finding a policy that maximizes the

reward when travelling through the space of the states. The policy is found while interacting with the environment. At the end we obtain an optimal policy that allows for selecting the best actions. Note that in RL problems there always exist at least one optimal policy, while also multiple optimal policies could exist.

It appears that to find the policy that maximizes the reward, also the optimal value function could be computed. This means that solving the control problem would seem to require a solution also to the prediction problem for the policies that we want to evaluate [21]. Practically, algorithms targeting the control problem allow for simultaneous value function and policy optimizations.

At the state-of-the-art, there exist several model-free methods to determine the optimal value function and/or the optimal policy. These methods can be categorized by the way they estimate the value function into the following three learning categories [20]:

1. *Monte-Carlo (MC) learning.* The RL agent performs actions moving through states, going to the end of a trajectory, and estimates the value function once it arrives at the end, looking at the rewards it received during the trajectory. Since the estimation is done only at the end of a series of states - at the end of the trajectory - MC learning is suitable only for episodic tasks, in which a terminal state exists. Differently from model-based methods, MC does not assume a complete knowledge of the environment. Indeed, MC requires only experience, which is composed by sample sequences of states, actions and rewards from the interaction with the environment. When MC is used for the prediction problem, in which the policy is fixed, the MC update of the value function $V(s)$ is the following one:

   $$V(s_t) \leftarrow V(s_t) + \alpha[R_t - V(s_t)]$$

   in which α characterizes the learning rate. At the time step t, the value function is updated with the value of the return - the expected cumulative reward R_t - obtained starting from that state and ending into a terminal state, which represents the end of the episode, while following the fixed policy. If MC is used for the control problem, it is necessary to consider the action value function rather than the state value function, in which we estimate actions values. In this scenario, if the state and action sets are finite, the value function is a matrix $Q(s, a)$ with a value for each state-action pair.

2. *Temporal-Difference (TD) learning.* In this approach, the RL agent takes one single step and estimates the reward right after it. As opposite to MC, this method does not wait until the end of an episode, and for this reason it is also suitable for continuous tasks, which do not have a terminal state. Also in TD learning the agent learns from experience. In the prediction problem, to

estimate the value function $V(s)$ at each state s, the update is performed in the following way:

$$V(s_t) \leftarrow V(s_t) + \alpha[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)]$$

in which α influences the learning rate and γ is the discount factor, that identifies how much to take into account the value of the successive state s_{t+1}. In this update, r_{t+1} is the reward obtained at state s_{t+1}. This represents the main difference from MC, in which the cumulative reward, so the return R_t, was used. Moreover, in this update an existing estimate is used, since the value function of another state s_{t+1} is used to update the value function of s_t. This bootstrapping feature is not present in the MC update[2]. As for MC, if TD is used to target the control problem, the value function $V(s)$ is replaced with a state value function $Q(s, a)$. Most common TD algorithms are SARSA and Q-learning, which are respectively an on-policy and an off-policy method. In on-policy methods the agent learns the values of the current policy, while in off-policy methods the agent actually learns the optimal policy, while it may be following a different one.

3. *TD(λ) learning.* This third and more recent approach has been developed to unify those two learning approaches previously presented. In this learning method the RL agent takes n steps and updates the value function looking at the current reward, obtained after having performed n steps instead of just one. To realize this idea, TD(λ) methods exploit eligibility traces. An eligibility trace $e(s)$ is a temporary record of the occurrence of an "event", which can be the visiting of a state or the performing of an action. The use of these traces requires an additional memory variable associated with each state for the prediction problem $e(s)$, and with each state-action pair if TD(λ) is used for the control problem $e(s, a)$. The most common TD(λ) algorithms are the extensions of traditional TD learning methods: SARSA(λ) and Q(λ). In fact, SARSA and Q-learning are equivalent to SARSA(λ) and Q(λ) when $\lambda = 0$.

When addressing the control problem, all these approaches need a strategy to select the current policy. Since these approaches are model-free methods, learning is based on the interaction with the environment. For this reason, the RL agent must assure continual exploration. The most used strategy is called ϵ-greedy policy selection. This strategy allows for continual exploration, since it well balances the trade-off between exploration and exploitation, necessary for the learning of the

[2]In literature, TD learning is called a bootstrapping method, since the value function is updated using an existing estimate.

RL agent. In the ϵ-greedy policy, the greedy policy is chosen with a probability of $1 - \epsilon$, selecting the action that has the highest value for the current state. With a probability of ϵ, instead, a random action is selected.

In the next section we will present some algorithms belonging to the last two categories, TD and TD(λ) learning, which are the algorithms we developed into our framework.

3.1.3 RL algorithms

In the following chapters we are going to show the design and implementation of our framework. Since in the framework we provide the support for multiple algorithms, in order to compare their results, here we present the algorithms we will use. To realize our RL agent, we focus on targeting the control problem. We decide to use both the TD and the TD(λ) techniques. Moreover, we also want to provide comparisons among on-policy and off-policy control algorithms. The algorithms we use are SARSA, Q-learning, SARSA(λ) and Q(λ).

We will always refer to the tabular case of these algorithms, meaning that we have a finite action space and a finite state space. Roughly speaking, the numbers of states and actions available are limited numbers. Moreover, we will use the ϵ-greedy policy selection for all the presented algorithms. Also, in all algorithms we propose some good values for parameters, based on theory of RL [20] and on literature works, as [24]. These values are suggested for general RL problems and are only for example purposes. As we will see in chapter 7, different values perform better in our scenario than the traditional ones.

The first algorithm to present is SARSA (State, Action, Reward, State, Action). As previously mentioned, SARSA is a TD learning algorithm, in which the learning process is improving the policy currently followed by the RL agent. For this reason, we present SARSA as an on-policy control method. This algorithm aims to estimate the action value function, which is identified in our case by the matrix $Q(s, a)$. The parameters for this algorithm are the following:

- $\epsilon \in [0, 1]$. As previously mentioned, it represents the trade-off between exploration and exploitation. In scenarios in which the action space is vast, ϵ should be high, near to 0.9 preferring more exploration;

- $\alpha \in [0, 1]$. It is defined as the step-size parameter, which determines to what extent the new information just acquired overrides the old one. The higher this value, the higher the probability of ending into a sub-optimal solution. Where multiple sub-optimal solutions exist, α should be selected carefully, with low values;

- $\gamma \in [0, 1]$ is the discount factor that defines the importance of future rewards.

Algorithm 1 SARSA algorithm. On-policy TD control algorithm. Good values for the constants are $\epsilon = 0.9, \alpha = 0.05, \gamma = 0.9$.

procedure SARSA(ϵ, α, γ)
    ▷ ϵ is the trade-off between exploration and exploitation
    ▷ α is the learning rate
    ▷ γ is the discount factor
    ▷ $Q(s, a)$ is the state-action value function
    ▷ r is the reward
    Initialize $Q(s, a)$ arbitrarily
    for each episode **do**
        Initialize s
        Choose a using policy derived from Q (ϵ-greedy)
        while s is not terminal **do**
            Take action a, observe r, s'
            Choose a' from s' using policy derived from Q (e.g. ϵ-greedy)
            $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma Q(s', a') - Q(s, a)]$
            $s \leftarrow s'$
            $a \leftarrow a'$
        end while
    end for
end procedure

If this value is near to 0, then the agent is short-sighted, in the sense that it is interested only in current rewards.

Algorithm 2 Q-learning algorithm. Off-policy TD control algorithm. Good values for the constants are $\epsilon = 0.9, \alpha = 0.05, \gamma = 0.9$.

procedure Q-LEARNING(ϵ, α, γ)
    ▷ ϵ is the trade-off between exploration and exploitation
    ▷ α is the learning rate
    ▷ γ is the discount factor
    ▷ $Q(s, a)$ is the state-action value function
    ▷ r is the reward
    Initialize $Q(s, a)$ arbitrarily
    for each episode **do**
        Initialize s
        while s is not terminal **do**
            Choose a using policy derived from Q (ϵ-greedy)
            Take action a, observe r, s'
            $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max'_a Q(s', a') - Q(s, a)]$
            $s \leftarrow s'$
        end while
    end for
end procedure

SARSA is presented in Algorithm 1. The full name of this algorithm (State, Action, Reward, State, Action) simply reflects that the update of the Q function depends on the current state s, the selected action a, the reward r obtained after performing the action, the new state s' and the next action a' chosen by the agent in its new state.

SARSA algorithm is updating the value $Q(s, a)$ with the values $Q(s, a)$ and $Q(s', a')$, that are the values that correspond to the current policy the agent is following. In Q-learning, instead, this value $Q(s, a)$ is updated with the best value among all the available actions from the new state s'. Q-learning is presented in Algorithm 2. As SARSA, Q-learning uses TD learning, but it is an off-policy method: the Q value function learns from actions that are outside the current policy, which are the optimal actions, while following an ϵ-greedy policy. The parameters for this algorithms are the same as the ones for SARSA: ϵ, α and γ.

Since TD(λ) learning should improve the learning process in terms of time, the previously described algorithms have been expanded into SARSA(λ) and Q(λ).

SARSA(λ) is shown in Algorithm 3. Besides ϵ, α and γ, this algorithm has also λ as parameter. λ is the trace decay parameter. The higher the value the

Algorithm 3 SARSA(λ) algorithm. On-policy TD control algorithm. Good values for the constants are $\epsilon = 0.9, \alpha = 0.05, \gamma = 0.9, \lambda = 0.9$.

procedure SARSA(λ)($\epsilon, \alpha, \gamma, \lambda$)
    ▷ ϵ is the trade-off between exploration and exploitation
    ▷ α is the learning rate
    ▷ γ is the discount factor
    ▷ λ is the trace decay parameter
    ▷ δ is the temporal difference
    ▷ $Q(s, a)$ is the state-action value function
    ▷ $e(s, a)$ is the eligibility trace
    ▷ r is the reward
    Initialize $Q(s, a)$ arbitrarily and $e(s, a) = 0$ for all s, a
    for each episode **do**
        Initialize s, a
        while s is not terminal **do**
            Take action a, observe r, s'
            Choose a' from s' using policy derived from Q (e.g. ϵ-greedy)
            $\delta \leftarrow r + \gamma Q(s', a') - Q(s, a)$
            $e(s, a) \leftarrow e(s, a) + 1$
            for all s, a **do**
                $Q(s, a) \leftarrow Q(s, a) + \alpha\delta e(s, a)$
                $e(s, a) \leftarrow \gamma\lambda e(s, a)$
            end for
            $s \leftarrow s'$
            $a \leftarrow a'$
        end while
    end for
end procedure

longer the traces: this means that a larger credit from a reward can be given to more distant states and actions. In fact, when $\lambda = 1$ the algorithm becomes a sort of MC learning, which updates values evaluating the reward at the end of an episode. SARSA(λ) uses the idea of eligibility traces, which are represented by the matrix $e(s, a)$, that counts the number of times the RL agent is in state s and takes the action a[3]. The values of the trace are multiplied by the temporal difference δ to update the Q value function. δ is the same quantity which is used by the plain SARSA algorithm to update the Q value function, opportunely sized by the learning rate α. Later, the values of the trace are discounted with the λ and the γ parameters for all states and all actions, as the algorithm shows.

Analogously, also Q(λ) is a TD(λ) algorithm and uses the same concept of eligibility traces. Two different methods to combine Q-learning and eligibility traces exist: Watkin's Q(λ) and Peng's Q(λ). In this work we are going to use the Watkin's version. The algorithm needs the same parameters as SARSA(λ), which are ϵ, α, γ and λ. In Q(λ), the temporal difference δ is computed in the same way as for the plain Q-learning update function of the Q value function, as presented in Algorithm 4. Unlike SARSA(λ), this algorithm does not look ahead all the way to the end of the episode in its update of the Q value: it looks ahead as far as the next exploratory action - so the next action chosen randomly - is performed. Practically, when a non-greedy action is selected, which is a random action and not the best one, the value of the trace is set to 0. In this way the updates of the Q value exploit eligibility traces until a random action is chosen.

3.1.4 Evaluation metrics

In this section we provide some metrics for evaluating results, hence for testing and comparing the performance of the various algorithms. Later, in chapter 7, we will refer to this description to present our results.

We define the notation we use in this work in Table 3.1. Note that, in our results, we assume that S and A are finite sets. In this way $Q(s, a)$ and $e(s, a)$ are both matrices. Moreover, we assume to consider only episodic tasks, in which a terminal state always exists. We are going to present the evaluation metrics, which will be used to define the graphs we show later in this work.

The first measure to evaluate a RL algorithm is based on the cumulative reward obtained per episode over episodes, which can be plotted into a graph:

$$FinalReward(x, y) = (E, R_E), for E \in (0, ..., N_E)$$

[3]Since they are counting the number of occurrences of an event they are called accumulating traces. Other types of eligibility traces exist [20].

Algorithm 4 Watkins's Q(λ) algorithm. Off-policy TD control algorithm. Good values for the constants are $\epsilon = 0.9, \alpha = 0.05, \gamma = 0.9, \lambda = 0.9$.

procedure Q(λ)($\epsilon, \alpha, \gamma, \lambda$)
    ▷ ϵ is the trade-off between exploration and exploitation
    ▷ α is the learning rate
    ▷ γ is the discount factor
    ▷ λ is the trace decay parameter
    ▷ δ is the temporal difference
    ▷ $Q(s, a)$ is the state-action value function
    ▷ $e(s, a)$ is the eligibility trace
    ▷ r is the reward
    Initialize $Q(s, a)$ arbitrarily and $e(s, a) = 0$ for all s, a
    for each episode **do**
        Initialize s, a
        while s is not terminal **do**
            Take action a, observe r, s'
            Choose a' from s' using policy derived from Q (e.g. ϵ-greedy)
            $a^* \leftarrow \arg\max_b Q(s', b)$ (if a' ties for the max, then $a^* \leftarrow a'$)
            $\delta \leftarrow r + \gamma Q(s', a^*) - Q(s, a)$
            $e(s, a) \leftarrow e(s, a) + 1$
            for all s, a **do**
                $Q(s, a) \leftarrow Q(s, a) + \alpha\delta e(s, a)$
                if $a' = a^*$ **then**
                    $e(s, a) \leftarrow \gamma\lambda e(s, a)$
                else
                    $e(s, a) \leftarrow 0$
                end if
            end for
            $s \leftarrow s'$
            $a \leftarrow a'$
        end while
    end for
end procedure

ϵ	exploration-exploitation trade-off
α	learning rate
γ	discount factor
λ	trace decay
t	time step
s_t	state at time t
a_t	action at time t
r_t	reward for the current performed action at time t
E	episode
R_E	cumulative reward or return obtained at the end of episode E
N_E	total number of episodes
N_{t_E}	total number of time steps t in episode E
N_a	total number of actions performed
E_{decay}	episode E at which the ϵ decay is performed
D	ϵ decay value
S	set of all states
A	set of all actions
π	policy
π^*	optimal policy
$V(s)$	state value function
$Q(s,a)$	action value function or Q value function
$e(s,a)$	eligibility trace
run	single execution of the learning process
N_{runs}	number of runs for the same learning process
avg	average
$mov-avg$	moving average
w	window size

Table 3.1: Formal notation for evaluation metrics.

Another analogous measure is based on the number of time steps t used in a single episode E to reach the terminal state:

$$TimeSteps(x,y) = (E, N_{t_E}), for E \in (0, ..., N_E)$$

The measures used in $FinalReward$ and $TimeSteps$ can be also used to show results about rewards and time steps averaged over multiple execution - which we call multiple "runs" - of the learning process, and about the moving average of these averaged values for a specified window size w. Therefore, for $E \in (0, ..., N_E)$

$$FinalReward_{avg}(x,y) = (E, Avg(R_E)_{runs}) = (E, \frac{1}{N_{runs}} * \sum_{run=1}^{N_{runs}} R_{E_{run}})$$

$$TimeSteps_{avg}(x,y) = (E, Avg(N_{t_E})_{runs}) = (E, \frac{1}{N_{runs}} * \sum_{run=1}^{N_{runs}} N_{t_{E_{run}}})$$

$$FinalReward_{mov-avg}(x,y) = (E, MovAvg(Avg(R_E))) = (E, \frac{1}{w} * \sum_{i=E-w}^{E} Avg(R_E)_i)$$

$$TimeSteps_{mov-avg}(x,y) = (E, MovAvg(Avg(N_{t_E}))) = (E, \frac{1}{w} * \sum_{i=E-w}^{E} Avg(N_{t_E})_i)$$

Average and moving average measures help to reduce the amount of noise and oscillations present in every execution of RL algorithms: their exploratory nature can badly affect the final reward and the number of time steps per episode, even if the learning process is working well.

$FinalReward_{mov-avg}$ and $TimeSteps_{mov-avg}$ can also be used to compare into the same graph multiple algorithms, or multiple executions of the same algorithm with different values for configurable parameters. With this approach, it is possible to perform a sort of parameter tuning to find the best efficient values for our RL problem.

Another interesting measure is to compute the cumulative reward from the beginning of the learning process over the number of actions performed, which corresponds to the number of time steps t. Given the cumulative reward: $Cum_{R_{n_a}} = \sum_{t=0}^{n_a} r_t$, the graph can be:

$$RewardOnActions(x,y) = (n_a, Cum_{R_{n_a}}), for\ n_a \in (0, ..., N_a)$$

RewardOnActions can be used to plot the cumulative reward over the actions for multiple executions of the same algorithm. To compare different algorithms an average among different runs is needed, as we made for $FinalReward$ and $TimeSteps$. The difference is that the number of episodes N_E is fixed, but the number of total time steps for the entire learning process can vary a lot. Hence, to compute some average values for *RewardOnActions*, we decide to make an average on the number of commands from 0 to the number of commands obtained in the execution with the minimum amount of commands sent. Thus, for $n_a \in (0, ..., N_{a_{MIN}})$, defining $N_{a_{MIN}} = \min_{run} N_{a_{run}}$:

$$RewardOnActions_{avg}(x,y) = (n_a, Avg(Cum_{R_{n_a}})_{runs}) =$$

$$= (n_a, \frac{1}{N_{runs}} * \sum_{run=1}^{N_{runs}} Cum_{R_{n_a}})$$

If the trend of the cumulative reward increases after n_a commands in the curve described by $RewardOnActions_{avg}$, we can say that the learning process for that specific algorithm needs at least n_a commands on average to learn something useful.

Considering another metric, given the average reward obtained at each time step in a single episode: $Avg_{R_E} = E[R_E]_t = \frac{1}{N_{t_E}} * \sum_{t=0}^{N_{t_E}} R_{E_t}$, we can compute the Cumulative Distribution Function (CDF) for the average reward for episodes:

$$CDFReward(x, y) = (Avg_{R_E}, CDF(Avg_{R_E})), forE \in (0, ..., N_E)$$

For $CDFReward$ the CDF for different executions can be computed, adding to the CDF all Avg_{R_E} values for all N_{runs} executions. In this way, a fair comparison between different algorithms can be made.

Finally, some more interesting values can be used to compare different algorithms or different configurations of parameters for the same algorithm:

- $LearningTime$: it is the time in seconds measured starting from the beginning of the first episode to the end of the last episode N_E;
- $LearningTraffic$: it is the number of commands sent by the RL agent to the targeted device during the entire learning process, from the first episode to N_E;
- *Average reward* AVG_{rew}: it is computed as the average over all episodes of the final reward R_E obtained for each episode E. This value can also be averaged over multiple runs, in order to allow comparisons between different algorithms and configurations;
- *Average time steps* AVG_{steps}: it is computed as the average over all episodes of the number of time steps N_{t_E} of each episode E. As before, this value can also be averaged over multiple runs to allow comparisons between different algorithms and configurations.

Chapter 4

RL in practice: Learning the TCP state-machine

In this work we design a Reinforcement Learning framework focusing our attention on IoT devices. When applying RL techniques, the main challenge lays in modelling the environment, which is strictly dependant on the task that the RL agent should perform. For example, when RL is applied to games, as the Atari game Space Invaders, notions of reward, states and actions are quite intuitive. Often the reward is equivalent to the score of the game, while the actions are the possible commands available to a user who plays the game - e.g. go left, go right, shoot. The states instead could be represented by the frames of the screen while playing, which contain information about the position of the player and of the other components of the game. When it comes to different scenarios, instead, these variables become less intuitive. For example, when applying RL to protocols, we do not have explicit values that can be directly translated into rewards, and also the definition of the state space is not trivial.

For this reason, we present a simplified integration of RL algorithms into the Transmission Control Protocol (TCP). Our goal is to train a RL agent to exchange TCP messages with a server in order to correctly establish a TCP connection with the server. We show the approach we used to integrate RL algorithms, like SARSA, SARSA(λ) and Q-learning, with an environment which simulates a TCP client and a TCP server. We first present the construction of the environment, describing how states, actions and rewards can be modelled. Finally, we present some results and show the performance of the RL agent during the learning process performed with different algorithms.

The software implementing this TCP case scenario we are going to present is openly available and it can be found at `https://github.com/giuliapuntoit/RL-tcp-toycase`.

4.1 Construction of the TCP environment

As stated in section 3.1, RL problems are goal-oriented. After formalizing the goal, it must be expressed in terms of reward, so that the RL agent can learn how to achieve it, maximizing the total reward. When working with network and communication protocols, the goal of the RL problem is not immediate and should be clearly defined, before starting to design the environment. One approach to define a goal for a protocol is to look at its state machine, if available. For the TCP protocol, a state machine exists and it is well-defined. The TCP state machine, presented in Figure 4.1, represents all possible states involved into the establishment and closing of a TCP connection. In this schema, there exist no distinction between a client and a server, which are actually the endpoints of the TCP connection we want to establish. The values of the arrows represent the event perceived by the server - e.g., "rcv ACK" - combined with the reaction of the client, which can send messages - e.g., "send ACK" - or perform other operations, like "x". The "x" in the arrows represents the do-nothing action, which could also be used to reflect lost packets or out-of-time sent packets.

Note that the starting and the ending states have the same name: the scheme is cyclic, hence they are the same state. Since the RL agent needs to know whether it is at the starting state or the terminal one, we split the "Closed" state into two distinct states.

For our purposes, we define our goal in the following way: the RL agent should learn how to establish a TCP connection and correctly close that connection in the shortest possible time, which means with as few steps as possible.

In the preliminary phase of this part, we model the environment - along with the state and action spaces and the reward signal - and the RL agent in the following way:

- The *environment* reflects the standard TCP state machine. *States* are 12 and correspond to the boxes in the diagram, while the *actions* are 11 and are the arrows in the figure. If an action performed at state s is different from the one expected by the state machine to reach the following state, we assume that the state $s_{t+1} = s_t$[1]. Actions and transitions between states happen at discrete time steps t. Note that no distinction among client and server exists and that the environment is emulating the states of the connection: neither a real server or real TCP connections exist in this first phase. We are working at an higher abstraction level, mainly focusing on the RL methodology. Concerning the *reward*, the environment gives to the agent a small negative reward of -1 at

[1]This assumption was made for educational purposes. This decision is not relevant with respect to the purpose of understanding how RL can be used for protocols.

each additional time step. This models the fact that we want to perform the desired operations - establishing and closing the connection - in the shortest possible time. Moreover when the agent reaches the terminal state, if the connection is established the agent obtains a small positive reward, as +10, and if the correction is correctly closed the agent obtains a high positive reward, as +200;

- The *agent* has no notion of the existence of a distinct TCP client and server inside the TCP state machine, and it simply treats actions as black boxes in order to try to transit from state to state and reach the terminal "Closed" state, while maximizing the reward. This means that it can take all the available actions, that are the ones in the figure. After picking one action, the environment reacts, changing the state of the TCP connection and evaluating the reward for the agent at each step. The agent selects and performs one action at each discrete time step t.

The highest achievable reward is given by the states "Closed", "SYN sent", "Connection Established", "FIN wait 1", "FIN wait 2" and eventually "Closed". Even if other possible paths exists, which both establish and close a connection correctly and obtaining a positive +200 reward at the terminal state, they have longer paths than the optimal one, which is composed by only 6 steps[2].

With these premises, we implement some RL algorithms, as we show in section 4.2, and the solution works, since our RL agent is actually learning how to reach the terminal state, maximizing its reward. Nevertheless, due to the simplifying choices we made, the scenario is extremely limited. First of all, as said previously, there is no notion of a client and an independent server, neither real or emulated. States represent actual states of a TCP connection but the actions are not actual messages that a TCP client would exchange with a TCP server. Moreover, our agent is forced at each step to try multiple actions before finding one useful for transiting into a new state. This happens because there exists a lot of unexpected actions for each state that does not cause any change of the state into the environment.

Because of these limitations, we decide to increase the complexity of our scenario. The goal of our RL problem remains the same: make a RL agent able to open and correctly close a TCP connection. The difference is that we want to achieve this into a more realistic environment, in which the Reinforcement Learning agent takes the place of the client side of the TCP protocol.

Before starting to develop the RL agent, we need to model a TCP server emulating the behaviour of a real one. In order to add complexity to our scenario,

[2]The count of steps for each path can be easily made counting the arrows in Figure 4.1.

we avoid to design a purely-deterministic server, in which each message sent by the TCP server is known a priori based on the current state of the TCP connection and on the message sent by the TCP client. Therefore, we model a probabilistic server, in which at each step the server could perform different actions with a certain probability distribution. The server can send to the client 5 different messages, which correspond to messages of the TCP standard: "SYN", "ACK", "SYN/ACK", "FIN","x". The latter message actually is used to simulate the situation in which the server does not send any message to the client, thus it does nothing. Practically, the non-action "x" could be also used to reflect lost packets or out-of-time sent packets.

In a real environment, the server would not send all of these messages at each state of the TCP connection. For example, if the we are in the starting state, "Closed", the server could perform action "x" and do nothing, while waiting for an active open of the connection by a TCP client. Otherwise, the server could send a "SYN" message to passively open a new TCP connection. This is a real possibility, but the passive open of a TCP connection is less frequent than the active one. For this reason, from the starting state, we decide to give to the server a higher probability of choosing the "x" action, hence a lower probability of choosing to send the "SYN" message. Following this approach for all states, we finally define all probabilities for all states and actions available to the TCP server. The new defined state machine is presented in Figure 4.2. In this figure, red arrows represent possible server actions, while green arrows represent the actions that the RL agent should learn to perform from each specific state. Due to the fact that we split client actions from the server ones, the state machine contains some additional states with respect to the previous one. Those states model situations in which the client sent a message and it is waiting for a server message back, or vice-versa. For example, in state 5, the server sent a "SYN/ACK" message to the client and it is waiting for an "ACK" message back in order to establish a TCP connection. Note that the names of states are assigned based on the point of view of the client. For example, the state "SYN sent with received SYN/ACK" refers to the client: the client has sent a "SYN" message and has already received a "SYN/ACK" message from the server. The red arrows are presented with a probability value, which defines the probability for the server to take that specific action in the state from which the arrow is coming out.

Therefore, the environment and the RL agent are modelled in the following way:

- The *environment* now emulates the extended state machine in Figure 4.2. The *states* are the boxes in the diagram. Both the client and the server can send the traditional TCP messages, but the behaviour of the probabilistic server is embedded into the environment, while the client is a RL agent. The *actions* available to the client are "SYN", "ACK", "SYN/ACK", "FIN", "x". The actions that the RL agent should learn are the ones specified in green inside

the figure. After the agent performs an action a_t, the new state is evaluated, then the server sends a message, based on the probability values we assigned, and then the new state is evaluated and identified as s_{t+1}. It could happen that the server does the "x" action and the state remains equal to the new state after performing the action a_t. From the client point of view there is no knowledge whether the server changed the state of the TCP connection or not: the client only sees s_t and s_{t+1}. Moreover, as before, an action performed by the agent at state s, which does not correspond to an outgoing arrow from that state, does not change the state of the connection, hence in this case $s_{t+1} \neq s_t$ only if the server action triggers a change of the state. The *reward* is modelled as before, since the goal of the RL problem is the same: the agent receives a small positive reward for correctly establishing the connection, of $+10$, and it obtains a high positive reward, e.g. $+200$, if the connection is correctly closed. For each additional time step a small negative reward -1 is given to the agent. Finally, if the chosen action is the non-action "x", this means that the client is not sending any message, thus not generating any traffic. For this reason, we decide to give an additional small negative reward of -1 to each selected action different than the non-action one. Hence, it could be convenient in reward terms for the agent to perform the non-action while waiting for a server action, as it could happen in state 10;

- Differently from the previous case, now the *agent* actually simulates a TCP client, while communicating with a TCP server. The server is part of the environment. The available actions correspond to the actual TCP messages that a real client could exchange. Since the state is dependent on both the client and the server actions, the agent should always maximize the reward, dealing also with cases in which the server can change the state of the TCP connection, and follow a different path inside the extended state machine. Since we embed the server behaviour inside the environment, the agent selects and perform one action at each discrete time step t.

In this scenario, after the learning process, the policy of the agent is no more fixed, since the policy followed by the agent is highly dependent on the messages sent by the server, which is not deterministic.

After finalizing the construction of the environment, it is possible to implement the RL agent exploiting different RL algorithms.

4.2 Model of the TCP client with RL algorithms

Once we define and implement our environment, RL algorithms should be implemented. For this toy case we decide to implement some of the algorithms presented

in subsection 3.1.3, in particular SARSA, Q-learning and SARSA(λ). Following the pseudo-code of the algorithms previously presented, we decide to implement them without relying on external libraries. In this way we have full control on the model of the environment, on states, actions and on the algorithm procedure itself. For explaining our choices, we want to focus on the second model of the environment, which distinguishes between a probabilistic TCP server and a TCP client, which is simulated by the RL agent. That is because it emulates a more realistic scenario than the first described environment.

The structure of the program is analogous for all algorithms. It first has an initialization phase, then it performs an outer loop over the number of episodes, and an inner loop over the time steps until the terminal state is achieved. For efficiency purposes, a maximum number of time steps is set so that if the terminal state is not reached the episode still finishes. Inside the inner loop the core of each algorithm is implemented. At the end of the outer loop, the learning process is finished and the best policy found can be evaluated. We go now into the details of our implementation.

The initialization procedure is analogous to all algorithms: a Q matrix has been initialized to 0, with rows equal to the number of states and columns equal to the number of available actions. This means that for the TCP scenario the Q matrix is 16×5. For the SARSA(λ) algorithm, the E matrix, with same dimensions as Q, has been initialized to 0.

Then, for each algorithm we select actions following the ϵ-greedy policy. All algorithms use the same function which first generates a random real number $n \in [0, 1]$. If $n < \epsilon$ a random action is chosen, otherwise the action with the highest value inside the Q matrix for the current state s is selected. Recall that ϵ represents the trade-off between exploration and exploitation. A value of ϵ near to 1 selects an action randomly in most of the time steps.

Inside the core of the program, all algorithms deals with the updates of the Q matrix - and of the E matrix for the SARSA(λ) algorithm - in different ways. For this reason, each algorithm has a specific update function, which updates both the Q matrix and, if necessary, the E matrix. Moreover, at each time step t a reward is given to the algorithm together with the information of the current state s_t in which the RL agent is.

Finally, after the learning process, the computed Q value function is evaluated in the same way for all algorithms: starting from the initial state, the best policy suggested by the Q matrix is followed and the final reward is computed.

The reason why we implement multiple RL algorithms is that we want to understand the meaning of each parameter and highlight the main differences of structures and performance of these algorithms. In the next section, we give a brief overview on the correct behaviour of our solution and start to analyze the differences between the implemented RL algorithms. This comparison process will

be studied more in deep in the following chapters.

4.3 Overview on results

In this section we present the results obtained with the RL algorithms previously described, using the TCP environment modelled as a TCP server and the RL agent which emulates a TCP client. Following the metrics defined in subsection 3.1.4, we show their performance in terms of total reward obtained for episodes and number of time steps over episodes. Also the duration of the learning process will be evaluated for different algorithms: since the model is completely simulated, it is possible to compare time differences between RL algorithms, without delays caused by network latency.

Note that results for this toy case are just preliminary, because deeper tests for RL algorithms will be made with the actual framework, and presented in chapter 7. In fact, for all algorithms we will execute 10 runs, and compute the averages among the results. Moreover, the learning process will last 200 episodes for all executions. All results here presented have been generated developing the entire TCP use case using Python 3.7.

In Figure 4.3, two graphs are illustrated. The first graph shows the final reward per episode R_E over episodes obtained by the RL agent with the Q-learning algorithm. In the graph on the right, the number of time steps N_{t_E} over episodes is shown. These graphs show curves for the results obtained in 1 single run, the results averaged over 10 runs and the moving average computed over 10 runs, with a window $w = 20$. We execute the learning process for 200 episodes. From the figure, we can notice that after about 50 episodes the reward overcomes a value of 100 and the number of time steps goes below 10.

We can see that from the dotted curve, the one for one single run, these graphs oscillates a lot over episodes. These oscillations are due to two different reason. First, it is because of the constant presence in RL algorithms of the exploration of new actions for the ϵ-greedy policy selection, which in this case is of 60%, since $\epsilon = 0.6$ Second, our TCP client is trying to learn the TCP state machine communicating with a TCP server. This server is probabilistic, hence it could happen that the server performs some actions that make the client deviate from the optimal path inside the state-machine.

We can repeat the evaluation of the performance also for the SARSA and SARSA(λ) algorithms. For all algorithms we use the following values for parameters: $\epsilon = 0.6$, $\alpha = 0.05$, $\gamma = 0.9$, and $\lambda = 0.9$ only for SARSA(λ). We chose this values basing on the proposed values in [20] and [24].

For the ϵ value instead, we manually performed different trials and finally we found a good exploration-exploitation trade-off with $\epsilon = 0.6$. We decreased ϵ with

respect to the value suggested in [20] since our RL agent has a very small set of available actions, only 5. Therefore, a value too close to 1 would not be helpful in learning useful actions in our situation in which the TCP server is probabilistic and could change frequently the current state.

In Figure 4.4, the previous graphs have been extended to show also the values for the other RL algorithms, focusing solely on the moving average curves. From these graphs we can notice that the results for the rewards, in Figure 4.4a, while being different until episode 50, then start to oscillate in a similar way for all algorithms. SARSA(λ) is the curve that has the highest moved average reward for the initial episodes, and it has also globally the best average reward value, as shown in Figure 4.4c. Concerning the number of time steps, SARSA(λ) is the algorithm that for the initial episodes takes the less number of steps to reach the terminal state. From these graphs, we could state that, for our toy case, SARSA(λ) is the algorithm which is learning faster.

Finally, from these results we can also evaluate the learning time and the learning traffic for all three algorithms. In Figure 4.5, we evaluate the average times for performing the learning process and the number of commands sent by the RL agent, that is the TCP client, to the server to move inside the environment. We compare those values for all algorithms.

According to our results, SARSA and Q-learning algorithms have similar performance, taking less than 2 seconds to execute the learning process for 200 episodes. SARSA(λ) has a greater average value and a greater variability among different runs, with respect to the other two algorithms. Concerning the learning traffic results in Figure 4.5b, on average SARSA(λ) generates the minimum amount of commands, with an average value around 2100 commands. However, whereas SARSA and Q-learning generate on average more commands inside our simulated network, they have a smaller variability among different executions, with respect to SARSA(λ).

As mentioned before, we just gave an overview on the results obtained with the TCP toy case, making some considerations between different RL techniques and laying the foundations for the in-depth analyses we will perform later in this thesis.

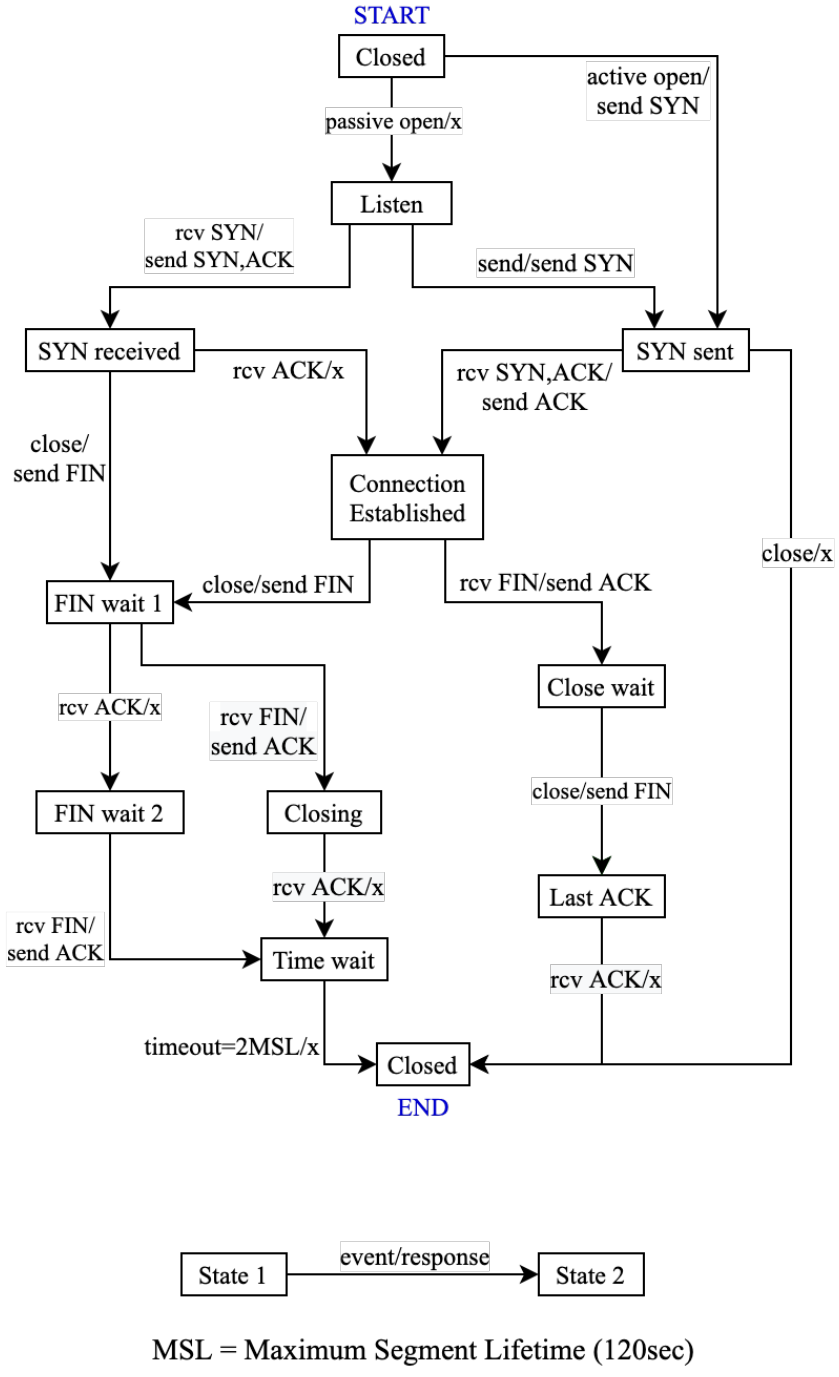


Figure 4.1: TCP state machine. Each box represents the state of the connection, each arrow represents the event and response sequence to move from one state to the subsequent one [25].

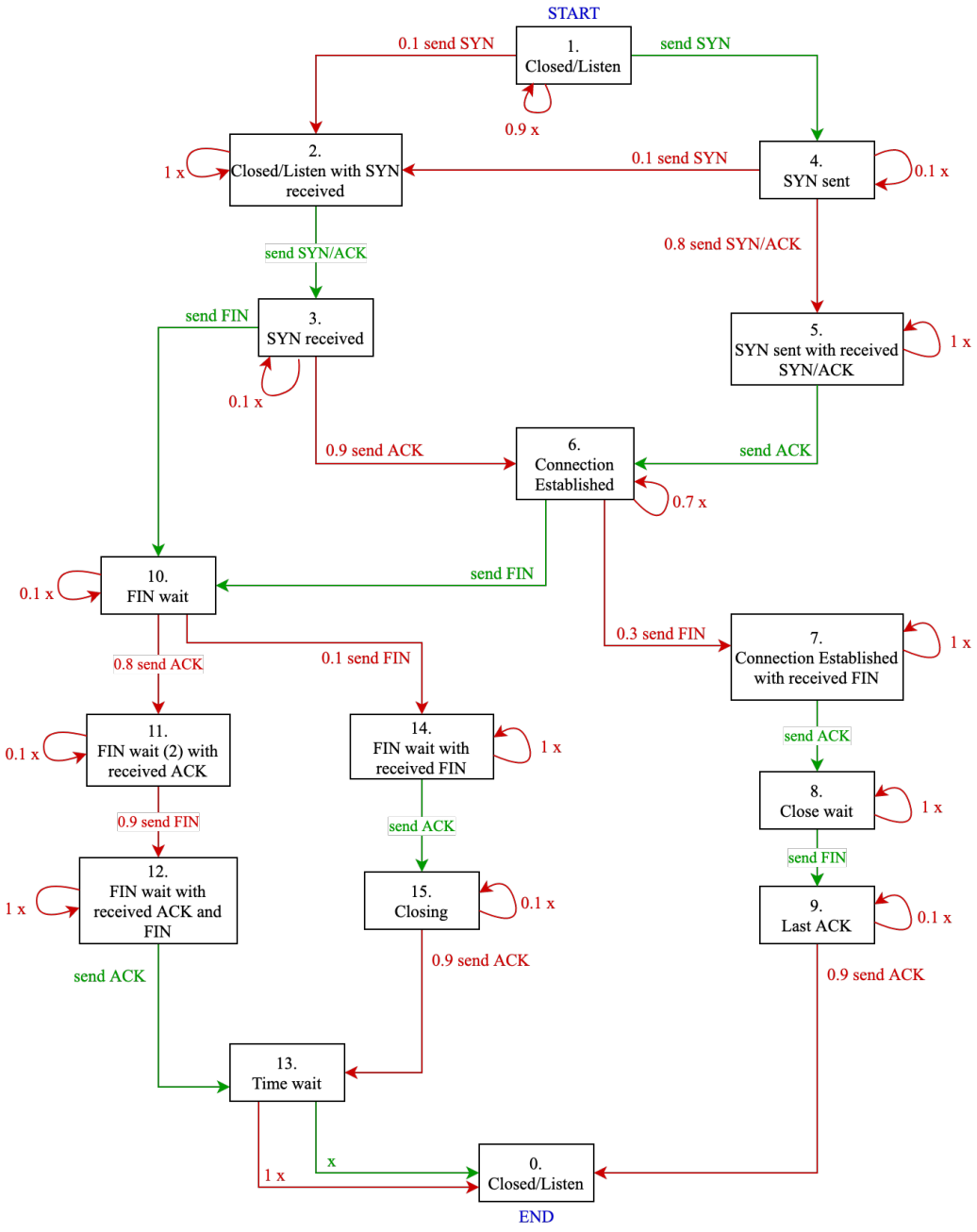
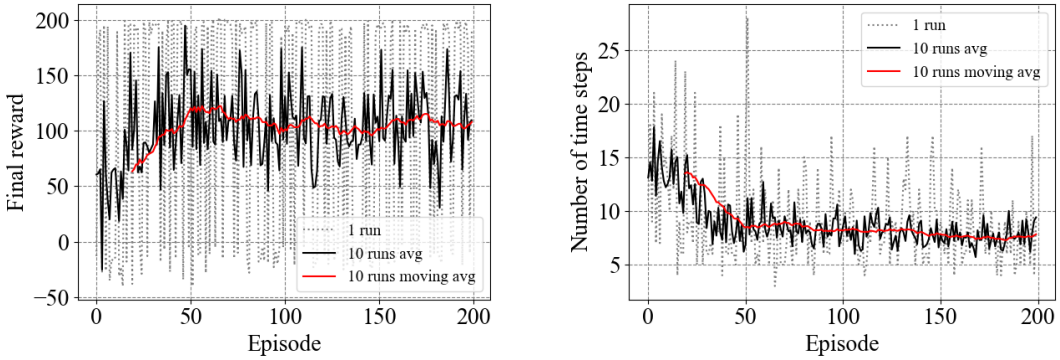


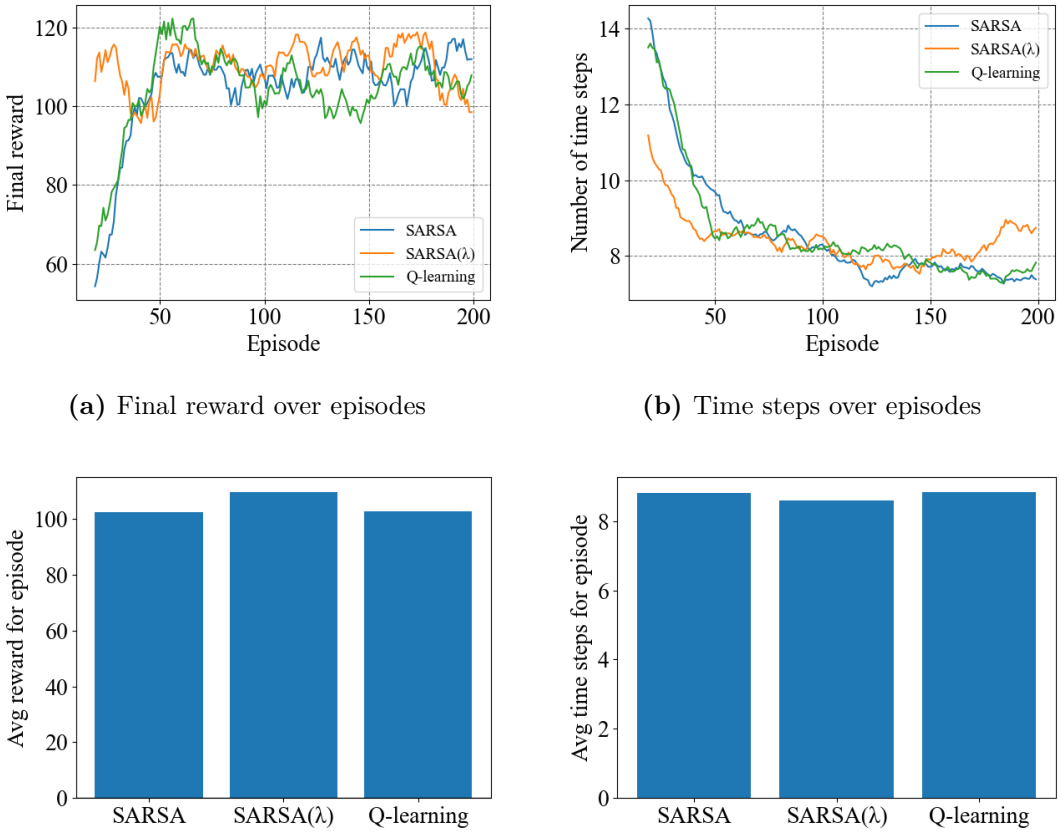
Figure 4.2: TCP state machine with the distinction between a TCP client and a TCP server. Green arrows represent actions of the RL client; red arrows represent actions performed by the probabilistic server.



(a) Final reward over episodes

(b) Time steps over episodes

Figure 4.3: Q-learning performance: TCP state-machine with $\epsilon = 0.6$, $\alpha = 0.05$, $\gamma = 0.9$. **(a)**: $FinalReward$, $FinalReward_{avg}$, $FinalReward_{mov-avg}$; **(b)**: $TimeSteps$, $TimeSteps_{avg}$, $TimeSteps_{mov-avg}$.



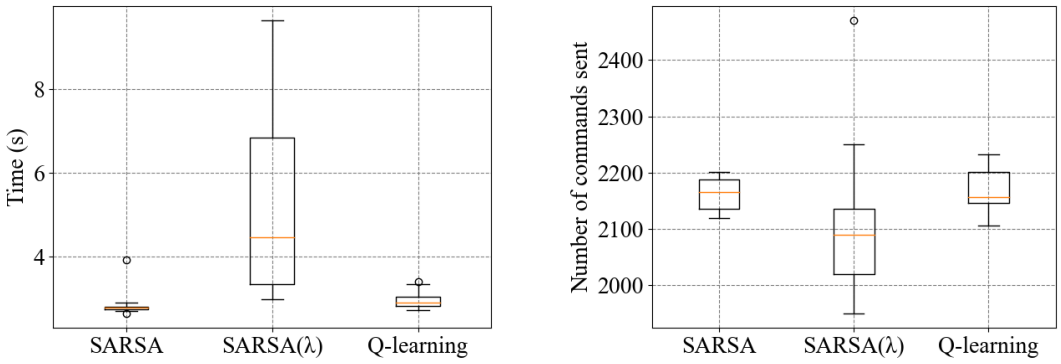
(a) Final reward over episodes

(b) Time steps over episodes

(c) Average reward

(d) Average time steps

Figure 4.4: All algorithms: performance with TCP state-machine. SARSA: $\epsilon = 0.6$, $\alpha = 0.05$, $\gamma = 0.9$; Q-learning: $\epsilon = 0.6$, $\alpha = 0.05$, $\gamma = 0.9$; SARSA(λ): $\epsilon = 0.6$, $\alpha = 0.05$, $\gamma = 0.9$, $\lambda = 0.9$. **(a)**: $FinalReward_{mov-avg}$; **(b)**: $TimeSteps_{mov-avg}$; **(c)**: AVG_{rew}; **(d)**: AVG_{steps}.



(a) Learning time

(b) Learning traffic

Figure 4.5: All algorithms: performance with TCP state-machine. SARSA: $\epsilon = 0.6$, $\alpha = 0.05$, $\gamma = 0.9$; Q-learning: $\epsilon = 0.6$, $\alpha = 0.05$, $\gamma = 0.9$; SARSA(λ): $\epsilon = 0.6$, $\alpha = 0.05$, $\gamma = 0.9$, $\lambda = 0.9$. **(a)**: $LearningTime$; **(b)**: $LearningTraffic$;

Chapter 5

Design: RL framework for IoT protocols

As we introduced in chapter 2, our goal is to exploit RL techniques to be able to communicate with devices speaking different protocols. Among all protocols, we focus on IoT protocols. We want to use RL to learn and explore the state-machine of the targeted IoT protocols and devices. Hence, for this purpose, we design a RL framework. The main functionality of the framework is to model a RL agent attacking IoT devices, the "RL attacker". We can go now into the details of our framework, presenting our design choices and the features we want to provide.

5.1 RL attacker

Following the premises introduced in chapter 2, the goal of our framework is to model a RL attacker that wants to target multiple IoT devices. The attacker should communicate in real-time with IoT devices, sending commands inside the LAN and collecting feedback directly from the devices. Hence, focusing on each IoT protocol, the main goal of the attacker is to learn and explore the state-machine of the targeted IoT protocol and device. Recall that we define the state-machine of a protocol as series of states, linked by a sequence of ordered commands that can be exchanged through that protocol in a correct way. This sequence of commands identifies a path inside the state-machine, composed by all states crossed when performing the commands. The commands for a protocol should be messages with a correct format and syntax, and that are valid for the specific IoT device that we are targeting, in order to correctly follow paths inside the state-machine.

In order to satisfy our needs, the framework should be designed with the following main functionalities:

1. Following the definition of our goal we explained so far, the framework should have access to the local network and should be able to exchange packets with IoT devices. Hence, we focus on IoT devices that communicate inside a LAN through Wi-Fi. Since there is no initial knowledge whether in the network there exist some devices at certain addresses and ports, a discover process should find devices and selects the ones we are interested in;

2. To learn how to automatically interact with different protocols, our approach is to use Reinforcement Learning. For using RL techniques, the framework should contain a way to model the environment: definition of states, actions and rewards. Moreover, in order to evaluate the reward, it is necessary for the modelled environment to have a feedback loop. The environment should be able to receive a feedback from the environment, that is from the IoT devices, in order to judge an action as positive or negative and be able to give the proper reward to the RL agent;

3. Giving the environment with state and actions spaces, we want to define the goal for the RL agent in a clear way. This is done defining a state-machine for the IoT devices. The state-machine is variable for different protocols and it can also be modelled in different way for the same protocol. For this reason, the framework should allow the definition of multiple state-machines. This provides flexibility to the framework, that could train the RL agent to learn how to achieve different goals;

4. By design, we want a framework that could be extendable. This means that if the support for a specific IoT protocol is implemented, other protocols could be added into the framework, providing the necessary information to interface to and use this new protocol, without having to disrupt the logic of the code;

5. Moreover, regarding usability issues, because of the fact that we use RL algorithms, an user should have the possibility to change the values of the parameters of the algorithms in order to try to improve the performance of the agent during the learning process;

6. We want to design the framework as modular. The RL attacker should be composed by almost independent modules, which provide different functionalities and different information. For example, there could be a module which defines the state-machine that the RL agent should learn. This design choice would help satisfying the previous requirements, making easier to extend its target protocols and improving the flexibility and the usability of the framework for new users.

Trying to satisfy all these requirements, we design our framework modelling the RL attacker as presented in Figure 5.1. The figure shows from a high level point

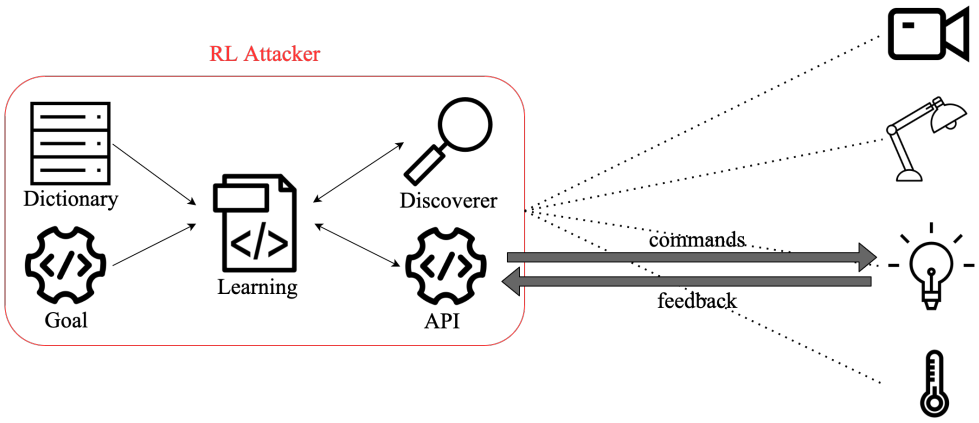


Figure 5.1: High level view of the structure of the RL attacker, with direct communication with IoT devices.

of view how the RL attacker is built and what are the main components of the structure of our framework: a Discoverer, a Dictionary, a Learning module, a Goal component and some API scripts.

The general idea is that the RL attacker first needs to find IoT devices in the network, discovering their IPs and possibly their communication ports, through the Discoverer. Then, the Learning module is the core of our framework, which is the module that implements the RL agent, along with the learning process and a testing process for evaluating that the learning process is correctly completed. The Learning module relies on the API scripts in order to directly exchange packets with the IoT devices. Moreover, the state-machine and so the goal of our RL problem is defined inside the Goal component, which is charge also of assigning a reward to the agent. From now on, we will refer to this component as the State Machine module. Since in a RL problem the agent can choose among a set of possible actions, we need to provide these actions to our Learning module. For this reason, we decide to create a Dictionary that contains a collection of messages for the IoT protocols we want to target.

Given this structure, if a protocol needs to be added to the framework, hypothetically a user should perform the following three steps: he should add messages of that protocol inside the Dictionary, provide the APIs to interact with devices relying on that protocol, and finally define the state-machine and the goal for that protocol inside the State Machine module.

Now we can describe all components just introduced in a more precise way. We will first analyze them individually and finally we will present how they interact together, modelling the behaviour of the RL attacker.

We will use Python 3.7 for our project, hence from now on we may refer to Python related modules and structures.

5.2 Discoverer

The first component that comes into play when the framework starts is the Discoverer module. As mentioned before, we focus on IoT devices connected through Wi-Fi to our local network. Thus, the Discoverer is in charge of performing a scan of the network, in order to find IoT devices connected to it. The Discoverer should retrieve information on how many devices are connected to the LAN, which are their IPs and ports for communicating and, if possible, it could be helpful to retrieve information about the protocol that each device supports. Indeed, if the IoT protocol is supported by the framework, then that device can be attacked by our RL attacker. For now, it is not possible to start a communication with a device with a protocol not supported, since the dictionary would not contain protocol-specific messages.

This module is structured into two sub-components: a Network Analyzer and a Discovery Report. The Network Analyzer scans the LAN using nmap[1]. Nmap is a free tool for network discovery and security auditing purposes. It uses IP packets to find hosts in the network, collecting data about services offered by the hosts, their operating systems and other characteristics. We use nmap through its implementation into a python module[2]. Using this tool, the Network Analyzer scans all IPs inside 192.168.1.1/24 and then analyzes the scan results. In this way we are performing a complete scan of the local network. Actually, other tools[3] or smarter techniques [26] could be used, in order to avoid to scan the entire network, mostly in situations in which the network to scan is potentially much larger. This is not a problem in our scenario since we can have at most 255 devices inside our LAN.

Among the hosts found inside the network, not all of them are IoT devices. We perform a filtering process to the scan results, selecting hosts with a name indicating the presence of an IoT device. We found out that IoT devices have some keywords in the host name that can help to recognize them to be devices of a specific IoT protocol. Hence, knowing already those keywords, we can filter scan results and select only those devices that match with one of the keywords.

[1] nmap: `https://nmap.org`

[2] python-nmap: `https://pypi.org/project/python-nmap`

[3] E.g., Fing: `https://www.fing.com` or Advanced IP Scanner: `https://www.advanced-ip-scanner.com/it`

Therefore, whenever the Discoverer finds an IoT device for the protocols we are interested in, it generates a Discovery Report.

A Discovery Report is an object that can be passed to the Learning module, so that the RL attacker can communicate with the device specified inside the report. Indeed, it has all the potentially useful information retrieved by the nmap analysis. Hence, the Discovery Report is an object that contains the IP address of the device, the target port, the name of the IoT protocol of the device, the timestamp and the complete result of the scan. The timestamp is used to date the network scan, since we do not want to rely on old scan results. The complete result of the scan is passed as an additional information which could be used if necessary for checking the correctness of the scan or for debug purposes: it could contain information about the host name, the service that the device is offering, open ports, etc. Both the timestamp and the result of the scan are not strictly necessary for the learning process unless error checks need to be performed.

From the RL attacker point of view, the fundamental information about the Discovery Report is represented by the IP, the port and the IoT protocol of the device found. Without these information, the RL attacker is not able to communicate with these devices.

Report	**Yeelight device**	**Shelly device**
Result	{‘addresses’:{‘ipv4’: ‘192.168... ‘hostnames’:[{‘name’:‘yeelink... ‘status’:{’reason’: ’user-set’, ... ‘vendor’:{}}	{‘addresses’:{‘ipv4’: ‘192.168... ‘hostnames’:[{‘name’: ‘shellybulb... ‘status’:{‘reason’: ‘user-set’, ... ‘vendor’:{}}
Protocol	yeelight	shelly
Time	1606667273.869332	1606667273.869638
IP	192.168.1.183	192.168.1.212
Port	1982	80

Table 5.1: Discovery Report samples: one for a Yeelight bulb, one for a Shelly bulb.

We provide an example of how a Discovery Report looks like. Suppose we have two devices connected to our LAN, one is a Shelly[4] device while the other is a Yeelight[5] device[6]. Assuming our framework to support both those protocols, we use the Network Analyzer to scan the network. For supporting the discovery of these

[4] `https://shelly.cloud`

[5] `https://www.yeelight.com`

[6] These devices are only for presentation purposes. We will further describe the Yeelight protocol, which will be used later in this thesis, in the following chapter.

two protocols, we add keywords to identify the devices, like “shelly”, “yeelight” and “yeelink”. We focus on these keywords, which are the ones we retrieved performing nmap tests on a LAN containing some IoT devices, which belong to these two protocols. The Network Analyzer will produce two reports. We present these two Discovery Reports in Table 5.1. At this point, these reports are returned to the caller of the Discoverer module.

5.3 API for IoT devices

When designing our framework, we would like to stay as general as possible, regarding to the specific IoT devices or protocols we are targeting. However, we must consider that, because of the heterogeneity of IoT protocols, implementing the support for a new IoT protocol is time consuming. In fact, it is unlikely that different IoT protocols could use the same infrastructure to allow the communication with their IoT devices. Therefore, we design the framework to be extendable, in a way that new protocols can be added without disrupting the correct working of the framework.

This functionality is realized using different API scripts for different IoT protocols. From the outside, the caller of the API module, which typically is the Learning module, will use methods from a common Client interface, which will be in charge of calling the API script specific for the targeted IoT protocol. The Client interface, and so all API scripts, should provide methods for exchanging messages with the devices:

- The most important method is a method that receives a command already formatted following the rules of the IoT protocol and creates a packet with that command and sends it to the device. For example, in cases where the protocol relies on JSON commands, this method should receive a JSON string as input. This string will be encapsulated into a packet and sent to the device. This method should also handle the response to the command in two different ways, providing to the framework a feedback loop. Hence, two versions of this method exist, that are different by the way they handle the response:
  - The first version checks only the correctness of the command sent. For example, if we need to set the state of the device to the initial state of the state-machine we defined, we need to check if the command actually took the device to the initial state and do not perform any additional operation;
  - The second version should handle the response and detecting if - for that specific protocol - the response reports an error about the previous request sent or if the response highlights a correct request. This version is aware

of what is a "correct" and a "error" response, so it directly assigns a negative reward whenever the response contains an error message. We will talk later about the reward signal, but the information about whether the command was correct is highly dependent on the specific IoT protocol, and for this reason a part of the reward is given by this method.

- A method to retrieve the state of the device is needed. Also this method is protocol specific and sends a command to the device in order to retrieve the current property values for the device. For example, in case the device is a light bulb, the method should retrieve information about power, rgb and brightness values and compute the current state of the device, exploiting methods provided by the State Machine module;
- Moreover, other methods can be added if necessary for the specific IoT protocol.

To make the API scripts work, we make two important assumptions. First, to make our RL attacker work we need a feedback loop. For this reason, the API script is told how to distinguish between a correct response or a response containing error messages for the specific IoT protocol. Second, for knowing which are the property values of a device, the API module is aware of what is the specific command to send to the device in order to retrieve those values. We are aware that these are two strong assumptions, but we leave their removal for future work.

In our framework, all API scripts and the Client interface are grouped together inside a higher level module, the Device Communication module.

5.4 Dictionary

The goal of the RL attacker of our framework is to learn sequences of commands for a certain IoT protocol that maximize the final reward. All the initial knowledge that the attacker has is stored inside the Dictionary module. The RL attacker knows commands for each IoT protocol, knowing their formats but not knowing what are the purposes and effects of that commands on the devices. We realize this module into two sub-modules: a Request Builder module and the actual Dictionary module.

The Dictionary module stores for each IoT protocol all available commands for that protocol. They can be stored in two ways: as plain texts or as templates with possible values for each field, depending on how the protocol commands are defined. Since the storage method depends on the definition of the protocol, we provide an example of a possible template for commands in the next chapter.

Because of this variability in the storage of commands inside the Dictionary, we add the Request Builder module. This module is the only one which has direct access to the Dictionary module and the only aware of the different storage

methods inside the Dictionary. The Request Builder works as a broker between the Dictionary and the calling module, which typically is the Learning module. Through the Request Builder, the caller can specify what command it wants to send to the device and the Request Builder, after accessing the dictionary and obtaining all needed information about that command, builds the command and returns it as a string to the caller. This string is ready to be sent to the IoT device through the Device Communication module.

The Dictionary module could contain multiple dictionaries for different protocols. For this reason, the Request Builder module contains a builder interface that refers to the protocol-specific request builder, accessing the correct Dictionary instance.

To add the support for a new protocol, the specific Dictionary should be added to the Dictionary module, along with the specific Request Builder module.

5.5 RL modules

The remaining modules of the RL attacker to be described are the Learning module and the State Machine module. These are the only modules related to the Reinforcement Learning aspect of the framework: they provide to the RL attacker the environment, the RL algorithms to solve the RL problem, the definition of the states and actions, and all necessary structures to support the learning process of the RL agent. Before going into the details of these modules, we define the entire RL problem, along with the environment and the RL agent.

5.5.1 Construction of the IoT environment

For our framework, the RL *environment* is built in such a way that all necessary information can be retrieved by external modules. We can say that the environment is built upon a general structure, suitable for all IoT protocols. The environment then defines different states, actions and reward signals for each IoT protocol. The definition of the state space is made inside the State Machine module, for each IoT protocol, and also all transitions between different states are defined and computed inside the same module. The action space can be retrieved from the Dictionary module. Finally, the reward signal is also defined inside the State Machine module and given to the RL agent at each time step.

The *states* of an IoT protocol are defined a priori, basing on the properties that an IoT device can have. For each device, the corresponding protocol defines some properties that are used to monitor the behaviour and settings of the device. Referring to the previously made example of a light bulb, this device can have as properties the power, the rgb and the brightness values. We use these properties to define our state space: the properties are turned into attributes v_i for $i \in (1, ..., n)$,

which define the existing states $s_t = (v_{1_t}, ..., v_{n_t})$ for the IoT protocol. For each combination of the values of the attributes we have a possible different state.

The *actions* that the RL agent can perform are the possible commands that can be sent with the specific protocol. These commands are stored inside the Dictionary module, divided by protocol. The agent can retrieve those commands accessing the Dictionary module through the Request Builder module, which, if asked, can provide also a list of all available commands. This list is used to initialize the Q matrix for the RL algorithms.

The *reward* signal is the numerical value assigned by the environment at each time step t. From state s_t, the agent selects a_t and ends in state s_{t+1}. At this point, the reward r_{t+1} is given to the RL agent. We model the reward signal to be dependant on different conditions, as the response of the device to the command sent, the number of steps the agent needs to reach the terminal state and the path that the agent follows inside the state machine. Thus, the reward value is computed as the sum of the following three values:

1. *Reward for the response.* The first component of the reward depends on the response of the device to the command sent by the RL agent, which is what we previously introduced as the feedback loop. Response messages for multiple IoT protocols are different but, in general, we can divide the responses in three categories. First, the response could describe the correct execution of the the command sent to the device. This answer gains no reward, 0, since this means that the command sent was correct and we do not want to punish the agent in this case. Often, these responses contain keywords as "success" or "ok". Then, a second type of answer exists, which contains keywords like "error" or "invalid", that refer to an invalid command sent or to the incorrect execution of the command on the device. This response results in a small negative reward given to the RL agent, as -10. Since at the beginning our RL agent knows all available commands for a protocol from the Dictionary, but it does not know whether they are valid for the targeted device, this reward helps the agent differentiate between valid and invalid messages. Moreover, if the response is not as we expect, we say that the response is bad formatted. Both in this case and when the response is not present, we give a negative reward, as -20. We consider this case worst than the previous one: before we used a command that was invalid for the specific device, but it was well-formatted so that the device properly answered, while in this case the device could have ignored the incoming request because of a bad format of our request, or because the message was not correct from the protocol point of view. Also the bad formatted response belongs to this worst case because we have no information on how to interpret it. Note that the structure of this component of the reward is general for all protocols, but the notion of what is a correct or an error answer highly depends on the specific IoT protocol, since also the

keywords depend on the protocol[7];

2. *Reward for the number of steps.* Since the goal of our RL agent is to go through the optimal path of the state machine as soon as possible, this part of the reward models the time in which the agent reaches the terminal state. A small negative reward, as -1, is given to each additional command, thus each action performed by the agent. This corresponds to giving a -1 reward at each time step t;

3. *Reward for the followed path.* This is the component of the reward that actually models the reward obtained by the agent for reaching the terminal state while following a predefined path, hence sending a specific sequence of commands to the devices. This value is 0 for all time steps except for the last one, when the RL agent arrives to the terminal state. This reward can be positive or negative. We assign a high positive reward for following the optimal path of the state-machine, the one we want the agent to learn. We assign a lower positive reward if the agent follows suboptimal paths. If the agent followed an undesired path, this part of the reward is 0. Note that the notion of optimal, suboptimal and undesired path is purely arbitrary. According to our needs and interests, we can model differently this component of the reward, in order to favor some paths inside the state-machine, and test our framework with multiple reward functions. While the other two parts of the reward are equivalent for each IoT protocol, this third part is specific to the IoT protocol, and more precisely it is specific to the state-machine defined for the protocol.

Given this reward structure, the RL *agent* selects actions from the Dictionary at each time step t, trying to maximize the final reward. In our framework, the RL agent solves a RL problem in which a model of the environment is not available and in which both the value function and the optimal policy should be learned. Hence, the agent can use different RL algorithms to perform both the prediction and the control problem: SARSA, Q-learning, SARSA(λ) or Q(λ).

5.5.2 Learning module

The Learning module is the core component of our framework. The main feature of this module is to actually implement the RL agent of our RL problem. In order to do so, it is connected to all other modules, calling their methods to perform specific tasks. We go now through the execution flow of this module.

[7]The problem of deciding how to divide answers into positive, error and bad formatted for all protocols in general is and will remain an open question throughout our work.

Before starting, this module receives a Discovery Report generated by the Discovery module, containing the information about the targeted IoT device, such as IP address, port and IoT protocol. With this information, the learning process can start.

At the beginning, it retrieves information about the environment. It retrieves the number of available actions from the Dictionary module and, from the State Machine module, it retrieves data about existing states, the optimal policy and the optimal path inside the state-machine. Information about the optimal policy and the optimal path are not needed nor used by the agent, but can be used after the learning process to check if the agent learned correctly.

At this point, the RL algorithm can start. Based on the RL algorithm chosen at configuration time, one between SARSA, Q-learning, SARSA(λ) and Q(λ) is used by the agent. These algorithms are implemented in this module in an analogous way to their implementation for the TCP use case, as presented in section 4.2. As previously, we rely on the pseudo-code of the algorithms presented in section 3.1, implementing them without relying on external libraries.

The high level structure of the algorithms implementation comprehends first an initialization phase. In this phase, collected information about the environment are used to initialize the Q value matrix, and the eligibility matrix if required by the algorithm.

Then, an outer loop over the number of episodes is started. Inside it, an inner loop is performed over the time steps, until the terminal state is reached. For efficiency purposes, a maximum number of time steps is set so that if the terminal state is not reached the episode can finish. Inside the inner loop the core of each algorithm is implemented.

We use the same implementation and structure of the algorithms as in the TCP toy case. In respect to it, we add Q(λ), which analogously to the SARSA(λ) algorithm relies on the eligibility matrix. For this new algorithm we also add a new update method.

Therefore, the configurable parameters for each algorithm are the exploration-exploitation trade-off ϵ, the learning rate α and the discount factor γ. For TD(λ) algorithms as SARSA(λ) and Q(λ), also the trace decay λ is a configurable parameter. Moreover, we add a decay process to the value of ϵ, in such a way that the ϵ exploration-exploitation trade-off is decreased over time. Hence, after E_{decay} episodes the ϵ value is updated in this way:

$$\epsilon \leftarrow \epsilon - D * \epsilon$$

where D is a small real number close to 0. Since we always want a percentage of exploration, we can set a minimum threshold for ϵ to 0.1.

When the outer loop ends, the learning process is completed and the Q matrix is saved inside an external file. Actually, throughout the entire learning process,

the Learning module collects data into external files. Before the process starts, all values for the configurable parameters are saved into an external file: information about the path to learn, the optimal policy, the algorithm chosen, the number of episodes, the values for α, γ, λ and ϵ. If one wants to reproduce an execution of the learning process, all the parameters saved inside this file allow for repeating the learning process using the exact same configuration. After the learning process has started, for each step t performed by the RL agent a log file is updated, with information about the current state s_t, the performed action a_t, the new state s_{t+1} and the reward r_{t+1}.

Then, at the end of each iteration of the outer loop, the Q matrix is written and updated inside an external file. These intermediate savings have been designed because the entire framework relies on the communication with IoT devices. A RL algorithm could send more than 1000 commands and multiple IoT devices set some threshold on the maximum amount of commands to receive inside a certain interval of time. For this reason, the IoT device could crash or could stop answering if the rate of commands is too high. Moreover, the answering time of a real IoT device can also be slow. Hence, saving the Q matrix at each episode allows for situations in which the learning process is interrupted. The learning process can be restarted later using the previously computed Q matrix. Also the eligibility matrix is saved at each iteration of the outer loop, in the cases in which the RL algorithm requires it. Moreover, inside the outer loop a file is created to store the reward R_E obtained at each episode E and the number of time steps N_{t_E} for the episode E. These values can be later used to evaluate the performance of the learning process and compare different RL algorithms, using the metrics described in subsection 3.1.4.

At the end of the learning process, some options allow to decide whether to call two external scripts: a script for showing some graphs about the learning process efficiency[8] and a script that allows for following the best policy found by the RL agent and evaluate the final reward, restarting from the initial state. Because we save all necessary information into external files, these scripts can be executed later, whenever it is needed.

5.5.3 State Machine module

The last module of this framework is the State Machine module, which mainly defines the goal of the RL agent. Therefore, this module is also called Goal module in Figure 5.1.

This module is specific for each IoT protocol because, as we mentioned before, each protocol has its own state-machine. Inside the state-machine it is defined the

[8]Also, there exists multiple scripts for generating different graphs, that are collected into a Plotter module.

goal that the agent should achieve. With this module, we decide to define the state machine and the reward signal outside the Learning module. The State Machine module contains methods for:

- Retrieving information about all existing states. The RL agent should know the states in order to initialize and update the Q matrix;
- Computing the current state after the agent performed an action. This method calls a specific API function, the one in charge of collecting information about the current property values of the targeted IoT device. This method, knowing the current state s_t, should compute the next state s_{t+1};
- Obtaining information about the optimal policy and the optimal sequence of states. This information is not provided to the RL agent but can be used to verify if the agent has learned how to walk through the optimal sequence of states and/or if it learned the optimal policy. Note that multiple optimal policies could exist, in situations in which an IoT protocol defines multiple commands to perform similar actions on the devices.
- Defining the reward signal. As we mentioned previously, a part of the reward is given based on the correct or "error" response of the device, or if there is no answer. Another part of the reward is given by each additional command sent to the device. Finally, the last part of the reward is the one depending on the state-machine. When the RL agent reaches the final state, a reward is given based on the path that the agent followed. If it followed an optimal or sub-optimal path, a high positive reward is assigned to it; otherwise no additional reward is given.

Through the usage of some configuration variables, multiple state-machines and paths can be selected and defined for a protocol, in order to change the "attack" pattern that our RL attacker should learn to perform. In the next chapter we will present multiple paths that can be defined for a single protocol, the Yeelight protocol.

5.6 Integration of all modules

In order to let the RL attacker work, all previously described modules automatically manage calls to other modules. To start and coordinate the execution flow, we add two extra components: a configuration file and a main script.

The configuration file allows for the current working of the entire framework, defining global information useful for all modules.

As the first step, the main script starts and runs the Discovery module. Hence it calls the Network analyzer and waits for a list of all Discovery Reports created by

the Network Analyzer. Then the main loops analyze the list of reports, running a thread for each device found in the network. Actually, a threshold to the maximum number of concurrent threads must be set. Each thread performs the learning process for the IoT protocol specific to the found IoT device. In fact, threads start executing the Learning module exploiting the Discovery Report passed to them. Inside the main script it is possible to select only the devices in which we are interested in performing the learning process.

The main script manages the runtime execution of this RL framework. As mentioned above, when the runtime execution ends, it is still possible to perform two different operations. First, it is possible to follow the best policy found by a RL agent exploiting a certain algorithm, through the appropriate script. Then, it is possible to access generated files and use the Plotter module to evaluate the performance of the learning processes.

In the next chapter, we will provide further details of the designed framework, presenting a first implementation of it.

Chapter 6

Implementation: Yeelight component

To prove the working of the RL framework we designed, we start the implementation process mainly focusing on an actual protocol. Starting from the structure of the RL attacker presented in the previous chapter, we develop the support for the Yeelight protocol. This is the first entirely supported protocol and its main goal is to show - before adding new protocols - that the solution we designed actually works and can achieve satisfactory results.

We decide to start with the Yeelight protocol for multiple reasons. The first and most important regards the presence of a clear documentation [27]. This document clearly describes how to communicate with Yeelight devices, what are all the available commands and responses to exchange with the devices. This documentation highlights also a clear and strict syntax, which helped us to construct a complete dictionary for the protocol. Besides, the protocol relies on standard technologies, such as commands encapsulated with the JSON format inside TCP packets, sent through Wi-Fi. Moreover, Yeelight devices are easily available and configurable to work inside the local network.

The Yeelight implementation reflects the design choices previously made. Indeed, the Yeelight implementation of the framework supports all four RL algorithms: SARSA, Q-Learning, SARSA(λ) and Q(λ).

The RL framework is developed in Python, because of its suitability for developing Machine Learning programs. Also, it is openly available and it can be found at `https://github.com/giuliapuntoit/RL-framework-iot`.

6.1 IoT protocols

In this work, we apply Reinforcement Learning to IoT protocols. The main challenge when working with IoT systems is that IoT covers a huge range of different industries and use cases. Since now, many different IoT communication protocols exist, even if new alliances are being created in order to find standard solutions for IoT protocols [28].

Because of their heterogeneity, IoT protocols can be classified with multiple strategies, as dividing them by layers as the enabling technology, the infrastructure, the discovery process, etc. For our purposes, we are mostly interested into a few of these layers, which we are going to present. A complete guide to all possible IoT variants can be found in [29] and [28].

We focus on the following layers in which an IoT protocol can be categorized:

- *Physical and MAC Layer.* For this layer the most used communication protocols in IoT are Bluetooth and Wi-fi or, in some cases, ZigBee;

- *Transport layer.* For this layer, the choice of the protocol is limited to UDP and TCP protocols. In general, in low power environments TCP is not suitable, since it has a large overhead due to its connection-oriented property, while UDP is connectionless. In situations in which power consumption is not an issue but reliability matters, the TCP protocol can be used;

- *Application layer.* This layer refers to all those protocols that are responsible for data formatting and presentation. In the Internet, typically the application layer is HTTP-based. Because of the overhead introduced by HTTP, most of IoT protocols use different application protocols, as MQTT and CoAP. MQTT relies on the TCP transport protocol, while CoAP relies on UDP. Some devices, instead, use RESTful HTTP over TCP, which exploits the availability of HTTP for various platforms;

- *Semantic.* In most IoT devices, the semantic of the IoT protocol is based on the JSON or JSON-LD standards. A new standard has also been introduced for promoting the interoperability between devices and the rest of the Internet, the Web Thing Model;

- *Syntax.* Finally, the syntax defines how data should be formatted inside, for example, a JSON payload, in order to build a correct command for the IoT protocol used. Each protocol has its own syntax.

For a clearer understanding of IoT protocols and of this work, we now give an example of an IoT protocol: the Yeelight protocol.

6.1.1 Yeelight protocol

We present the Yeelight protocol with its layer-specific components. In this project we mainly focus on the Yeelight protocol since it is well documented and it has a clear and strict syntax [27].

The Yeelight protocol is a protocol developed for Yeelight smart LED products. For the physical and MAC layer, this protocol relies on Wi-Fi.

The Yeelight devices are first connected to the local network through a configuration procedure in order to pair them with the SSID and password of the router of the LAN. This initial procedure is performed once relying on Wi-Fi, and due to security issues it uses a proprietary protocol, which is different from the Yeelight protocol. After the Yeelight device is connected to the network, it is visible and can be controlled by other users or devices inside the same network.

For the phase of local control, two procedure exists: the discovery and the control procedure.

The discovery procedure is based on a SSDP-like protocol, in which two kinds of messages exist: the searching and the advertising messages. These messages use the port 1982, differently from the standard port for SSDP. For our purposes, we are interested in the searching message. The device will listen on a multi-cast address and will wait for incoming search requests. The search request should use UDP and should be well-formatted: if these conditions are verified, the device will respond to the address of the searcher with sending its IP and port, and with some basic information about the current state of the device itself.

Once the IP and the port of the IoT device are known, a control protocol is available to send commands to the devices. The control protocol is the core part of the communication with the Yeelight devices, and it is the one on which we focus in this chapter.

The control protocol defines a set of commands which can be expressed using the JSON format and sent on a TCP connection. All messages must end with "\r\n". The Yeelight smart LED control protocol allows for three different messages: the "command", the "result" and the "notification" messages. Since we will design our framework to send command messages and receive result messages from the IoT devices, we are not presenting the construction of the notification message, which can be found in the documentation in [27].

We now present the two main commands:

- The command message can be generated by 3rd party devices and sent to the smart LED. The command has the following JSON format:

  $$\{\text{“}id\text{”} : id\_value, \text{“}method\text{”} : method\_value, \text{“}params\text{”} : par\_values\}\backslash r\backslash n$$

  The value of the id is an integer decided by the sender. The same *id_value* will be used in the result message and it is used to correlate request and

response messages. The method is a string which specifies which operation to perform on the device. The *par_values* field is an array containing values, if needed, for the specific method. For example, if the method is "*get_prop*" the parameters array could be ["*power*", "*rgb*", "*bright*"]. With this command it is possible to get information about the current status of power, color and brightness values respectively of a smart LED Yeelight device;

- The result message is sent by the smart LED to the sender of the command message. It could contain the result of the execution of the command on the device or some information about the values requested by the command message. It has the following format:

  $$\{\text{“}id\text{”} : id\_value, result\_outcome : result\_outcome\_value\}\backslash r\backslash n$$

  The value of the id should reflect the value of the command message. Then, if the method inside the command message has been executed successfully by the device, the *result_outcome* is the string "*result*". The *result_value* is an array containing either "*ok*" or some values if requested by the command message. Otherwise, if the command is invalid or not executed successfully, the *result_outcome* will be the "*error*" string. The *result_value* will be an object containing the error description.

To give a practical example, we focus on the "*set_rgb*" method. This method needs as parameters an integer representing the RGB value, a string representing the sudden or smooth effect for the color changing of the smart LED, and an integer for the duration of the effect when the effect is set to smooth.

Here we present a correct sequence of command-response messages:

$$\{\text{“}id\text{”} : 1, \text{“}method\text{”} : \text{“}set\_rgb\text{”}, \text{“}params\text{”} : [255, \text{“}sudden\text{”}, 0]\}\backslash r\backslash n$$

$$\{\text{“}id\text{”} : 1, \text{“}result\text{”} : [\text{“}ok\text{”}]\}\backslash r\backslash n$$

On the opposite, a sequence of command-response messages for an invalid command sent is presented in the following:

$$\{\text{“}id\text{”} : 1, \text{“}method\text{”} : \text{“}set\_rgb\text{”}, \text{“}params\text{”} : [255, \text{“}invalid\_string\text{”}, 0]\}\backslash r\backslash n$$

$$\{\text{“}id\text{”} : 1, \text{“}error\text{”} : \text{“}code\text{”} : -5000, \text{“}message\text{”} : \text{“}generalerror\text{”}\}\backslash r\backslash n$$

Additional information on this protocol can be found in [27].

6.2 Modules for the Yeelight component

After having presented the Yeelight protocol, we can coming back to our implemented framework. Following the design of the framework in chapter 5, for

implementing the Yeelight protocol we have to add some protocol-specific components, and add the support for this protocol into general components, as the Discoverer module. We already presented all necessary information about the Yeelight protocol structure in the previous section.

We will now describe each module in details, referring to chapter 5. Note that if some implementation details are not explicitly cited, we assume them to be implemented exactly how specified during the design of our framework.

Basing our description on the design process, the RL attacker for the Yeelight protocol is implemented in the following way:

1. **Discoverer**. Since the Discoverer is a general component, hence not specific for a single IoT protocol, this module is implemented as explained previously. It is made by a Network Analyzer and a Discovery Report. The Network Analyzer uses nmap to scan all IPs inside 192.168.1.1/24. From the results obtained, it filters the devices we are interested in.

   For the Yeelight support, it looks for Yeelight devices, searching for matching the host name to keywords for the Yeelight protocol. After having performed some experiments on our LAN, we decide to use as keywords for the Yeelight protocol the terms "yeelight" and "yeelink", which are the ones used by Yeelight devices. An example of a Discover Report for a Yeelight device can be found in Table 5.1;

2. **API**. The API script is the module in charge of interacting directly with IoT devices. The API script for the Yeelight protocol contains the required methods for sending commands and handling the responses - with or without computing a reward based on the answer, and a method for retrieving the property values for a Yeelight device. Moreover, we implement some additional methods. Because of the fact that the Yeelight protocol is the first supported protocol, we added some extra methods for performing a Yeelight specific discovery process. These methods can be used by the Learning module in order to discover and make a list of all Yeelight devices connected to the LAN. When the Learning module is executed directly, without the coordination of the main script, this module looks for Yeelight devices and performs the learning process for these devices. Indeed, if the main is not present, the Discoverer module is not used and no Discovery Report is passed to the Learning module. These discovery methods for the Yeelight protocol will no longer used when the support for more protocols is implemented;

3. **Dictionary**. For the Yeelight dictionary we decide to store commands as templates. Recall the example we made in subsection 6.1.1 for the structure of a Yeelight command, to change the RGB value of a Yeelight colored bulb:

$$\{\text{“}id\text{”} : 1, \text{“}method\text{”} : \text{“}set\_rgb\text{”}, \text{“}params\text{”} : [255, \text{“}sudden\text{”}, 0]\}\backslash r\backslash n$$

All commands are structured in an analogous way. For each available method, the documentation of the protocol specifies what are the possible parameters and what are the minimum and maximum numbers of necessary parameters for that method. Also, it specifies which of the parameters are mandatory and what is the type for each parameter, hence whether a certain parameter is a string or an integer, and what are the admissible values. For example, the "set_rgb" method allows for three parameters: an integer value from 0 to 16777215 representing the RBG value, a string between "smooth" and "sudden" for defining the effect of the method and an integer value that indicates the time in milliseconds for performing the method, in case the effect is "smooth".

With all these information for each command, we can define a common template for methods, as presented in Table 6.1. For each method we store in the dictionary information about the name of the method, the minimum and maximum number of parameters and the names and possible values for parameters.

From the dictionary, the Request Builder component for the Yeelight protocol accesses the stored method and retrieves all necessary information. It randomly decides the values for parameters and builds the final command as a JSON formatted string;

4. **Learning**. The Learning module works exactly how described during the design process. It retrieves information about the RL environment from external modules and it implements all four RL algorithms for the RL agent. It communicates with devices through the API script. It provides the available actions for the RL agent accessing the Dictionary module through the Request Builder module. From the State Machine module, it gets information about states, optimal path and optimal policy. The RL agent gets the reward at each time step as a sum of three values: one value is given for each additional step, one value is assigned by the API script based on the response of the device and the last value is obtained from the State Machine module, based on where the RL agent is inside the state-machine.

   Specifically to the Yeelight component, we add some sleep time for the RL agent at each action performed and a higher sleep time after having performed a certain amount of actions on the device. We implement this pause mechanism because the Yeelight protocol limits the number of messages that can be sent to a device in 1 minute. The maximum rate allowed is of 60 commands per minute [27]. If the agent exceeds this threshold, the device is switched automatically off and, whereas the lamp will respond to commands normally, it will not change its state anymore, unless an additional minute is waited.

   As expected, the Learning module saves all generated data inside external files.

In respect to the designed Learning module, the implemented module behaves differently if it is called by the main script or directly. If the main script coordinates the execution of the framework, it calls the Discoverer and passes to the Learning module the Discovery Report with data about the targeted Yeelight device. If no report is given to the module, or if the Learning is executed directly, the module first performs a discovery process inside the LAN to retrieve information about all Yeelight devices connected to the Wi-fi. This discovery procedure is done using specific methods provided by the Yeelight API script;

5. **State Machine**. The Yeelight State Machine module has information about possible states, the optimal path and the optimal policy. From state s_t, after the agent performs a_t, this module computes the next state s_{t+1} analyzing the current property values of the device, using the method defined for this purpose inside the API script. It also knows the state-machine defined for the Yeelight protocol, and it computes the last part of the reward, based on the position of the RL agent inside the state-machine diagram. This module will be described later, after presenting the Yeelight environment.

Template	**Method**
Name	set_rgb
Min parameters	3
Max parameters	3
Parameters	rgb_value → integer
	effect_value → string
	duration → integer

Table 6.1: Example of the template for storing Yeelight commands inside the Dictionary module.

The last two models are the ones related to the RL problem defined for the Yeelight component. Our RL problem is composed by the RL agent, which is implemented by the Learning module, and by the definition of the environment, which is described inside the State Machine module for the Yeelight protocol. To understand what the State Machine module contains, we need to present how we defined the state-machine for the Yeelight protocol, considering also how we model states and rewards.

6.3 Yeelight environment

The State Machine module is the one actually implementing and modelling the RL environment for a Yeelight device.

We first define the state space basing on the properties that a Yeelight device has, that are used to monitor the behaviour and settings of the device. For the definition of the environment, we focus on Yeelight smart bulbs, which can be colored or single color bulbs. For them, the Yeelight protocol defines more than 20 supported properties. For our purposes we focus on 7 of them, which are the most significant considering Yeelight bulbs. Therefore, the state space is defined as a vector $\mathbf{v}$ of 7 dimensions:

- v_1: reflects the *power* property of the device. It has two possible string values: "on" and "off";
- v_2: indicates the *rgb* value of the device. It is an integer value inside the interval from 0 to 16777215;
- v_3: indicates the *brightness* value of the Yeelight device. It is an integer value between 1 and 100;
- v_4: reflects the *color temperature* for a Yeelight bulb. Its integer values range from 1700 to 6500;
- v_5: is the *name* of the device. Any string is a valid value for this attribute;
- v_6: reflects the *hue* property of a colored bulb. Its integer values range from 0 to 359;
- v_7: reflects the *saturation* property of a bulb. Its integer values range from 0 to 100.

Hence, for each time step t, the state is uniquely defined by the values of these attributes at step t, as $s_t = (v_{1_t}, v_{2_t}, v_{3_t}, v_{4_t}, v_{5_t}, v_{6_t}, v_{7_t})$. Multiplying the possible values of each attribute of the state, we obtain that our state space is composed by more than 10^{20} different states.

All these attributes are retrieved by the State Machine module exploiting an ad-hoc method provided by the API script. This method retrieves all property values of the targeted device we are interested in.

For transiting from one state to another, the agent should perform actions. An action is any command sent to devices that may change their state. We already explained that the action space reflects the possible commands that can be sent with the Yeelight protocol. By design, these commands are stored inside the Dictionary module for the Yeelight protocol. The Yeelight protocol defines 35

possible methods to send to a Yeelight device. Note that not all of these are valid for all devices. Since our RL agent should be as much general as possible, we put inside the dictionary all those methods. Thus, we let the agent try all methods and after some time it will learn which actions are invalid for the current device, basing on the reward received during the learning process.

Example of methods are set_rgb, set_bright, set_power, toggle, etc. Their name are quite intuitive, as they change some of the properties we are interested in for retrieving the current state of the device.

For the Yeelight protocol, each method can also have one or more parameters. Some parameters, like the power value and the effect value allow for a small set of admissible values: power can be "on" or "off", and the effect can be "sudden" or "smooth". Most of the parameters, instead, can have a lot of possible values: the brightness can have 100 different values, the rgb parameter can have 16777216 possible values, the color temperature allows 4801 different values, etc. Therefore, one single method can have many possible variants, each variant with a different combination of the allowed parameter values. This results into more than 10^9 distinct commands that the RL agent could send to one Yeelight device.

With cardinalities of respectively 10^{20} and 10^9 for the state space and the action space, our RL problem is hard to manage. Moreover, these estimates are optimistic. Whereas RL algorithms have been designed for problems with small finite action and state spaces [30], many real problems have continuous or large spaces, as the ones we are presenting in this work. These big cardinalities make it difficult to manage Q matrices as we did in subsection 3.1.3, but would suggest instead the usage of Q value continuous functions. This would allow to approximate the computation of values for all possible state-action pairs. Using continuous functions in RL algorithms is a recent area inside the RL field of research and multiple alternatives have been proposed. The general idea is to use function approximation techniques [30] to compute continuous value functions. Then, these techniques are coupled with standard RL techniques, as SARSA and Q-learning algorithms. The most common example of function approximators are gradient-based methods, as neural networks: the most common RL algorithm for continuous spaces is the Deep Q-learning technique [31], which is based on convolutional neural networks and the Q-learning algorithm.

Since these methods are much more complex than standard RL algorithms, involving more data structures and computational resources, we prefer to simplify the definition of our environment. Our purpose is not to use complex methods in our entire state and action spaces, but to prove that our approach of using RL to automatically interact with IoT devices is actually working.

Therefore, we need a way to simplify and limit the action and the state spaces. To achieve this, we operate in two different directions. The first one is in charge of reducing the dimension of the action space. We decide to assign parameter

values to each method in a random way. Hence, the Request Builder, when it has to construct the command for a certain method, chooses randomly all the values, except the power value, with information about the range of value or admissible values. In this way, all values such as rgb, brightness and color temperature are chosen randomly. For what concerns the power parameter instead, we do not want it to be random. That is because setting the device on or off has two opposite consequences with respect to the state of the device. Thus, inside the Dictionary we split methods using the power parameter into two distinct methods: one with the "on" value and the other with the "off' value set. For the Yeelight protocol, only two methods rely on this parameter and they are split into two separated methods. Hence, instead of having 35 possible methods, we now have 37 methods. To make an example, the "set_power" method is stored into the Dictionary two times. The first stored method will be "set_power_on", while the latter will be "set_power_off".

The second direction in which we want to operate to reduce the number of dimensions of our RL problem regards the state space. Stated that our state space is composed by the previously introduced 7 attributes and we want to stick to this decision, we need a way to define the goal of our RL agent.

We will define our goal, and consequently model the reward signal. To do so, we build a simple state-machine, taking into consideration only three attributes: power v_1, rgb v_2 and brightness v_3. We want the RL agent to learn to send some commands in order to modify these attributes. Inside this state-machine, we perform a transition from one state to the other when one or more attribute values have been modified. For attribute values that are not binary, like rgb and brightness, we do not want to set each property of the device to a unique specified value, but we want that the RL agent learns how these properties can be modified. Indeed, these values inside commands are chosen randomly by the Request Builder module, and so the RL agent is not capable of learning how to put these values inside commands. Instead, the RL agent is learning which method to send when it is at a certain state of the state-machine. For the power attribute, instead, the command for changing the power value has been split into two distinct commands, hence the RL agent can learn when to select one or the other based on the Q matrix and on the current state s_t.

The first state-machine we draw is the one presented in Figure 6.1 and we refer to it as "Path 1". In this state-machine - and in the following ones we will present soon - we are not showing single states s_t, but each box represents a set of states in which one or more attribute values have been modified in respect to the previous state[1]. In this way the state-machine can be easily represented and then

[1]From now on, we could refer to the boxes of the state-machine as states, actually referring to

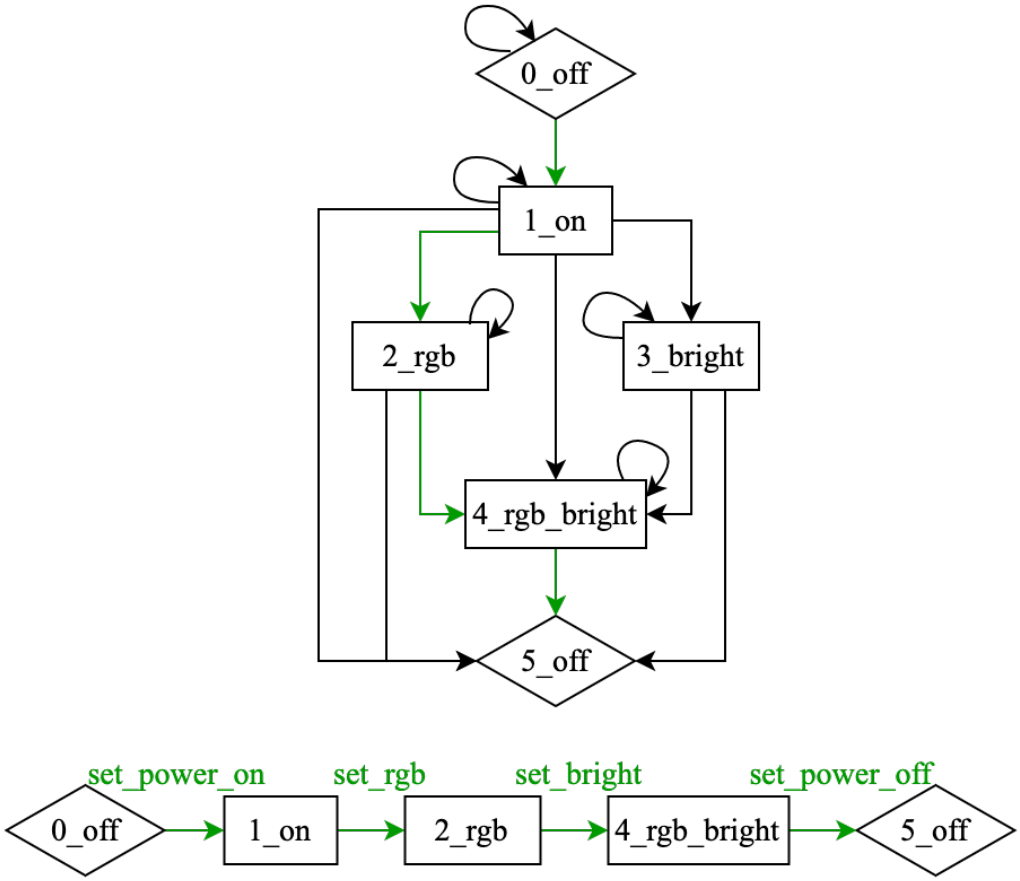


Figure 6.1: State-machine for **Path 1** defined considering 3 attributes: power, rgb, brightness. The green path represents the optimal path, which is composed by 4 steps. Note that the names of the boxes are identifiers for the current position of the RL agent inside the state-machine.

implemented. The names assigned to each box refer to the attributes modified by the RL agent during the current episode. The optimal path is the green one in the diagram, which is represented also detached from the state-machine, below it. One of the possible optimal policies is indicated by the commands in green in the figure. All possible policies that the agent could learn starts from state "0_off" and reaches terminal state "5_off", passing through the optimal path. Note that the integers at the beginning of each box name are used only to facilitate the reading and the explanation of the diagrams. There exist multiple optimal policies

the set of states that the boxes are representing.

because of the fact that multiple methods of the Yeelight protocol perform similar operations, modifying the same property values of a device.

Now, we show in details the meaning of each box in the state-machine, since we will use the same strategy to assign identifiers also in the diagrams we will present later:

- **0_off**: this box represents the starting state s_0, when the power attribute has the value "off". At the beginning of each episode E, the Yeelight bulb is switched off. Commands that modify the rgb and the brightness values have no effect when the device is in this state;

- **1_on**: the power attribute is turned to the "on" value. For example a Yeelight bulb can be switched one with the set_power_on or the toggle commands;

- **2_rgb**: this box represents a state $s_t = (v_{1_t}, v_{2_t}, v_{3_t})$ of the Yeelight device in which, with respect to the initial state $s_0 = (v_{1_0}, v_{2_0}, v_{3_0})$, the device has been switched on and the rgb value has been changed, hence $v_{1_t} =$ "*on*", $v_{2_t} = v_{2_0}$ and $v_{3_t} \neq v_{3_0}$;

- **3_bright**: this box represents a state in which, with the respect to the initial state, the device has been switched on and the brightness value has been changed, while the rgb value is the same as in the starting state s_0;

- **4_rgb_bright**: this box models any state $s_t = (v_{1_t}, v_{2_t}, v_{3_t})$ of the Yeelight device in which, with respect to the initial state s_0, the device has been switched on and both the rgb and the brightness values have been modified, hence $v_{1_t} =$ "*on*", $v_{2_t} \neq v_{2_0}$ and $v_{3_t} \neq v_{3_0}$;

- **5_off**: it represents the terminal state of each episode. When the agent reaches this state, which can be reached from any state different than the starting one, the episode ends. This state can reached sending commands such as set_power_off or the toggle command to the Yeelight bulb.

We decide the optimal path to be the one in green inside the figure, in which the RL agent will obtain the maximum total reward. This path consists in switching on the device, changing the rgb and the brightness values, and then switching off the device. One possible optimal policy is given by the actions set_power_on, set_rgb, set_bright and set_power_off. The optimal path, and therefore the optimal policy, has a length of 4 time steps. Note that in the figure the actions representing the commands are represented without parameters. That is because the RL agent retrieves its actions from the dictionary, in which methods are not distinguished basing on the values of their parameters, except for the set_power method. As explained before, the values for rgb and brightness are randomly chosen by the

Request Builder module, hence the agent is not aware nor it is learning how to properly change these values inside commands.

Finally, it is important to note that each box inside the state-machine has a self-reentrant arrow, meaning that there exist commands that do not make the RL agent to move inside the state-machine. For example, if the set_power_on command is sent in state 1_on, the state stays the same: the device is already switched on, hence $s_{t+1} = s_t$. Also, if in this state-machine we are considering only 3 attributes (v_1, v_2, v_3), if a commands modifies another attribute, like v_5, the RL agent is still in the same place inside this state-machine.

With the same considerations, we draw and implement also other two state-machines, for "Path 2" and "Path 3". In chapter 7, we will refer to Path 1, 2 and 3 to perform tests and make considerations on the results of our framework.

Note that the identifiers assigned for the boxes in these state-machines follow the same strategy we adopted for Path 1. Moreover, self-reentrant arrows exist also in these state-machines, but we decide to remove them from the diagrams presented in this work. This was done for clarity purposes, because these diagrams are much more complex than the previous one.

In Figure 6.2, the state-machine for Path 2 is presented. This diagram has more boxes, and considers 4 attributes: power v_1, rgb v_2, brightness v_3 and the name of the device v_5. The goal in this state-machine is to switch on the device, change the name, change the brightness value and switch the device off, in the less possible number of time steps and without changing the rgb value. We will see later when describing our reward signal that the agent will not obtain a positive reward if the rgb value is modified during the episode. For Path 2, the optimal path and the optimal policy have length 4, as for Path 1, but the diagram is more complex. For example, one could change the name of the device even if the device is still switched off. Also, if the rgb value of the device is accidentally modified, the RL agent will obtained a really low final reward. Recall that all these diagram and paths definitions are purely arbitrary.

The last state-machine we implement for the Yeelight component is presented in Figure 6.3. For this "Path 3" we decide to make things more complicated, trying to make the learning process harder for the RL agent. This state-machine relies on 4 attributes[2]: power v_1, rgb v_2, brightness v_3 and color temperature v_4. In this diagram, the optimal path is 7 steps long, and contains an intermediate state in which the Yeelight device is switched off and then re-switched on in order to process inside the state-machine. We model the reward of this path analogously to the Path 2, in which if the rgb value is changed no positive reward is given to the

[2]Using more attributes and more complicated state-machines is still possible, but they would be hard to understand visually.

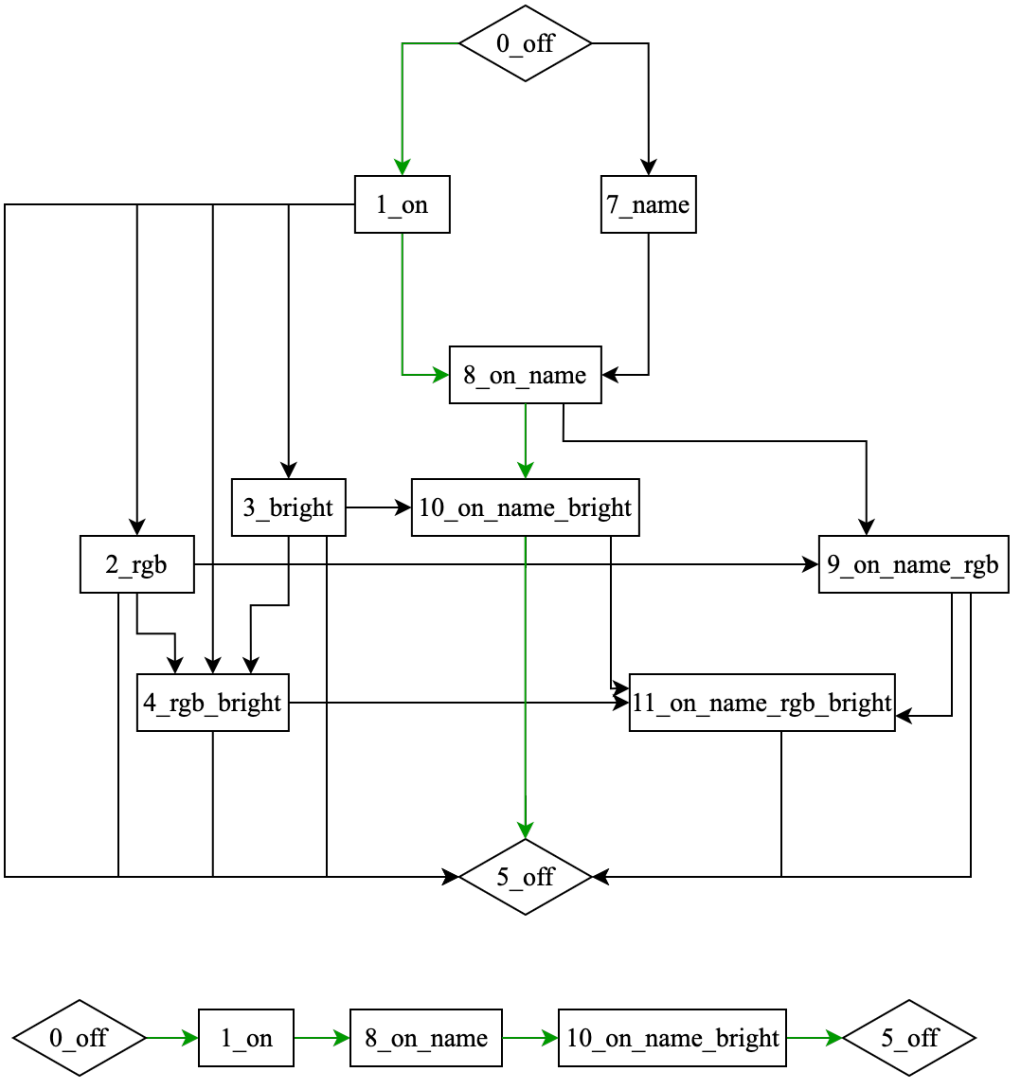


Figure 6.2: State-machine for **Path 2** defined considering 4 attributes: power, rgb, brightness, name. The green path represents the optimal path, which is composed by 4 steps. Note that the names of the boxes are identifiers for the current position of the RL agent inside the state-machine.

RL agent.

All these state-machines, along with their optimal paths and policies are implemented inside the State Machine module for the Yeelight protocol. For now, the specific path that the RL agent should learn can be selected in the configuration file of the RL framework. Based on the selected path, the State Machine

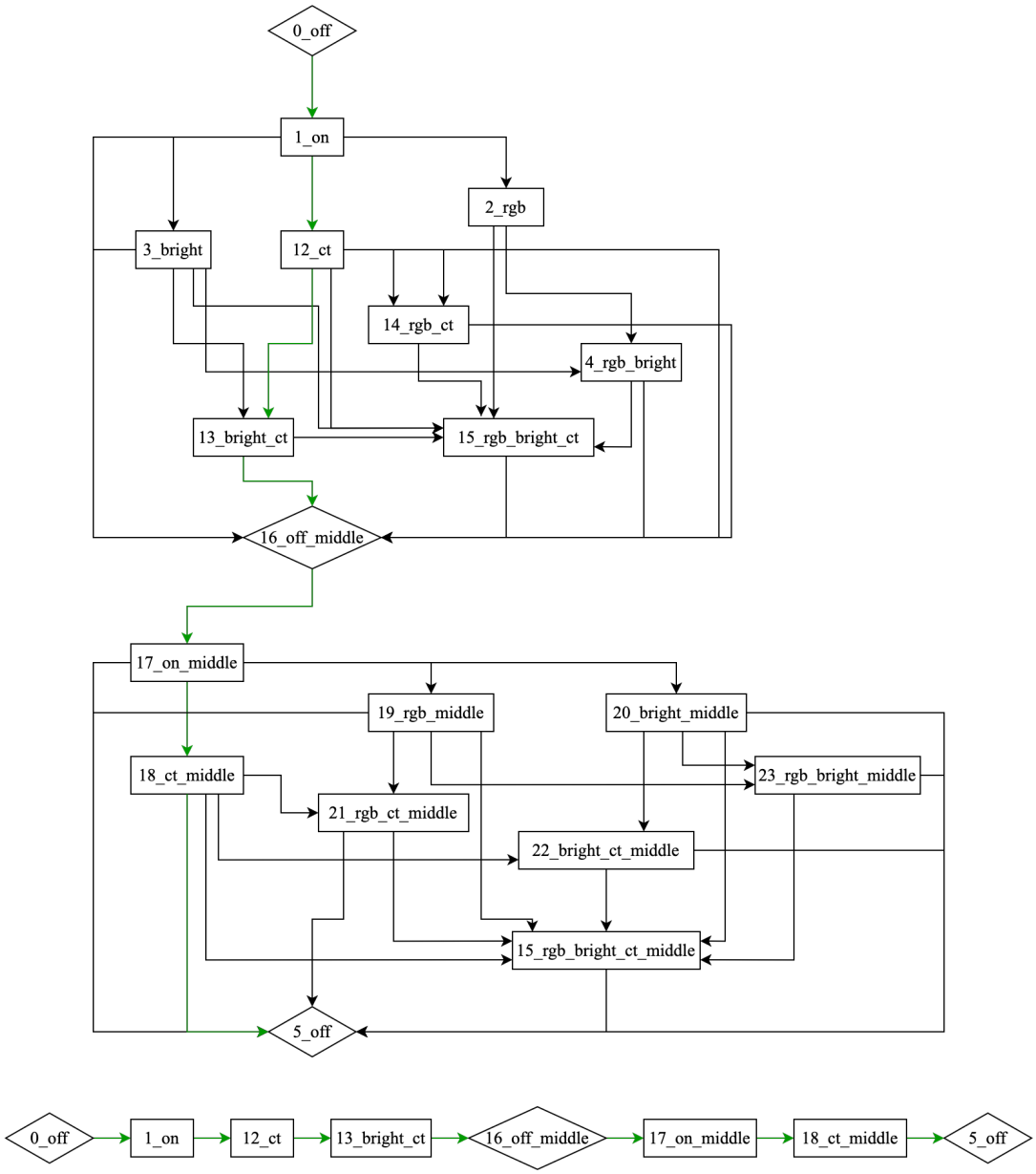


Figure 6.3: State-machine for **Path 3** defined considering 4 attributes: power, rgb, brightness, color temperature. The green path represents the optimal path, which is composed by 7 steps. Note that the names of the boxes are identifiers for the current position of the RL agent inside the state-machine.

module computes the transitions among the different states represented by the state-machine diagram of that path.

At this point we have fully described our environment and we need to present our reward signal. Following the design choices in subsection 5.5.1, the reward at each time step is computed as the sum of three values: one part is based on the response of a device to one command, the second part is assigned for each additional time step t to reach the terminal state, and the last part is based on the current position of the RL agent inside the state-machine. While the first two parts are fixed for any protocol, the third part depends strictly on the state-machine and on the optimal path we want to address for the Yeelight protocol. This part of the reward is the one modelled inside the State Machine module, which returns to the RL agent this part of the reward at each time step t.

For all state-machines, we define "optimal" paths, "sub-optimal" paths and "undesired" paths.

Optimal paths are the ones obtaining the maximum reward. If the agent follows these paths, it will gain a high positive value for the third part of the reward, as +200, when reaching the terminal state. At each step different than the last one, the agent receives a 0 value for this part of the reward. We give 0 reward because in RL problems we want the agent to learn which is the optimal path, not how to go through it [20]. E.g., if we give a positive reward to each state of the optimal path, we would be saying to the agent how to obtain the maximum final reward, helping it to follow the optimal path. In general, in RL problems, a positive reward is given only when reaching the terminal state after performing the actions we want the agent to learn [20].

To complicate the learning process of the RL agent, we also define sub-optimal paths, which are paths that still obtain positive rewards, but lower with respect to the optimal path. A RL agent could learn a sub-optimal path and find the optimal path later, or it could learn the sub-optimal path and be stuck into it.

Finally, all remaining paths reaching the terminal state are undesired paths. No positive reward is given at the end of the episode to the agent, hence in this case the reward received by the agent is only defined by the first two parts of the reward.

Referring to Figure 6.1, the optimal path is the one showed in green inside the figure. A sub-optimal path can be the one passing through states[3] 0_off, 1_on, 3_bright, 4_rgb_bright and 5_off, as we arbitrarily decide that we prefer to first change the rgb value and then the brightness value, and not vice-versa. An example of an undesired path in the state-machine for Path 1 can be the one in which the targeted device is switched on and immediately switched off. This path would pass through states 0_off, 1_on and 5_off. Even if this is the shortest possible path,

[3]Recall that the identifiers of the boxes in the diagrams are not states, but actually they represent sets of different states.

so the part the reward given at each time step is minimized, no positive reward is assigned to the agent when reaching 5_off.

Analogously, we define optimal, sub-optimal and undesired paths for state-machines for Path 2 and Path 3.

Regarding Path 2 in Figure 6.2, the optimal path consists in switching one the device, changing its name and brightness values, and then switch it off, passing through states 0_off, 1_on, 8_on_name, 10_on_name_bright and 5_off. Sub-optimal paths are the ones that deviates from the optimal path passing though states 3_bright or 7_name, in which the name attribute is modified when the device is still switched off. Undesired paths are all the remaining ones, which are practically the ones in which also the rgb attribute of the current state has been modified, with respect to the rgb value at the starting state.

For what concerns Path 3 of Figure 6.3, we complicate possible paths to test more in deep our RL agent. The optimal path consists in 7 steps: 0_off, 1_on, 12_ct, 13_bright_ct, 16_off_middle, 17_on_middle, 18_ct_middle and 5_off. The path consists in switching on the device, changing the brightness and the color temperature, hence turning the Yeelight bulb to white color. Then the device is switched off and turned on again, and the color temperature is changed again before the device is switched off. At a high level perspective, the goal for this state-machine is to perform some operation with the device without changing its rgb value but sticking to a white and not colored light for a Yeelight bulb. A sub-optimal path is the path passing through the same states as the optimal path except for the state 12_ct, which is replaced by the state 3_bright: hence, the agent first changes the brightness of the device and then the color temperature, and not vice-versa. Undesired paths instead are all possible paths which pass through states in which the rgb value has been modified.

The part of the reward signal related to the path followed by the RL agent is defined for each state-machine inside the State Machine module. As previously said, the targeted state-machine is defined through a configuration file, that will also influence how the reward given to the RL agent.

Eventually, concerning the RL agent, it is implemented exactly how previously modelled in subsection 5.5.1, selecting actions to perform between the ones provided by the Dictionary module.

6.4 Workflow of Yeelight component

At this point, we already described all single parts of the Yeelight component of our RL framework and we defined in details the environment and the agent for the RL problem.

A schema showing the complete RL framework is shown in Figure 6.4, which

focuses on the Yeelight component interacting with a single Yeelight bulb.

Before starting, in the configuration file some general information is present, for example the root directory in which saving output files or the state-machine and the goal that the RL agent should learn. When the framework starts, it is managed by the main script. This script first activates the Discoverer. After analyzing the local network, this module returns to the main script all reports created after the found devices. The main script receives these reports and calls the Learning module, passing to it also the Discovery Report for the Yeelight bulb found inside the LAN.

When the Learning module starts, it receives in input multiple parameters. Among these, the most important are the RL algorithm to use, with values for ϵ, α, γ and λ if needed, the total number of episodes, the number of maximum time steps per episode and some flags. These flags decide whether after the learning process the user wants to plot some results or to run the RL agent following the best policy found, using respectively the Plotter module or the Run Policy Found script. The Learning module is our RL agent, which iterates over episodes. For each episode it iterates until a terminal state is reached or when the number of time steps for that episode is above a certain threshold. During each episode, the agent asks for commands to the Request Builder, which access data from the Dictionary - the Dictionary for the Yeelight in our case - and returns to the agent a JSON string with the complete command to send to the Yeelight device. The Learning module then passes this string to the Device Communication module, more specifically to the API script for the Yeelight protocol, which sends commands to the Yeelight bulb and handles its responses. Moreover, at each time step the Learning module retrieves data about reward and current state from the State Machine module, more specifically from the State Machine Yeelight script. The State Machine module, in order to retrieve information about the state of the bulb, asks to the Dictionary module the command to retrieve all necessary information from the bulb and sends this command to the API script, which actually sends the command to the bulb and returns the response to the State Machine Yeelight module. At the end of the learning process, the Learning module generates some output files, for storing the Q matrix, intermediate reward and number of time steps values during the episodes and some log files, which can be used for error checking.

Output files can be used by the Run Policy Found script, which retrieves data from these files and follows the best policy found, through the Q-matrix. While following the policy, the script retrieves complete commands from the Dictionary module and sends them to the Yeelight device passing through the API Yeelight script. Also, output files can be used by the Plotter module to present graphically the results obtained in the learning process.

Recall that, as we said before, for the Yeelight component the Learning process

can also be started directly, without passing through the main and Discoverer parts. When this is the case, the Learning module performs a Yeelight specific discovery phase, accessing methods inside the API Yeelight script specific for this purpose. After information about the Yeelight bulb are retrieved, the Learning process works exactly as explained previously.

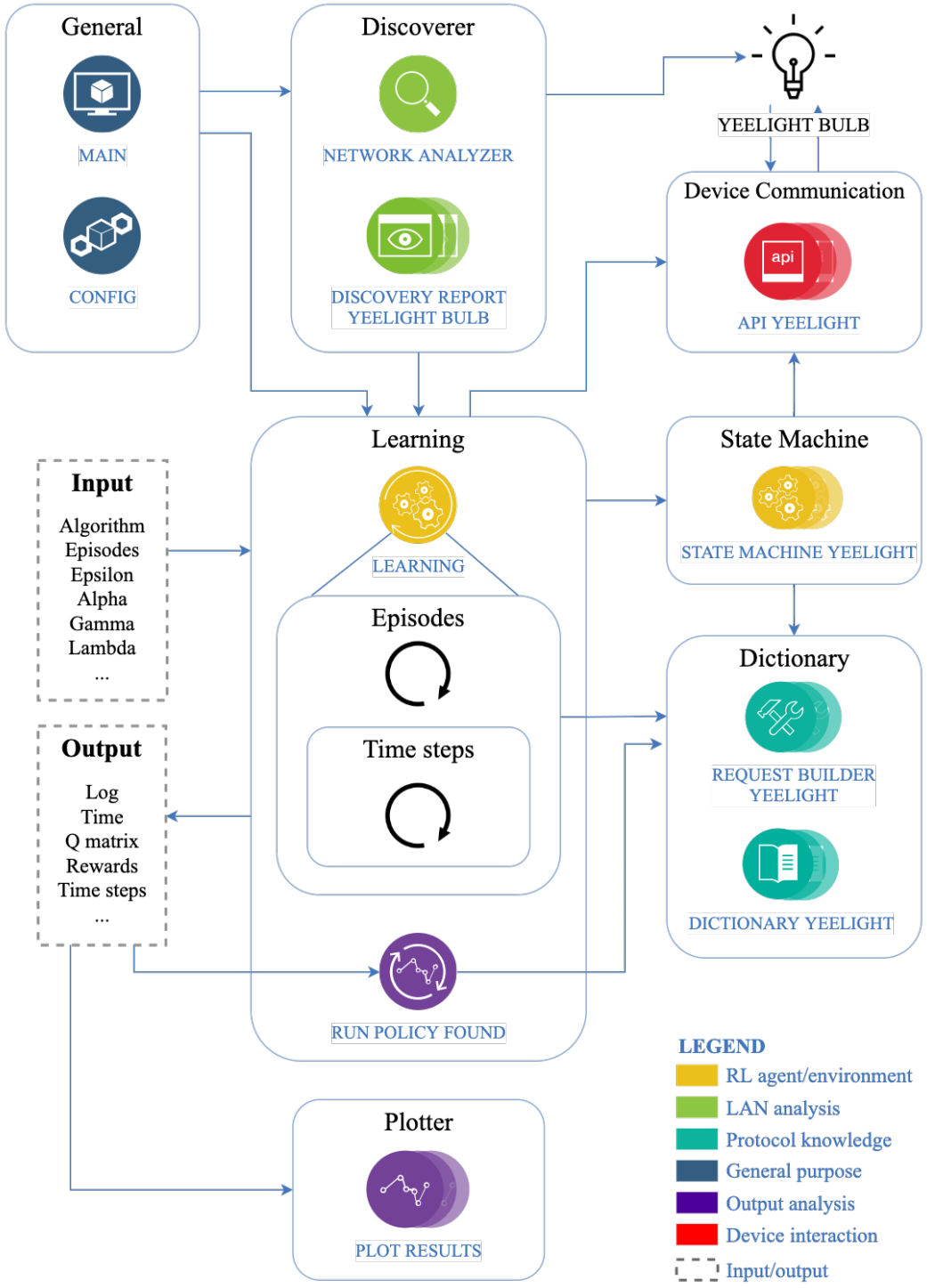


Figure 6.4: Schema of the RL framework focused on the Yeelight component.

Chapter 7

Results

In this work, we wanted to model a framework to learn how to interact with IoT devices, using Reinforcement Learning techniques. After having described the design of the framework and the implementation of the first component, the one targeting the Yeelight protocol, it is necessary to check whether this framework is actually working, before trying to target multiple protocols. In this chapter we present the results obtained with the Yeelight component of our framework, evaluating the performance of the RL algorithms we implemented. We follow the evaluation metrics description presented in subsection 3.1.4 to show our results.

To evaluate our framework, we first show that our RL attacker is actually learning how to follow a path, given a RL algorithm, a state-machine and a reward function. Then, we tune the parameters for all RL algorithms, fixing the path to learn inside the state-machine for Path 2. After computing the best values for parameters, we use them to compare the results obtained with different algorithms, in terms of reward, execution time and commands sent to the network. Finally, we will evaluate our RL algorithms also against multiple paths, to check whether our RL attacker can learn different and more complicated paths, with the best parameters found during the tuning phase.

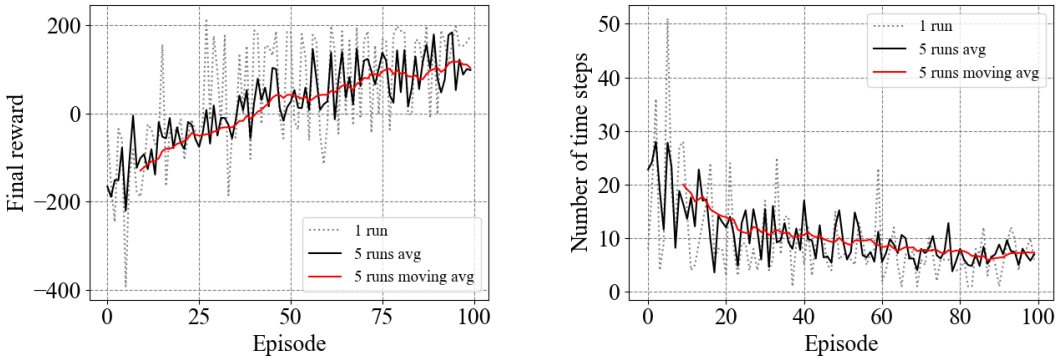
For all experiments, we target a single Yeelight device: a Yeelight LED Smart Bulb 1S Color (8.5W-E27-YLDP13YL). The RL attacker is run on a computer equipped with 4GB of RAM, a CPU Intel i5-4260U with 2 cores of 1.4 GHz. The framework is executed on macOS Mojave 10.14.6 on a 64-bit architecture. Both the Yeelight bulb and the computer were connected to the same Wi-Fi network.

7.1 Example of learning of the RL attacker

Before starting to perform the parameter tuning phase and compare different algorithms, it is necessary to make sure that the RL attacker is actually learning

while interacting with the target IoT device.

We select a single algorithm for our RL agent, the Q-learning algorithm. We first focus on this algorithm because it is one of the most famous and used RL algorithms. For our goal, we decide to focus on the state-machine defined for Path 2, represented in Figure 6.2.



(a) Final reward over episodes

(b) Time steps over episodes

Figure 7.1: Q-learning: performance with state-machine for Path 2 with $\epsilon = 0.6$, $\alpha = 0.05$, $\gamma = 0.95$. **(a)**: $FinalReward$, $FinalReward_{avg}$, $FinalReward_{mov-avg}$; **(b)**: $TimeSteps$, $TimeSteps_{avg}$, $TimeSteps_{mov-avg}$.

We present two plots in Figure 7.1, to be sure that the RL attacker, equipped with the Q-learning algorithm, is learning. In the figure, the plot on the left shows the final reward R_E obtained at each episode E for all episodes. The plot on the right instead shows the number of time steps N_{t_E} at each episode E. The dotted grey line in the figures represent the reward R_E and the number of time steps N_{t_E} for a single execution of the Q-learning algorithm with the following values for parameters: $\epsilon = 0.6$, $\alpha = 0.05$, $\gamma = 0.95$. The algorithm is executed for $N_E = 100$ episodes and we put a maximum value of 100 to the number of time steps inside a single episode. The ϵ decay process has a decay value $D = 0.001$, performed after $E_{decay} = 20$ episodes.

To present these first results, we chose values for all parameters based on theory of RL [20] and on literature works, as [24]. Moreover, the time to perform 100 episodes is of about 40 minutes, hence for this first example we do not want to try with more episodes. This duration for the learning process is due to the sleep time introduced between the commands sent to the device, necessary to avoid to exceed the maximum rate defined by the Yeelight protocol. The sleep time is also performed before retrieving the current state of the bulb, in order to wait for the correct execution of the sent commands on the device. In fact, the execution of

commands on a lamp is not instantaneous.

Returning to the figure, the grey lines, which correspond to the $FinalReward$ and $TimeSteps$ metrics, show an increasing trend over episodes, but they constantly oscillate from the beginning to the end of the learning process. This noise is due to the presence of exploration: we used $\epsilon = 0.6$, which means that about 60% of the actions are taken randomly by the RL agent. This leads to multiple episodes in which the final reward is negative and no positive reward is gained by the agent during the time steps.

To obtain clearer results, we run the Q-learning algorithms for 5 times with the same identical configuration of parameters we just described. The black lines refer to the average of final reward R_E and time steps N_{t_E} for multiple runs, following the formula for $FinalReward_{avg}$ and $TimeSteps_{avg}$ in subsection 3.1.4.

Since also the black lines contain a lot of noise, it would be difficult to make a comparison among different averages for different configurations of the RL agent. Therefore, we also computed the moving average over a window size of $w = 10$ for both these graphs, as indicated by the previously defined $FinalReward_{mov-avg}$ and $TimeSteps_{mov-avg}$ metrics. The moving average is shown into a red line, and it starts from $E = 10$ because it is computed over the previous 10 episodes.

Looking at the left graph, the red line clearly shows that our RL agent is learning to follow the optimal path inside the state-machine for Path 2, starting from having a moved average reward of about -100, to a moved average reward of $+100$ after 70 episodes. The moved average reward becomes positive after about 40 episodes, when the moved average number of steps starts to decrease below 10 time steps. Also, if we look at the graph on the right, the number of time steps needed by the RL agent to reach the terminal state has a decreasing trend, meaning that the agent learns how to reach the terminal state as fast as possible, starting from a moved average of 20 at episode 10, reaching a moved average value of about 7 time steps. Note that the optimal path in Path 2 is 4 steps long, but our average at the end of the learning process is about 7 time steps long: this happens because of the presence of exploration in RL algorithms, and in particular of this configuration of parameters in which $\epsilon = 0.6$, as explained before.

Takeaway. Since the trend of the left graph is increasing, and the trend of the right one is decreasing, we can state that our RL attacker with Q-learning is actually learning how to maximize the reward inside the state-machine, slowly getting close to the optimal values. Our RL agent is learning good actions after about 70 episodes.

7.2 Parameter tuning

Since in the previous section we showed that our RL agent after about 70 episodes is learning good actions and maximizing its final reward, we can now test different RL algorithms. For a fair comparison among them, we first perform a parameter tuning phase, searching for the best values of parameters for a fixed state-machine. From now on, we compare different configurations of parameters and different algorithms using the moving average values: $FinalReward_{mov-avg}$ and $TimeSteps_{mov-avg}$.

For this phase, we focus on Path 2, showed in Figure 6.2. We chose this one because it is not trivial as Path 1, but it is not as much complex as Path 2. In fact, it relies on a 4 steps optimal path, considering 4 attributes for defining the state of a device. The optimal path in this state-machine gains a reward slightly higher than 200, which is based on the path followed by the RL agent before reaching the terminal state.

To perform the tuning phase, the best choice would be to tune the parameters for all algorithms performing a grid search. The problem is that, as said for the Q-learning execution already presented, the time to perform 100 episodes is approximately of 40 minutes. Since we have 4 parameters to tune - ϵ, α, γ and λ - to test only two values per parameter, for all algorithms and executing multiple runs, the time for this tuning phase would exponentially grow, being of about one week[1].

Hence, we tune each parameter once at a time, acting greedily. This solution allows for trying multiple values for each parameters and execute multiple times each configuration to obtain some statistically valuable results.

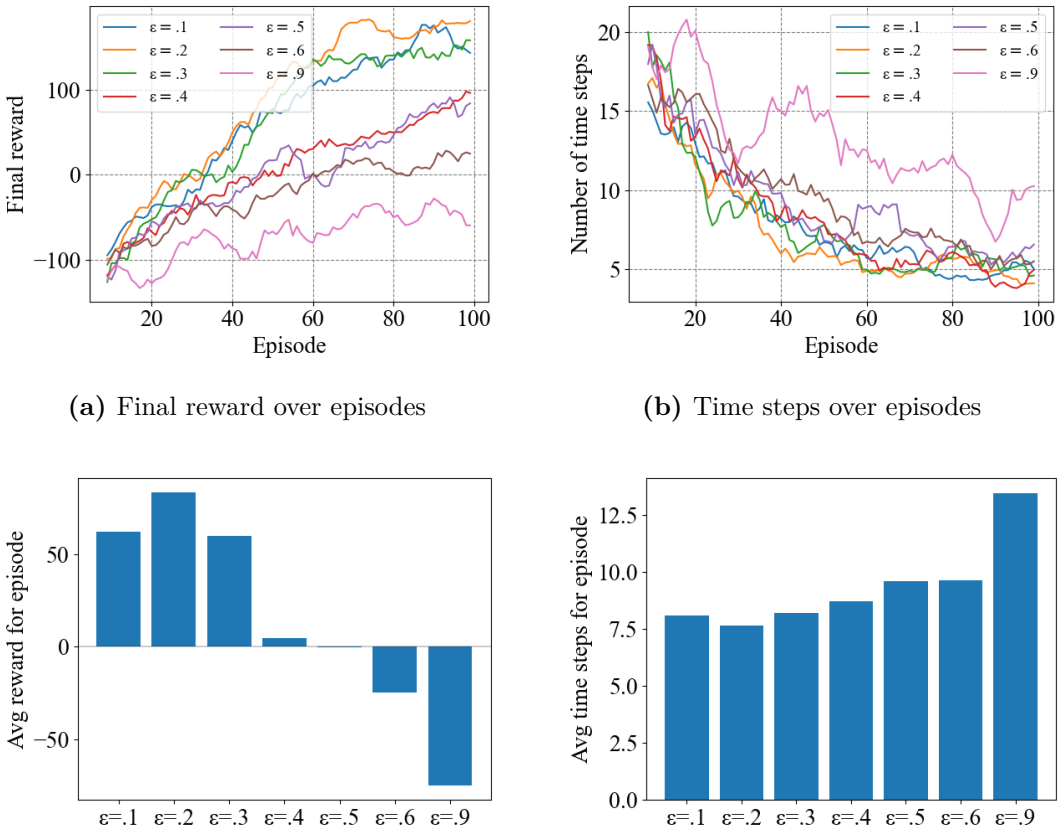
Our procedure followed for the tuning process is the following one. Starting from the values used in the previous section of $\epsilon = 0.6$, $\alpha = 0.05$ and $\gamma = 0.95$, we first fix α and γ values and tune ϵ, both for SARSA and Q-learning algorithms. Then, we select the best ϵ values, and repeat the same procedure for α. Again, selecting and fixing the best α value, we tune the γ parameter for both algorithms. At this point we obtained the best values for all parameters for SARSA and Q-learning. Since TD learning can seen as TD(λ) learning for $\lambda = 0$, we also use the values just found as the best ones also for SARSA(λ) and Q(λ). Then, we perform the tuning process for these TD(λ) algorithms for the λ parameter. Each combination

[1]This estimation is made in the following way. Assuming we want to try 2 values for ϵ, α and γ for SARSA and Q-learning and 2 values for ϵ, α, γ and λ for SARSA(λ) and Q(λ), this results in 48 possible combinations. Then, to have some reliable results, we run 5 times each combination, to be able to compute averages among results, obtaining a total of 240 runs. If we perform the learning process for 100 episodes, we would need about 40 minutes for execution, more or less depending on the performance of the specific algorithm. This ends in 9600 minutes - about a week - which is an unfeasible amount of time for the scope of our work.

is executed 5 times to be able to compute averaged results, for 100 episodes each. In this way, we obtain best values for all 4 implemented algorithms, testing 7 different values for ϵ and α and 5 different values for γ and λ, in a feasible amount of time.

Summing up, we test our algorithms with $\epsilon \in (0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.9)$, $\alpha \in (0.5, 0.2, 0.1, 0.05, 0.02, 0.01, 0.005)$, $\gamma \in (0.35, 0.55, 0.75, 0.9, 0.95)$ and $\lambda \in (0, 0.5, 0.8, 0.9, 0.95)$. For λ we also use also the 0 value to compare also the performance of TD and TD(λ) learning techniques.

Note that for all executions we made in this chapter, the ϵ decay process always has a decay value $D = 0.001$, performed after $E_{decay} = 20$ episodes, according to what we explained into subsection 5.5.2.



(a) Final reward over episodes

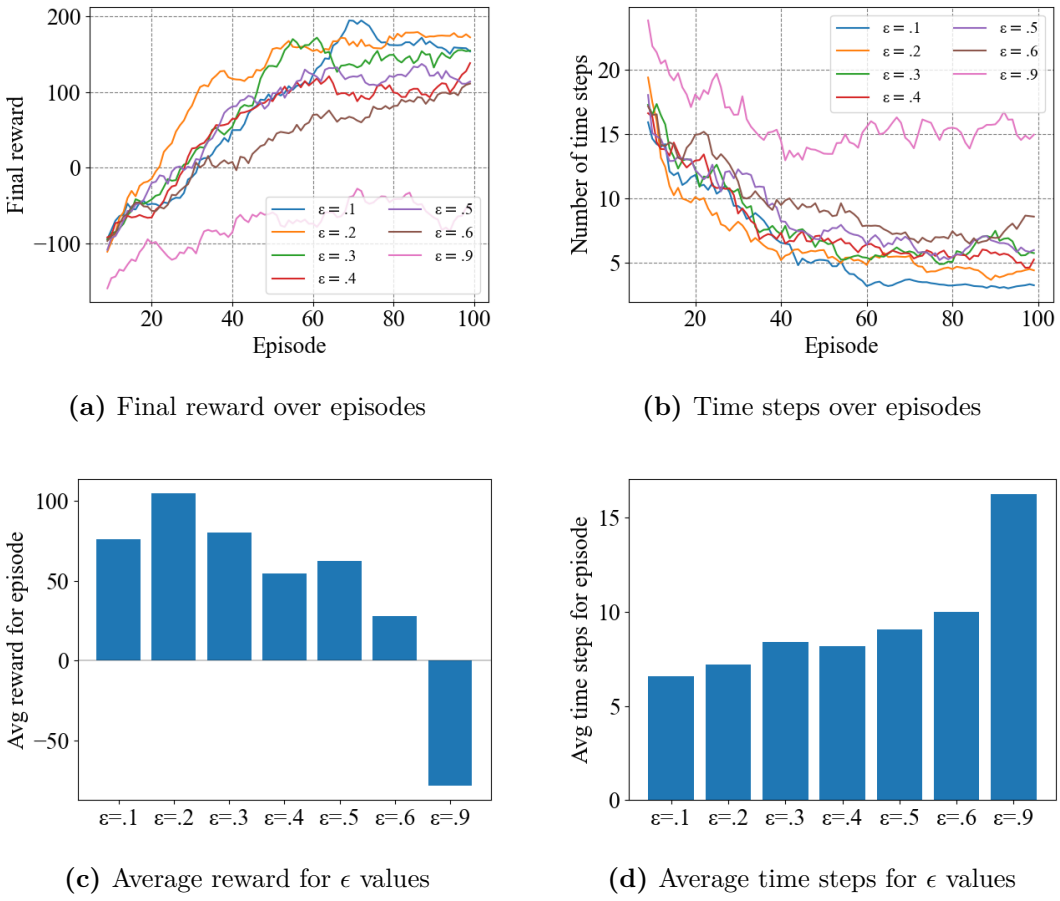
(b) Time steps over episodes

(c) Average reward for ϵ values

(d) Average time steps for ϵ values

Figure 7.2: SARSA performance: tuning of ϵ with state-machine for Path 2 with $\alpha = 0.05$, $\gamma = 0.95$. **(a)**: $FinalReward_{mov-avg}$; **(b)**: $TimeSteps_{mov-avg}$; **(c)**: AVG_{rew}; **(d)**: AVG_{steps}.

In Figure 7.2 and Figure 7.3 we present the results obtained for the tuning of ϵ,



(a) Final reward over episodes

(b) Time steps over episodes

(c) Average reward for ϵ values

(d) Average time steps for ϵ values

Figure 7.3: Q-learning performance: tuning of ϵ with state-machine for Path 2 with $\alpha = 0.05$, $\gamma = 0.95$. **(a)**: $FinalReward_{mov-avg}$; **(b)**: $TimeSteps_{mov-avg}$; **(c)**: AVG_{rew}; **(d)**: AVG_{steps}.

fixing $\alpha = 0.05$ and $\gamma = 0.95$, respectively for SARSA and Q-learning algorithms. For each result, four graphs are shown. Graphs **(a)** and **(b)** compare the moving averages of the final reward over episodes and the number of time steps over episodes for multiple values of ϵ. These moving average values are computed on the average of the values over 5 runs, as described for metrics $FinalReward_{mov-avg}$ and $TimeSteps_{mov-avg}$. The bottom graphs **(c)** and **(d)** represent the average reward AVG_{rew} and the average number of time steps AVG_{steps}, for multiple ϵ. Recall that the first is computed as the average over all episodes of the final reward R_E obtained for each episode E, averaged over 5 runs, whereas the second is computed as the average over all episodes of the number of time steps N_{t_E} of each episode E, also averaged over 5 runs.

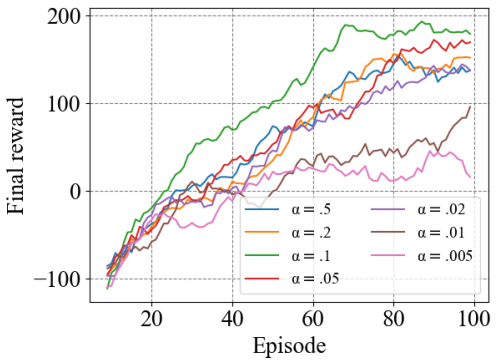
Looking at Figure 7.2a, we can see that the results achieved by different ϵ are really different. For high values, such as $\epsilon = 0.9$, the final reward is negative for each episode, and it is the worst configuration by the reward and time steps points of view. This is predictable, since with such a high value exploration is performed 90% of the times, thus the percentage of exploitation is too low for this kind of RL problem. Looking at the results both for SARSA in Figure 7.2c and Q-learning in Figure 7.3c, $\epsilon = 0.2$ is the optimal value among the selected ones, obtaining the maximum reward. For SARSA, it is also the one that reaches on average the smallest number of time steps per episode. For Q-learning, it reaches the second smallest number, after $\epsilon = 0.1$. This could mean that with $\epsilon = 0.1$ Q-learning learned the fastest path, but it is not the one with the highest final reward, that is the optimal path. We can also note that there exists a high variance between different ϵ configurations. For SARSA in Figure 7.2, in **(a)** curves are quite well separated from episode 30 to 100, and also in **(c)** the average rewards have very different values, from about -75 for $\epsilon = 0.9$ to about $+75$ for $\epsilon = 0.2$. This is analogous to Q-learning results of Figure 7.3, which however achieves highest moving averaged final rewards, according to **(a)**. Moreover, we can say that on average Q-learning reaches highest rewards in less time steps than SARSA, for almost all values of ϵ, except for $\epsilon = 0.9$.

From these initial results, we select $\epsilon = 0.2$ for the next tuning steps both for SARSA and Q-learning.

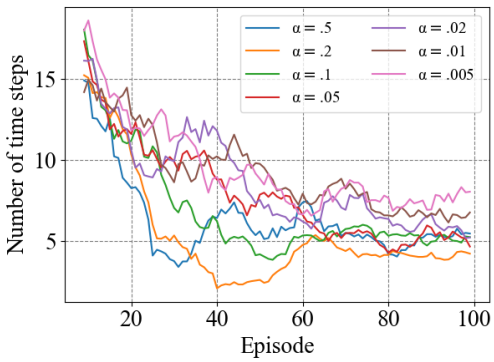
Now, we use $\epsilon = 0.2$ and $\gamma = 0.95$ for tuning α. In Figure 7.4 and Figure 7.5, results for SARSA and Q-learning are presented. The graphs are the same we showed for the tuning of the ϵ parameter.

For SARSA, we can note that all moved average rewards after 50 episodes are positive, reaching the highest average reward for episodes for $\alpha = 0.1$, obtaining the smallest average of time steps for $\alpha = 0.2$ instead. However, the average number of time steps for $\alpha = 0.5$, $\alpha = 0.2$ and $\alpha = 0.1$ is contained between 5 and 8 time steps. Nevertheless, focusing on Figure 7.4c, the configuration with $\alpha = 0.1$ reaches an average reward $AVG_{rew} > 90$ much larger than the other results, whereas all other AVG_{rew} values are below 60. Note also that different configurations of α reach a decreased variance between the final reward and the number of time steps obtained, with the respect to SARSA results for ϵ tuning. Analogously, this happens also for Q-learning, comparing the tuning results for α with respect to results for the ϵ parameter. Moreover, also for Q-learning, making similar considerations, $\alpha = 0.1$ reaches the best moved average and average reward AVG_{rew} with respect to all other values. Therefore, we select $\alpha = 0.1$ as the best value both for SARSA and Q-learning algorithms.

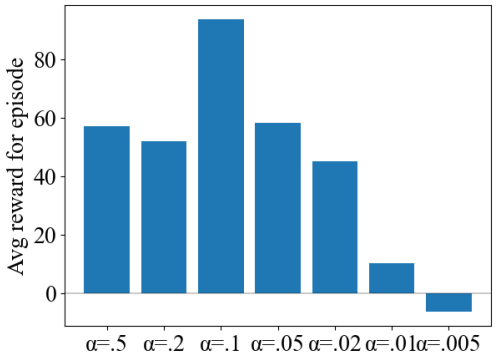
At this point we focus on tuning γ. In Figure 7.6 and Figure 7.7 we present the results for different values of γ, through the same graphs we used before for tuning ϵ and α.



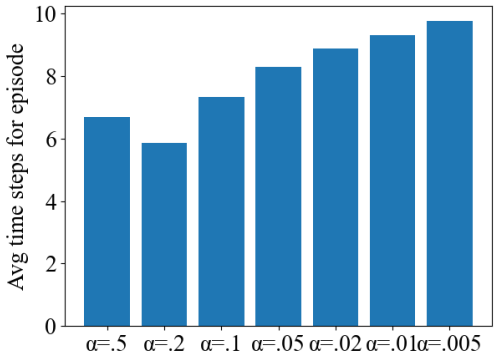
(a) Final reward over episodes



(b) Time steps over episodes



(c) Average reward for α values

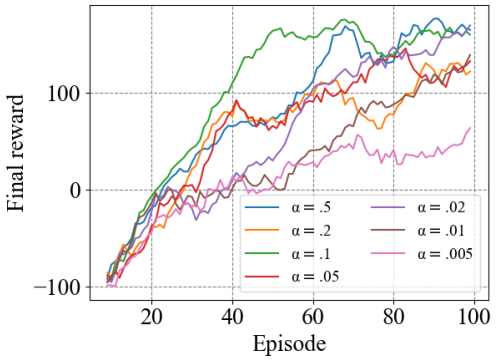


(d) Average time steps for α values

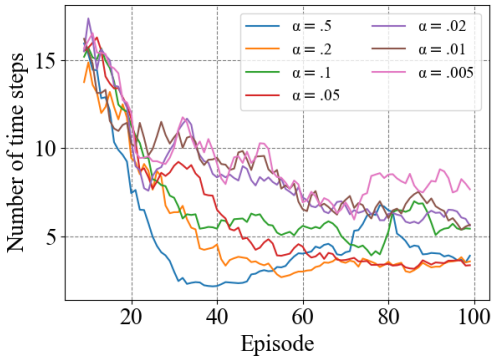
Figure 7.4: SARSA performance: tuning of α with state-machine for Path 2 with $\epsilon = 0.2$, $\gamma = 0.95$. **(a)**: $FinalReward_{mov-avg}$; **(b)**: $TimeSteps_{mov-avg}$; **(c)**: AVG_{rew}; **(d)**: AVG_{steps}.

Looking at these graphs, it is clear to understand why we add to the results of the tuning process **(a)** and **(b)** also graphs **(c)** AVG_{rew} and **(d)** AVG_{steps}. After having found best ϵ and α values, the first two graphs do not show clearly which parameter value gains the highest reward, hence we must rely on results showed by **(c)** and **(d)**.

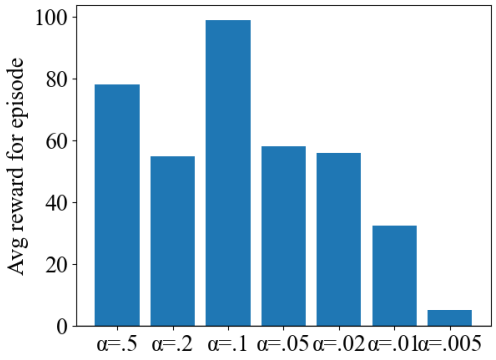
Following the considerations we made before, we select $\gamma = 0.75$ for SARSA as the best value, following results in Figure 7.6c. For Q-learning, instead, the distinction among different γ values is not so clear, since curves are superimposed. Indeed, from Figure 7.7c and Figure 7.7d we note that the average number of time steps is between 6 and 7 for all γ values, while all average rewards are between 80 and 100. From these results, we can say that after performing the tuning of ϵ and



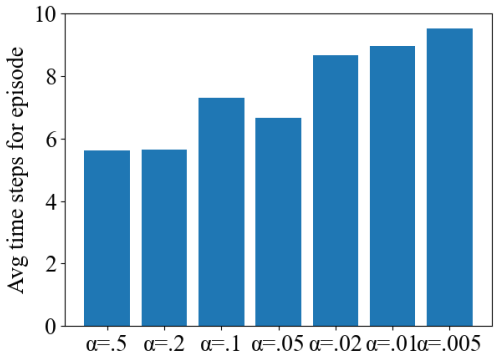
(a) Final reward over episodes



(b) Time steps over episodes



(c) Average reward for α values



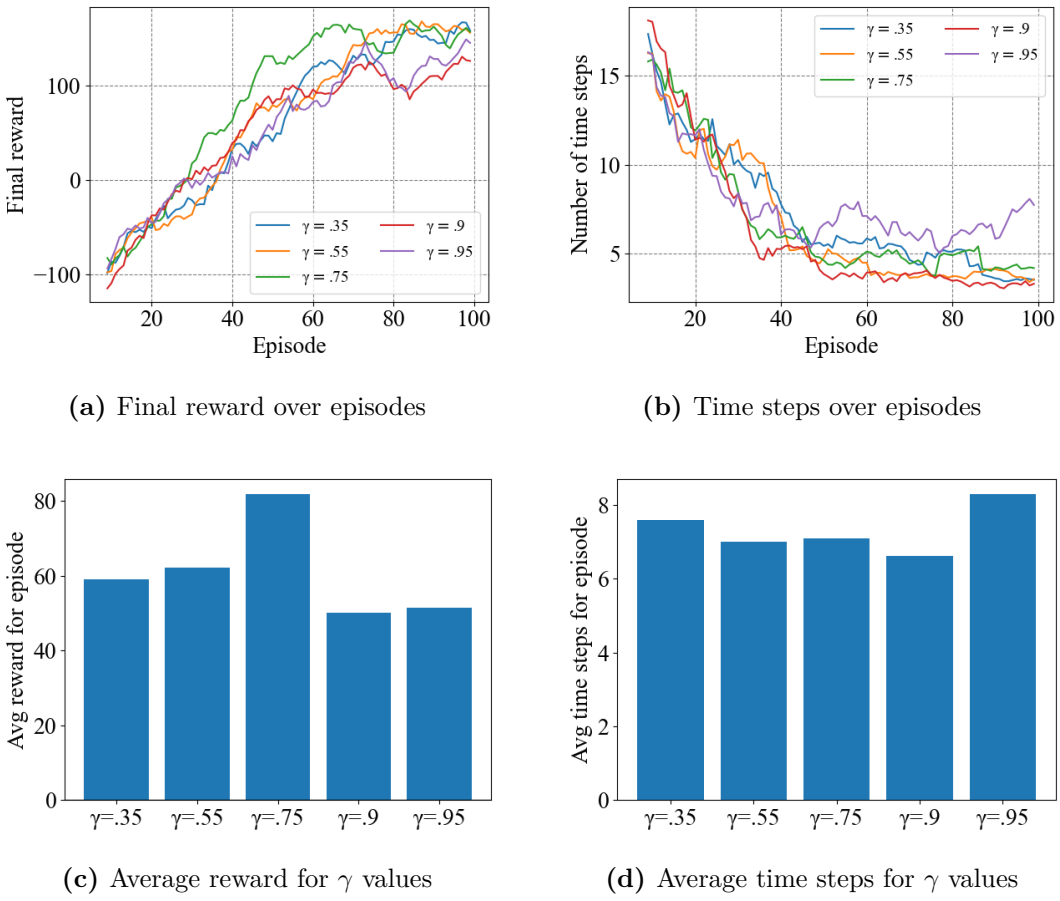
(d) Average time steps for α values

Figure 7.5: Q-learning performance: tuning of α with state-machine for Path 2 with $\epsilon = 0.2$, $\gamma = 0.95$. **(a)**: $FinalReward_{mov-avg}$; **(b)**: $TimeSteps_{mov-avg}$; **(c)**: AVG_{rew}; **(d)**: AVG_{steps}.

α, the value of γ becomes less influential with respect to the outcomes.

Considering results both for SARSA and Q-learning, we obtained respectively $\gamma = 0.75$ and $\gamma = 0.55$ as best values. These results are different from the suggested values for γ. Typically, γ should be set near to 1 in order to give more credit to future rewards, as being the discount factor. Actually in our case, because of the fact that the optimal paths and most of the paths are not long, between 4 and 8 time steps typically, having a high value of γ is less significant. In cases in which state-machines could contain much longer paths - e.g. of 100 steps - then having a γ value near to 1 fundamental for the learning process.

Thus, we computed the tuning for all parameters, obtaining $\epsilon = 0.2$, $\alpha = 0.1$ and $\gamma = 0.75$ for SARSA and $\epsilon = 0.2$, $\alpha = 0.1$ and $\gamma = 0.55$ for Q-learning. In the next



(a) Final reward over episodes

(b) Time steps over episodes

(c) Average reward for γ values

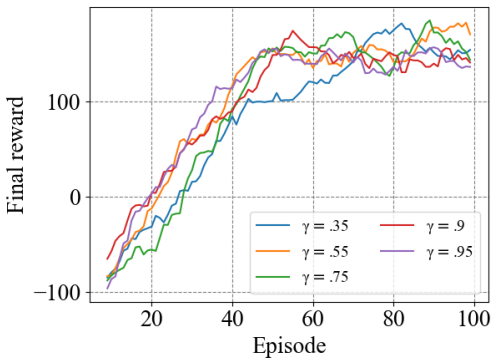
(d) Average time steps for γ values

Figure 7.6: SARSA performance: tuning of γ with state-machine for Path 2 with $\epsilon = 0.2$, $\alpha = 0.1$. **(a)**: $FinalReward_{mov-avg}$; **(b)**: $TimeSteps_{mov-avg}$; **(c)**: AVG_{rew}; **(d)**: AVG_{steps}.

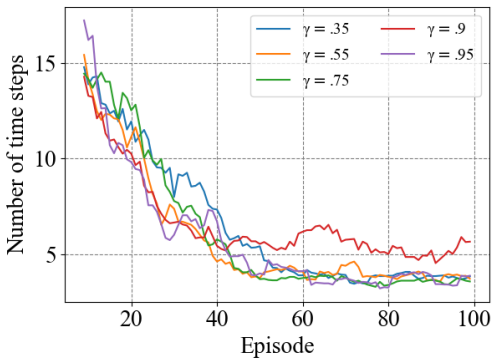
sections, these parameters will be used for these algorithms to make comparisons of the performance of multiple algorithms.

Until now we tuned parameters for TD learning algorithms, excluding SARSA(λ) and Q(λ). Because of the fact that these two algorithms can be shown as the extended versions of SARSA and Q-learning, we use as best values the already tuned values of TD learning. In TD(λ) learning, the only parameter left to tune is λ. We tune $\lambda \in (0, 0.5, 0.8, 0.9, 0.95)$, adding the $\lambda = 0$ just for a comparison with TD learning.

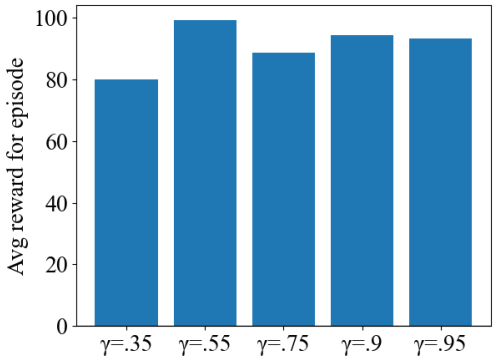
In Figure 7.8, the results obtained for SARSA(λ) are presented. Looking at Figure 7.8a and 7.8b it is not clear what is the best value to choose for λ. Actually, looking at the first 20 episodes $\lambda = 0.5$ seems to perform better, while after episode



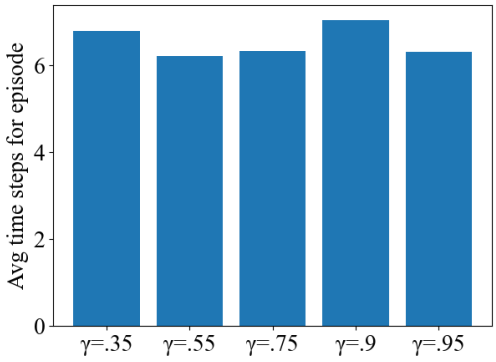
(a) Final reward over episodes



(b) Time steps over episodes



(c) Average reward for γ values

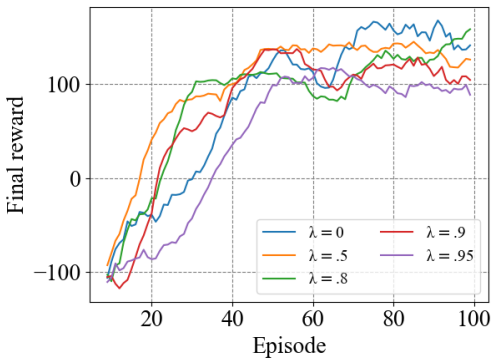


(d) Average time steps for γ values

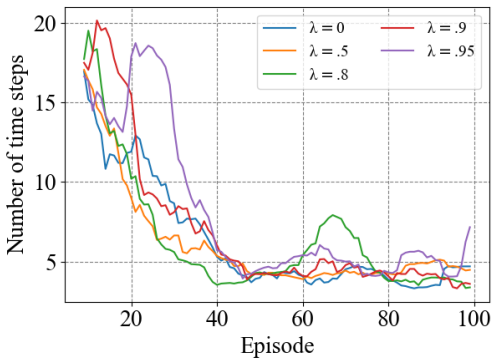
Figure 7.7: Q-learning performance: tuning of γ with state-machine for Path 2 with $\epsilon = 0.2$, $\alpha = 0.1$. **(a)**: $FinalReward_{mov-avg}$; **(b)**: $TimeSteps_{mov-avg}$; **(c)**: AVG_{rew}; **(d)**: AVG_{steps}.

$E = 70$, the plain SARSA with $\lambda = 0$ has a greater moving average reward. Looking at graphs **(c)** and **(d)**, it is clear that $\lambda = 0.5$ obtained the highest average reward value for episode, and also the smallest average number of time steps for episode. However, looking at **(d)**, the number of time steps for all λ values except for $\lambda = 0.95$ is about 7 steps. Therefore, we select $\lambda = 0.5$ for SARSA(λ). Note that λ is the trace decay parameter, and literature suggests to use λ values near to 1, in order to update values inside the value function for past states with respect to the current one, updating them with rewards received much later in the future. As for γ, higher values of λ are preferable in state-machines allowing very long paths.

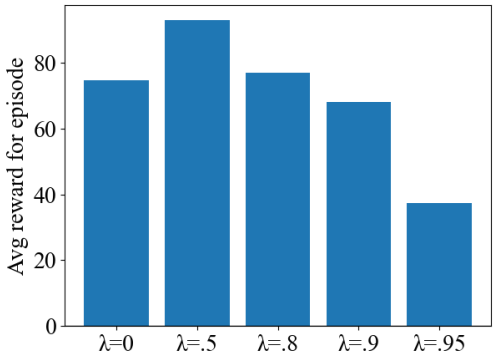
Finally, in Figure 7.9, results for Q(λ) are shown. With this algorithm, we note a clear distinction between the moved average and average rewards obtained for



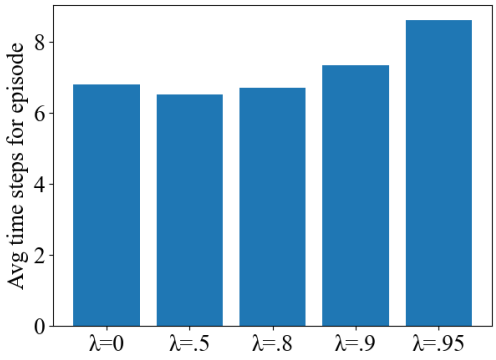
(a) Final reward over episodes



(b) Time steps over episodes



(c) Average reward for λ values

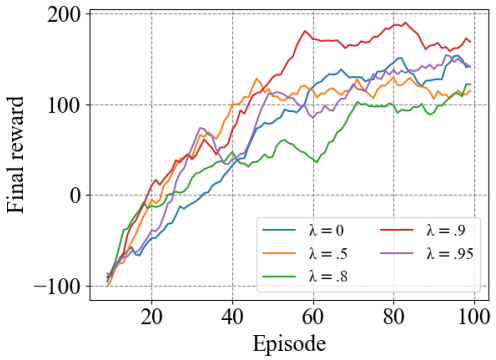


(d) Average time steps for λ values

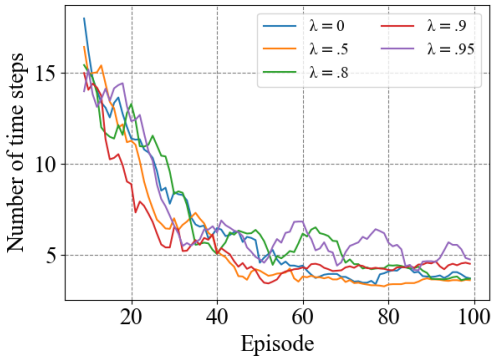
Figure 7.8: SARSA(λ) performance: tuning of λ with state-machine for Path 2 with $\epsilon = 0.2$, $\alpha = 0.1$, $\gamma = 0.75$. **(a)**: $FinalReward_{mov-avg}$; **(b)**: $TimeSteps_{mov-avg}$; **(c)**: AVG_{rew}; **(d)**: AVG_{steps}.

$\lambda = 0.9$ with respect to the other values, both from graphs in Figure 7.9a and in Figure 7.9c. Hence, we select $\lambda = 0.9$ as the best λ for Q(λ).

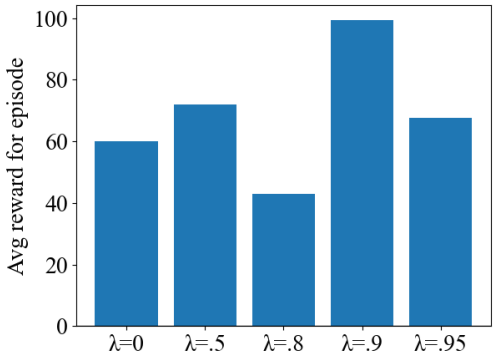
Takeaway. From our parameter tuning phase focused on Path 2, optimal values for all algorithms are $\epsilon = 0.2$ and $\alpha = 0.1$. We select $\gamma = 0.75$ for SARSA and SARSA(λ) algorithms, while we select $\gamma = 0.55$ for Q-learning and Q(λ) algorithms. For TD(λ) algorithms, best ones turn out to be $\lambda = 0.5$ for SARSA(λ) and $\lambda = 0.9$ for Q(λ). From our results, ϵ and α are the parameters which most affect the rewards obtained by the RL algorithms.



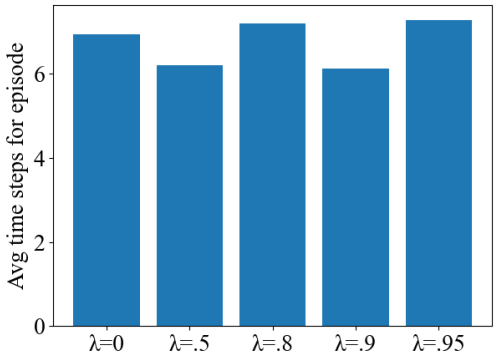
(a) Final reward over episodes



(b) Time steps over episodes



(c) Average reward for λ values



(d) Average time steps for λ values

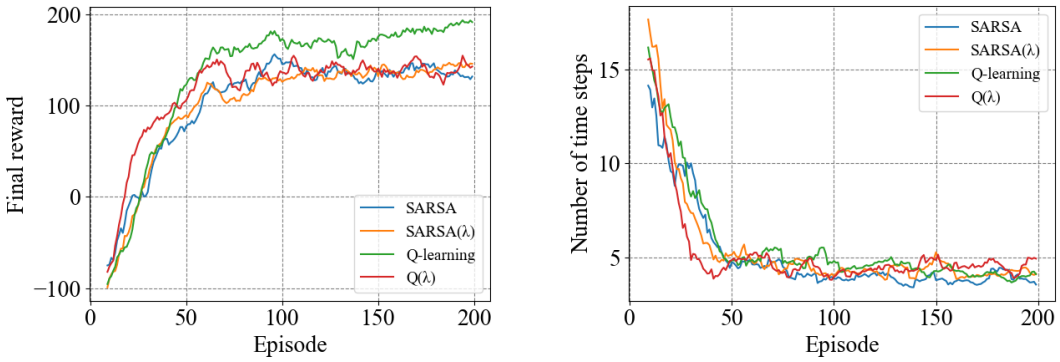
Figure 7.9: Q(λ) performance: tuning of γ with state-machine for Path 2 with $\epsilon = 0.2$, $\alpha = 0.1$, $\gamma = 0.55$. **(a)**: $FinalReward_{mov-avg}$; **(b)**: $TimeSteps_{mov-avg}$; **(c)**: AVG_{rew}; **(d)**: AVG_{steps}.

7.3 Comparison between algorithms for Path 2

After the tuning process of parameters, we fix values for ϵ, α, γ and λ for all algorithms. For a fair comparison, we can compare all four implemented algorithms against the same goal, exploiting state-machine for Path 2, which is the one we used to tune all parameters. Recall the values we will use for all algorithms:

- *SARSA*: $\epsilon = 0.2$, $\alpha = 0.1$, $\gamma = 0.75$;
- *Q-learning*: $\epsilon = 0.2$, $\alpha = 0.1$, $\gamma = 0.55$;
- *SARSA(λ)*: $\epsilon = 0.2$, $\alpha = 0.1$, $\gamma = 0.75$, $\lambda = 0.5$;

- $Q(\lambda)$: $\epsilon = 0.2$, $\alpha = 0.1$, $\gamma = 0.55$, $\lambda = 0.9$.



(a) Final reward over episodes

(b) Time steps over episodes

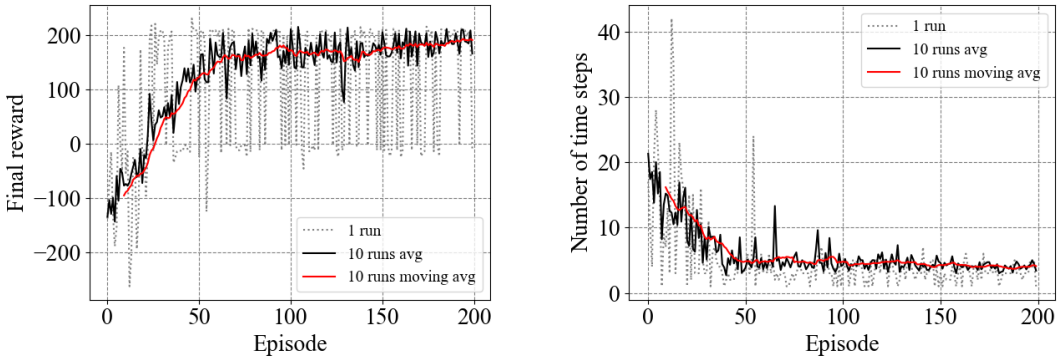
Figure 7.10: All algorithms: performance with state-machine for Path 2. SARSA: $\epsilon = 0.2$, $\alpha = 0.1$, $\gamma = 0.75$; Q-learning: $\epsilon = 0.2$, $\alpha = 0.1$, $\gamma = 0.55$; SARSA(λ): $\epsilon = 0.2$, $\alpha = 0.1$, $\gamma = 0.75$, $\lambda = 0.5$; Q(λ): $\epsilon = 0.2$, $\alpha = 0.1$, $\gamma = 0.55$, $\lambda = 0.9$. **(a)**: $FinalReward_{mov-avg}$; **(b)**: $TimeSteps_{mov-avg}$.

The results obtained for all algorithms are presented in Figure 7.10. For these results, we could afford more time, deciding to run our algorithms for 200 episodes instead of 100, and each algorithm has been executed 10 times instead of 5, hence $N_E = 200$ and $N_{runs} = 10$. These graphs show the moving average of the final reward over episodes and the moving average of the number of time steps per episode: $FinalReward_{mov-avg}$ and $TimeSteps_{mov-avg}$ computed for $N_{runs} = 10$.

In both graphs, all curves are close to each other. The curves for the number of time steps in Figure 7.10b are all similar, reaching a moving average value between 4 and 5 time steps. The Q(λ) is the one curve which seems to learn faster, since it reaches 5 time steps after about 30 episodes, while the remaining algorithms reach 5 time steps around the 50th episode.

Looking at Figure 7.10a, Q(λ) has the highest rewards for the first 50 episodes, while from episode 50 Q-learning is actually reaching the maximum moving average final reward, presented by the green line in the figure.

We now focus on Q-learning. It is the only algorithm reaching on average a value close to 200. To understand what is the trend of this algorithm, we present the same graphs as the one presented in section 7.1, showing the results for 1 run, 5 runs and the moving average already presented in the previous figure. Q-learning performance for Path 2 are shown in Figure 7.11. From the grey dotted line we can note a high oscillation on the reward for the reward of a single run. Starting from episode $E = 50$, the moving average of the final reward is always greater than



(a) Final reward over episodes

(b) Time steps over episodes

Figure 7.11: Q-learning: performance with state-machine for Path 2 with $\epsilon = 0.2$, $\alpha = 0.1$, $\gamma = 0.55$. **(a)**: $FinalReward$, $Graph_{avg}$, $Graph_{mov-avg}$; **(b)**: $TimeSteps$, $TimeSteps_{avg}$, $TimeSteps_{mov-avg}$.

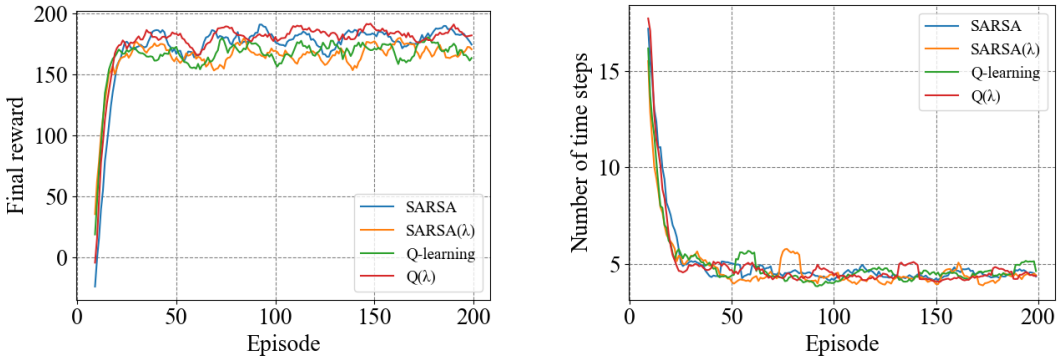
100, approaching to 200 in all subsequent episodes and almost reaching it at the end of the episodes. Instead, in the Q-learning example in Figure 7.1 performed without tuned values, the moving average final reward reaches 100 around episode 75, sticking to this value until the end of the learning process. Therefore, we can say that after the parameter tuning process we improved our results for the Q-learning algorithm[2].

Takeaway. For the state-machine for Path 2, Q(λ) has the highest rewards for the first 50 episodes, while from episode 50 Q-learning is actually reaching the maximum moving average final reward, overcoming the performance of Q(λ). Focusing on Q-learning, before the parameter tuning process this algorithm reached a moving average final reward of 100 around episode 75, sticking to this value until the end of the learning process. With optimized parameters, instead, it overcomes a value of 100 at episode 50, keeping to improve this value until the end of the learning process, and reaching a moved average value of about 175 at episode 100.

[2]This improvement could be seen also for all four algorithms. We are not showing graphs for all algorithms because here we selected only most significant graphs to present.

7.4 Comparison between algorithms for multiple Paths

In this section we compare all four algorithms against different paths. As in section 6.3 we presented three different state-machines for the Yeelight protocol, we test RL algorithms for all of these. In the previous section we already presented $FinalReward_{mov-avg}$ and $TimeSteps_{mov-avg}$ for the state-machine of Path 2.



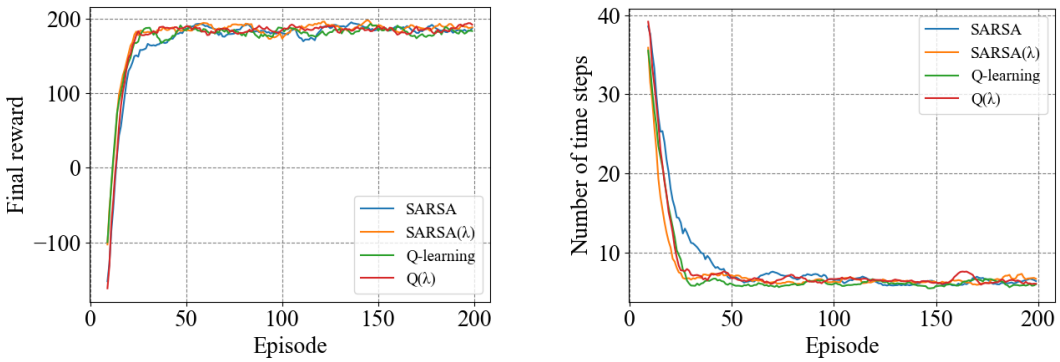
(a) Final reward over episodes

(b) Time steps over episodes

Figure 7.12: All algorithms: performance with state-machine for Path 1. SARSA: $\epsilon = 0.2$, $\alpha = 0.1$, $\gamma = 0.75$; Q-learning: $\epsilon = 0.2$, $\alpha = 0.1$, $\gamma = 0.55$; SARSA(λ): $\epsilon = 0.2$, $\alpha = 0.1$, $\gamma = 0.75$, $\lambda = 0.5$; Q(λ): $\epsilon = 0.2$, $\alpha = 0.1$, $\gamma = 0.55$, $\lambda = 0.9$. **(a)**: $FinalReward_{mov-avg}$; **(b)**: $TimeSteps_{mov-avg}$.

In Figure 7.12, the algorithms, with optimized values, learn Path 1. In the figure, $FinalReward_{mov-avg}$ and $TimeSteps_{mov-avg}$ are shown, for 200 episodes and 10 different runs, following the same strategy of Path 2. Moreover, in Figure 7.13, the same graphs are shown for the state-machine for Path 3. Recall that Path 1 takes into consideration 3 attributes, having a optimal path of 4 steps. Path 2 takes 4 attributes into account, with a optimal path of 4 steps, while Path 3 considers 4 attributes but having an optimal path of 7 steps.

Looking at those results we can see that for Path 1 the moving average reward after about 25 episodes continues to range between 150 and 200 for all algorithms. The same can be said for Path 3. Actually the moving averaged reward for Path 3 is slightly higher than the one obtained for Path 1. For Path 1, red and blue lines, corresponding to Q(λ) and SARSA, seem to perform better than the other two algorithms, since their curves are above the other ones. We can not make the same consideration for Path 3, since curves appear really closed and overlapped among each other. Looking at the number of time steps, in Figure 7.12b and Figure 7.13b



(a) Final reward over episodes

(b) Time steps over episodes

Figure 7.13: All algorithms: performance with state-machine for Path 3. SARSA: $\epsilon = 0.2$, $\alpha = 0.1$, $\gamma = 0.75$; Q-learning: $\epsilon = 0.2$, $\alpha = 0.1$, $\gamma = 0.55$; SARSA(λ): $\epsilon = 0.2$, $\alpha = 0.1$, $\gamma = 0.75$, $\lambda = 0.5$; Q(λ): $\epsilon = 0.2$, $\alpha = 0.1$, $\gamma = 0.55$, $\lambda = 0.9$. **(a)**: $FinalReward_{mov-avg}$; **(b)**: $TimeSteps_{mov-avg}$.

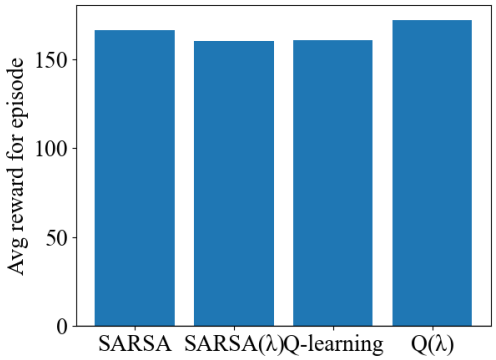
all curves are really close to each other, reaching a minimum of time steps between 4 and 5 starting from episode 50.

Therefore, in Figure 7.14, the final reward AVG_{rew} obtained per episode averaged on $N_{runs} = 10$ runs and on 200 episodes is shown for all paths. For Path 1, the best algorithm is Q(λ). Concerning Path 2, Q-learning is the winner, while for Path 3 SARSA(λ) reaches the highest value. However, for Path 1 and Path 3 all algorithms reach very similar values among each others.

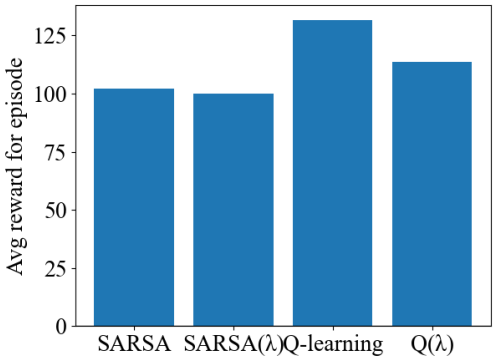
Concerning these executions for different paths, we provide some additional measures.

In Figure 7.15, the Cumulative Distribution Function (CDF) of the average reward Avg_{R_E} obtained per time step is shown, considering all 10 runs for each algorithm. We can note that for each time step, the average obtained reward is higher for Path 1 and Path 2, with respect to Path 3. That is explained by the fact that the optimal path for all state-machines reaches a maximum reward around 200, but for Path 1 and Path 2 the maximum reward is reached performing 4 steps, whereas for Path 3 the same amount of reward is gained in 7 steps: this causes a smaller average reward per step for Path 3.

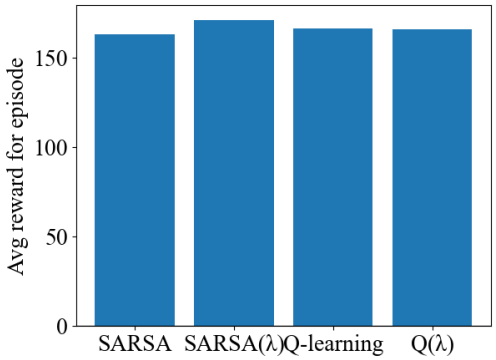
We can now show the measures for the learning time and learning traffic needed by each RL algorithm. Figure 7.16 shows the learning time for all three paths of the state-machines previously defined. We highlight how these times are really high, above $3000s$ for all algorithms and paths, which corresponds to about 50 minutes. Recall that, before tuning the parameters, Q-learning needed 40 minutes



(a) Path 1: average reward for all algorithms



(b) Path 2: average reward for all algorithms

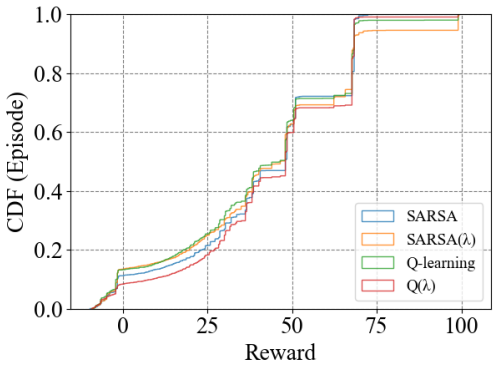


(c) Path 3: average reward for all algorithms

Figure 7.14: All algorithms: average reward value per step AVG_{rew} computed for different state-machines. SARSA: $\epsilon = 0.2$, $\alpha = 0.1$, $\gamma = 0.75$; Q-learning: $\epsilon = 0.2$, $\alpha = 0.1$, $\gamma = 0.55$; SARSA(λ): $\epsilon = 0.2$, $\alpha = 0.1$, $\gamma = 0.75$, $\lambda = 0.5$; Q(λ): $\epsilon = 0.2$, $\alpha = 0.1$, $\gamma = 0.55$, $\lambda = 0.9$. **(a)**: AVG_{rew} for Path 1; **(b)**: AVG_{rew} for Path 2; **(c)**: AVG_{rew} for Path 3.

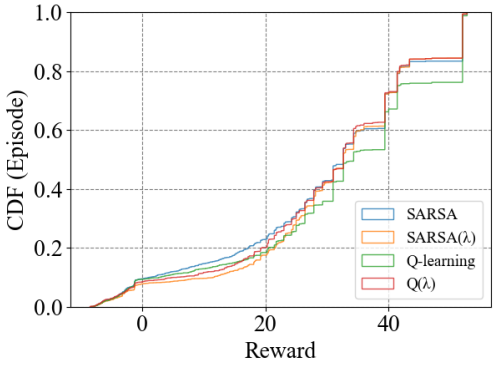
to perform 100 episodes. Now, instead, 50 minutes at least are needed for all algorithms, to execute the learning process for 200 episodes: hence, the learning time is not doubled. In fact, since the learning process learns faster than before, it takes less actions to get to the terminal state, having to perform less sleep time among each action[3].

[3]As introduced previously, the Yeelight protocol puts a threshold on the maximum rate allowed for sending commands to a Yeelight device, of 60 commands/minute.



(a) Path 1: CDF of average reward per episode

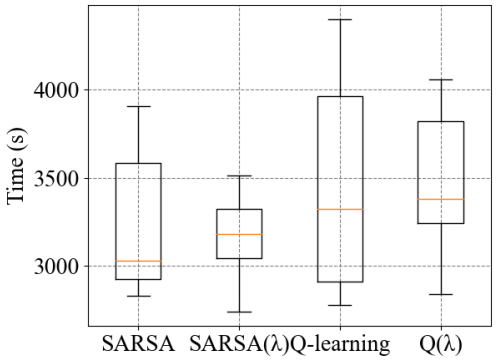
(b) Path 2: CDF of average reward per episode



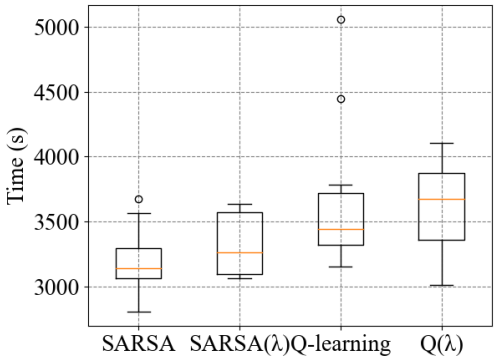
(c) Path 3: CDF of average reward per episode

Figure 7.15: All algorithms: CDF of the average reward per episode for different state-machines. SARSA: $\epsilon = 0.2$, $\alpha = 0.1$, $\gamma = 0.75$; Q-learning: $\epsilon = 0.2$, $\alpha = 0.1$, $\gamma = 0.55$; SARSA(λ): $\epsilon = 0.2$, $\alpha = 0.1$, $\gamma = 0.75$, $\lambda = 0.5$; Q(λ): $\epsilon = 0.2$, $\alpha = 0.1$, $\gamma = 0.55$, $\lambda = 0.9$. **(a)**: $CDFReward$ for Path 1; **(b)**: $CDFReward$ for Path 2; **(c)**: $CDFReward$ for Path 3.

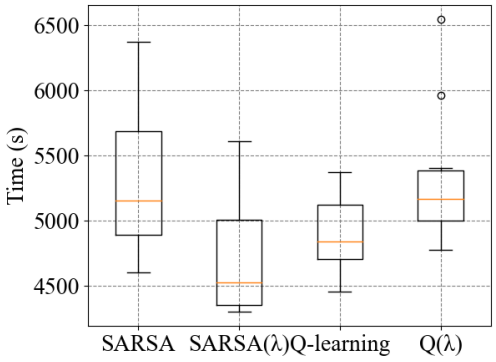
Regarding the learning traffic, Figure 7.17 shows the total number of commands sent by the RL agent to a Yeelight device for the entire learning process. Note that these numbers do not take into account the commands sent by the State Machine module to compute the current state of the agent inside the selected state-machine, since these commands are transparent to the RL agent, which only receives by the environment the information about the new state s_{t+1}. Looking at these graphs, we can see how, depending on the path, more commands are needed. For Path 1, about 1400 commands on average are needed. Path 2, which has the optimal path as long as Path 1 but considers a more extended state-machine, needs on



(a) Path 1: learning time



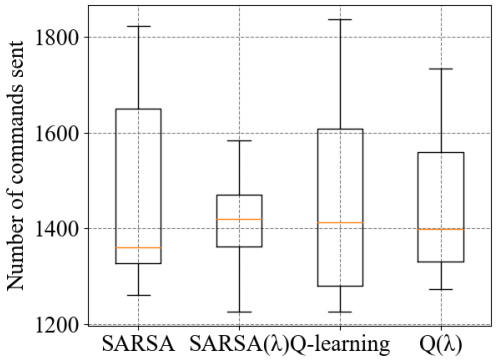
(b) Path 2: learning time



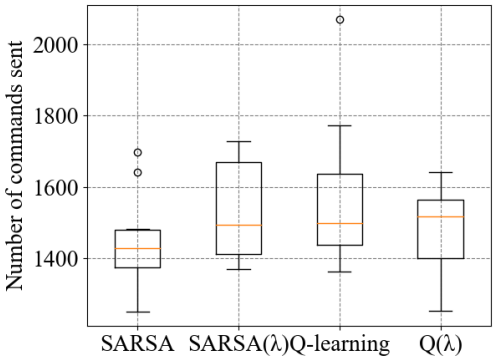
(c) Path 3: learning time

Figure 7.16: All algorithms: learning time for different state-machines. SARSA: $\epsilon = 0.2$, $\alpha = 0.1$, $\gamma = 0.75$; Q-learning: $\epsilon = 0.2$, $\alpha = 0.1$, $\gamma = 0.55$; SARSA(λ): $\epsilon = 0.2$, $\alpha = 0.1$, $\gamma = 0.75$, $\lambda = 0.5$; Q(λ): $\epsilon = 0.2$, $\alpha = 0.1$, $\gamma = 0.55$, $\lambda = 0.9$. **(a)**: *LearningTime* for Path 1; **(b)**: *LearningTime* for Path 2; **(c)**: *LearningTime* for Path 3.

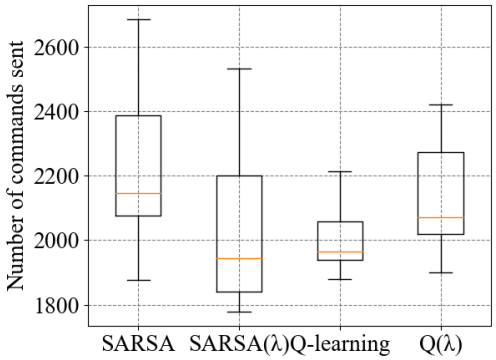
average 1500 commands for each algorithm. Finally, Path 3, which has a much more complex state-machine and a longer path than Path 1 and Path 2, needs on average more than 2000 commands to complete the entire learning process. Therefore, we can infer that the number of commands necessary to complete a learning process, giving optimal parameters and a fixed number of episodes, highly depends on the complexity of the designed state-machine and on the length of the optimal path to be learned. Note that the graphs in Figure 7.16 and in Figure 7.17 are strictly correlated, since, the number of commands affects directly the learning time.



(a) Path 1: learning traffic



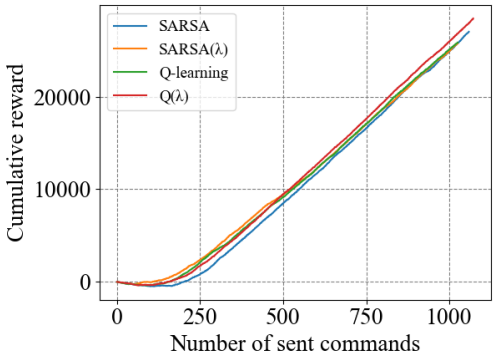
(b) Path 2: learning traffic



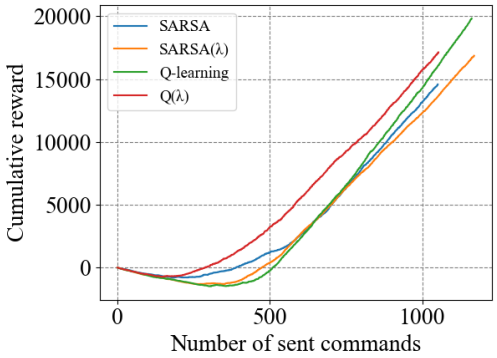
(c) Path 3: learning traffic

Figure 7.17: All algorithms: learning traffic for different state-machines. SARSA: $\epsilon = 0.2$, $\alpha = 0.1$, $\gamma = 0.75$; Q-learning: $\epsilon = 0.2$, $\alpha = 0.1$, $\gamma = 0.55$; SARSA(λ): $\epsilon = 0.2$, $\alpha = 0.1$, $\gamma = 0.75$, $\lambda = 0.5$; Q(λ): $\epsilon = 0.2$, $\alpha = 0.1$, $\gamma = 0.55$, $\lambda = 0.9$. **(a)**: $LearningTraffic$ for Path 1; **(b)**: $LearningTraffic$ for Path 2; **(c)**: $LearningTraffic$ for Path 3.

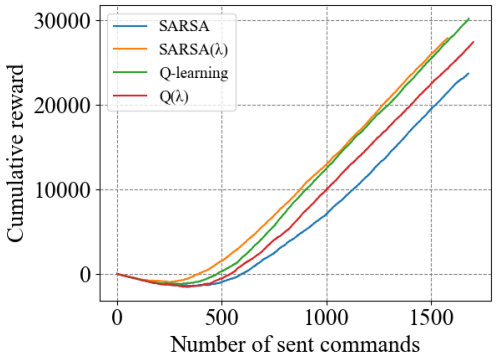
Focusing on the commands sent by the RL agent, we can plot the cumulative reward over the number of commands, as described in subsection 3.1.4. Figure 7.18 shows the trend of the cumulative reward over the number of actions performed - or commands sent - by the RL agent. Whereas the number of episodes is fixed, the total number of actions depends on each episode and on each different run, therefore we developed a way to plot in a meaningful way these results. Thus, these values are computed as described by the metric $RewardOnActions_{avg}$, computing the average of the cumulative reward over 10 runs, considering a range for number of commands from 0 to the minimum value achieved by all different runs for the same



(a) Path 1: Cumulative reward over sent commands



(b) Path 2: Cumulative reward over sent commands



(c) Path 3: Cumulative reward over sent commands

Figure 7.18: All algorithms: cumulative reward averaged on multiple runs over the number of sent commands, hence actions performed. SARSA: $\epsilon = 0.2$, $\alpha = 0.1$, $\gamma = 0.75$; Q-learning: $\epsilon = 0.2$, $\alpha = 0.1$, $\gamma = 0.55$; SARSA(λ): $\epsilon = 0.2$, $\alpha = 0.1$, $\gamma = 0.75$, $\lambda = 0.5$; Q(λ): $\epsilon = 0.2$, $\alpha = 0.1$, $\gamma = 0.55$, $\lambda = 0.9$. **(a)**: $RewardOnActions_{avg}$ for Path 1; **(b)**: $RewardOnActions_{avg}$ for Path 2; **(c)**: $RewardOnActions_{avg}$ for Path 3.

algorithm. Hence, the figure shows these measures for all paths and all algorithms.

Looking at Figure 7.18a, for all algorithms the cumulative reward starts to increase after about 125 commands, getting positive after 200 episodes for all algorithms. SARSA(λ) is the first curve to increase, while SARSA is the latest one.

Considering Figure 7.18b instead, the curves are not so close anymore. Q(λ) reaches the highest cumulative reward for the entire learning process, followed by

SARSA. Q-learning and SARSA(λ) achieve a lower cumulative reward with similar curves, overcoming SARSA after about 750 sent commands. For this path, the number of commands needed for reaching high rewards is higher. The curves stay below the 0 value for all algorithms until 250 commands sent. Moreover, SARSA(λ) and Q-learning curves get a positive cumulative reward after about 500 commands.

Finally, in Figure 7.18c, we note that SARSA(λ) is the best algorithm, followed by Q-learning. They both reach a positive cumulative reward before reaching 500 commands. SARSA and Q(λ) instead need more than 500 commands to address a positive cumulative reward. Globally, Path 3 needs more commands for the entire learning process, since curves reach more than 1500 actions: as said previously, this is due to the fact that the optimal path in Path 3 is 7 steps long and that Path 3 has a more complex state-machine, with multiple boxes and possible paths.

With all these measures, we note that for optimal parameters, algorithms do not have great differences on their performance, and do not have a unique winner for all state-machines. As shown in Figure 7.14, average rewards have all different values, except for Q-learning in Path 2 which outperforms the other algorithms. Also, Q-learning is also the one that, according to CDF curves in Figure 7.15, have lower and distant curves both for Path 2 and Path 3. By the CDF point of view, this means that Q-learning on averages reaches a higher reward value for each time step. For these reasons, we state that for our case Q-learning is the most appropriate algorithm and we will perform a robustness analysis on it.

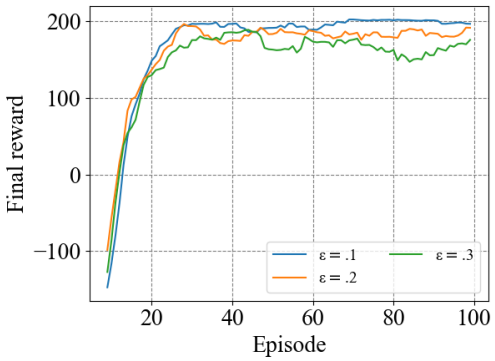
Takeaway. Using optimal parameters, algorithms do not have great differences on performance, and do not have a unique winner for all paths to learn. Q-learning is the only one which outperforms the other algorithms in terms of reward, but only for Path 2. For the same path, evaluating other metrics we have that SARSA is the fastest algorithm and Q(λ) gains a positive cumulative reward in the least amount of generated commands. For other paths, results are slightly different, hence we note that the performance of each algorithm strictly depends on the complexity of the path to learn. For this reason, we can select one algorithm depending on our needs: if we need to achieve higher rewards, generate less traffic or take less time.

7.5 Robustness analysis for Q-learning

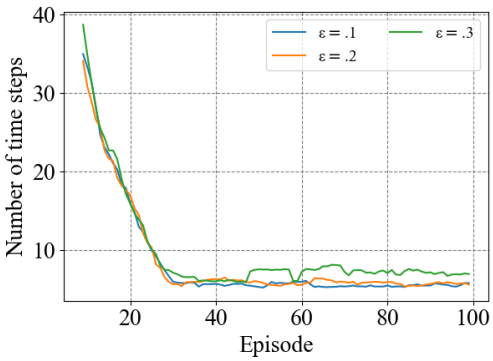
Following the considerations previously made, we want to provide a robustness analysis on the Q-learning algorithm. Our goal is to check whether the optimized parameter values are suitable also for the state-machine for Path 3, in Figure 6.3. We are going to test Q-learning for Path 3 for different values of each parameter. These values will be close to the optimal parameters found during the parameter tuning phase. We will compare the reward and the time steps graphs between different configuration parameters, looking at the oscillations obtained with different

values, if they exist.

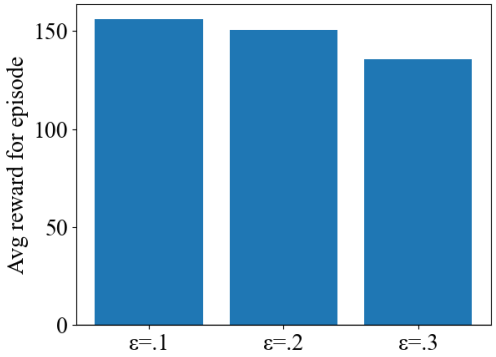
Optimal values for Q-learning were the following ones: $\epsilon = 0.2$, $\alpha = 0.1$ and $\gamma = 0.55$. Fixing each optimal value, we change one parameter at a time. ϵ will be changed to 0.1 and 0.3 values. For α we will use values 0.05 and 0.2. Then, for γ we will try Q-learning with $\gamma = 0.35$ and $\gamma = 0.75$.



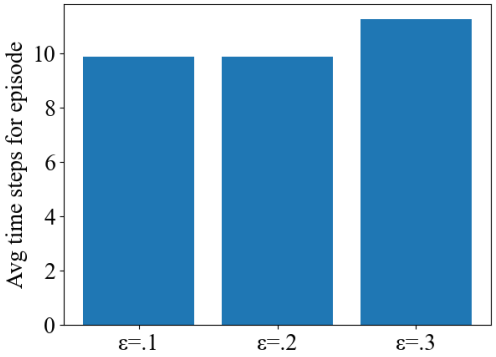
(a) Final reward over episodes



(b) Time steps over episodes



(c) Average reward for ϵ values

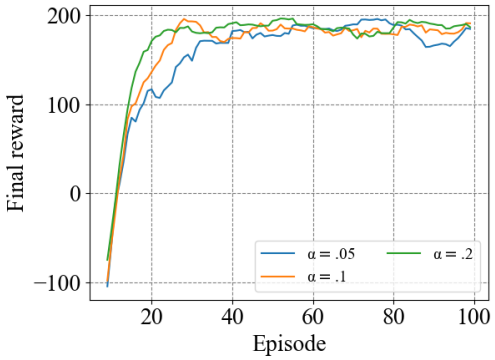


(d) Average time steps for ϵ values

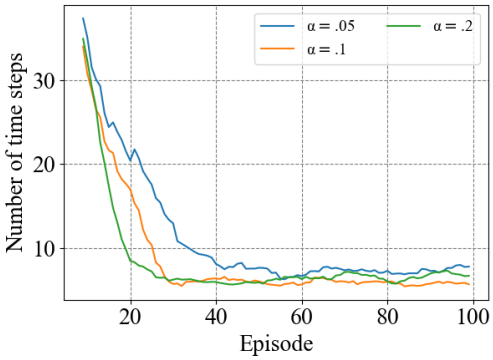
Figure 7.19: Q-learning performance: tuning of ϵ with state-machine for Path 3 with $\alpha = 0.1$, $\gamma = 0.55$. **(a)**: $FinalReward_{mov-avg}$; **(b)**: $TimeSteps_{mov-avg}$; **(c)**: AVG_{rew}; **(d)**: AVG_{steps}.

In all graphs, the yellow line indicates the execution of Q-learning with the optimal values found previously. In Figure 7.19 the performance of Q-learning for Path 3 is shown, varying ϵ. Looking at Figure 7.19b, the curves are very similar. Concerning Figure 7.19a, the curves have small variability. E.g., $\epsilon = 0.1$ leads to higher moved average rewards, but this is due to the fact that exploration is performed only 10% of the time. Indeed, from the average reward value in Figure

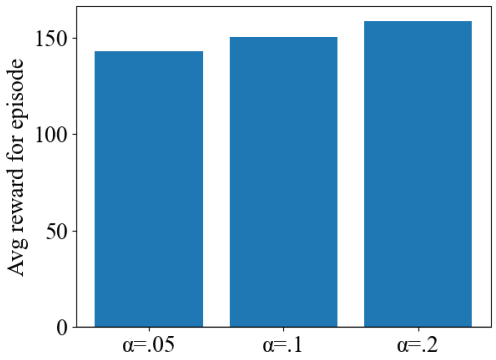
7.19c we can see that the average rewards obtained by $\epsilon = 0.1$ and $\epsilon = 0.2$ are similar.



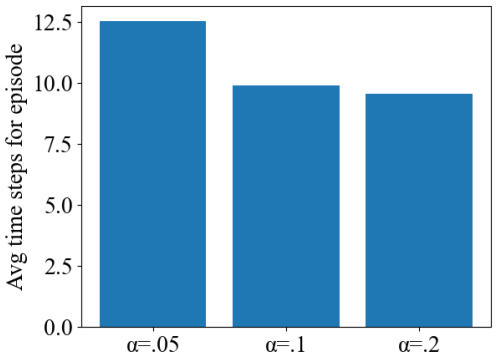
(a) Final reward over episodes



(b) Time steps over episodes



(c) Average reward for α values

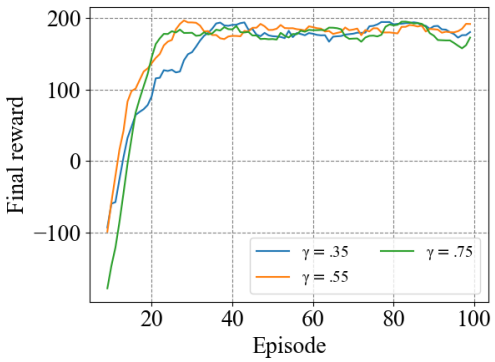


(d) Average time steps for α values

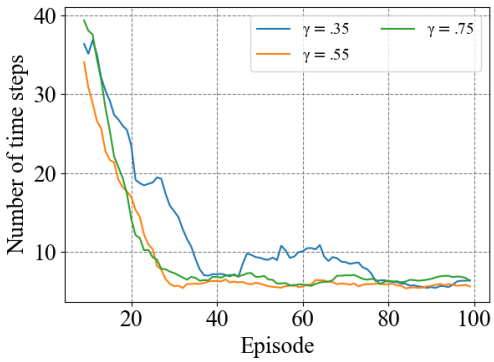
Figure 7.20: Q-learning performance: tuning of α with state-machine for Path 3 with $\epsilon = 0.2$, $\gamma = 0.55$. **(a)**: $FinalReward_{mov-avg}$; **(b)**: $TimeSteps_{mov-avg}$; **(c)**: AVG_{rew}; **(d)**: AVG_{steps}.

Then, in Figure 7.20 we modify the value of α and show the results. The curves are similar to each other. $\alpha = 0.2$ seems to be the one with the highest reward, followed by the $\alpha = 0.1$. However these two curves are not clearly separated, since they overlap between each other. However, the average reward for all three α values is around 150.

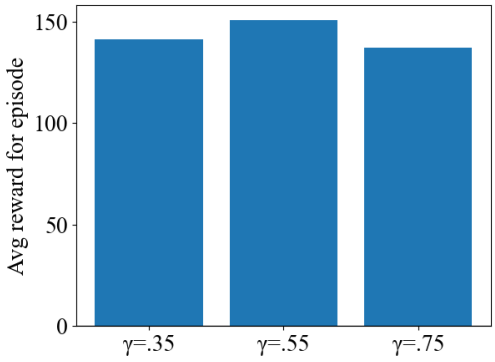
Finally, in Figure 7.21 performance of Q-learning for Path 3 are presented, varying the value of γ. These curves are all near each other, mostly by the reward point of view. Indeed, all average rewards have similar values, confirming this time $\gamma = 0.55$ as the optimal one for this path.



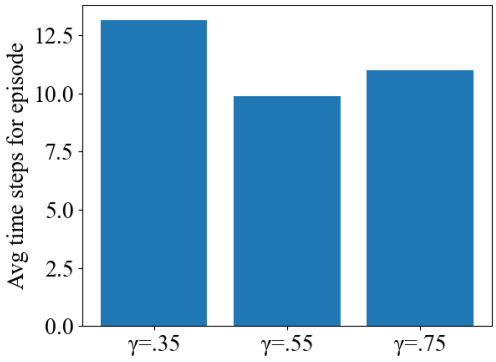
(a) Final reward over episodes



(b) Time steps over episodes



(c) Average reward for γ values



(d) Average time steps for γ values

Figure 7.21: Q-learning performance: tuning of γ with state-machine for Path 3 with $\epsilon = 0.2$, $\alpha = 0.1$. **(a)**: $FinalReward_{mov-avg}$; **(b)**: $TimeSteps_{mov-avg}$; **(c)**: AVG_{rew}; **(d)**: AVG_{steps}.

Takeaway. Considering now all the previous showed results for the robustness analysis of Q-learning, we can conclude that the results obtained for similar values with respect to the optimal values are very similar, with slightly differences for ϵ and α. However, all curves are similar and we can say that Q-learning with tuned parameters can deal with a more complex state-machine - the one defined for Path 3 - with the optimal values found for Path 2, or with values slightly modified. Confirming what stated before, ϵ and α are the parameters which affect the most the final rewards obtained by RL algorithms.

Chapter 8

Conclusions and future work

In this thesis we wanted to find a general approach to interact with IoT devices, by learning their protocols. The approach we chose was to apply RL techniques to achieve this goal. We designed a RL framework which uses four RL algorithms - SARSA, Q-learning, SARSA(λ) and Q(λ) - to interact with IoT devices. As initial knowledge, the RL algorithms have access to a dictionary, which contains commands available for a certain IoT protocol. The framework has been designed in a modular way, with multiple modules. The Device Communication module was used for directly interacting with devices, the Dictionary module to store protocol messages, the Request Builder to access the dictionary and the Discoverer to find IoT devices inside a LAN. Moreover, the RL environment has been implemented inside the State Machine module, whereas the most important component implementing the RL algorithms is the Learning module, which acts as a RL agent.

To start the implementation of this framework, we focused on the Yeelight protocol. The RL environment was modelled defining the states using available properties for Yeelight devices. Among these properties, we selected 7 of them to use as attributes, in order to define the state of a Yeelight device. For this environment, three different state-machines were built, considering different attributes, with different complexities and containing optimal paths of different length. The reward signal is defined for each different path inside the state-machine. The RL agent interacts with this environment performing actions. Actions are possible commands that can be sent to the Yeelight device, that are retrieved by the RL agent from the Dictionary of the Yeelight protocol.

With the Yeelight component of our framework, we execute multiple tests targeting one single Yeelight colored bulb. This allowed us to retrieve statistics

about the learning process performed with different algorithms and different state-machines. After some first results, in which we targeted the goal defined by the state-machine for Path 2, we decided to tune parameters for all algorithms, fixing the state-machine defined for Path 2. We did not perform a grid search, but a greedy tuning, in which we tuned each parameter at a time, while keeping the best values for the already tuned parameters.

We found out optimal parameters for all algorithms. For SARSA and SARSA(λ) we had $\epsilon = 0.2$, $\alpha = 0.1$ and $\gamma = 0.75$. For Q-learning and Q(λ) we had $\epsilon = 0.2$, $\alpha = 0.1$ and $\gamma = 0.55$. For SARSA(λ) we selected $\lambda = 0.5$ as the optimal value, while we selected $\lambda = 0.9$ for Q(λ) algorithm. With these values we test all algorithms against all paths, executing 10 runs for each configuration of 200 episodes each. We obtained an improved performance, gaining rewards close to the maximum reward, on average. We also evaluated the time necessary for performing the entire process with different algorithms and different paths, which takes on average from 50 minutes to around 90 minutes for some configurations, as the ones for the most complex state-machine for Path 3. Also, we computed and showed results about the trend of the cumulative reward, obtained by the RL agent episode after episode, looking at the number of commands generated by the RL agent and sent to the Yeelight device. For a simple path as the first one, considering only 3 attributes for defining the state and defining a 4 steps optimal path, the trend of the cumulative reward gets positive with less than 250 commands sent by the RL agent, for all algorithms. More complex paths need more commands for obtaining a positive value for the cumulative reward. For example, the longest path we defined - Path 3 - needs more than 500 commands to obtain a positive trend, for SARSA, Q(λ) and Q-learning. Finally, we performed a robustness analysis for the optimal values chosen for Q-learning. We chose Q-learning because it was the best performing in terms of obtained reward when applied to the state-machine for Path 2, which was the one used to tune the parameters. For this analysis we slightly variate the optimal values while executing Q-learning for Path 3. This analysis showed that the results obtained for similar values with respect to the optimal values were very similar, with slightly differences for ϵ and α. This suggests a good level of robustness for the values chosen for the Q-learning algorithm.

With these results, our framework demonstrated that RL can actually be used to learn to interact with IoT devices, given as initial knowledge the possible commands defined for an IoT protocol. However, we do not have a unique winner among the tested algorithms: for Path 2, Q-learning performs the best in reward terms, SARSA is the fastest algorithm and Q(λ) gains a positive cumulative reward in the least amount of generated commands. For other paths, we obtained slightly different results, noting that the performance of each algorithm strictly depends on the complexity of the path to learn. Therefore, we can decide which algorithm to use depending on our needs: if we need to achieve higher rewards, generate less

traffic or take less time.

Starting from the work presented here, we propose some future work. This can be done in many different directions, hence here we suggest some of these.

Since we proved that our RL algorithms can actually learn to communicate through the Yeelight protocol, the next step would be to implement more components for other IoT protocols. We designed our framework to be extendable, therefore new components should be added following the implementation procedure done for the Yeelight protocol. Implementing new protocol would arise new problems, and some challenges should be overcome: e.g., decide how to add the support for protocols with a complete different structure for commands or for the communication with the devices. Moreover, for new protocols also the definition of the environment, states and the state-machines would be necessary.

Also, we propose to focus on a more general problem, that is how to model the reward in a better way, since our reward signals were defined arbitrarily. This is a general open question in RL, in which the performance of algorithms are strictly correlated to how the reward function is modelled. To solve this problem, a technique called Inverse Reinforcement Learning (IRL) has been proposed in the RL field. This technique allows for modifying the reward function while performing the learning process. We suggest the usage of IRL to study how an improved reward function should be defined.

Regarding our results, to test the performance of our framework, it would also interesting to evaluate how the framework is impacted when many IoT devices of the same protocol are present inside the LAN. Since here we focused on evaluating the correct working of our approach, we did not evaluate its performance in cases with multiple devices.

Besides this, the results obtained for the algorithms we implemented are quite exhaustive. It would be interesting to implement the support for different kinds of RL algorithms, as for example actor-critic methods. New methods could be compared to these standard RL algorithms.

Finally, we recall that in order to work with standard RL algorithms we approximated our state and action spaces. Since these spaces could be considered continuous, we defined our environment in a more simpler way, considering sets of states inside our state-machines and, from the RL agent point of view, considering commands only by their methods, not also by their possible parameter values. To remove these simplifications, techniques to deal with continuous state and action spaces could be implemented. An idea could be to use artificial neural networks to approximate the value functions defined and used in standard RL algorithms. An example of this technique is called Deep-Q-learning, which combines neural networks with the standard Q-learning algorithm.

Bibliography

[1] *The next wave - the internet of things: it's a wonderfully integrated life*. NSA's review of emerging technologies 2. NSA, May 2016 (cit. on pp. ii, 2).

[2] J. Fruhlinger. *The Mirai botnet explained: How teen scammers and CCTV cameras almost brought down the internet*. Mar. 2018. URL: https://www.csoonline.com/article/3258748/the-mirai-botnet-explained-how-teen-scammers-and-cctv-cameras-almost-brought-down-the-internet.html (cit. on p. iii).

[3] D. Evans. *The Internet of Things - How the Next Evolution of the Internet Is Changing Everything*. White Paper. Cisco, Apr. 2011 (cit. on p. 1).

[4] Statista Research Department. *Number of IoT devices 2015-2025*. Survey. Statista, Nov. 2016 (cit. on p. 1).

[5] *Threat Intelligence Report 2020*. Tech. rep. Nokia, 2020 (cit. on pp. 1, 2).

[6] A. Acien, A. Nieto, G. Fernandez, and J. Lopez. «A comprehensive methodology for deploying IoT honeypots». In: *15th International Conference on Trust, Privacy and Security in Digital Business (TrustBus 2018)*. Vol. LNCS 11033. Regensburg, Germany, Sept. 2018, pp. 229–243 (cit. on pp. 2, 9).

[7] J. Wang and W. Tepfenhart. *Formal Methods in Computer Science*. Chapman and Hall/CRC, 2019 (cit. on p. 5).

[8] M. Nawrockia, M. Wahlisch, T. C. Schmidt, C. Keil, and J. Schonfelder. *A Survey on Honeypot Software and Data Analysis*. Aug. 2016. arXiv: 1608.06249 (cit. on p. 6).

[9] T. Luo, Z. Xu, X. Jin, Y. Jia, and X. Ouyang. «IoTCandyJar : Towards an Intelligent-Interaction Honeypot for IoT Devices». In: 2017 (cit. on pp. 7, 8).

[10] T. Gu, A. Abhishek, H. Fu, H. Zhang, et al. «Towards Learning-automation IoT Attack Detection through Reinforcement Learning». In: (June 2020). arXiv: 2006.15826 (cit. on p. 8).

[11] S. Kotstein and C. Decker. «Reinforcement Learning for IoT Interoperability». In: Mar. 2019, pp. 11–18. DOI: 10.1109/ICSA-C.2019.00010 (cit. on p. 8).

[12] A. Pauna and I. Bica. «RASSH - Reinforced Adaptive SSH Honeypot». In: *10th International Conference on Communications (COMM 2014)*. Bucharest, Romania, May 2014, pp. 1–6. DOI: 10.1109/ICComm.2014.6866707 (cit. on p. 8).

[13] A. Pauna, A. C. Iacob, and I. Bica. «QRASSH - A self-adaptive SSH Honeypot driven by Q-Learning». In: *12th International Conference on Communications (COMM 2018)*. Bucharest, Romania, June 2018, pp. 441–446. DOI: 10.1109/ICComm.2018.8484261 (cit. on p. 8).

[14] A. Pauna, I. Bica, F. Pop, et al. «On the rewards of self-adaptive IoT honeypots». In: *Annals of Telecommunications* 75 (Jan. 2019), pp. 501–515. DOI: 10.1007/s12243-018-0695-7 (cit. on p. 9).

[15] L. Huang and Q. Zhu. «Adaptive Honeypot Engagement Through Reinforcement Learning of Semi-Markov Decision Processes». In: *Decision and Game Theory for Security* (June 2019), pp. 196–216. DOI: 10.1007/978-3-030-32430-8_13 (cit. on p. 9).

[16] S. Dowling, M. Schukat, and E. Barrett. «Using Reinforcement Learning to Conceal Honeypot Functionality». In: *Machine Learning and Knowledge Discovery in Databases*. Vol. 11053. 2018, pp. 341–355. DOI: 10.1007/978-3-030-10997-4_21 (cit. on p. 9).

[17] S. Dowling, M. Schukat, and E. Barrett. «New framework for adaptive and agile honeypots». In: *ETRI Journal* (July 2020). DOI: 10.4218/etrij.2019-0155 (cit. on p. 9).

[18] Y. Pa, S. Suzuki, K. Yoshioka, T. Matsumoto, et al. «IoTPOT: A novel honeypot for revealing current IoT threats». In: *Journal of Information Processing* 24 (May 2016), pp. 522–533. DOI: 10.2197/ipsjjip.24.522 (cit. on p. 9).

[19] C. Kudera, G. Merzdovnik, and E. Weippl. «Smart Things Everywhere: AutoHoney(I)IoT - Automated Device Independent Honeypot Generation of IoT and Industrial IoT Devices». In: *ERCIM News* 119 (Oct. 2019) (cit. on p. 9).

[20] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: A Bradford Book, 2018 (cit. on pp. 10–16, 18, 22, 33, 34, 69, 75).

[21] F. Woergoetter and B. Porr. «Reinforcement Learning». In: *Scholarpedia* 3.3 (). DOI: 10.4249/scholarpedia.1448 (cit. on pp. 10–12, 14–16).

[22] W. Maass. «Networks of spiking neurons: The third generation of neural network models». In: *Neural Networks* 10.9 (1997), pp. 1659–1671. DOI: 10.1016/S0893-6080(97)00011-7 (cit. on p. 11).

[23] R. Bellman. «A Markovian Decision Process». In: *Journal of Mathematics and Mechanics* 6.5 (Nov. 1957), pp. 679–684 (cit. on p. 13).

[24] V. Kumar. *Reinforcement learning: Temporal-Difference, SARSA, Q-Learning & Expected SARSA in python*. Mar. 2019. URL: https://towardsdatascience.com/reinforcement-learning-temporal-difference-sarsa-q-learning-expected-sarsa-on-python-9fecfda7467e (cit. on pp. 18, 33, 75).

[25] A. Guerrero, V. Villagra, J. Lopez de Vergara, and J. Berrocal. «Ontology-Based Integration of Management Behaviour and Information Definitions Using SWRL and OWL». In: *Proc. IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM 2005)*. Barcelona, Spain, Oct. 2005, pp. 12–23. DOI: 10.1007/11568285_2 (cit. on p. 35).

[26] I. Roy, S. Sonthalia, T. Mandal, A. Kairi, and M. Chakraborty. «Study on Network Scanning Using Machine Learning-Based Methods». In: *Proc. International Ethical Hacking Conference 2019 (eHaCON 2019)*. Kolkata, India, Nov. 2019, pp. 77–85 (cit. on p. 43).

[27] *Yeelight WiFi Light Inter-Operation Specification*. Yeelight. 2015. URL: https://www.yeelight.com/download/Yeelight_Inter-Operation_Spec.pdf (cit. on pp. 54, 56, 57, 59).

[28] T. Harwood. *IoT Standards and Protocols*. Mar. 2018. URL: https://www.postscapes.com/internet-of-things-protocols/ (cit. on p. 55).

[29] P. Sethi1 and S. R. Sarangi. «Internet of Things: Architectures, Protocols, and Applications». In: *Journal of Electrical and Computer Engineering* 2017 (2017). DOI: 10.1155/2017/9324035 (cit. on p. 55).

[30] H. Van Hasselt. «Reinforcement Learning in Continuous State and Action Spaces». In: (Aug. 2013). DOI: 10.1007/978-3-642-27645-3_7 (cit. on p. 62).

[31] V. Mnihand K. Kavukcuoglu, D. Silver, et al. «Human-level control through deep reinforcement learning». In: *Nature* 518 (2015), pp. 529–533. DOI: 10.1038/nature14236 (cit. on p. 62).